

UNIVERSITY OF SURREY
Faculty of Engineering & Physical Sciences
Department of Computing

PROFESSIONAL PROJECT (COM3001) -
FINAL YEAR PROJECT REPORT

SOURCE CODE PLAGIARISM

Mr ARAS ABDO
AA01558 - 6367013
Supervisor: Dr Lee Gillam

Bachelor Thesis
2018

Declaration of Originality

“I confirm that the submitted work is my own work and that I have clearly identified and fully acknowledged all material that is entitled to be attributed to others (whether published or unpublished) using the referencing system set out in the programme handbook. I agree that the University may submit my work to means of checking this, such as the plagiarism detection service Turnitin® UK. I confirm that I understand that assessed work that has been shown to have been plagiarised will be penalised.”

Abstract

Academic communities across the globe are faced with increasing occurrences of plagiarism. This worldwide problem has motivated the need for a robust, efficient and fast detection system, that would otherwise be difficult to achieve manually. This thesis looks at the construction and implementation of a copy-detection tool that is designed to support academics in schools and universities in the detection of plagiarism for a pair of programs written in the programming language Java. Before the copy-detection tool can be designed, we analyse existing systems and algorithms in place that deal with the issue of plagiarism. Based on these approaches and the agile software development methodology we identify all the requirements necessary for a successful detection tool and the series of steps needed to implement a detection tool that has been well directed and advised.

Acknowledgements

I would like to dedicate this section of the thesis to truly thank my supervisor Dr Lee Gillam for his continuous guidance and support throughout the project and for providing this interesting topic. His advice and guidance played a very significant role in the development of this project, helping me overcome mental barriers in understanding what sections of this report entailed, combined with his analogies of rockets to the moon. Additionally, I thank my mother and siblings for constantly hassling me to complete my work to deadline.

Table of Contents

	TITLE PAGE	1
	Declaration of Originality	2
	Abstract	3
	Acknowledgements	4
	Acronyms	6
	Table of Figures	7
1.	Introduction	8
1.1.	Spotlight	9
1.2.	Goal Objectives Scope	10
1.3.	Structure of the Project	11
2.	Literature Review	12
2.1.	Types of Source Code Plagiarism Attacks Disguises	12
2.2.	Investigating Plagiarism Cases Manually	13
2.3.	Attribute-Counting and Structure-Metric Systems	14
2.4.	Overview of Source-Code Plagiarism Detectors	15
2.4.1.	Features Employed and Comparison of Existing Systems	16
2.4.2.	What Existing Tools Are Unable to do	18
2.5.	Algorithms used in Plagiarism Detection	19
2.5.1.	Desirable Properties	20
2.5.2.	Fingerprint Based Methods	20
2.5.3.	Content Comparison Methods	21
2.5.4.	Overcoming Plagiarism Disguises Through Pre-Processing	23
2.5.5.	Speed and Reliability of Plagiarism Detectors	25
3.	Language Software Methodology	26
3.1.	Java Programming Language	26
3.2.	Eclipse	26
3.2.1.	Eclipse WindowBuilder (SWT Designer & Swing Designer)	27
3.3.	Software Development Life Cycle	28
3.4.	Software Development Methodologies	30
3.4.1.	Waterfall Model	30
3.4.2.	Iterative Model	30
3.4.3.	Agile Model	31
3.4.4.	Chosen Software Development Methodology	31
3.5.	Summary	31
4.	Requirements & Planning	32
4.1.	Overview of Proposed System	32
4.2.	Ethical Issues	33
4.3.	Requirements	34
4.3.1.	Functional Requirements	34
4.3.2.	Non-Functional Requirements	35
4.4.	Planning	35
4.5.	Overall Approach for Designing and Implementing	36
4.6.	Summary	36
5.	Design	37
5.1.	Principles	37
5.2.	Top-Level Architecture	37
5.3.	Application Steps	39

5.4.	Blueprint ~	40
5.4.1.	Graphical User Interface [Frame]	40
5.4.2.	Parser [ReaderWriter]	41
5.4.3.	Normaliser	43
5.4.4.	RKR-GST	47
5.4.4.1.	HashTable	48
5.4.4.2.	TileValuesMatch	48
5.4.5.	SimilarityCalculator	48
5.4.5.1.	SimilarityValue	49
5.4.6.	MultithreadingWorker	49
5.4.6.1.	Job	49
5.4.7.	UML Class Diagram	50
5.5.	Summary	50
6.	Implementation	51
6.1.	ReaderWriter Class – Methods Explanation	51
6.2.	Frame Class – Methods Explanation	53
6.3.	MultithreadingWorker Class – Methods Explanation	55
6.4.	Running Karp-Rabin_Greedy String Tiling Class – Methods Explanation	56
6.4.1.	HashTable Class – Methods Explanation	58
6.5.	SimilarityCalculator Class – Methods Explanation	58
6.6.	Normaliser Class – Methods Explanation Issues	59
6.7.	Summary	60
7.	Testing & Evaluation	61
7.1.	Graphical User Interface	61
7.2.	File Reader Writer	67
7.3.	Normaliser	69
7.4.	Plagiarism Detection Experiments [RKR-GST]	72
7.5.	Summary	74
8.	How to Run/Use Application	75
9.	Conclusion and Future Work	77
10.	Statement of Ethics	78
	Bibliography	79

Acronyms

RKR	Running Karp-Rabin
RKR-GST	Running Karp-Rabin Greedy String Tiling
MOSS	Measure of Software Similarity
YAP	Yet Another Plague

Table of Figures

Figure 2.1:	Source Code Plagiarism Process	16
Figure 2.2:	JPlag Results Page	17
Figure 2.3:	Plagiarism Tools Comparison Table	18
Figure 2.4:	Popular Plagiarism Detection Systems	19
Figure 2.5:	Common Content Comparison Algorithm	21
Figure 2.6:	RKR-GST Joint Coverage	22
Figure 2.7:	Techniques to Overcome Disguises	23
Figure 2.8:	Example Java Code for Transformation	24
Figure 2.9:	Example Java Code After Tokenisation	24
Figure 2.10:	P-Matching Pair of Equal Code Fragments	25
Figure 2.11:	Speed Comparison of Copy-Detection Tools	25
Figure 3.1:	Drag & Drop (WYSIWYG)	27
Figure 3.1:	Software Development Life Cycle	28
Figure 3.2:	Waterfall Model	30
Figure 3.3:	Iterative Model	30
Figure 5.1:	Top-Level Architecture Diagram	37
Figure 5.2:	Use Case Diagram	38
Figure 5.3:	Application Steps Diagram	39
Figure 5.4:	Folder Structure	40
Figure 5.5:	Folder Structure	41
Figure 5.6:	Multithreading Diagram	49
Figure 5.7:	Initial UML Class Diagram	50
Figure 6.1:	Graphical User Interface	54
Figure 6.2:	Directory Chooser	54
Figure 7.1:	Menu to Directory Chooser	61
Figure 7.2:	Load and Set Copy-Detector Details	62
Figure 7.3:	Pair of Programs Visualisation	62
Figure 7.4:	Final UML Class Diagram	63
Figure 7.5:	Directory Chooser Test	64
Figure 7.6:	Directory Not Set	64
Figure 7.7:	Directory Path Set	64
Figure 7.8:	Directory Path Successfully Stored	65
Figure 7.9:	Language Extension JComboBox Drop-Down	65
Figure 7.10:	Run Comparison Validator	66
Figure 7.11:	Empty Directory	67
Figure 7.12:	Ten Files/Sub-Folder	67
Figure 7.13:	HTML File Writer	68
Figure 7.14:	Normaliser Original Code	69
Figure 7.15:	Code Pre-Processed	69
Figure 7.16:	Remove Comments from Code	70
Figure 7.17:	Remove String Constants	71
Figure 7.18:	To Lower Case and Map Synonyms	71
Figure 7.19:	Snippet of Modified Test 1	73
Figure 8.1:	Folder Structure	75
Figure 8.2:	Fill Information	75
Figure 8.3:	Location of Created Files	76

1. Introduction

Modern society is engulfed by the internet; it has become all-pervasive. Through digitalization and the widespread use of computers, the ease of information sharing has become easy and has encouraged online literature searching. With this the potential of intellectual property theft and academic misconduct has risen.

Plagiarism is an unethical behaviour, that involves the reuse of another individual work without explicitly stating or acknowledging the original author.

However, plagiarism hasn't awakened because of the internet, it's not a new issue. People have been plagiarising far before the internet was widely available. It required hard work and expertise, the individual needed the ability to find the work they desired, collect that piece of information, which was often difficult to acquire and finally present it to the audience. Information is everywhere, coming in the forms of written text, source-code, art, ideas and designs. Famous paintings from all over the world are likely to be plagiarised, dealers produce fakes of these and sell it off as the original, in other cases, individuals purchase the painting knowing beforehand that it is a replica, these from of 'information' are less likely to be detected by law enforcers as it required an in person viewing. These types of information can be exploited by individuals that seek an easier way to get 'data'. And in some cases the theft of such data, can be devastating to the victim, while being amazing for the attacker.

- Industry Plagiarism

It's important to note the impact of plagiarism in economics. The loss of a company's source code (through theft, extortion, etc.), to third parties could result in huge economic losses. The obtained code can be capitalised upon through re-selling, or modified to visually appear as an entirely new software. In this case normal detection tools could not be applied to determine an act of plagiarism, because companies source codes are typically confidential. Therefore, a new solution is needed to detect plagiarism if the source code is not available for a suspicious program. For instance, plagiarism events such as that of Google vs Oracle show that even global companies will resort to plagiarising the content of another company's source-code. Although Google was not found guilty by the jury of violating any of Oracle's patents, it was clear that parts of Oracle's source codes were plagiarised by Google.

- Academic Plagiarism

Undergraduate computer science courses all over the world deal with plagiarism within students' programming assignment submissions, and the Computer Science department at the University of Surrey is no exception. Learning programming skills requires a lot of patience, practice and time. For some programmers solving an exercise or assignment is difficult, or they simply do not want to invest sufficient time to complete the task. With deadlines closing in, it becomes tempting to cheat and submit another person's work. Students can also work together "too much", and partially re-use the work of each other when this is forbidden.

- Government Plagiarism

Governments all over the world plagiarise data, this can be done through espionage, direct assault of information centres to extract the data. The area of attack that is widely reported by the news is the theft of military designs, such as fighter planes, tanks etc. The American government has repeatedly accused the Chinese government of hacking into military facilities and computers that store military tech designs, using these design to develop their own war machines, these stolen designs have reportedly exceeded the Chinese military capabilities by several decades.

As concerns over plagiarism grow, more attention is placed towards overcoming this issue, within the past few centuries many automated detection systems have been implemented and are being used within industry and academia.

1.1. Spotlight

This thesis spotlights a particular type of plagiarism: Source Code Plagiarism, often noted as software plagiarism or code theft. Programming languages are implemented by users in a specific manner, in such a way that, if you were to glance at a random program, you should be able to identify what language it implements. This is because there is a specific way everything should be implemented, from the class structure, to methods, all the way down to how the variables should be written. Because of this structured nature, it is easier to identify plagiarism between two programs than between two essays, as essays can be written in completely unique ways, whereas two random programs will share similar traits. We should note that, two programs that have been designed for the same task will have similar traits because of the nature of programming structure, for example, when implementing a program that acts as a clock, these two separate programs will probably look similar, more so if they consist of only a few tens of lines. Thus, the question arises: when and what is considered as plagiarism?

Alan Parker and James O. Hamblen define software plagiarism as [1]:

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

Where “routine transformations” are identified in sections 2.1.

1.2. Goal | Objectives | Scope

The main goal of this thesis is to implement a source plagiarism detection tool. Such a tool would be useful for detecting whether a pair of programs have similarities that exceed a specified threshold. This system will be used by individuals or organisations that require the ability to compare code, such individuals include examiners, academics or teachers.

The system must be able to support the detection of plagiarism that exists in students' Java programming assignments. The program will provide a visualisation features that allows the examiners to identify the authenticity of the pair of submitted programming assignments. The examiners should be informed of suspected plagiarism that has been committed by the students whose work has been compared, via a final similarity result that is at least threshold value or greater.

Objectives

- To understand what plagiarism is and why do people plagiarise, to also discuss whether it is only individuals that plagiarise or do companies and governments also do it? Has it increased because information is more freely available?
- To investigate what types of plagiarism exist. E.g. text, art and source code.
- What is source code plagiarism, to investigate how source code plagiarism occurs and what strategies people use to disguise their plagiarism.
- To investigate existing systems for source code plagiarism and what features they have in place.
- To understand the algorithms currently implemented by the two most popular systems.
- Prepare through requirements for my copy-detection system.
- To design and implement a source code detection tool that can detect plagiarism in a pair of programming codes, based on one of the algorithms described earlier.
- Test and evaluate the implemented system against a detection tool that uses a similar algorithm to the one that I have used.

The scope of the research is limited to external plagiarism, i.e. when both of the programs are available to compare, the suspected code and the original, in this case where two or more students present source code that an academic has deemed "suspicious", and thus further investigation is undergone, through the use of a plagiarism detection tool. The quality of the system, from the coding to the design will correspond directly in respect to my own programming abilities, and will determine the overall complexity of the final software.

1.3. Structure of the Project

Chapter 1: Introduction - This section details the project's aims, the scope, the initial conception, the problem to solve and an initial overview to plagiarism in general.

Chapter 2: Literature Review - This section details the types of plagiarism attacks, the two types of code-detection methods and also covers the current technology used to detect source-code plagiarism and the features they provide. This is followed up by detailing several algorithms employed by these technologies.

Chapter 3: Language | Software | Methodology - This section details the programming language that will be used to implement the proposed program, based on this the appropriate software's are selected and a software development life cycle methodology is picked to structure the project.

Chapter 4: Requirements & Planning - The section details the requirements that need to be met by the final copy-detection system, and the planning for the project is designed.

Chapter 5: Design - This section details the design for each section of the program, identifying the main parts and how they will be implemented.

Chapter 6: Implementation - This section details the most important methods for each Java class in the copy-detection system. We discuss how each method works and what tools and technologies were used to make the methods.

Chapter 7: Testing & Evaluation - This section involves testing and evaluating the final system and determining whether it has been built to success.

Chapter 8: How to Run/Use Application - This section details how the instructions on how to run the copy-detection program, from start to finish.

Chapter 9: Conclusion and Future Work – This is the final section of the report that determines the overall success of the project.

Chapter 10: Statement of Ethics – In this section we look into the ethical aspects of the plagiarism detection, from both the legal and moral sides.

2. Literature Review

The aim of this chapter is to give background knowledge about plagiarism detection. We discuss the types of attacks used by individuals to disguise their plagiarism code, followed by the two types of code-detection methods that can be used for source-code plagiarism. A brief overview of the history of source code plagiarism over the past 30 years is detailed, and the software's identified are compared against each other, we then identify the features that these tools do not cover. Then the principal algorithms and techniques currently used are detailed.

2.1. Types of Source Code Plagiarism Attacks | Disguises

To create a tool that is able to detect plagiarism, we must first understand how someone goes about doing it, because programming languages are structured, it becomes trickier for a user to change the original code to something that looks like it wasn't copied, the user require a clear understanding of the programming language as well as the program itself. Here we identify the several ways that a program can be changed, from the very basic changes that do not affect the actual structure of the code, to features that are more advanced:

1. Word for word copy.
2. Adding comments and whitespace.
3. Renaming variable identifiers.

The plagiariser can be void of any understanding of the code to implement the first three methods, with methods 2-3 being lexical changes, these are very simple attacks that require little time to implement. Copy-detection systems should immediately identify programs that have be copied word for word, as they will match directly. It's also important to note that individuals tend to add unnecessary comments and whitespace throughout the code, in a mission to conceal their plagiarism, this doesn't affect the actual code, but is used as a means to add enough information to make it seem visually different from the original, however overusing comments or adding comments on code that is very basic, can be suspicious. The use of unconventional naming conventions for method 3 can make it difficult to understand what the variable is used for, however if the attack only involves the first 3, then these can be identified visually as the structure of the code remains the same, therefore simply removing the whitespace and comments will allow the user to match the structure, rather than the code.

4. Block reordering
5. Adding unnecessary statements/variables.
6. Transforming data types
7. Control replacement
8. Other Modifications

The last four methods require higher understanding of the program's code, as well as the implementation language. Method 4 and 5 are used to visually disguise the plagiarised code, while methods 6 and 7 transplant existing code to equivalent structures, further distorting the visual presentation of the code, however the underlying output of the code remains the same. These are considered structural changes, some examples include: the changing of iterations, conditional statements, the order of the statements. [2]

All the modifications to source code that have not been assigned to the previous categories, fall under “Other Modifications”. For instance, making use of small optimizations in the form of eliminating dead code, or using inline functions. It’s important to note that these are “small” optimisations, as knowledge is demanded to produce code that works better than the original.

- Translation

A form of plagiarism that was not been identified above, is the translation of source code from one programming language to another, such as translating code that is written in C++ to Python, this type of plagiarism is generally difficult to perform as it requires expertise in both languages, but also important to note is the fact that most languages cannot be completely converted into another, a programming structure in C++ can look different in Python, and as the user continues to translate, it will ultimately look quite different to the original, however patterns will remain, that can suggest it has been copied.

2.2. Investigating Plagiarism Cases Manually

Universities and other organisations have rules, penalties and detection systems in place to deal with plagiarism. However, when working with large groups, it becomes infeasible to detect plagiarism. The number of unique pairs that could contain plagiarism grows with the number of submissions n using this formula:

$$\text{Number of pairs}(n) = \frac{n * (n - 1)}{2}$$

The number of pairs to compare grows very quickly. In Surrey University, n may range from 60 up to 135, which makes the number of cases to consider manually uneconomical. Therefore, the use of computer programs designed to spot the similarities between a pair of code, greatly speed up the review process, with only code that has been detected to have a large similarity value, being sent for review by a human.

2.3. Attribute-Counting and Structure-Metric Systems

There exist two main types of code-based detection methods that are used to find similarity; attribute counting and structure metric systems, there also exists hybrid systems that combine features from both to create an entirely new method.

Early systems used the attribute-counting approach, which measures the key properties of assignment submissions. The earliest noted system that can be identified is Halstead's software metrics, which was the first to be used, over the next few years this system was improved upon and modified for more "advanced detection". The quantities identified by Halstead were used to approximate the similarity between the given programs, if the quantities in both of these programs were similar to each other then it would be considered plagiarism, however it is important to note that these systems only identify the rate of similarity, therefore it should only be deemed plagiarism by the "examiner". Systems like Hallstead's are great when working on basic programs. Therefore, in programs where only small parts have been copied it is hard to identify the plagiarism. This partial plagiarism will need to be picked up by attribute-counting systems, thus going unnoticed. In a report written by Verco and Wise [3] they state that attribute-counting systems are highly limited, and these systems perform best in detection when the plagiarism between two programs are very close. Usually, individuals add several additional pieces of redundant code statements to their programs to help further disguise their plagiarism. Because attribute-counting systems take all lines of code into account, including that of the redundant code, the resulting attribute score will greatly differ. [4]

The modern approach to plagiarism detection is with use of the structure-metric method, the detection tools that are currently being used by academia, industry for source code plagiarism exploit this method rather than attribute counting. These systems first became known in 1981, when they were used in Donaldson, Lancaster and Sposato, their system was a hybrid, because they used features from both attribute and structure systems. Structure-metric systems create representations of programs and compare these to detect plagiarism. These systems do not consider easy to modify elements, such as white space, variable names and comments, and thus they are less susceptible to redundant statements that could mislead the result. Structure-metric systems compare the structure of the programs rather than transforming them into a sequence of numbers (attribute-counting). In this approach source code is compared in two phases. Usually the original programs are initially tokenised, then a string comparison algorithm is used to compare the resulting token strings, to produce a final plagiarism similarity value. [5] [6]. Some of the most important and state of the art detection systems are based on structure-oriented detection, the most cited of these is MOSS and JPlag, which other systems include, Plague, YAP3, Marble, Sim, Plaggie, and FDPS

2.4. Overview of Source-Code Plagiarism Detectors

In this section we discuss the most popular copy-detection systems.

- Measure of Software Similarity (MOSS) [7]:

MOSS is an online web service that can be accessed through scripts, that enable users to upload programs for comparison, the system outputs the results through a web interface and offers a visualisation interface; from which the users are able to see the found plagiarism matches. Developed by Aiken et al at Stanford University in 1994, and is based on the popular fingerprinting algorithm called Winnowing. This system supports the detection of plagiarism in multiple languages, some include; Java, C++, C, JavaScript, Python and many more.

- Yet Another Plague (YAP) [8]:

There are three sequential versions of YAP, beginning with the first Yap which was introduced by Michael Wise in 1992. Later, after several improvements were made to the original Wise released YAP2 and finally YAP3. YAP3 is the latest version and overcomes many of the issues and drawbacks that were encountered with the previous versions, it is based on the efficient and popular Running Karp-Rabin Matching Greedy String Tiling algorithm which is able to perform string comparisons on pairs of programs, once the comparison is complete YAP outputs the result in a text file.

- JPlag [9]:

JPlag is a freely available token based system that is offered as an online service. JPlag originated as a student research project at the University of Karlsruhe in 1996, developed by Guido Malphol. Later in 2005, the tool was transformed into a web service that would be used to detect plagiarism in programs by converting them into a sequence of tokens, that represent the essence of the code. The system is based on the popular Greedy String Tiling algorithm created by Michael Wise, however it uses its own optimisations that produce better efficiency. Similarly, to MOSS, JPlag is also able to detect plagiarism in a wide variety of programming languages, some include; C, C++, Java etc. The results of the comparison are output through both text and as a visual representation of the found matches, these matches are highlighted for better readability.

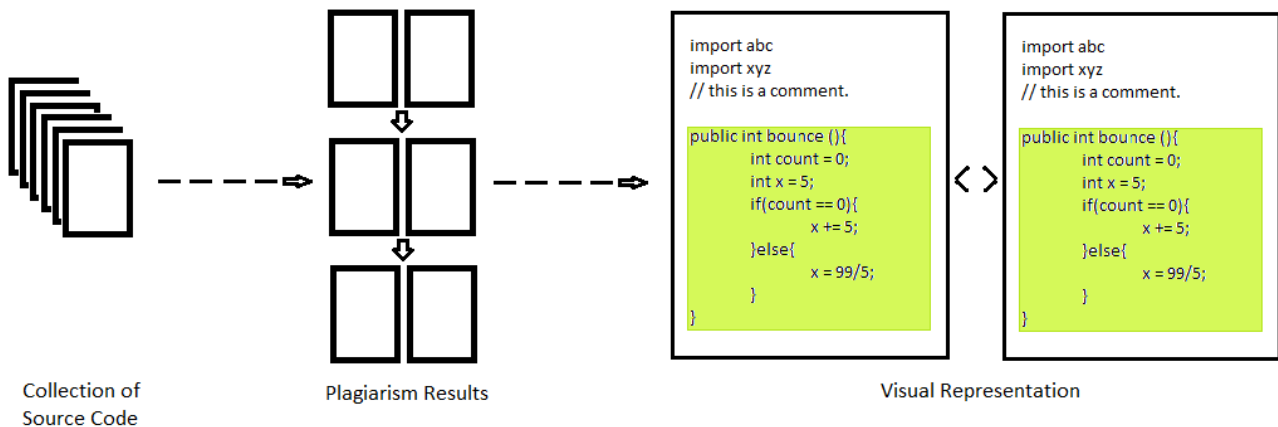
- SIM [10]:

SIM is a similarity tester for software and can be used to detect plagiarism in many programming languages, such as C, Pascal, Java and also natural language. Developed by Dick Grune at the University of VU in Amsterdam in 1989, it is an efficient detection tool that can be accessed through a command line interface. Sim implements its own unique algorithm that is discussed later in this thesis, the results of this comparison are output into a text file.

