Effective Clone Detection Without Language Barriers

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Matthias Rieger

von Österreich

Leiter der Arbeit: Prof. Dr. S. Ducasse, Prof. Dr. O. Nierstrasz
Institut für Informatik und angewandte Mathematik

Effective Clone Detection Without Language Barriers

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Matthias Rieger

von Österreich

Leiter der Arbeit: Prof. Dr. S. Ducasse, Prof. Dr. O. Nierstrasz
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 10.06.2005

Der Dekan:

Prof. P. Messerli

Abstract

Duplication is detected by comparing features of source fragments. The main problem for the detection is that source code is rarely copied exactly. The detection process must be able to ignore the superficial differences and to concentrate on fundamental similarities in order to find relevant duplication. While the high level information yielded by syntactic and semantic code analysis can be put to effective use, the drawbacks of these deep analysis techniques are most importantly the reduced adaptability to different programming languages. Because duplication is an ubiquitous problem, however, support for duplication detection and management is needed for every programming language in use.

In this thesis we investigate how the premises of simplicity and adaptability influence all phases of the clone detection process. We analyze how line-based string matching as basic feature comparison technique can be augmented by minimal parsing to improve detection sensitivity. We investigate which code normalization techniques remove the superficial differences and reveal the similarities. We show how clone candidates are retrieved from the results of the basic comparison. We propose measures to select the relevant clones from the set of all retrieved candidates. We finally develop a collection of quantitative visualizations that enable the assessment of the copied code in the context of the entire system.

We experimentally validate the proposed code normalization technique in terms of precision and recall, show how the proposed relevancy measures improve on simple size metrics, and discuss scalability issues. We also validate the line-based granularity, and perform a comparison of our technique with related string matching detectors.

Acknowledgments

I'd like to first thank Oscar Nierstrasz for the opportunity to work in the SCG, for advice, perspective, and trust. I then am much indebted to the tireless sponsors of this thesis: Serge Demeyer for handing me the initial assignment and continuing interest and advice up till the last minute, and Stéphane Ducasse for boundless encouragement, countless pep-talks, and for never shying away from getting his hands dirty.

I am grateful to Prof. Rainer Koschke for kindly accepting to write the Koreferat, Serge and Rainer for coming to Bern to join the jury of the PhD defense, and to Prof. Torsten Braun for accepting to chair the defense.

I am happy to say thanks to a long list of SCG members, students, and visitors, as well as FAMOOS collaborators whom I was fortunate to work with, learn from, and also play with: Franz Achermann, Gabriela Arévalo, Markus Bauer, Alexandre Bergel, Andrew Black, Oliver Ciupke, Juan Carlos Cruz, Koen DeHondt, Markus Denker, Markus Gaelli, Tudor Gîrba, Orla Greevy, Isabelle Huber, Christian Kaufmann, Michele Lanza, Markus Lumpe, Pietro Malorgio, Radu Marinescu, Robb Nebbe, Tamar Richner, Claudio Riva, Nathanael Schärli, Therese Schmid, Jean-Guy Schneider, Sander Tichelaar, and Roel Wuyts.

I don't want to forget the accidental helpers of this work: S. Wu, U. Manber and B. Gopal, who programmed and made available the AGREP tool, the *Drosophila melanogaster* of duplication research.

Finally, for the much needed basis, I thank family, friends, the righteous programmers out there, and Laura.

Matthias Rieger
May 2005

Contents

Chapter 1

Introduction

Meine Angst: die Wiederholung!

– Max Frisch[1]

As the first very large software systems were approaching their second decade of productive existence, the experiences in maintaining many a million source lines were accumulating and fueled the creation of the field of reverse and re-engineering of software. The research in this area has been growing steadily since the end of the 1980s. Within this context, research has for a decade now seriously looked at ways to cope with one of the scourges of the programmer who must keep an evolving system under control: multiple instances of the same fragment of code hidden in a system. Today, code duplication is recognized as one of the important problems in software development [FBB+99].

The utopian idea of software development is that an elegant solution is created which solves the task efficiently and which can be easily extended and adapted to new requirements. Pieces of code should be abstracted into reusable components on different levels, be it the method, class or component level. The slogan to promote programming which creates minimal redundancy and ballast goes

> *Say it once and only once!*[2]

In real life, software gets written under less than ideal conditions, often under considerable stress and deadline pressure. In real life, software systems are often too big to be fully understood by a programmer [DDN02]. Under these circumstances, programmers neither do nor can strive for ideal structures. They have to create a working system in the fastest possible way, which often means reusing pieces of software by *copy & paste*.

1.1 The Problem

Code duplication, or cloning, is essentially a form of software reuse: existing software artifacts are used in the construction of new code. As the alternative term *code scavenging* [Kru92] insinuates, duplicating code is considered to be a too primitive reuse activity for the righteous programmer. Reuse by scavenging is an informal and uncontrollable practice. Duplication activity is usually not documented, making the dependencies between parts of the code a system characteristic which is completely hidden. If we want to learn about it we must actively seek it out.

If a system is affected by duplication on a larger scale, the induced problems are summarized as follows:

- The spread of an error potentially contained within the duplicated code.

[1] "What I fear: Repetition!" Max Frisch, Swiss playwright and novelist, 1911–1991.
[2] Available from `http://c2.com/cgi/wiki?OnceAndOnlyOnce` [May 15, 2005]

- The increased amount of source code, leading to bloat, increased cognitive load and multiplied maintenance effort during updates and changes.

A system containing code duplication is more error prone and more resistant to change. If we want to keep our systems lean we have to expurgate the duplication from time to time, which requires us to first detect where it hides. However, the problem of detecting duplication is not trivial. Consider the following requirements for detecting duplicated code:

Self Similarity. We have no way of knowing beforehand which pieces of the code have been copied and cannot search for specific patterns. Ultimately we must extract the search pattern from the search domain itself, *i.e.*, compare each element with each other element. This task is of $O(n^2)$ time and space complexity. Depending on the granularity level on which we have set our aim, this can be a heavy burden for large systems.

Scalability. Duplicated code is most problematic in large, complex software systems. For this reason, a useful tool must be able to cope with very large code bases.

Multiple Languages and Dialects. There are thousands of programming languages in use today, and dozens of dialects of the most popular languages (like COBOL). A useful duplicated code detector must be robust in the face of syntactic variations in programming languages [BYM+98].

Avoid False Positives. False positives occur when code is marked as duplicated that should not be. Programming language constructs, idioms, and recurring statements should not normally be considered as duplicated code, since they do not indicate *copy & paste* problems.

Avoid False Negatives. Frequently, duplicated pieces of code are not textually equivalent. Any number of editing transformations applied to the copied code will disguise the similarity but not break the duplication relationship.

Useful Results. Detecting duplication is only the first step. The next step is to reengineer the code, *e.g.*, extract the cloned code into a function or macro which can be invoked from everywhere. The duplicates that are reported by a tool should thus be code fragments that lend themselves to such a reengineering effort easily.

These challenges cannot all be addressed simultaneously. For example, like every information retrieval technique a clone detection method which avoids false negatives will produce more false positives. A choice must be made to favor some of the challenges over others. Such a choice, made from the perspective of the reengineering endeavor itself and its context, is what guides the work presented in this thesis.

1.2 Research and Contributions

When surveying the literature on clone detectors we find that they mostly emphasize the requirements *Avoid False Positives*, *Avoid False Negatives*, or *Scalability.* However, from the viewpoint of continuous software maintenance we are convinced that since code duplication is an ubiquitous phenomenon, detection techniques need to be in the toolbox of everybody who is in charge of a reengineering effort. Such efforts are undertaken for systems written in any language. We therefore have to emphasize the requirement of *Multiple Languages and Dialects*. On the level of tool support this means that a tool should be easy to adapt to a different language without requiring an expert in parsing technology. Following these considerations we can formulate our research question:

> What are the potentials and the problems of techniques for clone detection and analysis which are built under the premise that switching the programming language should not require more than reconfiguration of a detection tool?

Mechanical detection of duplication in source code is achieved by comparing some selected attributes of the code. Not all attributes we can find in source code are of the same importance, however. Where some of them represent essential features and are therefore strong indicators for the similarity of two source fragments, others represent only superficial features which may be easily changed, disturbing the detector with unimportant dissimilarities. To be efficient we need to distinguish between the essential and the superficial attributes. Parsers are used to build source representations like abstract syntax trees where such distinctions can be made easily. A common approach to detecting duplication is therefore to compare syntax trees or attributes derived from syntax trees [MLM96b][BYM+98][Kri01]. The problem is however that a parser is an intricate mechanism. Getting a grammar—in the appropriate format—for the language at hand, and 'installing' it in a parser is an expensive part of the detection process.

Under the *easy adaptation* premise given above we will have to base the comparison on attributes which can be extracted from the code without a full parser. This thesis investigates how we can balance the use of parsers by *i)* minimizing their usage and trying instead to employ cheaper techniques, and *ii)* by building, where necessary, generic parsers that can be adapted by simple configuration.

The problems of using parsing only to a minimal extent are that less parsing means less information about the code. This makes it harder to eliminate false positives and to derive additional knowledge which could guide the reengineering afterwards. Detecting duplicated code without parsing the source code can be compared to playing the piano with thick gloves on. The goal of this thesis is to make the tune still pretty enough so people want to listen to it.

List of Contributions

The overall contribution of this thesis is an in-depth analysis of the possibilities of clone detection using simple methods which are adaptable for a large range of programming languages. In detail, the work presented in this thesis contains the following contributions:

- We identify four reengineering goals with which to categorize duplication detection approaches.
- We analyze the process of clone detection in general, identifying for example the notions of *fixed* and *free* granularity clones as an important distinction for clone detectors.
- In addition to the basic entities of the *clone pair* (two copied fragments) and the *clone class* (an aggregation of multiple similar fragments) we introduce the notion of *clone class family* which combines the clone relation with the physical neighborhood of copied fragments to form an aggregation criteria.
- We analyze and describe the process of code duplication detection under the premises of using only minimal parsing methods [DRD99]. This process encompasses three steps:
  1. *Input Transformation:* We propose a number of normalization mechanisms for lightweight syntactic elements in source code, and analyze the tendency of these methods to generate false positives in the comparison.
  2. *Comparison:* We describe a simple comparison algorithm which creates a very detailed picture of the duplication situation. We then describe how to extract instances of interesting duplication (clone pairs) from this representation, and how to assemble the pairs to clone classes and clone class families.
  3. *Ranking and Filtering:* We propose a number of measures for clone pairs to evaluate their relevance with respect to a number of reengineering tasks. As a basis for these measures we propose a representation which reduces source code to a number of *features* that are being used to get a finer grained similarity value between two code fragments.
- In order to ease the user's handling of clones we discuss a set of requirements for clone browsers and how they could be integrated into editors.
- As another way to give a user insight into the duplication we develop a new approach to visualize duplication quantitatively, setting it in relation to the context of the system it occurs in [RDL04].

- In a number of case studies we validate the effectiveness of the proposed techniques with respect to the chosen reengineering goals and compare our results to the results of other detection approaches of similar characteristics [DNR05].

1.3 Outline of the Thesis

In Chapter 2 we explain the problem of duplicated code, how it comes to pass, what its negative effects are and where it brings benefits. We introduce the measures that are taken to detect and deal with duplicated code. We present a categorization of the different approaches to detection according to four general reengineering goals, and we motivate our own position within this categorization.

A survey of the state of the art in clone detection can be found in Appendix A, including brief overviews of related fields like compiler optimization (Appendix A.7) and plagiarism detection (Appendix A.8), where self similarity of source code is detected and exploited under different premises.

In Chapter 3 we explore on a general level the definition of clones in source code and introduce the vocabulary used in the thesis.

In Chapter 4 we study the potential and the problems for detecting and analyzing clones under the restriction of using only language independent techniques. The chapter covers the three phases before, during, and after the comparison:

1. Transformation of the source code to normalize superfluous differences (Section 4.2).
2. The organization of the comparison and the retrieval of clone candidates from the atomic matches (Section 4.3).
3. Ranking and filtering the candidates in order to obtain relevant clones for a number of reengineering tasks (sectrefsect:rankingandfiltering).

In Chapter 5 we present different means which make it easier for users to handle the retrieved duplication. We first discuss the requirements for a code editor which lets users browse clones at the level of source code and we show two ways to visualize duplication:

- The *dotplot* is a viable way to investigate the duplication situation for smaller, constrained areas which contain many clones (Section 5.3).
- *Polymetric views* enable navigation of large amounts of duplication in bigger systems (Section 5.4).

In Chapter 6 we validate a number of aspects of the proposed techniques such as

- the degree of *language independence* that is achieved,
- the influence of the *normalization techniques* on precision and recall of the clone detection,
- a comparison of our results with findings from related string-based detection approaches,
- the influence of code layout normalization measures like *pretty printing* on recall,
- the effectiveness of the proposed *ranking* measures,
- and the *scalability* of the proposed techniques implementation.

In Chapter 7 we summarize the main contributions of the work and give an outlook on possibilities for future work in the field.

In the appendix we present some technical discussions:

- As already mentioned, Appendix A contains the overview of the related work.
- In Appendix B we list all editing operations which can change a piece of code once it is copied. For each edit operation we list how clone detection is able to deal with it.
- In Appendix C we present how the removal of superfluous detail in program code can be implemented using an expressive regular expression engine like the one found in PERL.

Chapter 2

Copy & Paste: A Simple Reuse Mechanism

The copying[1] and subsequent pasting of pieces of written works is an old problem. Even before the advent of computerized text processing it was well known that ethically challenged authors were taking more than mere inspiration from related works. Electronic text processing and the ubiquity of the originals in digital formats just made the task of finding relevant material (and being reasonably sure to go undetected) so much easier that the problem has gained epidemic status with the advent of the world wide web[2]. Whereas plagiarism always entails legal problems, the copying of source code, which has been described as the simplest of reuse mechanisms by Krueger [Kru92], is only doubtful from the practitioners viewpoint. For software maintainers duplication of code causes a number of severe problems such as increased work load and defect probability.

In this chapter we elucidate the background of the problem in more detail and in a broader context. Focusing on the problem of copying *program source code*, not natural language text, the chapter paints a kind of *clone life cycle*.

In Section 2.1 we investigate the duplicated code from close up. To again make clear why duplication is a problem we start by listing the negative effects that duplicated code has on, for example, the maintainability of a system. We explain some of the causes that lead to code being duplicated. We also briefly list situations in which duplicating code can not be avoided, and finally some actual benefits of clones.

In Section 2.2 we introduce the infrastructure that is developed to combat clones. We describe the basic structure of clone detectors and the different clone management strategies, most prominently the removal of clones via reengineering actions such as refactorings.

In Section 2.3 we finally formulate four reengineering goals which allow a categorization of clone detection approaches according to how they support these goals and we explain a viewpoint which guides our selection of the two goals that are important to us. The selected goals are the basis for our investigations in Chapter 4.

To avoid blocking the presentation with too much technical detail, the state of the art of clone detection is found in Appendix A, where we present a detailed comparison of different approaches under various aspects.

[1] The multiplication of the information only takes place if it is *copied* and pasted, not *cut* and pasted, as we have seen carelessly written in many articles and books on the topic.

[2] Available from `http://plagiarism.org/plagiarism.html` [May 15, 2005]

Our Contribution

The contributions of this chapter are:

- The accumulation of the most comprehensive list of duplication causes, problems, and benefits in the literature to date.
- The description of a design space of duplication detection approaches with a list four reengineering goals.

2.1 Duplication of Source Code

> Information on how a piece of software operates—be it an algorithm, a set of constants, human-readable documentation, or something else—should exist in only one place.
>
> –*The* OAOO *Principle (Once and only once)*

The name of the OAOO principle with its twofold incantation of *once* is of course tongue-in-cheek. The rhetorical device that is employed—repetition of a word for emphasis (Epizeuxis)—accentuates the importance of the concept of singularity in software development. On the other hand it also hints at how routinely the principle is broken. The duplication of information and the problems that arise when trying to keep the different instances synchronized is one of the major problems of software development, "Number one in the stink parade is duplicated code" as Fowler et al. [FBB+99] claim in their enumeration of bad software *smells*.

The range that is currently reported most often as the amount of duplication found in industrial systems is roughly 10% to 25% (see below). There has however not yet been a comprehensive study of duplication in industrial systems. All data that exists is anecdotal evidence stemming from the case studies performed by the detection researchers. Also, these numbers cannot all be compared directly, as the different detection methods disagree on the exact characteristics of what is considered to be a clone. In a comparison of different clone detection tools, Bellon [Bel02a] has found that for the same case studies and with a commonly agreed upon minimal size for a clone, the tools reported wildly differing numbers of clones on account of the differences in filter criteria employed to select valid clone candidates.

Baker [Bak95a] reports that 19% of the X Window System can be considered duplicated code. Mayrand et al. [MLM96b] report that 6.4% and 7.5% of the overall number of functions of the system were found to be exact copies in a large (15 million LOC) telecommunication system. Case studies that we have performed [DRD99] have revealed up to 50% of duplication in an accounting system. Jarzabek&Shubiao [JS03] even found that 68% of the JAVA Buffer Library from JDK 1.4.1 could be considered redundant and be removed with a fine-grained refactoring technique.

This section presents the negative consequences of code duplication, and the reasons that code gets duplicated. We also present duplication that cannot be avoided and some positive aspects of duplication.

2.1.1 Negative Effects of Code Duplication

Duplicating code has a number of negative effects on the quality of the code. Besides increasing the amount of code that has to be maintained, duplication increases requirements on the resources and on the cognitive performance of the programmer [Joh94a][MLM96a][MLH96]. The following list gives detailed overview of these negative effects:

- **Increased Work Load for Maintainers:**
  - When maintaining or enhancing a piece of code, duplication multiplies the work to be done.
  - If a copied software fragment is found to contain a defect, the defect will probably have to be corrected in every instance of the code. Since usually no record of the duplication process exists, one cannot be sure that the defect has been eliminated from the entire system without performing a clone analysis.
- **Increased Defect Probability:**
  - The adaption of duplicated code to the new context into which it is copied is an informal process which does not shield the programmer from any details (white box reuse). Errors are thus likely to happen, *e.g.*, name clashes between variables from the copied code and variables in the new context may go unnoticed. Dependencies that are not fully understood are another source of potential defects.
  - Adaptations may be forgotten by a hasty programmer because the copied code looks complete.

  - If in large systems multiple maintainers unaware of each other change or update clones, their fixes will very likely not be identical and threaten coherent system behavior.
  - It is even possible that adaptations are applied erroneously, out of the (wrong) belief that one method is a clone of another.

- **Increased Cognitive Load for Maintainers:**
  - If large pieces of software are copied, parts of the code may be unnecessary in the new context. Lacking a thorough analysis of the code, they may however not be identified as such. It may also be the case that they are not removable without a major refactoring of the code. This may, first, result in *dead code* which is never executed, and, second, the code will be a "red herring", increasing the cognitive load of future maintainers.
  - Larger sequences repeated multiple times within a single function make the code unreadable, hiding what is actually different in the mass of code that is the same. Code is then also likely to be on different levels of detail, slowing down the process of understanding.
  - If all copies are to be enhanced collectively at one point, the necessary enhancements may require varying measures in the cases where copies have evolved differently. As an extreme case, one can imagine that a fix introduced in the original code actually breaks the copy.

- **Increased Resource Requirements:**
  - The growth rate of the system is much higher than if it was only subject to the normal effects of new requirements. In systems with stringent hardware constraints, this may result in the premature exhaustion of the resources (application footprint). If a system is deployed in tandem with a specific hardware platform, like a telecommunication switch, a software upgrade could entail a costly upgrade in hardware as well.
  - Compilation times will increase if more code has to be translated which has a detrimental effect on the edit-compile-test cycle.

The overall effect of cloning has been described by Johnson [Joh94a] as a form of *software aging* or "hardening of the arteries" where even small changes on the architectural level become very difficult to achieve in the actual code. In the same vein the Portland Pattern Repository's Wiki claims[3] that "Redundancy is Inertia" since it takes more effort to move the system into a new direction.

2.1.2 How Duplication Comes to Pass

Most clones are no accidental creations. It is a rare case that two different programmers would unbeknownst of each other code up the same piece of functionality in a form that fulfills the definition of a clone. There are only a few scenarios which describe the creation of accidental redundancy:

- The *Not Invented Here* syndrome, an unwillingness to read and understand the code of other people, may preclude developers from reusing existing functionality.
- The phenomenon of *Reinventing the Wheel* is a possibility in large systems with a long history and poor reuse documentation. If a system or programming library is not prepared for reuse with the proper description and search facilities [Kru92], the effort to search the library for the needed functionality may exceed the effort to write the method from scratch.
- Baxter [BYM+98] mentions *mental macros*, a short idiomatic piece of logic which is implemented in the same way from memory every time it is needed.

[3] Available from `http://c2.com/cgi/wiki?RedundancyIsInertia` [May 15, 2005]

For these exceptional circumstances the functionality, albeit the same semantically, will probably be embedded in different syntactic representations, making it difficult to automatically detect it.

The bulk of software clones are however created when programmers scavenge code fragments they already know—which in most cases means they have written it themselves—from the existing code base and adapt them to the context of the new location. Code duplication is thus an intentional activity by people who (mostly) are conscious about what they are doing even when they are not fully aware of the consequences this activity may have in the long run. Code duplication does not just happen, it is actively produced. A number of environmental forces however are conducive to programmers reusing code by means of duplication. We roughly order the following list according to diminishing importance as we perceive it:

- **Time Pressure:**

  If the code has to be finished as quickly as possible there is no time to analyze the context and design a general solution.[4] It is quicker to copy the code and customize it for the new case. As an additional benefit existing code is already tested and is a safe bet for copying.

- **'Active' Programming Strategies:**

  As Rosson&Carroll have described [RC93], the practice of "reuse of use" is commonplace among professional programmers. Focusing on the product to be delivered, programmers shy away from a thorough analysis of existing example code and planning activities, but rather use the available resources for a quick start towards the implementation goal.

  Cordy [Cor03] reports that cloning is a commonly practiced reuse strategy in the financial industry, where new tasks (representing financial products) do not change very much from the existing ones. Programs to handle these tasks are copied from the existing source and adapted in the copy rather than being implemented on top of a newly introduced abstraction layer. The reason is that the high risk (monetary consequences of software errors can run into the millions in a single day) dictates that code that has been thoroughly tested (70% of the software effort in the financial domain is spent on testing) is not to be changed, *i.e.*, abstracted or parameterized.

- **Lacking Abstraction Mechanisms:**

  If the programming language lacks some abstraction mechanisms, *e.g.*, inheritance, generic types (called *templates* in C++) or parameter passing (missing from, *e.g.*, assembly language and COBOL), programmers will repeatedly have to implement these as *idioms*. This will lead to possibly small but potentially frequent clones. Patenaude et al. [PMDL99] report that 15% of all confirmed clones of seven open source JAVA systems were due to the absence of generics in the pre-1.5 versions of the language.

- **Unfamiliar Technology:**

  If the developers are unfamiliar with the technology they will use examples as educational and inspirational starting points. This will likely lead to *wide miss* clones, duplicated instances where the similarity is still apparent in the coarse structure but the details have been changed.

- **Code Ownership:**

  In a team development environment with code ownership, situations may arise where the owner is not able to give access to his code or cannot be convinced to update the function with the enhancements to make it reusable. The potential reuse client is then forced to appropriate, *i.e.*, copy and adapt the code.

- **Lacking Awareness:**

  Code duplication as a research topic exists for a little over a decade now, and refactoring has become a well known notion only recently. These topics are thus only now becoming integrated into software engineering education. Most programmers need a certain experience in maintaining software before they become aware of the detrimental effects of duplication.

[4] Yourdon, in the *Decline and Fall of the American Programmer*, estimates the additional expense to make a component reusable as *"twice the effort of a 'one-shot' component"* (qtd. in [Bro95]). Brooks l.c. increments the estimate to the threefold effort ratio.

- **Efficiency Considerations:**

  Function calls may be deemed too costly, *e.g.*, in real time programs. If the compiler does not offer to inline the code automatically, this will have to be done by hand.

- **Programmer Productivity Evaluation Measure:**

  If programmers are paid by the number of lines they write, duplication is just the right means to produce lots of working code fast.

- **Different Versions:**

  Different ports of the same subsystem will likely be similar. LINUX kernel device drivers, for example, have been found to contain large rates of duplication [GT00]. Among the catalyzers for copy and paste under these circumstances are:

  – The drivers have all the same interface and a rather simple logic. Drivers for the same family of devices have even more cloning.
  – Poor abstraction: The design of the system does not allow for more sophisticated forms of reuse.

- **Merging of Two Systems:**

  If two systems with similar functionality are merged the likelihood of clones is increased, especially in the base libraries [Gie03]. Since these systems have been developed by different teams, the syntactic similarity of the clones might not be high, making them harder to detect.

- **Deployment Constraints:**

  Dagenais et al. [DMLP98] mention the case where a function may not be modifiable because it is stored in the non-volatile memory of an embedded system and cannot be replaced.

There exist to our knowledge no theories and no data on the influence of the following factors on the existence and the amount of duplicated code found in a system:

- the programming language, or the programming paradigm in general,
- the abilities of the programmers,
- the size of the project,
- the domain of the project,
- organizational conditions of the project, or
- events in the project history like forks, merges, ports, and quick fixes.

It is difficult to find answers to these questions, since to be able to determine the influence of one of the variables one would have to keep the other variables unchanged. This is an impossible task for real world applications. All we currently have is anecdotal evidence.

Regarding some of these questions, the data of the comparative study by Bellon [Bel02a], for example, where four C systems and four JAVA systems were provided for clone-investigation, did not allow any conclusions on the dependence on project size, and only a feeble indication of a programming language dependency by a slightly higher duplication rates for the C code. The sample of case studies was however too small to be able to formulate a firm verdict. From our own research [DRD99] it seems that lower level programming languages like COBOL that are still very close to assembly language and thus lack a number of higher level mechanisms, are susceptible to duplication, indirectly confirming the argument of Krueger [Kru92] that higher level programming languages are the first and most successful code reuse technique in software engineering. It might however also be the case that the rampant duplication in one COBOL accounting system [DRD99] is due to the high risk domain which dictates that tested code must be copied rather than abstracted to extend the system.

We are currently of the opinion that duplication is mainly put forth by programmers remembering their own solutions in the context of the work they have already done on the system.

2.1.3 Unavoidable Duplication

Sometimes duplication cannot be avoided even with the best intentions. Some of these cases are *controlled* duplication, other cases do not have any negative consequences beyond an increased workload. Some of the reasons for such types of duplication we have already seen in the last section:

Missing Generics: Duplication is unavoidable if the programming language does not offer the necessary mechanisms to safely implement a generic solution to a problem. In C, one has to implement a list for `float`-elements as well as one for elements of type `char`, whereas in C++ one can use the template mechanism. The rigidity of static type checking which requires specific language constructs to allow for generics thus impedes reuse and leads to duplication.

Generated Code: Duplication that is created by code generators, *e.g.*, lexical and syntactic analyzers written using LEX and YACC, the generation of object stubs in CORBA, or the template instances in C++, is—except for the effect of the bloat on system resources—of no concern to the developer. There is always a single authorative source of the knowledge (*e.g.*, the interface definition in CORBA, the template source in C++) to which changes must be applied. The duplication is not only machine-generated, it is also machine-maintained.

Code Ownership: If libraries or frameworks are closed, *i.e.*, have been frozen and cannot be adapted any more, it is not feasible to extract and share the code.

Duplication can also be forcibly introduced by module borders which are declared impermeable due to architectural considerations. This can be a layered system architecture, or an object-oriented class hierarchy where the two methods are in unrelated classes or too far apart in the hierarchy to use inheritance for code sharing. If the creation of a utility layer or a utility class is not feasible (*e.g.*, when low cohesion is prescribed), the duplication cannot be extracted.

If we switch to a finer granularity level we find more duplication of information, for example in method signatures. This is mostly due to the distinction between *declaration* and *definition* of software entities. Usually, an entity can be declared multiple times and must be defined once. Examples are the header files in C/C++, and the import statements in JAVA. Object-oriented polymorphism additionally allows the definition of separate entities having the same form, *i.e.*, signature. Examples are interfaces and abstract classes which declare a set of methods. Two reasons speak against the status of "duplication" for this information. For one, the duplication is not without trace, *i.e.*, does not have to be *detected* specially. Also, inconsistencies between the different instances are immediately caught by a compiler. However, the information must be kept consistent manually which can be tedious.[5]

2.1.4 Benefits of Code Duplication

While being mostly a scourge to maintainers, duplication can have some benefits too: Software readability and understanding, design and maintenance, as well as smaller likelihood to fail may all increase thanks to duplicated code. We will in turn discuss the benefits of consciously implementing similar functionality more than once, the beneficial knowledge gleaned from finding the duplication, and the benefits derived from not removing found duplication. As a note of caution, however, it should be emphasized that the beneficial effects of code duplication are sparse and far between. In general, it can be safely said that duplications detrimental effects by far outweigh the benefits of a few special situations.

Benefits of Duplicating Functionality: There exist circumstances under which duplicated code is created with full awareness of the consequences. Among the benefits are:

[5] This is exemplified by the fact that newer languages like JAVA have abolished the distinction between header and implementation files from their forebearer language C.

- Two instances of copied code are, although conceptually linked, syntactically and semantically independent of each other. This means that they can evolve at different paces and there is no danger that a change to one location will inadvertently affect the other. Had we abstracted the shared functionality into a common piece of code and a change requirement for only one of the locations made a rewrite of the common code necessary, the second location could potentially be affected as well: a full test would be required for both locations. Keeping the two locations independent may thus speed up maintenance, especially when automated regression tests are absent. Cordy [Cor03] reports that for financial institutions where 70% of the software maintenance effort is spent on testing, keeping programs for similar domains separate is imperative.

- In life-critical systems redundancy is built-in by design to increase robustness. Often the same functionality is developed by different teams in order to reduce the probability that the implementations fail under the same circumstances. The exchange of code is thus strictly prohibited which makes this an example of duplication of functionality rather than code. Since maintenance just as initial development is split, our problem statement does not really apply to these cases.

- For functions that have a parameter with a very restricted domain, *e.g.*, boolean switches, we can make the parameter a constant and express it through the names of the copies of the function, kept nearby each other in the source file. This may make the calling code easier to read. In the same vein, storing a staple of related values in stand-alone variables instead of in arrays has the drawback of reducing their potential to be treated using a loop. On the positive side, however, the readability of `A1` is arguably higher than the one of `A[1]` or `valueA` as can be observed in Listing 2.2 on Page 16.

Benefits of Detecting Duplication: Besides the immediate benefit of knowing how to improve the quality of the source code through a refactoring, there are other advantages stemming from the awareness of different instances of the same code.

- Davey et al. [DBF+95] and Burd&Munro [BM97] have remarked that code that has been copied many times has apparently proven its usability which makes it a prime candidate for inclusion in some sort of library, to announce its reuse potential officially.

- From just the awareness of the presence of a piece of code in a certain area of the system we can derive information about the purpose of its context. For example, when we have a piece of code managing memory we know that all files which contain a copy must implement a data structure with dynamically allocated space.

- A collection of different copies of the same source fragment may be a good overview of the usage patterns for this functionality.

Benefits of Not Removing Duplication: There exist a number of reasons why one would deliberately let duplicated code live in a system. Note that many of these reasons depend on opinion and cannot be decided in the abstract.

- The simplest reason might be that inlining a piece of code is a possible means to save the cost of a function call. In the light of current and anticipated speeds of the CPU these kinds of efficiency concerns are however of diminishing importance.

- Another argument is that writing reusable code needs a lot of work. Maintaining two copies of the same code may be much easier than the effort to produce a general but probably more complicated solution.

- It is argued[6] that it might be better to let duplication live a little before removing it. If we have multiple copies of the same code we can discern the stable from the variable elements better. This guides an eventual abstraction effort. Trying to get an abstraction right from the start may be premature optimization.

[6] Available from `http://c2.com/wiki?CodeHarvesting` [May 15, 2005]

```
String[] lookup = new String[7];
lookup[0] = createLookupKey(operation,resource,username);
lookup[1] = createLookupKey(null     ,resource,username);
lookup[2] = createLookupKey(null     ,null    ,username);
lookup[3] = createLookupKey(operation,resource,null    );
lookup[4] = createLookupKey(null     ,resource,null    );
lookup[5] = createLookupKey(operation,null    ,username);
lookup[6] = createLookupKey(operation,null    ,null    );
return lookup;
```

Listing 2.1: An unfolded loop presents a readable repetition.

