Attack of the clones: an investigation into removing redundant source code

Bailey, John Oliver

Acknowledgements

I would like to thank my supervisor Dr Liz Burd for her support and guidance throughout this thesis. Dr Ira Baxter of Semantic Systems has been extremely generous with his time and allowed me to use CloneDr. Thanks also goes to Dr Toshiro Kayima of the Osaka University for allowing CCFinder to be used in this thesis. Plus a special thank you to my parents for obvious reasons.

Attack of the Clones: An Investigation into Removing Redundant Source Code

John Oliver Bailey

MSc

University of Durham

Department of Computer Science

2002

Table of Contents

Figures and Tables

Table of figures

Table of tables

Declaration

This thesis is my own work. Part of it formed the basis for a paper published in the Source Code Analysis and Manipulation workshop 2002.

Evaluating Clone Detection Tools for Use During Preventative Maintenance,
John Bailey, Elizabeth Burd

Copyright

Abstract

Long-term maintenance of code will often lead to the introduction of duplicated or 'cloned' code. Legacy systems riddled with these clones have large amounts of redundant code and are more difficult to understand and maintain. One option available to improve maintainability and to increase software reuse, is to re-engineer code clones into reusable components. However, before this can be achieved detection and removal of this redundant code is necessary.

There are several established clone detection tools for software maintenance and this thesis aims to investigate the similarities between their output. It also looks at how maintainers may best use them to reduce the amount of redundant code in a software system. This will be achieved by running clone detection tools on several different case studies. Included in these case studies will be a novel tool called Covet inspired by research of Mayrand [May96b] which attempted to identify cloned routines through a comparison of software metrics generated from each routine.

It was found that none of the clone detection tools achieved either 100% precision or 100% recall. Each tool identified very different sets of clones. Overall MOSS achieved the greatest precision and CCFinder the greatest recall. Also observed was that the use of automatically generated code increased the proportion of clones found in a software system.

1. Introduction

Code cloning occurs when code is transplanted via a copy and paste function. This area is of interest to various groups within the software engineering community. In particular the software maintenance community view cloning as bad because it increases the size of software unnecessarily. It is also performed on an ad-hoc basis and is never documented. Furthermore if the original code has some previously undetected error then when it is copied instead of just one error cloning means there are now at least two errors within the application. Cloning produces redundant code because the original code could have instead been reused properly through parameterisation.

Extra code adds to the complexity of a system and hence increases the cost of software maintenance. Clone detection can be applied to a software system to identify potential clones and therefore presents opportunities to reduce the size of a software system. This is of particular interest to software companies that produce software for hardware with limited storage capacity such as mobile phones and pocket PCs.

Cloning often occurs when programmers want to save time. Ducasse [Duc99] offers three main reasons why programmers would clone code.

"*(a) Making a copy of a code fragment is simpler and faster than writing the code from scratch. In addition, the fragment may already be tested so the introduction of a bug seems less likely.*
(b) Evaluating the performance of a programmer by the amount of code he or she produces gives a natural incentive for copying code.
(c) Efficiency considerations may make the cost of a procedure call or method invocation seems too high a price."

There is another specific form of code cloning which is related to Duccasse's first point, but is of interest to academic institutions. Clone detection techniques can be (and are) applied to the detection of plagiarism within programming courses. It would be nearly impossible for markers to manually check a large set of student

programs so this saves a great deal of time. As well as saving markers time plagiarism detection tools may pick up programs where students have attempted to hide their cheating which may have fooled a human. Examples of this would be the systematic re-labelling of variables or superficial cosmetic changes to a program's source code.

Recently the automated tools community has developed automated methods for detecting code clones using a variety of techniques. These tools can be classified as either language dependent or language independent. Language dependent tools make use of lexical and parser technologies to compare sections of source code at a more abstract level. These tools have the ability to ignore cosmetic changes to code (such as systematic changes to variable names). An example of a language dependent tool is Semantic System's Clone Detection and Removal (CloneDr) [Bax98] tool, which uses parser technology to compare sections of code. It also aims to automatically remove clones and replace them with a "unifying macro"[Bax98].

Language independent tools treat source code as plain text and use string manipulation to compare program sources line by line. These tools obviously do not require any parsing or lexical analysis and so can be used for any programming language and also for natural languages. Techniques such as removing white space and line ordering help limit a tool's sensitivity to minor changes in layout.

Visualisation is an important aspect of clone detection to allow maintainers / markers to confirm if a potential clone identified by a tool is an actual clone. Tools such as DUPLOC [Duc99] present the line-by-line comparisons in a dot plotting. Geneticists searching for similar strings of DNA first used this technique. "*Such 'dot drawings' allow immediate recognition of typical situations.*" [Duc99]. The ability for a maintainer or marker to view the suspected cloned sections of code side by side is vital to allow verification. Making the verification part of clone detection process as efficient as possible is the aim of any clone detection tool.

A good number of clone detection tools are available for both commercial clone detection and academic plagiarism detection and this thesis will compare the clones they identify to establish if they are similar. It will be interesting to see if the tools reliably identify clones or if there is a high percentage of false positives within their

results. Another point to investigate is how the development of a software system affects its level of cloning. For example, as systems increase in size does the proportion of clones within that system increase at the same rate? Also with the increasing use of integrated development environments and automatically generated code does this mean that automatically generated clones will become more prevalent?

As part of the investigation into code clone detection tools a new tool will be created which will aim to efficiently identify cloning based on metrics derived from source code (Covet). An interesting comparison will be how the clones identified by a metrics based tool compare to clones identified using other techniques.

1.1. Criteria for success

As part of the investigation into code cloning the criteria for success, which will be evaluated in the Conclusion chapter will be as follows;

- A literature survey reviewing current issues relevant to software maintenance and in particular code cloning
- The development of an efficient metric based clone detection tool
- A comparison of a range of clone detection tools, focusing on their precision, recall and intersection of results.

1.2. Plan for the Thesis

The format of this thesis is as follows. Chapter 2 presents a literature survey whose focus is software maintenance, metrics and visualisation. These three general areas of research are then combined in Chapter 3 which looks specifically at the current state of the art of clone detection. This consists of a broad overview of the causes of cloning, the need for / benefits of clone detection and a description of the tools that are currently available. Chapter 4 presents the method for the case studies describing the criteria for choosing the software systems and clone detection tools that will be used in the case studies. It also lists the hypotheses that will be used in the case studies and then describes the experiments that will take place to test them. In Chapter 5 the implementation of Covet and the adaptations to existing clone

detection tools is provided. Explanations of some of the interesting aspects of Covet's development are included here. Particular attention is paid to the metrics chosen for the clone detection algorithm. Results from the case studies are presented in Chapter 6. This chapter provides a breakdown of the results with commentary and graphical representation of the results achieved. Evaluation of the results is provided in Chapter 7. Each of the hypotheses is evaluated with respect to the experimental work carried out in the case studies. Conclusions are drawn in Chapter 8 including comments on the criteria for success and the overall results from the case studies. In addition ideas for further research are also included.

2. Literature Survey

2.1. Software Maintenance

2.1.1. Definitions

There are many definitions of Software Maintenance this section will discuss several. "*Maintenance is an incremental and iterative process in which small changes are made to the system. These changes are often bug corrections or small functional enhancements and should never involve major structural changes.*" [Com00].
This definition from the Software Engineering Institute at Carnegie Mellon University gives a "bug fixing" approach with minor alterations being applied to a legacy system. It suggests that maintenance is reactionary triggered only when an error is discovered. Comella-Dorda [Com00] goes on to define "modernization" as involving more extensive alterations to a system whilst still keeping a "*significant proportion of the existing system*".

"***Software maintenance*** *is the set of activities, both technical and managerial, that ensures that software continues to meet organisational and business objectives in a cost effective way.*" [CSM].
The above view is more abstract, not giving any specific details about potential activities. It does, however, relate software maintenance to business issues and acknowledges that managerial support is needed. Also definition raises the question is raised that if maintenance is not cost effective then what needs to be done instead.

Takang [Tak96] does not give a definition of maintenance but instead summarises motivating factors for software maintenance.

- "*To provide continuity of service*" – bug fixing, failure recovery and coping with changes in hardware or software.
- "*To support mandatory upgrades*" – changes in the law, for example the introduction of a single European currency, or "*attempts to gain a competitive edge over rival products*".

- *"To support user requests for improvements"* – improvements in usability, performance or customisations (for example company colour scheme).
- *"To facilitate future maintenance work"* code / data restructuring, updating documentation.

The first factor relates closely to the definition given in Comella-Dorda [Com00] describing an "odd job man" approach. Takang's third point about supporting mandatory changes could also fall into the definition given by Comella-Dorda [Com00] depending on the scale of the change. For example for a financial system the introduction of a new currency (i.e. the single European currency in the E.U.) would probably require large-scale changes. Responding to user requests would, as the Center for Software Maintenance in Durham [CSM] underlines, require managerial input. Customers will usually contact a support / commercial manager with requests for system improvements. Finally the facilitation for future maintenance work depends on economic as well as technical issues. Software engineers must consider whether the effort expended on this activity will produce a great enough reduction in future maintenance costs. This factor depends on the size and complexity of the required changes. These correspond to the technical aspects of the project.

2.1.2. Types of Software Maintenance

Once a system is being used there are a number of different types of changes that could be made to it. These different types of maintenance are as follows:

Perfective

Usually this type of maintenance aims to expand on the original requirements of the system. Munro [MMLecture] defines it as *"improving the function of software by responding to user defined changes"*. As users explore the system more novel or unexpected usage can arise. Enhancing the system by providing the extra functionality or performance improvements constitutes perfective maintenance. Changes will usually affect the requirements, code and design of the system. One of the dangers of this type of growth is that documentation becomes out-of-date and the overall structure of the system is lost.

Adaptive

Software engineers can produce a near perfect, stable system but any systems in active use has to evolve to reflect changes in its outside environment. A system's environment refers to "*the totality of all conditions and influences which act from outside upon the system*" [Tak96]. Examples might include new laws (i.e. the Data Protection act) a new company strategy or structure (mergers) and changes to hardware and other software being used in the organisation. Changes in these instances will usually affect the code and design of the system.

Corrective

Corrective maintenance involves correcting errors uncovered in the design of the system, the logic behind the code or the code itself. This type of maintenance is *ad hoc* by nature as it is reactionary (i.e. an error is found and the reaction is to fix it straight away). Such changes will usually only affect the system's code and so the fix can cause further errors. ("*unforeseen ripple effects*" [Tak96]). In the long term corrective maintenance increases the program's complexity, as changes are disruptive the structure of the system and documentation is usually not updated.

Preventative

It is widely accepted that prevention is better than the cure. Munro describes a preventative maintenance approach in terms of software maintenance as "*updating the system to forestall future problems and improve maintainability*" [MMLecture]. By taking steps to improve the structure of the system and update documentation, maintenance costs in the long term can be decreased. In order to make such major changes to a system some form of reverse engineering is required. Reverse engineering is looked at in detail later in this Chapter, Section 2.5.1. It allows maintainers to build abstract representations of the system by using the only reliable documentation at hand i.e. the source code. Such representations assist program understanding (see Program Understanding in Section 2.1.4) that is used in creating a mental model of the system in the maintainer's mind. Once an accurate model of the system is built then the process of restructuring the system and updating the documentation can begin.

2.1.3. Program Understanding / Comprehension

During maintenance software engineers are asked to make alterations to a system. Rugaber [Rug94] points out however that rather than the actual act of modification itself "*the greatest part of the software maintenance process is devoted to understanding the system being maintained*". Ideally before a change can be made the modifier should know how the alteration would affect the rest of the system. Thus, it is vital that maintainers acquire adequate knowledge and understanding about the system. "*A programmer must first understand the code well enough to know what changes are needed, how to make them, and how to integrate new code into existing applications.*" [May97]. Clayton [Clay98] makes the point that programmers have "*...no agreed-upon definition or test of understanding!*"

Rugaber [Rug95] presents a simple and clear definition for program comprehension, "***Program comprehension*** *is the process of acquiring knowledge about a computer program.*"
Rugaber [Rug95] also presents the difficulties involved in carrying out program comprehension. In particular, there are five gaps he identifies that must be bridged to enable successful program comprehension. These are:

1. "*The gap between a problem from some application domain and a solution to it in some programming language.*"

A program is written to solve a problem in a particular application domain. Utilising the source code to solve the problem relies on "*hints*" such as mnemonic variable names or comments. Rugaber [Rug95] points out that such hints "*are inherently informal and tend to be out-of-date*". Automatic program understanding tools can only work on the formal source code. Responsibility for bridging this gap then falls entirely on the reverse engineer or program reader (knowledge of both the programming language and the application domain is vital here).

2. "*The gap between the concrete world of physical machines and computer programs and the abstract world of high level design descriptions.*"

Reverse Engineers need to decide what are the most important aspects of the system to represent and at what level this abstraction should be presented. Abstractions can overlap and one abstraction can cover more than one concept.

3. "*The gap between the desired coherent and highly structured description of a system as originally envisioned by its designers and the actual system whose structure may have disintegrated over time.*"

Originally the architecture of the system was documented when the system was designed. Design documents present a highly structured and detailed description of how the system works and describe the purpose of each component. During the working life of the system bug fixing and various other changes will have undoubtedly occurred. These will have not only made the design out-of-date but will have also eroded the architectural structure. It is the task of the program reader to capture the higher-level design and actual purpose of what may in effect be a completely different system to the original design.

4. "*The gap between the hierarchical world of programs and the associational nature of human cognition.*"

"*Raw data are perceived, patterns are detected, and are constructed relating them.*" [Rug95]. Program readers must use their knowledge about the programming language, application domain and other aspects of software engineering to recognise patterns at a low level. Once this is done they must build up high level "*chunks*" [Rug95] from the information available at the lower level.

5. "*The gap between the bottom-up analysis of the source code and the top-down synthesis of the description of the application.*"

Higher-level constructs in a program are built from low-level patterns. Reverse engineers must analyse the program from the bottom-up whilst at the same time bearing in mind the idea of the higher-level purpose of the program. "*As the program is perused, the overall concept is refined into a more complete description by adding lower level details*" [Rug95]. So the synthesis is carried out top-down. The

complicating factor is that the synthesis and analysis must be carried out simultaneously "*in a synchronized fashion*" [Rug95].

Comprehension Aids

There are a number of aspects of a program that can be examined to aid comprehension. Takang [Tak96] lists the following:

Problem domain

The problem domain is the specific area in which the problem being addressed exists. Examples might include telecommunications, financial services or public transport. Familiarisation with the problem domain allows the developer to put into context the work being carried out. It also allows them to make more informed decisions about which algorithms and tools to use.

Execution Effect

The execution effect is how the system behaves when run. Initially maintainers may not need to know low-level details about program interaction. If this knowledge is required, control and data flow diagrams are often used to give a diagrammatic description. Understanding execution effect allows maintainers to establish whether or not changes have had the desired effect.

Cause-effect Relation

This relation allows maintainers to observe a specific effect and discover which part of the system caused it. The cause-effect relation is also useful to predict ripple effects / knock-on effects of changing a piece of code.

Product-environment relation

An understanding of how the product fits into its environment is vital to predicting and understanding how changes in the product environment will affect the system.

Decision-support features

Such features are attributes of the system such as complexity and maintainability. They provide both technical and managerial staff with information required to make decisions about the maintenance of the system.

Mental Model

Our understanding of something complicated relies on our ability to use abstraction to create a high level representation. The something here is termed the '*target system*' and the representation a '*mental model*'. Building a mental model of a system uses cognitive structures and cognitive processes. Takang [Tak96] gives the following definitions:

- "*Cognitive structures represent the way in which knowledge is stored in human memory.*
- *Cognitive processes describe how the knowledge is manipulated during the formation and use of mental models*"

Weiderman [Wei97] describes these cognitive aspects of program understanding as "*the study of problem-solving behavior of software engineers.*"

Forming a mental model requires the engineer to examine the target system in detail using "*observation, inference or interaction*" with the target system [Tak96]. A mental model does not necessarily have to be 100% complete but it should however highlight the key features of the system (i.e. the functionality).

In order to form a model a program comprehension strategy must be chosen which will dictate how the engineer will go about examining the system. Table 2.1 contains definitions of several popular strategies;

Strategy	**Description**
Top-down	"*The top-down approach begins with a pre-existing notion of the functionality of the system and earmarks individual components of the system responsible for specific tasks.*" [Til96]. Takang [Tak96] describes this as "*mapping how the program works (programming domain) to what is to be done (problem domain)*"
Bottom-up	"*The bottom-up approach reconstructs the high-level design of a system, starting with source code, through a series of*

	chunking and concept-assignment steps." [Til96]. The chunking mentioned here is the clustering of recognised patterns.
Iterative Hypotheses Refinement	"*The iterative refinement approach creates, verifies, and modifies hypotheses until the entire system is explained by a consistent set of hypotheses.*" [Til96].
Opportunistic	This involves combining all of the approaches to achieve the best results.

Table 2-1 Program Comprehension Models

Program Comprehension Tools

Takang [Tak96] lists one of the objectives of a Program Understanding Tool (PUT) as "*to serve as aids to enable the understander to speed up the understanding process*". Most PUTs analyse the source code of a system to build higher-level representations of the system. There are a great variety of such tools available. Some examples now follow:

GRASP – Graphical Representation of Algorithms, Structures and Processes

GRASP provides a development environment which supports the creation of Control Structure Diagrams (CSDs). These CSDs highlight data and control flow of the system. They are made explicit via the actual source code.

PUI - Program Understanding Implement

"*The main objective of PUI is facilitate the comprehension and is based on a matrix of relations between elements of a program*" [Cha97]. Elements are aspects of a program such as variables, types and functions and by presenting them in a controlled and gradual manner the user can build up his/her knowledge of the system at their own pace.

Foundations of Automated Program Comprehension

There are various levels of program analysis. Rugaber [Rug95] presents a list of approaches ranging from a purely textual approach to dynamic analysis of how the program executes.

- Textual Analysis

- Textual analysis involves looking at source code as purely a textual document. The main usage for this type of analysis is to look at the size of the system (in lines of code). This is the most important factor into how much effort will be required to understand it.

- Lexical Analysis
  - Lexical analysis allows recognition of identifiers, operators, keywords etc. This can be useful in tracking how frequently and where identifiers occur in the code. It also allows the compilation of software complexity metrics – i.e. number of unique operators or variables.
- Syntactic Analysis
  - Syntactic analysis relies on a language specific parser. Parsers can build an abstract syntax tree, which forms the basis of most sophisticated program analysis tools.
- Control Flow Analysis
  - Control Flow analysis relies on the syntactic structure of the program being known. Two types of control flow analysis exist *intraprocedural,* which determines the order of execution of statements within a subprogram and *interprocedural*, which looks at the calling relationship between subprograms within the system.
- Data Flow Analysis
  - Data flow analysis provides extra information that control flow analysis lacks. Data flow analysis looks at how variables are defined and referenced. It is significantly more complicated than control flow analysis. "*In particular, whereas CFA merely has to detect the possibility of loops, DFA has to describe what might happen to the variables inside the loop body.*" [Rug95]. Information such as whether variables are referenced before being defined or whether or not code will not execute is available.
- Program Dependence Graphs
  - Program dependence graphs treat control and data flow dependencies in the same representation.
- Slicing
  - Slicing targets a particular variable or line in code and determines either what affects that target or what the target effects.

- Cliché Recognition
    - "*Searching the program text for instances from common programming patterns*" [Rug95]. These are termed clichés or idioms. Tools are available which compare patterns with a library of previously defined idioms.
- Dynamic Analysis
    - Previous analysis approaches have been based on a static analysis of a system. Dynamic analysis involves "*systematically executing a program*" [Rug95].

2.1.4. Reverse Engineering

As previously mentioned, maintenance of legacy systems (especially corrective) can lead to out-of-date and therefore misleading documentation. Hence, these documents are no longer useful tools to maintainers wishing to understand a system. Thus, only the source code can be relied upon to describe the system correctly. In order to gain full advantage of this important documentation there needs to be some automatic way of building a higher-level representation of the code. Reverse engineering is a technique that can be employed to form such representations. Rugaber uses the term reverse engineering in his description of program understanding, "*The process of understanding a system involves reverse engineering the source code*" [Rug92]. Chikofsky and Cross [Chi90] have presented the following definition:

"*Reverse Engineering is the process of analyzing a subject system to:*

- *identify the system's components and their interrelationships and*
- *create representations of the system in another form or at higher levels of abstractions*"

The analysis mentioned in the definition above is carried out with the aid of a program understanding tool (see Program Understanding 2.1.4). Components are anything produced during the software life cycle such as requirements specification, detailed design and the source code itself. Abstraction is a key idea behind reverse engineering. By abstraction it is possible to convey the major features of the system without overwhelming the maintainer with masses of low-level information. There

are three types of abstraction used in reverse engineering that are applicable to software systems:

Function Abstraction

Function abstraction involves identifying the functions within a target system. The focus is what the function actually does rather than how it does it.

Data Abstraction

Data Abstraction involves identifying the actual data objects as well as the functions that use them. The creation of abstract data types based on the data available may be useful. Encapsulating the data item with its associated functions into a class might also be an option.

Process Abstraction

Process abstraction involves extracting the exact order in which operations are carried out and allows an insight into the processes in use. There are two types of processes that can be abstracted, concurrent and distributed. Concurrent processes communicated via a shared memory / data and distributed processes use message passing and do not have access to shared data.

Aims of Reverse Engineering

"*The goal of reverse engineering is to facilitate change by allowing a software system to be understood in terms of what it does, how it works and its architectural representation.*" [Tak96]. Reverse engineering can provide the following facilities to enhance a maintainers understanding.

- Recover lost information
    - "*Recovering lost information means recovering both development of never existing design documents as well as recovering information that has been lost during software development or even during years of maintenance operations.*" [Klo96]. Recovered information could be a formal specification in Z or a design document using generated control and data flow diagrams.
- Assisting with maintenance - identification of side effects and anomalies:
    - Identification and treatment of such unintended aspects of a system fall into the category of corrective maintenance. "*Reverse Engineering*

advocates this goal by several techniques, such as providing additional documentation and restructuring" [Klo96].

- Migration to another hardware/software platform or integration into a CASE environment:
  - "*In order to take advantage of a new software platform (for example, a CASE environment) or hardware platform a combination of reverse and forward engineering can be used*" [Tak96]. By extracting the specification and design of the old system it is much easier to redevelop the system and maintain consistent functionality on a new platform. This approach ensures developers reduce the risk of functionality loss.
- Facilitating software reuse:
  - Components extracted by reverse engineering techniques (documentation as well as source code) will be at a higher level of abstraction. These are therefore ideal candidates for reuse as they capture something general and useful.

Types of Reverse Engineering

There are three major types or levels of reverse engineering

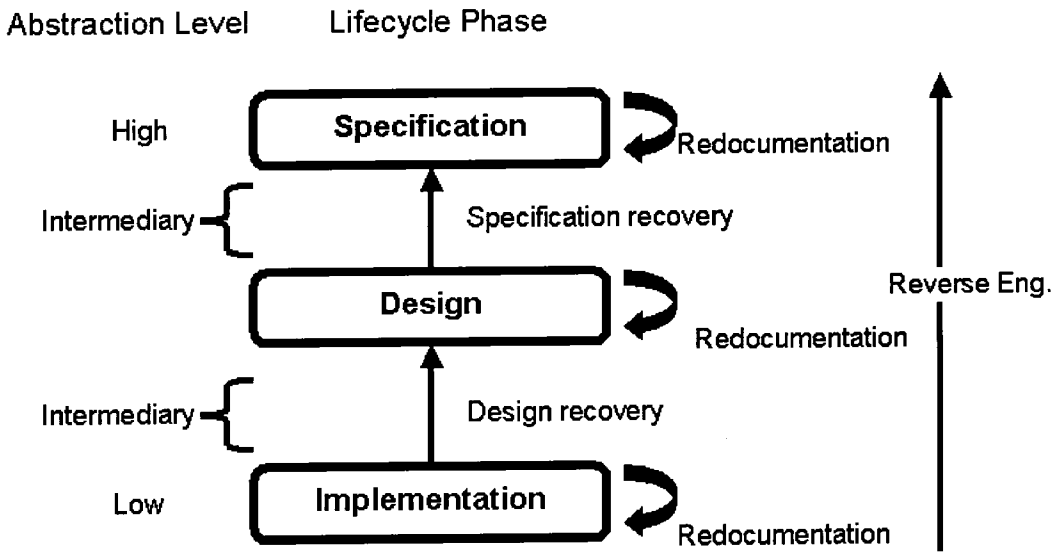


Figure 2-1 Levels of abstraction of a software system [Tak96]

Figure 2.1 is taken from Takang [Tak96] and shows the relationship between redocumentation, design recovery and specification recovery within reverse engineering.

Figure 2.1 shows the levels of abstraction at three stages of the software lifecycle. At the lowest level the implementation. Design recovery uses this to move to a higher level of abstraction, the design. From the design documentation it is possible to go a further stage and recover the specification of the implemented system. This may not be the same as the original specification of the system due to past maintenance processes. At each stage, redocumentation is carried out and can be browsed by the maintainer.

Redocumentation

Redocumenting a system is vital when the original documentation has been lost or has become out of date. It involves recreating a representation which carries the same meaning and at the same abstraction level.

Design Recovery

Design recovery is the process of recovering useful higher-level abstraction directly from inspecting the source code. This new recovered design may not be the same as the original design because of changes made during maintenance. It can be used as a baseline for redeveloping and modifying the system.

Specification Recovery

A system may be a candidate for a complete redevelopment. Here the design of the system may not be useful, as the new system will achieve the same functionality in a totally different way, for instance, in order to improve performance. By recovering the specification, the functionality of the legacy systems requirements are uncovered and a new design can be created to meet this specification. Obtaining such a specification requires access to the source code (and possibly a recovered design). Specifications can be produced in many forms for example UML or a mathematical specification language such as Z. A comparison of the original and recovered specifications may reveal that the system serves a different purpose than was originally intended.

2.2. Software Visualisation

After the process of clone detection has been completed the results have to be presented to the user. This section will focus on the visualisation of software in general.

2.2.1. Definitions

Software visualisation is a broad area of research consisting of several specialised sub fields. These include program visualisation, algorithm animation, data visualisation and code visualisation. Jeffrey [Jef99] defines software visualisation as "*the depiction of software artefacts such as directories, user data or log files*". With regards to program visualisation he identifies it as "*a sub field of software visualisation focused on the dynamic behaviour of programs themselves rather than the data they manipulate.*"

According to Domingue [Dom95] software visualisation is "*basically the unification of algorithm animation and program visualization*", algorithm animations are "*high level characterisations of how data is manipulated during a program execution.*" And finally program visualisation systems "*display graphical representations that are more tightly coupled with a program's code or data and show more or less faithful representations of the code as it is executing.*"

Large Software systems are often complex and in their original source code representation extremely difficult to comprehend. It is this comprehension that any form of visualisation must assist, "*Software visualisation aims to aid the programmer by providing insight and understanding through the graphical displays and views, and to reduce the perceived complexity through the use of suitable abstractions and metaphors.*" [Kni01a]

Maintainers or indeed anyone wishing to comprehend and then manipulate a piece of software must build a **mental model** (see Program Understanding 2.1.4). Jonassen [Joh95] describes such models as "*...the conceptual and operational representations that humans develop while interacting with complex systems*". In order to assist the

construction of such a model it is widely accepted that a graphical representation of the system makes understanding it easier.

Stasko [Sta93] states that "*The general term* ***program visualization*** *refers to graphical views or illustrations of entities and characteristics of computer programs.*" Stasko then introduces a method for characterising program visualisation systems with respect to four terms:

- Aspect
    - Focusing on a particular aspect of the system to represent. A simple form of aspect level program representation is "*an enhanced presentation of program text*". For example line highlighting.
- Abstractness
    - The same aspect of a program can be represented at different levels of abstraction.
- Animation
    - Animation involves representing the dynamic state of the system. Showing the stages involved in adding nodes to a linked list is an example provided in [Sta93].
- Automation
    - Program visualisations can be either almost totally generated automatically or may require significant input from the programmer to form the representations of a system.

Roman [Rom92] defines program visualisation as "*mapping from programs to graphical representations*". Like Stasko [Sta93], Roman identifies four characteristics; Scope, Abstraction / Specification and Technique. Figure 2.2 is taken from the paper and gives an overview of the mapping process.

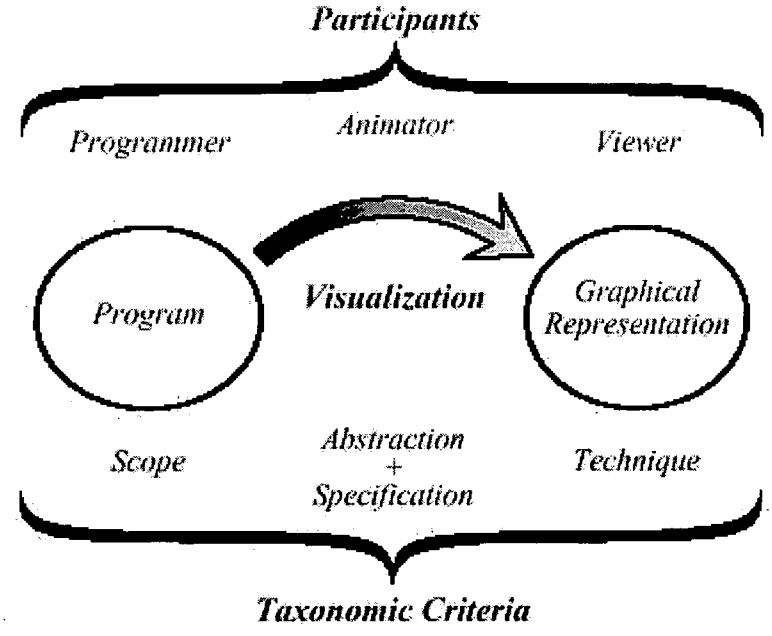


Figure 2-2 Mapping programs to visualisation

Figure 2.2 details the relationship between the user of a visualisation, the visualisation and the characteristics of visualisation.

According to Young [You98] there are six fundamental points (table 2.2) that have to be considered when designing visualisations. Although these are related to 3D visualisations they are general enough to relate to any form of visual representations of software. They also appear to overlap with those of Stasko [Sta93].

Representation	Designers must consider in what form they wish to present the software components and how they will map them onto a graphical form. Efforts must be made to make the information presented as clear and intuitive as possible and hence aid rather than hinder information retrieval.
Abstraction	Abstracting away from low level details (i.e. source code, textual reports on clones, raw data) is one of the main purposes of any form of visualisation. Decisions on what level to abstract to will ultimately dictate how the system is to be used.
Navigation	Large systems result in large visualisations. It is essential to

	provide adequate support for users to get to the information they require quickly. Young [You98] suggests the use of "*signposts, landmarks and paths*". Tools such as maps are also a useful aid as in the real world.
Correlation	Users require access to both the information and the visualisation to get some benefit. This requires that the visualisation be linked with the information it is representing.
Automation	Visualisations can be generated manually by users (as in the design stage), or automatically using a program understanding tool like CodeSurfer [CSurferWP] or by using a combination of both. Allowing the user to 'build' the visualisation as they explore a system. It is suggested that this approach may be more beneficial to their understanding of the system rather than relying completely on a visualisation of the system.
Interaction	Interaction may simply be the user navigating through the system. Users may however require more, data miners for example. Filtering and extraction are techniques that could aid the user focus on the information they are interested in.

Table 2-2 Fundamental aspects of 3D modelling

2.2.2. Software Visualisation of Legacy Systems

It has already been identified that software systems are complicated. This is especially true of legacy systems that have been maintained for a number of years. Eick [Eic96] states that "*Knowledge of code decays as the software ages and the original programmers and the design team move to the new assignments.*" Eick [Eic96] also offers software visualisation as a tool to "*help software engineers cope with complexity and to increase programmer productivity*". The expression "*a picture is worth a thousand words*" would seem to be apt here. Legacy systems can contain hundreds of thousands of source lines and without the aid of some form of overviewing the task of comprehension is both time and effort intensive. One possibility should be to consult the system documentation, however this is usually out-of-date, and so the source code is the only reliable description of how the system

operates. "*Pictures of the software can help slow down the knowledge decay by helping project members to remember and new members to discover how the code works.*" [Eic96]. According to Eick [Eic96] there are three basic properties of software to be visualised:

- Software Structure – directed graphs where the nodes can represent a method and an edge the calling relationship between two methods.
- Run-time behaviour – animated algorithms can represent the workings of an algorithm
- The code itself – syntax highlighting, line highlighting.

Eick [Eic96] claims that current software visualisations including algorithm animation are designed for smaller scale systems and do not scale up well. Also "*algorithm visualizations are usually hand-crafted and require the designer to understand the code before visualizing it* ". Hence, they offer little benefit to program comprehension. In order to gain the "big picture" of the system Eick describes a technique and tool that visualises "*program text, text properties, and relationships involving program text*". Each utilises four visual representations;

- Line Representation – colour coded program text seen at varying levels of magnification. Indentation, length and colouring is maintained even where a line is represented by a single row of pixels. Colour coding can be used to represent a particular statistical view and is used as "*an effective technique of layering information*" [Eic96].
- Pixel Representation – higher information density is achieved by representing a single line of code as a small number of colour coded pixels. Each file is represented in a rectangle whose size corresponds to the actual file size. This allows the user to quickly spot large and small files.
- File Summary Representation – file statistics are presented in a rectangle.
- Hierarchical Representations – by reflecting the hierarchical nature of the software system in a tree-map it allows users to compare the size of systems and their subsystems.

Eick [Eic96] also discusses the need for dynamic program slicing which allows the programmer to identify a line or data structure in a program and the system will automatically highlight code that is relevant to it (and hence will be affected if any modification is made).

2.2.3. Potential Representations

The previous section described the four representations presented in Eick [Eic96]. There are of course many other forms of representing the large amounts of data that any system examining a legacy code would produce. This includes virtual reality and graphs that are now described in detail.

Virtual Reality

Virtual Reality allows users to immerse themselves in a 3D world and to interact with virtual objects. These objects can represent various aspects of a software system. Maletic [Mal01] lists some the software visualisations that his project Imsovision (IMmersive SOftware VISualizatION) provides;

- Static structure of physical source code
- System architecture
- Software metric information
- Dynamic aspects of software
- Software Evolution and change
- Design patterns and reuse abstractions

The major difference and advantage of a virtual reality representation is given by Knight [Kni98] "*an extra dimension that can be used to encode some knowledge or to aid visualisation of the knowledge shown in the two dimensions*". By adding this extra dimension that humans take for granted in the real world, the user has an "*extra element of familiarity and realism*" [Kni99]. Three dimension systems are more intuitive and require less "*cogitative strain*" [Kni99]. Users can apply knowledge gained in the real world to the navigation of a virtual one. Our natural environment is one with three dimensions and so that is the one with which we are most familiar. This familiarity can be exploited by heeding the words of Chalmers [Cha95] "*dynamism, exploration and memory combine over time to help form our perceptions of the environment around us*". Chalmers [Cha95] asserts that on top of providing a

space for people to navigate through, designers must consider how the angles and points of view will affect what the user is viewing.

Chalmers [Cha95] defines "*semantic structure*" as a way to facilitate information retrieval. He states, "*the design should 'make sense' somehow*". An obvious example of this in the real world, which is cited in the paper, is the Dewey decimal system used in libraries. Books of similar textual content are in shelves physically near each other. A side effect of this clustering is that if we cannot find the precise object we wanted, and then there is a good chance of finding something else which may be useful or interesting.

Graphs

Graphs are a well-established representation of software. "*Directed graphs are an appealing target for visualization because of their pervasive presence in information systems*" [Mun97]. There are various types of graphs that can represent data / control flow and other features such as hierarchy and inheritance. More recent developments have been in drawing graphs in 3D space.

Liang [Lia98] presents a System Dependence Graph (SDG). SDGs contain one Procedure Dependence Graph (PDG) for each procedure. A PDG "*represents a procedure as a graph in which vertices are statements or predicate expressions*" [Lia98]. According to Liang there are also two types of edges in the graph; data dependence and control dependence-edges representing the "*flow of data between statements or expression*" and the "*control conditions on which the execution of a statements or expression depends*" respectively.

Burd [Bur97] uses a PERFORM graph to approximate a call graph within COBOL programs. PERFORM graphs are used as an abstraction aid in order to evaluate modules of code and specifically their similarities.

Unfortunately for large systems graph representations can be just as difficult to understand as attempting to read the source. They can become vast, confusing masses of vertices and edges and therefore provide no simplification of the data.

Dot Plotting

Dot plotting is a form of data visualisation particularly useful for spotting patterns. One of its most common uses is in biology for identifying similarities in DNA sequences (homology). "*when applied to software, dot plots identify patterns that range in abstraction from the syntax of programming languages to the organizational uniformity of large, multi-component systems*" [Chu93].