2.4.1. Features Employed and Comparison of Existing Systems

The detection tools used to detect source code plagiarism work as follows: they take in a collection of source code documents, and sort all unique pairs according to their measured similarities. Tools can also give an explanation of this ordering by visualising the similarities in the two programs. An abstract view of this task can be seen in figure 2.1.

Figure 2.1: Source Code Plagiarism Process.



- Visualisation

Besides differences in detection performance, some tools provide additional features that are helpful for viewers. Figure 2.2 illustrates a feature provided by JPlag in which the results of a comparison between two documents are presented side by side. This gives visual identification of where exactly the highest similarities between two documents are located, an arrow is placed beside each piece of code that has been detected as plagiarised, with the press of a click on one of these screens, the second screen is realigned to present the two-plagiarised code side by side, regardless of their position in the document. The same information that could be used to improve the quality of similarity detection, can also significantly improve the visual representation. This is an important feature that has been adopted by most copy-detection tools as visualisation of the results is the only way to determine whether the code has been plagiarised. It is important note is that tools like JPlag, MOSS or this thesis' implementation only show similarities. The found similarities never imply that plagiarism is found. The decision to declare found matches as plagiarism must be done by the user after analysing the found matches.

Figure 2.2: Snippet of JPlag results display page for a pair of programs



- Template Code Removal

It is very normal computing modules to provide a shared base code for students' programming assignments, from which each student needs to build upon and complete. Also, it is quite likely that something shared in the lecture can be used in the assignment. In both of these cases, the search of plagiarism in a pair of programs that share common code is likely to result in legitimate matches that do not indicate real plagiarism, but can instead be explained by one of the two cases. Some of current plagiarism detector offer the user an extra tool, to which they can insert a base code that will be removed from both of the files being compared. This can help prevent a lot of false positives from occurring.

- Comparison Table

A comparative study is carried out by [11] on software plagiarism detection techniques. The study compares seven different plagiarism detectors using a ten qualitative comparison criteria. The table below summarises the results obtained from this study: Figure 2.3

Feature	GPlag	JPlag	Marble	Moss	Plaggie	SIM
1 - Supported languages	All	6	1	23	1	5
2 - Extendability	Yes	No	No	No	No	Yes
3-Presentation of results (1-5)	5	5	3	4	4	2
4.Usability	5	5	2	4	3	2
5 - Exclusion of template code	Yes	Yes	No	No	Yes	No
6 - Exclusion of small files	Yes	Yes	Yes	Yes	No	No
7 - Historical comparisons	No	No	Yes	No	No	Yes
8 - Submission or file based	Submission	Submission	File	Submission	Submission	File
9 - Local or web-based	Web/Local	Web	Local	Web	Local	Local
10-Open source	Yes	No	No	No	Yes	Yes

Figure 2.3: Plagiarism Tools Comparison Table [11]

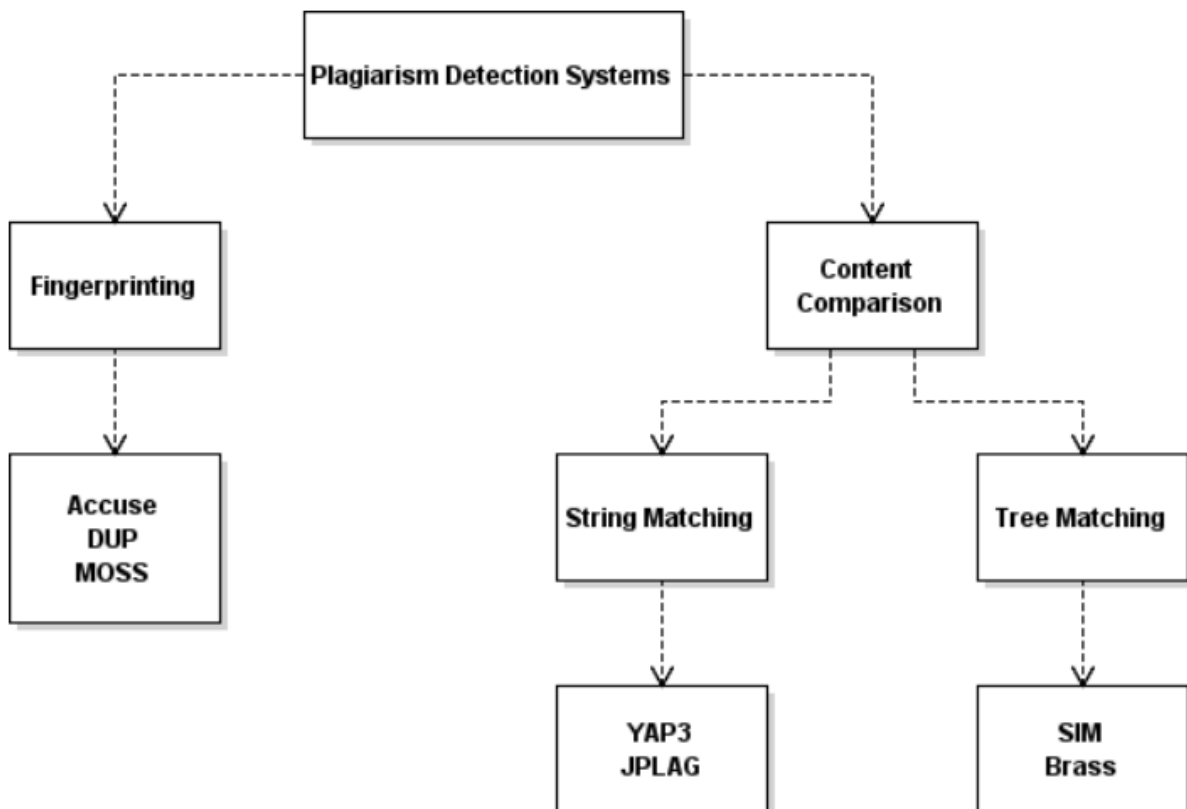
2.4.2. What Existing Tools Are Unable to do

Before we can determine what source-code detectors cannot do, we should understand what similar detection systems do, such as text detection tools like Turnitin. Turnitin is a tool that finds text which match other sources that are stored in the Turnitin database and shows presents these matches visually. This tool has a database that contains enormous amounts of web content, previously submitted papers and also publications and subscription based journals. When a paper is uploaded to Turnitin, the tool compares the paper against everything in their database and if it matches something, this is highlighted and the source of the match is included. But similarly to source-code detection systems it is up to a human to give the final verdict on the paper. Copy detection tools do not store sources in a database or repository, which they can compare the suspected program against. They are however online repository that store vast amount of programming code, one such web site is GITHUB. A future copy-detection system could utilise GITHUB as their databases of sources to compare their code against, in my case this far exceeds my scope and I will not be implementing it.

2.5. Algorithms used in Plagiarism Detection

This section presents the algorithms that are being used by detection systems that are currently employed, popular tools such as, MOSS and JPlag. The two code-detection methods used have been employed by these tools and have been identified and finally we provide a comparison of the popular systems. Before we identify the algorithms, it is important to note the desirable properties that these algorithms should have. The algorithms that I am primarily interested in evaluating are the current approaches implemented by existing software as can be seen in the Figure 2.4.

Figure 2.4: Popular Plagiarism Detection Systems



2.5.1. Desirable Properties

The methods stated in the “Types of Plagiarism Attacks” sections, 1 to 8 should be implemented by plagiarism detectors to insure that they are robust against such attacks. Not every copy-detection algorithm is based on the same technique, they use different algorithms, methods and ways of identifying code as plagiarised. Therefore, three universal properties that every detection tool should implement are defined : [12]

1. Insensitivity:
Whitespace, capitalisation, punctuation, comments, and the renaming of variable identifiers should not have an effect on the matches between a pair of programs as these are the easiest things that a plagiariser can do to disguise their code, thus such elements should not be considered, moreover they should be discarded from the code, as to insure they do not affect the matches.
2. Noise suppression:
Any matches that have been discovered must be large enough to imply that the material has been copied. Thus the detection of short matches, such as a try-catch block within a pair of programs, is uninteresting and will not deem the code to be plagiarised.
3. Position independence:
Adding parts of code to the program, shuffling the order of the blocks and removing parts of the program, should not affect the set of discovered matches.

2.5.2. Fingerprint Based Methods

The primary goal of fingerprinting is to take a collection of documents and create a fingerprint for each of them, where each fingerprint characterises a longer file by using a small sequence of bytes. For example, one way to obtain a fingerprint is by applying a hash function to a file. When it comes to plagiarism detection, fingerprints are often much more complex than simply containing hash code: the structure of the document is contained within several numerical attributes inside the fingerprints. Where these attributes usually include the average number of words in each line or the total number of unique lines or passages. A pair of documents are deemed similar if the two fingerprints are relatively close to one another, which is usually dependent on some specific criteria; such as a distance method. [2]

Formerly, attributing counting systems were used fingerprints, these were the very first systems that implemented computer plagiarism detection. In 1976 Ottenstein developed the first source-code plagiarism detection system, which used Halstead’s software metrics to detect plagiarism in FORTRAN programs. The program takes the several matrix and produces five attributes from them. It was deemed to be plagiarised if the two independently written programs had identical values for each of the metrics, once the system detected possible plagiarism, it is up to a human inspector to examine the results and then based on their findings, determine whether the code was actually plagiarised. Ottenstein asserts that there is an extremely thin probability that two systems which have been written independently will have the same attributes. [13]

In the following year more advanced systems were created, but these systems are inherently inferior to systems that employ content comparison techniques, because fingerprints are greatly affected by even subtle modifications to the programming code. Thus, modern systems do not usually do not implement this method of comparison. However, it is important to note that there are several modern systems that combine elements from both string matching and fingerprinting to produce an entirely new system, one such example is MOSS; based on the winnowing algorithm that improves the efficiency for comparisons based on document fingerprinting. [2] [4]

2.5.3. Content Comparison Methods

Content comparison methods are deserving of a throughout in-depth review, because these techniques are now the cornerstones of the plagiarism detection systems that are currently employed. There are several different algorithms that deal with file to file comparisons, these vary in speed, expected reliability, and also the memory requirements. Generally, these schemes follow the definition of similarity defined by Manber; “we say that two files are similar if they contain a significant number of common substrings that are not too short” [14].

- **String Matching Based**

Usually, content comparisons systems work by following the algorithm in Figure 2.5

```
1 FOR EACH collection file F
2   FOR EACH collection file G, F != G
3     Calculate Similarity Between F and G
```

Figure 2.5: Common Content Comparison Algorithm

The core function that is used by many systems to calculate the similarity between files varies. String matching methods compare a pair of files by dealing with them like ordinary Strings. It also reads a pair of source-code as raw data, rather than acknowledging the hierarchical structure of the programs. The similarity result is calculated differently between tools, we will examine these during the design phase, in Chapter 5.

The early detection systems like YAP [15] were based on the string match approach; which used a quite simple mechanism; in this case the pair of programs are compared line by line using a distance algorithm. This approach has been continuously developed into more advanced string matching; such that newer versions of YAP has been released, from YAP2 to the latest version: YAP3 [8], JPlag [16] is another system that utilises this approach. Both of these systems use the same algorithm Running-Karp-Rabin Matching Greedy String Tiling (RKR-GST).

The goal of RKR-GST is to discover the maximal-tiling, i.e. the strings that are contained in both programs which cover as many tokens as possible, while insuring they are not overlapping (Figure 2.6). Moreover, it is forbidden to use short tiles that have a length smaller than a specified threshold. Lastly, it is important to note that RKR-GST utilises the assumption that shorter matches are not as important or valuable as longer ones. thus the algorithm works by searching for longer matches and then works its way down.

The RKR-GST works as follows; The process first analyses the matches that have the length of the initial-search-length or greater. The Karp-Rabin procedure is called to obtain all of these matches. Then beginning with the longest match, each match is analysed. Next a new tile is created if and only if the current match is not overlapping any of the other existing tiles. Once every match has been process, a new smaller search length is used to restart the search. Finally, when the length of the match is equal to the minimum-match-length threshold value, the algorithm finishes its work. [4]

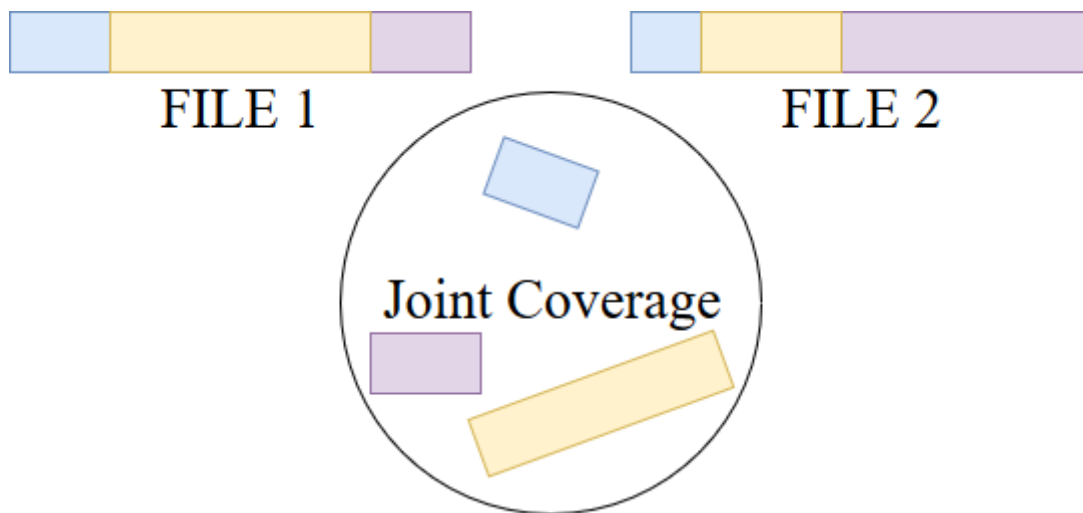


Figure 2.6: RKR-GST Joint Coverage

- **Parse Trees Comparison**

Obviously, if we want to truly describe a program, we wouldn't use the fingerprinting approach, but instead we would read and understand the contents of the file. However, as mentioned early, content comparison methods deal with files as raw data, which do not hold the structure of the file, thus to overcome this a new approach is required. Both natural text and source code files have structure; in natural text the structure is often similar to this; beginning with the tile, the different chapters, the sub-sections contained in each chapter, all the way down to the paragraphs. Similarly, to natural text, programming code also follows a common form, this is usually determined by the programming language; in most cases the file contain classes, functions, control structures and code statements.

There has not yet been a discovery into how best this information can be utilised effectively. However, there are several systems that currently exist that indirectly or directly perform a comparison on the files structures.

The first implementation of the parse tree comparison method was made by the Sim [10]. This system uses generic string matching, and calculates the similarity for a pair of program files, by using their corresponding parse trees, instead of comparing the source-code files directly. Thus, the front-end for the string matching algorithm is a parser. Tree comparisons are complex and therefore are slower than string matching, and because of this a new procedure is defined; the pure tree comparison which is implemented by Brass [17]. Brass only deals with documents that it has deemed as suspicious, this is done through their unique string comparison routine. Once the suspected documents have been identified, a small comparison algorithm is used to produce a result that is more reliable.

Parse tree comparisons have been identified as more advanced and significantly better, but there is little research carried out in this area and thus it is not a viable option for my project.

2.5.4. Overcoming Plagiarism Disguises Through Pre-Processing

Most modern plagiarism detection systems transform the inputted files before the comparison is made, this is done to fight against the plagiarism disguises mentioned in section 2.1. Usually this process includes either tokenisation or parsing. Steps 1,2,3,6 and 7 in section 2.1 are overcome by pre-processing and tokenisation, while the remaining disguise are dealt with using advanced algorithms like RKR-GST which is able to detect reordered matches.

In this section we discuss the different methods and algorithms that can be applied to overcome plagiarism disguises as seen in Figure 2.7.

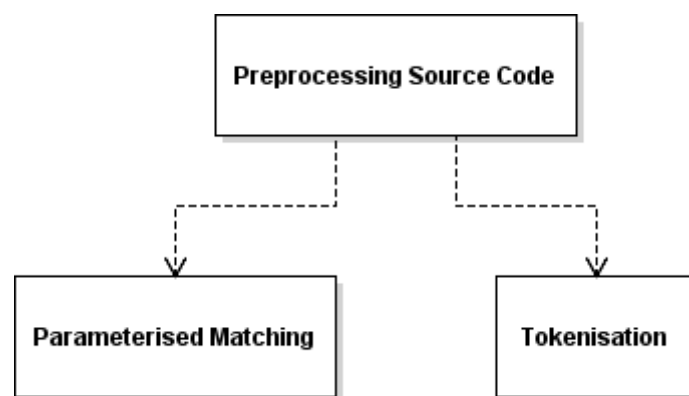


Figure 2.7: Techniques to Overcome Disguises

Most of the techniques made to source-code to disguise the plagiarism are lexical in nature, this means that we can eliminate these lexical changes through pre-processing without harming the semantic information.

- Tokenisation

The simplest method that can be applied to deal with this issue is a tokenizer. This method takes in source-code as input and deals with it as normal text, and produces a token sequence that represents the essence of the entire program, the outputted token sequence is dependent on the programming language of the file. [4] The extracted information includes function calls, control structures, statements and variables. Therefore, the tokenizer program discards all of the undesired properties making it robust against Insensitivity; defined in section 2.5.1.

Simply; a tokenizer program translates a program into a corresponding language. In the example below every line is transformed in to their appropriate structures; variable identifiers are replaced with <IDENTIFIER>, while all numeric values are translated to the token <VALUE>. This means that any changes that disguise the variable names are useless and will not impact the similarity result. It's important to note that different tokenizers translate code to different values.

For example, the follow java code will be taken in as input and transformed (Figure 2.8):

```
1 package finalyearproject;
2 import wrabble.*;
3 public class Report implements IReport{
4     RandomClass randomClass;
5     private Tile tiles [] [];
```

Figure 2.8: Example Java Code for Transformation

For better readability white spaces and line breaks will be kept, the above fragment of code can be pre-processed into the following sequence (Figure 2.9):

```
1 <PACKAGE><IDENTIFIER>
2 <IMPORT><IDENTIFIER>
3 <MODIFIER><CLASS><IDENTIFIER><MODIFIER><IDENTIFIER> {
4     <IDENTIFIER><IDENTIFIER>
5     <MODIFIER><IDENTIFIER><IDENTIFIER> [] []
6     <MODIFIER><IDENTIFIER><IDENTIFIER> [] []
```

Figure 2.9: Example Java Code After Tokenisation

The time complexity of the tokenisation phase needs to be small, usually this is $O(n)$, where the total length of the file is “n”. In section 2.6 we discuss the time complexity for plagiarism detectors, and this will state why it is important that the tokenizer phase does not consume a large amount of the overall complexity.

- Parameterised Matching

The drawbacks of using the tokenisation is it method of extracting information from the source code. Producing an output, from which all the values, loops and variables are replaced by their corresponding token, we lose a huge amount of information that describes the differences between the two programs being compared. Therefore, it is very likely that a lot of false positives will be detected. Although this method is good at identifying plagiarism in a pair of codes, it's important that we preserve the differences between the code to produce a more accurate similarity result.

This preservation can be implemented by a parameterised matching algorithm, often noted as p-match. This is a unique type of string comparison process, the algorithm deals with two inputted program codes as equal, only if one of these files can be retrieved from the other. This is accomplished through a series of regular substitutions of given identifiers, thus the system should be able to determine if the tokens are identifiers or not. An example is given below that illustrates two code statements as equal:

<pre> 1 String text1 = "this example text" 2 String text2 = "hello world" 3 int x = 45 4 x += 6 </pre>	<pre> String exampleString = "this example text" String helloWorldString = "hello world" int abc = 45 abc += 6 </pre>
--	---

Figure 2.10: P-Matching Pair of Equal Code Fragments

2.5.4. Speed and Reliability of Plagiarism Detectors

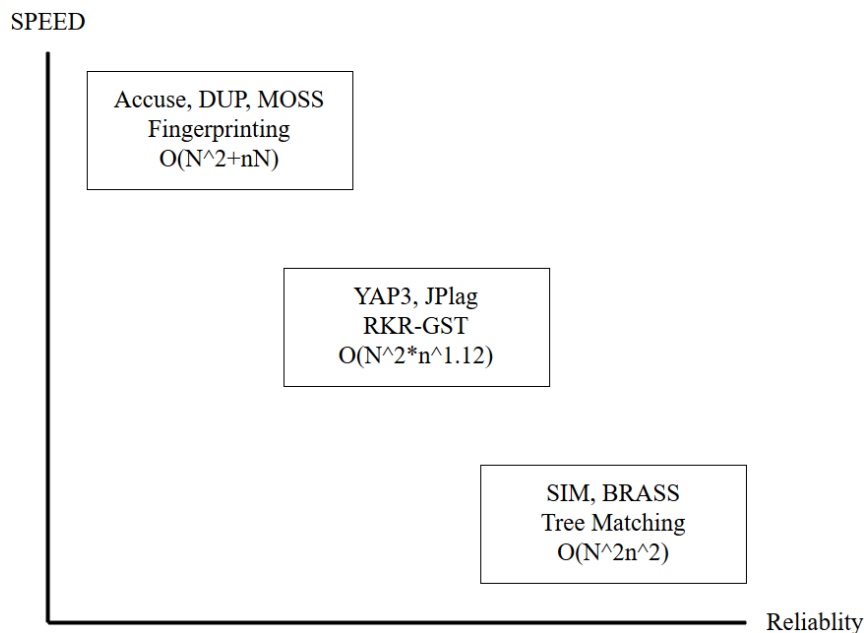


Figure 2.11: Speed Comparison of Copy-Detection Tools

The Figure above has been constructed using the time complexities defined in the following papers [2] [15] [10].

3. Language | Software | Methodology

This chapter gives an overview of the main software and technologies that will be used for the implementation of the proposed copy-detection tool that will be defined in the development section, the software and technologies identified will be based upon the programming language that will be used for the project, the reasoning behind using these are described. Following this a software development lifecycle methodology will be determined.