- Removing duplication may also be going against an essential purpose of a high-level programming language: readability for a human. Picture a function `foo` from which a part is extracted (for duplication removal reasons) and now forms the function `foo_extr`. Say that `foo_extr` needs a large number of parameters and performs a conceptually difficult task for which an intuitive name can not be found (abstract and fragmentary data manipulations are difficult to name). If we additionally assume that the navigation in the development environment from `foo` to `foo_extr` is hindered slightly (by long methods, or source files far apart), it may be likely for the programmer to mentally loose the context of function `foo` while trying to understand `foo_extr`. The original code with the duplication still in place is then certainly easier to understand than the refactored code.[7] The ease of reading source code can thus be an argument against the refactoring of duplicated code.

- As an extension of the previous argument, Church and Helfman [CH93] have claimed that the rhetorical device of *repetition* (which is used to convey emphasis or parallelism in human speech) should also be an admissible means for programmers writing code. This is not such a far-fetched thought since the rhetorical devices that use repetition for emphatic purposes rely on the fact that the repetitions occur nearby each other, *i.e.*, in the same sentence. Code duplication that has this same property, *i.e.*, occurring in the same file or even the same function, is not problematic as it is plainly obvious to the programmer and thus quickly adapted.[8] Repetition of similar code may contribute to the readability of the code (see Listing 2.1 and Listing 2.2), shedding the extra logic that would be necessary for the generalization and abstraction (Listing 2.3).

If it is decided, for whatever reason, not to remove duplication from a system, it will be necessary that this fact is recorded in the source code itself or in other texts which are consulted by the maintainers. *Documented duplication* will still require more effort to maintain but the potential for negative effects that arise from not knowing about the multiple instances can be controlled (see also *Compensatory Clone Management* in Section 2.2.3).

[7] The extension of this argument is of course that if `foo_extr` can be fully understood and an intuitive purpose can be mentally assigned to it by the programmer, then `foo` and the other places where `foo_extr` is invoked are easier to read than before the extraction.

[8] This is the reason that certain clone taxonomies, for example Kapser&Godfrey's [KG03], make a distinction between code copied within the same file or code copied over the boundaries of files or even across directories

```
r1 = r_Init1 & B0;
A0 = ((B0 >>1 ) & CMask) | r1;
r1 = r_Init1 & B1;
r2 =  B0 | (((A0 | B0) >> 1) & r_NO_ERR);
A1 = ((B1 >>1 ) & CMask) | r2 | r1 ;
if(D == 1) goto Nextcharfile;
r1 = r_Init1 & B2;
r2 =  B1 | (((A1 | B1) >> 1) & r_NO_ERR);
A2 = ((B2 >>1 ) & CMask) | r2 | r1 ;
if(D == 2) goto Nextcharfile;
r1 = r_Init1 & B3;
r2 =  B2 | (((A2 | B2) >> 1) & r_NO_ERR);
A3 = ((B3 >>1 ) & CMask) | r2 | r1 ;
if(D == 3) goto Nextcharfile;
r1 = r_Init1 & B4;
r2 =  B3 | (((A3 | B3) >> 1) & r_NO_ERR);
A4 = ((B4 >>1 ) & CMask) | r2 | r1 ;
if(D == 4) goto Nextcharfile;
```

Listing 2.2: Another example of self-similar, repetitive code that is more readable than a construction involving arrays and loops (from the AGREP system).

```
r1 = r_Init1 & B[0];
A[0] = ((B[0] >>1 ) & CMask) | r1;
for(j=0; j<4; j++)
  {
   r1 = r_Init1 & B[j+1];
   r2 = B[j] | ((( A[j] | B[j]) >> 1) &  r_NO_ERR);
   A[j+1] = ((B[j+1] >>1 ) & CMask) | r2 | r1 ;
   if(D == j+1) goto Nextcharfile;
  }
```

Listing 2.3: The code of Listing 2.2 refactored with array variables and a loop.

2.2 Duplication Detection and Management

As we have explained in Section 2.1.2, duplication is created both consciously or unconsciously. In any case, the activity seldomly leaves any traces, neither in the source nor in the documentation. Duplicated code must therefore be detected *ex post* by special means. This section discusses (on an abstract level) how duplication is detected, how clones are managed and eventually removed.

2.2.1 Catching Duplication at Creation Time

Whereas common duplication detection approaches work with the source text that results from the duplication activity of an engineer, an alternative idea is to catch duplication in the moment of creation. Horwitz mentions that, following this idea, text editors could be tagging the lines of the source code that are edited with them [Hor90]. Whenever the programmer uses copy and paste, the correspondence tags along with the source lines are transferred to the new location. Copied lines that are modified still keep their tags. The data thus collected represents the source line correspondences that must otherwise be detected by special tools. The information can be combined to clones straightforwardly. There are a number of open questions and problems with this idea that arise in all practical settings and can be readily seen:

- What amount of post-copy editing will remove the clone label from the copied line and give it a fresh one?
- Is the granularity of lines detailed enough or should code be tracked at the level of expressions?
- The tagging data would have to be held in a separate database since it should be hidden from the programmer and can thus not be put in the source file.
- The tagging scheme is bound to a development environment that supports it. If source code is altered with an alternate editor, the correspondence tags would still have to be computed by special means.
- Duplication due to “mental macros”—frequently used idioms that are not copied but written in the same fashion by remembering it—or simple transcriptions would still have to be detected by special means.

In any case, such a system could only be used for new code. Duplication, however, hides out in the millions of lines of legacy code developed by traditional means. To nab it, we need techniques which detect duplication after the fact.

2.2.2 Finding Duplication in Source Code

A clone detector must try to find the pieces of code of high similarity in a system’s source text. The main problem is that it is not known beforehand which code fragments can be found multiple times. The detector thus essentially has to compare every possible fragment with every other possible fragment.

The detection process is coarsely broken down into phases:

1. *Code Partitioning:* The entire source text of the system is broken down into the fragments between which the similarity relation is going to be established. At this point, code that is not interesting (*e.g.*, generated code) is removed from the input.
2. *Transformation:* The source code is represented in a format from which the comparable properties can be extracted.
3. *Comparison:* The source code properties of the fragments are compared. The output is a list of matches which are either already clone pair *candidates* or must be aggregated to form clone pair candidates

4. *Filtering:* Identification of relevant clone pairs among all candidates. A ranking can be established among the clone candidates to direct the energies of the reengineer.
5. *Aggregation:* To reduce the amount of clone data, the clone pairs are aggregated to clusters, classes, or cliques of clones.

What determines a detector approach to the greatest extent is the choice of the features which are used for the comparison. The source code representation must be chosen so that the comparable features can be extracted. The comparison function is in turn determined by the source code representation. The complexity of the detector implementation, the bulk of which is the transformation and comparison, depends largely on the code representation as well. A detailed list of the diverse code features used by known code duplication detectors can be found in Appendix A.

2.2.3 Clone Management

The awareness of the existence of duplication in source code requires activities to keep their detrimental effects in check. These activities can be summarized under the term of *clone management* [Gie03]. To manage clones they of course first have to be detected. Apart from ignoring the knowledge thus gained, one has three possibilities to deal with clones: Corrective, preventive and compensatory clone management:

Corrective Clone Management: The goal is to *reengineer* the system by removing the clones. An example would be a refactoring that extracts common code into a separate function.

Most of the approaches in the literature focus on corrective activities, as is exemplified by the stand-alone nature of the tools that are proposed. Correction is usually performed as a singular activity in the context of an overall reengineering project, akin to the addition of a new feature. The entire system (or a selected subsystem) is investigated and as much of the detected duplication as feasible is removed. Other development activities are halted during this operation. We will deal with this predominant management topic in more detail in Section 2.2.4.

Preventive Clone Management: The goal is to *prevent* new clones from being introduced into the system, for example by performing a regular clone check at defined points, *e.g.*, each time new code is entered into the version control system.

Code duplication, however, arises constantly during the life of a system. It therefore seems reasonable to institutionalize its removal. Mayrand et al. [MLH96] have proposed to enhance the software engineering process with preventive clone management activities in two places:

1. **Creation of New Code**
   A check is performed before new source code is entered into the system to reveal similarities with already existing code. If duplication is found, a sound justification is required for incorporating affected functions into the code base.
2. **Maintenance of Old Code**
   The maintenance process is extended by a clone check whenever maintained code is checked into the version control system. In this way, fixes can be made sure to propagate to all instances of the defect code.

Compensatory Clone Management: The goal is to *compensate* for (potential) negative effects of clones that are not removed for one reason or another. A list of duplicated functions, for example, will remind an engineer of all the locations where a change must be applied. Such a list also enables cost estimates of change propagation to all clone instances versus getting rid of the clones by a one-time reengineering effort.

Making the step from only supporting corrective actions towards support for all forms of clone management, clone based reengineering tools strive to integrate with the normal development environment of the programmer. The user is notified about new and old duplication in a similar way to error messages reported after a

recompilation attempt. By implementing incremental algorithms, and triggering them at frequent intervals, *e.g.*, whenever a method or class is added to the system, detection times can be kept small, an important aspect that furthers acceptance of the tools. We are currently aware of one such proposal for an implementation on top of the ECLIPSE development environment [Gie03].

2.2.4 Reengineering Duplication

As we have seen above, duplication adds unnecessary complexity to a system. The goal of reengineering is to get rid of this complexity. If similarities have been detected in a system, this knowledge can be used to aid with *"program renovation, program understanding, error detection, program maintenance, product redocumentation, quality assessment, and extraction of reusable components"* [CIQW+95]. Removal of duplication touches all of these application areas for duplication knowledge, put forth in an early workshop on duplication detection. The goal of the reengineering is to amend the four problem areas explained in Section 2.1.1: reduce work load and resource requirements by decreasing the amount of code, reduce defect probability by introducing clearly defined abstractions, reduce cognitive load by simplifying the code and localizing functionality in a single place.

An important question to ask at this point is how much of the reengineering can be done automatically. The support for duplication reengineering can either be absent, *i.e.*, the reengineer uses only his experience and some handbooks, or it can be partial, *i.e.*, computer *assisted* refactoring, and ideally it would be fully automated.

Assisting Reengineering with Clone Taxonomies

Assisted reengineering consists in guides to the problem areas which can be remedied most effectively, as well as hints and suggestions as to how the concrete clones might be refactored. Taxonomies for duplicated code have the intent to capture common properties of clones. These properties are then used to assess the chances and means of how the clones could be refactored. A category will thus directly suggest applicable refactorings. There are three angles which in the literature have been used to build categories.

Locations of clones: These taxonomies focus on the location differences or the *physical* distance between clone instance locations. Refactoring opportunities or hindrances are derived from the fact that the source fragments are found in the same function, same file, or in files from different directories. In object-oriented systems, clone instances are located at specific places in the class hierarchy. To derive this kind of categorization for a clone pair, only rudimentary parsing technology suffices. Examples are:

- *Kapser&Godfrey* [KG03] define a taxonomy based on the location of the copied code fragments relative to each other (same function, same file, same directory, different directory). For each category of clones they identify probable reasons why these kind of clones are introduced into the system and common problems caused by them, as well as reengineering scenarios to remove the clones from the system.
- *Golomingi* [KN01] investigates object-oriented systems (in SMALLTALK) and bases his categories on the class hierarchy relationships of the methods that contain the copied code fragments. From this information a number of refactoring *scenarios* like *"move to common superclass"* can be triggered. Giesecke proposes similar refactoring scenarios focusing on JAVA [Gie03] .

Differences between clones: Starting from the ideal of perfect clones comprised of two exact copies, these taxonomies measure which syntactic elements have been changed by the programmer after copying. For example, high-similarity clones include methods that are the same except for the name, or methods that are the same but for the types of parameters. This kind of information usually very directly suggests a refactoring. To derive these categorizations for a clone pair extensive information is required which can only be provided by a parser. Examples are:

- *Mayrand et al.* [MLH96] define an ordinal scale of eight clone levels for functions, going from exact copy, as the most obvious form of duplication, to clones which have differing control flow.

Each level is defined as a set of metrics which must have the same value for all the clones in a given category. This simple categorization can only inform about basic refactoring directions, however.

- *Balazinska et al.* [BMLK99] propose a classification scheme for cloned methods with 18 different categories. The categories detail what kind of syntax elements have been changed and also how much of the method has been duplicated.
- *Bellon* [Bel02b] defines three different clone types for the sake of a comparison between different detection tools: exact clones, parameterized clones (renamed variables), and clones that have had more extensive edits. This categorization is aimed at testing the detection and categorization capabilities of different tools.

Context dependencies of clones: From the refactoring perspective it is useful to know how easy the duplicated code can be extracted from its context (to be put in an unifying function, for example). Taxonomies for these kind of differences are based on the uses of variables and methods defined outside of the copied source fragment. The more such dependencies exist, the harder it will be to perform the refactoring. Sophisticated parsing is required to make this kind of analysis. An example is:

- *Balazinska et al.* [BMLK99] propose context analysis to complete the difference analysis of clones for computer-assisted refactoring.

Clone taxonomies can be useful for optimization of detection and reengineering techniques. By knowing the frequencies with which different categories of clones occur in source code, we can concentrate our efforts on the most prominent types or on the types which seem most relevant to the reengineering task at hand.

Automatization of Duplication Reengineering

One of the utopian ideas of duplication reengineering is to refactor duplicated code completely automatic, promising unrivaled gains in productivity. There are technical and social obstacles which make this problem hard to solve.

On the technical side, fully automating refactoring is difficult and not easily reconciled with the human maintenance effort. Except for exact copies and some standard cases, full automation can only be expected from computers which do their programming themselves, *i.e.*, from "intelligent" machines. Current attempts at complete automation have their own problems. Balazinska et al. [BMD+99] have automated refactoring for certain types of duplication. Their prototype introduces design patterns like *Strategy* and *Template*. Since for each clone two new classes and a couple of interfaces carrying artificially created names have to be introduced, the generated code is hard to read and maintain by a human developer.

The social barrier to the adoption of automated techniques is found in the developers who are in general averse to machines telling them how to program [Gie03]. In the experience of Cordy, programmers feel threatened by an automated technique that seems to make their jobs superfluous. To counter these fears, which hinder the adoption of automated program transformation tools, one must declare the results as *advice* and leave the last decision in the hand of the maintainer [Cor03]. The acceptance problem, too, will only go away once developers have all been replaced by the auto-programmed machine.

In summary, since the problems of automatic duplication reengineering will only be solved completely by the self-programming computer, a concept still far from realization, an emphasis on the human in the loop is justified.

2.3 A Categorization of Detection Approaches

This section categorizes clone detection approaches using four characteristics or *goals*. These describe how an approach fares with regard to important duplication and reengineering concerns. As we have stated in Section 2.2.2, the determining characteristics of a clone detector are the representation of the source code and the properties used for the comparison. We could thus describe any approach concisely by only mentioning the source code representation and the properties it uses for its comparison. The categorization presented here, on the other hand, indicates the consequences of the choices of code representation and compared properties in the context of overarching reengineering concerns. These goals help us to orient ourselves in the selection of an approach to pursue.

The four goals *Adaptability*, *Scalability*, *Duplication Sensitivity*, and *Reengineering Support* are motivated in this section. In the next section we will select the ones that we emphasize in our thesis.

Adaptability: The current landscape of programming languages for use in commercial systems is wide and varied and growing. On the one side, new programming languages are invented continously, and on the other side systems of considerable importance have been proven to be long lasting—thanks to the large investments and the aggregated business knowledge they represent—and therefore the languages they were written in as well. Older programming languages will be staying with us for years to come. If we want to build detection method which can be used by any engineer as part of his daily work, we need to be prepared for all the possible programming languages that can be found.

Scalability: Software systems are becoming larger and larger. The more has been invested in them, the more they become valuable and the likelier it is that they must be maintained and adapted to new requirements. Clone detection as part of a reengineering effort is confronted with source text going into the millions of lines of code. A scalable detection approach must be able to process all this code in a reasonable amount of time. If we use code detection in a dedicated reengineering effort the time needed for detection will always be dwarfed by the time necessary to analyze the results. If clone detection is integrated into the development process, however, then speed is important for the developer who does not want to be delayed by another bump in the edit-compile-run cycle.

Duplication Sensitivity: Duplication exists on a continuum from exactly copied fragments to pieces implementing the same functionality in a syntactically different way. Whereas the exact copies are easy to find there are many cases with more subtle similarities that are not easy to detect. Subtle duplication has however the same potential to hinder system evolution as the obvious copies. The potential of errors incurred because of duplication are even higher when the copying is not obvious and therefore overlooked more easily. Subtle duplication, when it is the result if many evolutionary changes that two copied fragments have undergone, can also be the last remnant of an original design decision, and thereby help to understand the program, but it must be found!

To detect subtle duplication both the comparison and the filtering phase must perform a deep analysis which requires a very detailed code representation. The drawback of methods that perform a fine-grained analysis is that their implementation is closely tied to a given language and its semantics, and that these tools are therefore "typically complicated to build, computationally slow and require enormous machine resources to apply to an entire program" (Atkinson&Griswold, qtd. in [Gri98])

Reengineering Support: After the detector has listed the suspected clones it must be decided what is to be done with them. It is first necessary to analyze which clones can and should probably be removed from the system. It is then necessary to determine the means by which this refactoring can be achieved. Since clones can be arbitrary source fragments, they can be embedded in any context. The various types of differences between clone instances offer different constraints for the refactorings. Human analysis of context dependencies and differences can be time-consuming and error prone. The usefulness of automatically gathered information increases with the number of clones retrieved from a system.

Goal Selection

The goals presented above are not orthogonal. Some goals can mutually benefit each other: The more we can automate refactoring support, for example, the more clone data can be handled by a reengineer, which improves scalability. Also, if an approach invests much in duplication sensitivity, the information gathered in this deep analysis can be used to improve the support of reengineering actions. Some goals are also mutually incompatible. For example, the more fine-grained the detection approach is the more time it will usually require. This conflicts directly with the scalability goal. It is also true that the more fine-grained an analysis is the more language dependent the algorithms will become, which hurts adaptability. To solve conflicts with the important scalability goal some approaches implement more than one pass: using a scalable approach in the first pass to narrow the search, and then using deep analysis on the found candidates in a second pass. This multi-method technique however tends to be detrimental to adaptability.

Due to the conflicting dependencies among the reengineering goals, we have to choose the goals which should be emphasized and, conversely, the ones which should be neglected. To make this choice we can take the stance of one of the following viewpoints:

- A dedicated tool builder who wants to sell a standalone commercial tool and needs competitive advantages.
- A tool builder who wants to integrate duplication detection into the tool suite of an existing development environment.
- A consulting company who sells reengineering services, among them the detection and reengineering of duplication.
- A developer who, not having much previous experience in reengineering, is suddenly charged with such a project.

In our case we take the pragmatic viewpoint of the developer who suddenly needs to make an investigation into the duplication situation of a project. This viewpoint can be made more explicit with a few slogans:

Lightweight. We want to develop methods that do not need a large up-front investment in source code extraction techniques needed for deep analysis. In the best case, the implementation of the detector should be doable in preparation of a reengineering project.

Developer in Charge. We would like to provide technology that can be easily learned, used and even adapted by any programmer as part of his normal duties.

Good Enough. We are satisfied with finding 80% of the duplication (expending 20% of the effort).

Given these, a proof-of-concept would be the implementation of a small tool in the tradition of the UNIX command line tools like GREP. A simple configuration interface should enable the use of the tool on a variety of source texts.

Going back to the reengineering goals, we can see that emphasizing the *Adaptability* and *Scalability* goals over *Duplication Sensitivity* and *Reengineering Support* will adhere most to the principle of *Simplicity* that is stressed by our viewpoint. There are no immediate conflicts between scalability and adaptability. They could only prove to be desynchronized if it was shown that a scalable detection method would be dependent on the programming language or paradigm itself, something which to our knowledge has not surfaced yet.

2.4 Conclusions

In this chapter we have given an overview of the problem of code duplication, its causes and its consequences. We have presented the basic stages of the duplication detection process, as well as proposals for the extension of the software development process to account for the existence of clones and the need to manage them. We have then described four high-level reengineering goals which help detector builders to orient themselves in their endeavors. By taking the stance of an everyday user of clone detection we have finally selected the reengineering goals which we will emphasize in the investigation that follows:

- With the *Adaptability* goal we want to emphasize that a detector should be configurable and made to understand programs in different languages with a minimum of effort.
- With the *Scalability* goal we want to strive for detectors capable of processing systems of large sizes using a reasonable amount of resources.

In Chapter 4 we will discuss how the selection of these goals leads us to choose strings as code representation and string matching as comparison method.

In the subsequent Chapter 3 we will focus on the clone pair itself. We explain how we can define the clone relation between source fragments. Using the notions thus defined we will explain in more detail the stages of the clone detection process introduced in this chapter.

Chapter 3

A Conceptual Model for Clones

In this section we want to establish the vocabulary for talking about code duplication detection. We present a conceptual model of duplicated code and clones which we use to describe all currently existing approaches to clone detection that we are aware of. The model establishes a framework of terminologies and issues that are necessary for the discussion. We first describe in a coarse overview the context in which clones are to be defined. In a next step, we present a detailed list of notions which model the field and describe the process of clone detection.

Duplication in software does not only occur on the level of the source code. Duplication exists also in requirements, design documents or user documentation. In this work we will consider out of scope all forms of duplication other than source code. The reason for this choice is that the formats in which the other documents are written differ significantly from the formal language which is source code. Be it natural language used in a requirements document or the graphical language of a UML design, the automatic interpretation of these documents is so much different in each case that we cannot hope to use insights from one field for the others.

Our Contribution

The contributions of this chapter are:

- A discussion of important aspects of the duplication detection, *e.g.*, clone relevance, and influence of source code structure.
- The presentation of a model of the comparison process which encompasses all known detection approaches.
- A discussion of *fixed* and *free* clone granularities as important distinction criterion for clone detectors.
- A proposal to aggregate free granularity clones in a hierarchical manner to be able to handle their multitude.
- The definition of *Clone Class Families* as aggregation of source fragments according to the similarities in both code *and* locations linked through copying.

3.1 Preliminary Notions of a Definition for Clones

Before getting into the grey areas of the nuanced description of clones, we circumscribe the phenomenon of clones by looking at two definitions, one very abstract and the other very concrete.

We start with the idealistic definition, a cousin of the OAOO principle mentioned in Section 2.1:

> *The* DRY *Principle (Don't repeat yourself)*
>
> Every piece of knowledge must have a single, unambiguous, authorative representation within a system. [HT00]

The DRY principle implies that whenever the same idea is expressed twice we have an example of unwanted redundancy. This extends to requirements, design, source, tests, and user documentation. If we constrain ourselves to the knowledge which is represented in the source code of a system, we can say that whenever a source fragment f_1 implements the same *logic* as a source fragment f_2, the pair (f_1, f_2) forms a clone, or the relation *clone*(f_1, f_2) holds. It also follows that the clone property of (f_1, f_2) does not depend on the concrete implementation of the logic at all: As long as the semantics of the code are the same, f_1 can be implemented in ASSEMBLER and f_2 in SQL but they still represent the same knowledge and thus form a clone.

If we want to actually find and process source code redundancy, the definition just sketched out is impractical. For one, due to the fundamental halting problem it is not possible to algorithmically determine the semantic equality of two implementations. This means that to be useful, our definition should not be based on formal semantic equality, but rather should be based on the concrete representation of the algorithm, *i.e.*, the source text of the implementation. Semantic aspects of the code such as control or data flow can be integrated into the analysis in small doses for an approximation of the test for semantic equality. These extensions will always result in increased detection costs.

The other definition of code duplication originates from the action that brings most of the clones into existence. We call this:

> *The Operational Definition of Code Duplication*
>
> A clone is created when a piece of source code is copied from one location using the copy function of an editor and then inserted at another location. Optionally it can be transformed by a variety of editing operations in order for the code to function properly in the new context.

Relying exclusively on this definition would however unreasonably prohibit to be regarded as clones the independently created source fragments which have—accidentally or just unconsciously like the *mental macros*—the same syntactic and semantic structures. Also, since tracing the *copy & paste* actions of a programmer is not standard practice, finding clones with this definition is not feasible. We therefore conclude that a clone must be defined based *only* on the implementation artefact: the source text.

With these two failed definition attempts and the realization that we have to extract (parts of) documents from source texts, we try to look to the neighboring field of information retrieval for help.

3.1.1 Clone Detection and Information Retrieval

The field of *information retrieval* has close ties to the field of clone detection. The exercise of comparing the two fields is worthwhile because information retrieval has been existing as a research field for a number of decades and much effort has been invested in developing terminology and models for its study area. The considerably younger field of clone detection is likely to profit from the insights and experiences of the older information retrieval [WL03].

"Information retrieval deals with the representation, storage, organization of, and access to information items" [BYG99]. Although this broad description does not immediately match the clone detection task, the central aspect of "finding information pertaining to a query" is very similar to clone detection. In information retrieval (see the schema in Figure 3.1) a set of transformations creates a *logical view* [BYG99] of a document. The logical view of the document should represent what the document is "about". Transformations remove superfluous elements which for natural language texts are articles and connectives and grammatical 'embellishments' like conjugations and declinations. These transformations, apart from reducing the size of the database, prepare the

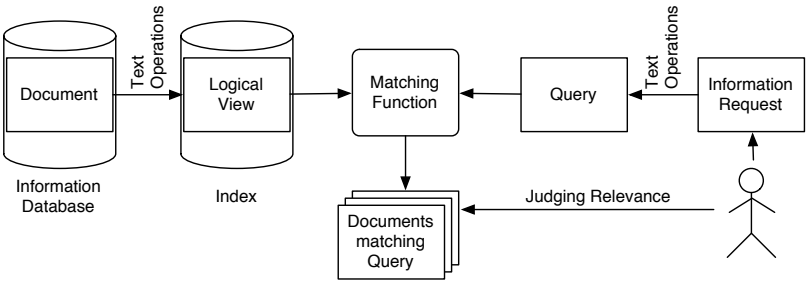


Figure 3.1: A schematic of the Information Retrieval Process (adapted from [Ing92]).

documents to be compared effectively with a query. The central element of the information retrieval process is the matching function which compares the logical view of the query with the entities in the index. In the same manner clone detection also transforms source code into a representation that is amenable for comparison with a query.

The decisive difference between the two fields is however that the queries in information retrieval are formulated by a human who requires the information. In clone detection, the query must be derived from the information database itself. The advantage is that clone detection does not have to deal with the problem of how to transform the human information need into a query that can be matched with the indexed documents. The disadvantage is that we have to find the query from within the index itself. This causes the "compare everything with everything" situation which drives up the computational complexity of the clone detection techniques. Another difference is that information retrieval has to deal with text collections and digital libraries of sizes that are currently measured in gigabytes and will grow to terabytes in the future. The indexing capabilities developed for information retrieval are thus also important to reduce the time to search through the collection. Their costs are amorticised because a one-time index creation serves a large number of queries. In clone detection, however, the systems which are searched today are only occasionally in the megabyte range. In a reengineering project, the system is searched once for the detection. The extra storage of an elaborate index will be justified only where clone detection is integrated into the software development process and is scheduled regularly.

3.1.2 Relevance of Retrieved Documents

The notion of *relevance* is central to evaluate the effectiveness of information retrieval techniques: "The purpose of an automatic retrieval strategy is to retrieve all the *relevant* documents and at the same time retrieving as few of the *non-relevant* as possible" [vR79]. Relevance measures how well the information contained in a retrieved document meets the demand for information that a user has expressed in the query. Relevance describes the user acceptance or rejection of a proposed relationship between a document and a query. More complete, relevance is

> the criterion used to quantify the phenomenon involved when individuals (users) judge the relationship, utility, importance, degree of match, fit, proximity, appropriateness, closeness, pertinence, value or bearing of documents or document representations to an information requirement, need, question, statement, description of research, treatment, etc. (A.M. Rees, 1966, qtd. in [Gre00])

Relevance is thus dependent on the judgment of the human user, "linked to the individual user's problem space and state of knowledge" [Ing92].

The *problem space* of the user of a clone detector is what we describe in Section 2.1.1. He must evaluate the documents reported by a clone detection mechanism from this position. Relevance in code duplication detection, just as in information retrieval, is thus dependent on the contextual information available to the user.

A clone detection user's context consists in many pieces of knowledge and many constraints: the exact tasks he has to perform, functional and non-functional requirements of the system, the architecture of the system, the current understanding he has of the system, the programming language, his proficiency in programming, administrative conditions, etc. It is impossible to program a clone detector to respect all of this potential knowledge in the detection process, it may sometimes even be impossible to formulate the knowledge as an automatic filter. In the end, judging the relevance of a clone retrieved by a clone detection mechanism is an essential part of clone detection which we cannot hope to fully automate.

3.1.3 Clone Detection Motivations which Affect Relevance

If we want to find relevancy criteria for clones, we can take orientation from the goals of an engineer who looks for duplicated code. The user of the clone detection system may be motivated by any of a number of different software maintenance tasks. These tasks influence the evaluation of the clones that are reported by the clone detection system (clones are said to be *task relevant* by Walenstein et al. [WJL+03]). We provide a short list of tasks accompanied by a description of issues that might influence the evaluation of a clone:

Reverse Engineering and Program Understanding: In program understanding we are interested in redundant code that does not lend itself directly to refactoring but instead helps to understand how different parts of the source code hang together thematically. Relevance in this context might mean anything that reveals similar structures on all levels of the code. The granularity of these repeated artifacts can be as small as part of an identifier name. From this point of view, the granularity and syntactic validity constraints that are needed for the reengineering task could be relaxed to a great extent.

Reengineering and Perfective Maintenance: In reengineering, we want to actively improve the quality of a system. Reengineering activities will be controlled by a cost-benefit analysis. Relevance in this context might mean clones of "refactorable" logic, that is, the duplication can be extracted, unified and put in a form that is accessible from its original locations via macro invocation, subroutine call or other control flow mechanisms. The effort to perform the duplication analysis and the subsequent refactoring must be justified by the gain in code quality. This might entail that the code fragments reported by the clone detection system must be syntactically valid constructs, *i.e.*, entire functions or blocks, and that the system is able to automatically propose a straightforward refactoring (see Rysselberghe&Demeyer [vRD04] for an evaluation how well the results of different clone detectors can be used for refactoring).

3.1.4 Sample Relevance Discussion

To illustrate the type of relevance questions a clone evaluator is confronted with, we discuss an example clone.

Figure 3.2 shows two functions from a medium sized system written in C, the open source MAKE replacement called COOK. The functions append a new element to a list. Directly from this code and from similar reports of the clone detector we can get the following bits of information that may affect our decision to refactor the code or not:

- The memory management idiom is found 34 times in 26 different files (10%) of the COOK system (8 files contain it twice). The files come from all subsystems of COOK. 9 of the files carry the suffix *list* in their name.
- The core idiom of all instances is 5 lines long (lines 5–9). It is used to dynamically manage memory for data structures like lists, stacks, and caches.
- A refactoring into a function requires about 5 parameters. See Figure 3.3 for a proposal.

Table 3.1 lists a number of arguments that speak for and against the relevance of the copied fragment, in this case the decision to perform a refactoring.

```
void
lex_filename_list_push_back(this, p)
  lex_filename_list_ty *this;
  lex_filename_ty      *p;    {
 if (this->length >= this->maximum) {
   size_t nbytes;
   this->maximum = this->maximum * 2 + 4;
   nbytes = this->maximum * sizeof(this->list[0]);
   this->list = mem_change_size(this->list, nbytes); }
 this->list[this->length++] = p; }
```

```
void
blob_list_append(lllp, llp)
  blob_list_ty  *lllp;
  blob_ty       *llp;  {
 if (lllp->length >= lllp->maximum) {
   size_t nbytes;
   lllp->maximum = lllp->maximum * 2 + 8;
   nbytes = lllp->maximum * sizeof(blob_ty *);
   lllp->list = mem_change_size(lllp->list, nbytes); }
 lllp->list[lllp->length++] = llp; }
```

Figure 3.2: Two fragments of C code (from the COOK system) performing memory management

```
void extend_if_necessary(len,max,addendum,chunksize,list)
     int len;
     int *max;
     int addendum; /* make this a constant? */
     int chunksize;
     void *list;
 {
  if(len >= *max) {
    size_t nbytes;
    *max = *max * 2 + addendum;
    nbytes = *max * chunksize;
    list = mem_change_size(list, nbytes); }
 }
```

Figure 3.3: A refactoring of the code from Figure 3.2.