2.2.4. Software Measurement

When measuring attributes of a software system the term metric is often used. Bache explains that this is because "*the term software metric means simply measurement applied to software*" [Bac94]. Both Bache and Fenton [Fen96] agree that within software engineering the term ***metric*** is used a synonym for ***measure.*** Frakes describes a metric as "*a quantitative indicator an attribute of a thing*" [Fra96]. Breaking down attributes into sub-attributes is required in order to generate a set of metrics needed to evaluate the system.

Fenton [Fen96] describes measurement as "*the process by which numbers of symbols are assigned to attributes of entities... ...in such a way as to describe them according to clearly defined rules.*". For example, an entity may be a human being and one attribute would be their height. In order to be useful, measurements must be appropriate, accurate and conform to a standardised system. It is also important to use the correct scale.

As software engineering aims to apply scientific principles to the production of software any measurements taken must be accurate and objective. Measurement can take place throughout the lifecycle of a system and this includes non-executable components. Bache [Bac94] defines such components as "*all the documentation associated with the program such as functional specifications, design documents, test plan, user manuals, etc.*" Simple metrics such as *number of words* or *number of pages* can be used as a measure for the size of documentation.

2.2.5. Software Measurement Goals

Engineers in any field have to take measurements. By measuring aspects of a project's development it is possible to keep a better understanding on its

development. Software is invisible; there are no physical properties on display that could indicate exactly how it functions and whether or not it is functioning properly. Each stage of software's development can be assisted by the use of metrics. Both Takang [Tak96] and Fenton [Fen96] list the main objectives for any software measurement (see table 2.3).

Fenton	Takang
Understanding	Evaluation
Control Prediction	Control
Assessment	Assessment
Improvement	Improvement
	Prediction

Table 2-3 Comparison of Fenton and Takang objectives for software measurements

Fenton's list is a broad overview of software measurement whereas Takang's concentrates solely on maintenance.

Understanding

Evaluating the state of a project is vital in order to build an understanding of whether or not changes to its development are required. This understanding is then used by the software engineer to predict what might occur in the future. Next is to decide what action to take and when to take it. Fenton [Fen96] describes this as setting "baselines" and setting "goals" for future work. For example, one measure of quality might be looking at the number of faults detected.

Control

Control is essentially interpreting the results gained from our measures. Then using this information to predict what is likely to happen in the near future. If this prediction is not favourable then changes can be made to allow the project to meet its set goals. For example, if the number of faults being detected for a specific software module is far greater than anticipated it may be necessary to make personnel changes or offer additional training.

Improvement

Improvement in the way it achieves its gaols is the major aim of any organisation. A high fault level in software may lead to extra training being provided as standard or the introduction of more frequent project evaluations.

2.2.6. Application of Software Measurement for Software Evolution

Evaluation is essential during maintenance because the maintainer may not be the original developer. It is therefore important for the maintainer to build up an understanding of their target system. An important attribute that affects the amount of effort required is size. This is generally measured in lines of code (LOC) it can be expressed as thousands of lines of code (KLOC). Complexity is also an important attribute that must be measured. One such measure is McCabe's cyclamate complexity [McC76]. This is used as an indicator of the psychological complexity of a system. This measurement estimates the relative amount of effort required to understand a section of code.

Maintainers must also decide which tools should be used to complete the task. Control is important to keep track of changes being made to the system and to minimize the amount of new problems introduced. This controlling involves an assessment of the system and whether or not it is economically feasible to carry out the change.

The side effect of this change should be an improvement in the overall quality of the target system. Without the correct measures for quality or productivity it is impossible to assess whether or not a system has been improved. For example, a maintainer may have the aim of simplifying a routine in a program. They can assess whether or not they have been successful by recording the complexity measure of that routine before a change and comparing it with the complexity measure after a change.

Lanza [Lan01] uses a combination of software visualisation and software metrics to present the changes in software systems over a number of releases. This is achieved through an "*evolution matrix*". The matrix depicts the evolution of each class in the

system through a series of versions. Columns within the matrix represent a different version of the software system and the rows contain classes within the system. Each class is represented by a two dimensional box whose width and height correspond to two class level metrics (see figure 2.3).

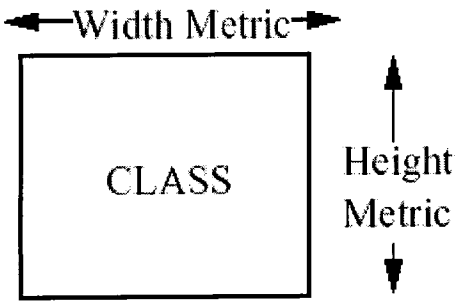


Figure 2-3 Depiction of metrics using matrix [Lan01]

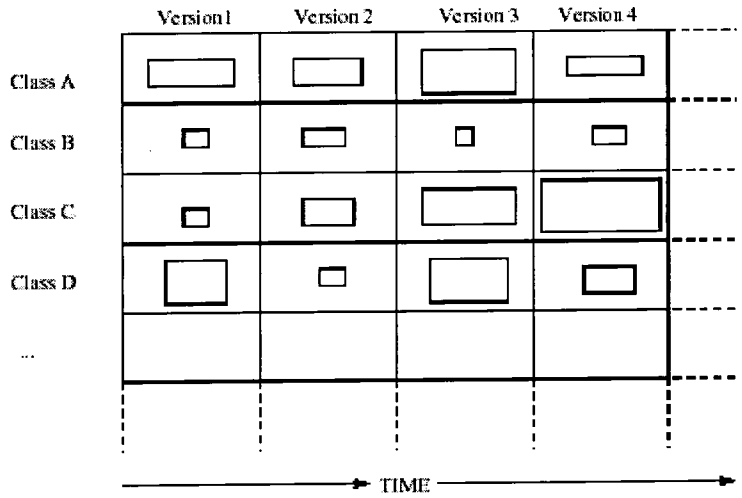


Figure 2-4 Visualisation of system evolution using Lanza's [Lan01] matrices

Lanza's work focuses on the evolution of classes within a system and uses the *number of methods* as the width metric and *number of instance variables* as the height metric. At the system level the evaluation matrix (figure 2.4) displays the following information:

Size of the system – the matrix clearly shows the number of, and size of classes, that comprise the system across the versions.

Addition and removal of classes – empty spaces mean a class has been removed new classes appear at the bottom.

Growth and stagnation phases in the evolution – the shape of the whole matrix is a guide to growth in the system. An increase in height of the matrix indicates growth whereas stagnation is depicted by the height remaining the same.

Lanza [Lan01] also presents a categorisation of classes based on the evolution matrix.

Pulsar – classes that grow and shrink in size repeatedly.

Supernova – classes that suddenly explode in size.

White Dwarf – classes that used to be of significant size but have lost functionality and now have little use.

Red Giant – classes that remain large over several versions.

Stagnant – classes that remain unaltered over several versions.

Dayfly – classes that are created in one version and removed in the next.

Persistent – classes that remain throughout the whole lifetime of the system.

2.3. Chapter Summary

This chapter has reviewed the important aspects of software maintenance. Specifically it has considered the problems in maintaining large legacy systems. It has considered the benefits to maintenance of reverse engineering, visualisation and measurement of aspects of a system. The next chapter will review the issue of code clone detection and identify why this is a critical issue for the maintenance of software applications.

3. Clone Detection

This chapter introduces and describes code cloning and code clone detection. The latter of which is the main focus of this thesis. Current work on these topics have led to the development of a number of techniques and tools to aid maintainers in the identification, presentation and potentially the removal of unwanted, cloned code. This thesis takes into account two distinct applications of clone detection; that of traditional clone detection for software maintenance and plagiarism detection in computer programs. Each of the algorithms used in the different clone detection techniques is described in sections 3.4.

Section	Algorithm
3.4.2	Johnson [Joh94]
3.4.3	Mayrand [May96b]
3.4.4	Baxter [Bax98]
3.4.5	Ducasse [Duc99]
3.4.6	Kamiya [Kam02]
3.4.7	Malpohl [Pre00]

The main distinction between these two forms of clone detection is ethical rather than technical. Students copying and pasting source code in order to cheat, use the same technique as a maintainer copying and pasting in order to save themselves time. One difference however, will usually occur. It is often the case that students will go to extra lengths to alter the copied code in an attempt to disguise their plagiarism.

3.1. Definitions

There is no single authoritative definition of a clone code. This lack of a universal definition makes clone detection difficult and ambiguous. If a piece of code is copied and pasted from one program to another without modification it is fairly obvious. However, this is rarely the case, a more realistic example (especially in plagiarism) is where the code is copied, pasted and then modified in some fashion. Thus the

question has to be asked, when does a clone stop being a clone? In plagiarism if a student copies another student's program and in an attempt to disguise this almost completely re-writes the program can this be considered plagiarism?

Each piece of research into code cloning directs detection according to their own definition. This section will attempt to summarise and compare these varying definitions.

Code duplication or cloning is the process of copying a fragment of code and pasting it to somewhere else. Balazinska [Bal00] describes it as "*manual source code copy and modification*". Clones then can be exact copies or as Mayrand [May96b] puts it "*mutants*" of other existing code fragments. Mutant clones are defined in Baxter [Bax97] as "*near miss clones*" where its definition of a clone is a "*program fragment that [is] identical to another fragment*". The original fragment is defined as an "*idiom*" in the paper and Baxter describes it as a fragment that "*implements a recognizable concept (data structure or computation)*".

Cloning is not just limited to the system where the original idiom was created. Two further categories are introduced in Burd [Bur97]. Firstly "*Replication within Programs*" describes cloning within a single file and "*Replication Across Programs*" identifies cloning from one file to another. Ducasse [Duc99] also examines this occurrence and equates files with "*high duplication ratio between each other*" as "*cloned files*".

Kamiya [Kam02] defines a clone relation as "*an equivalence relation (i.e. reflective, transitive and symmetric relation) on code portions*". This relation holds if (and only if) two portions of code are the same sequences of code. Further definitions are introduced by Kamiya, are that of a *clone pair* (i.e. a pair of code portions that belong to a clone relation). Also there is the notion of a *clone class*, which is a set of maximal sized portions of code that belong to the same clone relation. If three sorting algorithms all originated from the same code sequence they would be considered to be in the same class. This generalised class is comparable to the idiom described by Baxter [Bax97].

Johnson [Joh94] points out that cloning in software production can also occur in the documentation as well.

3.2. Reasons / Motivation for Code Duplication

Computer programs contain logic that aims to solve specific problems. If a programmer comes across a problem with a similar solution to one previously coded the temptation to simply cut and paste the logic from the previous solution can be overwhelming. Ducasse [Duc99] gives some simple explanations for the proliferation of clone duplication. He explains that "m*aking a code fragment is simpler and faster than writing from scratch*" and also "*evaluating the performance of a programmer on how much code he or she produces gives a natural incentive...*" finally another factor is the feeling that "*...the cost of a procedure call or method invocation seem too high a price*".

Organisations' business models evolve and so their software must also evolve to reflect their new environment. As the software evolves and new functionality is required programmers might decide that rather than "*risk breaking a working feature by making a major revision, a programmer might choose to leave the old section of code untouched and to add another slightly modified copy of it for the new feature.*" [Duc99]. This would seem to fit in well with the Mayrand's [May96b] remark that clones are modified to adapt to the new functionality required. He goes on to state that one of the major reasons for this slightly modified cloning is when an organisation "*does not have a good re-use process in place*".

Sometimes there may be valid justification for the duplication of code. Baxter [Bax98] points out that, "s*ystems with tight time constraints are often hand-optimized by replicating frequent computations*". It also highlights that a particular "*coding style*" used for tasks such as "*error reporting or user interface displays*" may form a "*mental macro*" meaning the programmer copy and pastes from memory. Baxter classes these coincidental clones as "*near misses*".

The use of integrated development environments can lead to the inclusion of duplicated code as they have a limited library of code and so clones are likely to occur.

For plagiarism, the motives are quite simple. The student for some reason is either unable or unwilling to complete the programming exercise set within the deadline.

3.3. Side Effects of Code Duplication

One of the major drawbacks to code cloning is the additional understanding required for maintenance. More code "*forces programmers to inspect more code than necessary*" [Bax98]. This is obviously more time consuming and causes greater cognitive strain. Burd [Bur97] illustrates the scale of the problem by identifying that in one particular COBOL legacy system up to a 50% reduction in SECTION's size was possible by removing clones where it existed.

The extra effort caused by replication is especially wasteful as the logic behind the code clones is almost identical, the only difference being cosmetic. Mayrand [May96b] gives a concrete example of the increase in resources required to store the extra code size and thus increased operational costs. The example given is the necessity to purchase new network cards when software becomes too large.

When code is copied and pasted systematic renaming of variables can lead to "*unintended aliasing, resulting in latent bugs*" [Joh94]. Johnson also establishes the fact that cloning is a form of "*software ageing*" or "*hardening of the arteries*" and this ageing process means "*even small design changes become very difficult to make*".

Errors found or changes made in a cloned function require alteration to all other clones throughout the program, "*when enhancements or bug fixes are done on one instance of the duplicated code it may be necessary to find other instances in order to perform the corresponding modification*" [Kom01]. In addition, the very presence of code duplicates indicates that the designers have not identified an important procedural abstraction and suggests that there are design flaws within the system.

3.4. Clone detection techniques

There are a number of techniques available for code duplication/clone detection. This section will describe in detail several of them and relate them to their corresponding tools. The various techniques will now be detailed in publication date order.

3.4.1. Baker Algorithm

Baker [Bak92] uses string matching to detect cloned lines in software. However the system not limited to only exact string matches. It can also detect near miss clones where there has been a "*systematic change of parameters such as identifiers and constants*".

For example

```
for (int i=0;i<=limit;i++)
{
      print "i. " + Person[i]
}
```

```
for (int j=0;j<=end;j++)
{
      print "j. " + Car[j]
}
```

Figure 3-1 Two similar code sections

Figure 3.1 shows two code sections that would be considered a "*parameterized match*" as the variables *i*, limit and Person have simply replaced with j, end and Car respectively. Baker introduces the notion of a parameterized strings or "*p-strings*". These are strings over two alphabets one of constant symbols and the other with parameter symbols. Two p-strings match ("*p-match*") if they are equal except for a one-to-one mapping of the parameter symbols.

axbxyazyx
aubuvaxvu

Figure 3-2 P match between two strings

Figure 3.2 shows a p-match where *x*, *y* and *z* in the first p-string map directly onto *u*, *v* and *x* in the second p-string. In order to confirm a p-match a parameterised suffix tree "*p-suffix tree*" is used in the paper [Bak93] as opposed to a standard suffix tree in previous work [Bak92]. To establish if a pattern p-string P contains a p-match in text p-string T it takes *O (m+n)* time and *O (n)* space (where m, n are the lengths of P and T).

3.4.2. Johnson Algorithm

Johnson [Joh94] treats source code purely as a text based document and by doing this provides a language independent approach to clone detection. Clones are detected by substring matching and the approach taken is as follows:

1. Text-to-text source transformation is carried out to remove characters that are not wanted in the matching process. There are a number of different types of transformations presented including:
    a. Remove all white space characters, carriage return, space, line feed and tab. This means the resulting matches are not layout sensitive.
    b. Remove all white space characters except for line separators. Produces similar results to 1.a but line layout is preserved.
    c. Replace all chains of white spaces with a single blank. This means matches are returned if spaces are in the same position in the text.
    d. Remove all comments
    e. Retain only comments
    f. Replace identifiers with identifier marker.
    g. Mix of the above.
2. Generate substring candidates that cover the whole source. The resulting collection of substrings will be checked for matches. Ensuring the correct number of substrings is generated is crucial; too many and performance will be hindered; too few results in matches being totally missed.
3. Identify raw substring matches involves a simple "*sorting a file containing the content of the substring and an indication of its origin*".
4. Transformation of the database of matches into a more concise description requires that a new set of substrings with minimal overlapping be generated. "*This set has the minimum number of substrings and each substring is of maximum length*".
5. Performing task-specific data reduction will obviously vary for each particular task.
6. Presentation of the high level data can be by report generation or some form of visualisation. (See section 3.7.1). The example given in his paper is that of a graph where vertices are files and an edge represents a match between a pair of files.

3.4.3. Mayrand Algorithm

Mayrand [May96b] advocates the use of software metrics to describe a file and then compare to the results. His paper presents a technique for comparing cloned functions not fragments of code. Mayrand et al devised a list of 21 metrics grouped into four "*points of comparison*". Metrics were extracted using a tool called Datrix, developed by Bell Canada. The four points of comparison and their metrics are as follows:

1. The name - this simply compares the names of each function to establish whether or not they are equal
2. Layout – this contains metrics about attributes such as number of non-blank lines, number of logical comments
3. Expressions – covers metrics such as number of declaration statements, total calls to other functions
4. Control flow – covers metrics such as number of loops, number of control statements, average nesting level etc…

Two functions are equal if all the metrics within that group are equal. Functions are considered similar if the absolute difference is equal or below a set threshold defined for each metric within that group. If two functions are neither equal nor similar then they are considered distinct.

Once two functions have been compared Mayrand et al provide an ordinal "*clones identification scale*" ranging from 1 to 8 to describe classify cloning (table 3.1).

1	ExactCopy	EqualName & EqualLayout & EqualExpression & EqualControlFlow
2	DistinctName	DistinctName & EqualLayout & EqualExpression & EqualControlFlow
3	SimilarLayout	SimilarLayout & EqualExpression & EqualControlFlow
4	DistinctLayout	DistinctLayout & & EqualExpression & EqualControlFlow
5	SimilarExpression	SimilarExpression & EqualControlFlow

6	DistinctExpression	DistinctExpression & EqualControlFlow
7	SimilarControlFlow	SimilarControlFlow
8	DistinctControlFlow	DistinctControlFlow

Table 3-1 Mayrand's level of cloning

In order to detect functional cloning it is necessary to compare every function with every other function in the system(s) under consideration. The functions are firstly tested at scale 1 (ExactCopy) if this returns false then the scale 2 DistinctName is carried out, this process carries on up to the scale 8. "*The effort required for testing each pair is approximately 500 mathematical operations*".

3.4.4. Baxter Algorithm

Baxter [Bax98] looks at software at the syntactic level to produce Abstract Syntax Tree (AST) representations. Building an abstract view of the system's logic allows "*the discovery of code fragments that compute the '**same**' result*". This is obviously different from the text-based approach taken by Johnson [Joh94] and Baker [Bak92]. Parsing the software allows the production of an AST for the source code. After this, a series of three algorithms are applied to the AST. Firstly a "*basic*" algorithm looks for sub-tree clones, next comes the "*sequence detection algorithm*" which looks for variable sized sequences of sub-tree clones. Finally, the third algorithm attempts to find near miss clones by generalising combinations of other clones.

Scale is a major problem, for an AST of N nodes comparison is of the order O (N^3) and the paper states that a system with M lines of code will means N=10*M. The computation required is even greater for the second algorithm O (N^4). Hashing is used to reduce the amount of computation required. As mentioned the sub-trees are compared by looking at their similarity, using the formula:

Similarity = 2 x S / (2 x S + L + R)

Where:

S = #shared nodes

L = #different nodes in sub-tree 1

R = #different nodes in sub-tree 2

3.4.5. Ducasse Algorithm

Ducasse [Duc99] puts a great deal of emphasis on the language independence of the clone detection technique it uses. It addresses three basic issues of clone detection; algorithms, visualisation and pattern matching. Clone detection is carried out by first performing a simple transformation on the source code (removing all white space and comments) to ensure that the approach would not be language dependent there is no conversion to a more abstract level. All transformations are "*within the realm of string manipulation*". Removing all white space and commenting, reduces the source code into "*an ordered collection of **effective** lines*". Each transformed line is compared with every other line and the result is saved in a comparison matrix (false, if not equal, else true). The coordinates of the source lines give the coordinates in the matrix where the result is saved. The search space for n lines of source is large (O (n^2)) so as with [Bax98] hashing is used to reduce computation required as the same line is stored in the same bucket or location in the hash table. There are two possible options when using the results gained from the comparison. Firstly visualisation is an option, the most obvious being dot plotting (plotting the lines of code side by side and marking lines that are equal with a dot, see Section 3.7.1). Secondly a pattern matcher can look for broken diagonals in the matrix, this indicates a clone sequence that has been altered. This produces a textual report providing a useful representation of the cloning in the system.

3.4.6. Kamiya Algorithm

Kamiya [Kam02] tokenises the source code into a single token stream and then uses a suffix-tree matching algorithm to detect similarity in the token stream. The whole clone detection process consists for four stages.

1. Lexical Analysis – each line of all the source files is tokenised and concatenated into a single token stream. White space is stripped but the tab, carriage return and comment characters are stored and sent to the formatting stage in order to allow the reconstruction of the original line numbering.
2. Transformation – there are two sub-processes that transform the token stream. The first sub-process uses predefined rules to standardise the token stream (such as the removal of package names and the removal of accessibility keywords). Following this the next sub-process is parameter replacement. Each identifier is replaced by a special token (making sure simple variable name changes will not fool the algorithm).

3. Match Detection – Taking all the substrings of the sequence clone pairs are detected. Each pair is recorded as the start and end indices of the two substrings involved in the match (called Left and Right).
4. Formatting – the location of each clone pair is converted back to the original location in the source code.

3.4.7. Malpohl Algorithm

Malpohl [Pre00] in his tool JPlag uses the Greedy String Tiling Algorithm to detect similarities within source code. It tokenises the source and compares pairs of token streams by attempting to cover one pair with substrings or "tiles" from the other. Similarity is measured by the percentage of token streams that can be covered in this manner. The first stage in this process is to tokenise the strings. JPlag uses a parser or scanner depending on the language to enable more semantic information to be extracted. Malpohl gives an example of generating a **BEGINMETHOD** as opposed to simply an **OPEN_BRACE**. This enhanced token set provides a more detailed description of the "*essence of a program*" and so it is harder for plagiarists to fool. JPlag ignores comments and white spacing, as these are the most likely targets for disguising copied code by students.

The algorithm itself compares two Strings **A** and **B** and consists of two phases. It attempts to find a set of substrings that are equal and follow the following rules:

1. Any token of A may only be matched with exactly one token in B.
2. Searching for substrings is done independent of their position within the string this ensures that simple reordering will not fool the algorithm.
3. Long substrings are preferred to shorter substrings this is because shorter matches are more likely to be "spurious". This is enforced with a minimum match threshold.

Phase one searches for the longest contiguous matches. Three nested loops are used the outer loop iterates over every token in String A. The second loop then compares each token with ever token in String B. Finally, the third loop tries to extend the match as far as possible (i.e. while the string tokens are equal and have not been matched before).

Phase two marks strings of maximal length to prevent them from being used again for matches in phase one in further iterations. This ensures that every token is used in only one match and hence satisfies the first rule mentioned previously. Marking strings and hence reducing the number of potential matches ensures that the algorithm terminates.

3.5. Comparison of techniques

Author	Level	Transformed Code	Comparison Technique
[Joh94]	Lexical	Substrings	String-Matching
[Duc99]	Lexical	Normalised substrings	String-Matching
[Kam02]	Lexical	Tokenised strings	String-Matching
[Bak92]	Syntactic	Parameterised strings	String-Matching
[Pre00]	Syntactic	Tokenised strings	String-Matching
[May96b]	Syntactic	Metric Tuples	Discrete Comparison
[Bax98]	Syntactic	AST	Tree-Matching

Table 3-2 Different clone approaches based on [DucLecture]

Table 3.2 above gives a comparison of the different techniques used to detect clones. Johnson [Joh94] and Ducasse [Duc99] are grouped together because they are language independent whilst the others rely on language specific parsing.

3.6. Language Independent vs. Language Dependent

Language dependent techniques can behave quite intelligently; by looking at the logic of system they cannot be 'fooled' by elementary cosmetic changes, however, there are a number of problems related to their use.

- Legacy systems can be written in a dialect of a language and thus a parser for that specific dialect needs to be used (which may be hard to find / may need to be rebuilt)
- Building a system graph takes a large amount of computation
- Multiple programming languages may have been used in the system

Language independent approaches view source code as just another form of documentation "*and analyse it the way other documents are analysed*" [Joh94]. This

is justified as it is the way maintainers and developers view software. Drawbacks to this approach include;

- Systematic variable renaming will totally 'fool' the system
- Non contiguous clones are missed

3.7. Current Tools

There are several tools that have been developed to aid with the identification of code duplication. Below is a list of the main utilities available and a brief description.

- DUPLOC
    - DUPLOC is the language independent tool used in Ducasse [Duc99]. It uses a two-step approach to the detection of clones. Firstly a transformation is carried out that "*reduces the entire file to an ordered collection of effective lines*". These effective lines are ones with all spaces stripped and comments removed. Finally the tool uses simple string matching to compare and locate clones. DUPLOC provides textual reporting of matches and also a dot plot visualisation (see section 3.7.1).
- MOSS
    - The Measure Of Software Similarity (MOSS) is a tool designed to detect plagiarism in software code written by students. It is an online service provided by the Berkeley University in the US. A perl script is used to submit program files. Details are not available on the algorithm used with the following explanation provided. "*While there is a big difference between a good cheating detection algorithm and a bad one, all such algorithms can be fooled if one knows how they work. It's best if we don't say too much here about the ideas behind Moss*" [MossHome].
- CloneDr
    - CloneDr is a commercial package from Semantic Designs presented in Baxter [Bax98]. It is a language dependent tool that transforms the source code into ASTs. Once the code has been parsed, transformed into an AST, three algorithms are used. Algorithm one is to find sub tree clones. Two is concerned with "*the detection of variable-size sequences of sub-tree clones, and is used essentially to detect*

statement and declaration sequence clones." Thirdly "*more complex near-miss clones*" are looked at by generalising combinations of other clones.

- Datrix
  - Datrix itself does not provide clone detection. However this product is a language dependent tool that can produce software metrics from source code. Mayrand [May96b] presents a "*technique to automatically identify duplicate functions in a large software system.*" This technique provides an ordinal scale of similarity of code clones from ExactCopy to DistinctControlFlow. These are worked out by using four "*points of comparison*" each point has a set of associated metrics which have been generated by Datrix.
- CodeSurfer
  - CodeSurfer is another language dependent tool spawned out of academic work. It is produced by GrammaTech and is the implementation of the technique presented in Komondoor [Kom01]. The technique transforms the program into a Program Dependence Graph (PDG) and uses program slicing to identify "*Isomorphic sub graphs*" which correspond to similar logic and hence clones. The major advantage cited by Komondoor [Kom01] is "*our tool can find non-contiguous clones*" the slicing process filters out segments of code that may have been added. Also line reordering does not confuse the tool.
- DUP
  - DUP uses parameterised strings (p-strings) to detect cloned strings (a p-match). Baker [Bak93] explains that "*each occurrence of **first**, **last**, **0**, and **fun** in one section may be replaced by **init**, **final**, **1**, and **g**, respectively, in the other section;*" Baker [Bak93] also provides a formal definition of a p-string, "*Two parameterized strings are a **parameterized match**, or **p-match**, if they are the same except for a one-to-one correspondence between the parameter symbols occurring in them*"
- JPlag
  - JPlag [Pre00] uses tokenised substring matching to determine similarity in source code. Its specific purpose like MOSS is to detect

plagiarism within academic institutions. Firstly the source code is translated into tokens (this requires a language dependent process). JPlag aims to tokenise source code in such a way that the "essence" of a program is captured and hence is effective for catching plagiarism. Once converted the tokenised strings are compared to detect the percentage of matching tokens, which is used as a similarity value. JPlag is an online service freely available to academia.

- CCFinder
    - CCFinder aims to have "industrial-size strength" with a limited amount of language dependence. It transforms the source code into tokens. CCFinder aims to identify "*portions of interest (but syntactically not exactly identical structures)*". After the string is tokenised a suffix tree algorithm is used to detect matches. CCFinder also provides a dot plotting visualisation tool this allows visual recognition of matched within large amounts of code.

3.7.1. Dot Plotting

Within the DUPLOC tool [Duc99] and CCFinder's Gemini add-on [Kam02], a dot plot visualisation, is provided to aid the identification of duplicated lines of code. DUPLOC uses a comparison matrix to examine each line of one program with another. (Figure 3.3)

Source Lines	a	b	c	d
a	O			
b		O		
c			O	
d				O

Figure 3-3 String matching using dot plot

It is a straightforward process to show such a matrix as a dot/scatter plot. Ducasse [Duc99] cites Church [Chu93] and gives several meaningful patterns that are immediately recognisable from a dot plot.

a. Diagonal lines of dots = copied sequences of code. (Fig 3.4a)
b. Sequences with holes = copied code with partial modification. (Fig 3.4b)
c. Broken Sequences with a shift in the lower half = new portions of code. (Fig 3.4c)
d. Rectangle formation of dots = reoccurring sections of code. An example given is that of the C command 'break'; within a switch statement. (Fig 3.4d)

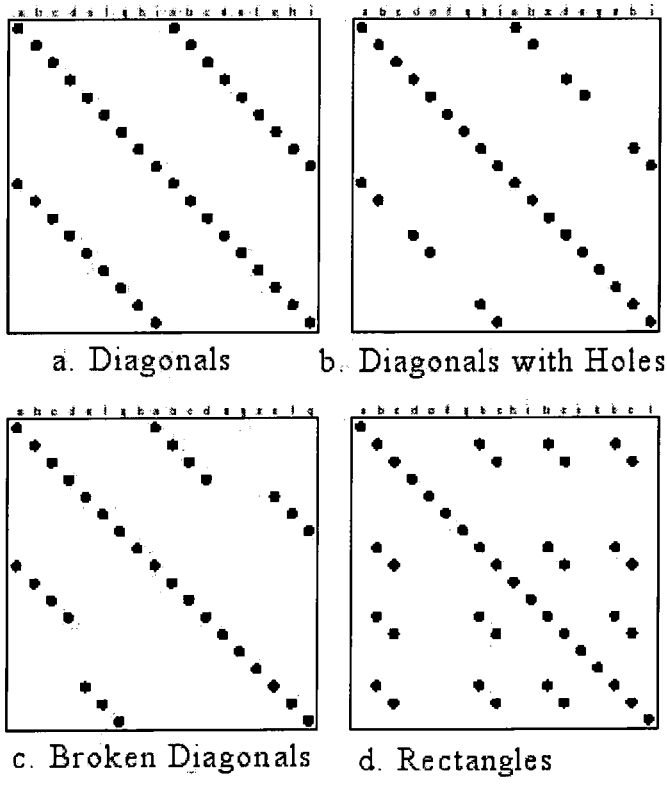


Figure 3-4 Dot sequence patterns (taken from [Duc99])

In order to reduce trivial matches (or "*noise*" as it is referred to by Ducasse [Duc99]) on screen such as "*int i;*" DUPLOC has incorporated a filtering process. Before the comparison a sweep is performed on the matrix to remove single dots.

One of the main benefits offered for the dot plotting visualisation or textual reporting is that it allows "*an exploratory approach to the investigation of the duplication*". Patterns in the dots attract the eye instantly and can lead to unexpected discoveries.

Whereas pattern matching (used within the text-based reporting) will simply catch "*preprogrammed, known configurations*".

By applying the predefined patterns stated earlier (in particular that of fig. 3.4d) the process of software evolution can be easily seen. Comparisons of different software versions reveal how the software has evolved. Broken diagonals progressively shifting to the right indicate where code has been added. Downward shifts in the diagram point to where code has been removed. The scale of such additions and removals are obvious because of the one-to-one relationship between a line of code and a matrix coordinate.

3.8. After Clones have been detected

Once a legacy system has been searched and code clones identified then it is important to consider what can / should be done with them. Unless the clones are intentional to improve performance as mentioned in Baxter [Bax98] then a change to the system is required. Several strategies have been implemented:

- Baxter [Bax98] describes a process where macros are generated that abstract each clone. This is justified by the argument that the "*act of copying indicates the programmer's intent to reuse the implementation of some abstraction*".

- Burd [Bur97] suggests that by looking at the "*degree to which the fragment of code is used*" can reveal whether or not the clone fragment is viable for reuse reengineering.

- Mayrand [May96b] considers code cloning from purely a maintenance viewpoint, "*the goal of the cloning reduction action plan is to increase the maintainability of a system*". It does not give a definite post-detection strategy. "*The selection of a specific technique will be based on the nature of the cloning between functions*".

- Komodoor [Kom01] takes a similar approach to Baxter [Bax98] suggesting the extraction of clones in order to create procedures that can be called in replacement stating that a "*good clone is one that is meaningful as a separate*

procedure (functionally) and that can be extracted out easily without changing program semantics".

- Balazinska [Bal99] presents a methodology for the "*process restructuring actions based on clone detection*" this process involves factorising common aspects of the clones identified and then disassociating the clones with their specific purpose and hence creating more general reuse candidates.

- Kamiya [Kam02] has devised a set of metrics that are used to quantify cloning and answer questions about the frequency and distribution of clones within the system.

    - **Length;** *LEN (p)* and *LEN(C)* this metric gives the length (which could be measured in tokens or lines of code) of either a portion of code ***p*** or ***C*** which looks at the maximal code portion within a clone class.
    - **Population of a Clone Class;** *POP(C)* measures the number of code portions within a clone class. The higher the value the more copies within the system.
    - **Deflation by a Clone Class;** *DFL(C)* using LEN and POP to estimate the amount of code that can be removed from the system by the extraction of a particular clone class. This estimation is computed by the following equation:

      ```
      DFL(C) = LEN(C) * POP(C) - (USELEN(C) * POP(C)
      + LEN(C))
      ```

      where USELEN(C) is the length of a method call for a generalized method which has been introduced to replace the functionality of the clone code portions.
    - **Coverage of Clone Code;** *COVERAGE (% LOC), COVERAGE (% FILE)* COVERAGE (% LOC) is the percentage of lines that include any portion of clone, and COVERAGE (% FILE) is the percentage of files that include any clones.
    - **Radius;** *RAD(C) RAD (p)* measures the longest path (in the directory structure) from each file containing a clone portion belonging to class C to the lowest common ancestor directory containing all the files. If

all clones appear in a single file then RAD = 0. This metric tells maintainers the extent of influence a clone class has on a system. Higher RAD values equates to a wide spread of clones.

3.9. Incorporating clone detection into development

Mayrand [May96a], [May96b] describes a "*cloning reduction plan*". This plan first addresses prevention and then reduction of cloning within a system by the following steps:

1. Establishing a multi-version control system that will provide sufficient metrics to monitor change in the system. This will enable maintainers to estimate the impact of change to the development process.
2. Review design and programming guidelines with specific attention paid to the issue of cloning. Obviously programmers should be discouraged from copy and pasting and designers are encouraged to look for commonality in the design which could then be abstracted.
3. Employ a clone tsar charged with the task of monitoring cloning within the system. Mayrand recommends in [May96a] that this person be a system architect "*since clone removal has a lot to do with the architecture and the libraries of the system.*"

These three steps should ensure that the introduction of new clones to the system is under control. After this Mayrand then goes on the list the steps required for clone reduction of the existing system.

4. Target the clones to be removed from the system. Areas of the system that are heavily maintained should be prioritised.

3.10. Metrics for clone detection

When attempting to discover a cloning relationship between functions Mayrand [May96b] defined four attributes or "*points of view*". Mayrand looked at the *name of a function, its layout, the expressions* used within the function and finally *its control*

flow. These points of view are used to compare two functions. Three possible outcomes were "*similar, equal or distinct*" functions.

Each point of view contained a set of metrics. A delta or threshold was assigned to each metric. For example, two functions were considered to have equal layout if all the metrics within that point of view were equal. If the absolute difference between each and every metric was not zero but within its set individual threshold then the two functions were considered to have similar layout. If neither is the case then the two functions had distinct layouts. The metrics used to examine layout are listed in Table 3.3.

Metric Description	Delta
Declaration comments volume	10
Control comments volume	10
Number of logical comments	5
Number of none blank lines	5
Avg. name length of variables	2

Table 3-3 Datrix layout metrics

3.10.1. CodeCrawler and Moose (Members of the FAMOOS Project)

Source Code Assessment and Presentation of Results

CodeCrawler [Duc01] is a language independent tool designed to support reverse engineering by using a combination of metrics and visualisation features. CodeCrawler was built as an add-on for the Moose re-engineering environment [Duc01]. Moose handles source code assessment and provides the metrics "services" (see figure 3.5) for CodeCrawler to visualise.