3.1. Java Programming Language

In the mid-1990s, java was created by a team run by James A. Gosling, owned by Oracle. Java is able to produce software for multiple platforms, and runs on most operating systems, these include Linux, Windows and Mac OS. Java was influenced greatly and derives a lot of its syntax from the programming languages C and C++. Like many programming language, Java has its own structure, syntax rules and programming paradigm.

Java's programming paradigm is based on the concept of Object Oriented Programming (OOP), this is a model that revolves around the notion of object rather than actions, and data rather than logic. The basic unit of object oriented programming is a class, in java a class is an object, that describes both static attributes and dynamic behaviour that is common to every object that is of the same type. For example, creating a Dog class in java, would require that all the features that encompass all dogs be implemented, such include, skin colour, breed, name, etc. This is beneficial for my copy detection tool as the various types within my system can be encompassed by separate classes, such as the normalise, RKR-GST algorithm, GUI frame.

Code written in Java is robust, the objects defined in java contain references only to themselves or other objects that are known by the language, unlike other programming languages like C++. This avoids crashes, that would have otherwise occurred as instructions would contain address of operating system itself or other applications.

Java is the programming language that I am most proficient, familiar and comfortable with using, the majority of my university coding programs in some way require the use of java, thus this is yet another reason I believe the use of this programming language over others, like python or C++ will be beneficial to this project. [18]

3.2. Eclipse

Eclipse is a generic integrated development environment (IDE), used in the development of applications using programming languages, such as Python, C/C++ etc. However, it is mainly used in the development of java applications. It provides a base workspace for programmers, that can be extended by the multiple available plug-ins offered by the platform. Eclipse is open source community focused on providing projects that extend the development platform and application frameworks, this insures that the software is continuously adapting to the needs of its programmers, offering the latest tools to implement complex programs. More than 70 open source projects are merged and contribute to the functionalities that Eclipse offers. Such

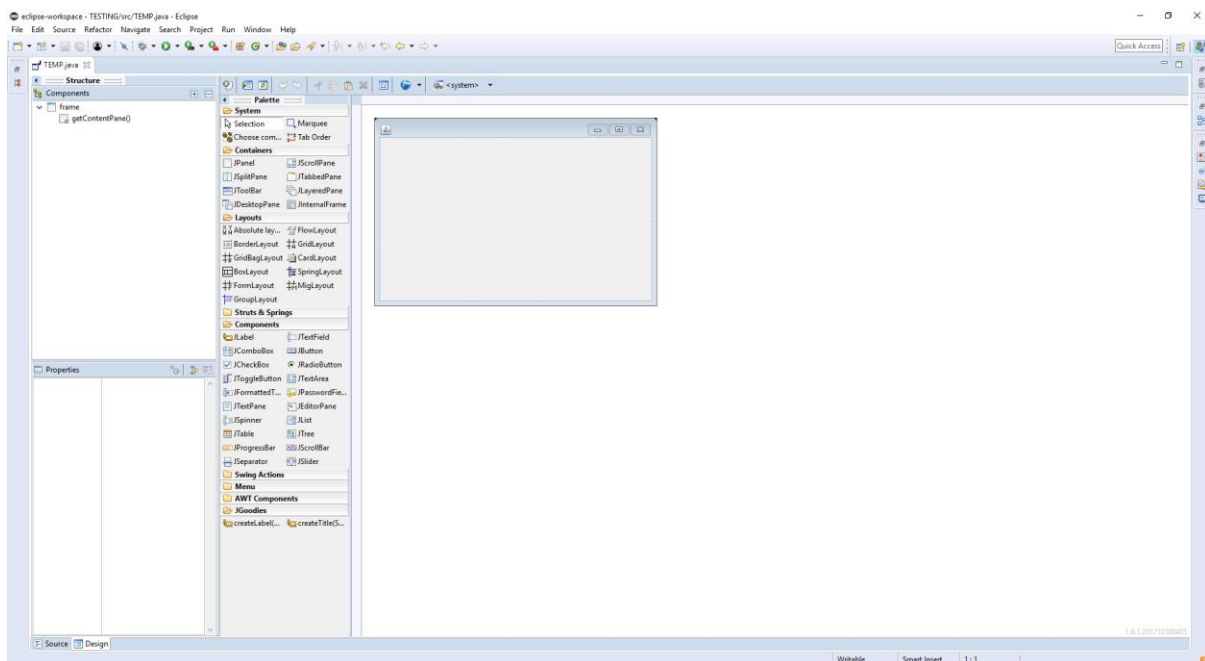
projects include; Modelling, Jetty, Web Tool Platform, Higgins Project. Eclipse is the software I have extensively used within my academic course, that has enabled me to implement java applications, that span multiple classes which are clearly presented by the software. The software is easy to use and I will be using this in the construction of my java code, which will implement everything that is required for the construction of the proposed copy-detection tool. [19]

3.2.1. Eclipse WindowBuilder (SWT Designer & Swing Designer)

A graphical user interface will be required by my copy detection tool, used primarily to display the results of the similarity matches between a pair code. This means that an appropriate plugin offered by Eclipse needs to be installed, one that meets the requirements for my GUI, such include, a way to display text, input text, file directory selector, buttons etc.

WindowBuilder is composed of two designers, SWT and Swing, it's a powerful and very easy to use Java graphical user interface designer that makes it very simple to design and implement a Java GUI without spending a lot of writing the code manually from simple forms to complex windows. WindowBuilder offers a drag and drop feature that lets you add controls to the window, event handlers to control how they operate and many other features that let you modify the controls to best suit your needs, the placed features will have their code automatically generated. WindowBuilder is a bi-directional Java GUI designer which means that the programmer can seamlessly move between two different modes of design, from the drag and drop design to the generated code. WindowBuilder is built as a plug-in on Eclipse and can be easily installed on to any system. [20]

Figure 3.1: Drag & Drop (WYSIWYG)



3.3. Software Development Life Cycle

Streamlined software development relies on a methodology that is consistent and follows a clearly defined process, that gets you from one point to another, A to B. A system exists that defines several methodologies that can be used for a project, this is known as the Software Development Life Cycle (SDLC). SDLC is a process which enables a software to be produced with lower cost, higher quality while being produced in the shortest time possible. SDLC achieves this by following a detailed plan which illustrates several distinct stages (Figure; these include, planning, analysis, design, development, testing, and finally evaluation. Throughout these stages all possible information that is required is anticipated, reducing the costly mistake of failing to identifying key areas, such as asking the target audience for suggestions. This is important as it typically reduces/removes the pitfalls faced during the software development process. [21]

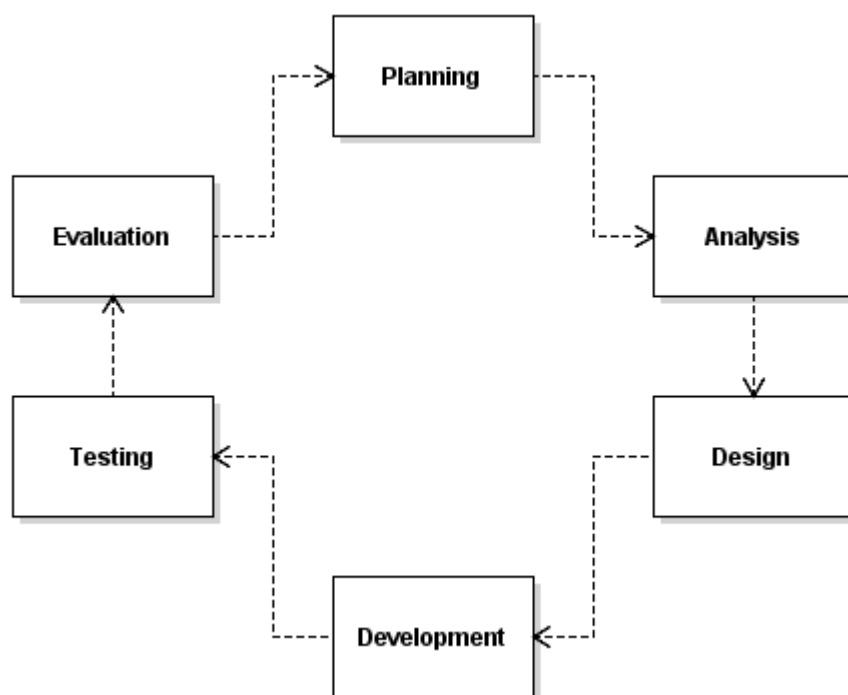


Figure 3.1: Software Development Life Cycle

Figure 3.1 describes the multiple phases of the software development cycle. Additionally, each of these phases will be briefly explained.

1. Planning

The very first phase of the SDLC involves planning the project. This phase outlines all of the chapters that need to be followed to complete the project, it is useful as it provides a visualisation and controls the development. This phase describes the introduction to the project, the goal, objectives and scope of the project, detailing the constraints and resources of the project which will enable a realistic plan that clearly represents the project to be defined.

2. Analysis

This phase of the project is an extensive analysis of the project, the purpose of this phase is to identify all the information that is required to fully grasp how the system will work, by firstly understanding what the project is, then researching into the current systems or tools currently employed that implement what you are trying to achieve, and how these systems work by understanding the underlying algorithms or techniques they use. This phase should identify the requirement for your own system, from the algorithm, techniques, appearance etc.

The technologies that will be required to implement the development phase of the project are also identified and the reason why they have been chosen should be state.

3. Design

This phase of the life cycle, takes in all of the requirements from the previous section and designs a system around them, which will be able to fulfil them to a high standard. This phase should also outline how to create the necessary diagrams and descriptions on what each part of the program is and how it will be developed.

4. Development

Upon completion of the project plan, the analysis of the proposed system and a design has been constructed which is derived from the set of specification and requirements that were identified, the development phase begins. In this phase will implement the proposed system, by coding the necessary classes/ methods, setting up the necessary databases etc.

5. Testing

The purpose of this phase is to insure that the system has been created successfully, without any errors or missing parts. The system is tested against the proposed requirements that were defined in the analysis phase, any issues that are identified will be resolved, for those errors that cannot be remedied, a description detailing why it was not is made. Once the system has been completely finished, the system is deployed to the task it was design to carry out. This phase should illustrate how the system works, so that anyone required to use it knows how to.

6. Evaluation

This is the last phase of the software development lifecycle. This step evaluates the system, on how well it works, the issues and future improvement. The system is also evaluated against all of the requirements and specifications it was built upon.

3.4. Software Development Methodologies

There are several different SDLC models, defining and designing their own process, each with their own unique step of steps. Several models that could be used for my project are defined below:

3.4.1. Waterfall Model



Figure 3.2: Waterfall Model

The Waterfall Model is the oldest SDLC model, and it follows the most straightforward linear sequential flow. Each phase depends on the previous, in that the next phase cannot begin until the current one is completed, thus each phase “waterfalls” into the next. The drawback with this approach is that everything that could be defined, must be defined, otherwise mistakes and errors will take place because of missing information or tasks. Also the incompleteness of a small task will hold up the entire project. Once a stage is completed you cannot simply move back and adapt it.

In regards to my project, I would be required to complete each task before beginning the next, this will be difficult and cause a lot of problems, as I will need to need to prototype with my copy detection design and implementation as I build it. [21]

3.4.2. Iterative Model

The iterative model follows a rinse and repeat attitude, instead of starting with a complete set of requirements, the project implements a set of smaller software requirements, tests them and then it evaluates the software, further requirements are identified and the process repeats again. At each iteration a new version of the software is defined, until there are no more requirements left, the software is complete. The benefits of this methodology is that a working version of the software is implemented early on in the project, it also is less expensive to make changes to the software. Project that need to be developed quickly will benefit from this model. [22]

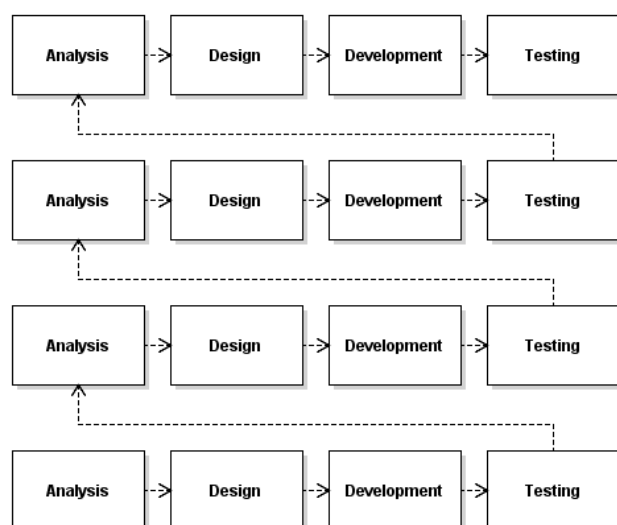


Figure 3.3: Iterative Model

3.4.3. Agile Model

The Agile approach to development involves producing ongoing release cycles, where each cycle featuring a small incremental change from the previous release. In this model, “fast failure” is a good thing, each iteration of the cycle is tested, identifying small issues that will be remedied, this avoids a set of small errors/issues building up towards something more significant. Each phases implements a section of the overall project, these sections are cleared of errors, thus the project continues cleanly. This approach is good for small project, and when the team designing the system is small. [21]

3.4.1. Chosen Software Development Methodology

The Agile Model has been chosen as the most appropriate SDLC for my project. Based upon the research I have carried out I have identified all of the requirements that my project needs to implement, the underlying algorithm that my system is based upon is RKR-GST, it have been heavily documented by its creator. I have identified one area that will be more difficult to implement, which is the Normaliser, from the research I have carried out I have not identified a document or report that describes how to implement it, only the requirement that the normaliser should implement are identified. Thus, I will need to make small iteration of my code and at each stage test to see that the requirements are met before I continue. This way I will be able to complete the several requirements of the normaliser in turn, one after the other.

3.5. Summary

In this chapter we have outlined the technologies that will be used to implement the copy-detection tool, the reasons for the chosen technologies and software have been justified and we have also identified the software development life cycle that is best suited for our project.

4. Requirements & Planning

This chapter aims to provide a comprehensive overview of the code detection tool requirements. The requirements for the proposed system have been split into functional and non-functional Requirements. Based on the research that has been undertaken in the literature review section, an overview of the proposed system is described. Lastly, a section on planning is included, that defines the time and resource requirements for the entire project, and the resources that are needed at each step for the project to progress.

4.1. Overview of Proposed System

The copy detection tool will be designed to detect cases of plagiarism in source-code that are written in the programming language Java, however it must be designed appropriately to allow for future extensions, which shall enable it to deal with other programming languages, such as python. Thus the tokenisation (normaliser) class should be extendable.

Language Support: Extending the copy detection tool to other languages should be relatively easy, the main aspect to which this feature is important too is the normalisation phase as it deals with the tokenisation of the file string based on the language it is written in. This means that a very java specific approach will not be used, e.g. not hard coded into the methods but rather as parameters the methods read, thus this approach will allow for the information to be changed instead of the methods. (Where ‘information’ identifies the java keywords, or python keywords etc.). The program should not degrade the performance of the detection when extended to other languages.

My copy detection tool is based on the Running-Karp-Rabin Matching in Greedy-String-Tiling (RKR-GST) and hence it must comply with the desirable properties described in section 2.4.1. These properties are: Insensitivity to simple changes, such as whitespace, comments, capitalisation and the renaming of variables identifiers. Position independence, transposed substrings should have a very small effect on the final similarity value, e.g. shuffling the order of the program code, added or removing parts of code. Breaking a statement into several statements that produce the same output should not result in a disproportionate degrade in the value returned.

So ultimately will my tool detect plagiarism? No, it will not, instead it will help educators and also their students make informed evaluations of the students work rapidly, saving valuable time, but more importantly it discerns the areas of work that the educator should support the student in, the set of instruction that will help further build the students ability, any corrections that need to be made and finally any judicial actions.

My copy-detection tool will simply state a similarity value between a pair of programs, it will not determine whether a program has been plagiarised. There is no score that is inherently “good” or “bad”. For example, a similarity score of 0% does not necessarily mean everything in the code is okay, similarly an 80% result doesn’t indicate that the code has been copied. The determination of whether the program has been plagiarised is in the hands of the examiner, and thus a useful feature for all detection tool is the visualisation of found ‘plagiarism’ matches.

My copy detection tool must be able to visually present the locations of found similarity matches in each pair of programs compared or the source of other features. The presentation must be adequate enough to allow for a clear manual exception by an examiner.

The detection performance for my program should be relatively similar to systems that also employ RKR-GST, and this performance should be able demonstrated on a variety of different tasks, e.g. a pair of source code should be compared on both systems and examined. The runtime performance for plagiarism detection in a pair of programs should preferably take less than 30 minutes for file of significant size, while smaller files should have an appropriately smaller runtime.

User Interface: The copy detection tool will be accessible via a GUI for the user, this interface will present all the necessary information to perform a comparison, this includes features that will have the user specify the directory path for the files that will be compared. The interface will also house the visualization for the match found. The interface must be user friendly and support any needs that requires user input.

Further Extendibility: Currently the only mention of extendibility is in regards to the extension of the application to support additional programming languages. However, this is not the only feature of application that needs to be designed to accommodate improvements. The proposed system is designed for the comparison of a pair of programs, but should not be limited to a design that only considers this. The system shall be designed so that it can deal with multiple programs being compared, rather than just 2. This is vital as programs usually do not consist of one or two classes, but tens of classes, thus it would be more appropriate for the system to compare an entire application against another application, e.g. 10 programs compared against another set of 10 programs, where each class in the first program is compared against all other classes in the other program, or each program is compared against the file that has the same name. This is not a feature of my application as it exceeds the scope of my project, I have either the time or the coding ability to construct this feature.

4.2. Ethical Issues

An issue that is of particular importance to this project is in regards to the storing of students' work into this system; the files that are read into the system should not be stored by our application, we do not own it and should discard of the data once the user is done with it. Because of this ethical issue, our system will store the results of the compared files on the user's computer, it will not be uploaded to a database or stored on another system that we can gain access to later.

4.3. Requirements

Functional requirements specify something that the system should do. Typically, functional requirements will specify a function or behaviour that must be implemented within the proposed system. For example; displaying the results of the plagiarism comparison, or the name of the files. Non-functional requirements specify how the system should behave, it is used to evaluate the operation of the system, rather than any specific behaviour.

4.3.1. Functional Requirements

This is a task that the system should perform.

ID	User Requirement
FR01	The System should allow the user to select the base directory that houses the files needed for comparison
Explanation	This will avoid the need for the user to manually input each file into the system, instead all the files can be located within one directory, that can be selected in the graphical user interface.
FR02	The system should allow the user to select the programming language of the files to be compared.
Explanation	The user will be able to choose the programming language of the files that they want to compare, this is enable the system to check the validate the inputted files and insure they have the correct language extension.
FR03	The user will be able to input file names to be blacklisted from the comparison.
Explanation	The user will be able to name the files that might be contained in the base directory, these files will not be parsed into the system, thus not compared.
FR04	The user will be able to perform comparison on the files located in the folder directory that they chose.
Explanation	The system should run a comparison on all the files that are contained in the base directory that was selected by the user.
FR05	The system should allow the user to change the sensitivity of the comparison
Explanation	This will be done by allow the user to choose the minimum token match during the comparison setup, allowing the user to experiment with the system until it is optimal for the chosen files. E.g. the value can be changed until the correct sensitivity is chosen which will identify all of the tokens that have been plagiarised.
FR06	The system should allow the user to examine the matches that have been found to be plagiarised.
Explanation	The system should display the results of the comparison, by highlighting all of the pieces of code in the program that have been matched as plagiarised, this will be presented in the window where the user will be able to view the highlighted program sections.
FR07	The system should save the results of the comparison on the users system.
Explanation	This will allow the user to continue to view the results after the application has been closed. The similarity result, the number of tokens and the highlighted code statements should all be saved by the system.

4.3.2. Non-Functional Requirements

These are requirements that identify the system operations.

ID	Operational Requirement
NFR01	The system will be accessible at any time of the day.
NFR02	The system will load within 5 seconds
NFR03	The system will work on windows 7 computers and higher.
NFR04	The system will comply with the Data Protection Act
NFR05	The system will not infringe on the privacy of the user's computer
NFR06	The system will produce the plagiarism result within 5 second from being run, on a pair of programs.

4.4. Planning

ID	Task	Complete By
1	Topic Selection	8 th August 2017
2	Initial research and understanding the project	31 st August 2017
3	Complete Report Chapter 1: Introduction	31 st September 2017
3.1	Project Introduction	10 th September 2017
3.2	Project Spotlight	12 th September 2017
3.3	Goals Objective Scope	26 th September 2017
3.4	Structure of Report	30 th September 2017
4	Literature Research	29 th October 2017
4.1	Reading and Research in to Existing Tools and Algorithms	18 th October 2017
5	Complete Report Chapter 2: Literature Review	24 th November 2017
6	Research into Programming Language, Software and Software Development Life Cycle	30 th November 2017
7	Complete Report Chapter 3: Language Software Methodology	1 st December 2017
8	Complete Report Chapter 4: Requirements & Planning	10 December 2017
9	Understand how the Application Should be Designed	20 th December 2017
10	Complete Report Chapter 5: Design	30 th December 2017
11	Implement the Copy-Detection System	1 st April 2018
12	Complete Report Chapter 6: Implementation	15 th April 2018
13	Test the Core System	20 th April 2018
14	Complete Report Chapter 7: Testing & Evaluation	31 th April 2018
15	Complete Report Chapter 8: How to Run/Use Application	2 nd May 2018
16	Complete Report Chapter 9: Conclusion & Future Work	4 th May 2018
17	Research on the Ethics relating to my project	6 th May 2018
18	Complete Report Chapter 10: Statement of Ethics	10 th May 2018
19	Proof Read the Report and Insure Everything has been Included	15 th May 2018
20	Make and Complete Video Presentation	20 th May 2018
21	Final Check to Insure everything is working	21 th May 2018
22	Submit the Completed Project to SurreyLearn	22 nd May 2018

Each Task must be completed before the next task can begin. For example, task 7 cannot begin until all other tasks have been completed it, this will insure that at each stage I have the necessary knowledge to carry out the current task.

4.5. Overall Approach for Designing and Implementing

In a bid to develop a copy-detection tool that would meet my goals, objectives and requirements, I initially developed a good structured methodology to base my project around. It was implemented to insure that I have all the necessary information to create the copy-detector, thus rigorous research was undertaken, where information was not gathered I will have to implement these tasks, by first splitting the requirements of that task and building up incrementally. The overall approach table below covers this methodology.