For Refactoring	*Against* Refactoring
The idiom has only small variations over all the encountered instances and can be refactored easily.	Since the C programming language does not offer generic types we would have to realize the shared implementation using **`void`** pointers. That would forfeit type safety.
Memory management is a basic system functionality which should not interfere with the domain specific algorithms (separation of concerns).	All the different parts of COOK would be dependent on a single memory management implementation.
The purpose of the idiom is very simple. It is thus not likely to evolve much. We could therefore refactor it without fear of many change requests.	If the requirements for the memory management functionality *do* evolve differently in the 34 locations, a change to the central implementation requires that all uses must be retested.
In some of the files which contain the idiom twice, a local refactoring would not disturb anything outside of the file.	A function invocation involving 5 parameters may be a bit unwieldy to read. Remembering the purpose of every parameter could prove to be harder than reading the inlined code, which as an idiom is instantly recognized.

Table 3.1: Arguments for and against the refactoring of the clones in Figure 3.2.

3.2 Definitions of Clones

The aim of this section is to develop a framework of notions with which we can describe automatic clone detection approaches. The main goal is to establish the properties of a pragmatic clone relation between source fragments. From the discussion in the previous section we recapitulate the two principles of a clone model: *i)* We derive clone-ship from similarities in the source text, and *ii)* we need human expertise to be the final arbiter on relevance. A clone definition must have the following characteristics:

Constructive: The definition must be concrete, based on the source text or any derivative thereof. We must be able to build a mechanical retrieval engine from the definition.

Include Relevance Criteria: Relevance of clones cannot fully be made part of the core definition since it depends on the various tasks of the engineer. Some but not all criteria from a task dependent list can be automatized. Many will however have to be left for the "human in the loop" to decide.

The desirable properties of a clone definition are [Gie03]:

- **Independent of a Programming Language.**

  We can express the same logic in all Turing complete programming languages. The form in which a program is written is thus accidental. We want to find duplication of logic, the essential property of a program.

- **Independent of a Detection Approach.**

  Clones exist independent of the fact that they are detected by some mechanism. A human programmer knows when two fragments form a clone. Ideally, we would like to replicate in algorithmic form this detection capability of the human arbiter.

- **Describe a Continuum of Clones from Exact to Non-Exact.**

  Copied code is changed to many different degrees, from no change to continued development that morphs the fragment into something different. But even for fragments where the common origin is almost unrecognizable, similarity knowledge is still valuable for a range of maintenance tasks.

We first give an overview of clone definitions that have been proposed in the literature. We then describe the framework in which clones occur, are detected and grouped. This is divided into four parts: *i)* The description of the source code and its properties which influence the clone detection process, *ii)* the description how the code of a system is partitioned into the source fragments which are ultimately compared, *iii)* the description of the clone relationship between two source fragments, and *iv)* the aggregation of clone pairs into larger clone sets, which describe similarity relationships between multiple source fragments.

Clone Definitions in the Literature

A commonly agreed upon definition for clones has not yet been established in the literature [LLWY03]. In the beginning of the research on clones the operational idea of cloning, the fact that programmers use *copy & paste* to reuse their code, was the direct inspiration for the work. Early attempts like Johnson's [Joh94a] are based on this view, implicitly defining a clone as an identical fragment of source text. More detailed models of how code duplication occurs are used by Baker [Bak93b], for example. She assumes that the code is copied and subsequently adapted and changed by various editing operations. This leads to a more refined detection approach. The definition of a clone was, however, still implicit: a clone is what can be detected by the particular approach. The definition "strategy" of letting the mechanism decide what is a clone is used by most proposals, circumscribing clones by more or less vague terms like 'identical' or 'similar' without specifying the meaning clearly.

A detection-independent definition was first given by Kontogiannis [Kon97]. He defines four basic types of clones, still based on the operational idea of duplication: *i)* exact clones where an identity function on each non-blank character maps fragment f_1 to fragment f_2, *ii)* clones that are exact except for systematically substituted variable names and data types, *iii)* clones where expression and statements have been modified, and *iv)* clones where statements and expressions have been either deleted or inserted. The three modification functions can be combined to get multi-type modifications on a clone.

In a first attempt to define clones outside of the actual research on detection, Giesecke, focusing on clone *management* issues, has proposed a formalization of the entire detection process, from the selection of the source fragments to be compared until the representation of the sets of reported clones [Gie03]. In his framework, exact and non-exact clones are described independent of the detection approach.

The idea of *relevance* of clones has been used implicitly by many of the approaches which focus on clones that can be easily refactored [BMLK99][KH02]. These are attempts at capturing (in automated filters) some of the relevance criteria. An explicit formulation of relevance criteria in the form of a handbook for human clone evaluators has been written by Walenstein et al. [WJL+03], who do not propose a detection mechanism of their own.

3.3 Important Source Code Aspects

Most clones must be presented to the user for scrutiny. It is therefore advantageous for them to keep the form and layout from the original source file. This section explains the properties of the source code which a clone detection mechanism must take care of. For languages like C/C++ we address the problem of code which includes preprocessor commands in Section 3.3.1. The structural properties of source code, which are compared by clone detectors is described in Section 3.3.2.

3.3.1 Preprocessing Source Code

The C/C++ programmer has the advantage[1] of being able to control compilation with the facilities of the preprocessor. The use of the preprocessor is associated with a number of problems which have to do with the macro expansion and conditional compilation.

[1] For the developers of reverse engineering tools it is often a disadvantage.

Macro Expansion: This poses two problems. First, it is difficult to map line numbers in the code after the expansion back to the original code. Using the post-expansion code as reference goes against the stated goal for clone reporting which is to refer to the source code in the form the programmer sees it in his editor. The second problem is that expanded macros generate duplication. If this is reported it generates confusion among the programmers who use macros to *avoid* duplication. As a reaction to these problems, many detection tools choose not to expand macros and to treat them as normal code where possible.

Conditional Compilation: The conditional compilation directives (`#if`, `#ifdef`, `#ifndef`, `#else`) enable code for different deployments, execution modes, or platforms to coexist in the same source file. At compilation time, flags select the code fragments that are fed to the compiler. The problems are, first, that a reengineer is interested in potential duplication in all of the mutually exclusive branches of the `#if` directives. This problem is solved by Baxter et al. [BYM+98] by treating the preprocessor directives `#if`, `#else`, and `#endif` as elements of the C grammar. In this way all branches of a conditional compilation can be represented in the abstract syntax tree. Merlo et al. [MAK] use a heuristic derived from their observation that the `#else`-branches are used rarely and thus contain much less code (in the LINUX kernel). As a consequence they parse the code in the `then`-branches only.

The more serious problem of the `#if` directives is however that they can appear anywhere in the code, *i.e.*, in between any two tokens of C code. In practice that means that the individual branches often contain incomplete source text which results in legal syntax only after having been preprocessed. A reverse engineering parser trying to understand both branches at the same time will fail on the illegal fragments. Parsers have to be made robust for these cases [Meh04].

Tools (like string matching-based tools) that are insensitive to syntactic invalidity do not understand the difference between preprocessor directives and normal code. They compare every piece of source text regardless.

Since the preprocessor issues are not at the core of the duplication detection problem and are constrained to systems written in C/C++ or COBOL, we will not discuss the handling of these problems in more depth here. We will leave it to the individual approaches to find solutions as to which part of the code is going to be taking part in the comparison.

3.3.2 Source Code Structure

The structure of source code is expressed on multiple levels. On the lowest level is the textual order of the tokens and statements. On a higher level we can build a control flow graph representing the syntactic order. This graph can also represent jumps in the code. To abstract even more from the non-essential ordering of the source text, we can include the data flow structure to form program dependence graphs.

Since structure is an essential aspect of the code it is so for clone detection too. Any comparison therefore must account for structural similarity somewhere. The string- or token-based approaches use the token- or line-order to aggregate the small matching source fragments into clones. Approaches that work on the abstract syntax trees use the syntactic order to combine matching nodes and subgraphs in close vicinity. Metrics-based approaches do not use order to aggregate comparison results. Rather they measure the structural properties and compare these values. Goodnow et al. [GIHK+97] measure the *statement order* of the code by counting adjacent accesses to pairs of identifiers. On the *control flow* level, the McCabe complexity metric [Kon97], or cyclomatic complexity metric [MAK] are used among others. Finally, data flow dependencies are exploited by Krinke [Kri01] and Komondoor [KH01b] with slicing based methods.

With the orders just mentioned we can order the statements of a function naturally. It is however not as trivial to define an order that goes beyond a single function, *i.e.*, that totally orders all statements of the entire program. The sequence in which the functions appear in a source file is mostly incidental and will only provide a partial order. The call graph of a program can be used to order the functions, but since there are a number of interpretations of what a call graph is [Mur96], some heuristic decisions are needed as well. Only few approaches exploit a call graph order. Jankowitz [Jan88] and Leitão [Lei03] guide a coarse comparison phase by the structure of the call graph.

To include the structure of entire programs into the comparison is mainly interesting for plagiarism detection because overall structural similarity can be a strong hint at large scale copying. In clone detection, rather than including the whole program in the computationally expensive comparison process, comparison usually stops at a level where, for example, refactoring opportunities of detected clones are still easy to decide. Similarities between larger entities than functions can be determined in a later phase by combining clones.

3.4 Source Code Partitioning

At the beginning of the clone detection process, the source code of the system is partitioned. The partitioning determines the domain of the comparison: Which pieces of the source are to be compared against each other?

The goal of code partitioning is threefold: *i)* Determine exactly which code is going to be compared, *i.e.*, remove the uninteresting parts, and *ii)* split the code into *source units*, *i.e.*, the entities which are related with the clone relation. Eventually, step *iii)* splits the source units further into *comparison units* which are fed to the comparison function(s).

3.4.1 Selection of Source Code

We may want to remove source code before the comparison for a number of reasons:

- A preprocessor can be run over C/C++ code to remove parts of the conditional compilation branches if the comparison cannot handle it (see the discussion on page 31).
- For embedded code (*e.g.*, SQL embedded in COBOL code, or ASSEMBLER inlined in C code) source partitioning is needed to separate the code for the different languages since they may have to be compared with different tools or tool-settings.
- Parts of the source code that are not under the direct control of the programmer such as generated code (LEX- and YACC-generated files, for example) can be excluded.
- Pieces of code which are likely to produce a lot of false positives, *e.g.*, table initializations, can be excluded.
- For very large case studies where either the time for comparison, or the time for working through the reported clones cannot be allocated, the parts deemed unproblematic in terms of duplication can be removed from the study.

3.4.2 Source Units

A system's source code that has not been removed in the previous step is partitioned into a set of disjoint fragments called *source units*. The selection of source units is motivated in the following way:

- We want to establish the border beyond which the order of the source code is not taken into account any more by the comparison.
- With a small amount of parsing we can assign types to the source units (in C, for example, we could have *function body* units, and *formal parameter definition* units). We can then remove uninteresting comparisons between incongruent types from the task list.

The most important property of a source unit is this: It is the largest source fragment that can be involved in a direct clone relation with another fragment. Since there is not necessarily an order relation defined on the

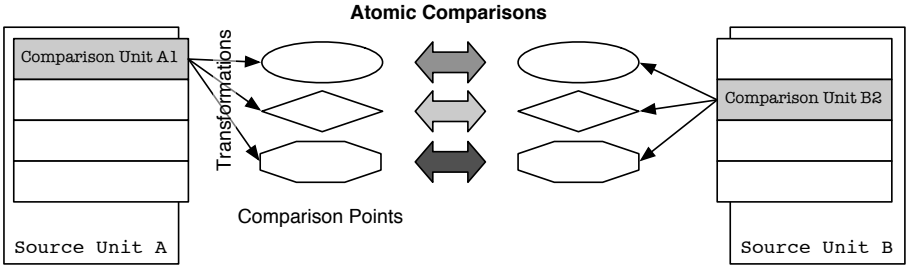


Figure 3.4: Two source units are compared on hetero-genic comparison points.

set of source units we cannot aggregate matching fragments beyond the border of a source unit. To find duplication relationships between entities larger than the chosen source unit granularity, we can apply combination functions on the set of clones (see Section 3.7) in a post-comparison phase.

Source units can be defined at many granularity levels; examples are files, classes, functions, blocks, statements, or sequences of source lines. If clone relations are established between source units only (what we define as *fixed* clone granularity in Section 3.6.1) the choice of source unit granularity is subject to a number of forces:

- The smaller the granularity becomes, the larger the cardinality of the result set will be.
- The smaller the granularity the less specificity can be expressed in the code of a source unit. This may result in many false positives.
- If the granularity too big, meaningful duplication on smaller granularity levels may be missed.

3.4.3 Comparison Units

The source unit, even though it is the bearer of the clone relation, must be subdivided further if the comparison function is to work on fragments of a different (*i.e.*, smaller) granularity level: Source units are then split into disjoint *comparison units*. This can either be done uniformly, for example by splitting code into lines or into tokens. Comparison units can also be derived from the syntactic structure of the source unit: An **`if`**-statement can be broken into `conditional` expression, `then`- and **`else`**-blocks. If the comparison units are typed in this manner, the comparison functions can be selected depending on the type. Comparison units are ordered within their containing source units, in contrast to the unordered set of source units. The order of comparison units is taken into account by the comparison function.

In some cases the source unit itself is the comparison unit. Metrics, for example can be computed for source fragments of any granularity [MAK] and a subdivision into comparison units is thus unnecessary.

3.5 Comparing the Code

The actual comparison can be described in three steps: *i)* transform the code of the comparison unit into a *logical* representation, *ii)* compare the transformed code, and *iii)* use the results to form clone pairs. In this section we are describing different implementations of these steps. The process is illustrated in Figure 3.4.

1. The comparison unit is transformed into one or more *comparison points*. A comparison point is a transformation or a property extracted from the source code.

   This transformation can for example simply mean to extract the name of a function [MLM96b], or the removal of comments and white space [Bak92], or a parsing pass which builds an abstract syntax tree [BYM+98] or a program dependence graph [Kri01][KH01a]. Metrics-based approaches usually

compute an attribute vector for each comparison unit. The combination of all comparison points can be seen as a signature for the comparison unit.

2. Once the comparison point attributes are computed the comparison takes place. If all attributes are of the same type, we can measure the similarity between the attribute vectors with a function like the Euclidian distance metric [Kon97] or the clustering abilities of a neural network [DBF+95]. If the comparison points are heterogeneous, a different comparison function is applied to each comparison point attribute [MLM96b] (this case is illustrated in Figure 3.4).

   The comparison functions themselves can be as simple as string matching [Joh93], or as complicated as finding isomorphic subgraphs of a program dependence graph [Kri01]. The comparison function may also vary with respect to the type of the comparison unit. Leitão [Lei03], for example, uses syntax-aware comparison, *i.e.*, each special form in LISP has its own comparison function.

3. Using the order of the comparison units, similarities of adjacent units are summed up. For fixed granularity clones all comparison units that belong to a source unit are aggregated. For free granularity clones, aggregation is continued as long as the aggregated sum is above a given threshold for the number of aggregated comparison units, making sure that the aggregation continues until the largest possible group of comparison units is found.

At the end of the comparison process we have declared a number of source units or aggregated comparison units to be in clone relations.

3.6 The Clone Pair

Although similarity relations in source code are in many cases more complicated than only two fragments which are copies of each other, we start off with the description of a clone pair for reasons of simplicity. We first explain the difference between free and fixed granularity clones, then list the properties of the clone relation, and finally discuss one important property: the size of a clone.

3.6.1 Clone Granularity

An important dichotomy governs the field of clone detection: The clone relation is either established among a set of preselected source fragments which are all of the same granularity (source units), or clones are combined (aggregated) from small fragments (comparison units) that have been found similar during the comparison phase. In other words: Either the clone granularity is *fixed* before the comparison or it is *free*, determined ad hoc from atomic comparison results (see Figure 3.5 for an illustration). We will explain these two choices with their advantages and disadvantages.

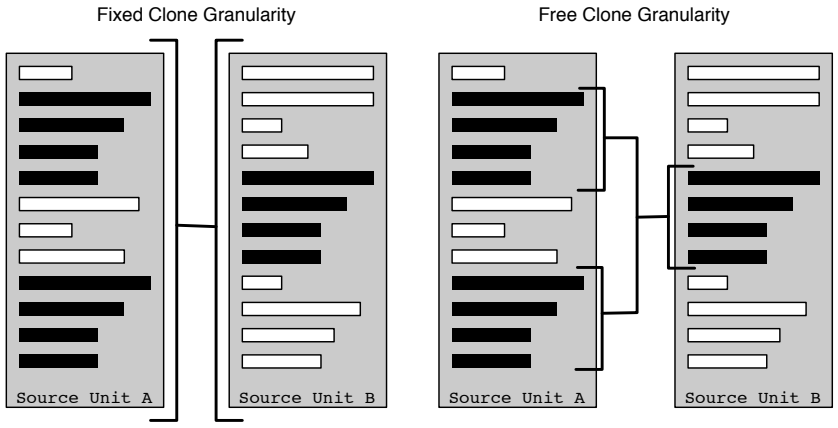


Figure 3.5: Clone relations in fixed and free clone granularities. The black bars represent lines of matching source code.

Fixed Clone Granularity: In this case, the source code of a system is partitioned into disjoint *source units*. These units are used as defining elements for the clone relation, *i.e.*, only pairs of source units are considered as clones. All similarities of smaller granularity are disregarded.

Advantages	*Disadvantages*
By selecting only valid syntactic fragments as source units, the detected duplication is likely to be more easily refactored.	If the source unit granularity is selected to be large, meaningful self-similarity *within* the source units may be missed.
The number of potential clones is known. In a system with n source units, there can be no more than $\frac{n(n-1)}{2}$ clones.	The partitioning of the source code may disrupt meaningful duplication that goes beyond a single source unit.
Relations within the set of clones are simple as clones cannot overlap with other clones.	When two source units match only partially, a manual investigation must determine the exact location of the similarity.

Free Clone Granularity: The source code of a system is partitioned into atomic *comparison units*. A clone is then formed *ad hoc*, derived from the result of the comparison by the aggregation of adjacent comparison units exhibiting high similarity.

Advantages	*Disadvantages*
We can detect clones at the smallest granularity level that the comparison supports.	Clones may extend over syntactic borders, *e.g.*, blocks. This may make some of them difficult to refactor [vRD04]. To avoid this problem, an additional filter pass is necessary [HUK+02].
We are able to detect self-similarity within source units.	The number of potential clones is much higher than for fixed clone granularity.
	The relationships between clones are more complicated: Clone pairs may overlap or be contained within each other. This cannot happen when the clone granularity is fixed.

The main advantage of the similarity induced aggregation is that duplication is found and reported on all granularity levels simultaneously. We are able to find the clone relations between shared fragments of 20 lines within two 200 lines functions. With a fixed function granularity this would be noted only as a small 10% similarity and probably fall under a threshold. Most importantly however, if granularity is free we are able to detect self similarity of source units. A function which contains 3 times the same 10 lines of code will not be noticed if we only perform inter-function comparisons.

3.6.2 The Clone Relation Properties

In this section we investigate more formally the notion of the *clone relation*. We define the clone relation to be a binary relation between source fragments, *clone*(f_1, f_2). For fixed granularity, *clone*(f_1, f_2) is defined on $\mathcal{P}(\mathit{SrcUnits})$, and for free granularity *clone*(f_1, f_2) is defined on arbitrary sequences of comparison units $\mathcal{S}(\mathit{CompUnits})$.

A useful notion of clone relation fulfills the requirements of reflexivity, symmetry but not necessarily transitivity.

Reflexivity: A clone relation is reflexive if *clone*(f, f) always holds, *i.e.*, if every source unit is always a clone of itself. This is obviously true, as any (syntactically valid) source unit performs the same task as itself which makes it a clone under the strict criterion of semantic equality (syntactically invalid fragments can be extended with the same text until valid). Reflexivity can thus be naturally assumed as property of a clone relation.

Symmetry: A clone relation is symmetric if *clone*$(f_1, f_2) \Leftrightarrow$ *clone*(f_2, f_1). This is not trivially true for duplicated code. If a clone instance is created via *copy & paste*, a direction from the original to the copy is implied. This information could be the basis for an asymmetric clone relation (which would be interesting in the context of evolution analysis). Since in this work we use structural similarity and not the creation history to identify clones, we have no information about any order between the members of a clone pair. Hence, symmetry can be claimed as a useful property of a clone relation.

Transitivity: For exact clones transitivity holds trivially. For non-exact clones where a single similarity value abstracts over matching and non-matching parts, or aspects, of two source fragments, it is easy to construct an intransitive counter-example: Let source fragment f_1 match the first half of source fragment f_2 and source fragment f_3 match the second half of f_2. The pairs *clone*(f_1, f_2) and *clone*(f_2, f_3) hold under an adequately relaxed threshold, but *clone*(f_1, f_3) obviously does not. We therefore cannot assume transitivity for clone relations in general.

However, as noted above, transitivity holds for exact matches. Free granularity clones that are built on exact matches between the comparison units are thus transitive, *e.g.*, the clone relation of Kamiya et al. [KKI02]. Fixed granularity clones that use a zero distance threshold (see Merlo's et al. [MAK]) also achieve transitivity for the relation.

Transitivity is a useful property: A relation which is reflexive, symmetric, and transitive is an *equivalence relation*. This enables the members of the relation to be represented by *equivalence classes* which possess a number of advantages:

- An equivalence class representation of clones has the lowest cardinality of all possible clone representations (see Section 3.7).
- Equivalence classes are strictly disjoint: a clone only belongs to one cluster.
- We can build the equivalence classes during the comparison. A source unit must then only be compared to a single representative of each class. This may reduce the number of comparison drastically.

```
if(num_pos > 30) {
 fprintf(stderr,"%s:regular expression too long\n",prgnm);
 free(r_pat);
 if (!EXITONERROR) {
```

```
if(M > 30) {
 fprintf(stderr,"%s:regular expression too long\n",prgnm);
 if (!EXITONERROR) {
```

Figure 3.6: The two only occurrences of the literal '30' in the AGREP system. The fragments stem from two files.

```
filename_split (spec_outfile, &base, &tab, &ext);
full_base_name = xstrndup (spec_outfile,
    (strlen (spec_outfile) - (ext ? strlen (ext) : 0)));
```

```
filename_split (infile, &base, &tab, &ext);
short_base_name = xstrndup (infile,
          (strlen (infile) - (ext ? strlen (ext) : 0)));
```

Figure 3.7: A clone pair of line size 3 from the BISON system.

```
olp->length = 0;
olp->maximum = 0;
olp->list = 0;
olp->break_label = 0;
olp->continue_label = 0;
olp->return_label = 0;
```

```
this->target = 0;
this->ingredient = 0;
this->set = 0;
this->pred = 0;
this->single_thread = 0;
this->op = 0;
```

Figure 3.8: Two object initializations from the COOK system.

3.6.3 Minimal Clone Size

To restrict the number of clones that are reported by a detection tool, filters must be implemented. A filter criterion that is chosen frequently is the minimal size of the clone. This excludes small clones under the assumption that small equals irrelevant. Logic, however, is embodied in source fragments of any granularity, there is no lower boundary *per se* for the size of a clone. Expressions, where primitive terms and operators are combined, can embed important knowledge. Such knowledge can even be embodied in a single constant, *e.g.*, a string that is the name of a command, or an integer number that is a configuration parameter. This claim is supported by the fact that programming languages have invented many mechanisms to help avoiding these types of clones and the potential errors introduced by them.

If a literal number that appears in two locations in the source is a clone or not has to be derived mostly from contextual information (hopefully from comments) and can only exceptionally be derived from the code, *i.e.*, from the number itself ("superstar" numbers like 3.141592 are usually recognized on their own account). In Figure 3.6 we see a clone of about 5 lines. Since one of the instances contains the line `free(r_pat);` which the other does not, a refactoring is not straightforward. Due to its small size a reengineer would probably deem the clone irrelevant. The literal parameter `30` however makes this clone relevant: changing one without the other may lead to inconsistent system behavior.

Minimal size is commonly measured in number of lines. For the comparative study of Bellon [Bel02a] a minimal clone length of 6 lines was agreed upon. This measure, however, can skew the results since the

amount of logic per line varies wildly. To illustrate we can compare two clone pairs: In Figure 3.7 we see three lines of copied code which would be filtered out by the 6-lines threshold. The clone, however, is an interesting candidate for a macro if found a significant number of times in the code. In Figure 3.8 on the other hand, we see two code fragments which fulfill the 6-lines threshold. The logic content per line is however small, and the code does not lend itself to being refactored. The only, quite vague, information that we can get from their similarity is the common intent of *initialization*.

3.7 The Clone Groups

The question of how the detected duplication is presented to the user—the engineer charged with the reengineering effort—becomes important when confronted with the number of clone pairs that result from analyzing a system. To facilitate the reengineering tasks it would be favorable to find a structured representation which reduces the number of clones that one has to process, *i.e.*, a representation which has a cardinality that is significantly below the number of clone pairs.

A group of clones is essentially a group of source units or source fragments. The fragments can be grouped together by two criteria: they are all copies of each other, or they are located in close proximity of each other, neighborhood being another, weaker form of *similarity*.

3.7.1 Grouping Clones by the Clone Relation

If the clone relation is the criterium for being a member of the group we have the following possibilities:

Clone Pair Enumeration: The simplest aggregation is just an enumeration of all clone pairs. This list, however, can potentially grow large and contain much redundant data. If we have n instances of the same source unit, for example, we will have $\binom{n}{2} - n$ clone pairs. To be able to efficiently process the duplication we must remove the redundancy by creating more concise representations of clones.

Equivalence Classes: The grouping of clones into equivalence classes is the best representation for a number of advantages:

- The cardinality of the data is reduced most effectively.
- Equivalence classes are disjoint: every clone is only part of a single group.
- Since equivalence implies transitivity it is likely that we can reach all members of an equivalence class with a single refactoring measure.

Equivalence classes can be formed if the clone relation is transitive. Transitivity is not a general property of clone relations as we have argued above. Even if transitivity does not hold for all clone pairs, there might, however, exist subsets consisting of clones which are mutual copies of each other. Groups formed from these sets might not be disjoint or the number of groups will not be as low as the number of equivalence classes. If transitivity is completely absent from every possible subset, the size of mutually copied clone groups will however not be greater than one, we will have no more than an enumeration of clone pairs.

Characteristic Sets: Another type of groups are the *characteristic sets* [Gie03] which groups all clones of a given source unit into a set:

Definition 3.7.1 *Let M : SrcUnits $\rightarrow$ SrcUnits $\times \mathcal{P}$(SrcUnits) be a function that maps every $x \in$ SrcUnits to all clones of x, where x is the characteristic element:*

$$M(x) = (x, \{\ y \mid \mathit{clone}(x, y)\ \})$$

This clone presentation has the following properties:

- The number of groups formed by this representation is |*SrcUnits*|. This reduces the cardinality of the clone data in cases where we have n instances of the same source unit (generating $\binom{n}{2} - n$ clone pairs).
- The sets of elements associated with the different x are not necessarily disjoint.

Overlapping and Containment among Free Granularity Clones

If the clone relation is defined between arbitrary source fragments, the situation is complicated by the relations of *overlapping* or *containment* which can exist between fragments of free granularity clones (in contrast to the disjoint source units of fixed granularity). Whereas copied fragments which overlap and contain other fragments enable a much more detailed picture of the duplication situation, they also require a more complicated analysis.

To reduce this complexity we first aggregate the fragments until we have only a single representant for all overlapping and containing fragments. The relation *contains*(f_1, f_2) means that fragment f_1 contains all of fragment f_2. The relation *overlap*(f_1, f_2) indicates that a non-empty fragment g is contained in both f_1 and f_2, but neither *contains*(f_1, f_2) nor *contains*(f_2, f_1) holds. For the detailed definitions of these relations see Appendix D.

Containment is a strong relation among source fragments since the containing fragment covers the contained entirely. The container can therefore represent the contained fully.

Definition 3.7.2 (Containers) *For a set of source fragments $\mathcal{S}$ and a fragment $f \in \mathcal{S}$, the set of containers of f in $\mathcal{S}$ is defined as*

$$\textit{containers}(f, \mathcal{S}) = \{\ g \mid g \in \mathcal{S}\ \wedge\ \textit{contains}(g, f)\ \}$$

By recursively establishing all possible container relations we get a hierarchy of clones. The top-level fragments are not contained by any other fragment.

Definition 3.7.3 (Top-Level Fragments) *For a set of source fragments $\mathcal{S}$ we define the set $\mathcal{T}$ of top level fragments of $\mathcal{S}$ as*

$$\mathcal{T}(\mathcal{S}) = \{\ f \mid f \in \mathcal{S}\ \wedge\ containers(f, \mathcal{S}) = \emptyset\ \}$$

The top level fragment can be said to represent all fragments it contains. The problem is that we can still have overlapping among the top level fragments. We need to create artifical source fragments that cover overlapping source fragments fully.

Definition 3.7.4 (Non-Overlapping Top-Level Fragments) *We identify all tuples of top level source fragments*

$$(x_1, \ \ldots\ , x_n),\ x_i \in \mathcal{T}$$

where $\forall i$ with $1 \leq i < n$ it is true that overlap(x_i, x_{i+1}).

For each tuple we create a new source fragment f where beg(f) = beg(x_1) and end(f) = end(x_n). We replace all source fragments from the tuple by f. The new set of source fragments is $\mathcal{T}'$.

The set $\mathcal{T}'$ effectively removes the overlapping relations from the set of source fragments involved in a clone. The elements of $\mathcal{T}'$ are disjoint, just like the elements of *SrcUnits*. We can now assemble all source fragments into disjoint containment hierarchies.

Definition 3.7.5 (Containment Hierarchy) *A containment hierarchy is created from the elements $x \in \mathcal{T}'$:*

$$H(x) = (x, \{\ y \in \mathcal{S} \mid contains(x, y)\})$$

Each source fragment that is involved in a clone is now represented by a top level fragment. Having a fixed set of source fragments we can now use the characteristic set representation from Definition 3.7.1 for free clone granularities as well.

3.7.2 Grouping Clones by Regional Coherence

In the previous section we have assembled groups of clones according to the *clone* relation and the relationships of the source fragments involved in the clones. Additionally, clones can be grouped according to the location of source fragments, *i.e.*, the containment of the fragments in certain larger regions of the system's code, for example files, directories, or subsystems. These regions are independent of the source unit and mostly larger. The general purpose of the code in these regions is familiar to engineers who have been working with the system, and by associating the regions with the clones the knowledge about the system is transferred to the clones.

We build the location-oriented grouping of clones on top of the representations discussed above. This leads to three levels of duplication entities as illustrated by Figure 3.9. The three entities form a containment hierarchy: Each higher order entity aggregates the lower level entities.

1. **Clone Pairs:** The lowest level of detail on which to describe duplication is the clone pair *clone*(f_1, f_2). The pair comprises two source code fragments f_1 and f_2 which are copies of each other.
2. **Clone Classes:** A *clone class* is any clone representation which declares more than two source fragments similar. In the optimal case, this would be the equivalence classes built from a transitive clone relation.

   The *domain* of a clone class is the set of source entities from which its source fragments stem. The domain of the clone class in the middle of Figure 3.9, for example, are the files *F11*, *F12*, and *F13*.
3. **Clone Class Families:** We group all clone classes that have the same domain to form a *clone class family*.

Note that the clone class family is not only a convenient way of reducing the cardinality of the set of all clone pairs. This representation also holds direct interest for the reengineer: First, since clone class families contain only entire clone classes, they assemble all instances of a source fragment found in the system. Second, a clone class family reveals duplication activity that goes beyond the duplication of a single continuous source fragment. If two fragments, which were initially copies of each other, evolve differently over time, they may not be recognized as one clone pair any more (the *split duplicates* problem mentioned in [Kri01]). The clone detector may identify smaller parts which are still similar individually. A clone class family will reunite those clone fragments. In summary, a clone class family aggregates all elements that are necessary to make informed decisions about refactoring measures for a particular fragment of copied code.