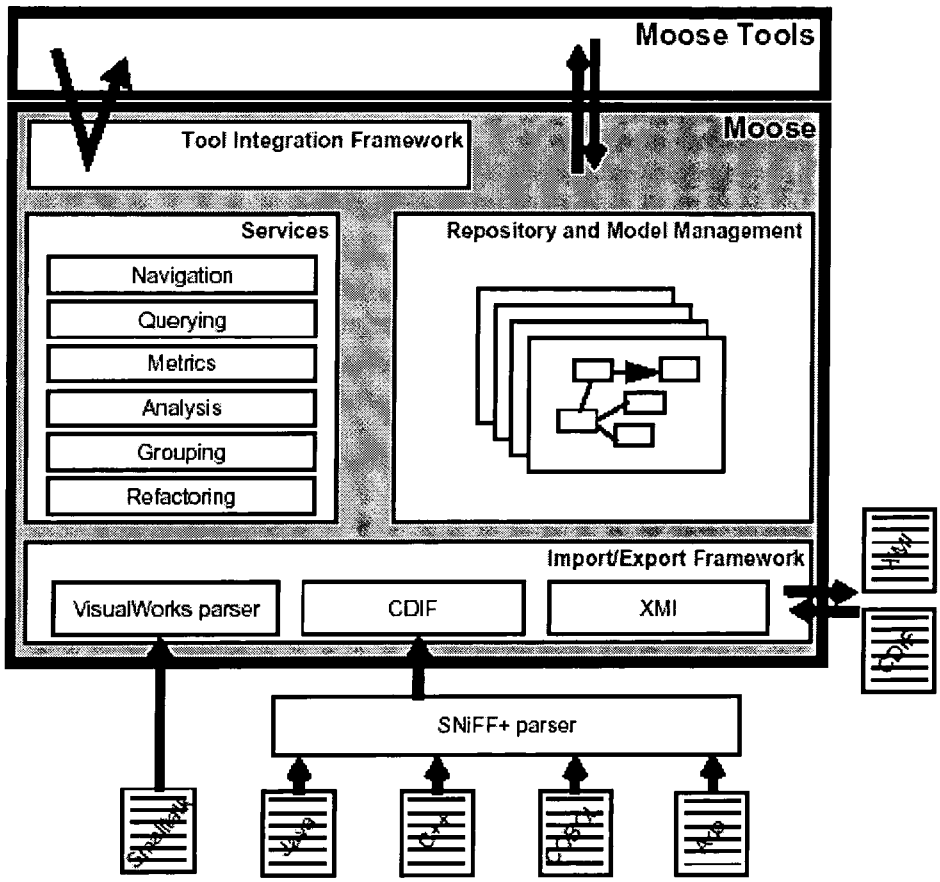


Figure 3-5 MOOSE architecture [Duc01]

Figure 3.5 shows how source code is imported into Moose. SmallTalk source is parsed by the VisualWorks parser which is available because Moose (and CodeCrawler) is written in SmallTalk. Java, C++, Cobol and Ada are parsed using parsers such as SNiFF+ into the interchange format CDIF [CDF97]. Source code written in other programming languages can be imported if they are transformed into CDIF also XMI [XMI]. These formats are industry standards and so allow Moose to include systems written in any language.

CodeCrawler uses a set of simple metrics and simple visualisation techniques and it is then possible to gain an overview of the evolution of large-scale systems. Boxes represent classes, the dimensions and colour of the boxes can be set to represent various class metrics. Metrics are divided into four categories; complexity, coupling, cohesion and inheritance tree metrics. Listed below are a selection taken directly from the FAMOOS Handbook [Bär99].

Complexity Metrics

Abbreviation	Description
LOC	*Lines of Code*: Measures the size of a class by counting its lines of code.
WMC	*Weighted Method Count*: Measures the complexity of a class by adding up the complexities of the methods defined in a class.
NOM	*Number of Methods*: Measures the complexity of a class by counting the number of methods defined in that class.

Table 3-4 Complexity Metrics used in MOOSE

Table 3.4 describes the three complexity metrics used within the MOOSE system.

Coupling Metrics

Abbreviation	Description
DAC	*Data Abstract Coupling*: Measures coupling between classes resulting from attribute declarations.
RFC	*Response Set for a Class*: Measures complexity and coupling properties of a class by evaluating the size of the response set of the class.

Table 3-5 Coupling Metrics used in MOOSE

Table 3.5 shows the Coupling metrics used within the MOOSE system.

Cohesion Metrics

Abbreviation	Description
TCC	*Tight Class Cohesion*: Measures the cohesion of a class as the relative number of directly connected methods. Methods are considered connected if they share at least one instance variable.

Table 3-6 Cohesion Metrics used in MOOSE

Table 3.6 shows the Cohesion metric used within the MOOSE system.

Inheritance Tree Metrics

Abbreviation	Description
DIT	*Depth in Inheritance Tree* - Measures the depth of a class in the system's inheritance tree.
NOC	*Number Of Children* - Counts the number of children (direct subclasses) of a class.
NOD	*Number Of Descendants* - Counts the number of descendants (direct and indirect subclasses) of a class.

Table 3-7 Inheritance Tree Metrics used in MOOSE

Table 3.7 shoes the Inheritance Tree metrics used within the MOOSE system.

Justification for metrics chosen

The metrics listed here are used to diagnose object-oriented legacy systems with a view to re-engineering. Complexity metrics allow a re-engineer to estimate how much effort it would take to understand or modify a particular section of a system (in this case a class). Since complexity cannot be measured directly, metrics that can be used to infer it are required. Obviously the greater the complexity of a module or class the greater the effort required understanding and re-engineering it. Coupling occurs between classes if classes depend, or are aware of, another class. For example, if one class invokes another's method or accesses its variables. Re-engineers are concerned with coupling because if two classes are tightly coupled then changes made to one can have repercussions to the other class. Also classes with high coupling usually form a key part of a system and therefore are an entry point into re-engineering. Cohesion describes how closely attributes within a class are related. For example, how many methods access the same variables, or invoke other methods within the class. Ideally classes should have high cohesion as it indicates a good design encapsulating related concepts. According to Bär [Bär99] "*Classes with low cohesion often represent violations to a flexible, extensible or reusable design*". Inheritance is a key concept within object-orientated design. It allows designers to describe relationships between classes with similar behaviours and to reuse common aspects of their design. Measuring the inheritance allows engineers to focus on a special type of coupling. Bär cites the example of classes with low DIT and high NOC (or NOD) values affecting lots of classes because they are super classes to the

child classes. Changes in the super class will probably require changes in many child classes.

3.10.2. Datrix

Source Code Assessment and Presentation of Results

Datrix is a source code analyser and one of its outputs is a set of metrics. As this is the only feature relevant to this study, this section will only give a broad overview to the aspects of Datrix which are not directly related to software metrics. Following this a detailed examination of the metrics produced by Datrix is provided.

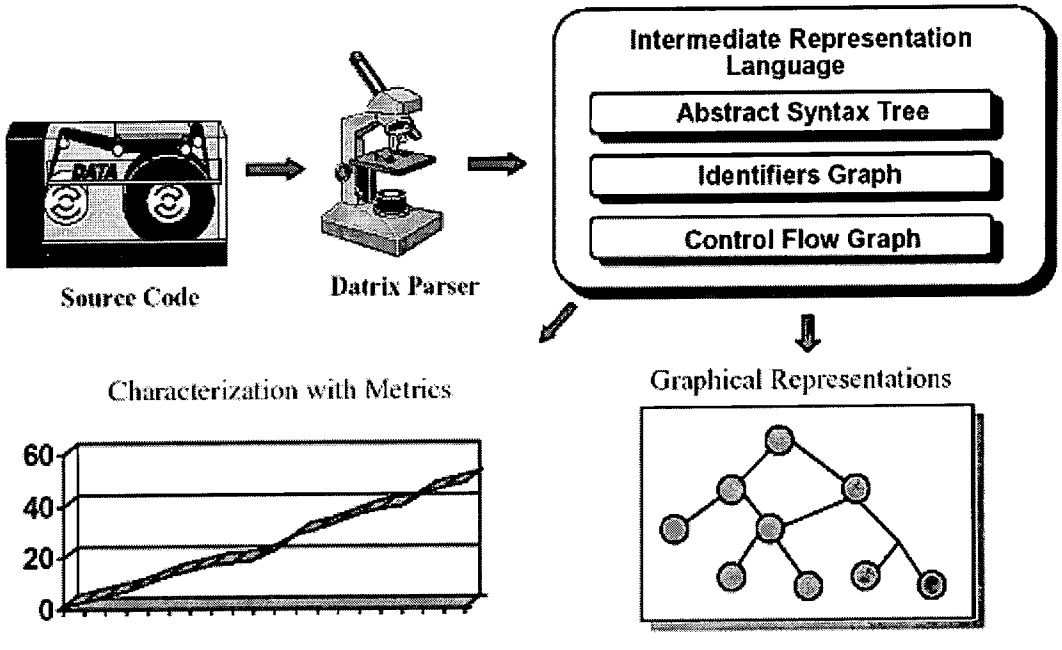


Figure 3-6 Datrix Source Code Assessment [Lag97]

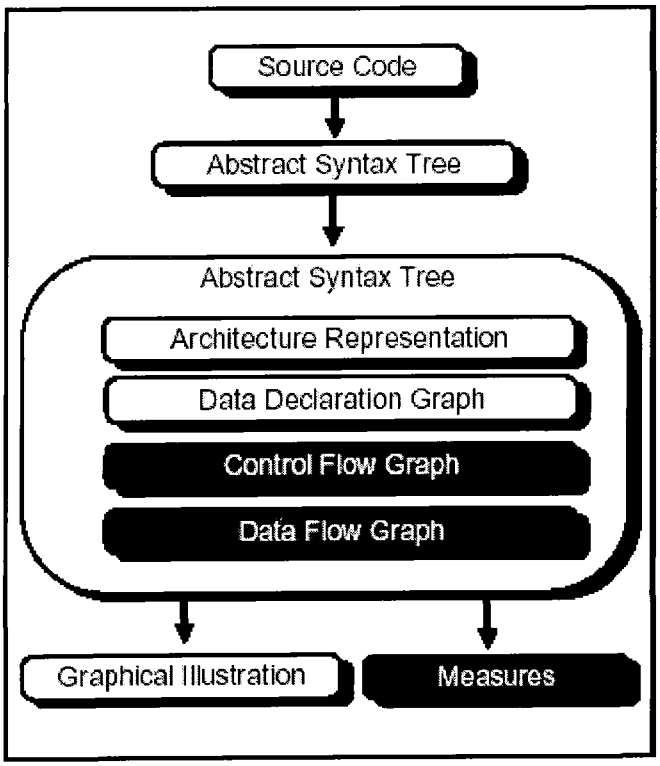


Figure 3-7 Source code abstraction process in Datrix

Figure 3.6 shows how Datrix [May96b] takes source code as its input and converts this into either graphical representations or metrics. Figure 3.7 shows in more detail the process of abstraction performed on the source code. Initially Datrix parses the source code and generates an Abstract Syntax Tree representation. The next abstraction translates the Abstract Syntax Tree into the Intermediate Language Representation. All the necessary information to produce software metrics is contained within this 2nd level of abstraction. Figure 3.7 shows that the "measures" or metrics are generated by a further transformation on the intermediate language representation.

Unlike CodeCrawler there is no visualisation feature. Results are presented in a simple summary format and can be stored in files.

Datrix Metrics

Datrix organises software metrics into three domains each domain corresponding to the scope of the metrics contained within them. Routine Metrics provide metrics at a routine or function level. Class Metrics are metrics that summarise a whole class.

File Metrics are metrics that look at a whole file. Within these domains the metrics are split into statement families.

- **Declarative statements** are those were a variable, object, parameter or type are declared.
- **Control-flow statements** are "statements that can alter the flow at intra-procedural level" [DatrixManual]. (i.e. decision statements such as if or switch, jump statements such as break, continue or goto and loop statements such as for, while and repeat).
- **Executable statements** are any expression other than declarative or control-flow statements within a function scope (i.e. '{' and ')'). Also executable statements are any statement, other than either a declarative or control-flow statement, that is separated by a statement-separator (such as a comma). Or any expression, "*other than a declarative or empty statement, that is stated in the initialisation or incrementation parts of a for-statement*" [DatrixManual].

Each metric produced is given an abbreviated name and these names follow the following rules:

The metric abbreviation (MetricAbbreviation) is formed by taking the domain of the metric (MetDomain), then the metric description (MetDescription) and finally the quantifier (MetQuantifier). i.e. MetricAbbreviation : MetDomain MetDescription MetQuantifier

MetDomain represents the metrics domain and as consist of:

- Cla : Class domain
- Rtn : Routine domain
- Fil : File domain

MetDescription describes the element that is being measured:

- Lns : Lines of code
- Scp : Scope
- ScpNstLvl : Scope nesting level

MetQualifier represents the "mathematical nature of the metric" [DatrixManual] and is one of the following:

- Nbr : Simple count
- Sum : Sum
- Avg : Average
- Rto : Ratio
- Wgt : Weighted metric value
- Len : Length
- Max : Maximum value
- Vol : Volume

This chapter reviewed the current state of the art in code clone detection. The review shows that there appears to be no universal agreement of what is considered a clone. It found that cloning presents a number of problems to a software maintainer including the addition of redundant code, the potential proliferation of errors throughout a software system. It focused on certain clone detection techniques and the tools that implement these techniques. The concept of a clone reduction program is also discussed with ideas for further application of clone detection in documentation.

4. Method

As mentioned in the introduction, several software case studies will be used as test subjects for a set of clone detection tools. In this chapter the method used for these case studies will be described. The rest of the method chapter is as follows. Firstly an overview of the method applied to each of the case studies. This will be a high level description of the method. Following this the hypotheses chosen are presented and described. The criteria for choosing each of the case studies is then given after this it gives the criteria for choosing the clone detection tools. After this an explanation of the manual verification of clones describes how potential clones identified by a clone detection tool were verified as an actual clones. Next, an outline of the qualitative evaluation of each of the clone detection tools used in the case studies. Finally brief descriptions of the experiments used to test the hypotheses chosen. This section will also introduce a novel visualisation technique for comparing the clones identified by different clone detection tools.

4.1. Overview of method

A selection of case studies will be carried out. Within each case study the same clone detection tools will be used to identify a maximal set of clones. Clones identified by the tools, are potential clones (because the clone detection tools can identify false positives). Every clone identified will be manually verified and if they are considered to be clones then will be marked as actual clones and added to a total base set of actual clones for that case study. As these case studies were taken "as is" and thus there can be no assurance that a variety of clone types were present.

This total actual clone base set will contain clones identified from all of the tools. This base set is considered to contain the maximal number of clones for that case study and will be used to determine precision and recall values for each of the tools. However, it is important to note that this base set does not contain the total number of clones possible. It is possible that some clones were missed by all of the tools.

As well as adding each tool's set of clones to the base set their clones will be compared to determine the intersection between each the sets of clones identified by

each clone detection tool (i.e. to determine the similarity of the clones identified by each tool).

Further analysis on the types of clones each tool identifies will be carried out. Aspects such are average size of clones identified by each clone detection tool will provide useful information as whether one tool identifies larger clones on average than another.

Finally a qualitative evaluation of the tools used in each of the case studies will provide information that will be of interest to maintainer such as customisation options and level of user support.

4.2. Hypotheses

During each of the case studies a series of hypotheses will be evaluated. These hypotheses are based partly on questions raised in the current literature and partly on testing aspects of cloning of interest to software maintenance. They are as follows:

1. Each clone detection tool will identify different proportions of cloning for the same case study.
2. Case studies of differing size and development background will identify different proportions of cloning for the same clone detection tool.
3. Case studies developed without the aid of an integrated development environment will contain on average clones which are greater in size.
4. Clones identified using metric comparisons will differ greatly in size from clones identified by tools that directly compare source code.
5. Case studies developed with the aid of automatic code generation will produce more clone classes.
6. Replication across programs is more prevalent than replication within programs.
7. Each clone detection tool will identify different sets of clones.
8. No clone detection tool will find every clone within a case study.
9. No clone detection tool can achieve 100% precision for every case study.

10. The proportion of cloning identified by clone detection tools within a case study is very dependent of the tools minimum size threshold

4.2.1. Hypothesis Justification

1. It was considered, because of the different approaches to clone detection (and indeed to the very nature of cloning) that each tool adopted, the the amount of cloning identified by different tools would vary.
2. Different coding styles it was felt would produce differing levels of cloning. For example, the use of automatically generated code. Development environments with less experience would be more likely to take shortcuts.
3. The use of an integrated development environment would allow the insertion of ready made clones for simple tasks. These clones do not necessarily contain any complicated logic but are usually short functions for initialising GUI components.
4. As the metrics based clone detection tool was focused entirely on cloning between whole functions it is artificially restricted in the potential size of each clone. For example, a whole program could be copied but this would be detected as a series of shorter clones by Covet.
5. Automated code generation copies from a library of code into a program's source this is in effect copy and pasting. This process will increase the amount of clones within a piece of software.
6. It was considered that the potential for replication across programs was greater than within the same program as programmers would often copy and paste from other programs because of the extra complication of including the other program. A function within in the same program would be easier to call.

7. Each clone detection tool's concept of a clone is different so their detection algorithm will detect different portions of code as clones. Further more, the restrictions on academic tools to only identify replication across programs means their clones will differ greatly from industrial clone detection tools.
8. As mentioned in 4.1 it is possible for all the clone detection tools to miss some clones. Therefore, it is considered that none of the tools will identify all the clones.
9. Total precision would be difficult to achieve because of the subjective nature of cloning. Tools are looking for attributes of two regions of code to match a certain pattern but that does not necessarily mean they are clones.
10. Raising the size threshold of a tool would lower the proportion of cloning identified as the criteria for what is to be considered a clone is narrowed.

A series of experiments are set out in this chapter (see Section 4.7) each aimed at testing one or several of the ten hypotheses described above. The results of these experiments are in Chapter 6. Table 4.1 shows which experiment covers which hypothesis/ hypotheses.

Experiment #	Experiment Title	Hypotheses Tested
1	Comparison of different tools output	1,2
2	Size Breakdown of each tool's results	3,4
3	Unique Clone Classes within results	5
4	Replication Within and Across Programs	6
5	Intersection between each tool's results	7
6	Precision and Recall Analysis	8,9
7	Size Threshold Sensitivity	10

Table 4-1 Experiments and their related hypotheses

4.3. Case studies

Each clone detection tool in the study will be evaluated using three case studies. Three will be chosen to ensure that any observations made about the clone detection tools they are not specific to the case study. Using case studies which are different sizes and which have been developed under differing conditions can provide insight into whether cloning is more prevalent in one type of system than another.

These case studies will consist of academic and non-academic projects. Academic projects may contain a higher proportion of cloning and so are ideal candidates for clone detection. It will also be interesting to compare the results of systems developed with the aid of integrated development environments and those that were developed without their assistance.

4.4. Clone Detection Tools

A sample of five clone detection tools will be used to conduct the case studies. Included in this sample is a novel tool Covet. Covet will be developed as part of this thesis and is an attempt to develop an efficient and effective mechanism for detecting clones. The remaining four tools are CloneDr, CCFinder, Moss and JPlag.

4.5. Manual Verification of clones

In order to reliably establish whether two sections of source code are part of a cloning relation it is necessary to manually check one section against the other. This code reading is a focused example of program comprehension. Maintainers must have a sufficient understanding of each region of code to be able to tell if there is some shared logic that has been copied and altered or if the similarities that caused the clone detection tool to identify that region of code was merely coincidental.

Program comprehension is time consuming. Although the size of each section of code being studied is small there are many of them. For example, if a tool produced 500 identified code sections each with a minimum threshold of 10 lines then the maintainer has a minimum of 500 x 2 x 10 (10,000) lines of code to read. These sections could (and probably will) be spread throughout the source code of an entire system. Hence even more time is taken finding the sections of code. It is for this reason that clone detection tools such as JPlag and CCFinder have developed user

interfaces that allow the maintainer to go directly from the result summaries to the actual source code. This manual verification is particularly important in the detection of plagiarism as it not feasible or indeed wise to accuse someone of plagiarising based solely on automated detection, as false positives are possible.

When carrying out the comparisons of clones from several tools it is not possible to take advantage of the user interfaces provided by any of the tools like JPlag. So other methods of facilitating the retrieval (for inspection) of specific code regions must be used. Initially a database containing the entire source of each system was planned. Maintainers could query the database and display code regions side-by-side using a graphical user interface. However, due to time restraints this was not completed. In its place a substitute was found. A utility program was created that took a set of clones identified by a tool and produced two files. Firstly a summary file was produced which lists each clone pair identified by a tool of a new line. Secondly, a file containing the specific regions of source code involved in that clone summary. Ensuring the order of each file was the same allowed the maintainer to browse the entire source in a single file and use the other summary file to record the results. However, maintainers still have to scroll through the files manually and if a clone is too long the both sections of code do not fit on the screen, which can cause a time delay. Another improvement found was if the second file containing the actual source code was saved as a ".java" and opened in a text editor with syntax highlighting this made verification of similar control flow easier.

Manual verification is obviously still fairly subjective. In order to attempt a more formalised approach a set of criteria were devised that if applied to source code regions would ensure consistent evaluation of clones. Clones were also judged by their significance to maintenance. For example, a series of declarations or imports would not be considered a clone. The criteria applied were as follows:

Similar / Identical control flow and layout: Series of repeated layout blocks could often point to a copy of another piece of code elsewhere in the system. For example, if two functions both contained the same number of if-statements testing similar conditions.

Similar / Identical method names: These usually took the form of a verb-noun

combination with the verb remaining constant and the noun being changed. (For example *saveGraph* and *saveGINGraph*).

Similar / Identical variables: Clusters of identical variables and assignments were often a good indication that the code originated elsewhere.

Similar / Identical comments: Occasionally the same or very similar comment blocks were interspersed in the code. This is quite obviously a legacy of cloning within the source.

During the verification of the potential clones a classification system (see table 4.2) was developed. Five categories were created and each were given a letter and a number which were used a shorthand to record the results in the summary file.

Each clone was considered either a clone (denoted by the letter 'C') or a non-clone (denoted by the letter 'N'). Within these two main categories were sub-categories denoted by numbers. These numbers have no weighting and merely represent the order in which they were created.

Letter	Number	Description
C	1	Same functionality, variable name changes
C	2	Identical
N	3	Non-clone – similar control flow but considered distinct
C	4	Similar functionality but with some changes
N	6	Possible cloned code but not considered significant, either too short or of no interest

Table 4-2 Clone categories used in manual verification

Table 4.2 documents the categories of clone used in the manual verification. These were used to mark a clone as either a non-clone or an actual clone.

Within Chapter 6 in the case studies clones will be viewed as either clones or non-clones. If a clone is not significant to maintenance then in industry they can be considered a false positive as no benefit is gained in removing them.

4.6. Qualitative Evaluation of each tool

Part of this study was devoted to evaluating each clone detection tool from a more qualitative viewpoint. Assessing usability aspects is crucial to support any decision to choose one tool over another.

Attribute	Explanation
Maximum Source Size (SLOC)	An important consideration is how many lines of code a tool can cope with. For example, if a file had 500K lines then a tool who with a maximum capacity of 200K is useless.
Languages Supported	This is a yes/no question the tool can either read the source code or it can't depending on the languages it supports.
Number of preparation steps	Storey [StoreyLecture] highlights the importance of taking into account the amount of extra work or "housekeeping" is needed to achieve a specific task.
Time Overhead	Will the tool become slow and perform badly when handling files close to its limits?
Visualisation Features	Since visualisation is an area of particular interest it will be interesting to look at how each tool represents the clone information / statistics.
Documentation level	How much documentation is provided and how often is it updated?
Support Level	Is there long-term support for the tool or is the user on its own once it has got the tool?
Learning Curve	How easy is the tool to learn and does this come at a price?
User Interface	Does the tool have a graphical front end or a command based interface? Is the GUI customisable is it extensible?

Table 4-3 Evaluation criteria for the clone detection tools

When deciding the evaluation criteria contained in table 4.3 it was important to consider what aspects of usability were most relevant to the very specific field of

clone detection. One definition of evaluation that seems particularly relevant (cited in [IntroEval]) is "*Evaluation is the systematic acquisition and assessment of information to provide useful feedback about some object*". The object or objects referred to are the clone detection tools. Systematic acquisition is the measurement of the attributes listed in table 4.3. Assessment of this information will be carried out using an evaluation table. By asking the same questions of different tools and presenting these answers in tables, like-for-like comparisons can be made and a more informed assessment is possible.

One of the most important attributes is the number of languages supported. If the clone detection tool does not support the language in which a software system was written in then it cannot be used. Time overhead, learning curve and number of preparation steps can all be indicators of how much time will be required to use the tool. If a maintainer is investing a great deal of time and effort to the removal of clones then they will require a tool that is well documented and has sufficient support if they encounter problems when using the tool. Visualisation features could also be required before actual clone removal if the maintainer requires a visual representation of the extent of cloning within a software system.

4.7. Experiments

To test the hypotheses identified in section 4.2 a number of experiments were defined. These are described below.

4.7.1. Experiment 1 Comparison of different tools output

The first and most obvious aspect to measure is the number of clones each tool identifies. Each tool's output is recorded for each case study. Combining these results and presenting them side by side with the number of actual clones (clones which have been manually verified) provides a clear comparison of the accuracy of each tool during the study.

4.7.2. Experiment 2 Size Breakdown of each tool's results

Closer inspection of the size of cloned regions will then take place. This will involve simple statistical analysis of the lengths of code regions identified by a tool. Also an assessment will be made of how much source code could potentially be removed if the clones were removed from the system. Hypotheses three and four are tested in this experiment.

4.7.3. Experiment 3 Unique Clone Classes within results

The study will look at the distribution of clones. How many "original" sections may have been cloned and where they appear in the system. This will be of particular use to software maintainers as if an error is found in one clone it is likely that the error has been copied along with the logic of the original code. It is also relevant identifying potential reuse candidates.

4.7.4. Experiment 4 Replication Within and Across Programs

Another interesting aspect of the clone detection is to look at replication across and replication within programs as described by Burd [Bur97]. In other words comparing the proportion of the clones that are found within the same files and what proportion are found in separate files. Results from this analysis will not be relevant for JPlag and MOSS as they are attempting to detect plagiarism and so do not examine replication within the same file. This is because plagiarism is the copying of someone else's work.

4.7.5. Experiment 5 Intersection between each tool's results

Further analysis of the clone results will examine the intersections and differences between the result sets. This will attempt to establish the extent to which each tools' results are similar.

Clone by clone visualisation

As part of the analysis of where the tools agreed and disagreed, a simple visualisation technique has been applied. Each set of clones outputted by a clone detection tool is presented as a table; each row within the table represents a clone detected by that tool. Within that table there will be four columns, one for each of the other tools. If the clone in that row was identified by another tool then the cell in that other tool's column is coloured in black. Whereas if the clone was not identified the cell is coloured in yellow (see table 4.4 for an example).

Tool A Clones	ToolB	ToolC	ToolD
Clone1			
Clone2			
Clone3			

Table 4-4 Example of clone by clone visualisation

This visualisation technique is useful to highlight which clones are being detected by some tools and not by others. If a row is entirely black it means that that particular clone was found by all the clone detection tools. Whereas a predominantly yellow row shows that the clone was only identified by one or two tools. If a column is mainly/entirely black then this shows that the tool in that column picked up most/all of the clones. For example tool B in table 4.4.

4.7.6. Experiment 6 Precision and Recall Analysis

Precision and recall values will be established for each of the case studies. In order to calculate these statistics a base clone set will be created. This base clone set will contain all the actual (manually verified) clones identified by each of the clone detection tools. Precision and recall will be calculated using the following formulae.

Precision = number of actual clones identified by tool / number of potential clones identified by tool

Recall = number of actual clones identified by tool / number of actual clones identified by all tools

4.7.7. Experiment 7 Size Threshold Sensitivity

During the experiments a minimum size threshold of 20 was used to filter out smaller spurious clones. However, it is interesting to investigate how altering this threshold affects the number of clones outputted by each clone detection tool. Clone detection tools were executed using thresholds ranging from 10 to 30. Focusing on the differences between the number of clones identified using different minimum size thresholds may give a better indication of a standard size threshold that should be used. This experiment is focused solely on the clones identified by each tool. No precision measurements will be taken as any extra clones identified for the lower thresholds are not treated as actual clones as they are considered too small to warrant maintenance effort. Experimenting with the threshold will test hypothesis eleven. Evaluation will consist of looking at the actual number of clones filtered out or lost with the increases in the minimum size threshold. Further to this the increases will be grouped into quartiles 10 – 15, 15 – 20, 20 – 25 and 25 – 30. By monitoring the percentage filtered out at each quartile it is possible to assess at which stage the most clones are being filtered out.

This chapter presented the hypotheses to be tested and the experiments that will be used to test them. It also gave criteria for choosing a set of case studies and the set clone detection tools to be used. The set of clone detection tools will include plagiarism detection tools. Case studies of different sizes and development environments will be used. There will also be a small qualitative evaluation of each of the clone detection tools focusing on attribute. Such as the number of programming languages supported and preparation time required. Details of the manual verification process were also demonstrated. This showed how potential clones will be verified as either actual clones or non-clones.

5. Implementation

As discussed in Chapter 4 a new tool called Covet was used in each of the case studies. This chapter will introduce the data structures used both in Covet and for each of the other tools. This chapter then gives a description of how the intersection between the clones identified by different clone detection tools was calculated. Following this is a description of how Covet was developed. In addition, details of modifications made to the other clone detection tools to facilitate the comparison of their outputted clones are provided.

5.1. Data structures used

When developing Covet it was necessary to represent the clones outputted. As well as this it was also necessary to create a tool-independent data structure that would allow the comparison of results.

5.1.1. Representing a clone: CodeRegion and CodeRegionPair data structures

Representing cloned code is quite simple. The information required is the name of the files where the clone was identified. In addition, the start and end indices (line numbers) of the regions of code identified as being clones.

The data structure used to model this was named a CodeRegionPair (figure 5.1). There are only three main components which make up a CodeRegionPair object, two CodeRegion objects (which as the name infers represent a region of code) and finally the name of the tool that identified the clone. Each CodeRegion stores the name of the file and the start and end indices of the region in question. Within a CodeRegionPair the two CodeRegions are ordered alphabetically by the name of the file and then by the start index.

Some clone detection tools provide additional information about a code clone such as the number of tokens matched in each region of code. However, as not all of the tools provide this feature this information was not taken into consideration.

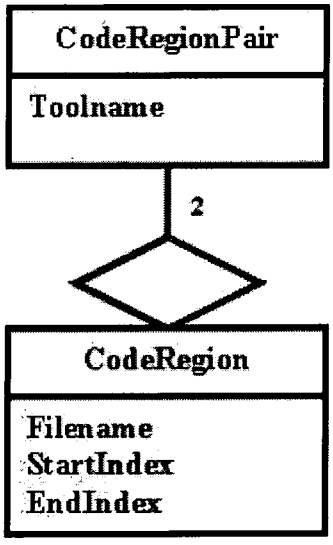


Figure 5-1 UML class diagram of CodeRegionPair

Variable	Value	Object
Toolname	JPLAG	CodeRegionPair
Filename	ContextPreferencesPanel.java	1st CodeRegion
Start Index	10	
End Index	32	
Filename	InterfacePreferencesPanel.java	2nd CodeRegion
Start Index	12	
End Index	34	

Table 5-1 An example CodeRegionPair

Table 5.1 gives an example of an instance of a CodeRegionPair. In this example JPlag has identified a match between the files ContextPreferencesPanel (1st CodeRegion) and InterfacePreferencesPanel (2nd CodeRegion) the 1st region of code starts at line 10 and ends at 32 while the 2nd starts at line 12 and ends at 34. By storing each match from all the tools as CodeRegionPairs it was possible then to perform comparisons between each tool's results set without any further conversion.

5.2. Determining the intersection between the clones identified by different tools

A comparison of the clones identified by each of the clone detection tools is required to establish whether the clone detection tools identify similar clones. To compare the clones identified by two tools each clone (CodeRegionPair) identified by one tool is compared with every clone identified by the other tools.

This comparison could be a strict equality function. For instance, equality requires two CodeRegionPairs contain exactly the same CodeRegion objects (i.e. the same files with the same start and end indices). However, this approach is not flexible enough and is not really useful to a maintainer. A more flexible approach would be to compare the amount of code that is shared between two clones. By defining a minimum percentage of code that has to be shared by both CodeRegionPairs for them to be considered a match, a more useful comparison is achieved. The minimum percentage used throughout the experiments was a constant 60%. 60% was chosen to ensure that the majority of the code regions overlapped whilst still allowing for the fact that different tools will probably not find the exact same regions of code.

The following examples show how the CodeRegions and CodeRegionPairs were stored in files and then how an overlap was calculated.

For example
FileOne.java (10 – 50) & FileTwo.java (11 – 62) (first CodeRegionPair)
FileOne.java (5 – 45) & FileTwo.java (16 – 56) (second CodeRegionPair)

The first CodeRegionPair in the example shows that lines 10 to 50 in FileOne.java and lines 11 to 62 in FileTwo.java are clones. Below is a CodeRegionPair that describes a clone involving similar but not identical CodeRegions.

To work out the percentage of code shared between these two CodeRegionPairs we first check that both the files involved in the CodeRegions are the same. In this case both CodeRegionPairs consist of CodeRegions from FileOne.java and FileTwo.java. Following this, the start and end indices of each corresponding CodeRegion are checked.

Table 5.2 gives details of how the overlap percentage was established. Clones A and B correspond to the first and second CodeRegionPairs given in the above example.

CodeRegionPair	1st CodeRegion	2nd CodeRegion
Clone_A Start – End (*length*)	10 – 50 (*41*)	11 – 62 (*52*)
Clone_B Start – End (*length*)	5 – 45 (*41*)	16 – 56 (*41*)
# lines shared	(45 – 10) = *36*	(56 – 16) = *41*
% of Clone_A	88%	79%
% of Clone_B	88%	100%

Table 5-2 Calculations required to work out the overlap percentage of two clones

Table 5.2 shows that the two CodeRegionPairs would be considered a match as there is an overlap of over 60% for each CodeRegions in both CodeRegionPairs. However if the 2nd CodeRegion in Clone_A had been significantly longer, and Clone_B's 2nd CodeRegion remained the same, the overlap percentage would be reduced. Table 5.3 shows how using this method of overlap percentage one clone might be considered a match for another but be the reverse might not hold true.

CodeRegionPair	1st CodeRegion	2nd CodeRegion
Clone_A Start – End (*length*)	10 – 50 (*41*)	11 – 162 (*152*)
Clone_B Start – End (*length*)	5 – 45 (*41*)	16 – 56 (*41*)
# lines shared	(45 – 10) = *36*	(56 – 16) = *41*
% of Clone_A	88%	30%
% of Clone_B	88%	100%

Table 5-3 A further example of the overlap percentage of two clones

Table 5.3 shows how although Clone_B is considered as a match for Clone_A using the minimum percentage overlap to determine a match Clone_A is not considered a match. The next example shows an example of when two clones are not considered a match.

FileOne.java (10 – 50) & FileTwo.java (11 – 62) (first CodeRegionPair)
FileOne.java (67 – 130) & FileTwo.java (16 – 56) (second CodeRegionPair)

These two CodeRegionPairs are not considered as a match because the CodeRegions in FileOne.java do not overlap.

5.2.1. Data structures used specifically for Covet: RoutineInfo, Metric and DatrixFileParser

CodeRegions only store the start and end of a region of code. Extending this to store only whole routines was required to create Covet. RoutineInfo objects also stored the start and end indices of a region of code, the main difference being this region encapsulated exactly one routine. Additional information was also required such as the routine's metrics (generated from Datrix). Each RoutineInfo object stored the name of the routine, the class to which the method belonged, filename, pathname, its list of parameters and a hash map giving a mapping from each metric name to a specific value.

Each metric was represented as a Metric object containing the name of the metric, a short description and a specific Delta (threshold).

Data stored in a RoutineInfo was generated mainly by the Datrix metric tool. It takes as input a source code file and produces a set of metrics for each routine within that file. These metrics along with other data are then stored in a RoutineInfo object. These objects are in turn held within a DatrixFileParser object. Figure 5.2 is a UML class diagram showing the relationship between these three classes.

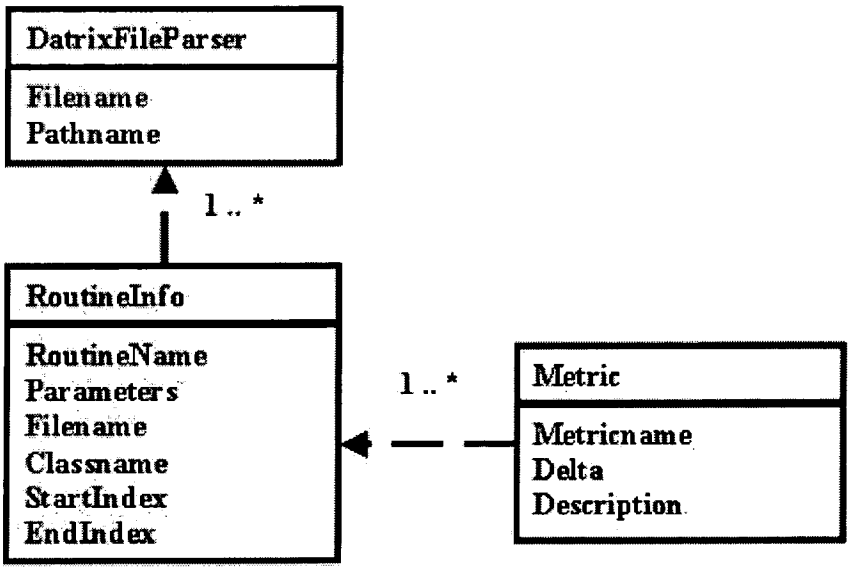


Figure 5-2 UML class diagram of the data structure used for Covet

Figure 5.2 shows that both DatrixFileParser and RoutineInfo store the name of the source file. The filename variable is used when all the RoutineInfo objects are stored in a larger set. This avoids naming clashes when comparing routines from other files.