Project Approach Steps	
1	Research and understand how copy-detection systems works, and more specifically identify the different types of algorithms used.
2	Research into the different methods that these systems use and identify the one that I will be implementing and research into those specific copy-detection tools and the algorithms that exist under this method in greater detail.
3	Choose the algorithm that will be used for the implementation and research into how it will be developed.
4	Research into the normaliser class and how to implement it. Understand the requirements that are need to make it successful.
5	Research into and understand how to implement a file parser that will identify java code and read it into the system.
6	Understand how I can design this system so that it can be extendable.
7	Understand how the front end of the system will look and how to design it, what features must be included to get all the necessary inputs from the user.
8	Design the set of requirements that must be fulfilled by the system, both functional and non-functional.
9	Design any diagrams that will outline the structure of the system, such as Class diagrams, Use Case diagrams etc.
10	Implement the system based upon the designs and requirements developed earlier. Insuring the tasks are checked regularly for any errors that can become more problematic later in the development cycle.
11	Test the final system that has been implemented, against the requirements and purpose of the project. Furthermore test the system against other tools that are similar.

4.6. Summary

Now that the system requirements, the planning and the overall approach of the project has been defined we can move onto designing the application based on this information.

5. Design

In this chapter the design and development of this projects copy-detection tool is detailed.

5.1. Principles

The objective of the application developed in this project is to perform plagiarism detection on a pair of programs written in the Java programming language. Future extensions to the amount of programming languages that can be compared will not be hindered by the design of the current system, extensions will be unencumbered and implemented in an appropriate amount of time. The copy-detection tool includes loading the pair of programs, the information contained in them will be transformed by the normaliser, where all the undesired information is removed making them ready for processing. The transformed programs are pushed into comparison, the similarity result is calculated and the matches highlighted in the normalised string and matched against the original programs (before they were normalised) and displayed on the GUI for the user to inspect. In order to be able to deal multiple different programming languages, the application uses a generic front-end.

5.2. Top-Level Architecture

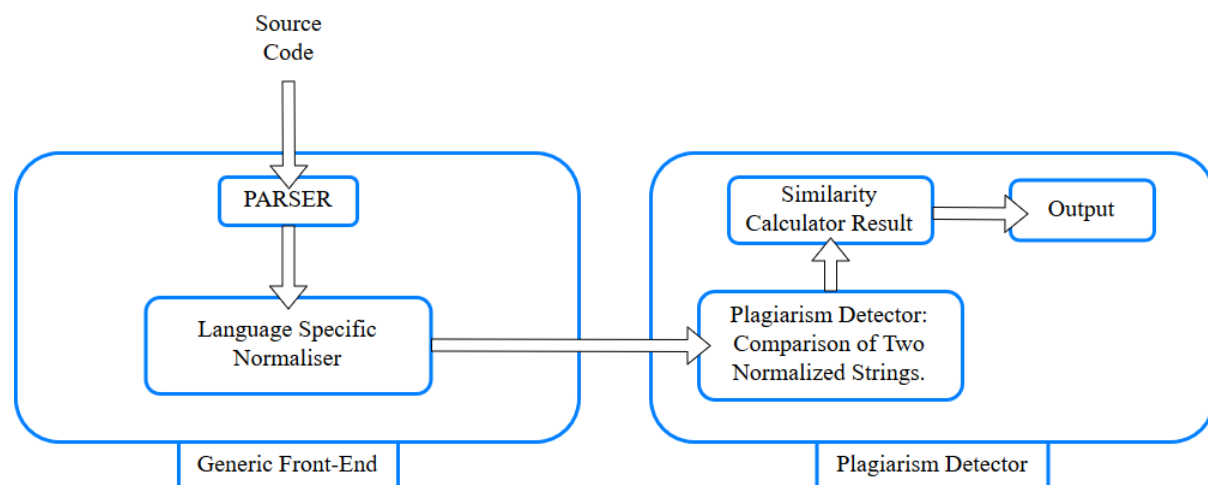


Figure 5.1: Top-Level Architecture Diagram

Figure 5.1 presents the top-level architecture of the application; the diagram is composed of two main parts, the generic front-end and the plagiarism detector, note that the graphical user interface is not shown in this diagram. [4]

In the first part, the generic front end is outlined. The parser reads in source code written in any language and processes the programs based upon their language, which will be identified by their file extension, this is important feature that will allow the first part of the system to be extended, by detecting the language the normaliser can chose the correct parameters to base its tokenisation on. This step insures that all programs are transformed into the same manner,

before they are sent for comparison. By the end of this part, the input programs should be transformed into a sequence of tokens that represent the essence of the code.

The execution of the program is design such that extensions to the number of files to be compared at one can be increase. For instance, rather than comparing a pair of programs, the system can compare entire applications against each other, where each application consists of tens of files.

This thesis is primarily focused on the detection of plagiarism, thus in section two we perform the comparison phase. The normalised java programs are inserted as parameters into the Running Karp-Rabin Matching and Greedy String Tiling algorithm. The algorithm I have chosen to used is a structure-metric system, that has been deemed by Verco and Wise to be more efficient than abstract counting systems. The strings are compared in pairs and the found matches are stored as tiles, the tiles are then used to calculate the final similarity result. The found matches in the programs are highlighted and output to the user.

The output must first be transformed back into the original strings, before they were normalised, where the found matches in the normalised strings are identified in the original string and highlighted. If this step is not taken then the output will be a series of highlighted token strings, which will not allow the user to examine the code.

A graphical user interface is built upon the two main parts above. The interface is design to accommodate any user input that is required to begin the top-level architecture. This includes requiring the user to select a base directory to which the pair of programs will be located. The selection of the programming language that the programs are written in and any file extensions that should not be compared. Finally, the output in the Plagiarism detector phase will be parsed into a viewer on the interface, that the user will be able to see and use to compare the matches found.

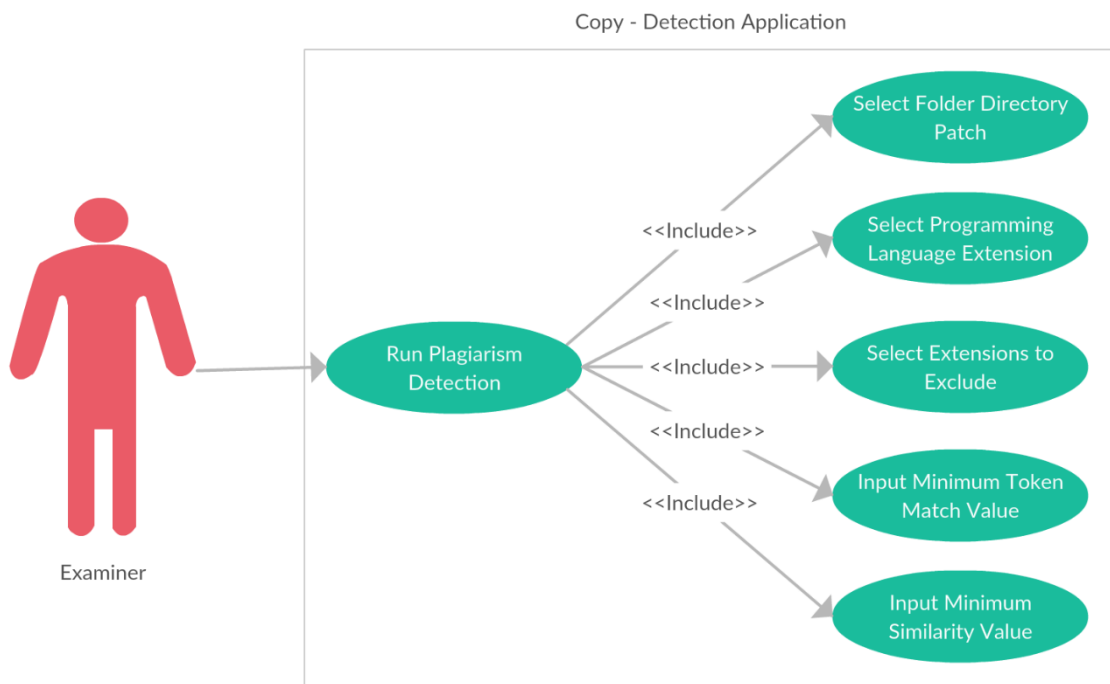


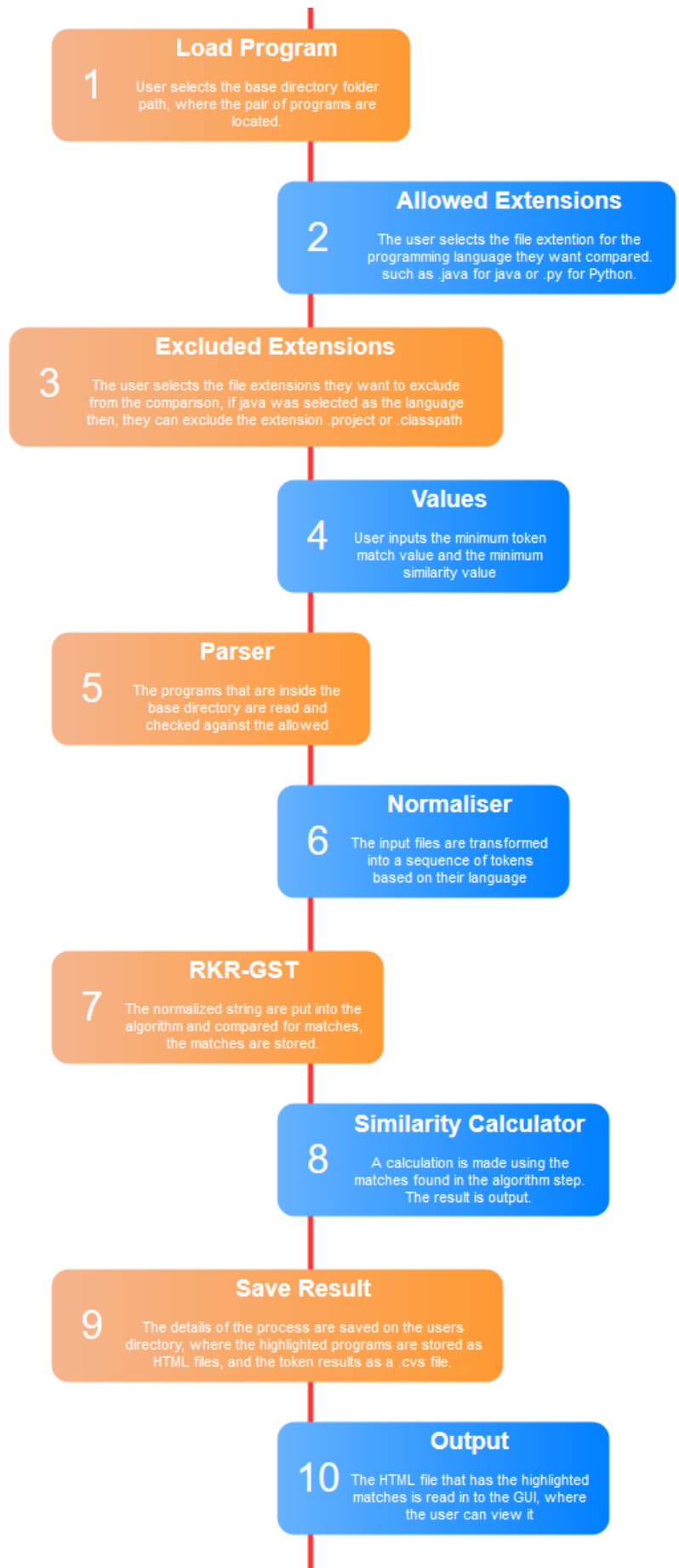
Figure 5.2: Use Case Diagram

Figure 5.3: Application Steps Diagram

5.3. Application Steps

The Use Case Diagram in Figure 5.2 is representation of the user's interaction with the copy-detection system, it shows the many relationships between the user and the use cases, to which the user is involved. In order to run the plagiarism detection, the user must first "include" the several use cases, this means that each of the use cases must be implemented by the user before they can run the comparison.

To better understand how the application process will take place I have created the following diagram. The steps that are undertaken throughout the application are listed in Figure 5.3, there are a total of 10 steps that summarise several tasks. The steps from 1 to 4 are user controls, while the rest are automated by the application.



5.4. Blueprint ~

In this section we outline the different areas of the system and detail what roles they play in the applications, several diagrams are included to support the descriptions, in each part of the application we talk about what the output should be and what is required beforehand in order to progress to that current area.

5.4.1. Graphical User Interface [Frame]

In order to make the user interaction with the implemented system easier the use of a GUI is required, this will enable the users to interact with visual indicators rather than a text based interface. The Graphical User Interface is built on top of the copy-detection system. However, the copy-detection system does not require the GUI to run the system, this can be done programmatically, because the interface is design separately from the system. The GUI is used to input the necessary data that will be used to direct the system.

1. The user is invited to select the home directory the houses the two individual folders that contain the programs that will be compared against each other, which is illustrated in Figure 5.5 Structure 1. The selected directories path is stored in a variable, to be later used in the Parser to extract the programs.
2. The selection of the programming language that the programs are written in and any file extensions that should not be compared.
3. Two pairs of value inputs by the user are optional, if they remain untouched then the default values will be used by the system.
4. Finally, the output in the Plagiarism detector phase will be parsed into a viewer on the interface, that the user will be able to see and use to compare the matches found.

Select Directory Directory Path: _____

Select File Extention File Extention: _____

Excluded File Extentions Excluded Extention: _____

Minimum Token Match _____

Minimum Similarity _____

PROGRAM ONE SIMILARITY RESULT

PROGRAM TWO SIMILARITY RESULT

Figure 5.4: Folder Structure

5.4.2. Parser [ReaderWriter]

In order to compare two programs, the system need to be able to read in files and identify their extensions to determine the language they are written in, and compare that against the user input on allowed extensions. Rather than having the user input the programs to compare 1 by 1, I have decided that it would be more appropriate to have a folder directory structure that consists of one base directory that contains two sub-folders, each folder holding 1 or the 2 source-codes, Figure 5.5 Structure 1 illustrates the structure.

The system will be designed such that future extensions of the folder structure can be easily integrated into the system. Structure 2 is a more common structure of an application, that consists of multiple programs codes. In structure 2 we can see that 2 folders are used to contain entire applications rather than single files, note however that the folders can contain more than 2 files. In this case the files in the first folder will be compared against each of the files in the second folder, this will be challenging to display on the final GUI result as there will be multiple tabs required to visuals all the comparisons.

Comparing 2 vs 2 files will require 4 tabs, each tab presenting a pair of programs that have been compared. In Structure 3 we can see that multiple folders are used, each folder containing 1 file, this specific case is similar to structure 1 and thus it will be implemented in the development, as the method that scans the files, will look for each folder, while there are still folders in the directory then the files within them will be parsed. However, I will not accommodate for such as execution, this will require more development time that I do not have, the time would be spent on presenting the comparison on the GUI.

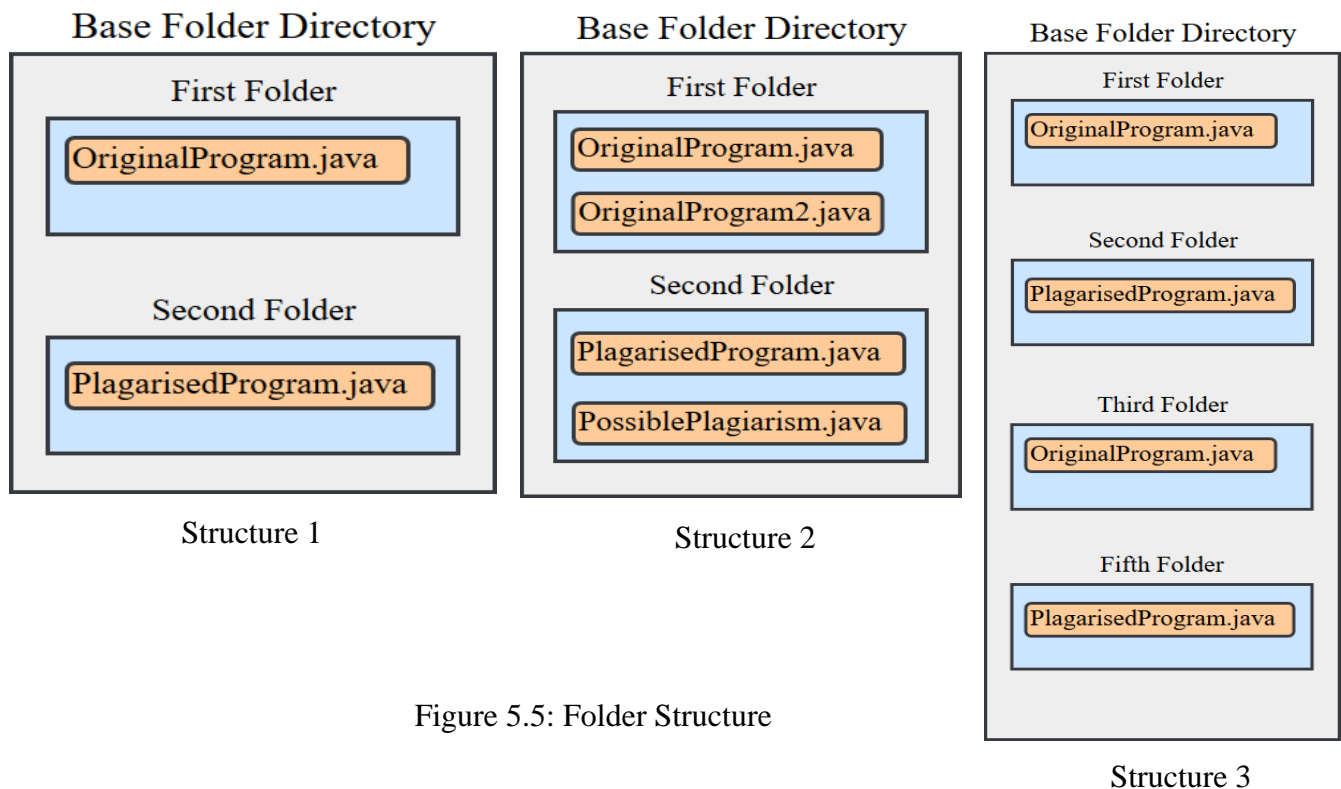


Figure 5.5: Folder Structure

Overview - Parser Operation:

1. A method will be designed to locate the directories within the path that has been specified by the user (e.g. the main directory), and for each of the folders within that main directory, the absolute paths will be stored, then another method will be called to read the files from that directory.
2. The method that gets the file from the directory will use the sub-folders absolute path to locate any files that are within that path, the method will need to scan the file name to determine if it is allowed, e.g. whether the file extension has been permitted by the user or whether it has been blacklisted. The file is read and stored in the system, using a StringBuilder.

This class shall also be responsible to creating the necessary files that can be used by the “Examiner” to review the results of the comparison. These files include a plagiarism indication HTML that will mark all the tokens in the programs that match as “plagiarised”, the mark will be in the form of a HTML highlight that will span the whole length of the token. StringBuilder will be used to insert strings that will transform the string into a HTML file, such as HTML declaration. This will enable me to insert additional HTML code into the string that can be used to highlight the code at specific sections. The plagiarism similarity result and the plagiarized tiles value (minimum token match) will be stored in a .csv file.

Overview - Writer Operation:

1. Identify the tokens that have been plagiarised in each of the programs
2. For each of the programs mark the identified tokens, by highlighting them.
3. Write the two highlighted strings as a .html file.

Once the HTML file has been created after the comparison has been completed, the HTML file will be parsed into the GUI visualizer for the examiner to view. The CSV file will not be read into the system, itself and the HTML file will be save into the directory of the copy-detection application. It will remain untouched until the user runs the system again, this will overwrite the current files located there with the results of the new comparison. To save the results of the comparison the user should move the created files into a new location.

5.5.3. Normaliser

This copy-detection system works in two phases. In the first phase, the normaliser transforms the input programs into a sequence of tokens, based on the language they are written in, removing undesirable properties that can be used by the plagiariser to disguise their code. The steps involved in this process are:

1. Pre-Processing

Searches through the content string for class and function definitions which are split over multiple lines. The definitions parts are combined into a single line, and the string is returned. This removes situations where the plagiariser has visually altered the look of the definitions, for example: in the java programming language you can split class and function definition over multiple lines, a class definition that looks like this:

```
public class MultithreadingWorker implements Runnable {
```

Can be split into multiple lines, and will not affect the functioning of the code, nor will it produce any errors.

```
public
class
MultithreadingWorker
implements
Runnable {
```

Such a change is important to address as it will interfere with the next steps in the normalisation. The question arises as to how one will be able to identify class and function definition, especially those that have been split over multiple lines, this can be implemented using Java Regular Expressions.

For class definitions, I have created the following Regex that will be able to identify all the different class definitions which are valid in Java. It consists of multiple OR statements that identify different definitions of a class, from the default “class name {“, to “public class name {“, and more complicated ones such as “public class name extends name implements name {“.

```
((((public|private|abstract|static|final|protected)\s+class\s+(\w+)\s+((extends\s+(\w+)|(implements\s+(\w+|( ,\w+)*))?\s*{)|(class\s+(\w+)\s+((extends\s+(\w+)|(implements\s+(\w+|( ,\w+)*))?\s*{)|(class\s+(\w+)\s+((extends\s+(\w+)\s+implements\s+(\w+|( ,\w+)*))?\s*{))((public|private|abstract|static|final|protected)\s+class\s+(\w+)\s+((extends\s+(\w+)\s+implements\s+(\w+|( ,\w+)*))?\s*{))
```

Similarly, for method definition I was able to create the following Regex that deals with all valid method definitions. It is important to note that these two regular expressions detect the class and method definitions that span multiple lines.

```
((public|private|protected|static|final|native|synchronized|abstract|transient)+\s)+[$_\w\<|>\w\s\[\\]]*\s+[$_w]+([^\s])?\s*{
```

The regex is able to deal with the following definitions:

```
private static String reorderFunctions(String normalisedString,) {
    and also multiple lines:
    private
    static String
    reorderFunctions(
    String normalisedString,) {
```

The two regex defined here can be stored into separate string variables and used within a method that can utilise them.

2. Removing Comments

The next step involves removing all of the Java comments from the program, from single line comments that are declared using “//” to multi-line comments which are defined using “/* */”. The comment definitions can be identified by using two Java Regular Expression that will be responsible for detection the single line or multi-line comments. Instead I have decided that it will be more efficient to simply detect both of the occurrences within the same Regex, and thus the following expression has been defined.

```
(/^(^*|[\r\n](\^(^*/|[\r\n])))*\^+/(/.*))
```

Example of comments detected: shown below:

```
/*
 * Array of numbers, where each index goes up to the total length of the string
 * lines. thus lineNumberList = total number of lines, each entry is the number
 */
// check for valid string argument
```

3. Removing String-Constants

This step involves identifying every occurrence of a String-Constant in the program and removing it. In java there are two possible definitions of a String; double quotes “” and single quotes ‘’, with the information inside the quotes also identified and removed. The following two Regex have been designed:

For double quoted: \"(.|\\n)*?\"

Result: "removeStringConstants – double Quotes

Even when On multi line "

For single quoted: \'(.|\\n)*?\'

Result: 'n' or ')

4. To Lower-Case Letters

Java has a built in method that is able to translate upper-case letter to lower case, and this can be implemented on an entire string that consists of hundreds of lines. This can be done using the following code.

```
String temp = "Some vast amount of text that have upper case letters"

temp = temp.toLowerCase();
```

5. Mapping Synonyms

This step involves mapping synonyms to more common forms, I have identified the following synonyms that need to be reassigned. On the left we have all of the synonyms and on the right side the desired form is shown.