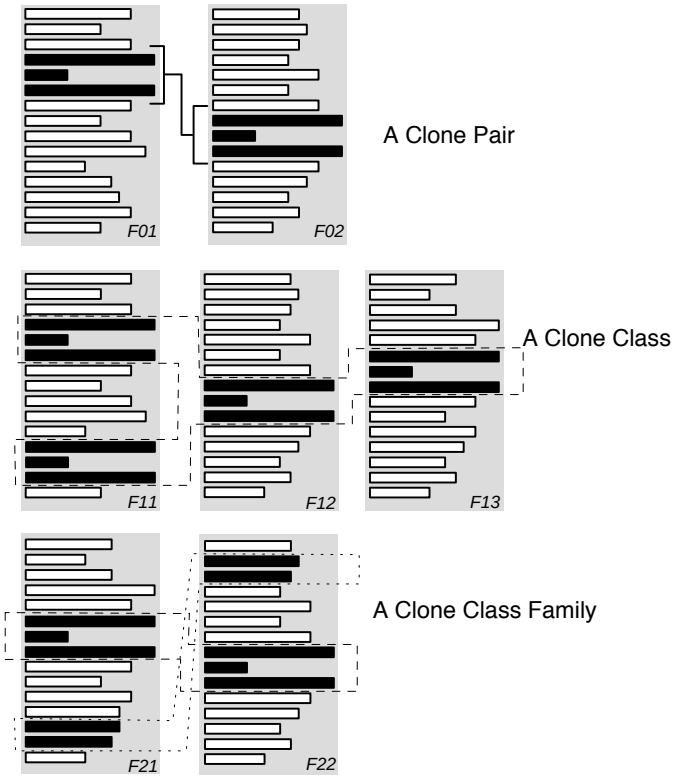


Figure 3.9: Containment hierarchy: *Clone Pairs*, *Clone Classes*, and *Clone Class Families* and the source files they are found in.

3.8 Conclusions

A clone is objectively and subjectively defined:

1. An objective way in which similarity between source code fragments is described. This serves as a basis for the construction of a mechanical detection tool.
2. The relevance of a clone which determines if the mechanically retrieved candidates are useful or not is dependent on the reengineers tasks. Relevance cannot be encoded in automatic filters in all occasions which requires human arbitration.

The clone relation that we establish between two similar source fragments is reflexive, symmetric, but only transitive under certain circumstances.

We can distinguish between fixed and free granularity clones. Fixed granularity constrains the number of potential clones and its results are interesting from a refactoring perspective. Free granularity clones trace the similarity of the source code in much greater detail. Not all of the information that we can gain from the free granularity clones is immediately usable for refactoring purposes, but rather helps us to understand the program better.

Clones are finally grouped into clusters by two properties: derivation from a common original fragment, and close physical location in the source text. We are not able to always build equivalence classes of clones since transitivity is not given in general for the clone relation.

Chapter 4

An Analysis of Duplication Detection by String Matching

In Chapter 2 we have selected *Adaptability* and *Scalability* to be the reengineering goals that guide our analysis. This chapter investigates methods for detecting duplicated code which support these goals. We first select the basic method and formats on which to base a clone detection approach in Section 4.1, motivating and explaining the impact that these choices have on the different elements of a detector. The analysis in the rest of the chapter is then a thorough investigation of the possibilities for clone detection when only using minimal parsing and string matching for a comparison method. Whereas string matching has been used as a comparison mechanism before [Bak92][Joh93][CDS04], we extend this investigation to the phases before and after the comparison.

The stages of the clone detection process that have been laid out in Section 2.2.2 give the chapter its structure:

Transformation. Even though string matching is used as comparison function, we do not treat source code as mere text. We recognize as much syntax elements as we can while remaining language agnostic. We identify the syntax elements which belong to the superficial features of the code in order to concentrate the comparison on the essential features. (Section 4.2)

Comparison. The organization of the string matching influences both the performance and the sensitivity of the detection. (Section 4.3)

Ranking. Since we aggressively remove features from source lines we risk an increase of false positives. To balance this we must use a post-comparison analysis phase in which to perform more expensive analysis on the comparatively small population of candidate clones. (Section 4.4)

Each section on a detection stage must be read as a list of options which are open to us if we want to build *adaptable* and *scalable* clone detectors. How many of these options—and in which combinations—are to be used for a concrete detector is not detailed here. Some selected combinations of ideas presented in this chapter are investigated in Chapter 6. We derive hints on the construction of a concrete detector from the results presented in Chapter 6.

Our Contribution

The contributions of this chapter are:

- An extensive list of source code normalization measures to improve string-based clone detection.
- An analysis of the normalization measures with respect to their potential for generating false positives.
- An argument for the selection of a word-based index to support exact string matching.
- An extensive analysis of the clone ranking measures that are open to a language independent approach.
- A simple representation of source code as *features* to extend the exact string matching with a fuzzy comparison.
- A list of reengineering tasks and how they can be supported with the ranking measures.

4.1 Characteristics of a Simple Detection Approach

To make the lofty goals of *Adaptability* and *Scalability* concrete, we need to choose characteristics of a clone detection methods which promise to fulfill these goals. In a nutshell these choices are the following:

Code Representation	Strings
Comparison Method	Exact String Matching
Clone Granularity	Free
Comparison Unit Granularity	One Source Line

We explain the motivations for these selections in the next sections.

Code Representation

The selection of the code representation has the largest impact on all aspects of the clone detector. By choosing the *string* as representation format, *string matching* as comparison method obviously follows. The string is the simplest of all possible code representations. This has the following advantages:

- Source code is already in string format. We can essentially just start the comparison without any further ado. The complexity of the transformation phase can be kept low.
- The string being the original representation of the code it contains all aspects of the code. When we use the same format we can base our comparison on every such property.
- Strings of source text is the most space efficient form to express the logic of the code.
- The universal data type string can hold any data. We can encode any source code property in string format.
- It is possible to transform strings efficiently with methods like regular expressions for which mature implementations abound, *e.g.*, PERL [WCO00].
- A single string or a sequence of strings are a flat format in contrast to linked structures like trees and graphs. The comparison of flat data is in general simpler than the comparison of structured data.

The problem we face when using strings is its uniform nature consisting only in characters. Conceptually different program elements are no longer distinct when represented as a string. We cannot separate the superficial from the essential with only string matching. For example, string matching will give equal weight to the name of a function and the statement delimiter in the string `f(a);`, whereas a programmer will weigh them totally different.

To overcome this problem we can employ parsers which interpret the source text, separating elements of different types and enabling an informed comparison with some distinction capabilities. Parsers, however, are hampering the goal of adaptability because they are usually tailored to a single programming language. They are difficult to maintain in the face of proprietary dialects or project dependent quirks of development teams:

> Parsing the program suite of interest requires a parser for the language *dialect* of interest. While this is nominally an easy task, in practice one must acquire a tested grammar for the dialect of the language at hand. Often for legacy codes, the dialect is unique and the developing organization will need to build their own parser. Worse, legacy systems often have a number of languages and a parser is needed for each. Standard tools such as Lex and Yacc are rather a disappointment for this purpose, as they deal poorly with lexical hiccups and language ambiguities. [BYM+98]

Following the goal of adaptability therefore means to employ only *minimal* parsing technology. Minimal parsing means two things:

- To only recognize fragments of a language, do not attempt to analyze the entire program. This is basically the same idea as the one of island grammars [Moo01]. An island parser will only recognize in detail some *islands* of interest in the grammar, and leave the rest of the code uninterpreted.
- To only parse elements which can be found in every programming language and identified easily so as to make the adaption of the parser to another programming language a matter of configuration.

Comparison Method

There are several advantages to a setup characterized by a simple comparison method like exact string matching:

- The middle step of the detection process is the one that is caught in the grips of the squared time complexity (depending on the size of the input). The first step, transformation, is performed in linear time depending on the size of the input, and the clone analysis after comparison, has linear time complexity in the amount of duplication detected. Therefore, using a fast method in the most time-complex area of the entire process will have the largest impact on the overall performance of the process.
- Since we are using exact string matching the operation is almost trivial. Furthermore exact matching defines an equivalence relation on the strings. This helps to reduce the time complexity of the comparison even further.
- Having only *yes*/*no* results on the lowest comparison level simplifies the interpretation and visualization of the data when compared to dealing with percentages of similarity.

Clone Granularity

The selection of free over fixed clone granularity is motivated by the following considerations:

- We want to be able to detect self-similarity in source entities like functions. We know from experience in reengineering projects that functions in legacy systems can be long and self-repetitive and thus contain interesting examples of duplicated code.
- We put much of the decision about the relevance of detected duplication into the hands of a human arbiter. The requirements of well-formedness can thus be relaxed compared to what must be given to an automatic post-processor. This strategy is however only affordable for small- to medium-sized systems.
- We also want to maintain the exploratory nature of duplication detection. The patterns of similarity revealed by the dotplot visualization presented in Section 5.2 are of a great variety which cannot all be captured with automatic pattern matchers.

Note that it is straightforward to map free-granularity clones to source entities, *e.g.*, functions, and to report fixed granularity clones.

Comparison Unit Granularity

For the chunking of the source code, *i.e.*, the selection of the granularity of the comparison units, we have a number of options: statements, substrings of fixed length (nGrams), lines, tokens, characters.

The motivation for using source *lines* is the following:

- The chunking of the code into lines is the simplest of all the possible granularities.
- We subscribe to the motivation that Baker has given:

  > The premise underlying DUP is that copying is most often accomplished by means of an editor. Therefore, the resulting copies will be largely the same line-for-line [...]. Given these assumptions, the approach taken in DUP is line-based. [Bak92]

- With a finer granularity resolution like tokens the parts of the algorithm which in time- and/or space-complexity depend on the size of the input would require more resources.
- Retrieved clone candidates that are presented to the maintainer for investigation are in their original format that the programmer recognizes from the source file.

User Control

With only the limited knowledge about the code that we can get from simple parsing, we cannot hope to automate the clone management very much. Manual control is therefore required to a large extent. We can however profit from some advantages. The following list collects motivations, some of which we have already mentioned above:

- Language independence enables the user to compare essentially every possible text input. Church&Helfman have compared DNA sequences, parliamentary discussion protocols, file names, and attributes from version control databases [CH93].
- The relatively small granularity which is the result of selecting lines as comparison units, together with choosing free clone granularity means that a rich variety of similarity patterns is exposed.
- Direct visualization of duplication patterns gives the user control over which clones are worth investigating.

4.2 Transformation of the Source Code

As the first step in detecting duplicated code we transform the code into an intermediate format which is afterwards fed into the comparison mechanism. The general goal of the transformation phase is to increase recall, *i.e.*, to avoid false negatives. False negatives are introduced by slight alterations of the copied code which fool the comparison mechanism into overlooking a clone. In Appendix B we give an overview of post-copy editing changes, which we here to collectively refer to as *variability*. Detection techniques must *normalize* the superficial differences and compare only the essential parts of the code. As in our case the comparison phase is designed to be simple, *i.e.*, does *not* cope with variability, we must take care of variability during the transformation *before*, and the aggregation *after* the comparison phase. The added benefit of such a configuration of the detection process is that the costly computations which deal with variability are only applied during the linear transformation phase, and not during the quadratic comparison phase.

This section discusses the possibilities of code normalization under the premises of the *adaptability* goal. Since all transformation measures are applied to every line of code once only, giving the transformation a linear time complexity, the *scalability* goal can be neglected here. The section starts with a discussion of the different normalization measures (Section 4.2.1 until Section 4.2.4), and then discusses how the number of false positives is likely to be affected by the different normalization measures (Section 4.2.5).

4.2.1 Types of Transformations

There are two important dichotomies in the transformation techniques we employ:

Layout Changing: Since our approach is based on lines, it is sensitive to changes in the layout of the code. Transformations that delete and add line breaks change this layout.

Content Changing: A transformation technique can change the actual source text by removing or replacing certain elements. Content changing transformations can be further subdivided:

- **Normalization:** A *normalizing* transformation replaces a specific element in the source text, like the identifier `counter`, with a generic element, *e.g.*, `p`.
- **Filter:** A *filtering* transformation deletes an element from the source code without leaving a trace.

We discuss layout changing transformations in Section 4.2.2 and content changing transformations in Section 4.2.3 (Normalization) and Section 4.2.4 (Filtering).

4.2.2 Reorganizing the Layout of the Source Code

In our approach the lines of the source code form the set of comparison units. By inserting and removing line breaks at certain positions of the source text, we influence the individual members of this set, deciding which strings are presented to the comparison. If we isolate any substring by surrounding it with line breaks we elevate it into the rank of a comparison unit. Instead of tokenizing the entire input, which incurs the disadvantage of a large increase of space and time requirements, we can select the exact substrings for which we want to increase the sensitivity of the comparison.

Pretty Printing of Source Code

Pretty printing is the most obvious technique which reorganizes the layout of a program. The goal of pretty printing in our context is to prepare a normalized layout of the code, *i.e.*, to split the source code in standard places. To normalize an **`if`**-statement, for example, we put the condition on a line by itself and all the statements from the `then` part on their own lines, as well as the statements of the **`else`** part. False negatives due to

individual alterations of line breaks in copied code—a frequent phenomenon for programmers adapting other people's code to their own style—can thus be avoided.

We can employ pretty printing without abandoning the goal of language independence. Two reasons exist: *i)* a pretty printer is readily available for every programming language, modern development environments integrate them into their editors; *ii)* if none should be available, building a pretty printer from scratch is not difficult.

Pretty printers are organized as a language dependent front-end, which transforms source code into a markup language like BOX [dJ00], and a language independent format engine which prints the BOX entities. The format engine of Oppen [Opp80] is even simpler: it recognizes only strings separated by blanks, and grouped in blocks.

We can build a configurable pretty-printer which layout-normalizes many languages. Since we only desire a normalized line as input for a tool, we do not care for a "pretty" format with optimized indentations that facilitates code reading. A back end that produces useful output for our purposes needs only to insert line breaks at the appropriate places of the code. For the front end we can build a minimalist parser which recognizes the basic structure of the code consisting of blocks, statements, expressions, and comma separated lists. The parser is made configurable with block and statement delimiters and parentheses to recognize expressions.

Since pretty printing does not add or remove anything except line breaks, a mapping of the preprocessed code back to the original layout for the presentation of the results is not difficult.

4.2.3 Normalizing the Source Code

Many of the changes that programmers apply to copied code are superficial. If fragment f_1 is the same as fragment f_2 except for the names of the variables, the semantics of the code will not be changed.[1] Such differences do not affect the clone relation between the original and the cloned fragment. Our comparison function is however confused by these changes which results in false negatives. *Normalizing* the source code means to replace language constructs which could be superficially different from their copies with a generic token that will let the comparison succeed. When determining whether we should normalize an element or not, the question is always if the normalized feature carries enough semantic weight so that its differences break the clone link between two copied fragments. We should only normalize elements which are *lightweight*. Examples of elements that should not be normalized because they are too important are:

Assignments: An assignment is an essential operation which is not easily exchanged with another operation.

Function Calls: Names of functions carry considerable semantic weight, namely the entire intent of the function.

The elements we *do* consider useful to be normalized are listed below. All of the normalizations are simple to handle with only regular expressions.

Identifiers: We normalize the names of variables. This helps to avoid false negatives of the sort that can be seen in a typical example from the COOK system:

```
i ^= 0x40;
b[0] = '\\';
b[1] = '0' + ((i>>6)&3);
b[2] = '0' + ((i>>3)&7);
b[3] = '0' + (i&7);
b[4] = 0;
```

```
c ^= 0x40;
buf[0] = '\\';
buf[1] = '0' + ((c>>6)&3);
buf[2] = '0' + ((c>>3)&7);
buf[3] = '0' + (c&7);
buf[4] = 0;
```

Normalization replaces each identifier with a special token, *e.g.*, p.

[1] To be exact, the semantics of the code are unchanged if the variable names have been α-converted.

Literal Constants: We normalize numbers and strings. The following is an example from the COOK system showing as the only difference one literal character:

```
/* look for the closing paren */
if (s_len < 1 || s[s_len - 1] != ')') {
```

```
/* look for the closing quote  */
if (s_len < 1 || s[s_len - 1] != '\'') {
```

It is common [Bak92][KKI02] to normalize identifiers and literals to the same token p. This makes sense since a literal is just as well a parameter of the computation as a variable is—they are interchangeable elements of the source code.

Built-In Types: In declarations, basic types like **`int`**, **`long`**, are replaced by a generic token like *numi*, and **`float`** and **`double`** are replaced by *numf*.

Operators: Arithmetic operators like `+` or `//` are replaced by a generic operator sign. Comparison operators like `<=` or `==` can be normalized by `<>`. The pre-increment (and decrement) operators in C can be normalized to post-increment operators.

Access to Member Variables: Members or instance variables can be accessed in different ways. The following motivating example is from a file in the COOK system, the fragments are found about 30 lines apart:

```
sp2->next = sp->next;
sp1->next = sp2;
```

```
sp2->next = sp->next;
first_shift = sp2;
```

This case must be extended to the access to member functions:

```
a.doIt();
```

```
b->doIt();
```

The normalization which unifies these different access modes is the removal of the names of structures/instances (`a` and `b`) and the access operators `.` and `->`, *i.e.*, leaving only the name of the field or the function (which in turn is then normalized as identifier).

4.2.4 Filtering the Source Code

By *filtering* the source text we mean the removal without replacement of certain code fragments from the source text prior to comparison. Removal of elements can be done on the level of the comparison units or below, down to single characters (since one single-character difference can make the exact string match fail).

Removal of Comparison Units: Entire comparison units (source lines) are removed if they occur in high frequencies and/or are of minor importance. The motivation for removing them is that they affect resource requirements through *i)* the time they take to be compared, and *ii)* the large number of atomic matches which are mostly spurious. A high frequency implies also that their similarity is unsurprising and thus uninteresting.

The following table lists for a number of systems some frequent comparison units with absolute occurrences and frequency rank:

Comparison Unit	AGREP		BISON		GCC 3.3		MAIL SORTING	
`{`	255	*(4)*	742	*(1)*	63817	*(2)*	5319	*(2)*
`}`	1894	*(1)*	739	*(2)*	65324	*(1)*	9575	*(1)*
`else`			94	*(5)*	11081	*(4)*	898	*(8)*
`break``;`	120	*(13)*	103	*(3)*	13961	*(3)*	474	*(13)*
`return``;`			28	*(9)*	3056	*(12)*	371	*(20)*
`return` `0;`	143	*(8)*			5299	*(8)*	440	*(15)*
`return` `-1;`	300	*(3)*			380	*(61)*		
`int` `i;`			51	*(8)*	1541	*(21)*	104	*(78)*

Note however, that even though these comparison entities are unimportant if copied alone, they may be important as part of a longer sequence of copied code, giving a smaller clone candidate just enough weight to be relevant in the retrieval processes after comparison. It is therefore advisable to remove them for memory optimization purposes only.

Removal of Tokens: Similar to the motivation for the normalization measures above, sometimes the mere presence of some token introduces variability without disturbing the fundamental clone relation of two copies. Contrary to the code constructs which are normalized because their presence is considered important, these tokens are however of an insignificant weight. To avoid the confusion that the presence of such tokens causes the comparison, we remove the tokens completely from the source text. Total removal of tokens is akin to the Information Retrieval techniques of *stopping* (removing words occurring in high frequencies), and *stemming* (removing word-suffixes to unify variant forms). Stop words and word-suffixes have an "ornamental" character: they do not carry much semantic weight for the task at hand.

Some of these filterings are useful only in a certain language. We note the names of some of the languages to which the filters are applicable in the following descriptions.

Delimiters: To remove the statement, expression or block delimiters from the source code, simplest pattern matching is sufficient.

The *this* Pseudo Variable (C++**,** JAVA**):** The variable `this` is optional when accessing members of C++ or JAVA classes. The two fragments below come from the same C++ class and refer to the same instance variable:

```
return (this->validity >= temp_synt_valid);
```

```
if (validity == temp_synt_valid)
```

Namespace Indicators (C++**,** JAVA**, some** SMALLTALK **dialects):** Since the indication of a namespace is optional if the reference is expressed within the same namespace, JAVA package names and C++ namespaces are removed to normalize the code.

Type Qualifiers (C++**,** JAVA**):** Statically typed languages have a number of type qualifiers like **`const`**, **`static`**, or the **`public`** and **`protected`** access specifiers in C++, which do not carry much semantic weight in clone detection. Type-casts, C++-template parameters, and the dereference operator `&`, are other artefacts of a static type system that do not affect the control and data flow.[2]

There is also a C idiom to give a **`struct`** its name again using **`typedef`**, for example

```
typedef struct expr_position expr_position;
```

[2]This is exemplified by the existence of dynamically typed languages like SMALLTALK which don't need any of these elements.

The removal of the **`struct`** keyword from the text helps to avoid some false negatives as in the following two function headers from the COOK system:

```
static int
 interpret(idp, ocp, pp)
    id_ty                         *idp;
    opcode_context_ty             *ocp;
    const struct expr_position_ty *pp; {
```

```
static int
interpret(idp, ocp, pp)
   id_ty                  *idp;
   opcode_context_ty      *ocp;
   const expr_position_ty *pp; {
```

Type Casts (C**,** C++**,** JAVA**):** Type casts are used for a variety of operations, some of which do not carry much semantic weight. The removal of *constness* from a parameter's type, for example, is a way to deal with the constraints of the type system which does not have any effect on the logic of the code. Arithmetic narrowing or widening is another such operation where the difference between code with or without the typecast is marginal. Some operations, however, that are triggered by type casts can be arbitrarily complex conversion routines. Unfortunately, the convoluted type cast syntax rolls all these different operations into one syntactic representation, making it hard to filter only one type of operation (except for the arithmetic narrowing or widening, which can be identified by the use of the builtin arithmetic type names). The more recent C++ standard introduced a range of casting operators, *e.g.*, **`static_cast`**<>, which allow us to filter the lightweight operations and keep the more important ones.

Labels (C**,** C++**):** Targets of **`goto`**-jumps are interesting only if we compare control-flow other than the sequential order of statements in a function. Our simple comparison disregards this information. The following fragments are from two files of the WELTAB system and are exact copies except for a label:

```
      if(card[0] == 'Y' || card[0] == 'N' ||
         card[0] == 'n' || card[0] == 'n') goto x541;
      cvcil(card,0,len,&vote[k]);
      if(vote[k] >= 0L && vote[k] <= vtpoll) {
        totoff = totoff + vote[k];
```

```
      if(card[0] == 'Y' || card[0] == 'N' ||
         card[0] == 'n' || card[0] == 'n') goto x541;
      cvcil(card,0,len,&vote[k]);
x543: if(vote[k] >= 0L && vote[k] <= vtpoll) {
        totoff = totoff + vote[k];
```

Note that the removal of labels implies the normalization of the targets in each **`goto`** statement.

4.2.5 The Introduction of False Positives through Normalization

When we normalize and filter source code in the manners explained in the previous sections, we remove distinguishing features from it because we perceive the eliminated differences to be unimportant. As a result, more matches between the normalized items are found by the comparison. Since we have relaxed the criteria for what is considered equal, some matches will not keep their equality status under scrutiny of a human expert. These are called false positives. The question is, how many false positives are generated by the normalization and filtering techniques. The discussion of this section groups the techniques according to *no*, *some*, or *high* potential for creating false positives.

No False Positives

Some normalization and filtering measures do not touch fragments that are relevant for the meaning of the code. Removal of these elements thus does not have any effect on the ratio of false positives.

- **Removal of Non-Code Text:** The removal of comments, white space, and especially the deletion or insertion of line breaks do not increase the number of false positives at all. Since the text fragments are not source code, their removal cannot make different source fragments the same. An exception are languages where white space has meaning, *e.g.*, PYTHON, where the beginning and the end of blocks are signaled by indentation. As an example, the following PYTHON fragments would be declared a clone even though they are not the same:

```
l = a[k]
if test(k):
   ol = b[k]
   if ol:
      c[k] = ol
else:
   c[k] = l
```

```
l = a[k]
if test(k):
   ol = b[k]
   if ol:
      c[k] = ol
   else:
      c[k] = l
```

  If one wants to avoid false positives in these cases, a small preprocessor can be written as an *de*-normalization measure to insert block delimiter tokens into PYTHON code.

- **Removal of Statement Delimiters:** It is not possible that the removal of a statement delimiter causes two valid statements to become similar to a single valid statement. The only example where this could happen is the following, where the first statements ends with an identifier and the second begins with an identifier on the same line.

```
int a = b; c = 3;
```

```
int a = bc = 3;
```

  However, if variable names are normalized *before* the removal of statement delimiters and white space, it is not possible that the two lines of code are transformed to the same string: the transformation result `intp=pp=3` on the left side is not the same as `intp=p=3` for the right.

- **Removal of Access Specifiers:** The C++ (and partially JAVA) access specifiers `public`, `protected`, and `private` do not belong to the part of the code where interesting duplication occurs. Their removal thus does not create additional false positives.

- **Normalization of Literal Constants:** Literal constants alone represent just data to the computation. If two fragments of code become the same after normalization, the constant can always be made into a parameter of the unified fragment.

```
a = b / 0.1;
```

```
a = b / 10;
```

  However, if constants and variable names are normalized with the same replacement token, the false positives rate will be that of identifier normalization (see below).

- **Removal of Type Qualifiers:** Type qualifiers like **`final`** in JAVA or **`const`** in C++ give compilers opportunities for advanced compile time checks. Their removal does not change the semantics of the code at all.

```
int foo(const str& a) const;
```

```
int foo(str& a);
```

- **Removal of Type Casts:** As mentioned above, type casts can represent a variety of operations, some of which have considerable semantic weight. For example, a type cast-triggered conversion between two types can involve a large amount of logic, *e.g.*, the conversion from a `String` to a **`float`** involves the parsing of the string representation. Removing the type casts then means the loss of an important step in the code. However, the code surrounding the type cast is found duplicated only if the converted value subsequently has the same operations applied to it than to the non-converted value. If this is the case, the two types are similar in nature and the code might therefore be considered a clone after all.

Low Rate of False Positives

The potential for causing false positives increases if transformations remove elements of the source code relevant to the data- or control-flow. The normalization measures in this category supposedly cause few false positives or only of such kind that the two fragments are so close as to make a refactoring possible.

- **Removal of Parentheses:** Parentheses are used for three different purposes. First they delimit conditionals in **`if`**- and **`while`**-statements, and loop specifications in **`for`**-statements. Removal of these parentheses can cause false positives under some (rare) circumstances and in conjunction with the removal of block and statement delimiters as can be seen in these examples:

```
for (a=0;a<n;) b++;
```

```
for (a=0;a<n; a++)
```

```
while() { b++;
```

```
while(b++) {
```

  Parentheses also enclose lists of actual parameters to function invocations. Removing parentheses together with whitespace collates function names with parameter names. In a presumably rare situation this can lead to false positives:

```
list(all);
```

```
listall();
```

  Parentheses are also used to determine precedence of expression evaluation. If they are removed from these formulas, false positives can definitely occur:

```
a = b * (c - 3);
```

```
x = y * z - 3;
```

- **Removal of Block Delimiters:** False positives may be caused by the removal of block delimiters, as can be seen in the following example:

```
if (cond) {
  a = 1;
}
b = 2;
```

```
if (cond) {
   a = 1;
   b = 2;
}
```

- **Removal of Pointer Type Declarator and Operators:** In C/C++ variable declarations, the pointer operator `*` represents type information. Its removal, in conjunction with normalization of identifiers and basic types, can cause false positives. The two follwing **`struct`**s would be considered the same eventhough they are not clones:

```
typedef struct {
  long*  vect;
  double val;
  char** ptr;
} a;
```

```
typedef struct {
  int    type;
  float* prcnt;
  char*  cptr;
} x;
```

  The removal of the pointer operators `*` and `&` in C/C++ causes false positives only if functions with the same name but different parameter types exist that have vastly varying purposes, a rare case in our opinion. The two following fragments would be declared clones even though their logic is quite different:

```
int   a = 1;
int*  b = &a;
doThis(&a,b);
```

```
int x = 1;
int y = 2;
doThat(x,y);
```

  Note however that these kind of false positives would also occur if the `*` and `&` tokens would not be removed, because the name of the variable usually (especially when normalized) does not tell if its type is **`int`** or **`int`**`*`.

- **Removal of Namespace Indicators:** A mismatch can result if classes from different packages have the same name. Since the same name likely implies similar functionality a match between them is probably not a completely false positive.

```
java.sql.Array;
```

```
java.lang.reflect.Array;
```

- **Removal of Labels:** This is a similar case to the removal of block delimiters in that control flow is occluded by the normalization, as can be seen in the following example:

```
 if(cond1) goto X
 else
   if(cond2) goto Y;
   a = 1
X: b = 2;
Y: c = 3;
```

```
 if(cond1) goto X
 else
   if(cond2) goto Y;
   a = 1
Y: b = 2;
X: c = 3;
```

  Labels, different from blocks, are however used only occasionally in structured programming. The contrived example we give here demonstrates that situations which are different but appear the same when shed of their labels are not occurring very often.

High Rate of False Positives

The normalization measures discussed in this section have the potential to cause many false positives. This is due to the high frequency of their application.

- **Normalization of Identifiers:** Identifiers are the most volatile element in source code. The logic of the code is mainly expressed in the operators and control flow, whereas identifiers 'only' describe data. There are fewer possible statement templates than there are possible identifiers. Removing the identifiers and leaving only the statement templates generates thus a large number of false positives.

```
a = b - c;
d = b * 2;
```

```
a = c - b;
d = a * 2;
```

  If, instead of normalizing all variable names from two sets with the same replacement token, a bijective mapping is installed between the sets, many false positives can be avoided [Bak92]. Such a normalization requires however a comparison mechanism that is more sophisticated than exact string matching, since the parameter names must be adjusted with regard to the start of source fragment that is currently under comparison. If we use free clone granularity, this starting point is constantly moving making re-adjustments of the parameter names a necessary part of the comparison routine.

- **Normalization of Member Access:** If access to member variables and functions are normalized to simple variable accesses, the number of false positives increases because many different source lines are mapped to a single string.

```
a = b->c + d.e;
d.foo();
```

```
x = y + z;
foo();
```

- **Normalization of Operators:** This normalization in combination with the normalization of identifiers have the potential to create many false positives. If we normalize the operators - and * to the same replacement token we get a false similarity between the two following statements:

```
a = b - c;
```

```
a = b * c;
```

- **Normalization of Built-in Types:** If we normalize all integer types with one replacement token, and all floating point types with another token, the rate will be small since all integer types are close in their behavior, just as the floating point types. False positive can occur due to range incompatibilities between the types, for example in declarations:

```
int a = -2147483648;
```

```
unsigned short a = 10;
```

  To normalize all arithmetic types with the same replacement token will lead to many false positives, especially in conjunction with normalized identifiers.

4.2.6 Related Work in Transformation

Clone detectors which use a parser do not normalize the source text which they need in its entirety to build an abstract syntax tree. They can disregard certain elements directly during the comparison phase with the benefit of having at their disposal the semantic knowledge which has been gathered by the parser. Code normalization in the source text is done mostly by clone detection approaches which do not parse the code. For these approaches the following code transformations are very common:

- **Removal of Comments:** Almost all approaches remove comments, except Mayrand et al. [MLM96a], who use metrics that measure the amount of comments, and Maletic&Marcus [MM00] who search for similarities of concepts extracted from comments.
- **Removal of Whitespace:** White space is disregarded by most approaches. Line-oriented approaches, however, remove all whitespace except line breaks. Davey et al. [DBF+95] use the indentation pattern of pretty printed source text as one of the features for their attribute vector. Mayrand et al. [MLM96a] use layout metrics like *Number of Nonblank Lines*.
- **Normalization of Identifiers:** This is practiced by all approaches. The most advanced normalization is the one of Baker [Bak93b] who identifies systematic name changes only. All other approaches change variable names indiscriminately to a single replacement token.

In general, Kamiya et al. [KKI02] have an extensive list of normalizations and filterings to normalize their token-based representation of the code. A method which compresses the code in conjunction with applying some normalizations is used by the plagiarism detector of Finkel et al. [FZMS02]. They reduce each keyword to a single character, *e.g.*, `f` replaces **`for`**. They also replace identifiers with `i` and numbers with `0`.