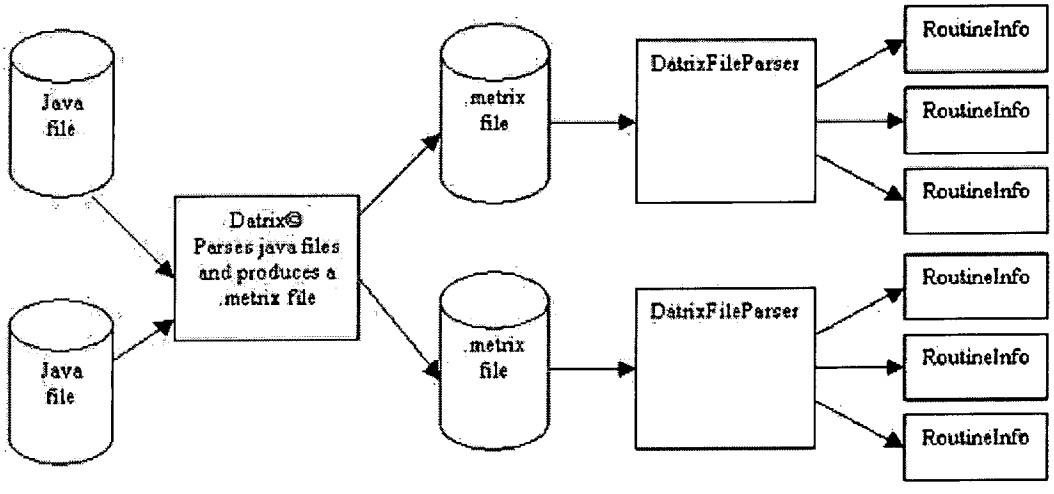


Figure 5-3 UML data flow diagram for extracting metrics from java files

Figure 5.3 shows how Datrix processes the java files and outputs files containing the metric information about each routine (called *.metrix* files). These are then read by a DatrixFileParser object which then creates and populates RoutineInfo objects.

After all the routines within a system have been read-in, the next stage is to compare them and to examine their metrics. This was achieved with the

MultipleFilesComparer program. Users input a list of files (.metrix), a minimum threshold (used to filter out routines which are too short) and a file containing a list of metrics (and their associated deltas) that are to be used in the comparison. This allows a pre-selected set of metrics to be used in any comparison. At the end a list of routine pairs is outputted which are considered potential clones.

5.3. Development of Covet

Some aspects of Covet have already been mentioned in this chapter. This section gives an overview and explains some of the background of Covet. After this a description of how the tool was "tuned" to improve its clone detection. It describes how metrics were chosen both manually, and later by trying to spot patterns in the metrics of the clones outputted by the other tools.

5.3.1. Overview and Background of Covet

Covet was inspired by the research carried out by Mayrand [May96b]. Mayrand attempted to identify routines that had been cloned. Four "*points of comparison*" [May96b] were examined in order to determine if two distinct routines belonged to a cloning relationship. These were name, layout, expressions and the control flow of each function. Within these points of comparison a set of metrics were used to determine whether two points of comparison were equal, similar or distinct.

Mayrand [May96b] used student source code (widely known to potentially contain a reasonable percentage of plagiarism) to select the metrics that were chosen. Thresholds were set in accordance from previous experience of large-scale systems. Mayrand [May96b] uses three levels of similarity for each "point of comparison". Routines are considered to have "*equal*" control flows if every metric within the control flow set is equal. If the absolute difference between each of the metrics in the set is less than or equal to that metric's threshold then they are considered "*similar*" from that point of comparison. Otherwise two routines are considered "*distinct*".

Table 5.4 shows RoutineA and RoutineB are considered here to have "equal" Control

Flows because the metrics defined in the Control Flow set are all equal.

Control Flow	Routine A	Routine B	Threshold
Metric1	3	3	5
Metric2	3	3	2
Metric3	2	2	2
Metric4	1	1	10

Table 5-4 Equal control flow metrics comparison

Table 5.5 shows that RoutineA and Routine B are considered to have "similar" Control Flows because the absolute difference between the corresponding metrics defined in the Control Flow are less than or equal to the threshold.

Control Flow	Routine A	Routine B	Threshold
Metric1	3	4	5
Metric2	3	5	2
Metric3	2	2	2
Metric4	1	7	10

Table 5-5 Similar control flow metrics comparison

Mayrand has attempted to choose a set of metrics that can be used to diagnose the cloning relationship between two routines and this is what Covet attempts to emulate. Covet uses a more recent version of Datrix© which offers additional metrics for example, as well as deriving metrics at a routine level there are Halstead metrics provided. Datrix can also now produce metrics for a whole class or file. However, these new metrics were ignored because the aim was to find cloning within routines. By only looking at class or file metrics cloning occurring within classes and files would be missed and also the chances of whole files and classes being cloned is much less than just single routines.

5.3.2. Extracting Metrics for Covet

Covet recieves all its metrics data from Datrix. Datrix is a command line driven program and uses an interactive, rather than batch mode, for producing metrics data for each routine within a program. As Datrix can only be run in interactive mode, a

wrapper program was required to run the tool in batch mode. This was a fairly simple task involving a perl script to create a batch file / c-shell script. The resulting batch file then made each individual call to Datrix on the correct source file and piping that output to the correct destination file. A naming convention was introduced to the Datrix produced files, each file had the suffix *.metrix*. Discussion of the format of these files is continued in Further Implementation Issues later in this chapter.

Covet Methodology

Rather than attempting to reuse the metrics chosen by Mayrand, experiments were carried out in an attempt to find a set of metrics that best suited clone detection. Five experiments were devised each using either differing metrics or thresholds. Initially the first two sets of metrics were based on general observations and ideas of cloning. Metrics describing both control flow and the volume of statements were chosen (table 5.7). The second set (table 5.8) has different threshold values. Following this metrics for the third set (table 5.9) used a similar technique to Mayrand's sample study. The only difference was, in this instance, instead of using student created programs the programs were artificially created.

Four basic programs were devised, a HelloWorld application, a BubbleSort algorithm, a currency conversion application and a Binary Search algorithm. These were altered in similar fashion to produce a series of clones. Metrics were taken from these programs and were inputted into a spreadsheet and plotted onto a graph. From these results a top ten metrics were produced.

Table 5.6 provides a description of each metric used in the experiments described previously.

Metric	Description
RtnStmCtlNbr	number of control flow statements
RtnStmDecNbr	number of declarative statements
RtnStmNbr	total number of all statements
RtnStmExeNbr	total number of executable statements
RtnCalXplNbr	number of explicit function/methods calls made within the routine.
RtnCplExeAvg	mean complexity of executable statements in the function

RtnCplExeMax	maximal complexity of executable statements in the function
RtnCplCtlAvg	mean complexity of predicate statements
RtnCplCycNbr	cyclomatic number or v(G) of the routine defined by McCabe [McC76]
RtnCplCtlMax	maximal control predicate complexity
RtnScpNstLvlAvg	mean nesting level of scopes in the function
RtnScpNstLvlMax	maximal nesting level of scopes in the function.
RtnStmNstLvlAvg	mean nesting level of statements in the function.

Table 5-6 Metrics used within the Covet tuning experiments.

Metric	Threshold
RtnCplCtlAvg	7
RtnCplExeAvg	3
RtnCplExeMax	3
RtnScpNstLvlAvg	3
RtnStmCtlNbr	3
RtnStmDecNbr	3
RtnStmNbr	3
RtnStmNstLvlAvg	3

Table 5-7 Intial set of metrics used in Covet tuning

Metric	Threshold
RtnCplCtlAvg	2
RtnCplExeAvg	2
RtnCplExeMax	2
RtnScpNstLvlAvg	2
RtnStmCtlNbr	2
RtnStmDecNbr	2
RtnStmNbr	2
RtnStmNstLvlAvg	2

Table 5-8 Second set of metrics used in Covet tuning

Tables 5.8 and 5.9 list sets of metrics and their associated thresholds used in the second and third metrics experiment respectively. As can be seen from a comparison of tables 5.8 and 5.9 the metrics used are identical the only change made is in the thresholds set.

Metric	Threshold
RtnScpNstLvlMax	2
RtnCplCycNbr	2
RtnCplCtlAvg	3
RtnCplExeAvg	3
RtnStmCtlNbr	2
RtnStmExeNbr	2
RtnStmNstLvlAvg	2
RtnCplCtlMax	2
RtnCplExeMax	5
RtnScpNstLvlAvg	2
RtnCalXplNbr	2

Table 5-9 Top ten metrics taken from pilot study

5.3.3. Automatically generated Thresholds

During the development of Covet attempts were made to automatically generate thresholds from the clones produced by the other tools. The formula used was as follows:

1. Retrieve all the manually verified CodeRegionPairs from the other clone detection tools (CloneDr, JPlag, Moss and CCFinder);
2. Attempt to map each CodeRegionPair to two distinct functions (one for each CodeRegion);
3. Record the metrics of function pair;
4. Work out the differences between the metrics within each function pair;
5. Use simple statistical analysis to estimate a threshold.

An example

CloneDr identifies a match between file_one.java from lines 10 – 100 and file_two.java lines 110 – 200. Within file_one.java there is a routine FunctionA which starts at line 5 and ends at line 90 and file_two a routine called FunctionB which starts at line 105 and ends at line 190. These two routines will then be taken as the function pair and the absolute difference between each metric will be recorded. The process is repeated for every other match identified and the differences are recorded as a mapping function from a metric name to a set of differences. (See table 5.10).

Metric Name	Differences	Mean	Median
Metric1	[1,2,3,3,2,1,2,2]	2	2
Metric2	[3,2,4,2,1,3,4,3]	2.75	3

Table 5-10 Example of automated thresholds method

Table 5.10 shows an example of how the automated thresholds were calculated using the differences in the metrics of routines that were identified by the other tools.

From these differences, simple statistics can be taken such as mean, maximum and median values. These values can then be used as thresholds. All the available metrics were chosen rather, than a subset, as this meant the results were not affected by any human decision.

When matching a region of code to a whole routine it was important to ensure that erroneous and misrepresentative matches were not made. It was decided that only CodeRegions that covered a minimum of 60% of the lines in a routine would be considered a match.

5.3.4. Preliminary Experiments: Results from Covet Metric trials

These are the results from the experiments involving Covet using differing sets of metrics and threshold levels. The selection process for the metrics used in the trials is described in section 5.3.2. These sets of metrics were then used to configure Covet and then run on an application called GraphTool. Appendix B contains the differences between the artificially created clones and the original programs.

Experiment	Metrics	#Potential clones	#Actual clones	Precision	Recall
1	Initial set derived from general observation. (Table 5.7).	248	41	16%	18%
2	Further refinement taken from the original set with thresholds all set to two. (Table 5.8).	58	32	55%	15%
3	Top Ten metrics derived from sample study (Table 5.9).	51	46	83%	20%
4	Metrics derived from the results of other tools (with verification).	121	41	34%	18%
5	Metrics derived from the results of other tools (without verification and using median differences).	276	65	23%	29%

Table 5-11 Results achieved by running Covet using the various sets of metrics.

The results from running Covet using these metrics were promising. However, a high proportion of false positives were found (see table. 5.11). After examining the clones identified by the prototype set, it was obvious that further refinement was needed. This was achieved by not altering the metrics but by lowering their thresholds to a universal level of 2. As table 5.11 shows this restriction produced a significant drop of 77% in the number of potential clones identified. This is not reflected in the recall which shows a small drop of just 3%.

Clones identified using the metrics taken from the sample study produced the best precision. Although the smallest set of potential clones was identified, it identified the second highest number of actual clones (46). Only experiment 5 achieved a

higher recall. However, experiment 5 also identified the highest number of potential clones and had a much lower precision.

5.3.5. Comparison of Results from Covet Tuning

An interesting comparison is to look at the similarity of the results produced by each experiment. Table 5.12 shows each of the four metric experiments, the number of clones identified in each (in brackets) and in the third column the number of clones that were found in both experiments. Column four lists the number actual clones manually verified from this intersection.

Results A (#)	Results B (#)	Intersection	Actual Clones
Experiment 1 (248)	Experiment 2 (58)	58	32
"	Experiment 3 (51)	35	32
"	Experiment 4 (121)	17	13
"	Experiment 5 (276)	60	33
Experiment 2 (58)	Experiment 3 (51)	30	29
"	Experiment 4 (121)	16	13
"	Experiment 5 (276)	38	29
Experiment 3 (51)	Experiment 4 (121)	14	13
"	Experiment 5 (276)	34	34
Experiment 4 (121)	Experiment 5 (276)	50	25

Table 5-12 Intersection results from the 5 Covet metric experiments

Table 5.12 shows the intersection analysis between the results of the Covet experiments. An interesting observation is that the results from Experiment 2 are a subset of the results from Experiment 1. However this is to be expected as the same metrics were used and the only difference was a reduction in the threshold settings not the metrics themselves. Experiment 3 has the highest proportion of matches. It retrieved 51 potential matches, 35 five of which appeared in experiment one's results, 30 in Experiment 2, and 34 in Experiment 5. Only 14 matches were found in the results from Experiment 4.

Experiment 4 appears to have produced the most dissimilar results. Sharing 17 with Experiment 1, 16 with Experiment 2, 14 with Experiment 3 and 50 with Experiment 5. This last result is much higher than the others but this is probably due to Experiment 5's large result set.

5.3.6. Further Implementation Issues

Producing Datrix *.metrix* files

Metrics produced by Datrix were stored in text files. The method for using these is explained in section 3.10.2 (see figure 5.5). This section provides more detail as to the issues faced by parsing the files and coping with incorrect line numbering.

```
BEGIN_RTN
BEGIN_GEN_INFO
  NAME "ColoredSquare"
  SCOPED_NAME "unnamed::ColoredSquare::ColoredSquare"
  MANGLED_NAME "__Q27unnamed13ColoredSquare5Color"
  FILE_NAME "ColoredSquare.java"
  PATH_NAME "c:\datrix\clones\GraphTool"
  LINE_START 7
  LINE_END 10
END_GEN_INFO
BEGIN_METRIC
  RtnArgXplSum 0.0000
  RtnCalXplNbr 0.0000
  RtnCastXplNbr 0.0000
  ...
  RtnStmNstLvlSum 1.0000
  RtnStmXpdNbr 2.0000
  RtnStxErrNbr 0.0000
END_METRIC
END_RTN
```

Figure 5-4 Example output from Datrix

Figure 5.4 shows an abridged example of the output produced by Datrix. The data is marked up with a simple tagging system. BEGIN_X and END_X tags encapsulate

groups of fields. As Covet was examining cloning on a routine level the only metrics it was interested in were those encapsulated within BEGIN_RTN and END_RTN tags. There are two types of information stored about each routine. Inside the BEGIN and END GEN_INFO were fields containing information about each routine including the method name, its scoped name, mangled name (which contained details of any parameters) and the start and end line of the routine. Specific metrics were presented inside the BEGIN_METRIC and END_METRIC tags. They are presented as the metric name followed by a space and then the value for that particular routine.

5.3.7. Parsing *.metrix* files

Data taken from the *.metrix* files was parsed using a Java file created called DatrixFileParser. Parsing the files was quite straightforward. The only slightly complicated matter was deciphering the mangled name to extract the parameters passed to a routine. There is little to no documentation available on how to use or interpret Datrix's results so this took a little time to work out.

Classname	Mangled Name	Parameters
FilePreferences	"save__Q27unnamed15FilePreferences11PrintWriteri"	PrintWriter Object and Int
Coord3D	"__Q27unnamed7Coord3Dfff"	Float, Float and Float
CLLOptionsDialog	"__Q27unnamed16CLLOptionsDialog6JFrame"	JFrame

Table 5-13 Examples of mangled names within Datrix files

Table 5.13 shows how the mangled name field in Datrix translates into parameters. The basic method for extracting parameters is as follows.

i. Find the name of the class in the mangled name so for the first example the classname was FilePreferences.

```
save__Q27unnamed15 FilePreferences 11PrintWriteri
```

remove upto and including the classname to leave the parameters.

ii. **11**PrintWriteri

iia. if the string begins with a number (n) then the next parameter is a class rather than a primitive whose name is n characters long. Next up to the end of the class name and store this as a parameter. Loop from ii. In the example given above the number 11 tells Datrix that the next parameter is a class whose name is 11 characters long. (PrintWriter)

iib. Else if the string doesn't begin with a number then it must be a primitive and will be exactly one character i = int, b = boolean, f = float etc... Read this character in and store it as a parameter. Loop from ii.

After initial testing of Datrix and its results, a problem / error was discovered. Datrix for an unknown reason produced incorrect line numbering for some routines. There appeared to be no pattern to the mis-numbering (it wasn't simply a case of Datrix not including commented out lines). One of the main experiments was to compare each tool's results to see how many find the same clones. As Covet identifies whole routines rather than just regions of code it was necessary to convert each routine into a region of code (using the start and end line numbers as the bounds). If these indices were inaccurate this would make any comparison involving Covet's CodeRegions invalid. The solution was to create a simple parser that extracted each method name, its class name, its parameters and most importantly accurate start and end indices. Results from this parsing were stored in a text mark-up file similar to the .metrix files. Each java file had its own method summary file and this was used within the DatrixFileParser to allocate the correct start and end indices to each routine.

5.4. Modifications to existing tools' results

In addition to the modifications made to Covet had to be made to the results produced by the other clone detection tools. Every clone detected by a tool had to be translated into a CodeRegionPair.

Each of the clone detection tools presents its clones in a distinct format. Table 5.14 gives a summary of each tools output format.

Tool	Format
CCFinder	Text file containing pairs of matched code regions
CloneDr	Text file containing series of matched code regions
JPlag	Multiple HTML files containing pairs of matched code regions
MOSS	Text file containing pairs of matched code regions
Covet	Internal set of code regions

Table 5-14 Clone detection tools with their output format.

Reading in each tool's results required a specific class to parse the data and convert them into CodeRegionPairs. There will now follow a brief description of each tool's results format.

5.4.1. CCFinder

FileL	FromL	ToL	FileR	FromR,	ToR
0.0	44,13,38	72,13,76	0.0	64,13,64	97,12,102
0.0	161,17,233	172,13,265	0.0	162,17,237	173,17,269

Table 5-15 Output snippet from CCFinder.

Table 5.15 shows a section of the CCFinder output that describes each matched code region. There are six main fields. FileL is the ID of the first file in the match, FromL of FileL and is composed of three sub fields separated by commas; Line number, column and token index which represent the start position of FileL. ToL is similar to FromL but each sub field represents the end index of FileL.

FileR contains the ID of the second file in the match and FromR and ToR hold the same information for FileR as FromL and ToL did for FileL. Each file's mapping from filename to file ID is recorded at the start of the CCFinder report.

5.4.2. CloneDr

```
=== Tree Clone Tuple ====
Tuple with 3 clones, 2 parameters; similarity = 0.9333333333333333
#4    #5b7f450    #4c9a7c0
```

```
Clone 1: 3 lines from Line 113 to 115 File:
F:/Customers/UDurham/GraphTool/Node.java
// Access function for groupParent
public Node getParentNode() {
  return groupParent;
}

-------------------------------------------------------------------
---
#4    #5b7f450    #84db380

Clone 2: 3 lines from Line 503 to 505 File:
F:/Customers/UDurham/GraphTool/Edge.java
public Node getRealLinkTo() {
  return linkTo;
}

-------------------------------------------------------------------
---
#4    #5b7f450    #84dbdc0

Clone 3: 3 lines from Line 508 to 510 File:
F:/Customers/UDurham/GraphTool/Edge.java
public Node getRealLinkFrom() {
  return linkFrom;
}
```

Figure 5-5 Sample output from CloneDr

As can be seen from Figure 5.5 CloneDr provides the actual code of each match in its report. This is probably to compensate for the lack of graphical visualisation available. The specific parts of this report required for converting the clone series into pairs of CodeRegions are the number of clones in the series (in this case 3), each region of code's filename (Node.java and 2 x Edge.java) and finally the start and end lines. From the example in figure 5.5, three pairs of CodeRegions would be generated.

5.4.3. JPlag

JPlag returns results in a series of HTML files. It was necessary to parse each file using a perl script and then store all results in a single text file. This text file uses a very simple format of two CodeRegions' toString() form separated by a hash character.

i.e.

```
Coord3D.java(1-72)#IntCoord3D.java(1-72)
SelectMenu.java(8-40)#GraphMenu.java(14-45)
SelectMenu.java(42-66)#GraphMenu.java(60-85)
```

5.4.4. MOSS

MOSS presents each match grouped into the two files involved.

```
Typical5.java + Typical4.java: tokens 964   lines 164
 total tokens 2090 + 2763, total lines 469 + 507, percentage matched
46% + 34%
 67-76, 67-76: 23
 85-165, 83-163: 380
 253-267, 164-178: 109
 273-282, 179-188: 67
```

Figure 5-6 Output from MOSS

As figure 5.6 shows MOSS provides two files and then a series of matches found within and between the two files. It also provides details of the number of tokens involved in each match. However, this information is not required to generate CodeRegions and is ignored. Figure 5.6 would produce 4 CodeRegions all involving the files Typical5.java and Typical4.java.

5.5. Chapter Summary

This chapter gave the details of the implementation phase of the thesis. This included the creation of Covet by making adaptations to Datrix such as the fix added to the line numbering system. It also included details of how the different tools clones were translated into a single data structure. The creation of such a data structure was important to allow the comparison of each tool's clones. An explanation of how the metrics used in Covet were chosen.

6. Case Studies

6.1. Case Studies Overview

This chapter will present the three case studies used in this thesis. Following this, the results from each of the experiments are presented (described in chapter 4).

6.2. Detection Tools and Target Systems

Each clone detection tool in the study was evaluated using 3 software systems (case studies) written entirely in Java. All of the systems were written at the University of Durham by either undergraduates or postgraduates from within the university. This offered several advantages, firstly if necessary, the original programmer could be contacted to verify whether or not a piece of code was the result of copy and pasting. It also made independent verification easier because of the uniformity of layout style. The three systems were as follows:

System name	Purpose	Development	Size
GraphTool	Graph plotting tool	Developed for the Computer Science department by a postgraduate student (1999)	16,335
Club Tropicana	Cocktail mixing and optimising system	Developed as part of a group project consisting of 6 people (2002)	23,967
Durham Barcrawl Planner	Route Planner for a "bar crawl"	Developed as part of a group project consisting of 6 people (2001)	8,741

Table 6-1 Systems used in the clone detection experiments

In the method it was stated that variation in size was an important factor to see how scale affects the clones detected. Table 6.1 shows that the tools range from just under 9K to nearly 24K. There are also differences in the development of each case study as Club Tropicana and the Durham Barcrawl Planner are both academic projects whereas GraphTool, although developed for the university, was in effect a

commercial project. GraphTool is also distinguished because it was developed by a single individual not a group. Club Tropicana is the largest of the three case studies in size but it includes an open source project called JavaLayer [JavaLayer], which is used to decode and play MP3 audio files. Both Club Tropicana and The Durham Barcrawl Planner were created with the aid of JBuilder which is an integrated development environment. JBuilder also provides automatically generated code for GUI's. Automatically generated code should provide a substantial amount of cloned code.

Four established detection tools[1] were used in the case study; JPlag, MOSS, CCFinder and CloneDr. JPlag and MOSS are web-based academic tools aimed at detecting plagiarism in student's source code. CloneDr and CCFinder are stand-alone tools looking at code duplication in general. A fifth tool Covet is also included in the results from the study.

6.3. Qualitative Evaluation of each tool

This section contains the results of the qualitative evaluation described in section 4.6. Each tool is evaluated on the same metric.

MOSS	
Supported Platform	Any platform, processing is carried out on a server at Berkeley University. Submission requires a perl script.
Languages	8 (C, C++, Java, Pascal, Ada, ML, Lisp, and Scheme)
Maximum Source Size (SLOC)	No maximum limit was stated. But it is designed for academic submissions.
Number of preparation steps	0. Submission to the server is carried out automatically by a perl script.
Time Overhead	Moss relies on submission of the source code over the internet. Turnover time can also be affected by network traffic and the speed of connection being used by the user.
Visualisation Features	HTML summary of matches found. Selecting a match takes user to the two sections of code presented side-by-

1 Originally DUPLOC was planned to be included but problems with the batch mode operation made this unfeasible.

	side and colour coordinated.
Documentation level	Minimal but sufficient explanation of how the tool works and how to interpret the results. Comments within the submission script explain its usage.
Support Level	MOSS is a non-commercial tool and so there is no formal support.
Learning Curve	MOSS provides a simple service and is very easy to operate minimal learning is required.
User Interface	Command line submission and web based presentation of results.

Table 6-2 Evaluation results for MOSS

Table 6.2 describes the various features MOSS provides. MOSS accepts submissions solely for academic purposes. This and the lack of security and support means that it is not suitable for commercial application. However, MOSS is compatible with a wide range of programming languages and is widely used in academia for plagiarism detection.

JPlag	
Supported Platform	Any platform, processing is carried out on a remote server. Submission uses either a Java application or applet.
Languages	4 (C, C++, Java and Scheme)
Maximum Source Size (SLOC)	No maximum limit was stated. But it is designed for academic submissions.
Number of preparation steps	0. Submission is carried out automatically by the Java applet/application.
Time Overhead	JPlag faces similar delays to MOSS. JPlag also requires the user to wait while it returns the results to the user as opposed to MOSS which simple emails the user the URL from where they can browse the results online.
Visualisation Features	HTML summary of matches found. Selecting a match takes user to the two sections of code presented side-by-side and colour coordinated. JPlag and MOSS use identical interfaces JPlag was the original.

Documentation level	More extensive than MOSS there is a paper explaining how JPlag works and detailed descriptions of how to interpret the results.
Support Level	Support is provided by Guido Malpohl via email responses were usually returned within 24 hours.
Learning Curve	JPlag provides simple functionality and is very easy to use.
User Interface	Command line submission and web-based presentation of results.

Table 6-3 Evaluation results for JPlag

As with MOSS the academic nature of JPlag (described in table 6.3) means that it is not viable for commercial application. It supports fewer languages than MOSS but still is used widely in academia.

An evaluation of CloneDr (table 6.4) is a difficult task as a Java version of the system was not ready for release. Semantic systems [SemSys] very kindly allowed the submission of source code and then returned the results via email. However, it is possible to describe several features of CloneDr that can be used to compare it with the other tools.

CloneDr	
Supported Platform	Windows NT / 2000
Languages	COBOL, C, C++ and Java but can be extended by providing the relevant parsing information.
Maximum Source Size (SLOC)	1 million source lines – which again according to Semantic Designs can be upgraded for larger systems.
Number of preparation steps	If the target system contains only source code written in the languages mentioned previously then none. Otherwise a "DMS domain language definition" must be provided.
Time Overhead	Unknown.
Visualisation Features	None.
Documentation level	Documentation consists of two text files Readme.txt (2 pages) and Userguide.txt (12) pages. They provide adequate detail on how to use the tool.
Support Level	Very Good. Semantic Designs appear very responsive to

	user enquiry.
Learning Curve	Unknown.
User Interface	CloneDr provides a graphical and command line interface.

Table 6-4 Evaluation results for CloneDr

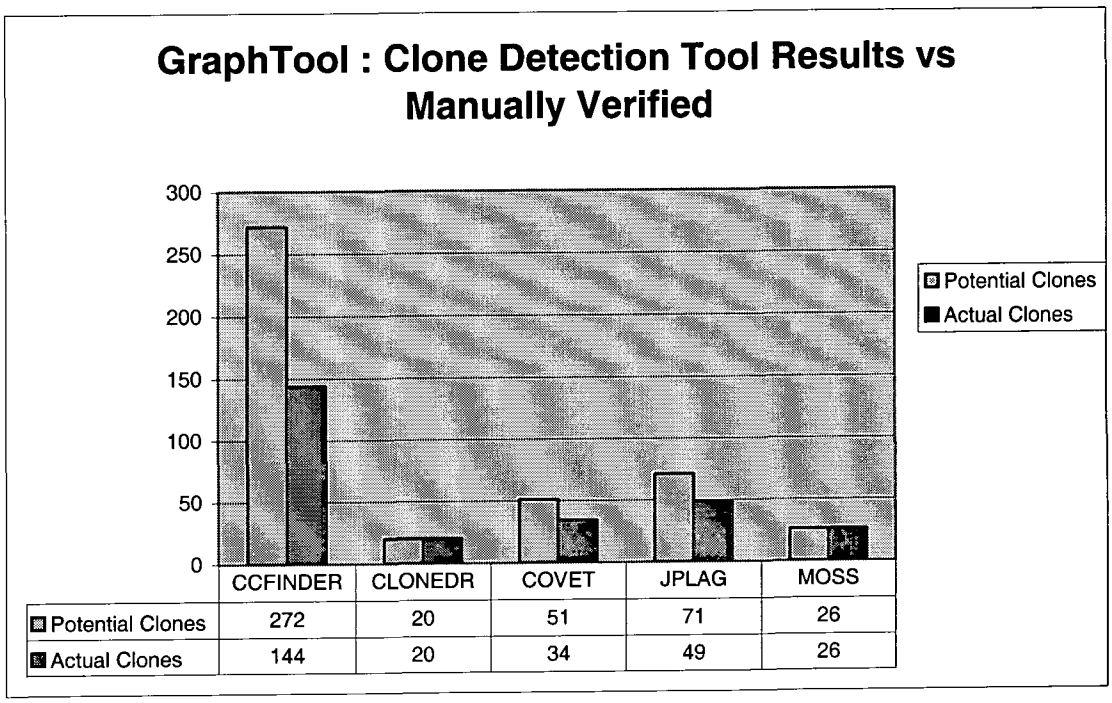
C(ode)C(lone)Finder	
Supported Platform	Windows NT / 2000.
Languages	C, C++, Java and Cobol but can apparently be extended.
Maximum Source Size (SLOC)	None. CCFinder is aimed at finding code in large scale systems.
Number of preparation steps	None.
Time Overhead	Minimum. CCFinder has been developed with speed in mind.
Visualisation Features	CCFinder comes with a Java application called Gemini. Gemini provides interprets the results as a dot plot. This compares the files with each other. Very similar to DUPLOC. Focusing in on a dot reveals the actual code.
Documentation level	Two Files. A text file describing how to use the command line driven CCFinder and a HTML page on how to use Gemini. Both are detailed.
Support Level	Unknown.
Learning Curve	Higher than the other tools as there is more functionality with the addition of Gemini.
User Interface	CCFinder is a simple command line interface and Gemini uses a GUI.

Table 6-5 Evaluation results for CCFinder.

CCFinder (see table 6.5) provides an efficient and scalable clone detection method. With the addition of Gemini, the detection of clones is fairly straightforward. Gemini also allows the user to change the settings, making clone detection much simpler. Although the additional visualisation features provided means the learning curve for the tools is steeper than the other tools.

6.4. Comparison of different tools output

Presented here are the results from executing each clone detection tool from the three case studies. For the following experiments the minimum size of clone considered (the threshold) was set to 20 lines. This value was eventually chosen as the best cut off point between wasting time reading through hundreds of spurious matches and missing valuable clones. Figures 6.2 to 6.4 are bar charts showing the results obtained for the case studies. Alongside the potential clones detected by each of the clone detection tools is the number of actual clones. These actual clones are the number of potential clones that were manually verified using the method described in section 4.7.1.



	CCFINDER	CLONEDR	COVET	JPLAG	MOSS
Potential Clones	272	20	51	71	26
Actual Clones	144	20	34	49	26

Figure 6-1 Clones identified for GraphTool case study

Figure 6.1 shows the clones identified by each of the clone detection tools from GraphTool. These results show clearly that CCFinder identified a much greater (nearly 4 times greater) number of potential clones (272) than any of the other tools. JPlag identified the second largest number of clones with 71. Following this Covet identified 51, Moss 26 and CloneDr identified the least number of potential clones (20). An interesting statistic to investigate is the percentage of actual clones. CloneDr and MOSS identified the lowest number of clones but achieved a 100% precision

because every potential clone identified was verified as an actual clone. As the number of clones identified increased it seems the general trend is the percentage of actual clones decreased. Covet and JPlag's precision ratings were 68 and 69% respectively. Out of the potential 272 clones identified by CCFinder only 52% were verified as actual clones.

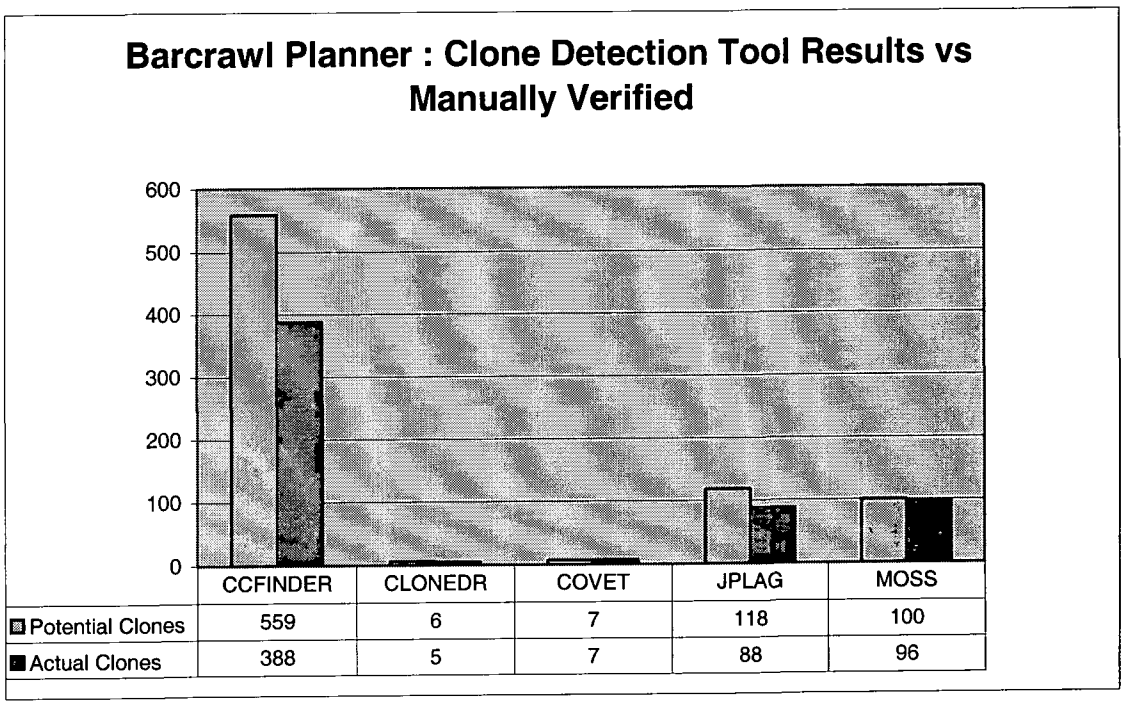
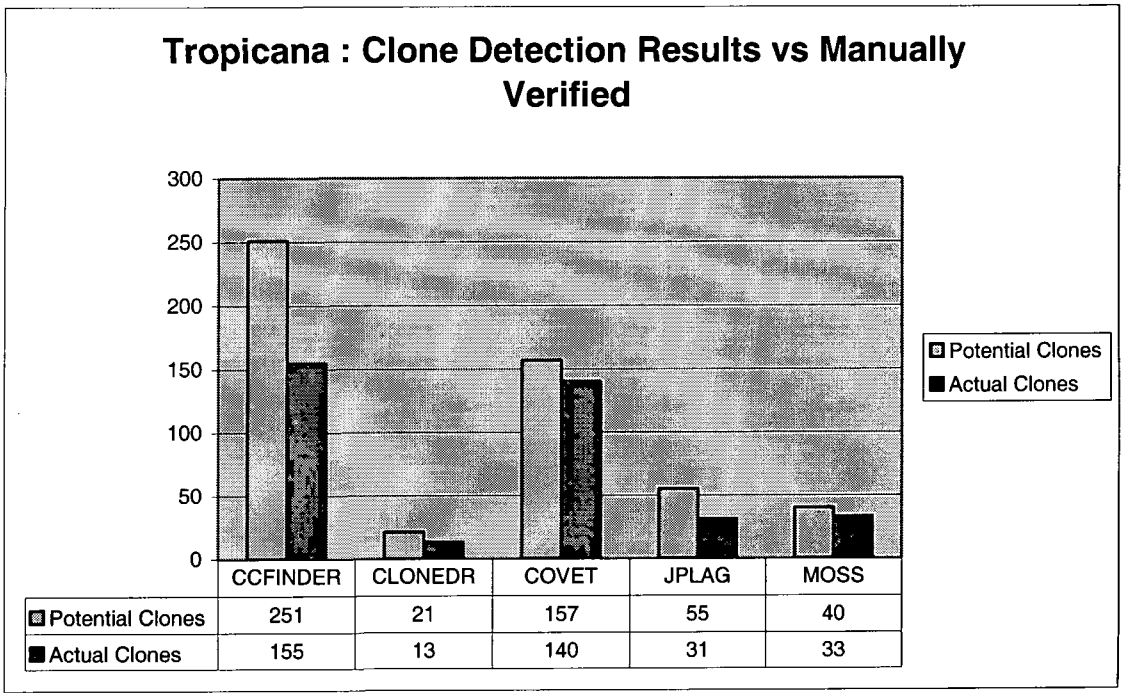


Figure 6-2 Clones identified for the Barcrawl planner case study

Figure 6.2 shows that as with GraphTool CCFinder identified a much greater number of potential clones than the other tools (559). However, in this case study CCFinder identified nearly 5 times as many clones as JPlag. JPlag was the tool that found the second highest number of potential clones with 118. MOSS found slightly less with 100. CloneDr and Covet identified a very small number of clones, 6 and 7 respectively. As with the GraphTool case study, CloneDr identified the least number of potential clones. This low figure for Covet is in contrast as it identified the second highest number of clones in the GraphTool case study. Examining the percentage of potential clones that were verified as actual clones shows that CCFinder's reliability rate increased by 17% from the GraphTool case study to 69% for the Barcrawl planner. Although MOSS and JPlag identified a similar number of potential clones there was a significant difference in their reliability ratings. 96% of MOSS's potential clones were actual clones whilst only 74% of JPlag's were actual clones.