Key	Value
+=	=
-=	=
*=	=
/=	=

This step can be easily implemented with the use of a HashMap, storing a key, value pair. In this case the synonyms are the keys, while the common forms are the values. A method can be used to utilise the HashMap, with a for loop that identifies every occurrence of the key in the program string and replaces it with the value, for each key in the HashMap.

6. Reordering Functions

In this step every function in the program is reordered into their calling order. The process involves expanding each function to its full sequence of tokens when it is first called in a program. Then all other subsequent function calls are replaced by the token word "FUN". This insures that the reordering of functions by a plagiariser does not affect the similarity result. In this section the beginning and end of each method and class should be marked, by the words "BEGIN_CLASS", "END_CLASS", "BEGIN_METHOD", "END_METHOD".

For example, a class is defined that has 3 methods, the "thisHouse" method is called twice, the first time in the "thisWorld" method and the second time in the "temp" method. Once the program undergoes the Reordering Function step the output will be:

```
public class example {
    public String thisWorld() {
        thisHouse();
    }
    public void temp() {
        for(int i =0; i<200; i++ ) {
            System.out.println(i);
        }
        thisHouse();
    }
    public void thisHouse method() {
        for(int i =0; i<200; i++ ) {
            System.out.println(i);
        }
    }
}

BEGIN_CLASS
BEGIN_METHOD
    for(int i =0; i<200; i++ ) {
        System.out.println(i);
    }
END_METHOD
BEGIN_METHOD
    for(int i =0; i<200; i++ ) {
        System.out.println(i);
    }
FUN
END_METHOD
END_CLASS
```

The beginning and ends of the classes and functions will be identified by the Regex defined in step 1, these will be utilised by an appropriate method and swapped for the keywords.

7. Removing Non-Target Language Tokens

The seventh and final step involves removing any and all tokens that are not members of the target language's lexicon, i.e. any token that is not a built in function, is a reserved word in the language, etc.

Java's Language Specific Keywords: These are the only keywords that are permitted by the normaliser. Any other keyword that does not match one of these will be removed.

public	void	private	boolean	true	false
null	abstract	assert	int	double	else
break	byte	case	catch	char	class
const	continue	default	do	enum	extends
for	finally	float	goto	if	implements
import	instanceof	interface	long	native	new
package	protected	return	short	static	strictfp
super	switch	synchronized	this	throw	throws
transient	try	volatile	while	Java Specific Keywords	

My own keywords for describing the structure of the program:

BEGIN_METHOD	END_METHOD
TEST	ASSIGN
BEGIN_CLASS	END_CLASS
FUN	APPLY

Patterns to apply: Here the normaliser will identify any string that matches the regex defined and will assign the found matches with the appropriate value seen in the assignment column.

Regex	String	Assignment
[^=\n]+=	Number =	ASSIGN
if .*{	If (condition)	if TEST
((\w+\\.)*\w+(((\s , \\[\\] \\w \\w+\\.)*\\n)*\\))	Method Call	APPLY

In the end, only a sequence of tokens should remain, where the values are a mixture of the values in the 3 tables above. It should look similar to this:

```
BEGIN_METHOD ASSIGN return ASSIGN APPLY ASSIGN ASSIGN APPLY ASSIGN
ASSIGN APPLY ASSIGN while APPLY if TEST ASSIGN ASSIGN APPLY ASSIGN
ASSIGN ASSIGN ASSIGN ASSIGN continue ASSIGN return ASSIGN for in ASSIGN
APPLY return ASSIGN for in ASSIGN APPLY return ASSIGN APPLY ASSIGN for in
ASSIGN APPLY return END_METHOD
```

5.4.4. RKR-GST

The second phase in the application is the algorithm process. Here we use the popular copy-detection algorithm Running Karp-Rabin Matching and Greedy String Tiling. The algorithm has been briefly explained earlier in this thesis, here we will dive deeper into how it shall be implemented. Note that when “two programs” are mentioned, these programs have been normalised into a sequence of tokens, implemented in the previous section.

The main objective of RKR-GST is the search for identical substrings in both files being compared, call them String1 & String2. The found substrings must comply with the requirements/rules listed below:

1. Each token must be matched exactly once.
2. Substrings can be discovered regardless of the position in the file.
3. It is assumed that longer substrings are more reliable than shorter ones, thus these the preferred matches.

Notions:

- **Tile:** A pairing of substrings from both files, will result in a new tiles being created with this token, once a token becomes part of a token it is marked.
- **Maximal Match:** this is similar to tiles, but the pairing of substrings may only be temporary
- **Minimal Match Length:** there is a minimum length that the algorithm is looking for and length that is below this is ignored, as it is most likely an artefact of the matching process. By default, the length will be set to 3.

Ultimately the algorithm is looking for a maximum tiling of the pattern string and the text string. i.e. a coverage of non-overlapping substrings from the text and pattern.

The first requirement rule implies that matching sections of the original source code which has been completely duplicated in the plagiarised source code is impossible. The second requirement insures that the reordering of statements/code blocks do not heavily impact the detection of plagiarism. i.e. if the reordered sections are longer than the minimal match length then this will not be an effective attack. Thus any reordered sections that are shorter than this value will be effective. Finally, when the third match is applied sequentially, for every matching step, it will result in an algorithm that is greedy, this greedy algorithm consists of two phases:

The first stage of the RKR-GST algorithm, the Karp Rabin procedure is used to search for the longest contiguous matches for two strings that are longer than the threshold value (minimum-match length). In the second phases, all the matches that have been found are marked, making them unavailable for further matches that can occur in a subsequent first phase iteration. This confirms that every token used will only be done so once. Once each of the matches has been marked the process begins again from the first phase until the match length is equal to the threshold value (minimum match length). [8]

The pseudo code for the first and second phases of this algorithm has been provided by Michael J. Wise in his documentation. [15] I will be using them to implement it in my application.

5.4.4.1. HashTable

This hash-table is used to reduce cost of comparison, rather than having the scanPattern method go through all of the text strings for the corresponding hash-value that matches with a specific pattern substring. The Karp-Rabin hash-value is hashed itself, and then a search through the hash-table will return all of the text substrings starting positions, that have the exact same Karp-Rabin hash value. Thus this class is created to store the information that is gathered.

5.4.4.2. TileValuesMatch

The algorithm will store all of the matched values in this class, it will contain the position of the pattern and text string and the length.

5.4.5. SimilarityCalculator

In this application, the Running Karp-Rabin Matching and Greedy String Tiling similarity measure of the token sequences in String1 and String2 can be calculated with formula 1 or 2:

$$sim(str1, str2) = \frac{2 * tilecoverage}{length(st1) + length(str2)}$$

Where tilecoverage is sum of all the tile lengths and the length(str) is the total amount of tokens in each token sequence. The similarity result can be calculated using another formula which is slightly different from the first. Below is the equation for formula2:

$$sim(str1, str2) = \frac{2 * tilecoverage}{slength(str1, str2)}$$

Similarly, for this formula the tilecoverage is the sum of all the tile lengths, however the slength is the smaller value of tokens from the two String token sequences.

Intuitively, the fraction of tokens that have been matched in the original program should be reflected in the similarity measurement. A maximum similarity value of 100% will be returned by Formula 1 and this will state that the two token strings are equal, both token strings must be considered in this computation. If, on the other hand, we find it more appropriate to have a similarity result of 100% when each of the programs has been entirely copied (and perhaps then extended), in this case we must only consider the shorter token strings. [23]

For this application I will be using formula 1, which will return a value of 100%, when the similarity of token sequences is identical to each other. Also, if the value of one of the token sequences is smaller than the other, the second formula will produce a rather high value.

This class will implement several methods, one that calculates the coverage; the sum of all the tile lengths. Another method that calculates the similarity result based on Formula1, and the third method that is called in a later class, which is responsible for setting a Boolean variable true, if the similarity calculated is about the allowed threshold that was input by the user in the GUI, if the user did not enter a value then the default 0.5 threshold will be used.

5.4.5.1. SimilarityValue

This class stores the information on the similarity, where the parameters are; a similarity Integer that stores plagiarism similarity between the files compared and Boolean value called suspectedplagiarism, which will be to True by the SimilarityCalculator if it is deemed plagiarized.

5.4.6. MultithreadingWorker

Java is a multi-threaded programming language, that allows for the concurrent execution of several parts of a program in parallel. Each part can deal with different tasks in a program, all at the same time, this makes use of all the available free resources on the computer.

I will be using this feature to execute the comparison of the two strings, as well calculating the similarity value. The implementation of this feature will enable future extensions to deal with folder directories that contain multiple programs as mentioned in section 5.5.2. Each pair of programs will have their own thread, where each thread executes the RKR-GST algorithm and calculates the result of this comparison. This system shall be built to deal with these future cases, but will currently only implement one thread that will handle the single comparison of the text and pattern string.

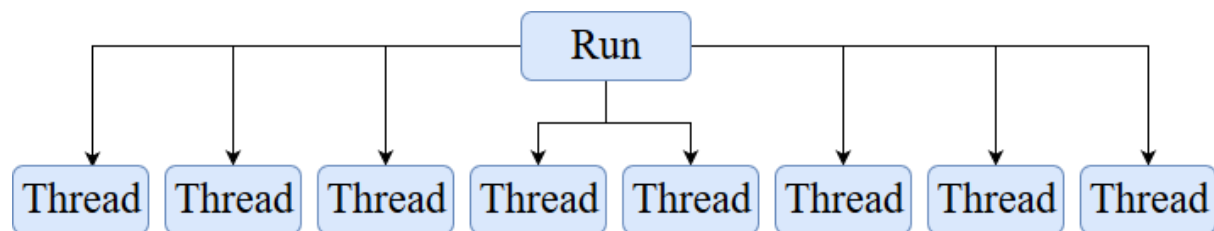


Figure 5.6: Multithreading Diagram

5.4.6.1. Job

This class will contain the information necessary to be used in the MultithreadingWorker, such as the contents of the two sub folders, i.e. the text and pattern strings. A Boolean value which will be used to state whether the job has started, additionally the similarity and suspectedPlagiarism value, also a list of the matched tiles.

5.4.7. UML Class Diagram

The Class Diagram defined in the Figure 5.7, represents the static view of the copy-detection application. It has been defined based on the information that has been gathered through my research and understanding, as outlined in the previous section of this chapter. The diagram describes all of the responsibilities of the system, it defines the attributes and methods that are required within each class in the application. The relationship between the different classes is also stated.

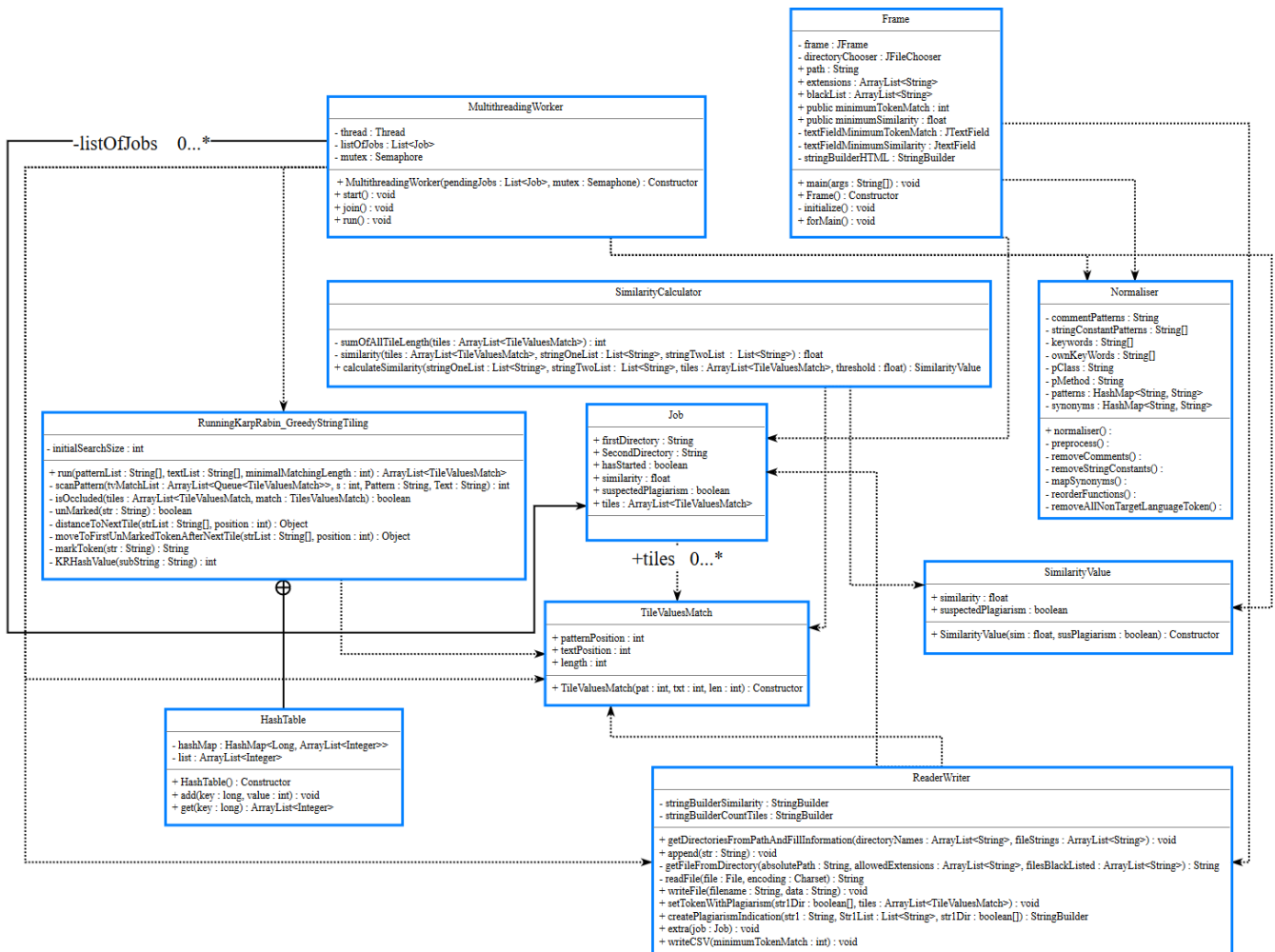


Figure 5.7: Initial UML Class Diagram

5.5. Summary

In this chapter we have described the features and designs of the different components within our copy-detection system, where required we have included diagrams that help to better present the system as a whole. These designs are made to insure that the requirements in the previous section are met and that the system is built correctly.

6. Implementation

In this chapter each Java Class in the application has its major methods explained.

6.1. ReaderWriter Class – Methods Explanation

1. `public static void getDirectoriesFromPathAndFillInformation(ArrayList<String> directoryNames, ArrayList<String> fileStrings) {`

The `Java.io.File` class is used in this method because it is able to represent a directory or file path name, this enabled me to distinguish whether the absolute path that is assigned to a variable is a directory or file, using the classes inbuilt methods. Thus I created a `File` object by passing in the absolute path `String` that locates the base directory.

An `ArrayList` is defined next, this list will store the directory based upon their absolute path, e.g. a sorted `ArrayList`. In order to sort the `ArrayList` I used the `Collection` class, because it is able to sort the elements of an `ArrayList`, queue etc. It has an inbuilt method called `.sort()`, the path `Strings` within the `sortDirectorys ArrayList` are passed into the `Collections.sort`, organising them in ascending order.

With the use of a `for` loop, we scan through each of the sub-directories absolute paths and check to insure that these are in fact directories and not files. A new `File` object is created for each of the sub-folders and then a method is called, which will extract the contents of the files from each of the folders (`getFileFromDirectory` method). The scanned files are checked to determine whether they are larger than 1000 characters, else it will not continue. If they are larger than the required amount, the names and folders and the contents of the files are stored in separate lists. The number of characters in the files are tested because there will be no logical reason to scan a program that is less than 1000 characters, this would be better suited to a manual inspection by an examiner.

2. `private static String getFileFromDirectory(String absolutePath, ArrayList<String> allowedExtensions, ArrayList<String> filesBlackListed) {`

This method fulfils the Non-Function Requirement “FR04”.

The next step in the class, after scanning through the sub-folders, is the scanning of the program source-codes that are contained in each folder. The method begins by defining two variables, a `File` object variable that stores the sub-folders `absolutePath` that was passed in the method definition. The other is a `StringBuilder` object that will store the contents of the file. This `StringBuilder` class is used because it will allow me to easily mutate the string, adding information later on in the method.

An `if` statement is used to determine whether the `File` object that has been defined is a directory, i.e. is a sub-folder. If the condition is met, the following set of code is run. A `File` object array is defined to store all of the files in the folder. This will mean that the folder can contain multiple files, rather than just 1, allowing it to be adapted in the future to accommodate the comparison of multiple files more naturally. Note that this application is designed to deal with only a pair of programs. However, where possible the initial steps to extended the system will be developed. This is one such case of that extension development.

For each of the files in the `File` object array, we test to see if these are in fact files and not directories, this is important because the user might accidentally put a folder within 1 of the

subfolders, such that the files needed for the comparison are located in the last folder. In such a case the else statement will be utilised, this will check whether the File type is a directory and rerun the method, with the new absolute path as the parameter.

If the `isFile` condition is met then two variables are defined, the first will store the name of the entire file, which includes the file extension. E.g. `worldClock.java`. Then the second variable will split the name into an array, where the delimiter is a “.”. This means that the name of the file and the extension are in different indexes of the array (`[name][extension]`). Now that we have isolated the extension, the system can check if it is a programming language extension that has been permitted by the user. Currently, the system should only take in Java Programs, because the Normaliser class will only be able to deal with that language. But the method does not check if the extension is just Java, it will be assumed that the user simply runs Java programs through the system.

The method will loop through each of the `allowedExtensions` and if the file extension is equal not equal to one of the blacklisted extensions then the program will continue onto reading the file into a temporary string that stores the return value of the `readFile` method. The `readFile` method simply reads all of the bytes and encodes it using the default charset. Finally, the `stringBuilder` variable has the file name, and the file contents appended to it and then returned.

3. `public static void writeFile(String fileName, String data) {`

This method fulfils the Non-Function Requirement “FR06”.

This method takes in two parameters, one that has the name of the file with its appropriate extension, that will be either `.html` or `.csv`. The second parameter contains the final highlighted program content or the csv data. We use the `FileWriter` class to create the files, this is because it makes it possible to write characters rather than bytes to a file.

We define a new `FileWriter` object and pass the `fileName` as a parameter, then several useful methods provided by the `FileWriter` class are utilised. The `.write()` method takes the `String` data and writes it to the file, then the `.flush()` method is called which insure that the buffered data is written to the disk destination, the destination in this case is the location of the application, which should lay in the eclipse workspace. It’s important to note that data can still be written to the stream after the flush method is called, thus the `close()` method is called to state that no additional information is will added.

This method is used to create two of the HTML pages and is utilised by another method that deals with making the `.csv` files (`writeCSV` method).

4. `public static StringBuilder createPlagiarismIndication(String str1, List<String> str1List, boolean[] str1Dir) {`

This method fulfils the Non-Function Requirement “FR07” and “FR06”.

Several parameters are passed through this method, the `str1Dir` houses are array of Boolean values that indicate whether a specific index in the array has been plagiarised, thus these are marked with the value `True`.

This method creates a viewable indication string that highlights all the tokens in the program that have been matched. The `StringBuilder` class is used to append the necessary modifications to the code. First the HTML declaration is appended to the beginning of the string, as to make

it the following strings, html. The method loops through the Boolean array and highlights the entire token, by appending `` to the start of each of the tokens and the adding the closing statement at the end of the token. Finally, the html closing tag is appended to the last position of the string and the `StringBuilder` is returned.

6.2. Frame Class – Methods Explanation

This class fulfils the Non-Function Requirement “FR01”, “FR02”, “FR03”, “FR06”, “FR05” and “FR04”.

This class was created using Swing Designer’s Application Window that is offered by the `WindowBuilder` plugin. Thus several default methods are automatically implemented by the program, these include the `main()`, the class constructor which calls the final automatically generated method `initialize()`.

Several global variables are defined, that can be called from other classes, the most important ones include the `ArrayList` of allowed extensions and blacklisted extensions, these variables store the extensions that have been picked by the user in the GUI. Two numerical variables are defined, `minimumTokenMatch` and `minimumSimilarity`, with the default values of 6 and 0.5f respectively.

1. `private void initialize() {`

This method is used to implement the necessary code that makes the graphical user interface, I mainly used the WYSIWYG layout window, that lets me drag and drop the necessary components onto the display, the `JFileChooser` is used to allow the user to select and scan through their system for the location of the base directory that houses the pair of programs, the path is saved and used as a parameter in the `ReaderWriter` class. The Labels that display the selected user choices are included on the display, the selection is made through a `ComboBox` for File Extensions to Check and BlackList of Files to Exclude, also a button that runs the plagiarism detector is added. Takes `ComboBox` input and stores them in to the appropriate array.

The actions that process the user input are placed within the appropriate `onClickListeners`, where necessary the user input is validated to insure that it is in the correct format. For instance, the `minimumTokenMatch` input is checked to insure it is not empty, and one that is greater than 1 and less than 15. The value type is not checked to be an integer, because the `JTextField` has been coded to only accept integers.

2. `public static void forMain() throws InterruptedException {`

This method is called within the `main()`, it is responsible for calling all of the necessary methods from other classes to run the program, such the methods that get the files from the directories, the execution of the `MultithreadingWorker` etc. It sets the two strings that need to be compared into the job class, then new job is added to the list. e.g. first directory (first for loop), is matched with all other directories (nested loop) and then finally writes the results of the comparison as HTML and CSV file using the appropriate methods from the `ReaderWriter` class.