4.2.7 Discussion

To remove from the source code superficial features which might disturb the comparison we have to invest most of our effort into the task of recognizing syntactic elements. In order to follow the goal of *adaptability* we do not employ parsers, but identify only entities which can be recognized via a set of marker tokens. This can be mostly accomplished with regular expressions. For the pretty printer, which needs to recognize recursive structures, we must resort to a real parser. Since this parser only needs an understanding of the top level structure of a program, we can make it configurable for a wide range of programming languages.

To keep the parsing amount small we apply very general normalizations. This will result in many false positives being retrieved in the comparison. We will therefore need additional analysis post comparison to remove false positives from the candidate list (see Section 4.4).

4.2.8 Other Transformation Options

The question of input transformations has two obvious fields of further research: the selection of the exact set of normalization and filtering techniques that should be applied to a given system, and improving normalization and filtering techniques to reduce the number of generated false positives.

Selection of Normalization Measures

Some of the normalization measures are liable to generate a large number of false positives. Other filtering techniques are quite aggressive in removing parts of the source code before the comparison, which again may lead to a high rate of false positives. We assume that this rate of false positives is dependent on the individual systems that are analyzed. To exercise a certain level of control the user should be given methods which would allow him to select only normalization measures which have a favorable ratio of detecting far apart duplication versus generating many false positives.

The question is very much open if it is possible to estimate the number of false positives before the evaluation of the candidate clones, or even before the comparison, which is the most time-consuming step. It would also be helpful if we would identify certain types or zones of source code where certain normalization measures should not be applied.

Improved Normalization Measures

A problem with the normalizations proposed above is that they are very general. If *every* variable name is normalized with the same token, ubiquitous statements like assignments and simple computations will all be matching and generate much noise. More discerning normalizations should group identifiers and replace all members of a group with the same token. The criterion for grouping identifier names should be a characteristic which is less volatile than the mere name of the variable. This is based on the assumption that if the programmer changes some elements (variables, function calls) in a copied fragment he will choose as replacement a similar one. This is because we think that, most likely, a fragment that is copied forms a conceptual entity and this context restricts the kinds of changes that can be made to the copied fragment. In the following list we are going to touch briefly on a few ideas what *similar* could mean, *i.e.*, what the identifier groups could be based on.

Exploiting Naming Conventions: If programmers use names consistently, *i.e.*, give similar names to variables that represent the same concept, we can cluster identifier names according to these rules. An example of a cluster is for example the names of iteration variables `i`, `j`, `n`, or `count`, etc.

Anquetil&Lethbridge [AL98] have investigated how it can be tested if a naming convention is reliable. It is however not so clear how the clustering of variable names could be done in a fully automatic mode. A solution will likely require manual intervention of the programmer. The question that arises then is, however, if the increased work required on the part of user justifies the increased precision of the detection.

Variable Types: The type of a variable in a given source fragment is more resistant to change than its name. We can reduce the potential for false positives due to uniform identifier normalization by interpreting the type of the variables and replacing their names not with a generic token but with an acronym representing the type, similar to Hungarian notation [Sim76] but without the individualizing name part. As an example, the left original source fragment is changed into the right fragment by using `i` as name for all variables of type **`int`**, `pc` for all pointers to **`char`**, and `pudt` for pointer to `user defined type #1`.

```
str_field(s,sep,fldnum)
  string_ty *s;
  int       sep;
  int       fldnum; {
char   *cp;
char   *ep;
cp = s->str_text;
while (fldnum > 0) {
  ep = strchr(cp, sep);
```

```
str_field(p, p, p)
  string_ty *pudt1;
  int     i;
  int     i; {
char   *pc;
char   *pc;
pc = pudt1->str_text;
while (i > 0) {
  pc = strchr(pc,i);
```

Baxter et al. [BYM+98] have mentioned that they were planning to use type information to filter false positives.

To realize this normalization we need to build a parser which understands the declaration of variables and user-defined types in a statically typed language. The parser must maintain a symbol table and needs to be aware of blocks and the scoping rules of the language. Such a parser is considerably *heavier* than anything we have proposed above. It is also not immediately clear if we could build a version which is configurable for multiple languages since variable declarations differ across languages. Note also that for dynamically typed languages this normalization is not possible since we miss the type information. Using a type inference engine for such a case would definitely break the *lightweight* constraint.

Normalizing Variable Types: Going one step further, we can start to normalize variable types by either the memory footprint of one of their instances, or the complexity of user defined structured types, measured for example by the number of fields that it contains and the variety of the types of these fields. Types of similar complexity would then get the same token to normalize variable names.

A fairly complete parser, including a symbol table, is required to derive from the source code the size and structure of a type.

Normalizing Function Names: We have argued above that function names should not be normalized since they represent considerable weight and are therefore less likely to be changed than the name of a variable. If the non-normalization of function calls leads to many false negatives, we can apply the same grouping techniques to function names to normalize them in a discriminating manner. Instead of memory footprint we can compute the size in lines of a function or derive simple complexity metrics to establish the weight of a function.

The heuristic assumption that programmers will change elements only to similar ones, which is the basis of the proposed normalizations, may of course break down. These normalizations therefore will, with respect to the simple version that uses the p token universally, not only decrease the number of false positives but they might eventually increase the number of false negatives at the same time.

It must also be said that the parsers that gather some of the information as proposed above are fairly large and it is questionable if we can build them general enough to assert adaptability over language borders.

4.3 Comparing the Source Text

Finding self similarity in a text using string matching and without a fixed clone granularity can be seen as a special case of text searching where the problem is as follows: Given a long text $T_{1...n}$ of length n and a shorter pattern $P_{1...m}$ of length m, retrieve all occurrences of P within T. The important difference where code duplication detection deviates from this general model is that P is not given and must be extracted from T itself. This leads to the *all-against-all* character of the detection problem because T and P are the same.

In this section we first motivate why we use exact string matching as a comparison method, then we explain the choice of source text representation to speed up the comparison, and finally we detail the detection algorithm step by step.

Exact String Matching as Comparison Method

Detecting duplicated code is a matter of text retrieval allowing errors. In spite of the transformations discussed above which try to cope with a number of potential differences there are always changes in the code that cannot be leveled by normalization. To retrieve copies despite larger changes is the goal of approximate matching. As our approach to approximate matching we use the principle of "reduction to exact matching" [NSTT00] where either P or T is split into substrings. Multiple exact matches must then be found in close proximity to trigger a verification for a full pattern match.

The question can be asked why we do not use an approximate string matching function which could be more sensitive to varied differences between the comparison units. We consider such a function for comparison inappropriate for a number of reasons:

- Approximate string matching algorithms have a much higher time complexity than exact matching. Since the comparison operation is the one algorithmic step which is executed most frequently, its time complexity will have a large impact on the overall time complexity of the algorithm.
- The standard approximate string matching functions are blind towards the different types of elements of source code. It makes a difference for a software engineer if the change between two lines consists in a keyword or a literal string.
- The main differences between source code fragments lie not in misspellings, *i.e.*, on the word level, but on the structural level.
- The result of an approximate comparison is a percentage value, expressing the difference that separates two comparison units. Partial equality removes transitivity from the relation that is established by the comparison. With exact similarity an equivalence relation is created which enables better optimization of the comparison.

Instead of *during*, we are coping with the differences in code *before* and *after* the comparison: the code transformation levels some of the smaller differences, and the larger disruptions are being taken care of in the analysis phase.

4.3.1 Indices for Text Searching

As we have said above, the initial absence of P requires us to run many queries searching for all possible patterns that can be extracted from T. Since we basically compare everything against everything it is economically feasible to build an *index* for T. An index helps in two ways: during the construction phase of the index, when we make *decision* queries to determine if a substring is already present or not, the index helps to minimize the number of comparisons. After construction the index represents in a compact form all the similarity of the source text and we need only to extract it.

There exist two kinds of indices for text retrieval: *word*-oriented indices for a T that can be regarded as a sequence of words (where a word is a maximal substring not including any symbol from a set of delimiters), and *sequence*-oriented indices for a T where words cannot be distinguished (such as DNA), or where arbitrary sequences are searched for. Both index types are used by existing string-based clone detectors.

Sequence-Oriented Indices: A well known idea in text-searching is to consider a text as the set of its suffices. Since every substring S of T is a prefix of a suffix of T, the search problem of finding S in T can be reduced to finding all suffixes that start with S. A suffix tree [McC76] is a data structure which contains all suffices of a text T and allows searches for P in $O(m)$ time where $m = |P|$. Moreover, all z occurrences of P can be found in time $O(m + z)$. Suffix trees exploit the self-similarity of T directly by representing all common substrings with the same nodes. Since all the suffixes of common substrings are again common substrings, a subsequent analysis step must enumerate all common substrings to find the matches of maximal length [Bak92].

The drawbacks of suffix trees are *i)* that they need a lot of space, $20n$ bytes in the worst case (where n is the length of the string), and *ii)* that "in most applications the suffix tree suffers from a poor locality of memory reference, which causes significant loss of efficiency on cached processor architectures" [AOK02]. Large systems which require the suffix tree to be cached on secondary storage, suffer a significant performance reduction [vRD03]. Suffix arrays [MM90] reduce the space by about a third at the additional cost of $O(\log n)$ during search. Space requirements can also be optimized if patterns are known to have certain properties, like starting only at certain positions in T. In such cases not every suffix of T must be entered into the index, the index points can be selected more sparingly which reduces the size of the index. Recently, new index data structures like the *String B-Tree* [FG99] are investigated which can manipulate large strings in external memory and offer the same search operations like suffix trees, improving them in the worst case.

Word-Oriented Indices: Instead of supporting the search for arbitrary substrings of T, word-oriented indices allow to find individual words or phrases. To build such an index we need to partition T into words or terms of a certain granularity, for example lines, or tokens of source text. An *inverted index* is a list containing all unique terms from the text (the *vocabulary*) and, attached to each term, a list of all the positions where the term occurs in the text (the *postings*). The inverted index has the property that its vocabulary is small compared to the text and can be kept in main memory even for large inputs. The size of the postings list is linear in input size: it requires a pointer into T for each index point. Large lists of postings can be easily managed in secondary memory. After index construction, the sets of postings represent all matches of a single word. If we are looking for phrases consisting of multiple words using a word-oriented index, we need an additional step: the atomic matches between word occurrences must be aggregated so that larger matches can be found.

Building a word-oriented index for T also exploits the self-similarity of the text: the more repetition occurs in T, the smaller the vocabulary is. Vocabulary growth is empirically known to be sub-linear for natural language text. Heap's Law[3] states that the average vocabulary size V depends in the size of the text n by the formula $V = O(n^\beta)$ for $0 < \beta < 1$. For some examples of reduction rates of source code vocabularies see the following table (lines were counted after removal of comments and white space):

System	*All Lines*	*Unique Lines*	
BISON	7432	3899	(52.5%)
GCC 3.3	792,051	325,604	(41.1%)
AGREP	12,246	3423	(28.0%)
MAIL SORTING	113,517	31,240	(27.5%)

The smaller the relative size of the vocabulary, the more similarity we will find in the code. The amount of duplication is therefore presumably higher in MAIL SORTING than in BISON.

[3] H.S.Heaps. Information Retrieval—Computational and Theoretical Aspects. Academic Press, 1978, qtd. in [BYN00].

Choice of Index Data Structure

Both index types, sequence-oriented and word-oriented, are useful for our problem. To motivate a choice between the two we list a number of advantages and disadvantages for each. For the sequence-oriented index the following issues are important:

Sequence-Oriented	
Advantages	*Disadvantages*
• The time to retrieve common substrings longer than a threshold t depends on the number of such common substrings, not on the number of atomic matches.	• The space requirements are large and the data structure cannot be managed well in secondary storage. • Only detects a single kind of pattern: an unbroken sequence of copied text. • The implementation of a suffix tree is not trivial.

For the word-oriented inverted index we have the following arguments:

Word-Oriented	
Advantages	*Disadvantages*
• Makes available for analysis more patterns than only unbroken sequences. • The inverted index is implemented easily. • After the construction of the index it is possible to make an optimization pass over all terms and remove those which will create too many atomic matches. • A visualization of the comparison matrix as dotplot is straightforward.	• Time for the extraction of patterns is dependent on the number of distinct atomic matches. • Additionally needs the construction of a comparison matrix to store the atomic matches in sequence. Matrix construction is dependent in time and space on the number of distinct atomic matches. • Helper data structures like the comparison matrix are fairly complicated to implement.

The important characteristic for suffix trees is that they store the equality information *in sequence*, *i.e.*, they combine the information about atomic matches with the information about their orderin T. Only a difference between P and T breaks a matching sequence apart. Suffix trees thus value the order of the elements over their similarity. In contrast to that, word indices break even continously matching sequences apart. Order between elements is disregarded completely and we need an additional data structure to reintroduce it. Hence comes the dependence on the number of atomic matches for pattern extraction.

Our final choice of the word-oriented index (inverted index) as the data structure to be used is based mainly on its smallness and the ease with which it can be implemented. Also we think that the richness of the duplication patterns to be found in the comparison matrix is of great interest, at least for exploratory duplication investigations.

4.3.2 Algorithm Description

The comparison algorithm which takes the transformed source code and delivers a set of candidate clones is divided into the following steps:

1. Construction of the inverted index.

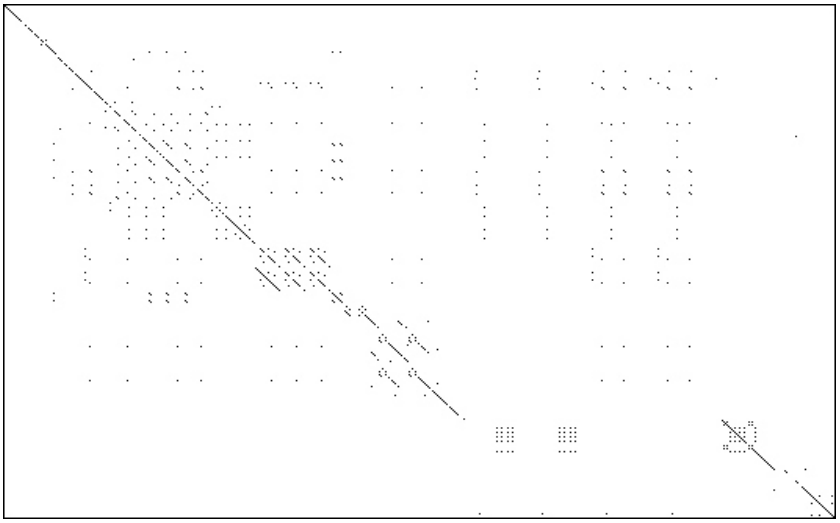


Figure 4.1: A sample dotplot of a comparison matrix between two source files. In our display convention, the coordinate *(1,1)* is always at the upper left corner of the matrix.

2. Construction of the comparison matrix.
3. Analysis of the comparison matrix and extraction of clone candidates.
4. Construction of clone classes.

In the following sections we detail these individual steps and mention briefly some implementation issues.

Step 1: Construction of the Inverted Index

We consider the code of a system to be a text written using the terms of a large alphabet. Each comparison unit is such a term. The vocabulary collects all unique terms that occur in the text. An entry in the vocabulary acts a representative for the equivalence class of all identical terms. This enables a major reduction of the number of comparisons that must be performed: Each new comparison unit must only be compared with all unique terms instead of all other terms in the systems source text.

The list of occurrences is filled with *postings* for each term from the input, recording its specific location.

Implementation Notes: The data structure for the vocabulary is most heavily used during the insertion of all comparison units. Once the entire system has been read in the postings containers are operated on individually. A good candidate for the data structure is a chained hash table, as they are performing best for the accumulation of large in-memory vocabularies [ZHW01].

Step 2: Construction of the Comparison Matrix

The postings for each vocabulary term represent all identical comparison units. By combining the coordinates of all these postings into pairs we create the atomic matches. In order to find matches longer than a single comparison unit we must now reinstate the order between the comparison units. We do this by transferring the atomic matches into a matrix. We can visualize such a matrix with a dotplot (see Figure 4.1). The human eye immediately discerns a number of patterns in the image, which we automatically detect and interpret in the next step of the algorithm.

When using exact string matching we are confronted with the problem of redundancy. If comparison unit c is identical with c', and c' is the same as c'', then transitively c is identical with c'' as well. In general, for n identical comparison units we get

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

atomic matches in the matrix. For example, 947 comparison units containing the text `msg_dt(m);` create 447931 matches. For large systems the number of atomic matches can be overwhelming as can be seen from the few examples:

System	*Lines*	*Atomic Matches*
BISON	7432	62,454
AGREP	9490	480,842
MFC	76,166	4,722,096
MAIL SORTING	113,517	80,568,528
GCC 3.3	713,262	3,648,389,733

We have the following options to optimize storage requirements:

Ignoring Uninteresting Comparison Units: The formula which computes the number of atomic matches is useful for deciding, in conjunction with the *weight* of the comparison unit, if we want to store its atomic matches in the matrix. If we deem the expense of time and space not justified we can simply ignore this comparison unit [Chu93].

Working in Parallel: The task of detecting duplicated code among a set of source units can easily be parallelized. For a vector $su = (su_1, su_2 \ldots su_n)$ of source units, we let a number of detector instances work independently, each comparing a range $(su_i \ldots su_j)$ against all of su.

Processing the System in Chunks: For each single detector instance we can partition the postings containers in *New* and *Old* compartments. All postings are first stored in *New*. At certain points in time we generate the atomic matches for all pairs that can be created among postings from *New*, and between postings from *New* and postings from *Old*. Once all the matches have been inserted in comparison matrices, the matrix analysis (see next algorithm step) is performed, after which the matrices are removed again from memory. Each *New* compartment is then emptied and its contents are added to the corresponding *Old*. The *Old* compartment then contains all postings for which the atomic matches have already been seen. We can then read the next source unit in, storing its postings again in *New*. If we order the source units by size, starting with the biggest one (or the one producing the most atomic matches, which we can know from a dry run), we make sure that towards the end, when the vocabulary and the *Old* containers become very full, the *New* compartments will contain less and less postings, thus again optimizing the number of atomic matches that have to kept in memory at any point in time.

Implementation Notes: Comparison matrices are usually populated sparingly with matches. Sparse matrices are the data structure of choice for storing this kind of data. Single linked lists as shown in Figure 4.2 can be used to reduce the overhead per cell. There are multiple possibilities for linking the individual cells. The upper variant in Figure 4.2 stores the populated cells in columns, whereas the lower variant stores them in diagonals. The diagonal representation makes diagonal traversal very easy, a method that is needed for the matrix scanning operation in step 3 of the algorithm (see below). The column organization on the other hand optimizes access of rectangular areas of the matrix. This form of access is needed when displaying the matrix as a dotplot in an interactive GUI. If the number of access from the GUI outweighs the scanning accesses, the column representation is the preferable one. Diagonal traversals on the rectangular representation are realized with a specialized iterator.

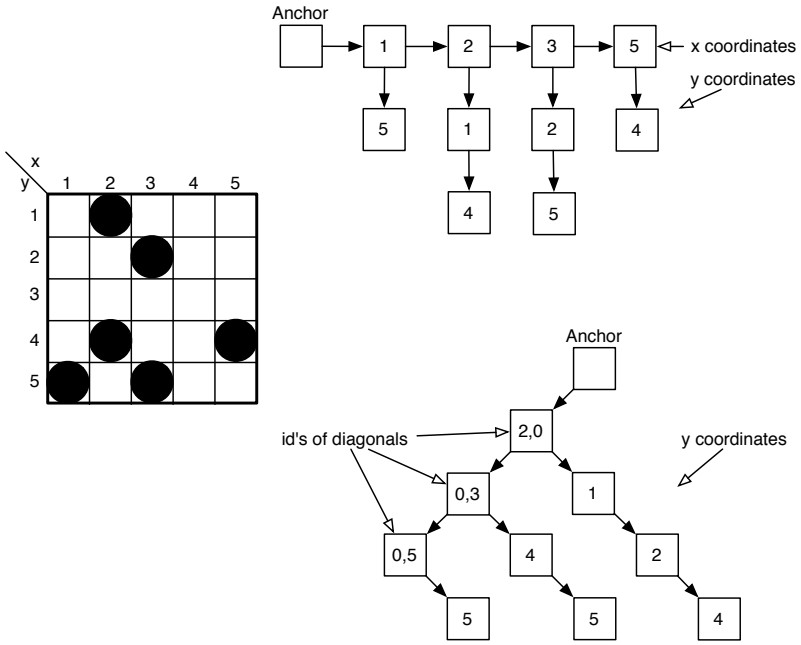


Figure 4.2: Two sparse matrix representations of the dotplot at left: The upper version is useful for accesses to rectangular areas of the matrix, the lower variant is optimized for diagonal traversals.

Step 3: Analysis of the Comparison Matrix

A single duplicated line of source code, represented by a single dot in the plot, is nothing to be worried about and thus not very interesting. What we are really interested in are the configurations of dots, *e.g.*, sequences of copied source lines. These *comparison sequences* form patterns which can be interpreted in terms of copied source text.

We interpret the following standard dot configurations [Hel95] as instance of copied code in the following ways:

a) Diagonals of dots indicate copied sequences of source code (Figure 4.3 *a)*).

b) Sequences that exhibit holes or gaps indicate that portions of a duplicated fragment have been changed (Figure 4.3 *b)*). For example, if we do not normalize variable names in preprocessing, a change of identifiers will result in such patterns.

c) Broken sequences with lower parts shifted indicate that a new portion of code has been *inserted* or *removed* (Figure 4.3 *c)*, *above* or *below* the main diagonal, respectively).

d) Rectangular configurations indicate periodic occurrences of the same code (Figure 4.3 *d)*).
An example is the `break;` at the end of the individual `cases` in a C/C++ `switch` statement where the code in the corresponding cases does not match.

We detect some of these dot configurations automatically with simple algorithms.

Identification of Diagonals and Diagonals with Gaps: The identification is done in a straight forward way, locating any free dot in the comparison matrix and stepping from there in a 45^o angle downwards to the right. The last dot in this range marks the end of the diagonal. If the process accepts adjacent empty cells up to a certain number, we get the diagonals with gaps. The procedure described here ensures that we record always

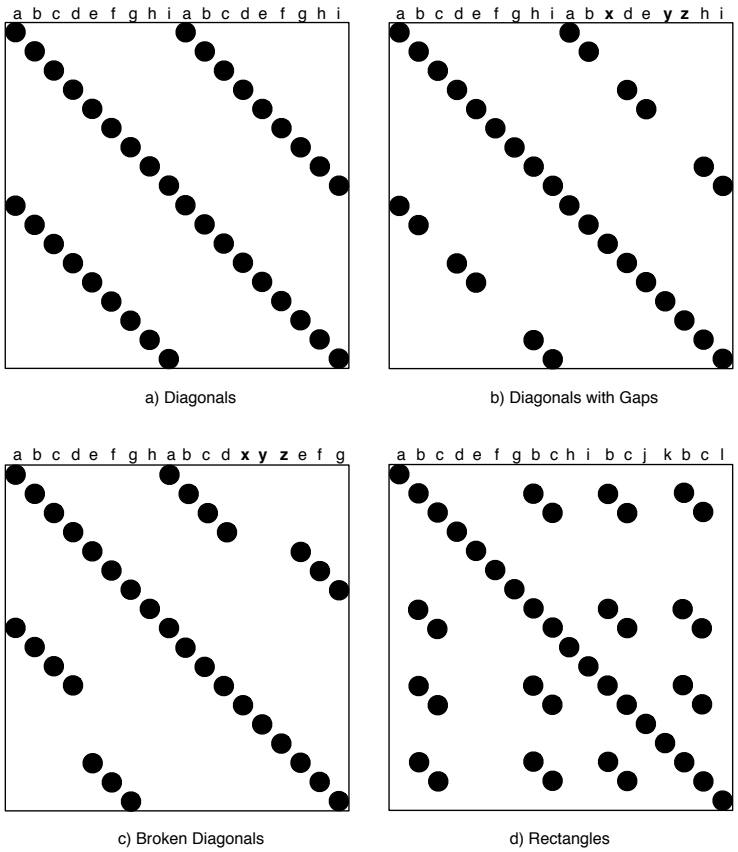


Figure 4.3: Different Configurations of dots. The symmetry of the diagrams stems from the fact that we compare the string to itself which leads to the characteristic middle diagonal where each line matches itself.

the longest match. All shorter matches are subsumed in the maximal one. This diagonal-traverse method has already been used by White et al. [WHHE84] for locating similarities in DNA sequences.

Identification of Broken Diagonals Ueda et al. [UKKI02b] have proposed a process to identify broken-diagonal clones—which they call *gapped clones*—based on user intervention. From the list of all non-gapped clones they automatically compute neighborhoods of clones, that is clones which end and start within a certain distance of each other, *i.e.*, the pieces of a broken diagonal. These neighboring pieces are presented in dotplots to the user who can then pick the combination that seems to be the best interpretation of copying and the editing actions. The advantage, they claim, is a $O(n \log n)$ time complexity versus $O(n^2)$ which a fully automatic approach would take. The same interactive way of stringing together partial diagonals has been proposed by Sonnhammer&Durbin [SD95].

Melamed, working on dotplots of bitexts (a correspondence between a natural language text and its translation into another language), exploits the geometric properties of partial alignments, like for example the constant slope, to build a greedy algorithm which constrains the search space for the next part of the alignment [Mel96b]. He however has the advantage of knowing that a single alignment path exists for the two texts, something that is not known for arbitrarily repeated fragments of source code.

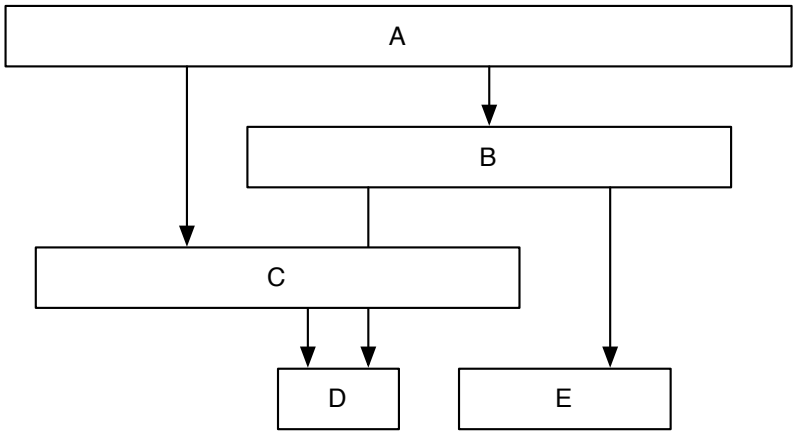


Figure 4.4: A hierarchy constructed from overlapping source fragments. Each level contains only fragments that are fully contained within its own boundaries.

Step 4: Construction of Clone Classes

Each dot configuration means a pairing of two source fragments. What we extract here are thus clone *pairs* only. Once all the clone pairs have been extracted, we want to reduce the amount of data in an aggregation step. From the set of all clone pairs we want to build clusters of code fragments that belong together, either because of common locations in the source text or because we have found a clone relation between them. A cluster of code fragments determines a clone class. We build such a cluster in two steps:

1. Aggregate all duplicated source fragments that include or overlap each other. This clusters source fragments purely according to their relative positions within the source text (see Figure 4.4).
2. We cluster the fragments that do not have a common location by their clone relations. We propagate the clone relations from the smaller fragment to their containers in the source hierarchy.

Note that since many of the clone candidates retrieved in step 3 will turn out to be false positives, the ranking and filtering of Section 4.4 should actually be done *before* aggregating clones into higher level entities. Since the discussion of the ranking techniques takes more space than explaining how clone classes are built we have chosen to present it out of order in the next section.

Implementation Notes: Each clone pair is formed by two source fragments which can be distinguished by the comparison matrix axis they belong to, *i.e.*, they are the *vertical* or the *horizontal* fragment. We split all clones into these two fragments and build two indices, one for the horizontal and one for the vertical axis, by aggregating containing and overlapping source fragments into *source fragment hierarchies* according to the definitions of Section 3.7.1. Each level in the fragment hierarchy links to a sorted collection of only its direct children. Insertion times are thus kept minimal.

In the second step of the clone class construction, the clone relation between two source fragments is propagated upwards in the two source fragment hierarchies they belong to. This propagation must be limited. It does not make sense to transfer the clone relation of a five line clone to the entire 1000 line file it is contained in. A reasonable range for which a clone relation can be transferred seems to be 500%. This guarantees that each propagated clone relation contains at least 20% of real copied code. For fragment hierarchies where the span of the code size from bottom to top is greater than this range, we create separate clone classes.

The construction of the highest level clone aggregation, the *clone class family* (see Section 3.7.2 on Page 41), is achieved through a simple sorting of all clone classes by the set of source units the clone class members are found in.

4.3.3 Time Complexity of the Algorithm

In this section we present a brief overview of the time complexity of our algorithm. We will not do a thorough formal analysis.

Step 1: The construction of the index depends on the length of the input text, n, both in time and space.

Step 2: The comparison matrix construction depends on the number of distinct atomic matches, *i.e.*, the number of distinct pairs of matching lines.

Step 3: The extraction of patterns from the comparison matrix depends on the number of atomic matches.

Step 4: The construction of the clone classes depends on the number of clones.

If we implement an *n-to-n* algorithm naively we achieve a best case of $O(n^2)$. The worst case is however $O(n^2)$ for every algorithm since we can construct cases where the number of matches is quadratic in the size of the input:

> For example, consider strings of the form $(0a1a)^i0$, of length $4i + 1$. In this case the i substrings a between 0 and 1 are maximal matches for the i substrings a between 1 and 0, contributing i^2 maximal matches to the total. [Bak93a]

4.3.4 Discussion

The *Adaptability* goal has been pursued in the transformation step. In the comparison step we are focusing solely on the *Scalability* goal. The major problem of this phase is the *n-to-n* nature of detecting self-similarity. To alleviate this we have selected a text index and the very simple comparison function of exact string matching.

From the set of atomic matches that the comparison creates we have to extract candidate clones by lining up the atomic matches in a matrix and extracting the longest diagonal match configurations. This process requires most of the space and must eventually be optimized with parallel or incremental organizations of the general algorithm.

The implementation overhead of all the methods proposed for this phase is of medium complexity with the exception of the sparse matrix implementation. The technique can therefore still be claimed *simple*.

4.4 Ranking and Filtering of Clone Candidates

At this stage we have a number of clone candidates that have been found to be similar in the comparison stage. For comparison methods like ours which are characterized by low precision and high recall this list can be very long and can still contain many false positives. Manually investigating them all can become very costly. The goal of the ranking and filtering phase is to identify clones that for the reengineer are either very interesting, or not interesting at all. The question that is addressed by ranking is the one of the *relevance* of a clone candidate. As we have seen in Section 3.1.3 relevance is to be determined relative to the task that we want to perform with the detected duplication.

Let us clarify the notions of ranking and filtering first.

Ranking means to order the clones according to a certain notion of relevance [WL03]. Ideally a reengineer could then work through the list and eventually define the cutoff point where tackling the duplication is no longer worth the effort.

Filtering means to remove clone candidates that are (most likely) false positives, reducing the amount of data to be checked by the user. Filtering is basically ranking with a defined threshold below which all candidates are eliminated automatically.

Ranking and filtering must be performed using information different from the one used by the comparison method. Such kind of information is usually more detailed than what the comparison method works with and thus more expensive to extract and to handle. Whereas the comparison method should be kept simple since it must be applied to the entire source text, the subsequent analysis methods can be more elaborated and costly since they are applied to the set of retrieved clone candidates which usually is considerably smaller than the entire source.

In what follows we will first describe the kind of information that in general is available to our approach for ranking and filtering purposes (Section 4.4.1). We will then describe the tasks for which we want to have relevance measures (Section 4.4.2), and afterwards detail the measures that we use to rank clone candidates (Section 4.4.3). We will finally discuss how these measures can be combined to form effective rankings and filters for clone candidates (Section 4.4.4).