CloneDr and Covet had very high reliability rates, 83% and 100%. CloneDr's reliability rating is lower but considering that it only identified 6 potential clones the inclusion of only 1 false-positive has a significant impact on the reliability rate.



	CCFINDER	CLONEDR	COVET	JPLAG	MOSS
Potential Clones	251	21	157	55	40
Actual Clones	155	13	140	31	33

Figure 6-3 Clones identified for the Tropicana case study

Figure 6.3 shows that CCFinder has again identified the greatest number of potential clones of all the tools (251). As with GraphTool, Covet has identified the second highest number of potential clones with 157. The difference between the number of potential clones identified by CCFinder and Covet is the smallest of all the case studies. JPlag identified the lowest number of clones for all the case studies with 55 potential clones. MOSS identified 40 potential clones, which is lower than the Barcrawl Planner but greater than GraphTool. CCFinder has a slightly lower reliability rating for Tropicana with 61% than the Barcrawl Planner (69%) but this is still greater than the 52% reliability achieved in GraphTool. CloneDr again, as with all the other case studies, identified the least number of potential clones (21). One particular difference here is the much lower reliability rating of 61%. MOSS identified 40 potential clones and for this case study has a fairly high reliability rating of 82%. JPlag identified more potential clones than MOSS with 55 but less actual clones (31) hence its reliability rating is much lower at 52%. Covet's reliability rating of 89% is again high. The number of actual clones identified by

Covet was only 15 less than CCFinder this despite identifying 94 less potential clones. These two facts indicate that Covet was the most successful of all the tools for this particular case study.

6.5. Size Breakdown of each tool's results

This section will attempt to analyse the sizes (in terms of LOC) of clones each tool identified as a potential clone. Obviously this figure is bounded by the minimum threshold set along with the size of the source code being examined. Calculating the statistics was a simple case of iterating through the CodeRegionPairs generated and totalling up the largest CodeRegion's size and keeping track on the maximum value within that set.

Tool	GraphTool		Barcrawl Planner		Tropicana	
	Max	Mean	Max	Mean	Max	Mean
CloneDr	**100**	37	**50**	29	**497**	61
Covet	**114**	37	**22**	21	**169**	40
JPlag	**78**	33	**112**	37	**426**	44
CCFinder	**80**	30	**50**	30	**423**	31
MOSS	**57**	27	**100**	31	**275**	41

Table 6-6 LOC statistics for the clones identified for the case studies

Table 6.6 shows the mean and maximum sizes for the potential clones identified for each of the case studies. There appears to be no trend to the size of clones identified by each tool. Whereas for each of the case studies it was evident that CCFinder consistently identified the greatest number of potential clones. For example for the Graph Tool case study CloneDr and Covet identified, on average, larger clones whereas in the Bar crawl case study the opposite is true. Table 6.6 shows that within Tropicana there was one very large clone. Three tools, CloneDr, JPlag and CCFinder found at least one clone with a size in excess of 420 lines of code however their mean sizes are not significantly higher than the means for the other case studies. Moss's maximum clone size is smaller than the other tools but in comparison to the maximum for the other case studies is very high. Covet's maximum for Tropicana is the odd one out at 169. This is an expected outcome if the clone spans more than one

routine. As Covet is restricted to cloning within routines it could only find a portion such a clone.

One of the most useful measures for any maintainer is the total percentage of a system that can be reduced through clone removal. Kamiya [Kam02] describes this as the potential deflation made possible by removing clones from a system. Figures 6.4 to 6.6 show the percentage of cloned lines for the three case studies according to each of the clone detection tools.

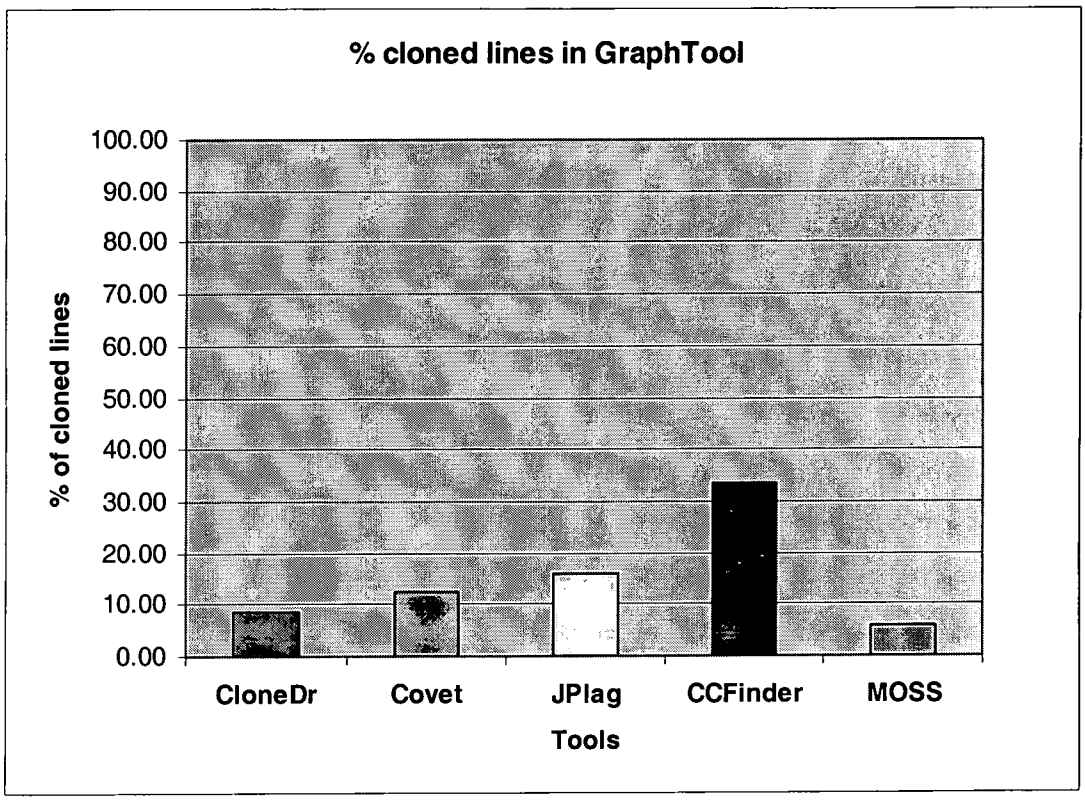


Figure 6-4 Percentage of cloned lines in GraphTool

Figure 6.4 shows that the percentage of code that can be potentially removed according to all the tools ranges from approximately 6% (MOSS) to 32% (CCFinder).

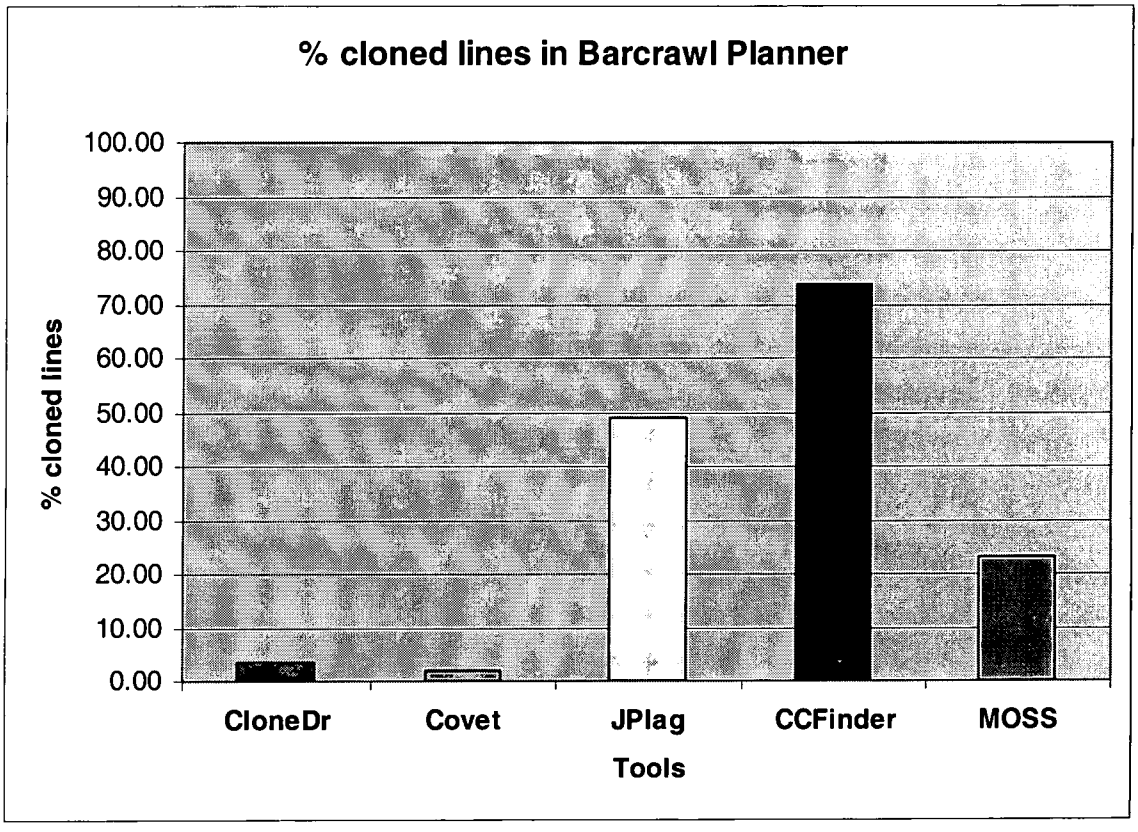


Figure 6-5 Percentage of cloned lines in the Barcrawl Planner

Figure 6.5 shows that compared with GraphTool a much greater percentage of the Barcrawl Planner is cloned and can therefore potentially be removed. According to the clone detection tools the estimate ranges from 2% to 73%. This much greater gap in estimates from the tools is expected considering the number of clones identified for Barcrawl Planner (figure 6.2).

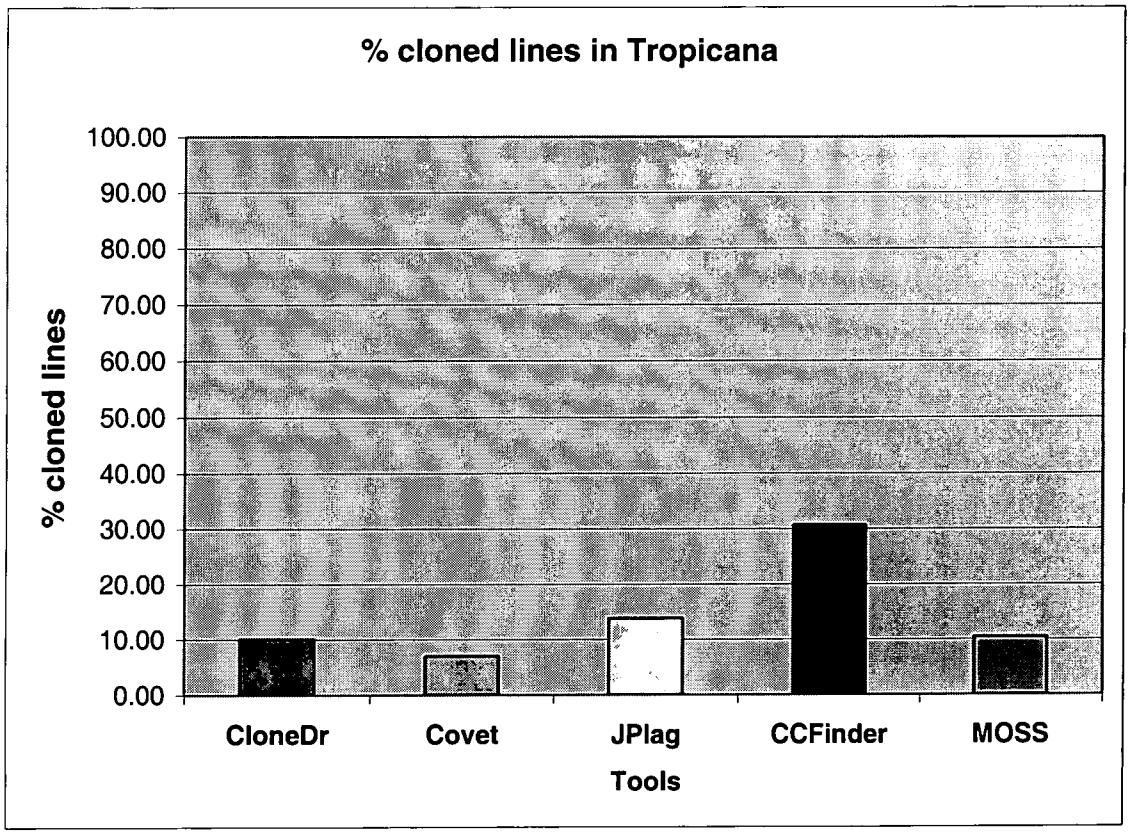


Figure 6-6 Percentage of cloned lines in Tropicana

Figure 6.6 shows overall a more conservative estimate of cloning percentage for Tropicana than the Barcrawl Planner. The estimates from the tools range from 8% (Covet) to 31% (CCFinder).

6.6. Unique Clone Classes within results

Kamiya [Kam02] describes series of related clones as clone classes and Baxter [Bax97] as idioms. They both relate to an original piece of logic which has been proliferated throughout a system. Within each clone class are regions of code that belong to the same cloning relationship and therefore identifying these allows maintainers to identify related clones and carryout impact analysis. For example, if one piece of code has an error in it there is a good chance that its clones will also contain that same error. A class's length (or size) is the number of CodeRegions that belong to that class's cloning relation.

Tables 6.7 to 6.9 describe the number of clone classes found for each of the case studies identified by each clone detection tool. They also contain the maximum and mean lengths. The term *length* is defined by Kamiya [Kam02] to represent the number of clones within a clone class. However because of the confusion with may arise (because length can also mean the size of code regions) the term *set size* will be used instead.

Tool	# Clone Classes	Max. Set Size	Mean Set Size
CloneDr	18	2	1.05
Covet	20	6	1.7
JPlag	33	14	3.09
CCFinder	59	90	6.81
MOSS	15	9	2

Table 6-7 Clone classes for GraphTool

Table 6.7 shows that the number of clone classes each tool identified for the GraphTool case study varied greatly. CCFinder found nearly 60 unique classes of clones whereas MOSS found only 15. CloneDr and Covet found a similar number of classes (18 and 20 respectively) but on average the classes in Covet were larger. CCFinders clone classes contained an average of nearly 7 clones in them which was more than double those found by JPlag. JPlag identified the second highest number of clone classes (33) and these classes had a mean set size of approximately 3. The longest clone class contained 90 related CodeRegions and was identified by CCFinder.

Tool	# Clone Classes	Max Set Size	Mean Set Size
CloneDr	5	2	1.2
Covet	3	2	1.67
JPlag	38	36	4.66
CCFinder	46	148	15.98
Moss	21	36	6.42

Table 6-8 Clone classes for Barcrawl Planner

Table 6.8 shows that the clone detection tools identified less clone classes in the Barcrawl Planner case study than GraphTool. CCFinder ,as with the GraphTool case study identified the most and largest clone classes. It identified 13 less clone classes for the Barcrawl Planner than GraphTool. However, the clone classes were much larger with nearly 16 instances. All the clone detection tools found larger clone classes for the Barcrawl Planner in comparison to GraphTool.

Tool	# Clone Classes	Max Set Size	Mean Set Size
CloneDr	19	2	1.05
Covet	17	15	2.65
JPlag	29	10	2.34
CCFinder	68	44	4.98
Moss	24	8	1.96

Table 6-9 Clone classes for Tropicana

Table 6.9 shows that the clone detection tools identified a similar number of clone classes in Tropicana as they did in GraphTool. CCFinder again found the most clone classes of all the case studies with 68. However, CCFinder's classes were smaller in Tropicana with an average set size of approximately 5. CloneDr's results were almost identical in Tropicana to those in GraphTool. The only difference was in Tropicana, CloneDr identified one more clone class, the maximum and mean set sizes of these clone classes were identical to GraphTool.

6.7. Replication Within and Across Programs

Burd [Bur97] distinguishes between clones that appear throughout the same program and those that appear in other programs within a system. Replication across programs could indicate the need for a new program which can be included in each source. Replication within a program can be tackled by applying a unifying method. Tables 6.10 to 6.12 show the results for each system. It must be noted that both JPlag and MOSS, due to the specific nature of their clone detection (i.e. academic plagiarism), only look for replication across programs (because there is no rule against students copying their own code) and so these figures have been marked N/A in the tables 6.10 – 6.12.

Tool	Across Programs (%)	Within Programs (%)
CloneDr	35	65
Covet	25	75
CCFinder	63	37
JPlag	100	N/A
MOSS	100	N/A

Table 6-10 Percentage of identified clones identified within / across programs for GraphTool

Table 6.10 shows that Covet found the highest proportion of replication within programs followed by CloneDr. However, CCFinder differs from the other two tools as it finds a higher proportion of replication across programs.

Tool	Across Programs (%)	Within Programs (%)
CloneDr	83	17
Covet	57	43
CCFinder	98	2
JPlag	100	N/A
MOSS	100	N/A

Table 6-11 Percentage of identified clones identified within / across files for Barcrawl planner

Barcrawl Planner has some very interesting results. Table 6.11 shows that all the tools identified a higher proportion of replication across programs. Covet and CloneDr identified a small number of clones (see figure 6.2) for Barcrawl Planner and therefore their results are probably not useful in attempting to evaluate the replication across and within programs. This low number of clones is in contrast to CCFinder's results as it retrieved its largest number of potential clones for all the systems. 98% of these were found across programs. This would seem to indicate that there is a much higher replication of cloning across programs as opposed to within programs. Although looking at the difference in replication across and within programs for JPlag and MOSS is possibly not useful it may be useful to note that JPlag and MOSS identified more clones in Barcrawl Planner than any of the other systems.

Tool	Across Programs (%)	Within Programs (%)

CloneDr	62	38
Covet	22	78
CCFinder	77	23
JPlag	100	N/A
MOSS	100	N/A

Table 6-12 Percentage of identified clones identified within / across files for Tropicana

From table 6.12 there appears to be no pattern to the results for Tropicana. CloneDr and Covet are almost symmetrically different with Covet finding 22% of its clones across and 78% within programs. CloneDr finds 77% of its clones across and 23% within programs. CloneDr appears to agree with CCFinder by showing a higher proportion of cloning across programs.

6.8. Intersection between each tool's results

One of the aims of this thesis is to investigate the similarity in the results returned by various clone detection tools. In other words the experiment is trying to establish whether the different clone detection tools are finding the same clones. In order to investigate these issues a decision has to be made as to what constitutes a match. This is discussed in the Implementation Chapter in the Section 4.5. A specific CodeRegionPair **A** is considered a match with another CodeRegionPair **B** if **A** overlaps more than 60% of **B**. This overlap relation is not necessary reflective. Because of the minimum percentage criteria, **A** overlaps **B** could be true but **B** overlaps **A** false. It is dependent on the size of each CodeRegion and the number of lines shared between them.

Tables 6.13 – 6.15 present the results from the experiments carried out on the three case studies to see if the clones identified by each tool were similar. In each table the first column lists the clone detection tool and the row next to it (column two) contains the total number of potential clones returned by that tool. Subsequent columns (headed by the name of other clone detection tools) contain the number of potential clones identified by both the clone detection tool on that row and the other clone detection tool named in the column. Next to this figure is its percentage of the total results (in brackets).

For example, the second row of table 6.15 shows the results for CloneDr in the GraphTool case study. The second column (under Total Results) shows that CloneDr identified 20 clones in GraphTool. Next to this is an x because comparing CloneDr's clones with themselves is not useful. In the fourth column (labelled Covet) is the number of clones identified by CloneDr that Covet also identified (in this case 8). The percentage in brackets is the number of CloneDr's clones that Covet also found, divided by the total number of clones identified by CloneDr (40%).

	Total Results	CloneDr	Covet	JPlag	CCFinder	MOSS
CloneDr	**20**	x	8 (40%)	3 (15%)	7 (35%)	2 (10%)
Covet	**51**	6 (11%)	x	5 (10%)	15 (29%)	2 (4%)
JPlag	**72**	3 (4%)	6 (8%)	x	38 (53%)	12 (17%)
CCFinder	**272**	9 (3%)	20 (7%)	35 (13%)	x	14 (5%)
MOSS	**26**	3 (11%)	2 (8%)	14 (54%)	17 (65%)	x

Table 6-13 Clone detection tool intersection for GraphTool

Table 6.13 shows the intersection between the clones identified for the GraphTool case study. There appears to be little intersection between each of the tools. The tables show that with its large recall, CCFinder has proved the most successful at finding the other tool's clones. CCFinder found 65% of MOSS's clones and 53% of the clones identified by JPlag. Covet appears to have identified very different clones to the other tools in GraphTool, with the exception of CloneDr. Covet found less than 10% of each of the other tools' clones. CloneDr also found a very small percentage of the other tool's clones but this is expected as it identified the lowest number of clones for GraphTool.

	Total Results	CloneDr	Covet	JPlag	CCFinder	MOSS
CloneDr	**6**	x	2 (30%)	2 (30%)	5 (83%)	0 (0%)
Covet	**7**	2 (28%)	x	0 (0%)	1 (14%)	0 (0%)
JPlag	**118**	2 (2%)	0 (0%)	x	90 (76%)	37 (31%)
CCFinder	**559**	4 (1%)	1 (0.01%)	91 (16%)	x	79 (14%)
MOSS	**100**	0 (0%)	0 (0%)	40 (40%)	84 (84%)	x

Table 6-14 Clone detection tool intersection for Barcrawl Planner

Table 6.14 shows the intersection between the clones identified for the Barcrawl Planner case study. In this instance, there is less intersection between the tools than in the GraphTool case study. One interesting fact is that JPlag and Covet found no clones in common. Also CloneDr and JPlag only agreed on two clones. This was also the case with MOSS and Covet and MOSS and CloneDr. Obviously the low number of clones identified by Covet and CloneDr meant that they were unable to find a significant number of the other tool's clones. As with the GraphTool case study CCFinder found the highest percentage of other tools' clones. It found 83% of CloneDr's clones, 76% of JPlag and 84% of MOSS's clones. The only exception was Covet these two tools only agreed on one clone.

	Total Results	CloneDr	Covet	JPlag	CCFinder	MOSS
CloneDr	**21**	x	3 (14%)	6 (28%)	17 (81%)	7 (33%)
Covet	**157**	4 (2%)	x	6 (4%)	43 (27%)	8 (5%)
JPlag	**55**	4 (7%)	14 (25%)	x	37 (67%)	24 (43%)
CCFinder	**251**	21 (8%)	32 (13%)	39 (15%)	x	36 (14%)
MOSS	**40**	10 (25%)	16 (40%)	26 (65%)	38 (95%)	x

Table 6-15 Clone detection tool intersection for Tropicana

Table 6.15 shows an overall higher level of intersection between the clone detection tools' clone sets for Tropicana than the other two case studies. Yet again CCFinder found a high percentage of the clones identified by other tools. With the exception of Covet it found between 67 – 95% of the other tool's clones. Throughout all three case studies CCFinder has found a lower percentage of Covet's clones than any other tools which indicates that the two tools identify very different clones. Conversely CCFinder consistently identifies more of MOSS's clones than it does for the other tools. CCFinder found 65% of MOSS's clones in GraphTool, 83% in the Barcrawl Planner and finally 95% in the Tropicana case study. This indicates that CCFinder identifies similar clones to MOSS.

6.9. Clone by Clone visualisation

The clone-by-clone visualisation technique (Appendix A) could be used to spot which clones are found by the majority if not all of the clone detection tools. An

enhancement to this would be the ability to focus in on the actual source code identified by each row in the table.

6.10. Precision and Recall Analysis

Precision and recall are two widely used metrics in the evaluation of information retrieval systems such as search engines. Recall can be used as a measure of the proportion of actual clones identified by the tool. It is unfeasible to find all clones within a case study as it would require an extremely large amount of code reading. This degree of accuracy was not deemed necessary given the breadth of the analysis covered within the thesis. Precision looks at the proportion of potential clones identified that were verified as actual clones. These metrics obviously depend on the manual verification process which was explained in Chapter 4, Section 4.5.

	CloneDr	CCFinder	Covet	JPlag	MOSS
Precision (%)	100	52	52	69	100
Recall (%)	8	59	14	20	11

Table 6-16 Precision and recall results for the GraphTool case study

Table 6.16 shows that both MOSS and CloneDr achieved a precision value of 100%. However, CloneDr and MOSS's recall is quite low, 8% and 11% respectively. JPlag managed the second highest precision value with 69% and has a higher recall value of 20%. Covet and CCFinder recorded a precision value of just 52%. However, CCFinder had the largest recall value, with 59%, whereas Covet scored quite poorly with a recall of just 14%.

	CloneDr	CCFinder	Covet	JPlag	MOSS
Precision (%)	83	69	100	74	96
Recall (%)	1	79	1	18	19

Table 6-17 Precision and recall results for the Barcrawl planner case study

In contrast to the results for GraphTool, table 6.17 highlights Covet showed a significant increase in its precision for the Barcrawl Planner. However although all the potential clones identified were verified it only identified 1% of the total clone set. This small recall value was shared with CloneDr whose precision also dropped

from 100% in GraphTool to 83% in the Barcrawl Planner. MOSS's precision fell in this case study by 4% but its recall value increased by 8%. JPlag's values, conversely, showed a 5% increase in precision and a 2% drop in recall (when compared to GraphTool). CCFinder's precision and recall values were both significantly better in the Barcrawl Planner with an increase of 17% in precision and 20% in the recall value.

	CloneDr	CCFinder	Covet	JPlag	MOSS
Precision (%)	56	61	90	56	82
Recall (%)	4	48	43	9	10

Table 6-18 Precision and recall results for the Tropicana case study

Table 6.18 shows a large increase in Covet's recall value (43%) whilst still returning a high degree of precision (90%). CloneDr's precision in contrast dropped to 56% and it also only achieved a 4% recall. CCFinder again had the highest recall with 48% but its precision was only 61%. MOSS and JPlag performed considerably worse with lower recall (82% and 56% respectively) and lower precisions (9% and 10%) than they managed for any of the other case studies.

6.11. Size Threshold Sensitivity

For each of the experiments described so far a minimum size threshold of 20 was set. However, low recall values for certain clone detection tools using a threshold of 20 may indicate that one size threshold may not be an appropriate approach. To investigate this further, clone detection was carried out using thresholds ranging from 10 to 30. The results are presented in figures 6.7 to 6.9. Figures 6.10, 6.11 and 6.12 show the percentage of clones identified by each tool as the minimum clone size threshold increases. The minimum size thresholds are separated into quartiles along the x-axis. Each bar shows the percentage of clones identified for each tool during the increase from the minimum size threshold of ten for that range.

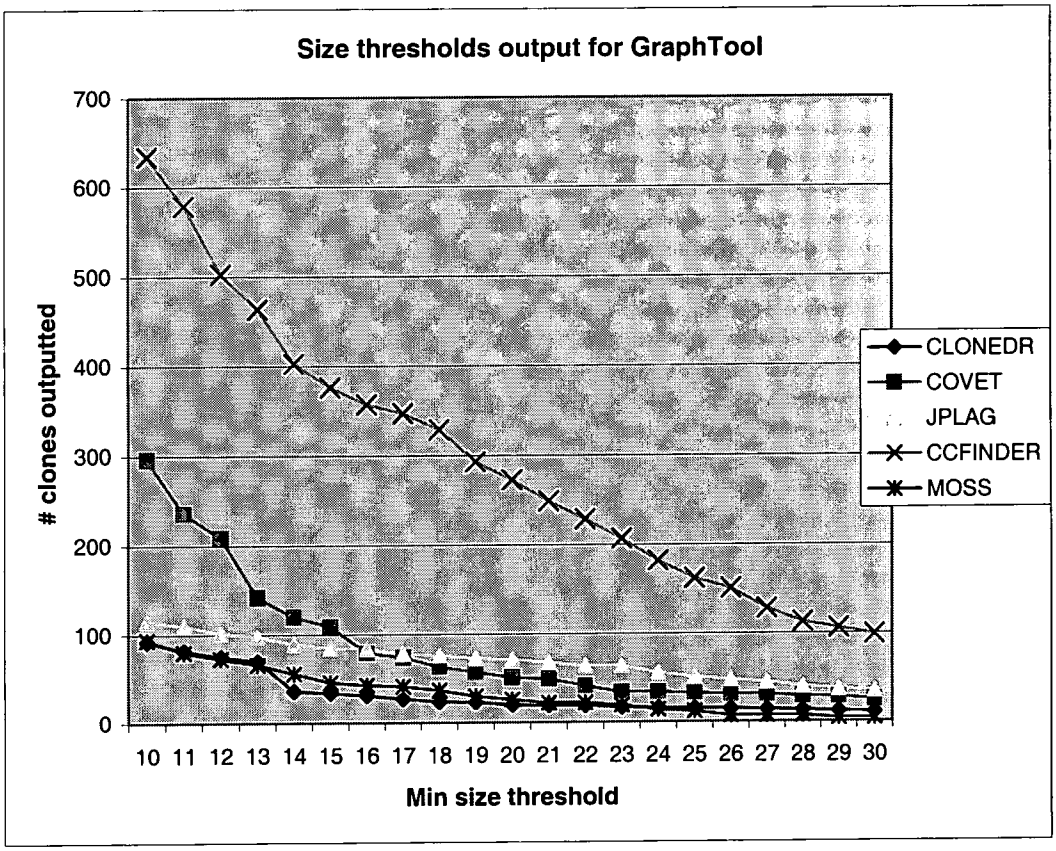


Figure 6-7 Results from minimum clone size threshold experiments for GraphTool

Figures 6.7 – 6.9 show the actual number of clones identified by each tool for the three case studies GraphTool, Barcrawl Planner and Tropicana respectively. Figures 6.10 – 6.12 show the decrease in percentage of the number of clones outputted by each of the tools for the three case studies. The percentage is calculated from an "original" clone set where the minimum size threshold was set to 10.

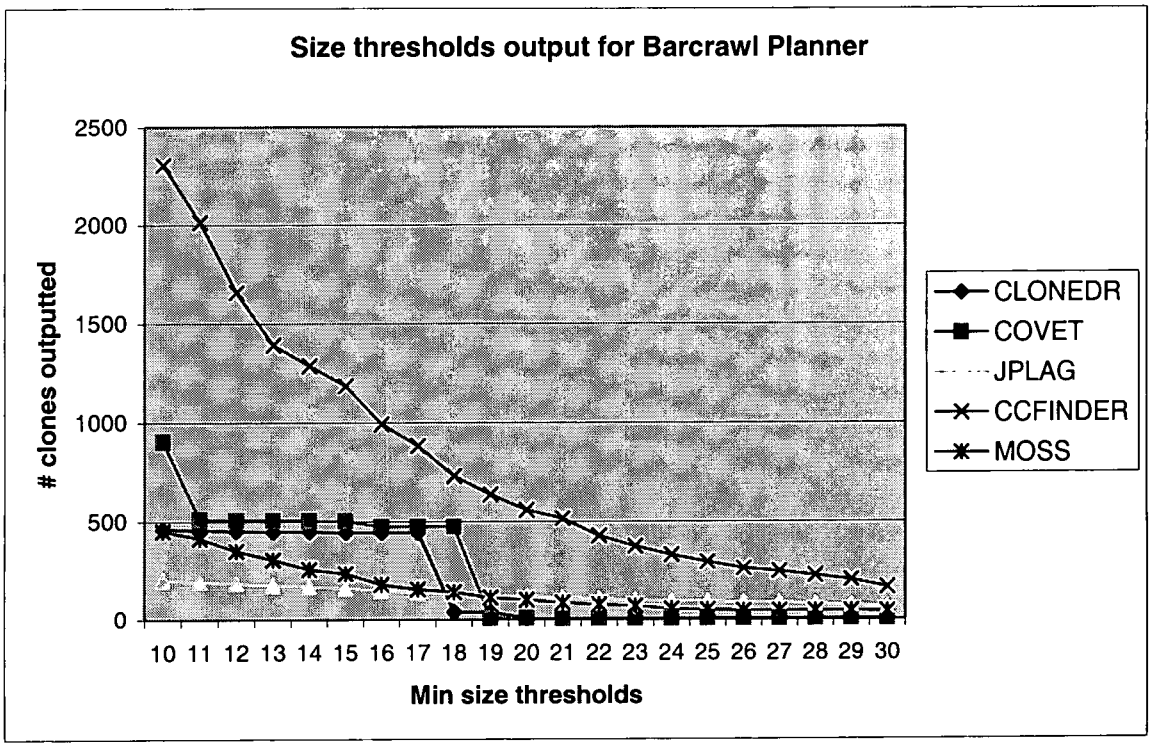


Figure 6-8 Results from Size threshold experiments for Barcrawl Planner

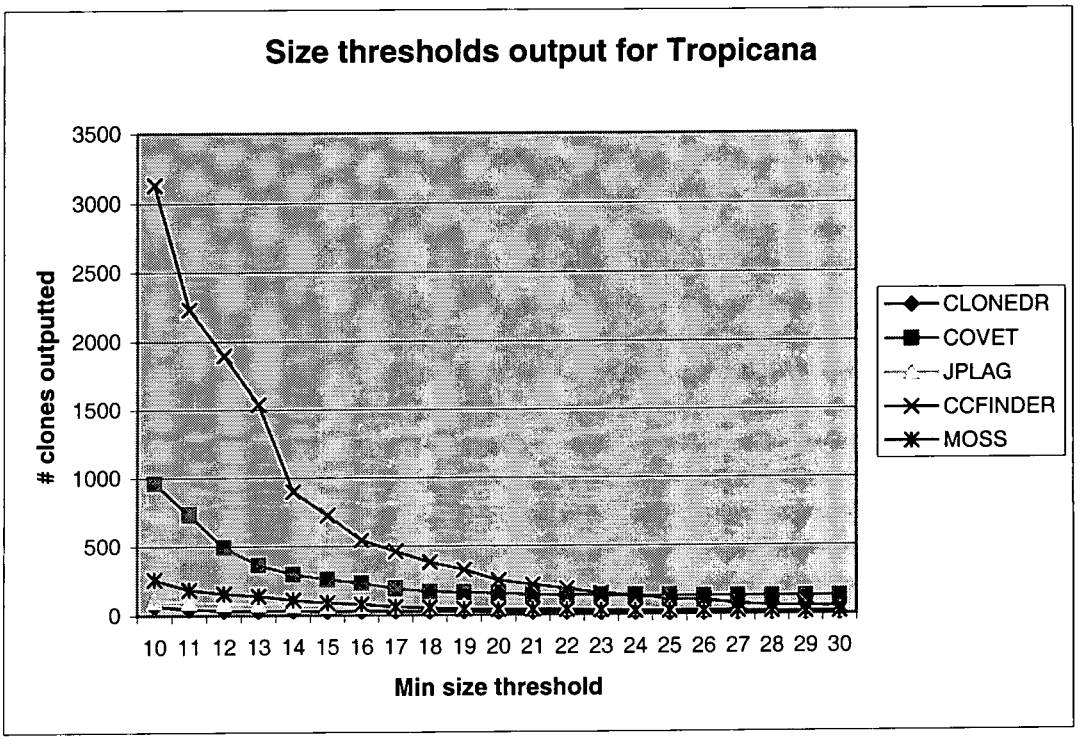


Figure 6-9 Results from Size threshold experiments for Tropicana

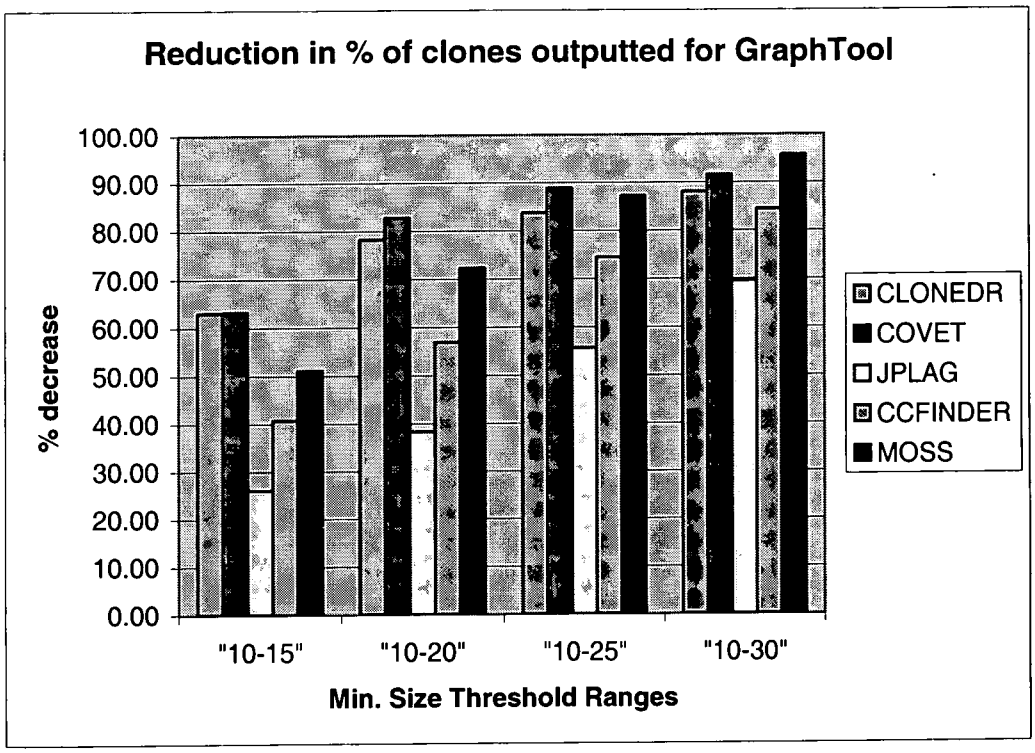


Figure 6-10 Reduction in % of clones outputted for the minimum size threshold ranges for GraphTool

Figures 6.8 and 6.12 show interesting results for the number of clones identified for the Barcrawl Planner case study. The result indicates that clones identified for Barcrawl Planner were far more sensitive to increases in the minimum size threshold. Figure 6.11 shows a remarkable change in the results for CloneDr. In the first quartile there was only a reduction of approximately 4% from 462 clones outputted to 445. This figure had dropped to just 6 clones identified using a minimum size threshold of 20 a decrease of approximately 99%. Figure 6.11 shows that there are two large drops in the number of clones identified at threshold settings of 18 (from 444 to 38) and then again at 20 (from 38 to 6). Finally by the fourth quartile nearly 100% of the clones had been filtered out and only one clone remained with a size greater than 30 lines. Again as with the GraphTool case study, Covet's results followed a similar pattern to CloneDr's. The gap between the 1st and 2nd quartiles were not as great as 44% of Covet's clones were already filtered out by 15. Examining figure 6.8 it is clear that the increase from 10 to 11 filtered out approximately 40% of Covet's clones. This reduction was low and steady until a minimum size threshold of 19 was reached. A drop from 473 clones identified to just 7 clones at 19 explains the reduction figure of 99% in the second quartile. By the third quartile (and more precisely a minimum size threshold of 23) 100% of Covet's

clones were filtered out making Covet the most sensitive to increases in the minimum size threshold for Barcrawl Planner. Results for CCFinder and MOSS were very similar both lost 48% of their clones in the first quartile. CCFinder filtered out slightly less than MOSS in the second quartile 76% as opposed to 78%. The third quartile again showed CCFinder filtering out slightly less than MOSS with a loss of 87% to MOSS's 89%. Overall, CCFinder filtered a higher percentage of clones by 30 (93%) whereas MOSS only lost 91% of its clones in total. JPlag (as with GraphTool) was the least sensitive to the increases in minimum size threshold. 68% of its clones were filtered out by 30, slightly less than with GraphTool. In the first quartile only 20% of its clones were lost, by the second quartile the reduction was 41% and 55% in the third quartile.