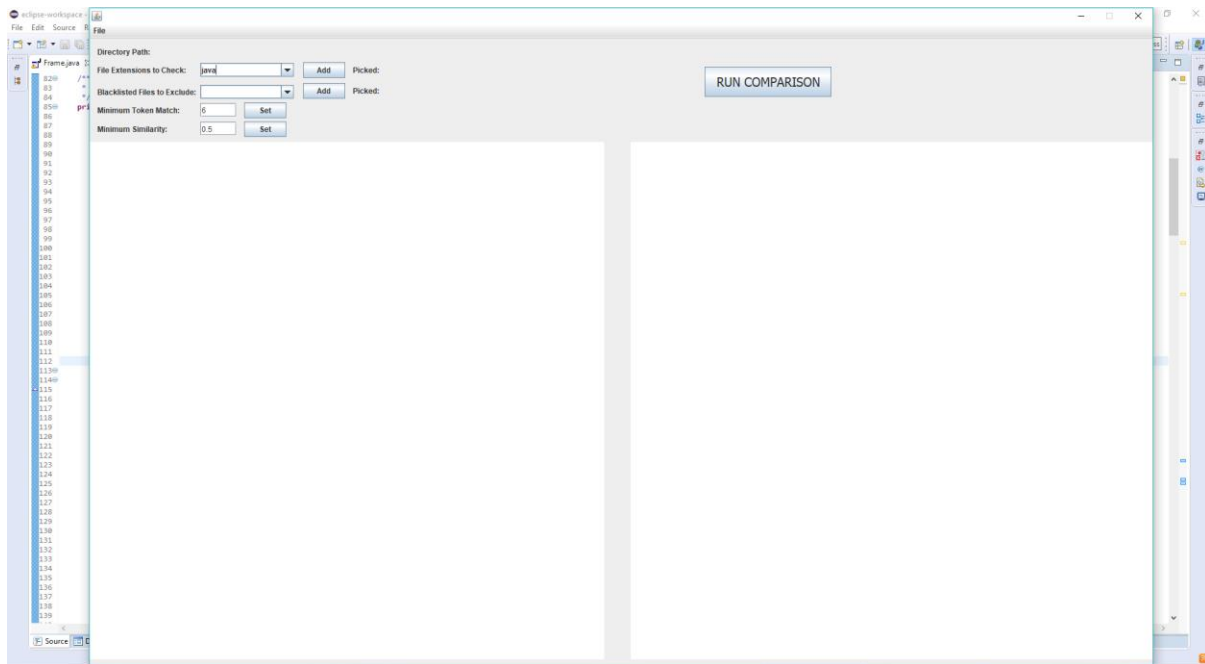


Figure 6.1: Graphical User Interface

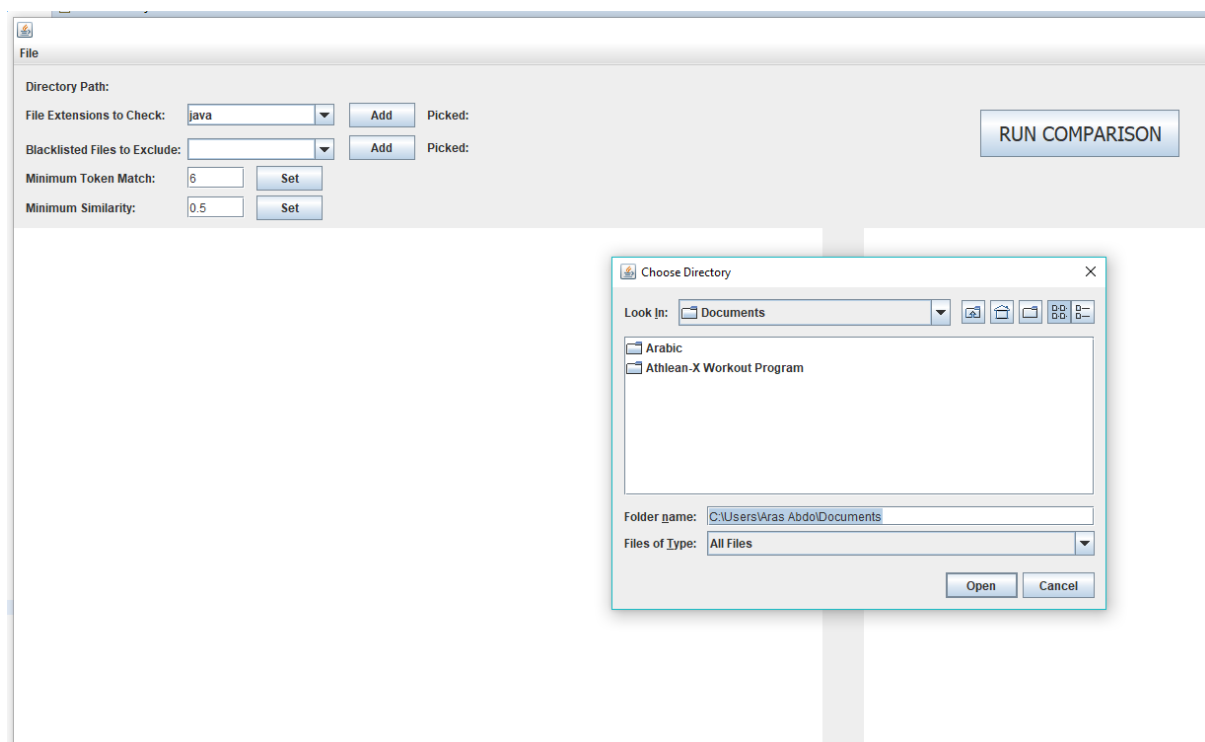


Figure 6.2: Directory Chooser

6.3. MultithreadingWorker Class – Methods Explanation

This class is designed to be executed as a thread, this is achieved by implementing the Runnable interface. Several global variables are defined, this first is a Thread object, the second is a List of Job objects, each Job object contains the necessary information to perform a comparison.

1. Job Object

This includes two String variables that hold the contents of the two programs to compare. Two Boolean values; one that declares whether a job has started, the other states if the final similarity result has been deemed as plagiarised, and so the Job object also houses the similarity value. Finally, an ArrayList of TileValuesMatch objects are declared, this variable stores all of the found matches during the RKR-GST execution.

The final global variable is the Semaphore mutex; it is used so that only one thread can enter the area. Once it has acquired the key(lock), it can proceed with the work and others must wait until the lock is released.

2. `public void run() {`

This method is the entry point for the thread, it contains all of the logic that is required to run a comparison by calling the appropriate methods. While there are Job objects, each Job must acquire the lock to proceed to their work, this is declared by setting the Boolean hasStarted value to true. Now, the current Job in the list will have both of its String contents Normalised before the RKR-GST algorithm is called and the comparison is made. Several additional values are calculated and set to the variable in the Job object. This process is repeated for every Job in the list.

6.4. Running Karp-Rabin_Greedy String Tiling Class – Methods Explanation

This class fulfils the Non-Function Requirement “FR05”,

The algorithm in this class makes multiple passes, where each of these passes has two phases. The first phase involves executing the `scanpattern()` method, in this method we collect all of the maximal-matches that are a specific size or greater. The second phases, involves executing the `markString` method, which is located in the top-level algorithm method called `run()`. This class has one global integer variable called `initialSearchSize`, the initial value of this variable should be small, because it is very rare to have very long maximum-matches, thus having an initial value which is large is inappropriate, this is because it will generate several empty sweeps of the `scanpattern` method, until a match can finally be found. The value has been set to 20, which should accommodate this issue.

1. `public static ArrayList<TileValuesMatch> run(String[] patternList, String[] textList, int minimalMatchingLength) {`

The contents of the pattern and text programs are passed through this method as parameters, and the value of the `minimalMatchingLength`. The method initially checks the value of the `initialSearchSize` and `minimalMatchingLength`, setting them to the appropriate value if the condition is met. This is the top-level algorithm for Greedy String Tiling. The first value of the `searchLength` is equal to the `initialsearchlength`, which is set to 20 by default. While the Boolean value `stop` is not equal to false, the method calculates the `largestMaximalMatch` for the current scan, by calling the `scanpattern()` method. Depending on the value obtained from the `scanpattern()` call, we either move onto the second Phase of the algorithm (`markStrings`) or the `searchLength` value is reassigned a larger value. For very long strings, the tiles are not marked, instead we iterate again with a larger `SearchLength`, this will happen if the `largestMaxiamlMatch` value is twice greater than the value of the `searchLength`. It should be noted that only during the first iteration will the long strings be found. After the first iteration, no matches that are twice greater than the current search-length(s) will be found.

The second phase of the algorithm: `MarkString` occurs in this method, rather than placing this phase in its own method, I determine that it would be suitable to place it within the top-level algorithm because it is only being called once and that’s in this method. The Maximal-matches, starting with the longest one are each taken in turn and tested to determine if it has been occluded by any of their sibling tiles. i.e. if a section of the maximal-match has already been marked by another tile. If it is not, then by marking two substrings an entirely new tile is created. Finally, once we have dealt with all of the maximal-matches, a new smaller value is chosen for the `searchLength`. The `markString` sections works as follows: it creates tiles from matches that have been taken from a list of queues. For each of these queues, starting with the top queue, and while there is a non-empty queue do: retrieves and removes the head of this queue and sets it to the `TVMQueue`, thus the first head of the queue is set to the variable “match”, or it will return null if the queue is empty. If it is not Occluded then a specific index in the text and pattern list will be marked using the `MarkToken()` method. The match variable is then added to the `ArrayList` of `TileValueMatch` objects and returned. [8]

2. `private static int scanpattern(ArrayList<Queue<TileValuesMatch>> tVMATCHList, int s, String[] Pattern, String[] Text) {`

This is the first phase of the algorithm: scanpattern that uses Running Karp-Rabin matching, this method is used to discover every maximalMatch that is above the searchLength size or greater, all of the found matches are stored in a collection. The method scans through both the text and pattern string arrays for matches, and if it finds a match which is twice the size of the search length(s), then this size is returned so that the method scanpattern can be restarted with this new value. I have used a doubly linked list of queues ordered in decreasing order to store all of the maximal matches that have been discovered. Every queue in the list records maximal-matches that have the same length. It's important to note that a pointer has been defined that points to the newest maximal-match in the queue, this is because it is very likely that the next maximal-match that is discovered will be of similar length.

Firstly, the method deals with the Text string. While the line(t) in String(Text) is less than the length of the String(Text), we do the following. Check to see if the tile is marked, if it is then we increment to the next line in the string, so that the next unmarked tile can be the subject of the scan, the tile is deemed marked if the very first index of the line matches to the character “*”.

Now that we have determined that we are dealing with an unmarked token, we need to calculate the distance to the next tile, this is because we want to find out whether the value calculated is less than the searchLength. If it is then we check if there is no next unmarked token after the tile. This means that we are at the end of the Text string, thus we set the current line value to the length of the Text (e.g. last index). Otherwise we move onto the first unmarked token after the next tile. If the distanceToNextTile is greater than s. Then we create a Karp-Rabin hashvalue for the substring (T[t] to T[t + s - 1]). This value is saved and added to the HashTable, Finally the current line “t” is incremented and the next line in the string is dealt with. Similarly, for the Pattern String array we carry out the same set of code, although some additional code is used in this section.

3. `private static boolean unMarked(String str) {`

This method is used to check whether a specific index in a string has been marked. It returns the Boolean value True if the String(str) at a given index is greater than 0 and also does not contain the character “*” at index 0. This will state that it has not been marked. Otherwise if it does contain “*” then the Boolean value false is returned. It is possible to determine whether a token is marked by using !unMarked.

4. `private static boolean isOccluded(ArrayList<TileValuesMatch> tiles, TileValuesMatch match) {`

This method tests to see if a match has been marked during the creation of another tile, and returns True if it has been occluded. Rather than testing the whole maximal match for occlusion, we only test the ends of each tile, because larger tiles are created before shorter ones, thus testing the ends of the tiles will suffice to detect for occlusion.

5. `private static String markToken(String str) {`

Marks the token by appending the character “*” to the very first index of the line. This can later be used to determine whether a token has been marked, by testing if this character is contained in the very first position.

6. `private static int KRHashValue(String subString) {`

This method creates a Karp-Rabin hash value for a substring that have been passed into the method as a parameter and returns this value.

The implementation of this class was very difficult, even with the provided pseudo code, it took several iterations to correctly code this class using the agile methodology. I split the several methods up implementing the smaller methods first which will be used by the larger ones. This enabled me to insure each method was complete and without errors, before moving on to the next one.

6.4.1 HashTable Class – Methods Explanation

This is the class that will be used by the RKR-GST algorithm, once called a new HashMap is created, it contains two methods and a constructor. The HashTable has two global variables, the first is a HashMap, that assigns each key with an ArrayList of integers. The section variable is an ArrayList of integers.

1. `public void add(long key, int value) {`

This method adds a new value to the HashMap. An if statement is used to see if the HashMap does not contain the key that has been passed through the method, if it's not contained, that means there is no ArrayList, thus a new ArrayList is created, the value is then added to the new ArrayList and finally that list is assigned to the key. Else, the key is contained in the HashMap, and so we get the ArrayList that is already at the key, and add the new value to the ArrayList. Finally put the new adjusted ArrayList back into the key.

2. `public ArrayList<Integer> get(long key) {`

This method is designed to return the ArrayList stored in the specified key location. The method uses an if statement is used to determine whether the HashMap does not contain the key that has been passed through the method, if so the method returns null, otherwise the ArrayList designated to the key is return, which is an ArrayList of integers.

6.5. SimilarityCalculator Class – Methods Explanation

Based on the chosen formula identified in section 5.5.5 I have created a class that computes the similarity between two source-code files. First an ArrayList of TileValuesMatch objects are passed into the sumOfAllTileLengths method, the value returned is then used within the similarity method and finally, the calculateSimilarity method utilises the results.

1. `private static int sumOfAllTileLengths(ArrayList<TileValuesMatch> tiles) {`

This method calculates the sum of all the tile lengths and returns this value. For loop is used to iterate over each tile and summing up the lengths in turn.

2. `private static float similarity(ArrayList<TileValuesMatch> tiles, List<String> stringOneList, List<String> stringTwoList) {`

Method returns the similarity value for string one and two's token and the similar tiles. $2 * \text{Sum}(\text{length of each tile}) / (\text{length of string one} + \text{length of string two})$

3. `public static SimilarityValue calculateSimilarity(List<String> stringOneList, List<String> stringTwoList, ArrayList<TileValuesMatch> tiles, float threshold) {`

The result of the similarity method is checked to determine whether it is greater than the threshold value, and returns true if it is. Finally, a new SimilarityValue object is returned, the class stores the similarity value and a Boolean variable that states whether the comparison has detected plagiarism, if it is set to true.

6.6. Normaliser Class – Methods Explanation | Issues

This class can be adjusted to handle any programming language, currently this is designed for java source code. Change it for your language by adapting the global variables to suit your language. Or add several languages and test to see what language the source code is and set normaliser to the String with that parameter.

The class contains multiple global variables; these are defined to represent the programming language java. For example; the “keywords” array is used to identify all of the java specific inbuilt words. This is useful because it allows the class to extra information that is valuable in describing the essence of a program, and all keywords that have been identified by the system but are not included in the final token sequence are matched against the array and discarded, those that do match will be included in the sequence.

1. `public static void normaliser(String content, boolean returnLinesForEachToken) {`

This is the central method of the whole class, the methods that I have listed below are all called here, and they carry out their tasks in turn. Several global methods are initialised here, rather than having a constructor. An important variable that is defined in this method is the lineNumberList, which is an array of integers; where each index represents a line in the String. For example: line 4 in the program String is the 3rd index in lineNumberList, this is important because it keeps track of the String layout, at each stage of the transformation, making sure that it has been correctly adapted.

2. `private static String preprocess(String content, ArrayList<Integer> lineNumberList) {`

This is the first method that is called, here we search through the content string, for method and class definition that have been split over multiple lines, and combine them back into one single line, this is an important step that is required to insure that the later stages can function correctly.

The content String is initially split into a list, where each entry in the list is a line. While there are still lines in the list we identify class and function definitions that exist on that line using the appropriate regular expression. While there are still found matches, we count the number of parenthesis that still remain open on that line. We identify the line at which there exists no more open parenthesis, e.g. when they have been closed (). This states the number of lines the definition is split over, based on the values we extract the information and put the definition onto one line. The lineNumberList is modified appropriated to represent the new changes made to the content.

3. `private static int countParenthesis(String line) {`

Line in the string is checked for the number of times the parentheses occur, loops through each character in the line and checks if it matches '(' and increments the counter by 1, every close decrements it by one.

4. `private static String removeComments(String content, ArrayList<Integer> lineNumberList) {`
`private static String removeStringConstants(String normalisedString,`
`ArrayList<Integer> lineNumberList) {`

These two methods work similar to each other, in that they both pass the appropriate variables into the `removePatternMatchesFromString` method below, the passed parameters represent the pattern that needs to be matched in the content. However, the `removeComments` method, first needs to be converted into an array so that the method below can deal with it.

5. `private static String removePatternMatchesFromString(String content, String[] patterns, ArrayList<Integer> lineNumberList) {`

This method will remove the given pattern that has been passed through as a parameter, from the String contents and it will also adjust the `lineNumberList` accordingly.

While True, we loop through each pattern in the array and match the given pattern at the particular index with any occurrences found in the content. We are using the `Pattern` class to store the given pattern as this will allow us to simplistically utilise the pattern within the `Matcher` class, that automatically finds all occurrences of the pattern in the string. While a match has been found we store the beginning and end of the pattern location, and also the contents of the pattern that has been identified. Then we count the number of lines the pattern is split over and store this value. If the pattern is split over more than 1 line, then we count the number of lines from the beginning of the String content all the way to the start of the pattern match found and store this value. Similarly, we used the value that stored the number of lines the pattern is split over to identify the end of the pattern String. This enables us to remove the pattern from the string based on its identified start and end positions within the content String. Now all occurrences of the pattern in the string have been removed, however there still remains empty lines, where the pattern used to be.

6. `private static String mapSynonyms(String normalisedString) {`

This method is simpler than the previous ones, it maps the synonyms to more common forms, by looping through the `HashMap`, finding any and all occurrence of the key in the string and replaces it with the value set at that key location.

6.8. Summary

This chapter has described the important methods within the main application classes and has out stated when and where a non-functional requirement was met.

7. Testing & Evaluation

In the agile software development model the testing phases is carried out though each iteration of the implementation, here in this chapter we will test and identify the final system for issues and detail why they exist and how it can be fixed in the future. We also note why some issues that were identified during the implementation phases were not fixed immediately.

7.1. Graphical User Interface

The graphical user interface is checked for errors and to determine whether it offers a comfortable user experience, that is both fast, fluid and without issues.

1. Usability

Testing the usability of the GUI is critical, this is the only area of interaction between the user and the application that needs to be very fluid, fast and comfortable. Figure 6.1 and 6.2 presents the initial GUI for the application, the usability was tested through 30 interactions, where each interaction required new user input in to the comboBox, textArea and the directory chooser, what I found after continuously running the comparison was that it became tiresome, and I began to get irritated using the application, because for each input I was required to first input the required values, but then also click the appropriate buttons that would set the input into the system, this meant that 6 buttons would need to be clicked before the comparison results are displayed.

Another issue that relates to the usability is the layout of the components. Firstly, in order to input the base directory path, the user would have to use the drop down menu, which can be seen in Figure 7.1. This was an unnecessary step; the menu should have instead lead to another window that would be used to input all the information to run the program. This menu bar needs to be removed and the directory chooser needs to be implemented the same way as the other inputs.

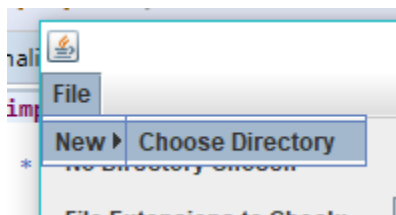


Figure 7.1: Menu to Directory Chooser

The overall layout of the user inputs is limiting, the efficiency is diminished greatly, the presentation of the inputs is not appealing and honestly looked unprofessional and rushed. I found that this was an important issue that needed to be changed, I went back and designed a GUI that would fix the issues I found. The system now starts up with a window that presents all of the information that needs to be input by the user. In Figure 7.2, we can see the new design for the user interface, the window now only has 2 buttons before the similarity result is

shown. The directory chooser button was not integrated into the Run Comparison button because that would mean the first selected path would need to be the correct one. However, the possibility of a user clicking on the wrong directory is a high, thus the option to re-click the directory chooser is important and was implemented to accommodate for this problem.

Once the user clicks the “Run Comparison” button, another window will popup showing all of the necessary information about the comparison. Figure 7.3 shows the second window, note that two new elements have been included, that were not used in the first iteration of the GUI. The similarity result and the suspected Boolean value are defined below the HTML visualizer. The core functionality of the “Frame” class has not been changed, several methods and variables were defined to create the new layout for the application. It’s important to note that there will be no further changes to the programs, whether improvements or error fixes, because my deadline is approaching and I have consumed all the allocated time available for implementation. In Figure 7.4 the final UML Class Diagram is shown.