4.4.1 Source Code Characteristics for Ranking

Deep code analysis methods involve abstract syntax trees or control- and data-flow graphs. This kind of knowledge would enable us to pinpoint the exact syntax elements which cause similarities and differences of a clone pair, facilitating the precise assessment of refactoring opportunities [BMD+00] or even the automatic extraction of procedures [KH02]. If we refrain from parsing the code fully we have to rely on information that can be gained with simpler means. This information is either produced during the execution of the various algorithms, or it can be extracted with lexical analysis from the source code, or it can result from the application of basic (UNIX) tools. We are giving here an exhaustive list of the information sources that are at our disposal:

Characteristics of the Source Code: With lexical analysis we can extract the following data from the code:

- Number of lines, tokens, and characters.
- Complexity of the code measured by simple source measures, *e.g.*, number of keywords, variables, operators, and nesting levels.
- Block boundaries in the source code, *e.g.*, the boundaries of functions or complex statements.
- Location of the code, *i.e.*, occurring within a function body, or a data structure definition.
- Names of variables and functions referenced in the code.

Note that in the presence of preprocessor directives some informations like the boundaries of blocks might not be extracted easily.

Characteristics with Respect to the Entire Source: The vocabulary gives us the frequency of every line in the source text.

Characteristics with Respect to the Normalization Measures: The mappings of variable names and constants to generic names (see Section 4.2.3) can be compared between two clone instances [Bak92]. Inconsistent mappings are hints at false positives.

Characteristics with Respect to Gaps: The gaps which break the diagonals apart (Figure 4.3 on Page 65) represent unforeseen differences between fragments which have not been removed by the normalization measures. The differences can be very small (a single character is able to make the string match fail), or they can indicate a mismatch between completely different code fragments. These characteristics influence the ranking:

- Size and number of gaps.
- (Edit) Distance between the differing fragments in the gap.
- Weight of matching code versus weight of differing code.

Characteristics of the Clone: The copied code can be investigated with respect to its occurrence in other clones.

- Number of clone instances in which the fragment appears.
- Distance between the locations of the shared source fragments of a clone, *e.g.*, same file, same directory, or different subsystem [KKI02][KG03].
- Relative positions of co-located copied fragments. If more than one clone covers a region in the source text, the two regions are probably related beyond what is evidenced by the duplicated fragments alone (see also the definition of clone class families in Section 3.7.2 on Page 41).
- Relative position of the two copied source fragments. Clones which relate two overlapping source fragments can be excluded.

An Abstract Representation for Source Code

In addition to the characteristics described above, we need a simplified representation of code which helps us to derive estimates for the complexity of the code and to approximately compare lines. Similar to the normalizations this *description* must abstract from peculiarities like user-chosen identifier names. To accommodate size measures, the elements of the description must be easily counted.

We describe a code fragment, *e.g.*, a line l, as a bag of features *feat*(l), where a feature is a code element like a keyword, a function call, or an operation.[4] We do not include representations for variables or constants (as they are implicitly represented by operations) and we do not represent delimiters for expressions or statements. Note that *feat*(l) is not a set as multiple elements of the same type can occur in l.

Some examples of C-fragments and their representations can be seen in the following table:

Source Line l	*feat*(l)
`for(i=1;i<maxhash;i++){`	`KW:for OP:= OP:< OP:++`
`if(i==j){`	`KW:if OP:==`
`tc_patptr[tc_hash[hash]]=pattern;`	`OP:= SUBS:[] SUBS:[]`
`tc_f_prep1(pat_index,pattern)`	`FUNCTIONNAME OP:,`

We can transform source code into this representation with lexical means similar to the transformation measures we have employed for the normalization of the source code. To configure the analyzer for a specific programming language, a list of its keywords and its operators is sufficient.

[4] Some operations like the reference operator in C can be considered too unimportant to be represented.

4.4.2 Relevant Tasks for Ranking

Clone relevance must be determined with respect to a task that the reengineer wants to perform. We present a list of tasks inspired by Li et al. [LYW03] and constrained by the kind of support our information sources can give us. Each task description is supplied with a description of the code and clone properties that we can employ to select clones relevant for this task.

Refactor Duplication and Eliminate Redundancies. Refactoring the duplicated code is a central concern of duplication reengineering. To determine the refactorability of a clone to all possible extent, deep code analysis is necessary. Without parsing we can only assume that clones that surpass a certain size threshold should be investigated for refactoring opportunities. The following hints can be used to assess the refactorability of a clone candidate:

- The amount of copied code is an important indicator if a clone should be refactored. This includes the weight of the copied code, a low gap density, or small distance of the fragments in the gaps.
- A high frequency of the copied lines in the entire source text indicates that the copied code is common and uninteresting, a low frequency indicates that it is surprising and thus more interesting.
- Code that is shared over near distances, *e.g.*, both fragments are located in the same file, is usually more easy to refactor than code that is copied over directory boundaries.
- A clone that is *syntactically cohesive* [HUK+02], *i.e.*, its code aligns with a syntactic block like a function or loop body, is easier to refactor than duplication which crosses these boundaries. This property can tell us that an entirely copied function of two lines is important, whereas a two lines clone appearing in the middle of a function is more likely to be irrelevant.
- If we find a clone class that has many instances, it might be advantageous to extract the functionality into a separate component.

Document the Existence of Duplication. We can help to keep copied fragments synchronized by including comments which make future maintainers aware of the shared code. This task is a consequence of the inability to eliminate a clone which has been selected for refactoring and the same indications are valid as for this first task.

Extraction of Reusable Components. A component, which we take here to mean a set of mid-level source entities, *e.g.*, functions that are semantically related, is a candidate for being made into an explicitely reusable entity if it is cloned to a large extent. To support this task we must be able to aggregate the amount of copied code located within the boundaries of a region of the code, *e.g.*, a file.

Understand the Program via Distributed Aspects. If we know that certain source fragments embody a specific functionality, everyone of its copies gives us some information about its surrounding code.

For this kind of analysis we need to know the content or semantics of a clone, and then all the places where it has been copied to.

4.4.3 Measures for Ranking Clones

In this section we define a number of measures with which to rank clone candidates. Each measure individually describes a single aspect of a clone. Some of the first measures in the list will be used in later measures. The measures concerning the gaps and clone dissimilarity will be needed to assess the similarity of clones which contain gaps.

Code Weight: Determines the importance of a piece of copied code.

Gap Difference: Determines in detail how dissimilar the two lines in a gap are. Instead of a boolean value we compute a percentage value for the similarity.

Gap Weight: Determine how much a gap detracts from the similarity of the copied fragments. Added to the weight of the matched code we get a single weight number for a clone, with or without gaps.

Fragment Distance: Determines the likely effort for the extraction of shared code.

Clone Frequency: Determines the gain of extracting shared code.

These simple measures are combined into task relevant rankings in Section 4.4.4.

Code Weight

One of the most important measures is the size of a clone. Whereas clones consisting in a single line will likely be the result of accidental duplication, a copy of 50 lines will most certainly mean that deliberate duplication has been going on. The size of a clone can however not simply be equated to the number of lines of the participating source fragments because lines of source code can differ vastly in length and content. The length of the line as well as the amount of logic that is found on it must therefore determine how much weight is given to the fact that the line is copied. In order to determine the *mass* of a fragment we need to define a measure, which we call *code weight measure*.

A measure which takes into account the length of the code in terms of characters would already improve on the number of lines measure. For our measure we will however count the tokens of the matching code, or, similar, the number of features in the feature representation introduced above. This gives us the weight of the following source fragment:

Source Line l	*weight*(l)
`m = msg_ct(cat,title_m,"Start system");`	4
`cmd->addTranslation("title",msg_text(m));`	5
`msg_dt (m);`	1
Total Code Weight:	10

To give the measure some kind of code understanding we can weigh the different feature types according to some criterion. If we are of the opinion that code complexity is determined by the adherence to structured programming rules, we could for example give a **`goto`** statement double the weight of an **`if`** statement because it violates those rules. The drawback is of this idea is that it requires assigning a weight to all operators and keywords in a consistent manner.

We can however make the measure more expressive without intervention of the user by using the frequency of the source lines as indication of how interesting their occurrence in a given fragment is. In information theory this is measured by the negative logarithm of the probability of the statement occurring, *i.e.*, $I(l) = -\log P(l)$ [Lin98]. We assume, for simplicity reasons, that the probability of a line l is independent from its context, *i.e.*, $P(l) = \frac{\textit{frequency}(l)}{n}$ where n is the total number of lines in the input. Weighing each line by its information content we get the following numbers for the above code fragment (example frequencies are taken from an actual file in the MAIL SORTING system):

Source Line l	**Freq**	*weight*(l)
`m = msg_ct(cat,title_m,"Start system");`	1	34.4
`cmd->addTranslation("title",msg_text(m));`	876	9.1
`msg_dt (m);`	947	1.8
Total Code Weight:		45.3

With this scheme the weight of a code fragment depends on the contents of the entire system's code. For targeted investigations of specific subsystems, the frequency of a source line should be counted only in the source files which contain at least one of the clone instances.

Gap Difference

Gaps are the result of mismatches between two lines among a number of matching lines. A mismatch can occur for two reasons: *i)* It can be due to a small difference that was not anticipated by the transformation measures. Such differences do usually not detract from the relevance of a clone. *ii)* It can also be the result of the two lines being very much different, which as a consequence might mean that the surrounding matching lines are only accidentally the same and the clone therefore irrelevant. A distance measure, *i.e.*, the result of an approximative matching, between the two corresponding lines is able to distinguish the two cases. Using an approximative matching technique at this point of the process has a significantly less severe impact on the overall time complexity of the comparison than using such a device in the main comparison since it is only applied to parts of the retrieved clone candidates.

We use the similarity measure proposed by Lin [Lin98]. He measures the similarity of two entities A and B by the ratio between the amount of information needed to state the commonality of A and B and the information needed to fully describe A and B .

$$sim(A,B) = \frac{\log P(common(A,B))}{\log P(description(A,B))}$$

The *description* of a source fragment is the list of all its features. The commonality $common(A,B)$ computes the intersection of the two descriptions. $-\log P(s)$ is again the information content of proposition s.

To exemplify how *sim* works we order a number of source lines according to the value of *sim*. We use a fixed set of (normalized) lines of C code and we alternately use various members as the target line to which all other lines are compared. In the following tables, the targeted line is put fully in bold at the top, and the other lines are listed according to their rank:

sim	**if(p+1>=p[p])**	*sim*	**foo(p,p)**	*sim*	**p=p[1]-p**
0.57	`if(p>=p) break`	1.00	`foo(p,"...")`	0.79	`p=p-1`
0.54	`p=p+p[p]`	0.82	`foo(p,"...",s)`	0.68	`p=p+p[p]`
0.29	`p=p[1]-p`	0.65	`if(p) foo(p,p,p)`	0.53	`p=p+p-p+p+p`
0.28	`if(p) foo(p,p,p)`	0	`p=p+p-p+p+p`	0.29	`if(p+1>=p[p]) {`
0.19	`p=p+p-p+p+p`	0	`p+=p`	0	`p+=p`
0	`p+=p`	0	`p=p+p[p]`	0	`if(p) foo(p,p,p)`
0	`p=p-1`	0	`p=p-1`	0	`foo(p,"...",s)`
0	`foo(p,"...",s)`	0	`if(p>=p) break`	0	`if(p>=p) break`
0	`p++`	0	`p=p[1]-p`	0	`p++`
0	`foo(p,p)`	0	`p++`	0	`foo(p,p)`

As another example for *sim*, we list the 10 lines in the AGREP system that are closest to an arbitrarily chosen line (see Table 4.1).

Gap Weight

Not only the distance between gap lines in a clone is important but their number and weight as well. In a gapped clone we must contrast the weight of the lines that have matched to the weight of the code that did not match. In the two fragments of the example below, which are retrieved as clone candidate by a pattern matcher that accepts diagonals with one dot gaps, only the boilerplate lines of the **`case`** statement match, whereas the code which contains the interesting logic does not match at all.

sim	**if((mfp==-1)&&((mbuf==null)||(mlen<=0)))return-1;**
0.701	**if**(temp==null||push(&stk,temp)==null) **return** null;
0.699	**if**((token=='...')||(token=='...')) **return** n;
0.699	**if**(s==null||node==null) **return** null;
0.699	**if**((cur=='...')||(cur=='...')) **return** 1;
0.693	**else if**((ret==0)&&(mfd==-1)&&(mbuf==null)) **return** -1;
0.657	**if**((fpos==null)||(*fpos==null)) **return**;
0.657	**if**(s==null||*s==null) **return** null;
0.654	**if**(ret<=-1) **return**-1;
0.634	**if**(!aparse&&((m=maskgen(pattern,d))==-1)) **return**-1;
0.618	**if**(((limitoutput>0)&&(limitoutput<=num_of_matched))||

Table 4.1: The top ten most similar lines to the source fragment set in bold at the top (from the AGREP system).

```
case 1:
  v_mangle_res(str_c(""));
  break;
case 3:
  v_mangle_res(str_c(","));
  break;
case 8:
  v_mangle_res(vsp[0].lv);
  break;
```

```
case 13:
  list_ctor(&parse_val.list);
  break;
case 14:
  parse_val.list = vsp.list;
  break;
case 15:
  parse_val.expr = 0;
  break;
```

To balance the matching and mismatching lines of a clone candidate we subtract the weight of the code in the gaps from the weight of the matching code. We distinguish between the larger and the smaller weight of two lines (k, l) forming a gap:

$$\begin{aligned} \mathit{maxw}(k,l) &= max(\mathit{weight}(k), \mathit{weight}(l)) \\ \mathit{minw}(k,l) &= min(\mathit{weight}(k), \mathit{weight}(l)) \end{aligned}$$

To compute the weight of a gap we must consider two possibilities: When *sim* does not find any similarity between the lines of the gap, we remove $\mathit{maxw}(k,l)$ from the sum of the measure. When *sim* finds some similarity between the lines, we multiply $\mathit{minw}(k,l)$ with the similarity value, adding to the weight of the gap. The total weight of all gaps of clone candidate c is computed as follows:

$$\mathit{gw}(c) = \sum_{(k,l)\in \mathit{gaps}(c)} \begin{cases} -\mathit{maxw}(k,l), & \text{if } \mathit{sim}(k,l) = 0 \\ \mathit{minw}(k,l) \cdot \mathit{sim}(k,l), & \text{if } \mathit{sim}(k,l) > 0 \end{cases}$$

In fact, this formula can be applied to every line of the copied fragments. For matching lines, $\mathit{weight}(k) = \mathit{weight}(l)$ and $\mathit{sim}(k,l) = 1$. This measure therefore gives us a single number for any clone, gapped or not.

Fragment Distance

The two source fragments participating in a clone pair can be related via the files they are a part of. The distances between the two locations can be represented either as an ordinal scale, *e.g.*, *same file*, *same directory*, *different directory*, which leads to a taxonomy for clones [KCS02]. Another way is to define a measure as the length of the path from the files to the lowest common ancestor [KKI02]. A similar measure can also be defined for object-oriented code which can be located within a class hierarchy.

We can extract this information by simply keeping book about the origins of each line in the code.

Clone Frequency

The frequency of a clone determines its importance. The more times a source fragment is copied, the more reduction can be achieved with a refactoring that unifies all instances. Even small fragments if repeated many times warrant replacement by a macro.

The clone frequency can be determined from the size of the clone classes and clone class families as defined in Section 4.3.2.

4.4.4 Filters to Remove False Positives

The measures that we have introduced in the preceeding section can be combined to better characterize clones which are relevant for one of the reengineering tasks introduced in Section 4.4.2. For each task we list the measures involved, and the general tendency of the measured values (we will only indicate if the values should be high or low. The definition of concrete thresholds must be done for a concrete example). A combination of fixed weights would probably not be independent of the system that is screened and therefore we only list the measures in order of importance giving a rough outline of a possible filter.

Refactor Duplication and Eliminate Redundancies:

Measure	*Value*
Code Weight	high
Gap Difference	low
Clone Frequency	high
Fragment Distance	low

The combination of the measures is due to the following reasonings. To be refactored, code should be of considerable weight (high *Code Weight*). Experience shows that customers who have their systems screened for duplicated code usually want the n largest clones removed from their code.[5] In the same vein, a high similarity (low *Gap Difference*) will make the refactoring of the clone easier. If the fragment has been copied many times a removal action has the chance to affect a large piece of the total amount of duplicated code (which is called the "high impact" category by Balazinska et al. [BMLK99]). Finally, the farther apart the fragments are found, the more difficult it is to refactor them because the context (the copied code *plus* its surroundings) that must be considered for a refactoring grows with the distance.

Document the Existence of Duplication:

Measure	*Value*
Fragment Distance	high
Gap Difference	middle
Clone Frequency	low
Code Weight	high

We have to document duplication if we do not have the possibility to extract the shared code. This means that we should first try to remove everything we can and only document the rest. For the clones that are documented instead of removed, a slightly different set of ranking criteria is important. Fragments that are close together do not need as much documentation as fragments that are fart apart, where the danger is greater that when changing only one the other gets forgotten (high *Fragment Distance*). Clones which have already experienced a diverging evolution are difficult to reconcile, and rather must be documented (middle *Gap Difference*). Even clones which have only two instances can encompass information which needs to be updated in synchronization (low *Clone Frequency*). That larger clones represent more opportunity and therefore greater dangers for diverging evolution (high *Code Weight*) is clear.

[5] Personal Communication by M. Mehlich of Semantic Designs, Inc. who sell professional duplication detection services.

Extraction of Reusable Components:

Measure	*Value*
Code Weight	high
Clone Frequency	high
Fragment Distance	–
Gap Difference	–

Reusable components must be of a certain size to make a function invocation worthwhile (high *Code Weight*). The duplicated code must be of a considerable frequency showing that they have proven their usefulness (high *Clone Frequency*). If fragments are found in different places of the code (*Fragment Distance*), we have probably found code that is applicable universally, but even code copied frequently in only a single subsystem can be a candidate for a reusable component. Whether cloned fragments encompass differences (*Gap Difference*) does not play that important a role, since a component that is used in many contexts will necessitate quite a bit of parameterization anyway.

Understand the Program via Distributed Aspects:

Measure	*Value*
Fragment Distance	high
Clone Frequency	high
Code Weight	high

Program understanding is a reverse engineering task and thus notably different from the previous ones which concentrated on reengineering issues. The most important criterion is the distance of the clone fragments (high *Fragment Distance*) and the large population (high *Clone Frequency*): Only code that is distributed over the entire system can be useful in helping to understand diverse system entities. To be able to explain the system a piece of copied code must represent a considerable amount of logic (high *Code Weight*).

4.4.5 Discussion

The combination of heuristic normalization methods (Section 4.2) and a simple comparison method (Section 4.3) can lead to a list of clone candidates which contains many false positives. In order to reduce this number, we have proposed a set of simple measures which can be computed with minimal investment in difficult-to-adapt parser technology.

Following the idea that the relevance of a clone depends on a specific reengineering task one wants to achieve, we have shown how these measures could be combined into filters which help to emphasize clones that are particularly interesting for a given reverse engineering task.

To better rank code fragments we need more and improved measures which can be computed from less than a full-fledged abstract syntax tree. A short list of measures that could be investigated on top of what we have proposed in this section follows:

- Control flow measures can be computed from a simplified syntax tree representation of the program. The necessary parser to identify the basic control flow of a piece of code could be built using configurable island grammars.
- The conformance of a fragment of copied code to the borders of syntactic structures like blocks or functions can be derived using a minimal parser which understands blocks. We can then identify the outermost syntactic block b which is coincident with any part of the copied fragment f. The amount of overlap between the entire b and f will be a good indicator of how much f conforms to syntactic boundaries as represented by blocks.

The measures that we have described above are used to rank clone pairs only. Since the engineer will be working with the clone sets *clone class* and *clone class family*, all the measures must be formulated for these aggregations. Especially the reengineering goal of extracting reusable components makes less sense if it is applied to clone pairs alone, because a function, for example, may be covered by more than a single duplicated fragment.

The code representation that we have introduced in Section 4.4.1 can be used as input transformation measure, before the comparison. The question which should be investigated is then if such a normalized representation will generate too much noise and overshadow the increased recall.

4.5 Conclusions

We have selected string matching as basic comparison function. Strings are the format in which the source code is already from the start, which fosters *adaptability*. String comparison is a simple function which helps to achieve *scalability*. Any duplication detection mechanism has to face the two challenges of *i)* how we can recognize duplicated code despite superficial variability in the copied code, and *ii)* how we can handle the n-to-n character of the self-similarity detection problem. The choice of string matching leads to the following answers to these challenges:

i) Since the comparison function is not sensitive to the structure of source code we therefore must

- preprocess the code in order to increase recall: we need to normalize certain volatile syntax elements.
- identify copied fragments despite larger differences that have been applied to their middle parts.
- analyze the detected clones to weed out the false positives that are due to the normalization of the code.

ii) To improve upon a naive comparison of everything against everything we organize the comparison around an index.

In the normalization and post-comparison analysis we use multiple minimal parsers for different aspects of the source code. These parsers only recognize superficial code structure, but we are capable of keeping the infrastructure simple enough so that it is *configurable* for many programming languages.

The are always differences between fragments of code which we cannot normalize. We therefore allow our matching fragments to contain gaps of non-matching code. Changes in the copied code which exceed our normalization capabilities do not hinder detection if they appear in the middle of a longer copied fragment.

The problems which result from the choice of string matching as comparison function are twofold:

- Some of the normalization measures tend to increase the number of false positives considerably. The question is then which normalization measures from the set of possible measures to apply for a given case study. The reengineer should be given some hints which help make this choice.
- Since the comparison resolves duplication on the level of a single line of code, there is a great number of atomic matches which has to be stored in the comparison matrix. This can be taxing in tight memory situations.

To cope with the high recall that our method entails, we propose measures for ranking clones (Code weight, Gap Difference, Fragment Distance, and Clone Frequency) that, when combined, characterize clones that are relevant for several reengineering tasks.

The drawbacks of choosing not to fully parse the source code are tied to the fact that information from deep code analysis is not present. With only superficial understanding of the code structure we are unable to detect similarities which are found in the semantic layers of the code. Instead of automatically evaluating clones our post-detection analysis will be largely in the hands of the engineer. In the next chapter we therefore investigate clone presentations which both in overview and in detail allow the engineer to access the clone information easily.

Chapter 5

Visualization of Duplication

The result of any code duplication technique is a clone pair, essentially a pair of token or line coordinates. With this information at hand, we can begin reengineering operations, automated or not, which are the ultimate goal of the endeavor. A natural question, then, is: Why do we want to visualize the duplication?

The human ability to recognize visual patterns is eminent. The field of *Information Visualization* investigates the exploitation of these capabilities for the purposes of understanding large amounts of data. According to Card, Mackinlay and Shneiderman, Information Visualization is *"the use of computer-supported, interactive, visual representation of data to amplify cognition"* [CMS99]. Andrews sums it up as being *"the visual presentation of abstract information spaces and structures to facilitate their rapid assimilation and understanding"* [And02]. Information visualization takes advantage of the enormous powers of the human visual system: *"The eye and the visual cortex of the brain form a massively parallel processor that provides the highest-bandwidth channel into human cognitive centers"* [War00]. Much of the processing is done *pre-attentively*, *i.e.*, before we consciously interpret the parts of an image. For example, the human visual system comprises, among others, a kind of cells called *simple cells* which are excited by lines that have a particular orientation.[1] The diagonal lines which are so easily spotted in a dotplot represent duplication and are thus directly relevant.

By visualizing similarity data we can shift a part of the mental burden we must exercise to understand to the built–in cognition machinery. The tasks of selecting, assessing, and prioritizing the data can be helped largely. We want to support the engineer in the understanding of the overall duplication situation as well as in the fine-grained analysis of concrete cases of duplication.

We take three different perspectives, from the detail (the source code) to the overview (the entire system):

Editors: The most fine-grained view on duplicated code is through an editor which presents copied fragments side by side. Since clones are frequently members of a clone class, a clone browser must make all related fragments accessible to foster the understanding of the situation (Section 5.1).

Dotplots: For regions in the source text containing many overlapping clones or clones which are split into pieces by many small changes, a *dotplot* can give insights faster than a mere listing of all clones. It is also a useful starting point for interactive exploration of the duplication situation. While still on a low level it is possible to gain information about the global duplication situation from a dotplot (Section 5.3).

Since the dotplot is a well known technique developed originally for similarity analysis of DNA sequences, we will outline its history and the surprisingly many applications it finds in a variety of fields (Section 5.2).

Polymetric Views: For large systems where the amount of duplication data precludes individual assessment, we display aggregated quantitative information and relations between high level entities of the system using *polymetric views* that enable orientation and selection (Section 5.4).

[1] Work by Huber and Wiesel on the physiology of the nerve cells in the visual system of the brain, qtd. in [Gla02].

Note that visualizing clones is *not* an example of *software visualization* [SDBP98] in the truest sense of the word, despite the fact that we use program code as data. Software Visualization tries to make a program easier to understand, *i.e.*, it deals with the algorithms which are a *high-level* description of the program. We, however, are interested in the self similarity of the source text which is a *low level* description of the program. Knowledge about repetition of code fragments will contribute only indirectly to the understanding of the program. Rather than visualization of software dotplots and polymetric views are *data visualization*.

Our Contribution

The contributions of this chapter are:

- A comprehensive list of requirements for clone browsers.
- A history of the origins of the dotplot.
- A new approach for the visualization of quantitative duplication data in the context of the system it was found in.

5.1 Browsing Duplicated Code

If duplicated code is not refactored automatically, the final assessment of a clone instance can only be made by looking at the source code in an editor. It is thus important to equip code editors with support that lets programmers investigate clones easily.

The manual investigation of copied code can be supported in the aspects of *i)* access to the clones and clone classes or access to the regions which have been partially copied, *ii)* special features of the editors, and *iii)* presentation of the clones.

Access to the Clones: The clones should not only be ordered by the categories in which they are collected, but also by the clone classes they are aggregated in. All fragments of a clone class must be comparable since only then can we decide on a refactoring strategy. The user must be able to combine any two members of the clone for presentation in the browser window. He should be able to keep a single fragment fixed in one editor window and to scroll through the all the other clone class members in the second editor window.

Instead of focusing on the clones, the user should be able to select and browse regions of source code [KG04], *e.g.*, a function or a file. All the clones which are located in a given source region R should be displayed in the editor. The user must be able to browse through all regions which are related to R in the other editor window. The user should be aware of all clones contaiend within the region at once.

Finally, it should be possible to select clones via the dotplot visualization. Access to the source code must be possible by clicking on the diagonals in the plot.

Editor Enhancements: Clones must be shown in two adjacent browser windows. A horizontal arrangement is more convenient if the user wants to make line-by-line comparisons. The drawback of the horizontal arrangement is that lines will quickly extend beyond the right border of the editor. We need therefore to be able to switch to a vertical arrangement of the two editor windows. In the vertical position, it is easier to add more than two views and still see a reasonable part of the copied fragments.

The most important enhancement of a clone browser is however the synchronized scrolling of the two windows. The user should be able to view the copied fragments as a single entity. Especially for fragments which either horizontally or vertically extend beyond a single page, line-by-line comparison is only convenient if we can change the browser location in the two windows with a single mouse movement.

Clone Presentation: Whereas the user can understand clones which consist of entire functions by seeing only the copied function, clones which occur in the middle of a larger text must always be presented with their context so that their dependencies on the surrounding code can be gauged. The copied code must be highlighted to make it stand out to non-duplicated code. Differences of notice, *i.e.*, more than simple variable renaming, between the copied fragments must also be highlighted to raise the attention for elements that must be normalized in the clone. It would probably be convenient to implement navigation of only the differences in a clone, like the TAB-Key in dialog windows of graphical user interfaces. The user can then directly assess what keeps the code fragments from being unified.

If multiple overlapping clones must be highlighted at the same time, for example when browsing a source code region containing more than one clone, the standard typographic highlighting method of changing the background color must be replaced with methods which can accommodate more than a single clone in the same location. As an example we can use rectangular braces on the left border of the editor marking the beginning and the end of the clones.

Related Work: Current implementations of clone browsers, for example the GEMINI environment [UKKI02a], the EMACS clone browser by Bellon[2], and the source browser of DUPLOC [RD98], only implement a part of these features. CLONEDR [BYM+98] reports not only the copied fragments but also produces a unification of

[2] Available from `http://www.bauhaus-stuttgart.de/clones/` [May 15, 2005]

all instances, showing the text that is the same for all fragments interspersed with parameters that represent the differences.

The related source text visualization approaches of SEESOFT [ESEE92] and the Aspect Browser NEBOULOUS [GJK01] provide ideas which could be used in the presentation of clones as well. These tools shrink the text to a one pixel height without changing indentation. This keeps the familiar image of the code and enables to show large portions of the system on screen at once. Like in a *map* unneeded detail is abstracted away. By color-highlighting certain features of the code all over the map, the size, location and the location relationships between highlighted features can be assessed easily. If, for example, all source lines that belong to a clone class are highlighted, the source map immediately conveys the size of the affected code as well as the spread of the clones, *i.e.*, which subsystems are involved. This is the same way that a dotplot presents a map of the source code, indicating location information about the duplicated code 'aspect'.

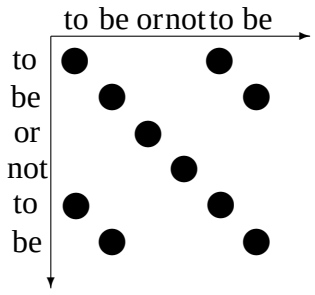
5.2 The Dotplot

A *dotplot* is a visual display of a matrix. A dotplot is not a *scatterplot*, albeit the two are similar. A scatterplot is a visualization tool to aid analysis of systematic relationships between two rational variables. The two values combined form a coordinate in a two-dimensional[3] grid. Any number of data points may fall into each cell of the grid. If too many data points are concentrated on a small area of the grid, *overplotting* will reduce the usefulness of the plot in visual analysis.

The dotplot is a variation of this schema. As dotplots visualize a matrix, they have discrete scales on both axes, *i.e.*, each axis is a vector, v and w. Each matrix cell at coordinate (i, j) contains the result of a function $f(v[i], w[j])$. In contrast to scatterplots, there is exactly one value for each cell in the grid which eliminates the problem of overplotting. The function f can be anything, but is mostly computing a kind of similarity, the simplest being exact equality. If we take string equality as an example for $f()$, the cell at coordinate (i, j) is set to 1 iff the elements $v[i]$ and $w[j]$ are equal strings. This results in the following matrix:

	to	**be**	**or**	**not**	**to**	**be**
to	1	0	0	0	1	0
be	0	1	0	0	0	1
or	0	0	1	0	0	0
not	0	0	0	1	0	0
to	1	0	0	0	1	0
be	0	1	0	0	0	1

This matrix is visualized straightforward by a dotplot as seen in the next figure, where each value 1 is plotted as a dot, and each cell with value 0 is left blank in the plot.



Note that placing the origin of the graph in the upper left corner is a convention we adopted from Church and Helfman [CH93]. For people who read text (and thus source code) from left to right and from top to bottom, a dotplot which has the same orientation makes it easy to correlate the image with the text.

[3] See [HHHW98] for the *32D Hypercube* visualization, an extension of scatterplots into three dimensions.

Note to the Reader: The following Sections 5.2.1 to 5.2.5 explain the origins and related work about the dotplot visualization. They are not necessary for the understanding of the use of dotplots for duplication detection. Readers which are uninterested in dotplot background can skip to Section 5.3 on Page 86.

5.2.1 Origins of the Dotplot

The dotplot was first proposed by A.J.Gibbs (a biologist) and G.A.McIntyre (a statistician) in 1970 [GM70]. The *diagonal match* or *diagram* method was introduced as a simple way ("can be done by hand if necessary") to find similarities between nucleotide or amino acid sequences. The idea for the comparison matrix which underlies the diagram visualization was apparently communicated by Saul Needleman to Walter Fitch in September 1965. Fitch himself published an early dynamic programming diagram in [Fit69]. Needleman&Wunsch used the matrix as basis for the well known algorithm for global sequence alignment [NW70]. The first step of their dynamic programming algorithms fills the comparison matrix with similarity values between corresponding nucleotides or amino acid residues. Instead of computing the global alignment, Gibbs&McIntyre then just visualize the comparison matrix as a dotplot.