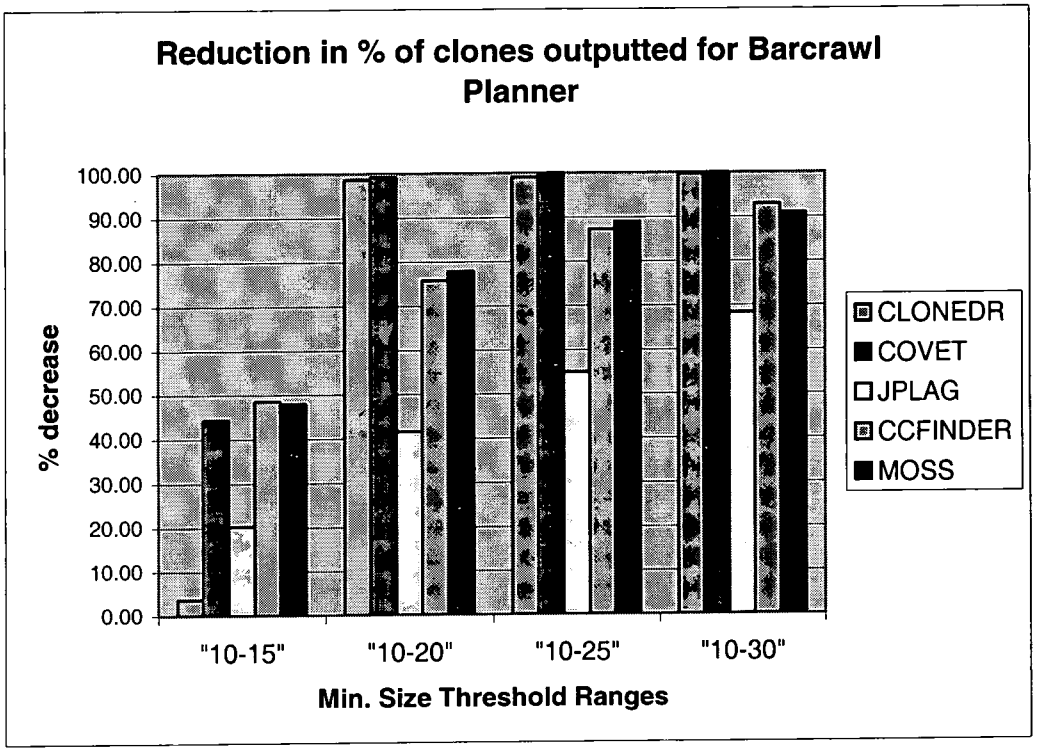


Figure 6-11 Reduction in % of clones outputted for the minimum size threshold ranges for Barcrawl Planner

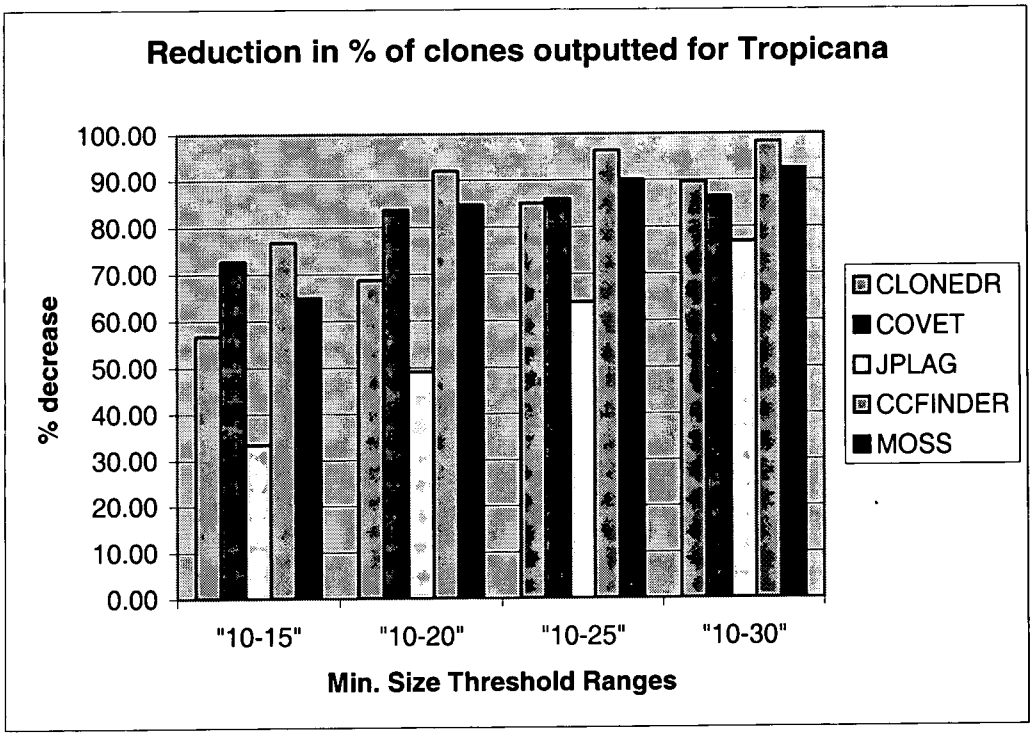


Figure 6-12 Reduction in % of clones outputted for the minimum size threshold ranges for Tropicana

Figures 6.9 and 6.12 show a steadier reduction in the number of clones filtered out for Tropicana. CloneDr especially has a steadier filtering out of the clones produced; There are no large increases in the number of clones filtered out from one size threshold to the next. Figure 6.12 shows that 57% of its clones are filtered out in the first quartile this increases to 69% in the second, 85% in the third and 89% in the fourth. This is a flattening out of the number of clones being filtered out (see figure 6.6) the largest drop is seen at 11 with the number of clones identified decreasing from 67 to 47. This figure then decreases to 34 clones by 12 but from 12 to 30 the largest drop in number of clones identified leaving just 4 clones left.

The reduction in the number of clones identified for Covet is greatest in the first quartile, losing 73% of the clones originally identified with a threshold of 10 by 15. In particular, the increase in threshold from 10 to 11 and 11 to 12 sees a large reduction in the number of clones identified. After the first quartile the number of clones filtered out is much smaller in the second quartile. A further 11% is filtered out taking the total to 84% and this reduction increases by 2% in the third quartile and remains at 86% in the fourth and final quartile. MOSS filters out 92% of the clones identified by the final minimum size threshold of 30. Initially with a threshold

setting of 10 MOSS identifies 262 clones and at 15, 92 clones this shows a reduction of approximately 65%. Significant loss is also made in the 2nd quartile with 85% of the original clone set filtered out. This loss is steadier in the 3rd and 4th quartiles increasing by 6% to 90% in the 3rd and finally 92% in the 4th quartile.

JPlag continues to be the least sensitive to changes in the minimum size threshold having the lowest total reduction (77%) in clones identified from the original clone set. It also filters out significantly less than all the other clone detection tools in the 1st quartile at 33%. By the 2nd quartile it has filtered out 49%, 64% by the 3rd and 77% by the 4th.

As with the Barcrawl Planner case study CCFinder filters out the greatest percentage of clones from the original clone set and therefore is the most sensitive to increases in the minimum size threshold. CCFinder actually filters out approximately the same percentage of clones in its 1st quartile as JPlag filtered out in total (approximately 78%). CCFinder's larger recall means that 2232 clones are lost in its 1st quartile whereas just 83 clones were lost for JPlag. The number of clones filtered out for CCFinder increases in the 2nd quartile to 92%, which is greater than or equal to every other tools' total reduction in percentage terms. In the 3rd quartile 96% of the clones in CCFinder's original clone set have been filtered out and by the 4th quartile 98% of CCFinder's original clone set are lost.

6.12. Chapter Summary

This chapter presented the results from each of the case studies of the experiments. It details the results of the qualitative experiments. The next chapter provides further evaluation of the results from the case studies.

7. Evaluation

The purpose of this study was to investigate different techniques and tools for removing cloned (and therefore redundant) code. This particular investigation can be split into two main parts. Firstly the creation of an experimental clone detection tool called Covet. Secondly a wider study (which included Covet) attempted to discover which tools best aid preventative software maintenance.

Clone detection tools address code optimisation, which is a specific part of preventative maintenance. Less code means less code reading which is one of the most time and labour intensive aspects of software maintenance. Chapter 3 outlined a set of ten hypotheses which were to be tested through a series of experiments. This chapter first evaluates the development of Covet and then examines the results for each of the experiments carried and relating them to the hypotheses they tested.

7.1. Covet's development evaluation

Results from Covet's initial metrics experiments would seem to justify Mayrand's [May97] use of a test group to select the metrics used in clone detection. With a precision of 83% and the second highest recall (20%) of all the metric groups experiment three would seem to point out that any further work in refining Covet should again make use of a preliminary study using sample programs which are known to contain significant cloning. It appears that the attempts to automatically generate thresholds were not a success. This may not be surprising when considering that the other clone detection tools can find matches that do not conform necessarily to routines. Other clone detection tools might find a cloning relationship between the last 30 lines of two 50-line routines. However, if the first 20 lines in these routines are not clones and are completely different then this will produce unhelpful threshold settings that will not identify actual clones. One solution to this might be in future to ensure that when the CodeRegions from other tools are "rounded" up or down to whole routines that at least 90% of the routine is overlapped. Also within the automatically generated clones there were clones that had not been manually verified (experiment 5). These may have included false positives and so have been unhelpful.

This chapter will now focus on the hypotheses described in the chapter 4.

7.2. Hypotheses 1 and 2

Hypothesis 1. Each clone detection tool will output different proportions of cloning for the same case study. **TRUE**

Hypothesis 2. Case studies of differing size and development background will output different proportions of cloning for the same clone detection tool. **TRUE**

Hypothesis 1 predicts that the percentage of cloned code identified by each tool will vary significantly and the results for each case study prove this to be true. Throughout each case study CCFinder identified a much larger set of potential clones than any of the other tools whereas CloneDr consistently identified the least number of clones. One interesting fact was that out of the three case studies none of the tools identified the same number of clones. This then poses a dilemma for a maintainer. Which tool should be believed? One use for a clone detection tool might be to run as a preliminary stage of re-engineering. The percentage of cloning can be used to indicate the state of a legacy software system. Johnson [Joh94] points out that cloning is a symptom of "*software ageing*". The maintainer must decide whether to trust a diagnosis from CloneDr or the to believe the percentage of cloning found by another tool such as CCFinder. An obvious solution, which is analogous to the medical profession, is to seek a second, third or even fourth opinion. If two or more tools agree that there is a high proportion of cloning within a legacy system then this will provide the evidence a maintenance team requires to justify some form of modification.

Within this thesis clones were categorised as replication within a program and across programs. From the case studies it appears that some of the tools tend towards identifying a majority of clones either within a program or across programs. This is certainly the case with JPlag and MOSS which only attempt to identify replication across programs. Furthermore, CCFinder consistently identified the majority of its clones across programs, 68% in GraphTool, 98% in the Barcrawl Planner and 77% in Tropicana.

Case study	Total Potential Clones	Total Actual Clones
GraphTool	440	273
Barcrawl Planner	779	584
Tropicana	524	372

Table 7-1 Total potential and actual clones for each case study.

Hypothesis 2 predicts that the development environment and size of a case study will have a significant effect on the proportion of clones within it. Table 7.1 summarises the total number of potential clones identified for each of the case studies alongside the total number of actual clones. Of the three case studies used in this thesis the majority of the clone detection tools found that the smallest case study in terms of lines of code contains the most clones. GraphTool is the only tool to have been developed without the aid of an Integrated Development Environment and is also the only non-academic project. Despite being nearly twice the size of the Barcrawl Planner, GraphTool contains approximately half the number of clones (both actual and potential). An explanation for this high proportion of cloning might be the inclusion of automatically generated Graphical User Interface (GUI) code. During manual verification of the clones in each case study it was found that a high proportion of the clones detected were in GUI related code. If this is the case then why is the number of clones identified in the Tropicana case study not higher? Tropicana was also developed with the aid of an Integrated Development Environment and is much larger than the Barcrawl Planner case study. An explanation might be that included within Tropicana is an open source project that decrypts and plays MP3 music files. This project has no GUI related source code in it and is 12,710 LOC in size. Subtract this from the total 23,937 LOC and Tropicana is only 11,257 LOC in size.

7.3. Hypotheses 3 and 4

Hypothesis 3. Case studies developed without the aid of an integrated development environment will contain on average clones which are greater in size. **FALSE**

Hypothesis 4. Clones identified using metric comparisons will differ greatly in size from clones identified by tools that directly compare **FALSE**

source code.

Hypotheses 3 and 4 focus on the average size of the clones identified in the case studies. Size in this instance refers to the number of lines of code contained within a clone and was found to be one of the most successful metrics for verification. The larger the sections of code identified as potential clones by a tool the less likely their similarity is coincidental. Taking advantage of this fact allowed the use of a minimum size threshold. Potential clones below a certain threshold were filtered out. Maintainers will also use this threshold because there is very little benefit in the detection and removal of small clones. Large clones were found to be of greater significance and value to software maintenance. These large clones also are better candidates for reuse as they usually captured a whole piece of logic, for example file operations. Maintainers may wish to remove only a percentage of the clones within a case study and prioritising clones on the basis of their size may provide a useful device.

Hypothesis 3 states that case studies developed without the aid of an integrated development environment will contain larger clones. This hypothesis was chosen because it was thought that clones created by a human would contain a significant piece of logic and would therefore be quite large. Integrated development environments are incapable of generating original code and therefore even small sections of code will be clones. However, this was not found to be the case. Results from the size breakdowns of each case study showed although GraphTool contained on average clones of greater length than the clones contained within the Barcrawl Planner case study its clones were much shorter than Tropicana's clones.

Tool name	First Clone (lines cloned)	Second Clone (lines cloned)	Third Clone (lines cloned)
CCFinder	9-66	82-505	
JPlag	9-77	79-504	
CloneDr	9-505		
MOSS	83-357	475-494	
Covet	164-242	264-303	305-472

Table 7-2 Clone detection tools' clones between Backdrop and oldBackdrop

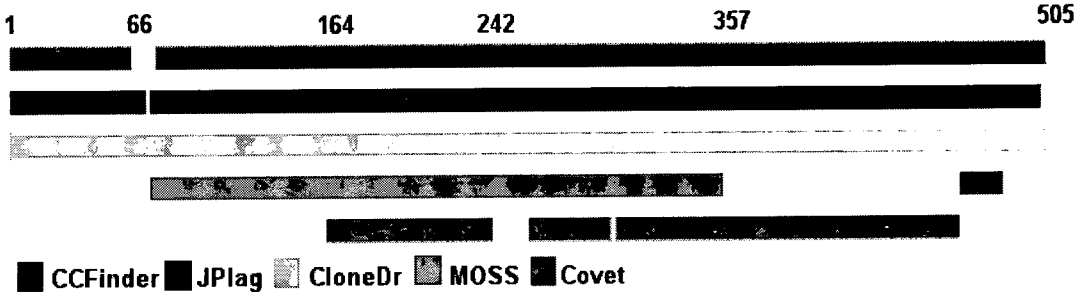


Figure 7-1 Visual representation of table 7.2

Hypothesis 4 tests the differences in the average sizes of clones identified by Covet and the other tools used in the case studies. One obvious reason why this might be the case is the scope at which the metrics were generated and compared. Covet uses metrics generated at the routine level and therefore any clones identified are restricted in size. It is impossible, for example, for Covet to identify a whole file even if it is compared to another identical file. Table 7.2 and Figure 7.1 illustrate this point. Two virtually identical files (Backdrop.java and oldBackdrop.java) were included in the Tropicana case study. Comparing these two files showed that apart from the renaming of the class and constructor there were only two differences; a variable called `delay` set to a different value and a GUI layout statement which had been commented out. Table 7.2 shows how Covet identifies three relatively smaller (79 LOC, 40 LOC and 168 LOC) clones in comparison to the other tools (CloneDr found only 1 clone which was 497 LOC in size). These smaller clones were sub-clones of the longer clone identified by CloneDr.

However the difference observed in the example given in Table 7.2 is not reflective of cloning as a whole. Results from the three case study show that although for the

GraphTool and Tropicana case studies the maximum clone size identified was significantly lower than the other tools this was not reflected in the mean. None of the tools consistently identified clones of the same size. It therefore appears that there is no great difference in the size of clones identified using metric comparisons as opposed to other clone detection techniques.

In addition, the scope of the metrics can be altered. If Covet were changed from comparing the metrics of routines to the metrics of files then it would have probably identified the whole of Backdrop and oldBackdrop.java.

7.4. Hypothesis 5

Hypothesis 5. Case studies developed with the aid of an automatic code generation will produce more clone classes. **FALSE**

Clone classes are important to maintainers because they represent groups of clones that share a common source. Identifying these classes allows the maintainer to potentially replace all the clones within a class with some unifying construct. If a clone class is large it indicates that the logic being cloned is widely used and, as with larger individual clones, may be ideal candidates for extraction for reuse.

Hypothesis 5 attempts to link an increase in the production of clone classes with the use of automatically generated code. It was thought that since automatically generated code has a very limited capacity for variation in programming then this would lead to clusters of clones all sharing a common ancestor. The act of including generated code is in effect cloning. The only difference is it is the machine that is doing the copy and pasting.

However the results from the case studies were inconclusive. Although CCFinder found significantly larger clone classes for the Barcrawl planner, nearly 16 clones per class on average as opposed to approximately 7 in GraphTool, it found on average just fewer than 5 clones in the Tropicana. CloneDr found clone classes of exactly the same size in Tropicana and GraphTool an average of just over 1 clone. There seemed to be no overall pattern to the size of clone classes with Covet finding the largest

classes (2.65) in Tropicana and classes virtually identical in size for GraphTool and the Barcrawl Planner (1.7 and 1.67 averages respectively).

As with the other experiments described in this thesis CCFinder found more than any of the other tools. It identified more classes and these were larger than any of the other tools. Covet and CloneDr identified similar numbers of clone classes but their sizes were different with CloneDr finding on average larger clone classes than Covet.

7.5. Hypotheses 6

Hypothesis 6. Replication across programs is more prevalent than replication within programs. **TRUE**

Replication across programs pose different maintenance problems than replication within the same program. Clones spread widely throughout a system cause the potential ripple effect to increase. Kamiya [Kam02] presents a metric called *radius* for measuring exactly how far a clone has spread. Clones with a large radius cause maintainers greater problems because it is *"difficult to maintain their consistency correctly"* [Kam02]. This is the case because the clones spread further away from their original source the relationship between the two is lost. This increases the program complexity of the system because future maintainers will attempt to understand both pieces of code. When two cloned code routines are serial it is fairly obvious to the maintainer. The maintainer then only has to understand one of the functions (of course an understanding of modifications to the clones is required as well). This thesis did not look specifically at the radius of clones instead the focus was simply whether a clone was contained within the same program or belonged to more than one program. Systems with a high proportion of replication across programs will require greater effort to maintain than those with a higher proportion of cloning within programs.

Hypothesis 6 focused on the proportion of replication that exists across programs. It was thought that replication within a program would be less prevalent because a programmer would not design a single program to contain duplicated logic. However, if the software system was extended further on in its life cycle additional programs may be added. These additional programs may share similar functionality

with the existing programs but as a time saving device the programmer decides to clone existing code rather than re-engineering it. Also the prevalence of automatically generated code in Tropicana and the Barcrawl Planner means that cloning will happen across files as each GUI screen would usually be contained within separate files and these files would contain cloned code. Hence, if cloning exists it was thought that it would more likely exist across programs. From the results this appears to be the case for the three case studies. Case studies containing more clones had a higher proportion of replication across clones. Tropicana contained the most clones and was the only case study to contain a proportion of replication across clones greater than 50% for all the tools. Tropicana contained the second highest number of clones and the tools CloneDr and CCFinder both found a much higher percentage of replication across programs than for GraphTool.

7.6. Hypotheses 7 and 8

Hypothesis 7. Each clone detection tool will identify different sets of clones. **TRUE**

Hypothesis 8. No clone detection tool will find every clone within a case study. **TRUE**

Ideally each clone detection tool should find every actual clone within a case study. If this was the case then all the tools used in the case study would return the same clones and there would be no need to use more than one clone detection tool. Unfortunately for maintainers this is not the case and as the results show none of the tools consistently record a 100% precision for actual clone detection. It is also the case that the tools disagree on the clones identified within the same case study.

Hypothesis 7 states that different tools do not identify the same clones. Results from the case studies confirm this by demonstrating a low level of intersection between many to the tools' results. Out of all of the tools CCFinder found the highest percentage of the other tools' clones. Its large recall means that there is greater chance of finding clones identified by other tools. In the GraphTool case study CCFinder found on average 46% of the other tools clones. This figure increased in the Barcrawl planner case study to 64%. Finally in the Tropicana case study CCFinder found on average 67% of the clones identified by other tools. Of all the

tools CCFinder found the least amount of intersection with Covet. However this was not just the case for CCFinder as it appears that Covet has the lowest intersection with the clone sets of the other tools. One interesting relationship is that between MOSS and JPlag. JPlag found a greater percentage of clones identified by MOSS than it did for any other tool. For example, in the GraphTool case study JPlag identified 54% of MOSS clones whereas it only identified 15% of CloneDr's, 10% of Covet's and 13% of CCFinder's clones.

Intersection between the clones identified by each tool is influenced greatly by the size of each tool's clone set. This is demonstrated in the three case studies. The overall level of intersection is lowest in the Barcrawl Planner case study. Although this case study has the greatest number of potential clones, 72% of these were identified by CCFinder. As a result CCFinder was the only tool to find a significant percentage of the other tools clones. GraphTool and Tropicana case studies had a more even spread of clones identified by all the tools and this meant a greater level of intersection.

In the Barcrawl Planner MOSS and JPlag failed to find any clones in common with Covet and very few in common with CloneDr. This lack of agreement indicates that the different tools are looking for different attributes to identify clones. This is a reflection of the subjective nature of clone detection for clones that have been modified.

Hypothesis 8 states that no clone detection tools will find every actual clone within a case study. This was proven true as none of the tools had a 100% recall. In this case then it means that maintainers may wish to use more than one clone detection tool in order to capture the greatest number of clones possible. Of all the tools CCFinder had the highest recall throughout the case studies. In the case of GraphTool its recall was 59% (39% greater than the second highest recall which was JPlag). Its recall was even higher at 79% in Barcrawl Planner, as was the gap between itself and the tool with the 2nd highest (60% greater than MOSS whose recall was just 19%). In the Tropicana case study CCFinder's recall fell sharply to 48%. This figure was only 5% greater than Covet's who had the 2nd highest recall (43%). CloneDr was the tool with the lowest recall. Throughout the case studies CloneDr's recall did not reach 10%.

Intersection between the tools is important to a maintainer because if two tools consistently find the same clones then the maintainers can omit one and save themselves effort. JPlag identifies more potential clones than MOSS and also finds a significant proportion of MOSS's clones. This could be motivation for just using JPlag. An argument for using as many tools as possible can also be made as if several tools all identify the same clones then the maintainer can be more confident that these are actual clones and not false positives. In effect the maintainer could support manual verification with tool verification.

One possible method experimented with during this thesis was the use of a visualisation feature which highlighted where tools agree and disagree each other's clones.

7.6.1. Clone by clone visualisation

The visualisations in Appendix A provide an extremely useful insight into the similarity of the clones produced by each tool. For example table A15 gives a very clear indication that most of MOSS's clones were found by the other tools because of the amount of black cells. The size of the tables also shows the amount of cloning identified by each tool. Table A15 also shows that MOSS identified the second smallest number of clones for Tropicana. By looking down a specific column for each case study it is possible to assess how a specific clone detection tool intersected with the other tools. In each of the three case studies column D had the greatest number of black cells which told the maintainer that CCFinder had found more of the other tools' clones. Another interesting fact that has been highlighted by the visualisation is that the number of clones found by all five tools is only three. In the tables this fact is shown as an entire row of all black cells or a "black row". Two of these occurrences appear in table A12 and the other in A13 (CloneDr and Covet's clones for Tropicana).

7.7. Hypothesis 9

Hypothesis 9. No clone detection tool can achieve 100% precision for every case study. **TRUE**

Each of the clone detection tools in one sense obtains 100% precision. However, they are 100% precise because they identify every instance of what the developers of that tool considered to be a clone. Ideally maintainers would be able to select a software system, pick a clone detection tool, and that tool would find every clone as if the maintainer themselves had inspected the code. However, this is not the case. Firstly each maintainer may have a different idea of what they consider is and is not a clone. Some maintainers may consider automatically generated code whereas others may choose to ignore it. As there is no agreed definition of what a code clone is, measuring precision can be considered subjective.

Hypothesis 9 looks at precision for the purposes of software maintenance. It states that none of the tools can be relied on completely by a maintainer because there will always be false positives. When manually verifying clones, potential clones were identified as non-clones not only if they obviously had not been copied and pasted but also if there was no benefit from removing them. For example, if the clone contained only a few lines of executable code or the last few lines of one routine and the first few of the next. Of all the clone detection tools MOSS was the most reliable out of all the three case studies MOSS had an average precision of 93%. CloneDr was the next most reliable. In the GraphTool case study CloneDr managed 100% and in the Barcrawl Planner the small number (6) of potential clones identified meant that as one of the potential clones identified was not considered relevant to maintenance the precision dropped to 86%. This was also the case in the Tropicana case study, seven of the potential clones were not considered useful to maintenance even though they were similar. Overall each tool's precision value was quite good with no tool having a precision value less than 50%. CloneDr and MOSS's unique objectives require a high precision value. CloneDr offers the facility of automatic removal of clones so it is important that it does not remove code that shouldn't be removed. MOSS is used to identify plagiarism and if institutions are to be confident enough to use MOSS then a minimum number of false positives is very important.

7.8. Hypothesis 10

Hypothesis 10. The proportion of cloning identified by clone detection tools within a case study is very dependent of the minimum size threshold. **TRUE**

The use of a threshold to filter out potential clones that are too small to be considered is important as it reduces the amount of time required to manually verify non-clones. This mechanism of filtering allows the maintainer to impose one of his or her own criteria on what is and is not a clone. By setting a minimum size threshold of 20 lines for the experiments described previously only potential clones of a "useful" size were presented by the tools. However, this constant size threshold appeared to be unsuitable for all tools in all case studies. One example is the number of potential clones filtered out of Covet and CloneDr for the Barcrawl Planner case study. A size threshold setting of 18 for CloneDr and 19 for Covet would have produced much larger clone sets. Plotting the number potential clones identified by a clone detection tool against the minimum size threshold could be used by a maintainer to select the most appropriate threshold. This threshold can then be used for that tool to identify the potential clones that will then be manually verified.

Hypothesis 10 aims to establish whether changes in the minimum size threshold has a significant effect on the number of potential clones identified by a clone detection tool. Results from the size threshold sensitivity experiments show that Covet is the most sensitive tool to increases in the size threshold as it consistently filters out over 80% of the potential clones identified between the thresholds of 10 and 20. This is because of the discrete nature of its clone detection. For example, suppose the size threshold was set to 20, if two 18 line routines (which are physically next to each other in the program source) are copied and pasted to another part of the system as Covet compares each routine individually it will filter these two routines out because neither meets the set threshold. This is not the case with the other tools as they will be able to find both routines as a single region of code whose size will be greater than the size threshold. JPlag is the least sensitive to increases in changes as the percentage of clones filtered out is the lowest by some margin. Throughout the case studies JPlag filters out between 40-50% of the potential clones identified between the thresholds of 10 and 20.

8. Conclusion and Further Ideas

8.1. Conclusions

This thesis has presented some interesting results, the lack of agreement between clone detection tools is a reflection on the lack of agreement between the tool's developers as to what exactly to consider a clone and what to consider a non-clone.

8.2. Evaluation of criteria for success

Three main aims were set out in section 1.1 these will now be reviewed

8.2.1. Literature survey reviewing current issues relevant to software maintenance and in particular code cloning

The literature survey was crucial to the evolution of the thesis as it highlighted the variety in approaches to clone detection that currently exist. It also allowed the discovery of the tools that were used later in the thesis as part of the case studies. Covet was inspired by the work on metric based clone detection presented by Mayrand [May96b]. The amount of recent literature on the detection / removal of clones proves that there is need for such an activity and that the current state of the art is far from agreeing on a single solution to the problem.

8.2.2. Development of an efficient metric based clone detection tool

Covet was developed to a satisfactory level of efficiency and gained acceptable precision and recall values. Covet operates with a command line interface which is not ideal for use within industry. It is however still in the prototype stage and in two of the case studies produced very high precision. However, Covet achieved a very low recall for the GraphTool and Barcrawl Planner case studies.

8.2.3. Comparison of a range of clone detection tools, focusing on their precision, recall and intersection of results.

Comparing the clone detection tools was essential to establish if maintainer can rely on a single tool to identify reliable all instances of cloning. The disparity in the clones that each tool identified proves that none of current tools available will identify every clone in a system with 100% precision.

8.3. Cloning results are significantly different for each tool

The results of the experiments carried out for each case study showed that the clones identified by each clone detection tool differed greatly. No two tools consistently had high intersection ratings with another tool's results. Although CCFinder's found a high proportion of the other tools clones a very small proportion of its clones were found by the other tools. One surprising aspect was how dissimilar the results of JPlag and MOSS were. These are both academic plagiarism detection tools and neither searches for replication within programs yet the intersection between the tool results sets for each of the case studies was much lower than might be expected. It is impossible to attribute these differences to completely different detection strategies as, although the detection method employed by JPlag has been published, no details of MOSS's detection method have been published. Of the clone detection tools Covet appears to have identified the most dissimilar clones. This is not surprising as it is the only tool (with the possible exception of MOSS) that uses a whole routine as its primary point of comparison rather than tokens or lines. It also differs in comparing metrics rather than the actual program text or syntax. CloneDr's deep syntactical analysis of programs enables a high degree of precision but this also restricts the tool's recall.

Clones identified by each clone detection tool also appear, as would be expected, to be completely dependent on the case study. The only patterns that can be taken from comparisons of each case study is that CCFinder consistently found a greater number of clones than the other tools for every case study. The number of clones identified by each tool varied for each case study and apart from CCFinder there was no consistent ranking of the tools performance across case studies. For example, in GraphTool Covet identified more clones than MOSS whereas the opposite was true in the Barcawl Planner case study.

8.4. Minimum size thresholds should be adapted for each clone detection tool and case study

Results from the experiments show clearly that the minimum size threshold has a significant impact on the proportion of clones. It is not the case that one minimum size threshold fits all tools or all case studies. For example in the Barcrawl Planner case study (figure 6.8) when the minimum size threshold is at 17, CloneDr found 444 clones but an increasing of just one to 18 reduces the number of clones identified to 38. Similarly for Covet with a minimum size threshold of 18, found 473 clones whilst with a threshold of 17 the number of clones identified to reduced to seven. In order to take this sensitivity into account a solution might be to run a preliminary clone detection tool at various stages and record the number of clones identified from each tool for the case study in question. From these results it would be possible to select which minimum size threshold to use for the main clone detection task based on the expectations of the maintenance process.

During each of the case studies a constant minimum size threshold of 20 was used. This setting as the specific example of Covet and CloneDr (described above) highlights was not ideal for all the clone detection tools as it filtered out a large proportion of some of the clones and so affected the intersection between clone sets. The minimum size threshold in terms of lines of code can be misleading. For example some integrated development environments can impose double spacing or other forms of padding.

8.5. No single clone detection tool consistently identifies every clone within a case study. No clone detection tool produces 100% precision.

CCFinder identified the most clones for each case study. It did not, however, find every clone within the case studies. In order to achieve the greatest recall a combination of tools should be used. This would give the maintainer not only a much greater volume of clones but from comparisons of results can lead to a form of automatic, instead of manual, verification. For example, if a maintainer uses a set of five clone detection tools and all five identify the same clones then there is a high probability that this will be an actual clone. This is one aspect of clone detection where visualisation would be extremely useful. In this case the effect would be to

reduce the amount of manual checking required by a maintainer. An effective combination might be the use of CCFinder to gain an overview of the level of cloning within a system. If CCFinder found a high degree of cloning within a system then a second scan of these results using CloneDr would reduce the amount of manual verification required. The clone-by-clone visualisation technique (Appendix A) could be used to spot which clones are found by the majority, if not all, of the clone detection tools. An enhancement to this would be the ability to focus in on the actual source code identified by each row in the table.

If due to cost or efficiency reasons a maintainer wishes to use only one tool then CCFinder identifies the greatest recall of clones. However, for precision and efficiency CloneDr would be a better choice. CloneDr has a much lower recall but greater precision. This needs to be the case due to the potential for automatic re-engineering of clones provided by this tool. If CloneDr's precision was not as high then potentially costly mistakes could be made by altering code that did not require alteration.