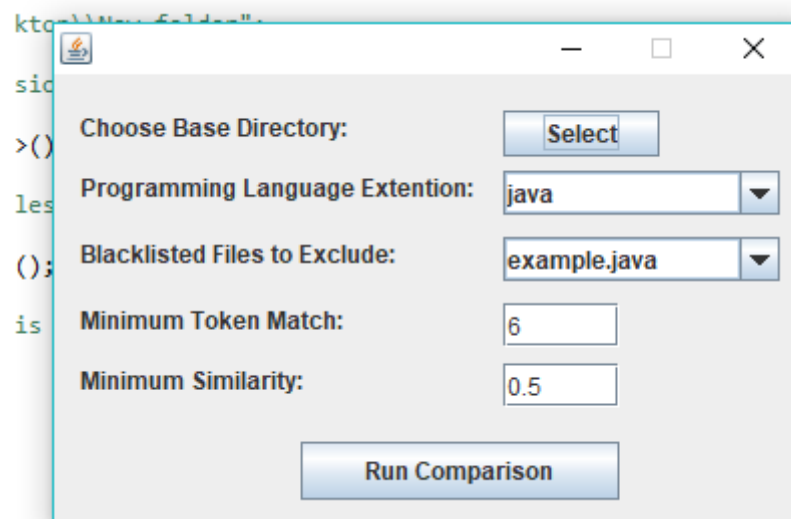


Figure 7.2: Load and Set Copy-Detector Details

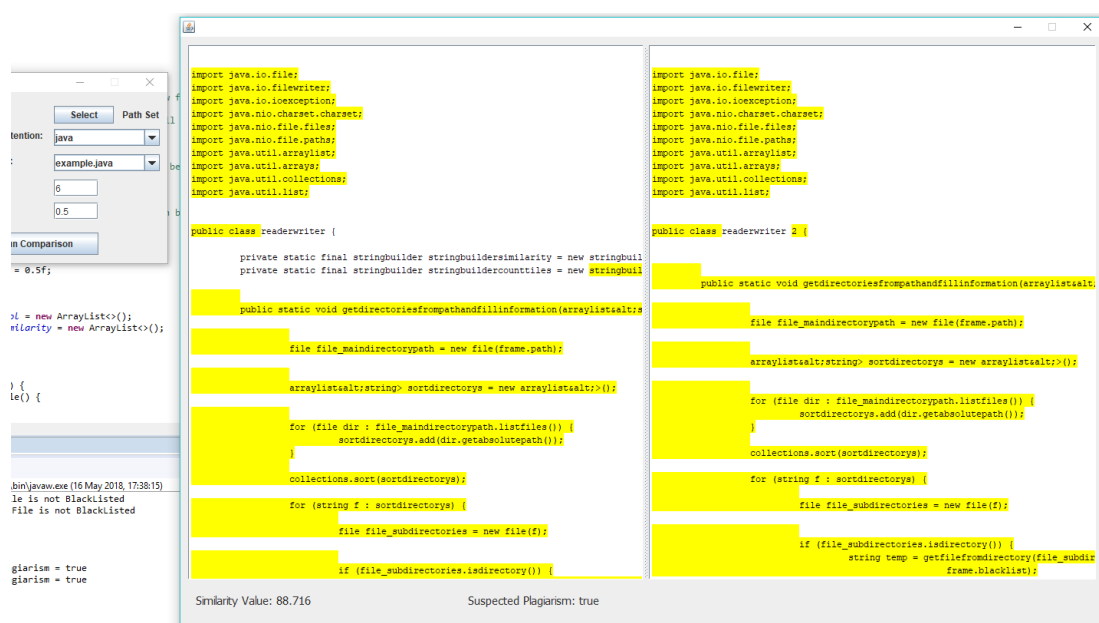


Figure 7.3: Pair of Programs Visualisation

- Final UML Class Diagram

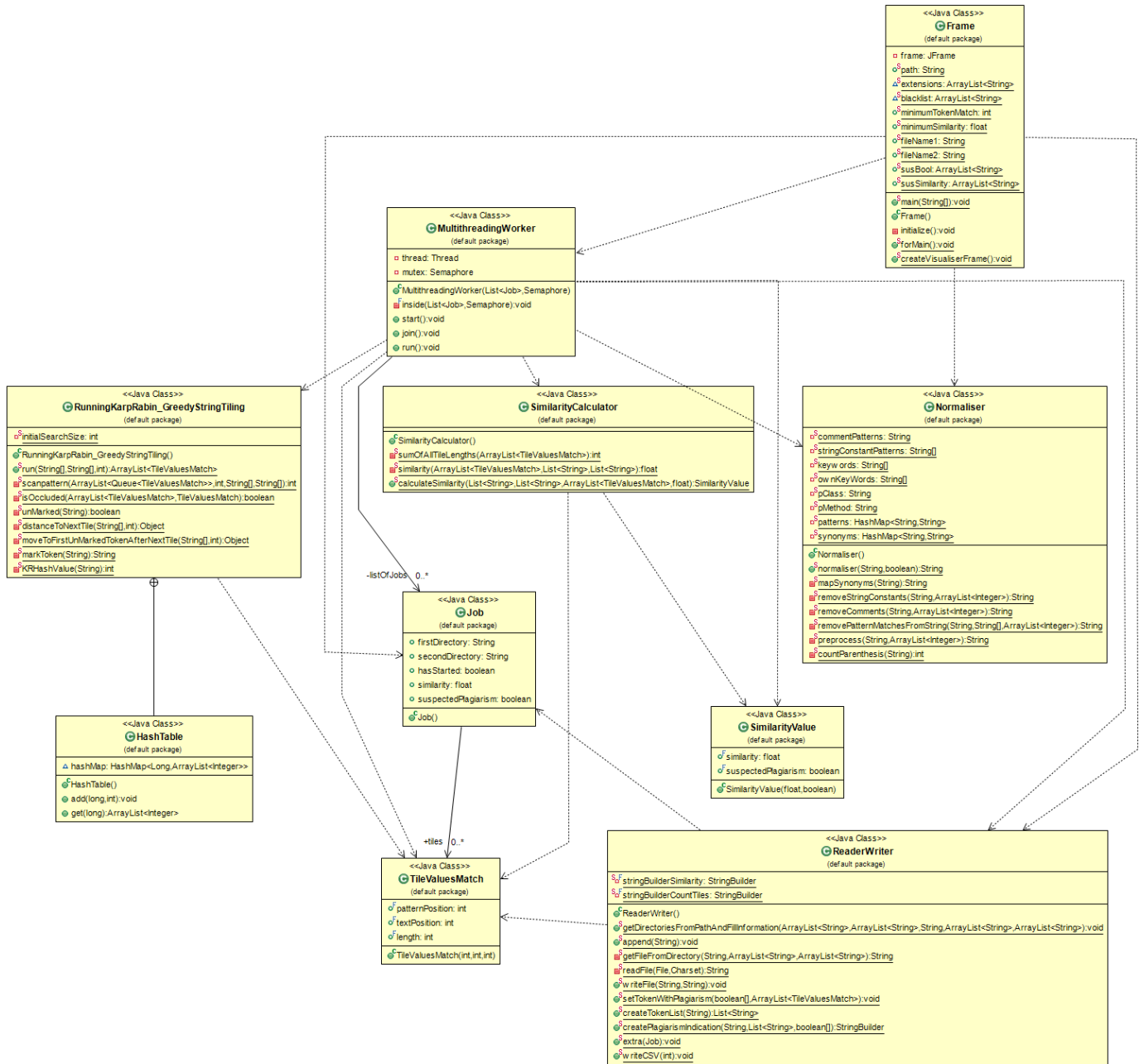


Figure 7.4: Final UML Class Diagram

2. Input Components

In this section the new graphical user interface is checked for errors that can arise from the user input components. Multiple different values are inserted into the input components to check for any issues that may arise.

- Base Directory Chooser

Initially the base directory component is checked; this works correctly and upon selection of the directory the path is stored, the Chooser does not show any of the files in the folders, this insure that the user cannot click on a file rather than the directory that stores the files. In Figure 7.5 we can see the directory for a Workout program folder, this folder contains many files, such as images, PDF's, Microsoft Word Documents and more. The directory chooser ignores all of the files and only presents the other folders within the directory.

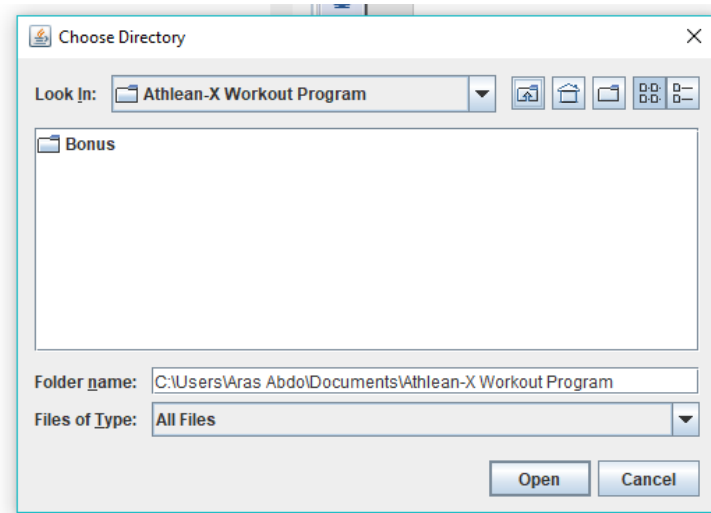


Figure 7.5: Directory Chooser Test

Next I tested the “set” component that informs the user as about the if the base directory path has been set into the system. From the Figure 7.5, I pressed the cancel button and the correct response was displayed on the GUI, that states the system has not set the path, as seen in Figure 7.6.

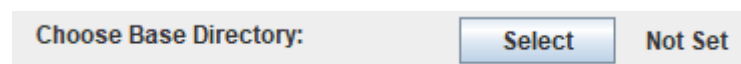


Figure 7.6: Directory Not Set

The process was repeated, but in this test the button “Open” was clicked after I had navigated into the correct directory folder. This can be seen in Figure 7.7, the “Path Set” is shown and the user can now complete the rest of the inputs.

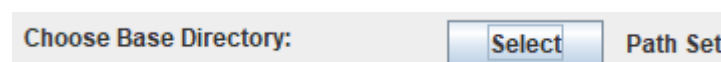


Figure 7.7: Directory Path Set

This input should save the selected directories absolute path into the global path variable. This is tested by using a simple print statement on the “path” variable after the code statement. In Figure 7.8 we can see that the path has been successfully stored, the absolute path as seen in the bottom left corner is in the correct format and points at the correct directory.

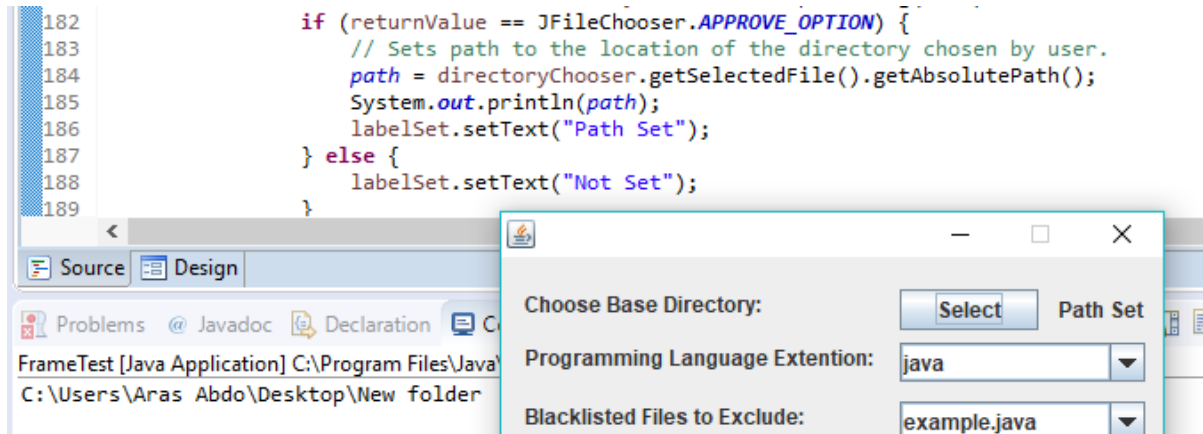


Figure 7.8: Directory Path Successfully Stored

- Programming Language Extension & Blacklisted Files

The JComboBox for the language extensions by default has 4 values allocated, the drop down option is tested to see if the all the values are correctly presented, as seen in Figure 7.9. The dropdown values can be selected and edited into other values, once the application is run the extension is stored into the system. However, an issue that I have discovered is that the system does not check to see if the inputted value is a real programming language extension, this can result in problems later on in the application process. The inputted value should be tested against an array of programming languages that exist. More importantly, the system doesn't check to see if the value inserted is only “Java”, this is important because the Normaliser only handles Java programs currently.

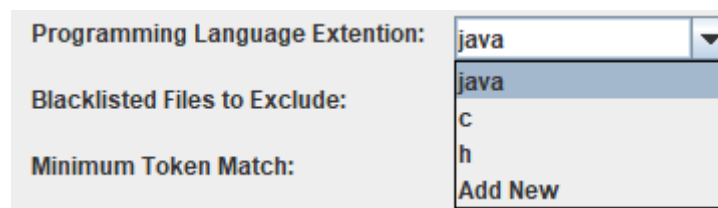


Figure 7.9: Language Extension JComboBox Drop-Down

Similarly, the blacklisted files JComboBox is tested, the drop down option correctly presents the two default values which illustrate to the user how the value should be written. The values are stored within the system upon button click. I have identified an issue that is a result of the changes I made to the GUI, the user should be allowed to select multiple files to exclude, this worked correctly in the first design. However, the new GUI does not accommodate for this feature. The “Set” button has been replaced with the “Run Comparison”, once clicked the similarity detection begins, this means that the user is currently only able to exclude one file from the comparison.

- Minimum Similarity & Token Match

The minimum token match field should only accept integers that are greater than 1 and less than 15, this is implemented correctly and the input field also ignores any character that is not a “Backspace” or an integer.

One issue that I was unable to deal with was with the user allowed input in the JTextField for minimumSimilarity, the problem was limiting the amount of dots (.) the user was allowed to input, currently the field only accepts integers and dots, but the dots are not limited to 1, thus it is possible to cause an error with the input. This can be dealt with in a future patch of the system, currently multiple dots will not be detected by the system.

- Run Comparison Button

The run comparison button is designed to deal with all of the input fields, apart from the directory chooser, the button checks each field to insure that they are not empty and meet a set of requirements. For example, the similarity value input is tested to insure that it is greater than 0 and less than 1, based on these conditions the values are either saved into the system and the comparison is run or an appropriate dialog box is output on to the user’s screen, indicating the issue that occurred. The button works correctly and opens up the second window, for the results of the comparison.

```

} else if ((textFieldMinimumTokenMatch.getText().isEmpty()
|| Integer.parseInt(textFieldMinimumTokenMatch.getText()) < 1
|| Integer.parseInt(textFieldMinimumTokenMatch.getText()) > 15)) {
    JOptionPane.showMessageDialog(null,
        "Please insure you have entered a value greater than 1 and less than 15.");
} else if ((textFieldMinimumSimilarity.getText().isEmpty()
|| Float.parseFloat(textFieldMinimumSimilarity.getText()) < 0
|| Float.parseFloat(textFieldMinimumSimilarity.getText()) > 1)) {
    JOptionPane.showMessageDialog(null,
        "Please insure you have entered a value less than 1 and greater than 0.");
} else if (path == null) {
    JOptionPane.showMessageDialog(null, "Please select the base directory.");
} else {
    extensions.clear();
    blacklist.clear();

    extensions.add(allowedExtentionPicked);
    blacklist.add(excludedExtentionPicked);
}

```

Figure 7.10: Run Comparison Validator

3. Visualizer Window

The visualizer window has been tested to insure that the created HTML files are parsed back into the system and displayed in each of the JEditorPane windows, the allocation of space for the windows is identical, neither window takes more space than the other, and the scrollable component is usable, navigating the user to lower/higher parts of the program. The results of the comparison are correctly output onto the window, and into the two JTextFields, I ran two comparisons, the first had a pair of programs that should be identified as plagiarised by the system and the second comparison was two original programs, I tested to see if the textField values would be changed to correctly identify the result. In each case the correct similarity and Boolean value were output onto the screen.

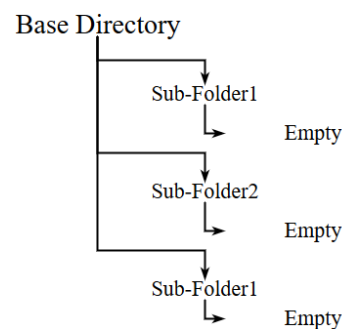
7.2. File Reader | Writer

This class is tested to insure that it is functioning correctly, the files are being read into the system without any errors and the system identifies and stores files that have the allowed extension, discarding any files that are blacklisted. Also, the HTML and CSV files are checked to insure they have been created correctly. Below we have listed several issues that have been identified during the testing of the application.

1. Empty Directory

I tested the system to see if errors would arise when the base directory is set to a location that does not contain any programs, this immediately cased an `OutOfBoundsException`, because as the application scans through the base directory and each of the sub directories; it searches for files that match the allowed extension, and if there are no programs that have that extension this error will occur when we loop through each of the indexes that hold the files, this can be fixed quickly in a future iteration of the application.

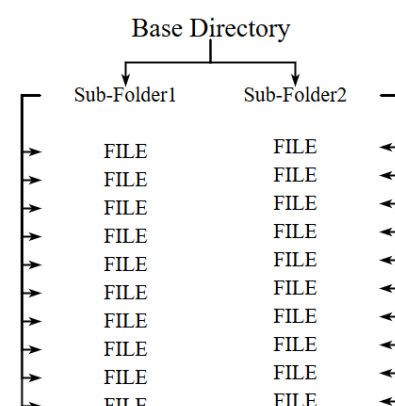
Figure 7.11: Empty Directory



2. Multiple Files in Each Sub-Folder

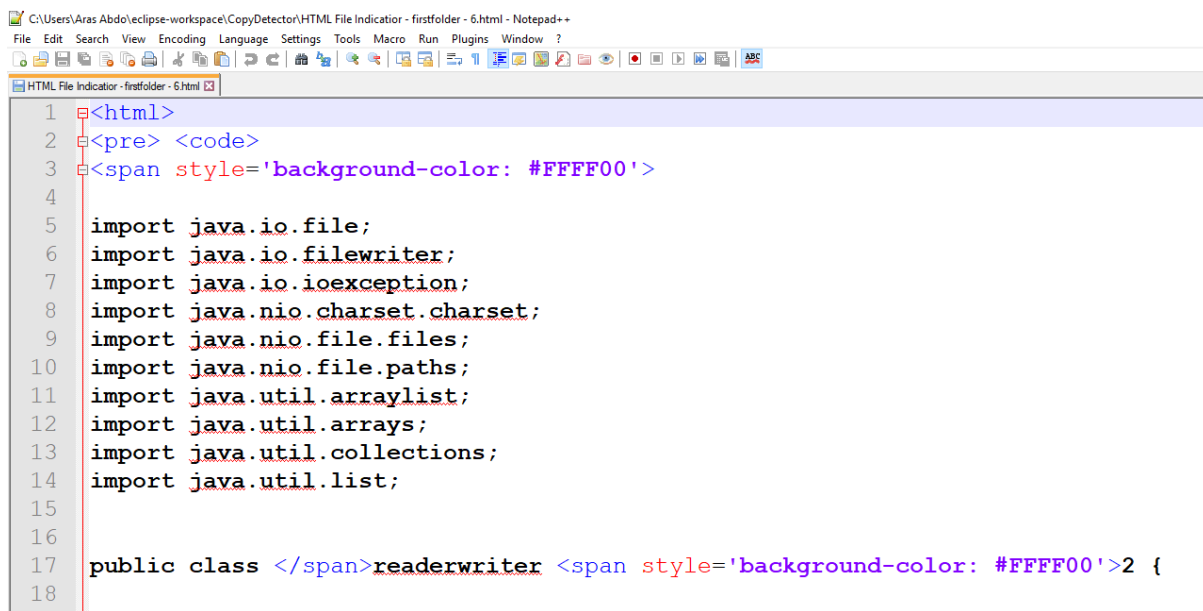
I then tested the application using a file directory that housed more than 10 programs in each of the two sub folder as seen in Figure 7.12. The implementation of the `ReaderWriter` class was designed to only accommodate the storing of one file from each folder, this means that if multiple files are read into the system they will all be combined into one stringbuilder. E.g. the contents of very program file in the folder will all be combined and stored in the same `String`. This is not necessarily an issue as the intent of this class was to deal with a pair of programs rather than multiple programs, it was designed to be extendable, and thus this is actually a positive response from the system, it has scanned the subfolders and parsed every file. However, there should be some validation upon selection of the base directory that indicates to the user that each folder must only contain one file, until the system is extended.

Figure 7.12: Ten Files/Sub-Folder



3. HTML & CSV Writer

I examine the files that have been created by the system upon completion of the comparison, the tokens that have been matched are highlighted correct and cover the correct begging and end of each match. The CSV file has also been successfully created, there is slight issue with the size of the tabs in the document, this is superficial and doesn't not degrade the contents. Figure 7.13 shows that the html tags have been used in the correct places, we can also see that after "readerwriter" the span tag is used to highlight the code is within it, such as the "2 {".



```
1 <html>
2 <pre> <code>
3 <span style='background-color: #FFFF00'>
4
5 import java.io.file;
6 import java.io.filewriter;
7 import java.io.ioexception;
8 import java.nio.charset.charset;
9 import java.nio.file.files;
10 import java.nio.file.paths;
11 import java.util.arraylist;
12 import java.util.arrays;
13 import java.util.collections;
14 import java.util.list;
15
16
17 public class </span>readerwriter <span style='background-color: #FFFF00'>2 {
18
```

Figure 7.13: HTML File Writer

7.3. Normaliser

The normaliser class is tested to insure that the correct transformation is made at each step, when each method is called a series of changes should be made to the original string, here we test to insure that this is the case by outputting several print statements after the code block has been performed. Figure 7.14 contains the code that will be passed through the normaliser class, at each stage of the transformation the new changed code will be presented.

```
1 private String preprocess(String content,
2 ArrayList<Integer> lineNumberList) {
3     // Split String content by lines, and set each line as entry in array.
4     String lines[] = content.split("\\r?\\n");
5     int lineNumber = 0;
6
7     while (lineNumber < lines.length) {
8         if (Pattern.matches(pClass, lines[lineNumber]) || Pattern.matches(pMethod, lines[lineNumber])) {
9             int count = 0;
10            String s = lines[lineNumber];
11            int openP = countParenthesis(lines[lineNumber]);
12
13            while (openP != 0) {
14                count += 1;
15                openP += countParenthesis(lines[lineNumber + count]);
16                s += " " + lines[lineNumber + count];
17                // remove the newly empty lines Strings by setting them to ""
18                lines[lineNumber + count] = "";
19            }
20            lines[lineNumber] = s + "/n"; // CHECK IF YOU NEED /N?????????????
21            // also have to remove the empty lines ints from the numberList
22            lineNumberList.subList(lineNumber + 1, lineNumber + count + 1).clear();
23            lineNumber += count + 1;
24            continue;
25        }
26        lineNumber += 1;
27    }
28    return "" + lines;
29 }
30 /* asdasdasda */
```

Figure 7.14: Normaliser Original Code

- Pre-processing the String

The definition of the method in the original code has been split over two lines, after the code has been pre-processed; the definition has been combined and put into one line. In Figure 7.15; on line 1 we can see that this method has been successfully executed.

```
1 private String preprocess(String content, ArrayList<Integer> lineNumberList) {
2
3     // Split String content by lines, and set each line as entry in array.
4     String lines[] = content.split("\\r?\\n");
5     int lineNumber = 0;
6
7     while (lineNumber < lines.length) {
8         if (Pattern.matches(pClass, lines[lineNumber]) || Pattern.matches(pMethod, lines[lineNumber])) {
9             int count = 0;
10            String s = lines[lineNumber];
11            int openP = countParenthesis(lines[lineNumber]);
12
13            while (openP != 0) {
14                count += 1;
15                openP += countParenthesis(lines[lineNumber + count]);
16                s += " " + lines[lineNumber + count];
17                // remove the newly empty lines Strings by setting them to ""
18                lines[lineNumber + count] = "";
19            }
20            lines[lineNumber] = s + "/n"; // CHECK IF YOU NEED /N?????????????
21            // also have to remove the empty lines ints from the numberList
22            lineNumberList.subList(lineNumber + 1, lineNumber + count + 1).clear();
23            lineNumber += count + 1;
24            continue;
25        }
26        lineNumber += 1;
27    }
28    return "" + lines;
29 }
30 /* asdasdasda */
```

Figure 7.15: Code Pre-Processed

- Removing all Comments

The original code includes several code comments, the single line and multiline comment definition and I have also included a tailing comment on line 20 in Figure 7.14. This will allow me to test if all comments are being targeted by the program, and the detection is not affected by code that preceded the comment. After running the remove comment method, we have the following output, shown in Figure 7.16. Both the single and multi-line comments have been successfully targeted by the system and then removed. On line 20 we can see that the method has correctly identified the start and end of the comment definition and removed it, without negatively affecting the code it was tailing.

```

1 private String preprocess(String content, ArrayList<Integer> lineNumberList) {
2
3
4     String lines[] = content.split("\\r?\\n");
5     int lineNumber = 0;
6
7     while (lineNumber < lines.length) {
8         if (Pattern.matches(pClass, lines[lineNumber]) || Pattern.matches(pMethod, lines[lineNumber])) {
9             int count = 0;
10            String s = lines[lineNumber];
11            int openP = countParenthesis(lines[lineNumber]);
12
13            while (openP != 0) {
14                count += 1;
15                openP += countParenthesis(lines[lineNumber + count]);
16                s += " " + lines[lineNumber + count];
17
18                lines[lineNumber + count] = "";
19            }
20            lines[lineNumber] = s + "/n";
21
22            lineNumberList.subList(lineNumber + 1, lineNumber + count + 1).clear();
23            lineNumber += count + 1;
24            continue;
25        }
26        lineNumber += 1;
27    }
28    return "" + lines;
29 }
30

```

Figure 7.16: Remove Comments from Code

- Remove String Constants

This step should remove both “ and ” instances that are contained in the code, and the string/character that are held within them. There exist 5 instances of comments and we can see that they have all been targeted and removed by the system in Figure 7.17.

```

1 private String preprocess(String content, ArrayList<Integer> lineNumberList) {
2
3
4     String lines[] = content.split();
5     int lineNumber = 0;
6
7     while (lineNumber < lines.length) {
8         if (Pattern.matches(pClass, lines[lineNumber]) || Pattern.matches(pMethod, lines[lineNumber])) {
9             int count = 0;
10            String s = lines[lineNumber];
11            int openP = countParenthesis(lines[lineNumber]);
12
13            while (openP != 0) {
14                count += 1;
15                openP += countParenthesis(lines[lineNumber + count]);
16                s += + lines[lineNumber + count];
17
18                lines[lineNumber + count] = ;
19            }
20            lines[lineNumber] = s + ;
21
22            lineNumberList.subList(lineNumber + 1, lineNumber + count + 1).clear();
23            lineNumber += count + 1;
24            continue;
25        }
26        lineNumber += 1;
27    }
28    return + lines;
29 }
30

```

Figure 7.17: Remove String Constants

- Transform to Lower Case and Map the Synonyms

Here we will be targeting both the uppercase letters and mapping the synonyms to more common forms. In Figure 7.18 the code that been transform completely to lowercase letters and all the synonyms have been successfully mapped to their appropriate value, as can be seen in line 26, the “+=” has been replaced with “=”.

```

1 private string preprocess(string content, arraylist<integer> linenumberlist) {
2
3
4     string lines[] = content.split();
5     int linenumber = 0;
6
7     while (linenumber < lines.length) {
8         if (pattern.matches(pclass, lines[linenumber]) || pattern.matches(pmethod, lines[linenumber])) {
9             int count = 0;
10            string s = lines[linenumber];
11            int openp = countparenthesis(lines[linenumber]);
12
13            while (openp != 0) {
14                count = 1;
15                openp = countparenthesis(lines[linenumber + count]);
16                s = + lines[linenumber + count];
17
18                lines[linenumber + count] = ;
19            }
20            lines[linenumber] = s + ;
21
22            linenumberlist.sublist(linenumber + 1, linenumber + count + 1).clear();
23            linenumber = count + 1;
24            continue;
25        }
26        linenumber = 1;
27    }
28    return + lines;
29 }
30

```

Figure 7.18: To Lower Case and Map Synonyms

7.3. Plagiarism Detection Experiments [RKR-GST]

In this section we detail the experiments carried out on our implemented copy-detection tool, by testing the RKR-GST algorithm on a set of 50 different test files.