The big difference between the diagram method and the alignment algorithms is that the former does not presuppose a single alignment path between the two sequences. A dotplot presents all possible alignment fragments of any granularity down to the single comparison unit, without selecting some of them as parts of a single path through the matrix. This is called *local alignment*. On the other hand, a *global alignment* always needs an a priori granularity definition, *i.e.*, the definition of a start and an end point, with the assumption that there is a single alignment path between the two. Any local alignments that do not lie on the main path will be ignored, which may preclude the detection of smaller but nevertheless important homologies. As Church&Helfman have said concerning DIFF like global alignment tools

> Such programs attempt to find a single match and are therefore unable, in principle, to find the richer texture of multiple overlapping matches. [CH93]

Additionally, computing the global alignment is an extra step over the computation of the comparison matrix. It is thus advisable to perform it only for sequences that have a good probability of actually yielding an alignment.[4]

5.2.2 Dotplot Applications in Molecular Biology

Global and local alignment analysis methods were invented in the context of research in molecular genetics. Two applications for the dotplot can be found in the literature:

- Analysis of DNA and protein homology, *i.e.*, the attempt to detect similarities between sequences in order to derive evolutionary relationships, or functional convergence among related genes or proteins. Alignment data for a family of genes or proteins is the basis for the construction of ancestral trees, *i.e.*, how through mutation and rearrangement one gene has evolved from the other.
- The search for secondary structure of the DNA molecule. Since two strands of DNA can be held together by weak bonds between the nucleotides adenine (A) and thymine (T), or between cytosine (C) and guanine (G), it is possible for parts of a molecule to fold in a particular manner.

For the homology analysis, the dotplot is applied to a regular comparison between a sequence on the horizontal and on the vertical axis. Gibbs&McIntyre [GM70] simply put a dot wherever two elements of the sequence matched. McLachlan [McL71] has employed a more sophisticated similarity function which reports partial similarity. Instead of just putting a dot at a matrix cell, then, four different symbols indicate how similar the

[4] Balazinska [Bal99] follows this principle by extracting candidate clones using a metrics-based approach and then determining fine-grained differences between candidate pairs with an alignment of the two token sequences that have already been proven similar.

two sequences are at this coordinate. The same has been achieved later with letters (Pustell&Kafatos [PK82]) and color (Reisner&Bucholtz [RB88]).

For the analysis of secondary structure, a nucleotide sequence is compared with its complimentary sequence ($G = C$,$A = T$). The diagonals in the dotplot can then be interpreted as places where the strand of nucleic bases can pair with itself; horizontal or vertical gaps representing loops or internal bulges of the protein structure. This has been used already by Gibbs&McIntyre [GM70]. Tinoco et al. [TUL71] have also estimated secondary structure. They were using a measure of the stability of the helical binding between two nucleic acids as the similarity function.

The important aspects of dotplot analysis of biological sequences are:

Similarity Function: The range of possible similarity functions is much larger than simple equality of the character symbols A,C,G, and T. For example, the probabilities of mutation, deletion or insertion of single bases within a sequence can be derived from sequences for which relations are known. These cost matrices are the basis for similarity functions.

Noise Reduction: Since DNA has a 4–symbol alphabet and proteins are commonly constructed from 20 different amino acids, the level of noise due to random matches is very high. To enable the visual detection of significant alignments, noise reduction is tantamount.

Statistical Significance: It is important to discern between chance matches, which due to the smallness of the alphabet have a considerably high probability, and biologically meaningful matches.

Display Scaling: The amount of sequenced DNA has increased over the course of the last 30 years. Today, the whole human genome—numbering 3 billion bases—has been sequenced completely. Methods and tools have to be implemented to display an entire matrix on a single screen and to zoom into smaller areas of interest on demand.

The currently most widely used homology detectors like BLAST [AGM$^+$90] employ a different approach: they first construct a list of sequences that are possible mutations of the words from original sequences, and then search instances of these mutations in the target sequence via pattern matching methods. Matches are combined to local alignments. Dotplots are only used for visualization purposes.

5.2.3 Dotplot Applications in Natural Language Processing

After a long career in the biological sciences, the dotplot has crossed over into the field of text processing, displaying similarities in natural language texts.

Church [Chu93] has used dotplots to align *bitexts*, *i.e.*, parallel text corpora. He used 37 million words of the Canadian Hansards which are parliamentary debates published in both English and French. The technique is based on *cognates*, *i.e.*, words that are spelled the same or similarly and have the same meaning in both languages. The matches of cognates between the two corpora form a trail in the dotplot which describes the alignment. This has advantages in that it does not require the paragraphs or sentences to be pre-aligned. It can also deal with noise, *e.g.*, artefacts from OCR, much better than previous techniques. The rough alignment produced by the dotplot is used as input for a second phase, which refine the *bitext map* using statistical methods (probabilities at the word level) [DCG93].

Melamed [Mel96b][Mel99] pursues the same goal of an accurate bitext map, relying exclusively on the information in the dotplot. A greedy search algorithm tries to detect the chain of dots of the bitext map. Relying on the fact that for a bitext map there is only a single valid chain in the entire dotplot, geometric properties of found chains, like their slope, are used to asses the validity of a partial chain extracted from the plot. In a similar application, a slope criterion for chains enables the identification of omissions in a translation [Mel96a].

Chang and Chen [CC97], too, use the image characteristics of dotplots to find bitext maps. Observing that non-literal translations with many deletions, or language combinations which do not provide many cognates, *e.g.*, English-Japanese, will quickly unsettle many bitext map detectors, their goal is to make the technique more robust. Interpreting the dotplots as binary images, they leverage a number of well known image processing methods: convolution (local edge detection), texture analysis (noise reduction), Hough transform (line detection). Since these image processing techniques work with gray-level images, which means that the grid cells of the dotplot can contain a similarity percentage.

Reynar [Rey94][Rey98] uses dotplots to identify topic boundaries in newspaper articles or other texts. The idea is based on the observation that the high number of words repeated within a topical text will lead to many matches in squares around the main diagonal of the dotplot, where a text is compared with itself. These areas will show up dark against their surroundings. The boundaries between topics can now be determined by searching for such a region with maximal density.

Gershon et al. [GLW+95] count the number of times two words appear in close proximity in a corpus of text and enter this number in the matrix cell. After some clustering—which can be done easily since the order of the words is not relevant—a dotplot enables analysts to look at word correlations. The count of word proximities can be seen as a kind of similarity metric. The view generated by this method is symmetric.

5.2.4 Dotplot Applications in other Domains

The connectivity (or adjacency) matrix is an alternative for the node-link representation of graphs. The rows and columns of the matrix represent the vertices of the graph. The connection between two vertices is represented by a *true* value in the cell at the intersection of the corresponding row and column (the value can of course indicate any edge-weight). The advantage of the adjacency matrix is that thanks to the fixed grid structure it does not become cluttered, a frequent problem for graphs with many links. Consequently, Ghoniem et al. [GFC04] have found that as the size of a graph increases, the readability of the connectivity matrix is better than the one of a nodes-and-edges drawing for almost all basic reading tasks (estimating nodes and edges count, finding nodes and edges, finding paths). The connectivity matrix is used often for large scale graph visualizations as can be seen in the following (not exhaustive) list of examples:

- Becker et al. [BEW95] use a matrix to visualize traffic between network nodes. Each node is placed on the vertical and the horizontal axis. The matrix is not symmetric: the cell (i, j) contains the amount of traffic flowing from node i to node j whereas cell (j, i) contains the traffic in the other direction, *i.e.*, from node j to node i. The dotplot complements a geographical map where network nodes are placed in their correct locations. The map visualization is in danger of overplotting when numerous nodes and their connections are all shown at the same time. The abstracted dotplot view gives equal emphasis to all network nodes. A problem not solved to satisfaction is, however, in the order in which the nodes should be arranged on the matrix axes in so that the resulting picture is easily interpreted.

- In building an Information Exploration System which lets users investigate large number of documents and their interrelationships, Hetzler et al. [HHHW98] have, among other techniques, implemented the *Connex* visualization. This is a 3D extension of dotplots where the axes of the matrix are occupied by the documents from the collection. The values of the grid cells are represented as colored beads. By stacking the beads one upon the other, a number of interrelationships between documents can be correlated. Examples of documents relationships visualized in a simple dotplot are “same author”, “same publication date”, “same topics”, or “same level of detail”. They also plot asymmetric relationships like “published before”.

- In software visualization, De Pauw et al. [PHKV93] employ dotplots to show the amount of communication that goes on between classes and methods. They also present dotplots to illustrate allocation relationships between classes or highlighting the use of inherited code in subclasses. Aebi [Aeb03] has extended this work, grouping classes into higher-level entities and recording usage relationships of different kinds between the classes. Displaying both types of entities, the groups and their constituents, on the matrix axes he shows the detailed and the abstracted view of the collaboration patterns at the same time. Like in the Connex view discussed above we have multiple dotplots stacked on top of each other. Also

> Ham [vH03] has recursively stacked adjacency matrices atop each other (using a hierarchical component structure) to visualize the call graph of a large software system.

Similar to the visualization of ordered sequences of elements (DNA sequences, lines of source code), Zou and Godfrey [ZG03] compare sequences of function names, ordered by the files in which they are defined, across different versions of a system. Their dotplot allows quick insights into functions moving from one file to the other, and source files being merged or split.

5.2.5 Categorization of Dotplot Applications

From the various dotplot applications presented above we are able to deduct a small categorization of this visualization technique. The following characteristics can be discerned:

Order of Data Elements: The order of the elements on the axes are either given by the data or can be changed in the process of visualization. Sequential data like words of a sentence or lines of a text will give the plot a fixed pattern. If the data elements can be rearranged, it may be necessary to do so in order to find the plot patterns that are most easily interpreted. The disadvantage in this lies in the work required on the part of the user.

Values of Matrix Cells: If the range of the function $f(v[i], w[j])$ is equal to the boolean set $[0, 1]$, almost all information from the plot will be gleaned through the arrangement of the dots and their geometrical relationships. If a broader range of values can be displayed in a cell, the additional complexity of patterns potentially occurring in the plot will tend to require that fewer data elements are put on each axis.

The dotplots we use for visualizing duplicated source code can be categorized as having a given order for the axes and having boolean values in the matrix cells.

5.3 Dotplot Visualizations of Duplicated Code

We employ the dotplot for visualizing duplication most of all to get to know *where* the clones are located. To a lesser extent we also can derive from a dotplot *how much* duplication is going on in the system.

The question about location is answered by the dotplot in the following ways:

- It shows the location and the length of the clones, see for example Figure 5.1.
- If we focus on the gaps between the diagonals, the dotplot shows the exact locations where two closely related fragments differ. This can be helpful when we compare two versions of the same file, as in Figure 5.2: the places where source code has been inserted and deleted are clearly visible. The user can connect the split fragments to form larger structures of copied and changed code.
- A dotplot relates the positions of multiple clones. In situations where duplication relations exist between many overlapping fragments, a dotplot which extends in 2D space brings insight quicker than a mere list of the involved clones.

  Pustell&Kafatos (using the dotplot for DNA sequence analysis) summarize these advantages of the dotplot display:

  > The important feature of all matrix methods is that they display the regions of homology in context, *i.e.*, that they reveal the distribution of matched and non-matched segments along the sequences. ... The overall picture of the forest [instead of only the trees] is communicated. [PK82]

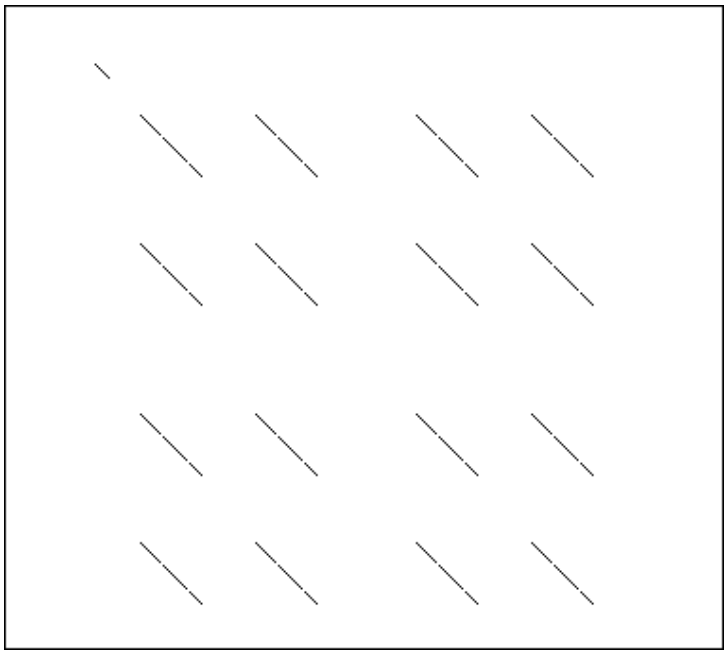


Figure 5.1: A dotplot showing the common fragments between two files from the AGREP system. All spurious matches and small sequences have been removed so that we can see clearly the location of important clones.

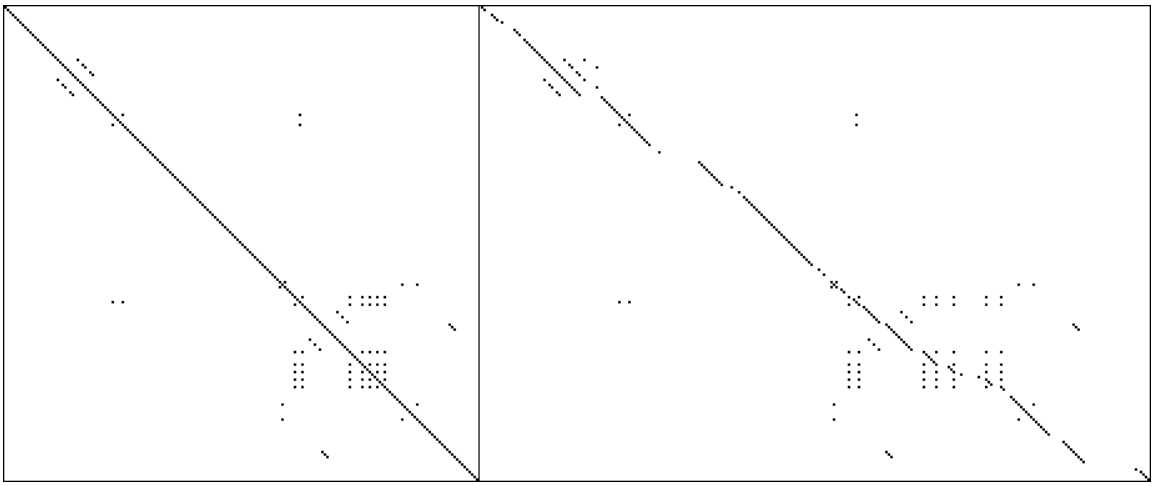


Figure 5.2: A dotplot of evolution in the MESSAGE BOARD system: the square on the left contains the comparison of the old version with itself. The plot on the right side shows the comparison of the old with the new version.

This can be seen best in the example of repetitive fragments in Figure 5.3. The dotplot always shows all equivalences at one glance, whereas a list of source fragments requires the mental combination of the fragments.

We can compress dotplots with Information Mural methods [JS96][Mal99] to show plots of arbitrary side length in a constrained space (see Figure 5.4). These views do not exhibit the detail of individual comparison sequences any more. They reveal large-scale structures as well as the overall amount and the distribution of duplication.

Since dotplots reveal more patterns than can be preconceived and built into an automated pattern matcher, they support exploratory data analysis. On the other hand, if the goal of the duplication analysis is to uncover clones that can be immediately refactored, dotplot visualizations are mostly unnecessary. Dotplots are not good at answering the question of "What?" which is required for a detailed analysis by refactoring engines. Complex information about the meaning of the matched code cannot be presented in such an abstract visualization as the dotplot is.

Use of Dotplot In Clone Detection: Dotplots are widely used for visualizations of duplicated code: Helfman et al. have looked at versions of a variety of C programs [CH93][Hel94][Hel95][BCHK99]. Baker, who computes the clones using a suffix tree, illustrates the findings with dotplots [Bak92]. Kamiya et al. also use dotplots extensively to show where clones occur [KKI02] and to get user input for reconnecting split clones [UKKI02b]. Detection approaches which focus on the automated analysis of refactoring opportunities usually do not utilize visualization techniques apart from pretty–printing the affected source code [BYM+98][BMD+99][Kri01][KH01b].

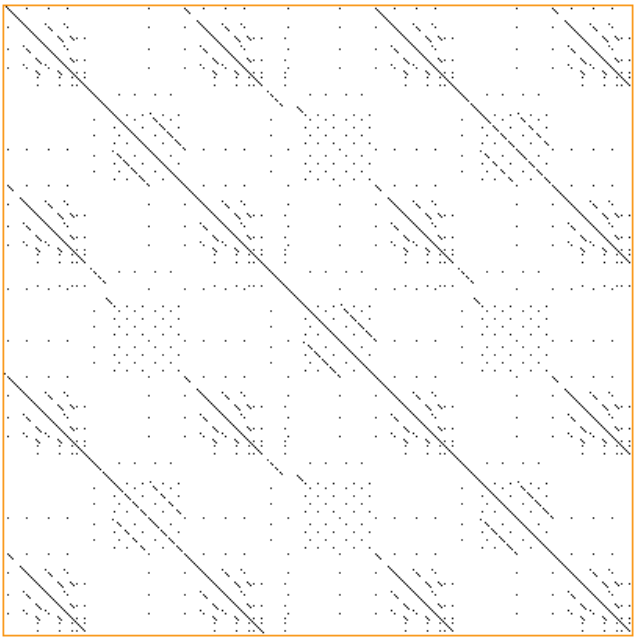


Figure 5.3: An extract from a comparison matrix of the AGREP system. We see that the code consists in regular repetitions of the same few lines with only marginal variations.

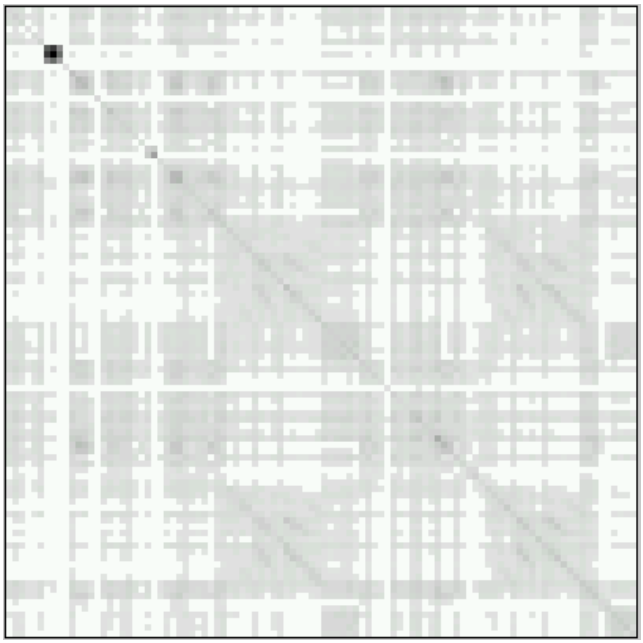


Figure 5.4: An overview dotplot of 32041 lines of FORTRAN90 code. It reveals the overall distribution similarity as well as a large copied sequence.

5.4 Quantitative Duplication Visualization

One problem of the reengineer who must deal with duplicated code is that clone detectors report large amounts of data for which he has little tool support. For industrial systems a duplication rate of 5-10% is considered a low but common estimate. This means that in a system of 1 million lines of code, 50'000 to 100'000 lines of code are involved in duplication. If we assume an average length of about 25 lines for a copied fragment, we get a minimum of 1000 to 2000 clone pairs that have to be investigated.[5]

The engineers charged with duplication investigation and removal are subject to the usual time and cost constraints of an industrial setting. They most likely do not have the resources to remove every last instance of duplication from the system but have to prioritize and decide which clones to remove. To do so, they have to

- assess the system regarding the occurrence of duplication, *i.e.*, get a *mental picture* of the redundancy situation.
- identify and select duplication that is "problematic" or "worthwhile to refactor". This includes, for example, large fragments that have been copied multiple times but eventually also duplication that is easy to refactor.

Moreover, the engineers need to process the duplication data in an organized way by prioritizing the investigations they must perform. For example one way is to start with the largest clones or the ones involving the most source files, or the ones where a refactoring would have the most impact. Since the ultimate decision on whether to refactor or not usually involves a manual investigation of the source code, the information presentation must be interactive and connected to the underlying code, to allow for short examination cycles.

To solve this problem we propose to apply polymetric views [LD03] to the context of duplicated code, *i.e.*, we visualize duplicated elements of different levels of abstraction and enrich the views with metrics that present qualitative information of these abstractions.

Our approach to support the understanding of duplicated code is based on data visualization. According to Ware [War00], visualization is the preferred way of getting acquainted with and navigating large data pools. Duplication data is relational data: two source code entities are related by shared pieces of code. A natural way of expressing these relations is a graph: the nodes of the graph represent the source entities whereas the edges of the graph represent the duplication relations.

5.4.1 Entities and Relationships

Our hypothesis is that when investigating a system affected by duplication, our mental model basically consists of the following elements:

The *source entities* represent (fragments of) the source code. In the context of this section we use files as source entities. Other entities such as subsystems, modules, classes, and methods can be used as well.

The *duplication relationships* connect the source entities.

The *duplicated fragments* are the source code that two (or more) source entities have in common.

Since the investigated systems are of huge proportions (millions of LOC), data growth reaches unwieldly amounts (thousands of clones). Would we visualize all individual clones, we would get views where overplotting of nodes and especially of edges are hindering interpretation. To achieve scalability we therefore aggregate related clones into higher level entities. We use the containment hierarchy defined in section Section 3.7.2: *clone class families* aggregate *clone classes* which in turn aggregate *clone pairs*.

The visualizations we propose use the source files and the clone class families as entities. The decision not to display clone classes or clone pairs is due to scalability constraints. The lower level clone entities, *e.g.*, the

[5] In this chapter we are not interested in the method of clone detection and assume that the problem of false positives has been solved.

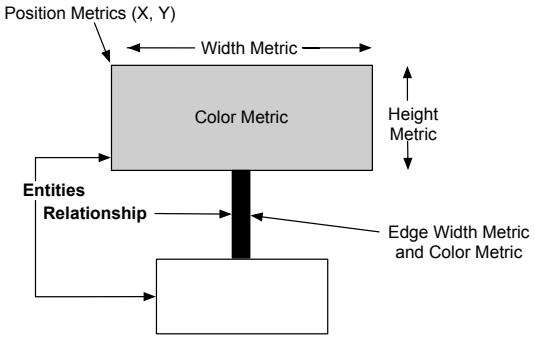


Figure 5.5: The principle of a polymetric view.

clone classes and clone pairs which are the real targets of eventual refactoring operations, must however be reachable from their containers.

5.4.2 Polymetric Views

Polymetric views [LD03] are a visualization method for nodes-and-edges graphs enriched with semantic information such as metrics. Figure 5.5 illustrates how two-dimensional nodes representing entities, *e.g.*, software artifacts, and edges representing relationships can be enriched by software metrics. A node figure is able to render up to five metric values: in its width, height, x- and y-position, and in its color. An edge figure is able to visualize two metric values: width and color.

By applying metrics to the x- and y-position of the nodes, for example, similar entities are clustered close together in an easily identifiable region of the graph exhibiting some of their defining characteristics. Entities with differing characteristics are then placed in a distinct region of the graph. In this way the shape of the visualized graph is able to communicate useful facts about the set of all visualized items.

5.4.3 Duplication Metrics

To discern between instances of code duplication we select a number of metrics that characterize the source files and clone class families (see Table 5.1). The choice of metrics is guided by the goal to create views that visually distinguish the entities in the view most effectively and intelligibly. The metrics are simple and can be computed from the results of any duplication detection tool without the aid of a parser.

The distinction between LIC and LEC is motivated by the possibly more complex situation that has to be understood when clone instances are located in different source entities. The smaller the amount of code that is involved in the duplication (the copied code *and* the surrounding context), the lighter is the cognitive load. Kapser and Godfrey [KG03] have proposed a clone taxonomy which is built on this distinction.

5.4.4 Display Scalability

A well known problem in graph layouting is *overplotting*, *i.e.*, when too many nodes and edges are crammed on too little screen space, making a diagram unintelligible. Since we want to be able to display large datasets we are forced to take precautions against overplotting. We employ the following techniques:

Reduction and filtering: By pooling related clones into clone classes and clone class families we reduce the size of the clone sets significantly. In the same manner source files can be combined into directories and subsystems.

Adaptive Graphical Representation: Since our views are intrinsically interactive, visual enhancements like

Source File Metrics	
Name	*Description*
LOC	**Lines of code** The size of a file is a common metric, despite its obvious drawbacks. It is immediately understood by every programmer and thus well suited to identify important files.
LCC	**Amount of copied code in the source file** This is the central aspect we are interested in. This records every piece of code in the file that has been copied somewhere else in the system, including in the file itself.
LIC	**Lines of code copied file-internally** A subset of LCC, this metric records code that has been copied *within* a source file.
LEC	**Lines of code copied file-externally** Another subset of LCC. This metric records code that is shared with other files. Note that LIC and LEC are not necessarily disjoint.
Clone Class Family Metrics	
Name	*Description*
NSF	**Number of Source Files** In how many source files are the copied fragments found? This is the set that defines the clone class family.
NCC	**Number of Clone Classes** How many clone classes have been grouped together in a family? This says how many different source fragments are shared by all the files in the group.
LCC	**Lines of Copied Code** How many lines of code does the clone class family encompass? For each clone instance that is part of the clone class family, the number of copied lines is summed up.

Table 5.1: Duplication Metrics for source files and clone class families

highlighting can be triggered by roll-over mouse events. Multiple selections of nodes, *e.g.*, via their names or their connections, are necessary as well to take advantage of the views.

5.5 Polymetric Views of Duplication

This section proposes a set of polymetric views that support the understanding of the duplication situation in a system and can guide refactoring actions. Each view is presented using the following schema:

Details. Gives a tabular technical description of the view, its entities, relations, and its layout.
Intention. Explains how the view can support the engineer in his tasks.
Symptoms. Details what kind of duplication problems the view reveals.
Examples. Shows sample views and explains their features.
Scaling. Investigates how the size of a system affects the view negatively and what can be done about it.
Overplotting. Investigates if the amount of data can cause overplotting problems and how they can be avoided.

We order the description of the views in a sequence that suggests a way for the engineer to walk through the task of understanding a system's duplication (a *reengineering road map*). After the discussion of each view we present a short overview of questions answered and potential further questions that are of interest at this stage.

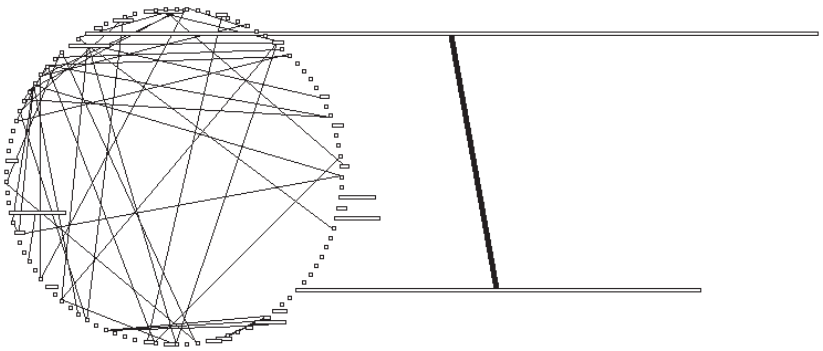


Figure 5.6: The Duplication Web of the MAIL SORTING system with LIC (Lines of file-internally Copied Code) as node width.

5.5.1 The Duplication Web

The Duplication Web is the first view that an engineer can use as it introduces the user to the duplication situation. It shows all files in the system and all existing clone connections between them.

Nodes	Source Files
Edges	Clone Connections
Metrics	
Node Size	Height = – Width = LIC (Internally Copied Code)
Edge Width	LCC Lines of Copied Code
Layout	Nodes placed on a circle; Nodes with many connections are placed apart on the diameter.
Examples	Figure 5.6, Figure 5.7

Intention: This view gives an impression of the number of files in the system and the amount of duplication that connects them. It shows the entire system at once in a well defined shape that is independent of the physical organization. It improves on a textual report detailing all clones detected in a system.

Symptoms: The view reveals the following duplication problems in a software system:

- Wide nodes represent files that contain a lot of internal duplication.
- Thick edges connect files that share a lot of duplication.
- Nodes with many connected edges represent files sharing duplicated code fragments with many other files.

Example 1. Figure 5.6 of the MAIL SORTING system shows 101 nodes, 57 of which share code with one or more other files. Most of the files are *copy-connected* to only one or two other files. The two files with the largest amounts of internal duplication also exchange the most external duplication.

Example 2. Figure 5.7 shows the application of the Duplication Web view to the Microsoft Foundation Classes (MFC). It is formed by 240 source files, 50 of which are connected by duplication links. In this variant, node size corresponds to number of connections. Following the edges one is able to divide the duplication activity of MFC into two larger groups of multiple interconnected files, and a few file pairs.

Scaling: The dimensions of the view can be controlled because of the fixed shape of the circle. For large numbers of files the radius of the circle must be reduced, but its *gestalt* can still be recognized. If there are too many files, grouping them into directories, modules, or subsystems is helpful.

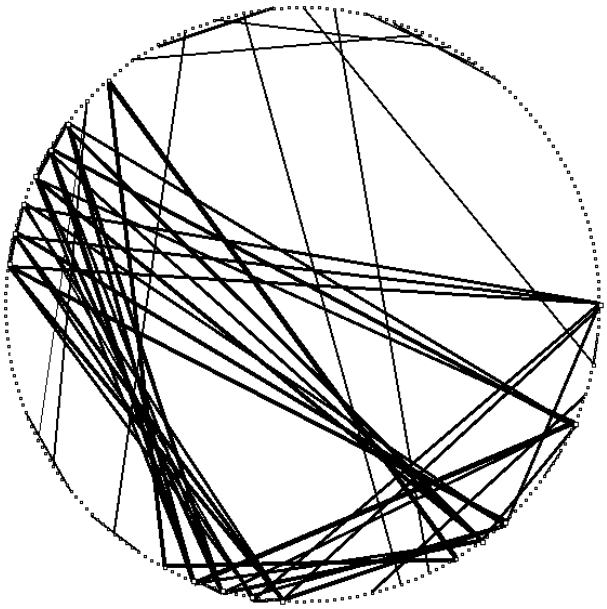


Figure 5.7: The Duplication Web view of MFC. Setting heavily-connected nodes apart on the diameter emphasizes the overall number of duplication connections.

Overplotting: Since the intention of the view is to give an overview rather than to guide actual refactoring actions, the overplotting is not too problematic. Thanks to the fixed position for each node, overplotting can only become a problem if many nodes have a very high LIC value. If too many clone connections exist between the files, the edges in the center of the view will become impossible to distinguish. The view then only conveys the information that a lot of *copy & paste* programming has been going on.

Reengineering Roadmap

Having gotten an impression of the duplication activity in general, we want to focus a bit more on the individual files. Which are the files that are heavily duplicated, which are the ones where only a small part has been copied?

5.5.2 The Clone Scatterplot

The Clone Scatterplot displays the same nodes and edges as the Duplication Web but the layout takes into account the size and duplication metrics for each file. It has still overview character but enables informed selections since more information is included in the presentation.