8.6. Integrated Development Environments increase the proportion of clones within a case study

During the manual verification of clones identified for the three case studies it was observed that a significant proportion of the clones identified for Tropicana and Barcrawl Planner were the result of automatically generated code produced by JBuilder. This is also reflected in the actual clones identified for each case study (see Table 6.1) with the two case studies developed using JBuilder resulting in a much greater number of verified clones than GraphTool. This trend would seem extremely logical because of the method JBuilder uses to build applications. Users design the look and feel of their applications from a GUI with JBuilder copying and pasting from its library of code to the source of the user's applications. The only changes made to JBuilder's original generated code are those in the variables such as width or height attributes of a frame. The same process was found in sections of GraphTool involving the GUI. Because of the desire to display a consistent user interface copy and pasting would seem the perfect solution. Also programming GUIs can be a lengthy but not overly complicated process. For example, setting up a second menu bar will be done in almost exactly the same way as the first with the only

modifications being the labels used and the method names to which the labels refer. Of course from a maintainers point of view it would be better to have a single parameterised menu bar set-up method.

8.7. Further Ideas

Clone detection is an evolving research topic. This section will look at some of the options available in extending the work carried out in this thesis.

8.8. Clone visualisation

One issue that has only been touched upon within this thesis is that of visualisation of clones. Several of the clone detection tools used in the case studies provide their own visualisation techniques. None of these tools, however, provide a facility to import the results of other clone detection tools into this visualisation. Appendix A and Chapter 4 (see figure 4.4) describe a novel approach to viewing clones produced by one tool and allowing comparisons to the clones identified by other tools. This could be extended to allow the maintainer to select a particular clone of interest and view the relevant source code. Another visualisation technique might be a Venn diagram of each tool's clone set which could show clearly how much intersection there are between each of the tools.

8.9. Inclusion of a Radius Metric

Kamiya [Kam02] recently published a set of clone specific metrics (see section 3.7). Amongst them was the clone radius which records how wide spread clones are through out a case study. This would have been a very useful statistic to have included in the case studies and would have provided the facility to quantify how far replication across programs had spread for specific clone classes. Unfortunately Kamiya's paper [Kam02] is still only a draft and was only released towards the end of this thesis.

With this metric it would also be possible to create a visualisation of the cloning throughout a system. This could be added to by colour coding clone classes and displaying the clones on a graph of the system.

8.10. Inclusion of a language independent clone detection tool

The intention was to include both DUP [Bak95] and DUPLOC [Duc99a] with the other clone detection tools used in the case studies. Unfortunately due to licensing issues DUP was unavailable and technical difficulties DUPLOC was not used. The inclusion of a language independent detection tools would provide useful data to see how similar / dissimilar the clones between that and CloneDr's syntactic analysis.

8.11. Replication across systems

One aspect of cloning that has not been considered is whether cloning is prevalent across different software systems. This could be used to justify the creation of a reuse program and if one already exists then it indicates it is not working properly.

9. References

[Bak92] Baker B. S., *A Program for Identifying Duplicated Code*, Proceedings of the 24th Symposium on the Interface: Computer Science and Statistics, pp. 49-57 1992

[Bak93] Baker B. S., *Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance*, SIAM Journal on Computing 1993

[Bak95] Baker B. S., *On Finding Duplication and Near-Duplication in Large Software Systems*, 2nd Working Conference on Reverse Engineering, pp. 86-95 1995

[Bal99] Balazinska M. et. al. *Partial Redesign of Java software Systems Based on Clone Analysis*, 6th Working Conference on Reverse Engineering, p. 326 1999

[Bal00] Balazinska M. et al. *Advanced Clone Analysis to Support Object-Oriented System Refactoring*, 7thWorking Conference on Reverse Engineering, pp.98-107 2000

[Bax98] Baxter I.D., Yahin A., Moura L., Sant'Anna M., Bier L., *Clone Detection Using Abstract Syntax Trees*, International Conference on Software Maintenance, pp. 368-377 1998

[Bur97] Burd E.L., Munro M., *Investigating the Maintenance Implications of the Replication of Code*, International Conference on Software Maintenance, p. 322 1997

[Cha95] Chalmers M., *Design perspectives in visualising complex information*, International Federation for Information Processing, pp. 103-111 1995

[Cha97] Chan P., Munro M., *PUI : A Tool to Support Program Understanding*, International Conference on Software Maintenance, pp. 192 - 198 1997

[Chi90] Chikofsky E. J., Cross II J.H., *Reverse engineering and design recovery: A taxonomy* IEEE Software, pp. 13-17 1990

[Chu93] Church Ward K., Helfman J., *Dotplot: a Program for Exploring Self-Similarity in Millions of Lines of Text and Code*, American Statistical Association, pp. 153-174 1993

[Cla98] Clayton R., Rugaber S., Wills L., *On the Knowledge Required to Understand a Program*, Working Conference on Reverse Engineering, pp. 69 - 78 1998

[Com00] Comella-Dorda S. et al, *A Survey of Legacy System Modernization Approaches*, Software Engineering Institute, Technical Note CMU/SEI-2000-TN-003.,

[Dem99] Demeyer S., Ducasse S., Lanza M., *A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualisation*, Working Conference on Reverse Engineering, pp. 175-186 1999

[Dom95] Domingue J.B., *Using Software Visualization Technology In The Validation Of Knowledge Based Systems*, Knowledge Acquisition Workshop 1995

[Duc99a] Ducasse S., Rieger M., Demeyer S., *A Language Independent Approach for Detecting Duplicated Code*, International Conference on Software Maintenance, pp. 109-118 1999

[Duc99b] Ducasse S., Rieger M., Golomingi G., *Tool Support for Refactoring Duplicated OO Code*, Workshop on Experiences in Object-Oriented Re-Engineering, pp. 177-178 1999

[Duc01] Ducasse S., Lanza M., Tichelaar S., *The Moose Reengineering Environment*, Smalltalk Chronicles 2001

[Eic96] Ball T.J., Eick S.G., *Software Visualization in the Large*, IEEE Computer, pp. 33-43 1996

[Fra96] Frakes W., Terry C., *Software Process Reuse In An Industrial Setting*, Article in ACM Computing Surveys, pp. 22-30 1996

[Hel95] Helfman J., *Dotplot Patterns: A Literal Look at Pattern Languages*, Theory and Practice of Object Systems, pp. 31-41 1995

[Joh94] Johnson J. H., *Substring Matching For Clone Detection and Change Tracking*, International Conference on Software Maintenance, pp. 120-126 1994.

[Joh94b] Johnson J. H., *Visualizing Textual Redundancy in Legacy Source*, CASCON, pp. 120-126 1994

[Joh95] Jonassen D. H., *Operationalizing Mental Models: Strategies for Assessing Mental Models to Support Meaningful Learning and Design- Supportive Learning Environments*, Computer-Supported Collaborative Learning, pp. 182-186 1995

[Jor95] Jorgenson M., *An empirical study of software maintenance tasks*, Software Maintenance: Research and Practice, pp. 27-48 1995

[Kam01] Kamiya T., Ohata F., Kondou K., Kusumoto S., Inoue K., Maintenance Support Tools for Java Programs: CCFinder and JAAT, International Conference on Software Engineering, pp. 837-838 2001

[Kam02] Kamiya T., Kusumoto S., and Inoue K., CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code, IEEE Trans. Software Engineering, pp. 368-377 2002

[Kaz97] Kazman R., *Playing Detective: Reconstructing Software Architecture from Available Evidence*, Software Engineering Institute Technical Report CMU/SEI-97-TR-010 1997

[Klo96] Klosch R., *Reverse Engineering: Why and How to Reverse Engineer Software*, Proceeding of the Software Symposium, pp. 32-40 1996

[Kni98] Knight C., Munro M., *Visualisation for Program Comprehension: Information and Issues*, University of Durham, Computer Science Technical Report 12/98 1998

[Kni99] Knight C., Munro M., *Comprehension with[in] Virtual Environments Visualisations*, International Workshop on Program Comprehension, pp. 411 1999

[Kni01] Knight C., Munro M., *Visual Information; Amplifying and Foraging*, Proceedings of SPIE 2001

[Kni01a] Knight C., Munro M., *Software Visualisation Conundrums*, University of Durham, Department of Computer Science Technical Report 2001

[Kom01] Komondoor R., Horwitz S., *Using Slicing to Identify Duplication in Source Code*, Symposium on Static Analysis, pp. 40-56 2001

[Kri01] Krinke J., *Identifying Similar Code with Program Dependence Graphs*, Working Conference on Reverse Engineering, 2001

[Lag97] Lague B., Proulx D., Mayrand J., Merlo E., Hudepohl J., *Assessing the Benefits of Incorporating Function Clone Detection in a Development Process*, International Conference on Software Maintenance, pp. 314-321 1997

[Lan01] Lanza M., *The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques*, International Workshop on Principles of Software Evolution, pp. 37-42 2001

[Lia98] Liang D., Harrold M. J., *Slicing Objects Using System Dependence Graphs*, International Conference on Software Maintenance, pp. 358-367 1998

[Mal01] Maletic J. I., Leigh J., Marcus A., *Visualizing Software in an Immersive Virtual Reality Environment*, (Research Group Knowledge and Language Processing) WSV 2001

[May96a] Mayrand J., Leblanc C., Merlo E., *Evaluating the Benefits of Clone Detection in the Software Maintenance Activities in Large Scale Systems*, Workshop on Empirical Studies of Software Maintenance, 1996

[May96b] Mayrand J., Leblanc C., Merlo E., *Automatic Detection of Function Clones in a Software System Using Metrics*, International Conference on Software Maintenance, p 244-254 1996

[May97] Mayrhauser A., Marie Vans A., Howe A., *Program Understanding Behaviour during Enhancement of Large-scale Software*, Software Maintenance: Research and Practice Vol 9 pp.299–327 1997

[McC76] McCabe T. J., *A Complexity Measure*, IEEE Transactions on Software Engineering, pp. 308-320 1976

[Mun99] Munzner T., *H3: Laying Out Large Directed Graphs in 3D Hyperbolic Space*, *IEEE Symposium on Information Visualization*, pp. 2-10 1997

[Nie97] Niessink F., van Vliet H., *Predicting Maintenance Effort with Function Points*, International Conference of Software Maintenance, pp. 32-39 1997

[Pre00] Prechelt L., Malpohl G., Philippsen M., JPlag: *Finding plagiarisms among a set of programs*, Technical Report 2000-1

[Rom92] Roman G-C., Cox K.C., *Program Visualization: The Art of Mapping Programs to Pictures*, International Conference on Software Engineering, pp. 412-420 1992

[Rug92] Rugaber S., *Program Comprehension for Reverse Engineering*, AAAI Workshop on AI and Automated Program Understanding 1992

[Rug95] Rugaber S., *Program Comprehension*, Encyclopaedia of Computer Science and Technology 1995

[Sta93] Stasko J. T., Patterson C., *Understanding and Characterizing Program Visualization Systems*, Graphics, Visualization, and Usability Centre Technical Report GIT-GVU-91-17 1993

[Stor97] Storey M.-A.D., *Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration*, International Workshop on Program Comprehension, pp. 171-185 1997

[Til96] Tilley S. R., Smith D. B., *Coming Attractions in Program Understanding*, Software Engineering Institute Technical Report CMU/SEI-96-TR-019 1996

[Von98] von Mayrhauser A., *Program Comprehension and Enhancement of Software*, Information Technology and Knowledge Engineering, 1998

[Wei97] Weiderman N.H. et al, *Approaches to Legacy System Evolution*, Software Engineering Institute Technical Report CMU/SEI-97-TR-014 1997

[You98] Young P., Munro M., *Visualising Software in Virtually Reality*, International Workshop on Program Comprehension, 1998

WWW pages

[Aik02] Aiken, Alex, *A System for Detecting Software Plagiarism (Moss Homepage)*, Last visited 11th April 2002

[Amazon] http://www.amazon.co.uk, Amazon Website, Visited 2nd November 2001

[Bär99] Bär et al, http://www.iam.unibe.ch/~lanza/Publications/PDF/handbook.pdf *The FAMOOS Object-Oriented Reengineering Handbook*, Visited 1st July 2002

[CDIF97] http://www.eigroup.org/cdif/intro.html , Introduction to CDIF Visited 30th August 2002

[CSurferWP] http://www.grammatech.com/research/slicing/slicingWhitePaper.pdf, CodeSurfer Technology Overview – Dependence Graphs and Program Slicing. Visited 26th October 2001 15:00

[DatrixManual] http://www.iro.umontreal.ca/labs/gelo/datrix/refmanuals/metricdoc-4.1.pdf, Datrix Metric Reference Manual Version 4.1, Visited 18th January 2002

[DucLecture] http://iamwww.unibe.ch/~scg/Archive/Lectures/OOSR-W99.pdf, Lecture Notes from Object-Oriented Software Reengineering, Visited 26th December 2001 17:00

[Lag97] http://www.iro.umontreal.ca/labs/gelo/datrix/RD/qamc97.pdf, Assessing Risks related to Software Source Code using Datrix, Visited 30th August 2002 10:40

[MossHome] http://www.cs.berkeley.edu/~aiken/moss.html, Homepage for the Moss appliciation, Visited 26th October 2001 11:00

[CSM] http://www.dur.ac.uk/~dcs0www/research/csm/rip/introduction.html, Definition of software maintenance from the Centre for Software Maintenance, Visited 7th November 2001 12:00

[MMLecture] http://www.dur.ac.uk/malcolm.munro/local/lectures/sm-Introduction.ppt, Introduction to software maintenance lecture slides, Visited 7th November 2001 17:10

[IntroEval] Introduction to Evaluation website, http://trochim.human.cornell.edu/kb/intreval.htm, Visited 14.11.2001

[ISO] http://www.iso.ch/ International Standards Organisation, Visited 28th June 2002 14:00

[IWPC-talk] http://www.ai.univ-paris8.fr/UPU/IWPC00-slides.ps, Talk on Program Comprehension, Visited 8th November 2001 09:20

[JavaLayer] http://www.javazoom.net/javalayer/javalayer.html, Main page for the JavaLayer project Visited 23rd July 2002

[StoreyLecture] Storey P., Wong K., Müller H., *On Evaluating Program Understanding Tools*, Presentation from http://www.cs.ualberta.ca/~kenw/talks/dagstuhl98-talk.pdf , Visited 14th November 2001

[XMI] http://www.omg.org/technology/documents/formal/xmi.htm OMG XML Metadata Interchange page. Visited 30th August 2002

Books

[Bac94] Bache R., Bazzana G., Software Metrics for Product Assessment, McGrall-Hill Book Company 1994

[Ber94] Bergman S., Compiler Design : Theory, Tools and Examples, Wm.C. Brown 1994

[Fen96] Fenton N. E., Pfleeger S. L., Software Metrics A Rigorous & Practical Approach, Thomson Computer Press 1996

[Fow00] Fowler M., UML Distilled 2nd Edition: A Brief Guide to the Standard Object Modelling Language, Addison-Wesley 2000

[Jef99] Jeffrey C. L., Program Monitoring and Visualization: An Exploratory Approach, Springer 1999

[Mol93] Möller K. H., Paulish D. J., Software Metrics: A Practitioner's Guide to Improved Product Development, Chapman and Hall 1993

[Tak96] Takang A., Grubb P., Software Maintenance: Concepts and Practice, Thomson Computer Press 1996, ISBN 1-85032-192-2

Appendix A.

Tool	Label
CloneDr	A
Covet	B
JPlag	C
CCFinder	D
MOSS	E

Table A1 Tool Key

GraphTool Results

CloneDr

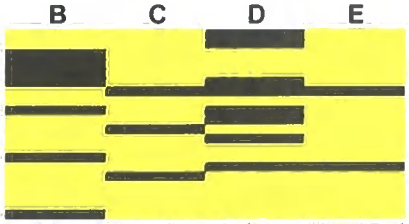


Table A2 GraphTool clones identified by CloneDr

Covet

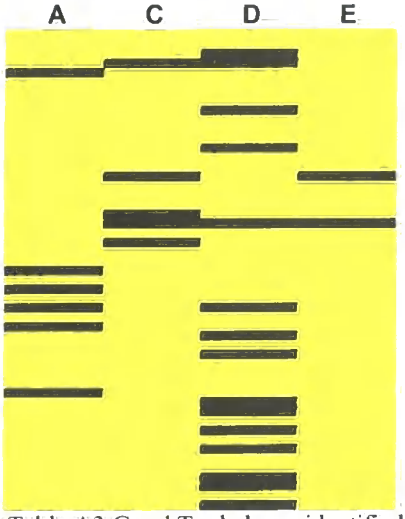
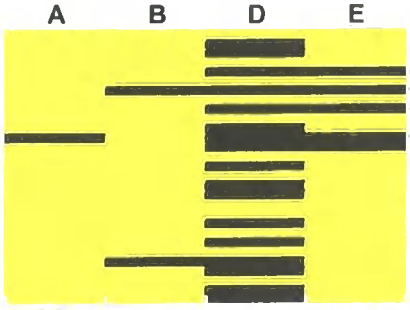


Table A3 GraphTool clones identified by Covet

JPlag



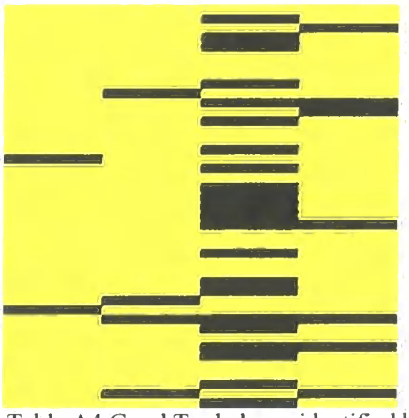


Table A4 GraphTool clones identified by JPlag

MOSS

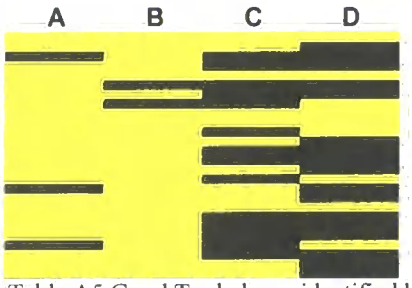
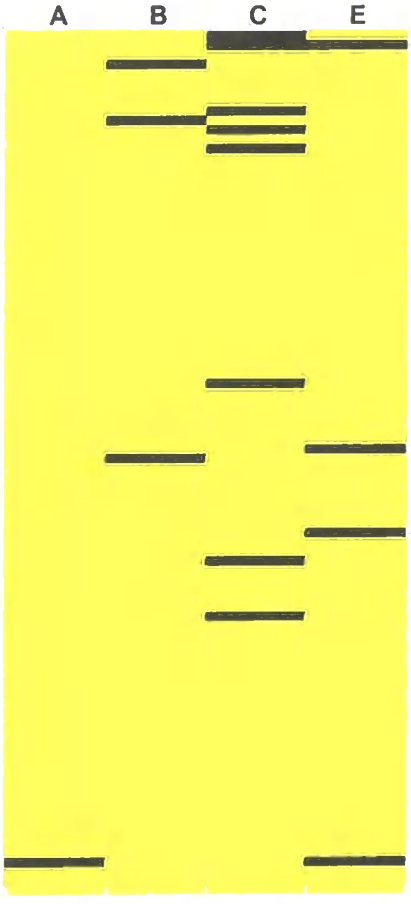


Table A5 GraphTool clones identified by MOSS

CCFinder



Appendix A.

Tool	Label
CloneDr	A
Covet	B
JPlag	C
CCFinder	D
MOSS	E

Table A1 Tool Key

GraphTool Results

CloneDr

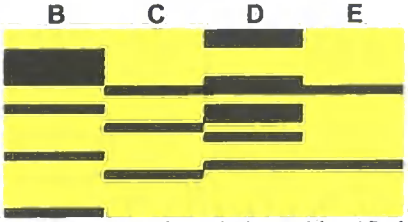


Table A2 GraphTool clones identified by CloneDr

Covet

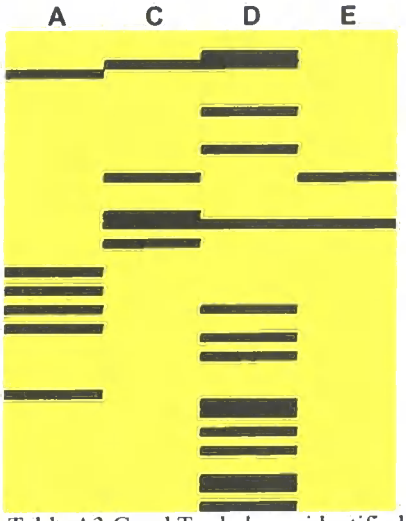
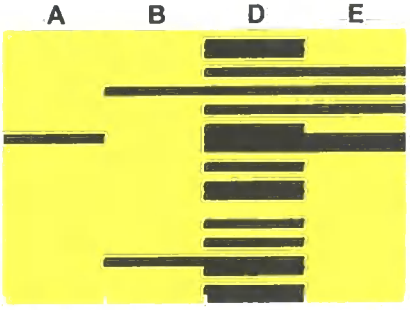


Table A3 GraphTool clones identified by Covet

JPlag



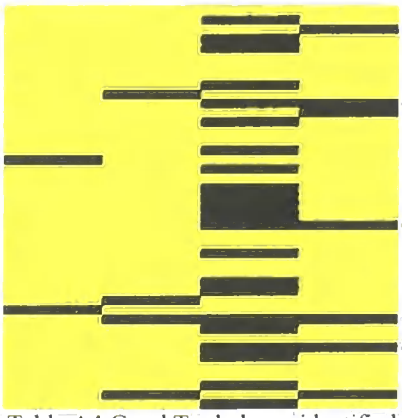


Table A4 GraphTool clones identified by JPlag

MOSS

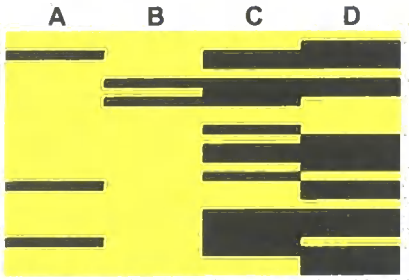
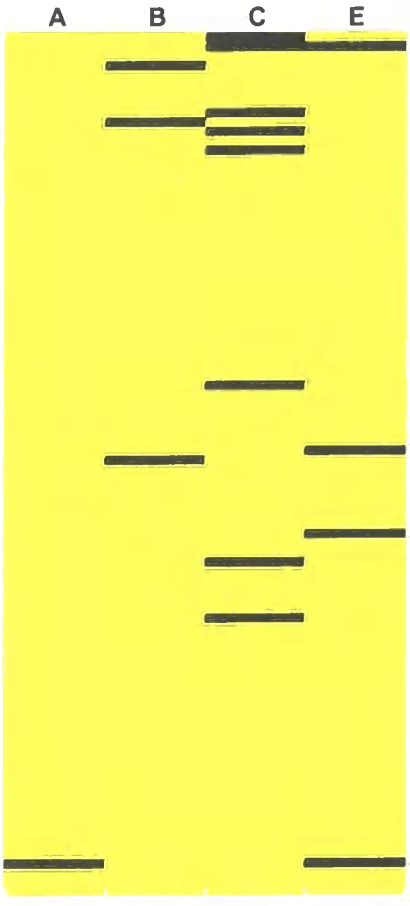


Table A5 GraphTool clones identified by MOSS

CCFinder



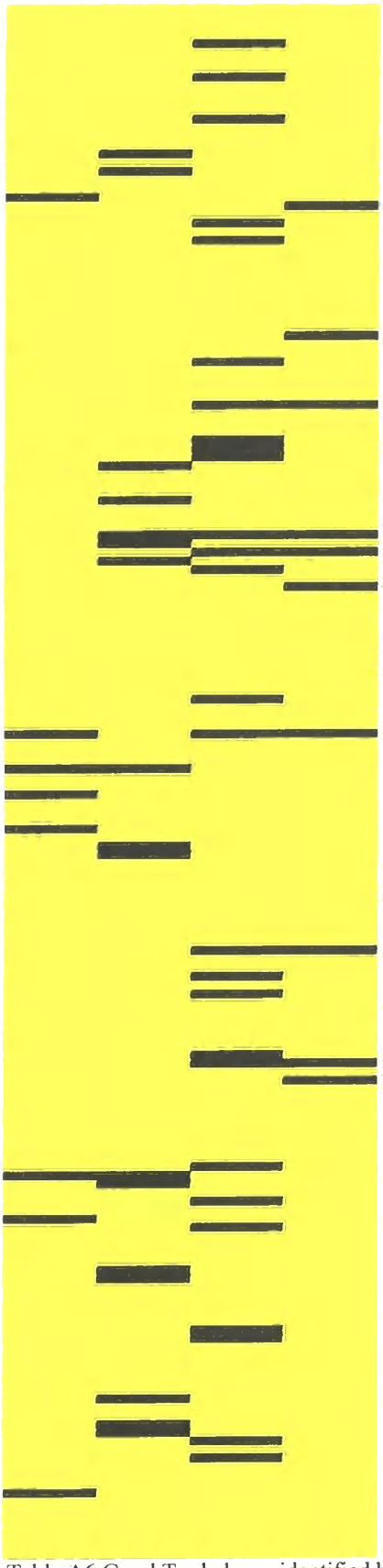


Table A6 GraphTool clones identified by CCFinder

Barcrawl Planner Results

CloneDr

Table A7 Barcrawl Planner clones identified by CloneDr

Covet

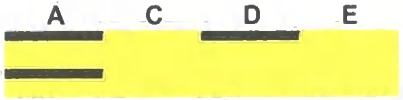


Table A8 Barcrawl Planner clones identified by Covet

JPlag

Table A9 Barcrawl Planner clones identified by JPlag

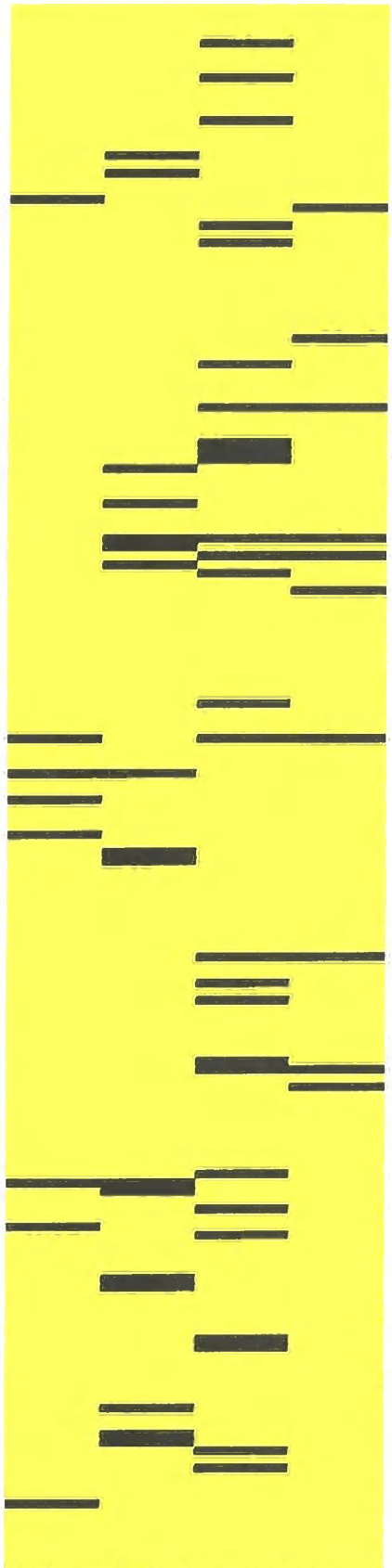


Table A6 GraphTool clones identified by CCFinder

Barcrawl Planner Results

CloneDr

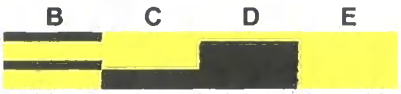


Table A7 Barcrawl Planner clones identified by CloneDr

Covet

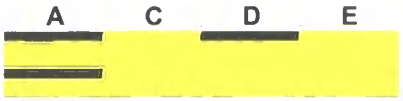


Table A8 Barcrawl Planner clones identified by Covet

JPlag

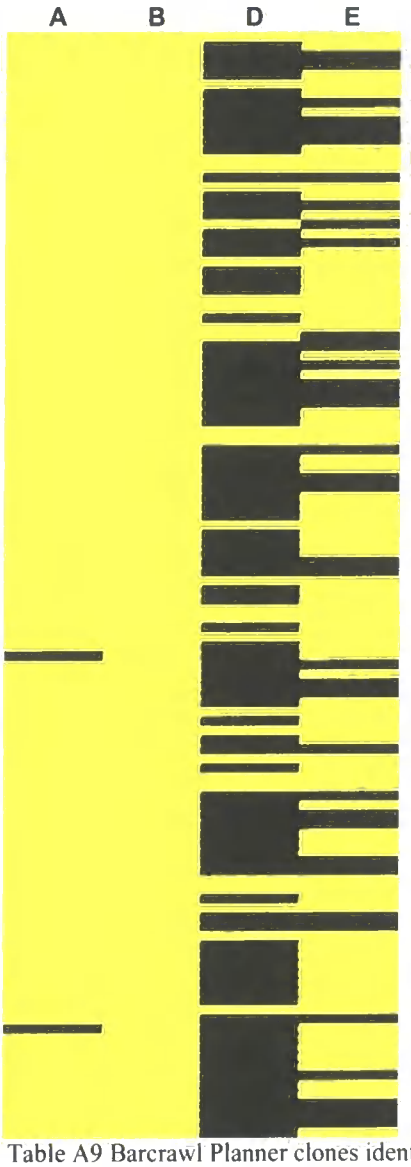
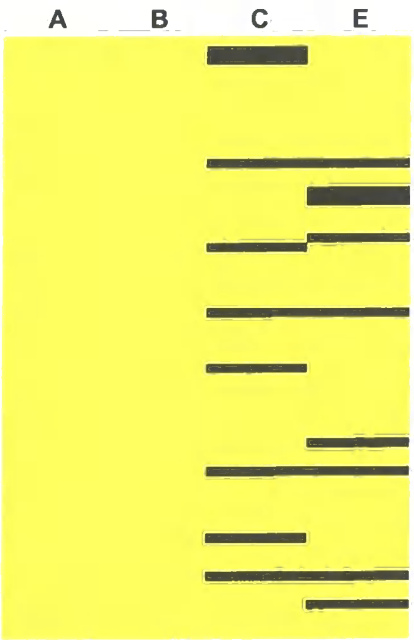


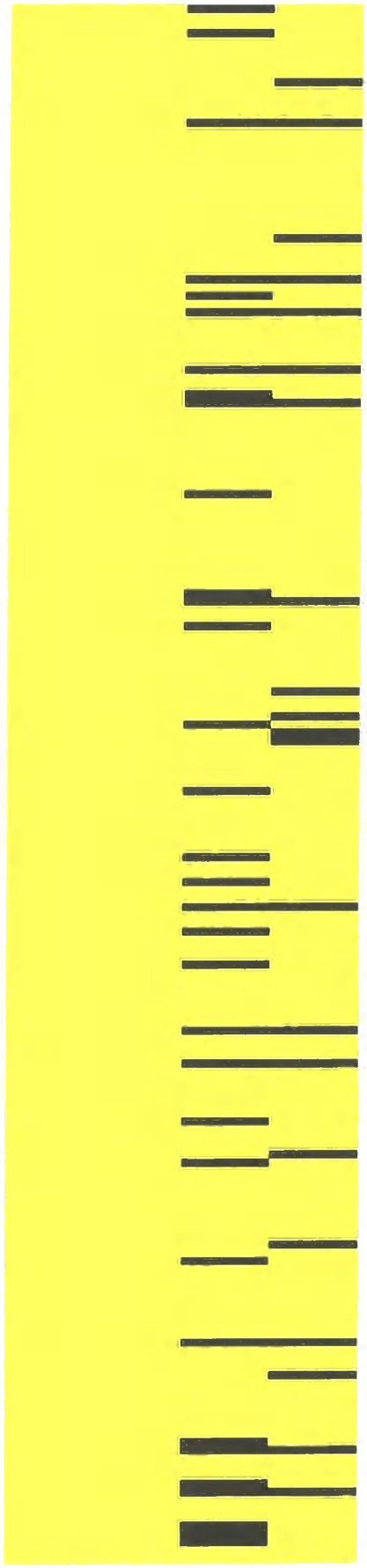
Table A9 Barcrawl Planner clones identified by JPlag

MOSS

Table A10 Barcrawl Planner clones identified by MOSS

CCFinder





MOSS

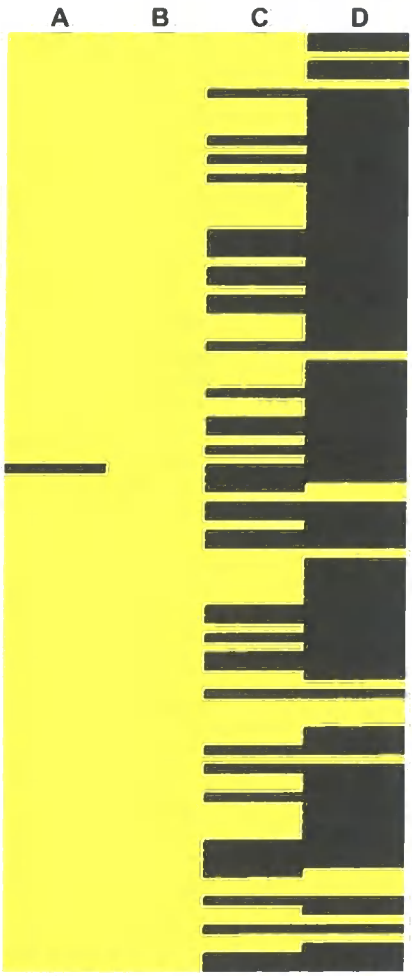
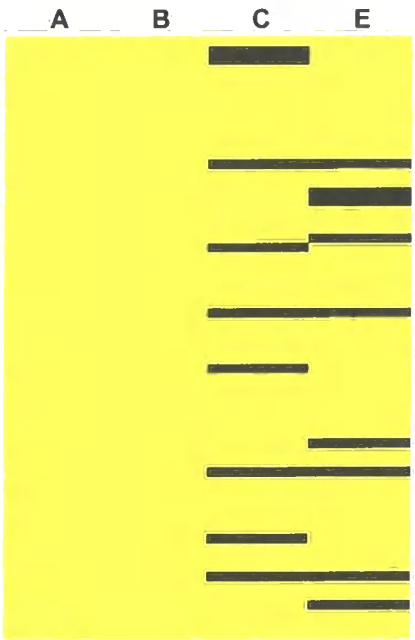
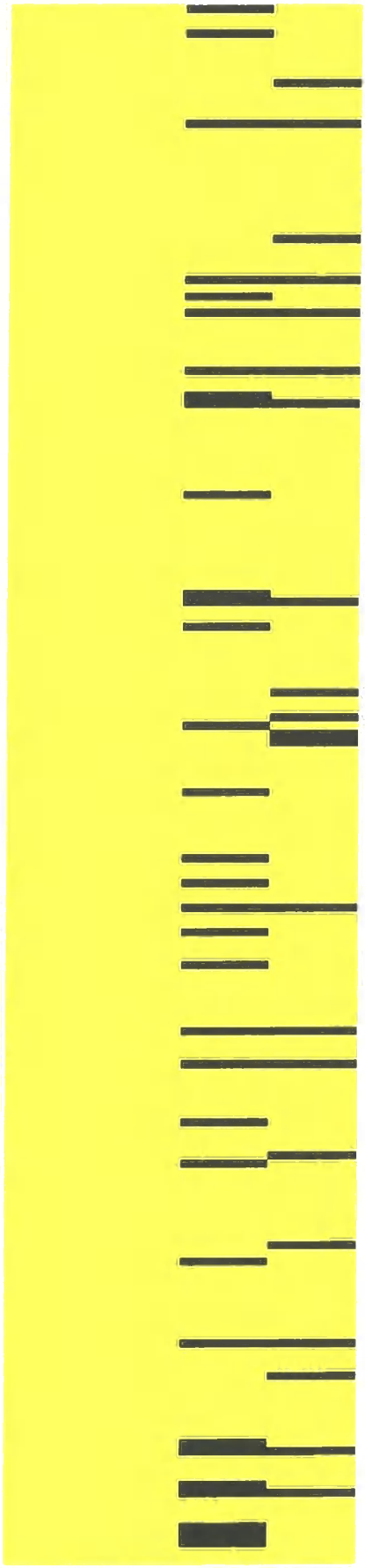
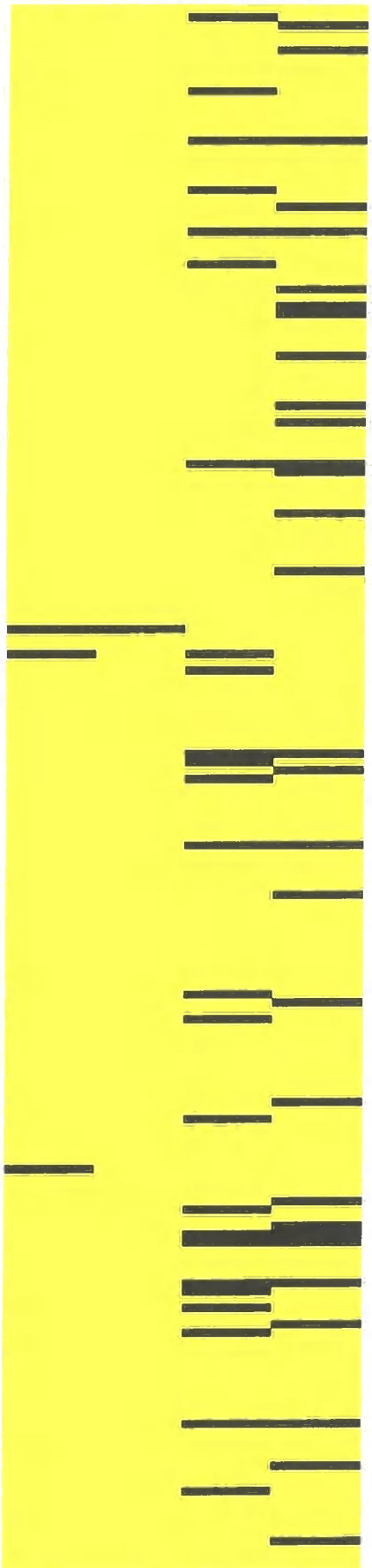