50 Java test files will be used to test my copy-detection tool, these files have been taken from the JPlag example matches page [16], all the files use the standard Java library, and have an average of 5000 non-whitespace characters, this insure that they can all be read by my system as it only accepts files containing above 1000 characters. JPlag is a copy-detection tool that uses the Greedy String Tiling algorithm with its own optimizations for better efficiency, while my system also uses the Greedy String Tiling algorithm and Running Karp-Rabin matching for better efficiency, this is the main difference between the two tools; the optimisation. JPlag is a widely used within academic institution for the detection of plagiarism, therefore I have decided to take advantage of its free accessibility, and used the provided text samples on their website.

A comparison will be made for similarity results against those produced by JPlag, which will allow me to determine if my system works correctly, this will be seen though a similarity result that is within a similar range of JPlag's result. Furthermore, I will manually inspect the HTML files created for each comparison, and compare those to the ones shown on JPlag's site. Each HTML file will be examined to determine if any tokens were missed and not matched during the similarity matching phase.

I would like to note that the Source for each of these test files is added as a Java comment within the code. We will carry out a total of 25 comparisons using the test files, this will produce a total of 50 HTML files and 50 CSV files.

Test ID	File Name	File Name	Best Token Match Value (Sensitivity)	My Similarity Result	JPlag Similarity Result	Issues?
1	826764	826366	6	100.0%	100.0%	
2	943151	942261	3	69.2%	70.4%	
3	943151	862531	4	30.7%	21.1%	
4	878135	862531	5	63.4%	57.7%	
5	861301	861196	3	39.4%	40.1%	
6	861301	862246	3	35.3%	35.9%	
7	861301	827825	6	15.4%	20.4%	
8	861005	861641	15	158.0%	39.8%	YES
9	861005	861061	15	159.4%	25.0%	YES
10	861005	827654	4	18.7%	20.2%	
11	862564	862326	4	36.1%	35.9%	
12	862564	942909	5	18.5%	20.7%	
13	862564	862246	5	20.7%	20.3%	
14	829683	826833	6	26.1%	30.6%	
15	861061	827654	3	24.5%	25.1%	
16	861061	862246	6	23.2%	20.8%	
17	861732	870711	6	19.7%	24.4%	
18	861732	745586	4	22.4%	21.1%	

19	862246	729057	4	14.8%	23.6%
20	862246	861196	3	26.5%	22.6%
21	862246	827825	6	19.0%	22.5%
22	862246	792145	3	27.1%	20.1%
23	861130	791299	8	21.4%	21.1%
24	861196	827825	3	23.5%	20.9%
25	720061	705604	3	21.3%	20.3%

Looking at the table above we can see that the output from my copy-detection system is very similar to that produced by JPlag, most results are with a 7% range from each other. Each test has been run multiple times, each time a different minimum match value is used to find the comparison where the sensitivity of the system is best able to identify the plagiarism between the two programs, and for each of these changes I have examined the produced html file and looked for tokens that should have been highlighted and when tokens that should have been matched were. Both systems are unable to completely detect every token that exists in both files, and there are occasions when tokens are matched when they shouldn't have. This means that future improvements are required to both systems, mine and JPlag's. However, the comparisons made, correctly highlight 90% of the tokens that are plagiarised, this means that it is important the examiner makes the final decision on whether the programs have been plagiarised. There are two cases where my system was unable to correctly produce a similarity result, Test's 8 and 9, after changing the sensitivity multiple times the output remained above 100%, I am unable to determine why this issue has raised, further analysis is required to determine the issue, this could be a task for the future.

The parameters that were chosen for each of the tests were different, because each program had its own structure and size. Therefore, using a minimal match length that is greater than 4, for a plagiarised program that consists of less than 4 statements from the original program; would cause an inaccurate detection of the plagiarism.

- Statement and Blocks Reordering

The code statements and blocks have been reordered in in 20 test files. Variable declarations and entire code blocks have been placed in entirely different positions, in order to determine if the system can successfully detect these plagiarism attacks. One example of the results obtained by this experiment is in Figure 7.19; two method blocks have been moved to the top of the class, and the system was able to 100% successfully match the plagiarism with the contents in the "original" program.

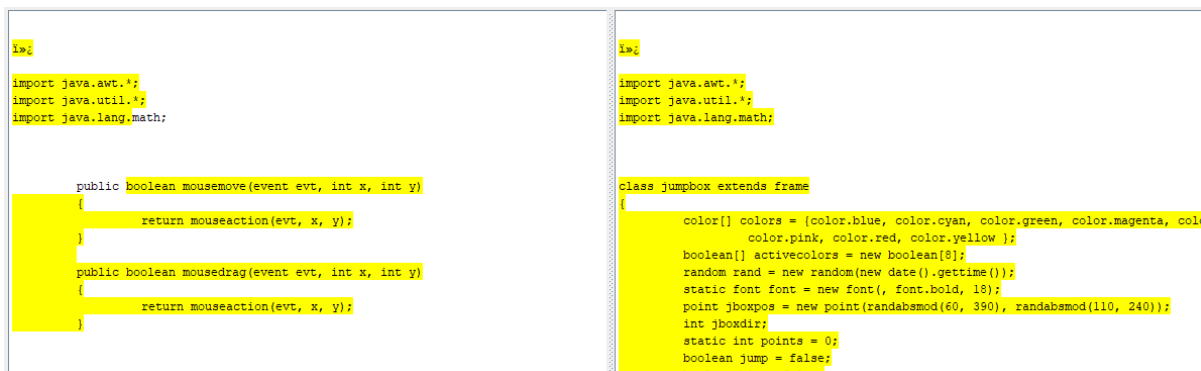


Figure 7.19: Snippet of Modified Test 1

- Performance

The copy-detection tool is able to complete a comparison on a pair of programs, almost instantly from when the “Run Comparison” button is clicked, offering a fast and accurate response for the client’s needs, this meets the non-function requirements “NFR06” and “NFR02”. Ultimately, the results in the table suggest that my copy-detection tool has been created to a high standard as the similarity results obtained lay within a similar range of those produced by JPlag’s system.

7.4. Summary

The system has been tested and all the issues identified, we moved onto experimenting with the copy-detection algorithm with 50 Java classes that ran in our system, the results were then compared with JPlag’s and we found that our produced similarities is similar to that produced by JPlag’s, this shows that the application developed and implemented in this project produces believable results, thus it is deemed a success.

8. How to Run/Use Application

This chapter details the steps needed to be undertaken in order to run the application successfully and explains what will cause errors.

1. Folder Structure

You must first create a base directory folder, you can call this folder “baseDirectory” or any other name. Inside the base directory folder, you need to create two new additional folders, call them firstFolder, secondFolder. In each of these sub-folders you must place one program file, as illustrated in Figure 8.1. Make sure the file you place in these folders has the Java programming language extension. E.g. “.java”. The system will not check files that contain less than 1000 characters.

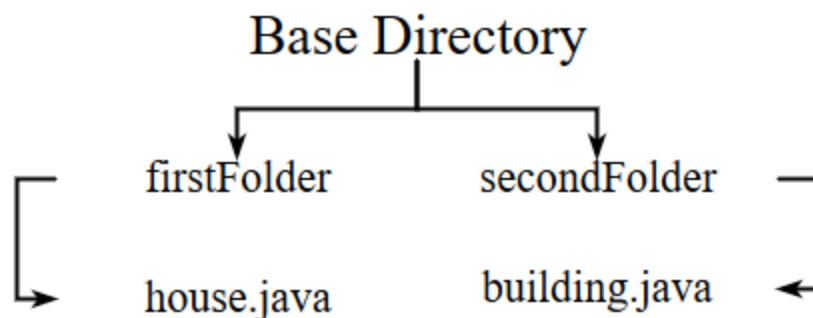


Figure 8.1: Folder Structure

2. Load the program into Eclipse and Run the application
3. Fill Information

The application will run and a new window will open, Figure 8.2. Fill in the necessary information, by first clicking on the “Select” button, this will open up a directory chooser window, inside the window navigate to the location where your base directory lays and click “Open”. Press the “Run Comparison” button to run the detection, for now you should leave the “Language Extension” and “Blacklisted Files” alone. You can change the other two parameters to meet your needs.

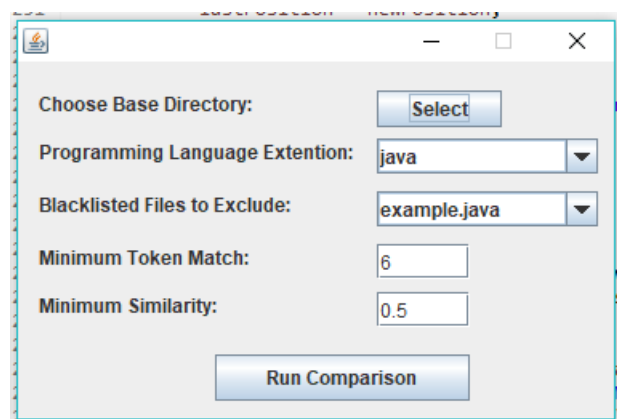


Figure 8.2: Fill Information

4. Run Comparison

Once the “Run Comparison” button has been pressed a new window will open and 4 files are created into your eclipse-directory that houses this copy-detection application, within this folder you will find two HTML files and two CSV files that you can copy into another folder to save for later viewing, as seen in Figure 8.3. The results of the comparison can be viewed in the new window. Scan through each window to view the matched similarities between the pair of programs.

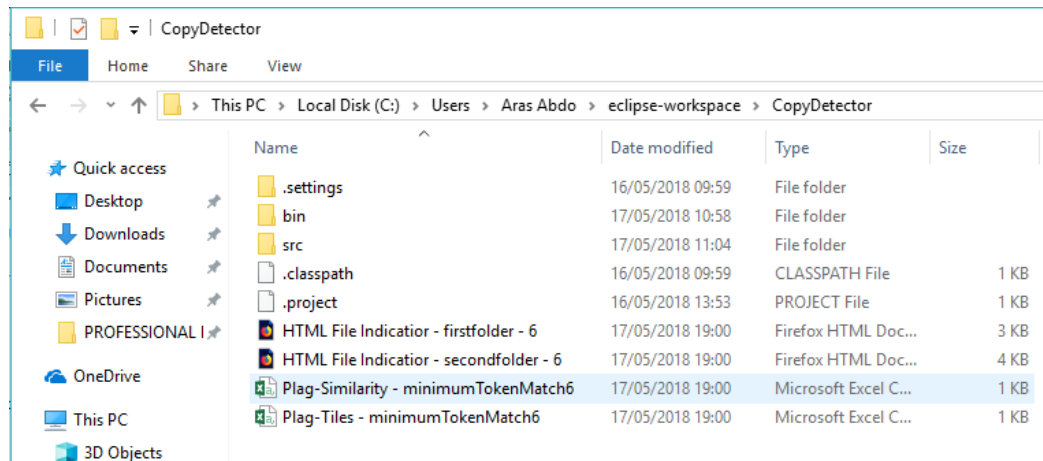


Figure 8.3: Location of Created Files

9. Conclusion and Future Work

In this thesis, plagiarism was introduced and characterised, its use in source-code was identified and understood. The algorithms used to detect plagiarism are constantly being developed, though our research we have seen how similarity comparisons have evolved, from the simple attribute-counting systems to the more complex structure-metric approaches. The most popular of these detection techniques are demonstrated in this paper, detailing how they fight against the many different disguises/attacks, from the simplistic approaches of reordering the methods, to the more complicated disguises; where code statements are translated to equivalent structures.

Our purpose for this project was to build a system/tool that would be capable of analysing a pair of programs for similarities, keeping in mind that it would be likely implausible to detect every single case of plagiarism. Such a system would support the already burdened academic institutions. I believe that the implementation of this thesis proposed detection tool was successful in the grand scheme of the project goals and objectives., every goal and object was met, the success of meeting these goals broaden my knowledge throughout the project and help develop and implement a tool that has been tested in Chapter 7: as been successful in detecting plagiarism on a wide variety of subject. I personally gained an appreciation for an area that was initially dull to me, and I only hope that this appreciation continues in the form of future improvements to my copy detection tool, healing the issues and problems that were identified with the system and ultimately producing a tool that would find a home in Universities.

Unquestionably the project was both challenging and interesting, I have learned a tremendous amount from this project, not only did it widen my understand on plagiarism, but I was able to grasp the social and economic problems that are caused by this challenging issue. My ability to program in Java has improved, I am now more confident to venture into the new unknown areas, to try and find a solution to a problem. Moreover, to help support a solution that already exists. I can only hope that this project is published on the web and viewed by individuals who can further the area.

For future work: With the use of the internet, the search for information is becoming increasingly more easy, because of this we need to continuously find new efficient approaches to detect source-code plagiarism, and optimise the accuracy and speed of the detection process. Although the RKR-GST algorithm has been proven to be very efficient in most situation, there are still several weaknesses. The average complexity of the algorithm is close to linear, this can be improved by, for example; currently the system half's the search length(s) when a string that has a greater length or equal to it is found, and then the process is repeated. A better strategy can be used to improve the complexity, by varying the decrease of the searchLength(s), depending on the number of matches that have been put into the heap previously. If the value is large then it may be better to slowly decrease "s". More practical and theoretical work need to be carried out on the subject of detection algorithms. This is a challenging problem that will require much work, perhaps in the future I will be a part of a community that tries to solve this issue.

It would be possible to extend the application to deal with multiple different programming languages, by adding the appropriate global variables to the Normaliser class, additionally the Normaliser class needs to be completed so that it is able to deal with the last two requirements;

reordering methods and removing non-target language tokens. The RKR-GST needs to be examined and improved to fix the issues that occurred during the testing phase in Chapter 7.

10. Statement of Ethics

Ethics in the realm of computing consists of issues of copyright, privacy concerns, intellectual property, and ultimately how the given application may negatively affect society. The discussion of ethical issues in regard to plagiarism detection is vast and complex, and so we only cover the more obvious issues here.

What are the Examiner's motives for choosing to do plagiarism detection on a person's source-code? I have personally raised this question on many occasions, has the instructor's instincts kicked in, did they notice something that would otherwise be dismissed by an ordinary person? Or have they deliberately performed this detection to mentally affect a naughty individual that hasn't plagiarised (Student). Well regardless of the decision to perform detection, the individual on the other side of the conversation is usually preplaced by the decision. Although plagiarism violates the academic and social norms that have been set out by the "community". I believe that plagiarism occurs because it hasn't been properly taught, instead it is assumed that everyone understands why it is not tolerated. Lessons should be taught on what plagiarism is and how it occurs, combined with the real costs of such an action, for both the plagiariser and the victim. This will help develop our attitudes and thoughts on the subject, I have personally had previous teachers tell me that a specific form of writing was not plagiarism, this stems from the fact that these teachers themselves did not understand plagiarism.

Another issue that is of particular importance is the storing of students work into a system that is either electronic or paper. To whom does this work belong to, is it the students or do the rights of the work belong to the institution? In regards to Textual plagiarism, when a student copies another's essay, and that essay is uploaded to a third part system, such as Turnitin, that will hold a copy of this students work in their database. Is this morally right? Especially if the copied version is almost identical to the original work, in this case the original work is essentially stored as well, regardless of whether plagiarism has been detected. This relates to the Data Protection Act (DPA).

The determination of whether a piece of work has been plagiarised must be made by a human, and not the copy-detection tool itself. There are certainly cases where faculty members in academic institutions who have little to no experience with plagiarism software, like the one constructed in this thesis, assume that the similarity result produced at the end of the comparison will neatly divide into good and bad groups, those that are not guilty and the ones that are. This situation must not occur, the student's future should not be determined by a program, but a sophisticated human being.

At the beginning of the development process, the legal, ethical and practical issues should be considered, and should be a prevalent thought throughout the development. For a vast majority of these questions, it is unlikely that a correct answer will surface. Thus, it will only suffice when a system, enables options for the user to select between. One such example, is the question stated earlier, the ownership of the paper/source-code. An option should be made

available as to whether the students work is stored, and if it is stored whether it should be uploaded onto a local or global system. Of course several other standards should be set in place. In my copy detection tool, we deal with the storing of the Students work very simply, the matches that have been detected by the system are highlighted and the entire program is save to a HTML file that is stored on the user's computer, what they chose to do with that file is entirely up to them. Ultimately, nobody else is able to see the files, other than the user that process them into the system.

Bibliography

- [1] **A. Parker and J. O. Hambley, "Computer algorithms for plagiarism detection," IEEE Transactions on Education, 1989.**
- [2] A. S. Bostan, "Winnowing Algorithm for Program Code," Hamburg University of Technology, 16 July 2017.
- [3] K. V. a. M. Wise, "Software for detecting suspect plagiarism: Comparing structure and attribute-counting systems.," in *Australian Conference on Computer Science Education*, Sydney, July 1996.
- [4] P. Cornic, "Software Plagiarism Detection Using Model-Driven Software Development in Eclipse Platform," University of Manchester, 2008.
- [5] A. G. L. & A. Ahmad, "Plagiarism Detection in Java Code," Linnaeus University, 2011.
- [6] S. Burrows, T. Seyed and J. Zobel, "Efficient and Effective Plagiarism Detection for Large Code Repositories," in *Second Australian Undergraduate Students' Computing Conference*, 2004.
- [7] K. W. Bowyer and L. O. Hall, "Experience Using "MOSS" to Detect Cheating On Programming Assignments," University of South Florida, Florida.
- [8] M. J. Wise, "YAP 3: Improved detection of similarities in computer program and other texts.," in *Technical Symposium on Computer Science Education: Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, Philadelphia, Pennsylvania, 1996.
- [9] L. Prechelt, G. Malpohl and G. Philippsen, "JPlag: Finding plagiarism among a set of programs," Karlsruhe Institute of Technology, 2000.
- [10] D. Gitchell and N. Tran, "Sim: A Utility For Detecting Similarity in Computer Programs," in *SIGCSE technical symposium on Computer science education* , 1999.
- [11] L. Divya, J. L. Sony, S. Sreepurba and V. B. Elizabeth, "Software Plagiarism Detection Techniques: A Comparative Study," *International Journal of Computer Science and Information Technologies*, vol. 5, p. 4, 2014.
- [12] S. Schleimer, D. S. Wilkerson and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting," in *2013 ACM SIGMOD International Conference on Management of Data*, Canada, 2013.
- [13] K. J. Ottenstein, "An Algorithmic Approach to the Detection and Prevention of Plagiarism," Department of Computer Science Technical Reports, 1976.
- [14] U. Manber, "Finding Similar Files in a Large File System," Department of Computer Science , Arizona, 1993.
- [15] M. J. Wise, "String similarity via greedy string tiling and running Karp-Rabin matching," Department of Computer Science, University of Sydney, Australia, 1993.
- [16] "What is JPlag," Institute for Program Structures and Data Organization, [Online]. Available: <https://jplag.ipd.kit.edu/>. [Accessed 18 May 2018].

- [17] B. Belkhouche, A. Nix and J. Hassell, "Plagiarism Detection in Software Designs," in *Annual Southeast Regional Conference*, 2004.
- [18] M. Rouse, "TheServerSide," December 2016. [Online]. Available: <https://www.theserverside.com/definition/Java>. [Accessed 30 April 2018].
- [19] A. Diawn, "Eclipse," Tutorialspoint, [Online]. Available: <https://www.tutorialspoint.com/eclipse/index.htm>. [Accessed 20 April 2018].
- [20] "Eclipse," [Online]. Available: <https://www.eclipse.org/windowbuilder/>. [Accessed 20 April 2018].
- [21] Stackify, "Stackify," 6 April 2017. [Online]. Available: <https://stackify.com/what-is-sdlc/>. [Accessed 15 April 2018].
- [22] "What is Iterative Model," professionalQA, 31 January 2018. [Online]. Available: <http://www.professionalqa.com/iterative-model>. [Accessed 3 May 2018].
- [23] M. P. Oakes, "Plagiarism and Spam Filtering," University of Wolverhampton.

END OF THESIS