Table A10 Barcrawl Planner clones identified by MOSS

CCFinder







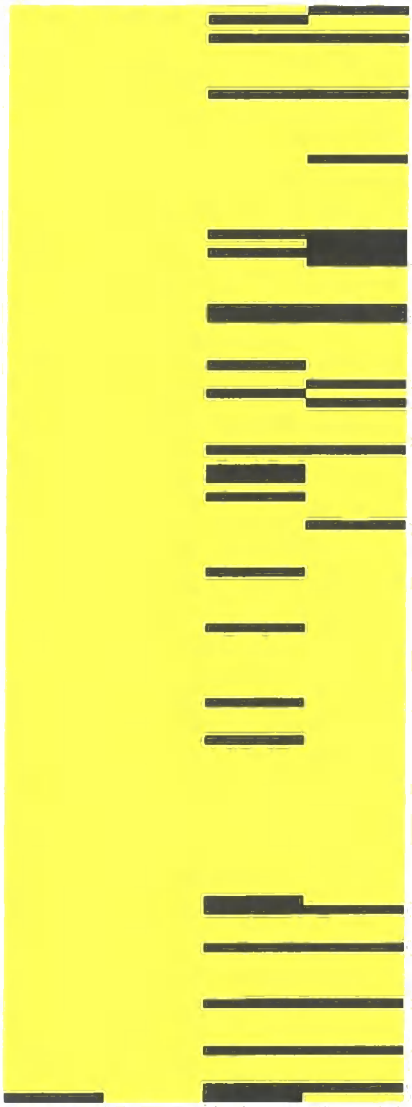


Table A11 Barcrawl Planner clones identified by CCFinder

Club Tropicana Results

CloneDr

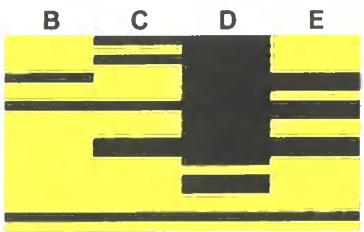
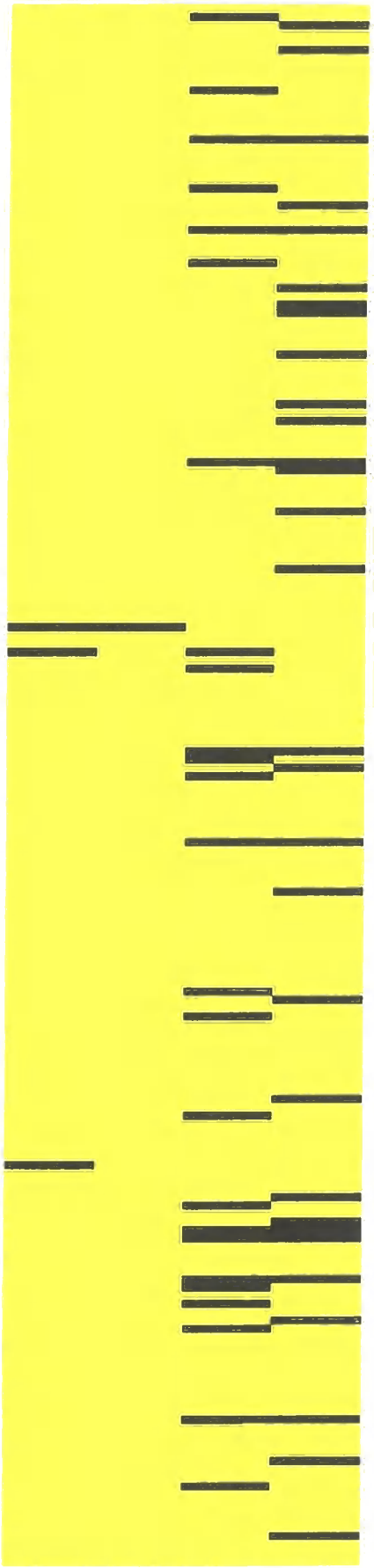


Table A12 Tropicana clones identified by CloneDr



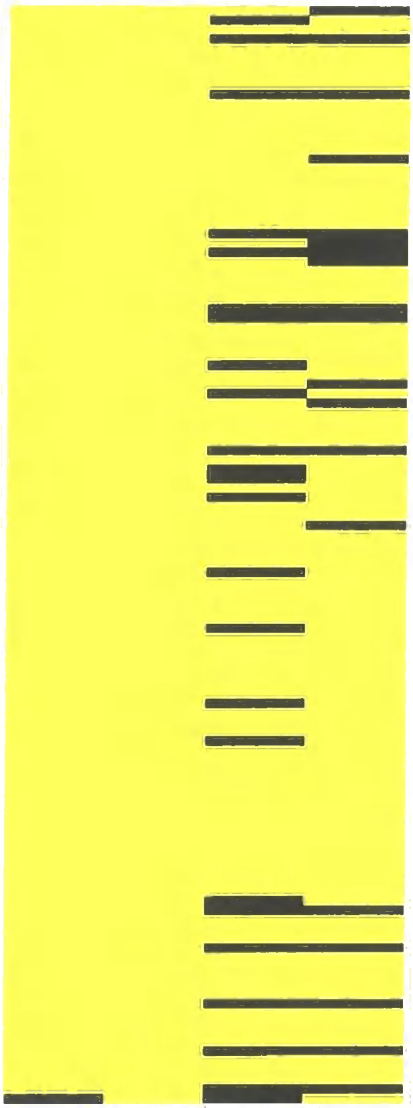


Table A11 Barcrawl Planner clones identified by CCFinder

Club Tropicana Results

CloneDr

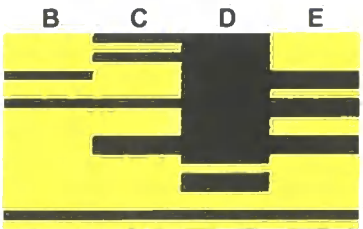


Table A12 Tropicana clones identified by CloneDr

Covet

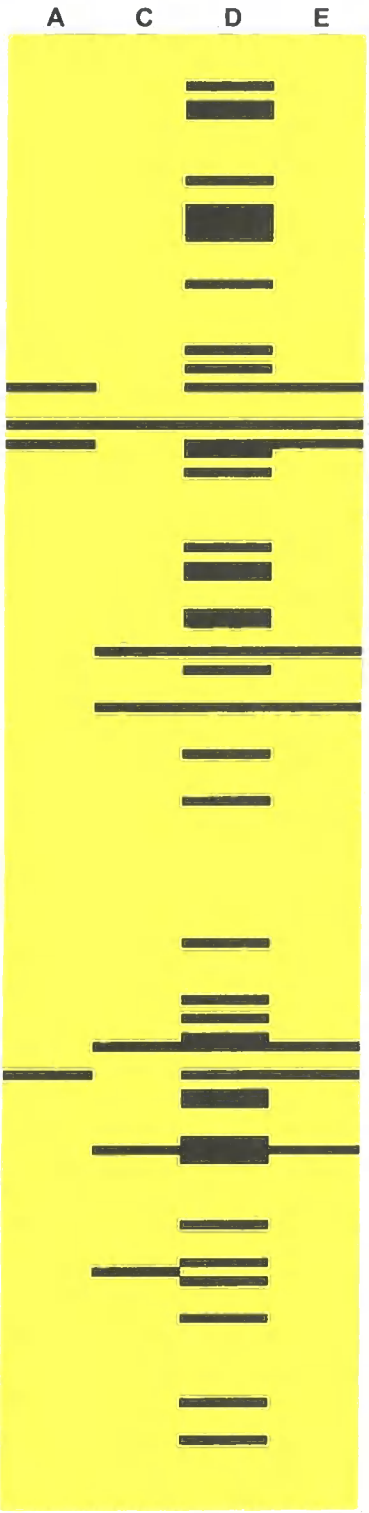


Table A13 Tropicana clones identified by Covet

JPlag

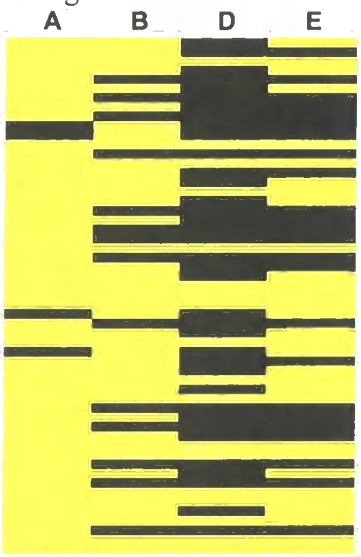


Table A14 Tropicana clones identified by JPlag

MOSS

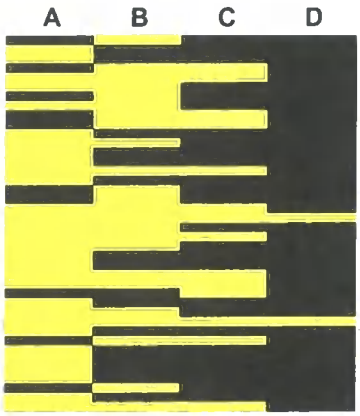
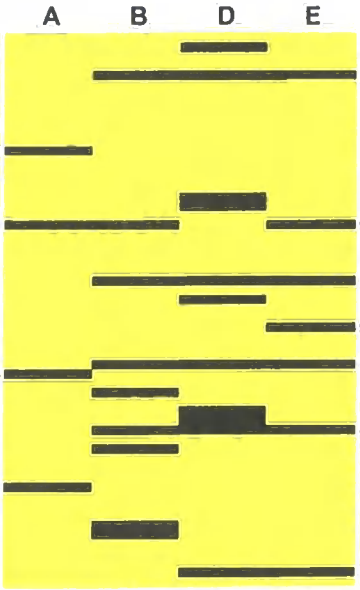


Table A15 Tropicana clones identified by MOSS

CCFinder



Covet

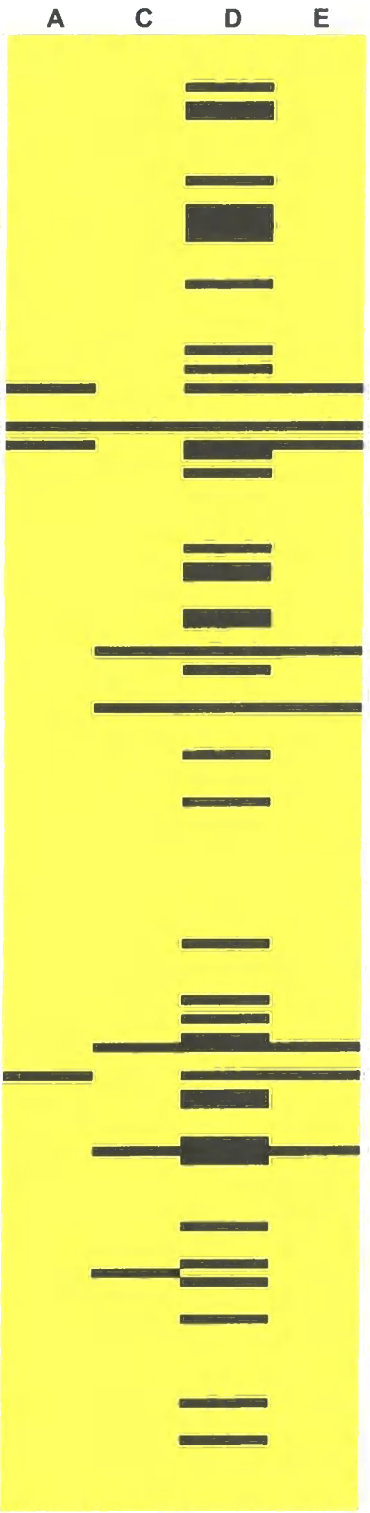


Table A13 Tropicana clones identified by Covet

JPlag

Table A14 Tropicana clones identified by JPlag

MOSS

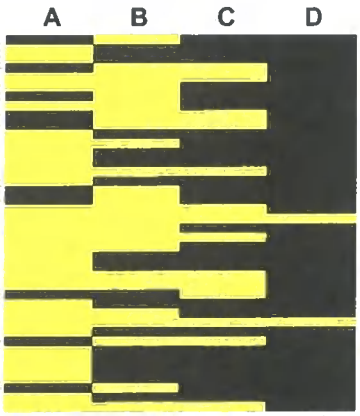
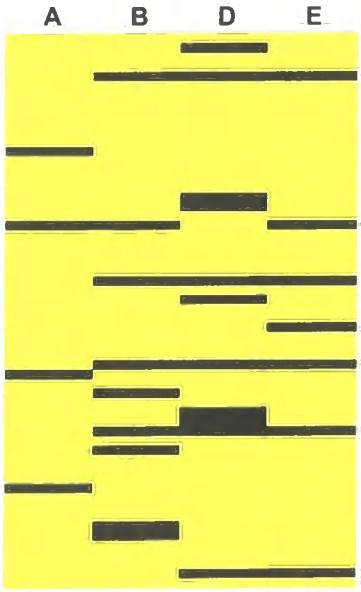


Table A15 Tropicana clones identified by MOSS

CCFinder



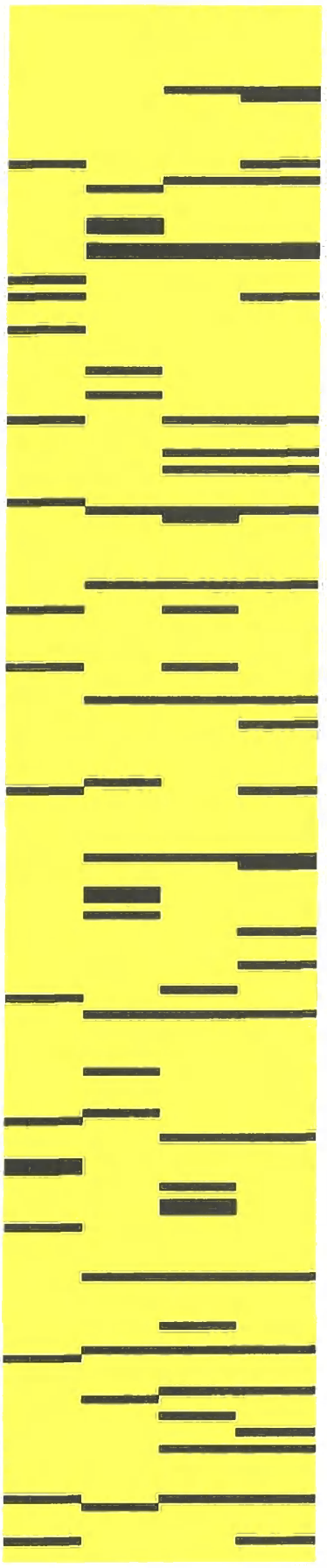


Table A16 Tropicana clones identified by CCFinder

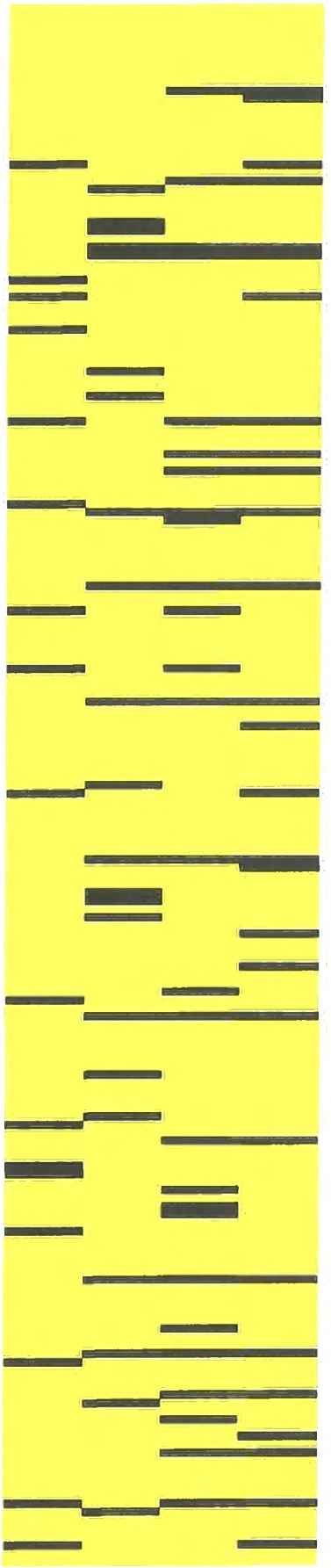


Table A16 Tropicana clones identified by CCFinder

Appendix B

Difference results for experimental programs

This appendix presents difference summaries for the experimental programs used in the development of the clone metrics and their thresholds. These summaries were produced by the text editor Textpad©.

HelloWorldApp and Clones

```
Compare: (<)C:\covet\clones\HelloWorldApp.java (287 bytes)
   with: (>)C:\covet\clones\HelloWorldAppAddedFor.java (393 bytes)

1,5c1,6
< /**
<  * The HelloWorldApp class implements an application that
<  * simply displays "Hello World!" to the standard output.
<  */
< public class HelloWorldApp
---
> /**
>  * The HelloWorldAppAddedFor class implements an application that
>  * simply displays "Hello World!" to the standard output.
>  * a meaningless for loop has been added
>  */
> public class HelloWorldAppAddedFor
9c10,13
<         System.out.println("Hello World!"); //Display the string.
---
>         for (int i=0;i<1;i++)
>         {
>           System.out.println("Hello World!"); //Display the string.
>           }


Compare: (<)C:\covet\clones\HelloWorldApp.java (287 bytes)
   with: (>)C:\covet\clones\HelloWorldAppErrors.java (352 bytes)

1,5c1,6
< /**
<  * The HelloWorldApp class implements an application that
<  * simply displays "Hello World!" to the standard output.
<  */
< public class HelloWorldApp
---
> /**
>  * The HelloWorldAppErrors class implements an application that
>  * simply displays "Hello World!" to the standard output.
>  * Simple Syntax errors
>  */
> public class HelloWorldAppErrors
10a11
>         System.out.print()


Compare: (<)C:\covet\clones\HelloWorldApp.java (287 bytes)
   with: (>)C:\covet\clones\HelloWorldAppOneXtraVar.java (407 bytes)
```

```
1,9c1,11
< /**
<  * The HelloWorldApp class implements an application that
<  * simply displays "Hello World!" to the standard output.
<  */
< public class HelloWorldApp
< {
<     public static void main(String[] args)
<     {
<         System.out.println("Hello World!"); //Display the string.
---
> /**
>  * The HelloWorldAppOneXtraVar class implements an application that
>  * simply displays "Hello World!" to the standard output.
>  * an extra string member variable is added that will be outputed
>  */
> public class HelloWorldAppOneXtraVar
> {
>     String iMessage = "Hello  World";
>     public static void main(String[] args)
>     {
>         System.out.println(iMessage); //Display the string.


Compare: (<)C:\covet\clones\HelloWorldApp.java (287 bytes)
   with: (>)C:\covet\clones\HelloWorldAppTwoXtraVars.java (475 bytes)

1,9c1,12
< /**
<  * The HelloWorldApp class implements an application that
<  * simply displays "Hello World!" to the standard output.
<  */
< public class HelloWorldApp
< {
<     public static void main(String[] args)
<     {
<         System.out.println("Hello World!"); //Display the string.
---
> /**
>  * The HelloWorldAppTwoXtraVars class implements an application
that
>  * simply displays "Hello World!" to the standard output.
>  * two extra string member variables are added that will be
outputed
>  */
> public class HelloWorldAppTwoXtraVars
> {
>     String iFirstMessage = "Hello"
>     String iSecondMessage = "World";
>     public static void main(String[] args)
>     {
>         System.out.println(iFirstMessage + " " + iSecondMessage);
//Display the string.


Compare: (<)C:\covet\clones\HelloWorldApp.java (287 bytes)
   with: (>)C:\covet\clones\HelloWorldAppXtraClass.java (592 bytes)

1,9c1,24
< /**
```

```
<   * The HelloWorldApp class implements an application that
<   * simply displays "Hello World!" to the standard output.
<   */
< public class HelloWorldApp
< {
<     public static void main(String[] args)
<     {
<         System.out.println("Hello World!"); //Display the string.
---
> /**
>  * The HelloWorldAppXtraClass class implements an application that
>  * simply displays "Hello World!" to the standard output.
>  * Internal Class added and an array, and a loop!
>  */
> public class HelloWorldAppXtraClass
> {
>     static class HelloClass
>     {
>           char[] iMessage = {'H','e','l','l','o','
','W','o','r','l','d','!'};
>
>           public HelloClass()
>           {
>                 for (int i=0;i<=11;i++)
>                 {
>                       System.out.print(iMessage[i]);
>                 }
>                 System.out.println();
>           }
>     }
>
>     public static void main(String[] args)
>     {
>         HelloClass tHelloClass = new HelloClass();


Compare: (<)C:\covet\clones\HelloWorldApp.java (287 bytes)
   with: (>)C:\covet\clones\HelloWorldAppXtraComments.java (459
bytes)

1,6c1,10
< /**
<  * The HelloWorldApp class implements an application that
<  * simply displays "Hello World!" to the standard output.
<  */
< public class HelloWorldApp
< {
---
> /**
>  * The HelloWorldAppXtraComments class implements an application
that
>  * simply displays "Hello World!" to the standard output.
>  * extra comments have been added
>  */
> public class HelloWorldAppXtraComments
> {
>
>     //main method outputs hello world
>     //args is the parameters passed to the program by the command
line
```

```
Compare: (<)C:\covet\clones\HelloWorldApp.java (287 bytes)
   with: (>)C:\covet\clones\HelloWorldAppXtraIfs.java (606 bytes)

4,5c4,6
<   */
< public class HelloWorldApp
---
>   * Two meaingingless if statments are added
>   */
> public class HelloWorldAppXtraIfs
9c10,24
<          System.out.println("Hello World!"); //Display the string.
---
>          if (args[0].equals(""))
>          {
>            if (args[1].equals(""))
>            {
>                  System.out.println("Hello World!"); //Display the
string.
>                  }
>                  else
>                  {
>                        System.out.println("Hello World!"); //Display
the string.
>                  }
>            }
>            else
>            {
>                  System.out.println("Hello World!"); //Display the
string.
>            }


Compare: (<)C:\covet\clones\HelloWorldApp.java (287 bytes)
   with: (>)C:\covet\clones\HelloWorldAppXtraLine.java (382 bytes)

1,5c1,6
< /**
<   * The HelloWorldApp class implements an application that
<   * simply displays "Hello World!" to the standard output.
<   */
< public class HelloWorldApp
---
> /**
>   * The HelloWorldAppXtraLine  class implements an application that
>   * simply displays "Hello World!" to the standard output.
>   * command printing out is split over two line
>   */
> public class HelloWorldAppXtraLine
9c10,11
<          System.out.println("Hello World!"); //Display the string.
---
>          System.out.print("Hello");
>          System.out.println("World!"); //Display the string.


Compare: (<)C:\covet\clones\HelloWorldApp.java (287 bytes)
   with: (>)C:\covet\clones\HelloWorldAppXtraSwitch.java (585 bytes)

1,5c1,6
```

```
< /**
<  * The HelloWorldApp class implements an application that
<  * simply displays "Hello World!" to the standard output.
<  */
< public class HelloWorldApp
---
> /**
>  * The HelloWorldAppXtraSwitch  class implements an application
that
>  * simply displays "Hello World!" to the standard output.
>  * Switch statement added 3 cases ' ' , 'a' and default
>  */
> public class HelloWorldAppXtraSwitch
9c10,27
<         System.out.println("Hello World!"); //Display the string.
---
>         switch (args[0].charAt(0))
>         {
>                 case ' ':
>                 {
>                         System.out.println("Hello World!");
>                 };
>                 break;
>                 case 'a':
>                 {
>                         System.out.println("Hello World!");
>                 }
>                 break;
>                 default :
>                 {
>                         System.out.println("Hello World!");
>                 }
>                 break;
>           }
```

EuroConverterApp and Clones

```
Compare: (<)C:\covet\clones\EuroConverterApp.java (938 bytes)
   with: (>)C:\covet\clones\EuroConverterAppAddedFor.java (1158
bytes)

1,3c1,8
< import java.io.*;
<
< public class EuroConverterApp
---
> import java.io.*;
>
> /* all the questions asked are put into an array as are the answers
> *  the for loop is a static i = 0..2 and uses the index i to
recognise the question, answer pair
> */
>
>
> public class EuroConverterAppAddedFor
11,20c16,28
<           System.out.println("Which Currency do you want to work
in? For Example; German,  Portuguese");
```

```
<            String tCurrency = stdIn.readLine();
<            System.out.println("Converting t.o or f.rom euros?");
<            String toOrFrom = stdIn.readLine();
<            boolean toEuro = true;
<            if (toOrFrom.toUpperCase().charAt(0) == 'F')
<                  toEuro = false;
<            System.out.println("How much are you converting?");
<            String tAmmount = stdIn.readLine();
<            double tMoney = new Double(tAmmount).doubleValue();
---
>            String[] tQuestions =  {"Which Currency do you want to
work in? For Example; German,  Portuguese","Converting t.o or f.rom
euros?","How much are you converting?"};
>
>            String[] tAnswers = new String[3];
>            boolean toEuro = true;
>            for (int i = 0; i <= 2; i++)
>            {
>                  System.out.println(tQuestions[i]);
>                  tAnswers[i] = stdIn.readLine();
>            }
>            String tCurrency = tAnswers[0];
>            if (tAnswers[1].toUpperCase().charAt(0) == 'F')
>                  toEuro = false;
>            double tMoney = new Double(tAnswers[2]).doubleValue();


Compare: (<)C:\covet\clones\EuroConverterApp.java (938 bytes)
   with: (>)C:\covet\clones\EuroConverterAppErrors.java (1021 bytes)

1,3c1,5
< import java.io.*;
<
< public class EuroConverterApp
---
> /* missing ';''s from the logic */
>
> import java.io.*;
>
> public class EuroConverterAppErrors
12,13c14,16
<            String tCurrency = stdIn.readLine();
<            System.out.println("Converting t.o or f.rom euros?");
---
>            //missing ';' s on the next two lines
>            String tCurrency = stdIn.readLine()
>            System.out.println("Converting t.o or f.rom euros?")


Compare: (<)C:\covet\clones\EuroConverterApp.java (938 bytes)
   with: (>)C:\covet\clones\EuroConverterAppOneXtraVar.java (1178
bytes)

1,3c1,5
< import java.io.*;
<
< public class EuroConverterApp
---
> /* extra variable to hold the exchange rate once the currency has
been selected */
>
```

```
> import java.io.*;
>
> public class EuroConverterAppOneXtraVar
11a13,15
>           //added variable
>           double tExchangeRate = 0.0;
>
23c28,33
<           System.out.println("Exchange Rate is : " +
tEuroConverter.getExchangeRate());
---
>
>           //here the variable is set
>           tExchangeRate = tEuroConverter.getExchangeRate();
>
>           //variable outputed
>           System.out.println("Exchange Rate is : " +
tExchangeRate);

Compare: (<)C:\covet\clones\EuroConverterApp.java (938 bytes)
   with: (>)C:\covet\clones\EuroConverterAppTwoXtraVars.java (1391
bytes)

1,3c1,6
< import java.io.*;
<
< public class EuroConverterApp
---
> /* two extra varibles are added here String holding the name of the
file that contains the exchange
>  * rates and the second variable is a useless variable that does
nothing */
>
> import java.io.*;
>
> public class EuroConverterAppTwoXtraVars
9c12,21
<           EuroConverter tEuroConverter = new
EuroConverter("xchangerates.txt");
---
>
>           //first extra varible
>           String tFilename = new String("xchangerates.txt");
>
>           //second extra variable
>           String tUselessVariable = new
String("IAMAUSELESSVARIABLETHATDOESNOTHINGEXCEPTWASTESPACE.");
>
>           tUselessVariable = tUselessVariable.trim().toLowerCase();
>
>           EuroConverter tEuroConverter = new
EuroConverter(tFilename);


Compare: (<)C:\covet\clones\EuroConverterApp.java (938 bytes)
   with: (>)C:\covet\clones\EuroConverterAppVarNameChange.java (1366
bytes)

1,23c1,33
< import java.io.*;
<
```

```
< public class EuroConverterApp
< {
<     private static BufferedReader stdIn = new BufferedReader(new
InputStreamReader(System.in));
<
<     public static void main(String[] args) throws IOException
<     {
<           EuroConverter tEuroConverter = new
EuroConverter("xchangerates.txt");
<
<           System.out.println("Which Currency do you want to work
in? For Example; German,  Portuguese");
<           String tCurrency = stdIn.readLine();
<           System.out.println("Converting t.o or f.rom euros?");
<           String toOrFrom = stdIn.readLine();
<           boolean toEuro = true;
<           if (toOrFrom.toUpperCase().charAt(0) == 'F')
<                 toEuro = false;
<           System.out.println("How much are you converting?");
<           String tAmmount = stdIn.readLine();
<           double tMoney = new Double(tAmmount).doubleValue();
<
<
      System.out.println(tEuroConverter.convert(tCurrency,tMoney,toEu
ro));
<           System.out.println("Exchange Rate is : " +
tEuroConverter.getExchangeRate());
---
> /* systematic variable name changes using find/replace function in
text editor */
> import java.io.*;
>
> public class EuroConverterAppVarNameChange
> {
>     //in used to be stdIn
>     private static BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
>
>     //pArguments used to be pArguments
>     public static void main(String[] pArguments) throws IOException
>     {
>
>           //tEC used to be tEC
>           EuroConverter tEC = new
EuroConverter("xchangerates.txt");
>
>           System.out.println("Which Currency do you want to work
in? For Example; German,  Portuguese");
>           //tCountrySelection used to be tCurrency
>           String tCountrySelection = in.readLine();
>           System.out.println("Converting t.o or f.rom euros?");
>           //tConversionDirection used to be toOrFrom
>           String tConversionDirection = in.readLine();
>           //tEuroConversion used to be toEuro
>           boolean tEuroConversion = true;
>           if (tConversionDirection.toUpperCase().charAt(0) == 'F')
>                 tEuroConversion = false;
>           System.out.println("How much are you converting?");
>           //tMoneyStr used to be tAmmount
>           String tMoneyStr = in.readLine();
>           //tMoneyDouble used to be tMoney
```

```
>           double tMoneyDouble = new
Double(tMoneyStr).doubleValue();
>
>
      System.out.println(tEC.convert(tCountrySelection,tMoneyDouble,t
EuroConversion));
>           System.out.println("Exchange Rate is : " +
tEC.getExchangeRate());


Compare: (<)C:\covet\clones\EuroConverterApp.java (938 bytes)
   with: (>)C:\covet\clones\EuroConverterAppXtraClass.java (1251
bytes)

1,5c1,21
< import java.io.*;
<
< public class EuroConverterApp
< {
<     private static BufferedReader stdIn = new BufferedReader(new
InputStreamReader(System.in));
---
> /* extra class added - takes user input */
>
> import java.io.*;
>
> public class EuroConverterAppXtraClass
> {
>     //extra class
>     static class UserInputClass
>     {
>           private BufferedReader stdIn;
>
>           public UserInputClass()
>           {
>                 stdIn = new BufferedReader(new
InputStreamReader(System.in));
>           }
>
>           public String getUserInput() throws IOException
>           {
>                 return stdIn.readLine();
>           }
>     }
10,14c26,31
<
<           System.out.println("Which Currency do you want to work
in? For Example; German,  Portuguese");
<           String tCurrency = stdIn.readLine();
<           System.out.println("Converting t.o or f.rom euros?");
<           String toOrFrom = stdIn.readLine();
---
>           UserInputClass tUserIO = new UserInputClass();
>
>           System.out.println("Which Currency do you want to work
in? For Example; German,  Portuguese");
>           String tCurrency = tUserIO.getUserInput();
>           System.out.println("Converting t.o or f.rom euros?");
>           String toOrFrom = tUserIO.getUserInput();
19c36
<           String tAmmount = stdIn.readLine();
```

```
---
>           String tAmmount = tUserIO.getUserInput();


Compare: (<)C:\covet\clones\EuroConverterApp.java (938 bytes)
   with: (>)C:\covet\clones\EuroConverterAppXtraComments.java (1565
bytes)

1,5c1,24
< import java.io.*;
<
< public class EuroConverterApp
< {
<     private static BufferedReader stdIn = new BufferedReader(new
InputStreamReader(System.in));
---
> /* extra comments added - takes user input */
>
> import java.io.*;
>
> public class EuroConverterAppXtraComments
> {
>     //extra class
>     static class UserInputClass
>     {
>           //buffered reader standard input
>           private BufferedReader stdIn;
>
>           public UserInputClass()
>           {
>                 //new instance of the class
>                 stdIn = new BufferedReader(new
InputStreamReader(System.in));
>           }
>
>           //wrapper function for the BufferedReader.readLine method
>           public String getUserInput() throws IOException
>           {
>                 return stdIn.readLine();
>           }
>     }
9,15c28,39
<           EuroConverter tEuroConverter = new
EuroConverter("xchangerates.txt");
<
<           System.out.println("Which Currency do you want to work
in? For Example; German,  Portuguese");
<           String tCurrency = stdIn.readLine();
<           System.out.println("Converting t.o or f.rom euros?");
<           String toOrFrom = stdIn.readLine();
<           boolean toEuro = true;
---
>           //EuroConverter Object that will read in a list of
currencies and work out conversions
>           EuroConverter tEuroConverter = new
EuroConverter("xchangerates.txt");
>           UserInputClass tUserIO = new UserInputClass();
>
>
>           //user interaction
```

```
>            System.out.println("Which Currency do you want to work
in? For Example; German,  Portuguese");
>            String tCurrency = tUserIO.getUserInput();
>            System.out.println("Converting t.o or f.rom euros?");
>            String toOrFrom = tUserIO.getUserInput();
>            boolean toEuro = true;
>            //are we converting to or from Euro's
19,21c43,46
<            String tAmmount = stdIn.readLine();
<            double tMoney = new Double(tAmmount).doubleValue();
<
---
>            String tAmmount = tUserIO.getUserInput();
>            double tMoney = new Double(tAmmount).doubleValue();
>
>            //output the results


Compare: (<)C:\covet\clones\EuroConverterApp.java (938 bytes)
   with: (>)C:\covet\clones\EuroConverterAppXtraIfs.java (1016 bytes)

1,3c1,4
< import java.io.*;
<
< public class EuroConverterApp
---
> /* extra decision points int he form of ifs */
> import java.io.*;
>
> public class EuroConverterAppXtraIfs
21,23c22,26
<
<
      System.out.println(tEuroConverter.convert(tCurrency,tMoney,toEu
ro));
<            System.out.println("Exchange Rate is : " +
tEuroConverter.getExchangeRate());
---
>            if (true)
>            {
>
      System.out.println(tEuroConverter.convert(tCurrency,tMoney,toEu
ro));
>                  System.out.println("Exchange Rate is : " +
tEuroConverter.getExchangeRate());
>            }


Compare: (<)C:\covet\clones\EuroConverterApp.java (938 bytes)
   with: (>)C:\covet\clones\EuroConverterAppXtraLine.java (1028
bytes)

1,3c1,4
< import java.io.*;
<
< public class EuroConverterApp
---
> /*extra line added on the end by splitting up last line */
> import java.io.*;
>
> public class EuroConverterAppXtraLine
```

```
23c24,25
<            System.out.println("Exchange Rate is : " +
tEuroConverter.getExchangeRate());
---
>            System.out.println("Exchange Rate is : ");
>            System.out.println(tEuroConverter.getExchangeRate());

Compare: (<)C:\covet\clones\EuroConverterApp.java (938 bytes)
   with: (>)C:\covet\clones\EuroConverterAppXtraSwitch.java (1066
bytes)

1,3c1,3
< import java.io.*;
<
< public class EuroConverterApp
---
> import java.io.*;
>
> public class EuroConverterAppXtraSwitch
20a20,27
>            switch (tAmmount.charAt(0))
>            {
>                  case '0':
>                  {
>                        System.out.println("less than 1");
>                  }
>                  break;
>            }
```