Nodes	Source Files
Edges	Clone Connections
Metrics	
Node Position	X-Pos = LOC Lines of Code Y-Pos = LCC Lines of Copied Code
Edge Width	LCC Lines of Copied Code
Layout	Scatterplot
Examples	Figure 5.8

Intention: The Clone Scatterplot confronts the size of the files with the amount of duplication they contain. Files of different duplication levels can be identified by the region they are positioned in. The edges tell us

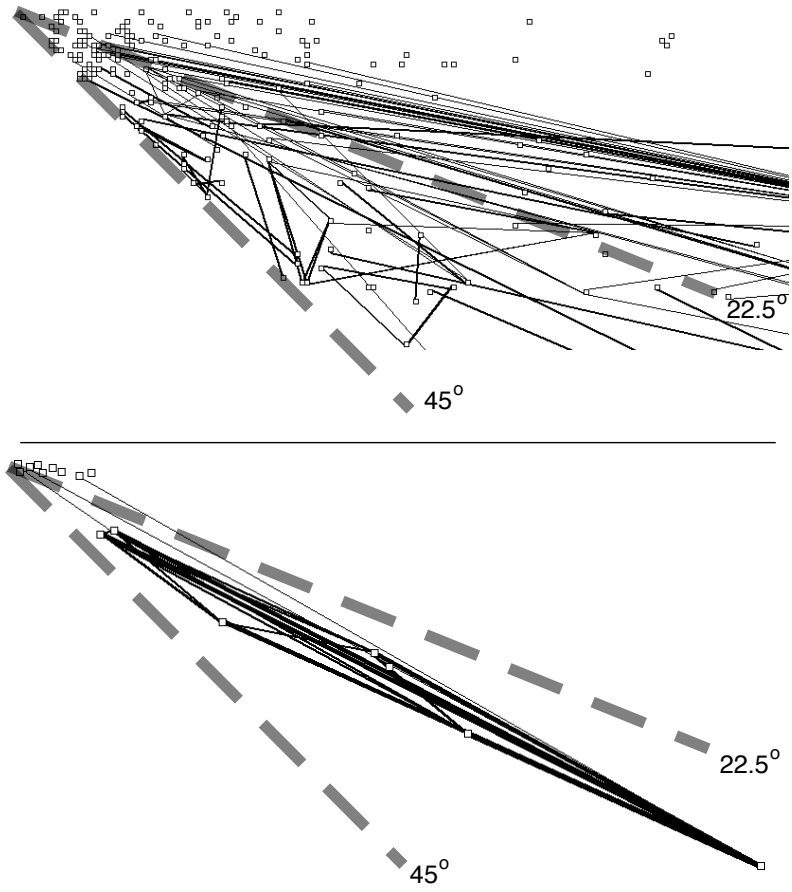


Figure 5.8: Two examples of the Clone Scatterplot: On top is an extract of the ACCOUNTING system. Below, the entire AGREP system is shown. Both views are overlaid with lines indicating duplication rates of 100% ($45°$) and 50% ($22.5°$).

if code is shared between large and small files, or between files of similar size. Heavily copied files can be selected for closer inspection.

Symptoms:

- Nodes on the left represent small files, while the ones on the right represent large files.
- Nodes at the top of the view represents files having little or no duplication.
- Nodes that are not at the top of the view but are unconnected represent files having only internal duplication.
- Nodes close to the $45°$ diagonal represent files containing a lot of duplication with respect to their size.

Examples: The *gestalt* impression that this view gives can be best observed in the scatterplot of the AGREP system in the lower half of Figure 5.8. Here the system has very little variation around the main diagonal. This indicates that the level of duplication is equally high in all of the larger files. The largest file has common code with all the other files involved in external duplication.

Scaling: Since we use the LOC metric as X-Position, the view can grow very large when files contain a lot of lines. Logarithmic scales can then be applied to the X-metrics especially.

Overplotting: Thanks to the use of the LOC metric as X-Position, the source files are spread out over the view area, ameliorating the overplotting situation for the nodes. Smaller files without duplication are clustered in

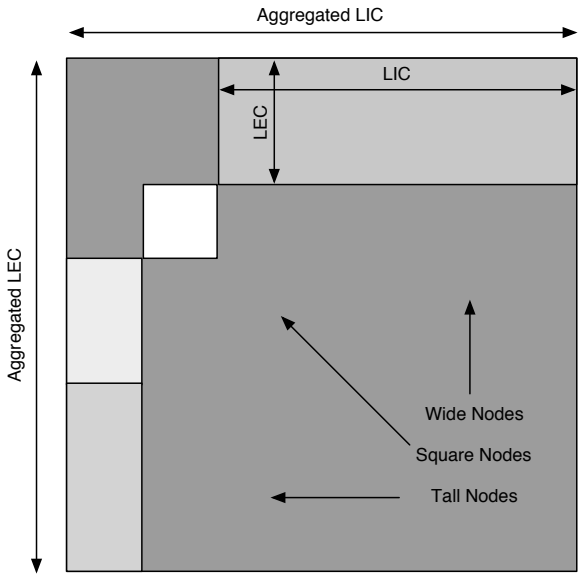


Figure 5.9: Node placement in the Treemap. Nodes are separated by their shapes and arranged so that the values of LIC and LEC are aggregated horizontally and vertically, respectively.

the upper left corner of the view, frequently overplotting each other. Since these files are not interesting for the user the problem does not have any impact. Clone edges tend to overplot quickly around the $45°$ diagonal, where the files with high duplication rates are located.

Reengineering Roadmap

Until now the views only contained nodes representing individual files. Files are however part of organizational system structures. We want to know how these larger entities are affected by duplication. This raises the abstraction level and we get the useful side effect that we can connect gained duplication knowledge more easily with the fewer elements of the coarse system structure.

5.5.3 The Duplication Aggregation Tree Map

This view aggregates the duplication that until now we have only seen attached to individual files. It shows the entire system top-down along the directory structure, annotating each directory node with the recursively aggregated amounts of internal and external duplication of its files and subdirectories. The view emphasizes system parts according to their involvement in duplication.

Nodes	Source Files, Directories
Edges	–
Metrics	
Node Size	Height = LEC (Externally Copied Code) Width = LIC (Internally Copied Code)
Node Color	LCC Lines of Copied Code
Layout	Tree Map; nodes are arranged according to the principle illustrated in Figure 5.9.
Example	Figure 5.10

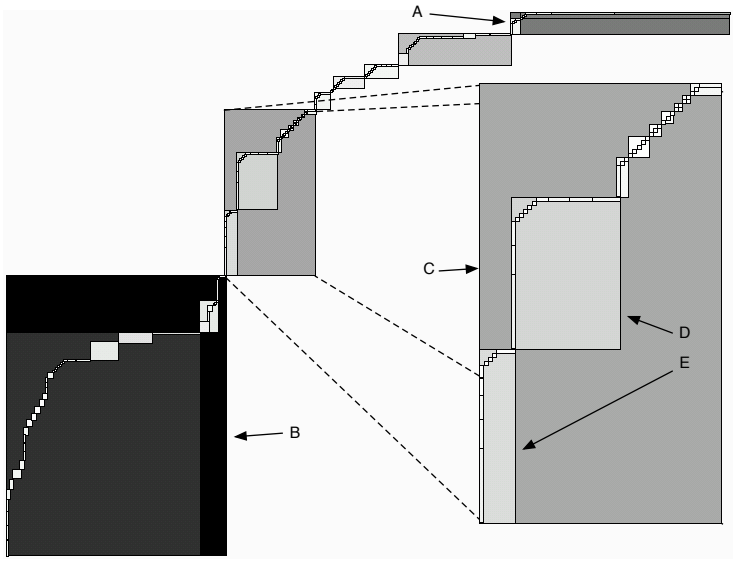


Figure 5.10: The tree map of the APACHE system. The rectangle on the right marked C is an enlargement of the second top-level node from left.

Intention: The tree map aims to give an overview of the ratio of internal to external duplication, aggregated from the individual source files up to the root directory of the system. The parts of the system which exhibit high amounts of duplication can be identified at a glance from the top level. Relative comparison of structures in the hierarchy is made possible. The view has a *gestalt* property, *i.e.*, it can give useful information immediately.

Symptoms:

- Nodes towards the lower left have increasing amounts of external duplication.
- Nodes towards the upper right have increasing amounts of internal duplication.
- Nodes in middle have no duplication or equal amounts of both kinds.
- Wide nodes have more internal than external duplication and vice versa.

Note that node height shows the sum of externally copied code only with regard to files. If two files within a directory d share some code, this amount will be aggregated as LEC for the node representing d, even though the code is not copied to files external of d.

Examples: From the shape of the overall diagram in Figure 5.10 we can determine that there is a bit more internal duplication than external duplication in APACHE. The rightmost node A representing the directory `lib` contains the most internal duplication, whereas leftmost node B representing the directory `modules` and its subdirectory `standard` contain most of the external duplication. The directory `os` (represented by node C) contains two subdirectories `win32` (node D) and `netware` (node E) which have a similar amount of external duplication (possibly shared between them).

Scaling: Thanks to the fractal property of treemaps we can display systems of any size on every screen. Zooming provides an adequate instrument to navigate even very large systems. Aggregation of data will provide useful information even at the highest level where the smaller details are not discernible any more. Contrary to traditional treemaps this variant visualizes two values in every node, resulting in some waste of screen space. The advantage over the traditional treemaps is that the display is less crowded while still showing every element of the tree.

Overplotting: The layout precludes all overplotting problems.

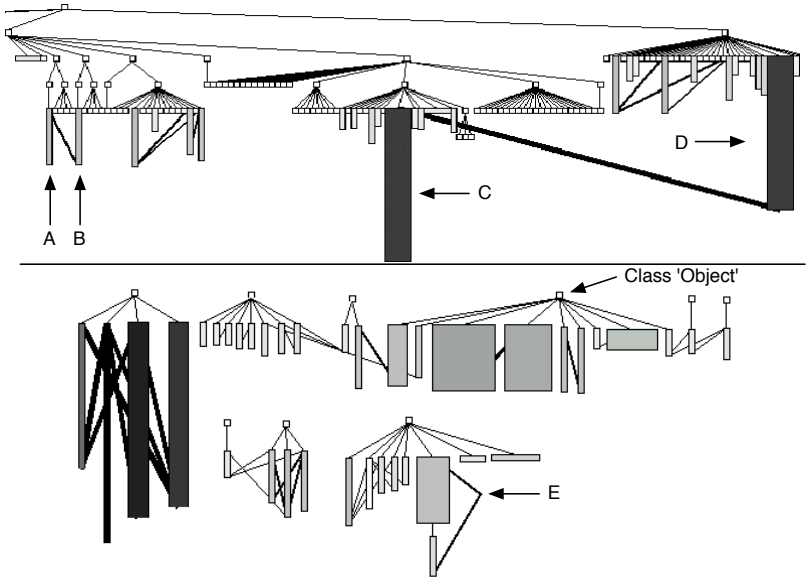


Figure 5.11: Two variant System Model views of JBOSS. The upper half shows part of the directory structure. The thicker edges represent clone relationships between files. The lower half shows extracts from the class hierarchy. Small squares represent superclasses defined outside of JBOSS.

Reengineering Roadmap

Having gained an overview of the parts of the system involved in duplication, we want to know details about the copying. Is code shared within directories only, or also across directory borders, and even subsystem borders? These informations are interesting since they uncover functional relationships between system parts that may not be documented. Such knowledge can also further the understanding of the design of the system.

5.5.4 The System Model View

This view shows the directory structure of the application, or alternatively the inheritance tree, using the familiar tree layout.

Nodes	Source Files, Directories
Edges	Clone Connections, Directory Containment
Metrics	
Node Size	Height = LEC (Externally Copied Code) Width = LIC (Internally Copied Code)
Edge Width	Clone connections = LCC (Copied Code) Directory Containment = –
Layout	Spaced Tree
Examples	Figure 5.11

Intention: The System Model view shows the duplication within the physical location of files, *i.e.*, their directory structures or the classes and their inheritance relationships. It helps identifying problematic subsystems and functional connections between subsystems.

Symptoms:

- Small squared boxes represent files without internal or external duplication.

- Flat wide boxes represent files that contain a lot of internally duplicated code.
- Tall narrow boxes represent files sharing a lot of duplicated code with other files.
- Thick edges among tall boxes represent the amount of duplicated code exchanged between them.

Examples. In the upper half of Figure 5.11 the directory structure of the JBOSS system is the basis for the arrangement of the source files in the view. Internal and external duplication are the metrics that are shown. Files A and B, as well as C and D share code as indicated by the duplication link between the files, as well as by the similar shapes of the nodes. What can additionally be seen in Figure 5.11 is that A and B are located in sibling directories, whereas the duplication between C and D crosses 4 directories, *i.e.*, probably into another subsystem. This information is useful when deciding about refactoring measures. In the lower half of Figure 5.11 extracts from the class hierarchy of JBOSS are shown. On the left side, sibling classes copy heavily from each other. E marks a clone relation between a class and its superclass.

Scaling: The view becomes very large in a system with thousands of source files. Clone edges will likely go over the screen boundaries when connecting files in directories that are far apart, making good navigational facilities a necessity.

Overplotting: Trees are simple to layout without node overplotting. Displaying the clone edges, however, can lead to serious overplotting problems, especially if the system model is a shallow tree.

Reengineering Roadmap

Until now, our focus has been entirely on the files. We know their sizes, their locations and their connections. We now turn to an investigation of the connections, the code that is shared. How large is it? How many files has it been spread to? Are other common fragments copied along with it?

5.5.5 The Clone Class Family Enumeration

This view reduces the redundancy of the duplication connections that has been present in all the previous views. The clones are shown in a concise nodes-and-edges view.

The layout uses the LCC and the LOC metrics to place clone class families and source files, respectively, on the horizontal axis. The intuition “the farther to the right the bigger” thus can be used to mentally classify both entity types presented in the view. The edges connect the upper half of the view - the clone class families - with the source files on the lower half.

Nodes	Clone class Families (CCF), Source Files
Edges	Participation in a Clone class Family
Metrics	
CCF Level	NSF (Number of Source Files)
CCF Position	X-Pos = LCC Lines of Copied Code
File Level	Number of clone class groups
File Position	X-Pos = LOC Lines of Code
Layout	
Upper half	Multiple levels of cloneclass families
Lower half	Multiple levels of source files
Examples	Figure 5.12

Intention: This view presents the clone class families to the user in a way that eases investigation of individual instances of duplication. It characterizes the families by the criterion of how many source files they comprise and how much code they contain. The user can start on a clone class family node and see which source files are participating. Or he can start with a source file node and see in how many clone class families the file participates. To make the view fully useful, lower level duplication entities, *i.e.*, clone classes and finally clones, must be made available to the user via the nodes in this view.

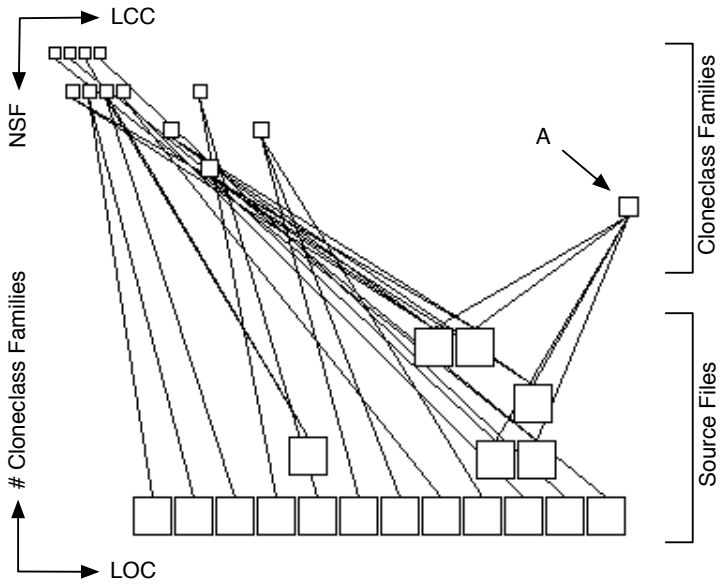


Figure 5.12: The clone class families of the MULTIMARKE system.

Symptoms: Clone class families in the top rows are less important since they connect only a few source files. The families located on the rows towards the middle of the view have an increasing number of participating source files which makes them interesting targets for investigation. Symmetrically, source files at the bottom of the view are only involved in a single clone class family, whereas files in the middle of the view are more interesting. Small files are to the left of the view and large files are to the right of the view.

Examples: Figure 5.12 presents 18 files and 13 clone class families which stand for 55 clone pairs (a 76% reduction of duplication entities). The largest clone class family A encompasses duplication in the 5 largest files, as can be seen from the figure. Clone class family A represents two clone classes—this means two different source fragments that are present in all 5 files—or 24 clone pairs.

Scaling: Clone class families or source files containing a lot of code are positioned at the far right, likely offscreen, which will require navigation.

Overplotting: The nodes must not overplot since the user has to be able to select from them. The layout mechanism thus arranges them side by side. Edge overplotting is of minor concern since the focus of the user lies on the nodes. Eventually, clone class families which represent only internal duplication in a single file could be removed from the view.

5.6 Discussion

The views achieve the goal of data reduction on different levels. We are able to display even very large systems on restricted screen space. Many of the views have a *gestalt* property, *i.e.*, they provide overview information *at a glance*.

The reduction of the cardinality of the clone sets, however, is sometimes not enough, resulting in cluttered displays which are hard to read. We must further support readability with interactive enhancements of the views, *e.g.*, with the highlighting of connected elements on mouse-over.

By using simple and heuristic layout mechanisms we provide a fixed arrangement of the nodes for all views except the System Model view. This is an advantage as there is no need for the user to rearrange the nodes to get a better view. This enables him to start *interpreting* the view immediately.

What is missing from the description in this section is the necessary ingredient if the duplication is to be reengineered: making the source code of the clone instances reachable directly via the nodes and edges of the views by code browsers. This must be addressed by tool-builders.

That tool support is only one piece of the duplication refactoring puzzle in an industrial context is a fact which we have not included into our considerations. How business decisions and process questions affect the engineers in this matter will have to be addressed still.

5.7 Related Work

Visualizing Duplication with Graphs. Johnson [Joh94b] has used Hasse diagrams to visualize textual similarity between files. For each duplication-related cluster of files (a clone class family in our terminology) the diagram shows the copied source text and the source files as nodes, and the inclusion relationships between the different code pieces as edges. The height of a node in the graph is determined by its size: large files or code fragments are towards the bottom, smaller pieces of code towards the top. His graph is similar to the clone class family enumeration proposed in Section 5.5.5.

In [Joh96] Johnson proposes to navigate the web of files and clone classes via hyper-linked web pages. Although the entities and relationships that he defines are the same as we have used in this section, his system lacks the overview and selection features that we think are necessary to find one's way in the mass of duplication data. His browsing system could however act as a backend for the views proposed in this section.

Visualizing Duplication with Dotplots. The dotplot as we have described it in Section 5.3 has some drawbacks when it comes to the visualization of large systems:

- Dotplots produce spacious images. The size of the image depends on the size of the input, not on the amount of the duplication found.
- In a dotplot visualizing the comparison of multiple source entities there is no predetermined organization of the image. Some features may only be detected after rearrangement of the display.
- Dotplots contain a lot of redundancy. This can be overwhelming in the case of frequently repeated pieces of code.
- Dotplots give a detailed account of the duplication situation. As a consequence they convey overviews rather poorly.

Dotplots and polymetric views can be used as complementary duplication visualizations. The polymetric views are good as a starting point for the assessment phase. They give the user a *ToDo* list that has to be cleared point for point. Having selected a source file or a clone class, a targeted dotplot displaying only the clones belonging to the selected clone classes can be presented to the user for close inspection of the situation.

Visualizing Duplication in OO Class Hierarchies. Golomingi [KN01] has investigated how the information about the location of clones within an object-oriented class hierarchy can be utilized to decide upon refactoring measures. The focus of his work however was automation rather than visualization, *i.e.*, seeing the classes and their relationships was not the primary goal.

5.8 Conclusions

If a reengineer has to investigate and refactor duplication in a large system, he is in dire need of support for understanding and dealing with the potentially huge amount of copied code. To manage the overwhelming lists of detailed duplication information produced by duplication detection mechanisms, we reckon that he needs to *i)* overview of the duplication situation and *ii)* navigate through the sea of information.

The approach discussed here is putting emphasis on the "human in the loop", giving human expertise the helm, instead of pushing automation. In this section we have proposed a number of polymetric views which structure the data and combine it with the knowledge about the system that the engineer already possesses.

We have only used a small and very simple set of metrics which can be computed without much investment in parsing infrastructure. Future work should include investigations of metrics or attributes oriented towards qualitative aspects of duplication. This will increase the selective capabilities of the views.

A pertinent problem is the overplotting of edges and nodes when systems and the amount of data get too large. We have proposed some aggregation abstractions to reduce the amount of data that must be presented on screen. Sophisticated filtering techniques should be the focus of tool development efforts if a visualization tool wants to be applicable to very large systems. Since the views greatly rely on their interactivity they have a limited usefulness when committed to paper.

Future Work

The views proposed in this section have not yet been validated in reengineering settings. Such an evaluation could prove the usefulness or eventually lead to the development of better views.

As we have noted above, the metrics we choose to annotate the views are basically various types of size metrics. If we are able to derive metrics which characterize other aspects of the duplication, *e.g.*, the suitability for a given type of refactoring, we are able to produce views which cold answer very specific reengineering questions.

The problem of overplotting is ever present in views which show nodes *and* edges. An avenue to reduce this problem is to investigate interactive means of highlighting edges and nodes. One possibility is for example a *slope selector* with which the user could designate a certain range of slopes. All edges which have a slope outside of this range would be made invisible. In views like the Clone Scatterplot where the slope of an edge has semantics (a horizontal slope, for example, indicates that two files are involved in no other duplication than in the fragment shared among them) slope selection could identify files with similar duplication "profiles".

Chapter 6

Experimental Validation

The techniques that we have proposed in Chapter 4 need to be validated against the goals that we have selected in Section 2.3. We formulate the overall hypothesis that we test here as follows:

> We are able to build a clone detector which finds a good nummber of difficult-to-detect clones. Ranking is an effective means to handle long list of candidates clones containing many false positives. We are able to implement such a clone detector for a wide range of programming languages with only small adaption costs. The clone detector is able to treat systems of moderate to large sizes.

The evaluation of this hypothesis is broken down into the following aspects which are each tested in a separate section of this chapter:

Language Independence

We can implement the techniques presented in Sections 4.2 and 4.4 in a way that is easily adaptable to a wide range of programming languages. (Section 6.1)

Code Normalization

Normalizing elements of the source code decreases the number of false negatives for non-exact duplication. We can distinguish normalization methods which cause an overly large increase of false positives. (Section 6.2)

Comparison with Other Approaches

The performance of the proposed techniques is comparable with other string-based clone detectors. (Section 6.3)

Assessment of Line-Orientation

The choice of line-orientation as comparison unit is not to the detriment of the detection accuracy. (Section 6.4)

Ranking of Clone Candidates

We can effectively rank clone candidates so that false positives are ranked behind candidates that are relevant. (Section 6.5)

Scalability

We can apply our clone detector on case studies of moderate to large sizes. (Section 6.6)

We conclude in Section 6.7.

Recall and Precision

In this section we explain two important measures that we use in many of the evaluations presented this chapter.

In Information Retrieval, two numbers are used to characterize the effectivity of a retrieval method: *recall* and *precision*. In Figure 6.1 we illustrate their definition. Let us suppose that there exists some ideal set A of *actual clones*, which can only be assessed by inspection. In practice, it is too expensive to determine A by manual inspection, and in any case the results will depend heavily on the subjective opinion of the person performing the inspection.

An automated tool will identify some set C of *candidate clones*. Clearly we would like C to be as close as possible to the ideal set A. D is the set of candidates that would reasonably be accepted as being actual clones. *Recall* measures the fraction of actual clones that are identified as candidates, and *precision* measures the fraction of candidates that are actually clones. A good technique should exhibit both high recall and precision, but depending on the context, and the numbers of candidates returned, one might accept, for example, better recall in return for poorer precision.

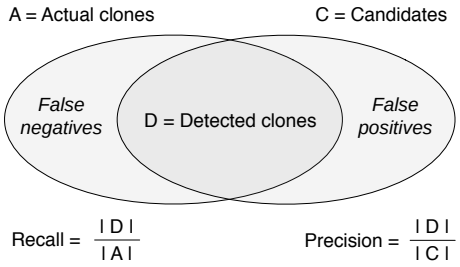


Figure 6.1: Recall and Precision

6.1 Validation of Language Independence

Whenever source code is treated programmatically as *code* and not only as *text* (as Johnson [Joh93] does), a certain amount of lexical analysis is necessary. Language independent does thus not mean that a tool does not require any adaption for a new programming language. We understand language independence to mean that a technique can be made to work with a new programming language spending only a modest amount of effort. The difference between a 'lightweight' and a 'heavyweight' approach is thus a gradual one.

In the context of detecting duplicated code, we need to recognize certain aspects of the program structure and certain syntax elements for the purposes of pretty-printing and normalizing the source code. The following list categorizes the different elements according to how much effort it takes to adapt a parser:

Regions of Text: A sequence of characters in the source text, delimited with some marker tokens:

- Comments
- Literal strings, literal characters
- Blocks, statements, expressions

Simple Words: Single words in the text which can be recognized without any or with only minimal context:

- Keywords, operators, builtin types
- Literal numbers
- User-defined variables
- Function declarations
- Function invocations

Context-Sensitive Elements: These elements require larger grammatical rules and context, as well as symbol tables eventually:

- Variable declarations
- User-defined types
- Specific expressions such as loop-conditionals, if-conditionals

To be able to switch from one language to the other, we need to make the recognition of these elements configurable, *i.e.*, we must provide a specification facility which does not require to provide a complex and error prone formalism like a formal grammar.

We will look in bit more detail how we can build such a specification interface for the two first categories specified above. We have not described any uses of the information that could be from recognizing the syntax elements of the third category and will therefore treat them in a future work section.

6.1.1 Recognizing Regions in the Source Text

The most obvious example of a region of source code that we want to recognize is the commentary. Regions of commentary text are delimited with a marker at the beginning and one at the end. These marker can either appear anywhere in the text, or only at a fixed position of the line for some older languages with fixed source layout like COBOL and FORTRAN.[1] To recognize commentary we can *i)* trivially interpret the position based syntax and *ii)* use a simple scanner for the delimited text. Both types of *code cleaners* can be made configurable with the exact positions and/or delimiters. Note that in order to *only* remove comments delimited by character

[1] In COBOL comments always take up the entire source line and are introduced by an asterisk '*' at position 7 of the line. The part of the line containing the code ends at position 71. Beginning with position 72 version identifiers may optionally appear, which are again considered to be commentary by us.

sequences, the parser also must recognize literal strings. The reason is that comment delimiters can appear deliberately (in programs which produce source code as output, such as BISON) or accidentally within strings (see Listing 6.1). The scanner will be derailed by unbalanced occurrences if it does not take literal strings into account.

```
  printf("  %-5ld /**",totalines);
  ncout = ncout + 11;
  break;
 } else {
  printf("  %-5ld /*",totalines);
  putchar(c);
```

Listing 6.1: An extract from WELTAB where literal strings contain start- but no end-delimiters for C comments.

A generic implementation of the outlined scanner algorithm lets us adapt the tool to a new language within minutes. The parameters needed are the following (the provided examples are for the C++ language).

- Start- and end-delimiters for multi-line comments, *e.g.*, '`/*`' and '`*/`'.
- Start-delimiters for end-of-line comments, *e.g.*, '`//`'.
- Delimiters for literal strings and literal characters, *e.g.*, '`"`' and '`'`'.
- Escape signs for string-delimiters within literal strings, *e.g.*, '`\`'.

In order to recognize blocks, statements, and expressions which will allow us to pretty-print a program, we need to build a parser that can interpret recursive structures. We can configure the parser with the tokens which indicate (the provided examples are for the C++ language).

- Beginning and end of blocks, *e.g.*, '`{`' and '`}`'.
- Beginning and of Expressions, *e.g.*, '`(`' and '`)`'.
- End-of-statement delimiter, *e.g.*, '`;`'.
- Beginning and end of array indices, *e.g.*, '`[`' and '`]`'.
- Delimiters which separate parameters or other list elements, *e.g.*, '`,`' or '`;`' (in **`for`**-conditionals).

With the these few tokens we can create a parse tree down to the level of parenthesized expressions. A normalization of the layout of source code is possible for any language for which these parameters can be defined. There are some languages, however, for which this simple approach must be extended, or fails:

PYTHON uses indentation levels to indicate blocks. We additionally need a small preprocessor which tracks the indentation and inserts block delimiter tokens at the appropriate places.

FORTRAN does not have block delimiters. It also misses statement delimiters: each statement normally ends with the line.

However, since FORTRAN programs are formatted according to a fixed schema, the programmer does not have much freedom for 'creative' formatting experiments, which would necessitate a pretty printer to undo. The only option is to split statements over multiple lines using continuation lines. In FORTRAN77 a continuation line contains a character in column 5. In FORTRAN90 the continuation character is a '`&`' as last character of the line. For both cases we can use a small preprocessor to glue the parts together.

COBOL does not have statement delimiters. Since programmers are allowed to continue statements on subsequent lines without any lexical indication, we would need to fully parse the code to detect statement ends. There is no heuristic solution for this case.

LISP is not a block–oriented language and needs a different treatment, too. However, LISP code is visually intolerable if not beautified and pretty printers are therefore used everywhere. Should it still be necessary to write our own we can rest assured that there is no language simpler to parse than LISP.

6.1.2 Recognizing Simple Words

The normalization mechanisms that we have introduced in Section 4.2.3 require that we find syntax elements like literal strings and numbers, variables and function names. For the abstract code representation we employ in Section 4.4.1 to compute similarity between lines of code we need to recognize expressions, operators, and array accesses.

All of these elements are single tokens as they come out of a lexical analysis performed with regular expression matching. All that is necessary is to have for a given programming language the list of the keywords and operators, also including the diverse delimiters. To distinguish variable names from function names, we use the heuristic that function names are—in most languages—always followed by the character '(', the beginning of the formal/actual argument list. Note that with this heuristic we cannot distinguish is a function invocation from a function declaration.

There are some exceptions where the heuristics needs to be extended:

FORTRAN: FORTRAN uses parentheses not only for expressions and conditionals but also for array indices. In the following FORTRAN fragment, the variable `A` names an array and the identifier `DAXPY` names a function.

```
    IF (L .EQ. K) GO TO 20
       A(L,J) = A(K,J)
       A(K,J) = T
20  CONTINUE
    CALL DAXPY(N-K,T,A(K+1,K),1,A(K+1,J),1)
```

To distinguish the array access from the function invocation we need the context of the preceding token, which must be the keyword **`CALL`** for a function invocation.

COBOL: In COBOL, which does not have parameter lists, names of functions must be recognized by the preceding tokens **`CALL`**, or **`PERFORM`**, declared function names are recognized by a following **`SECTION`** keyword.

PL/1: There are no reserved words in PL/1, "keywords" are recognized in context by the parser. Variable names can be the same as keywords as can be seen in the following code:

```
IF IF = THEN THEN THEN = ELSE; ELSE  ELSE = IF;
```

Although this practice is deprecated for PL/1 programmers, a recognizer for normalizing variable names must in principle have some understanding of statement structure to make the distinction between variable names and keywords.

For these cases, regular expressions are however still enough to do the matching.

6.1.3 Related work

The VIM editor[2] provides a syntax highlighting facility which can be programmed with a powerful specification language. The language has all the hallmarks of a grammar: it does not only allow the programmer to define tokens (keywords, operators, identifiers), and regions (comments, strings), but one can also define inclusion relationships between regions and other regions or tokens. Even recursive structures and context sensitive syntax elements can be recognized.

The A2PS tool[3] which converts ASCII files written in many programming languages to a pretty-printed postscript also offers a specification language which identifies keywords, operators, and sequences like comments and strings. This specification language is entirely based on regular expressions.

[2] Available from `http://www.vim.org` [May 15, 2005]
[3] Available from `http://www.gnu.org/software/a2ps/` [May 15, 2005]

The CTAGS tool[4] recognizes and cross-references a number of syntax elements in a wide variety of programming languages. CTAGS does not, however, provide a common specification language for syntax elements. For a new programming language a parser must be provided, which can be based on regular expressions or on more traditional techniques.

The work on Island Grammars [Moo01] is a generalization of the idea to describe with detailed grammar productions only the parts of a language that one is specifically interested in (the islands), and to catch all remaining, uninteresting constructs with very liberal productions (the water). In our case the islands are literal strings and comments, as well as block and expressions delimiters. The rest of the source code is recognized by the catch-all productions.

It is more difficult to describe an island grammar for code normalizations, since variable names can occur everywhere in a program, *i.e.*, the grammar would quickly grow to be close to a regular and complete grammar of the language.

A variant of island grammars, *fuzzy* parsers [Kop96] identify language constructs based on an initial anchor token. The 'water' that lies between recognized constructs is not described by the grammar, which consequently can be kept small.

6.1.4 Discussion

A small amount of lexical analysis is required even for detection techniques which do not rely on parsed code.

The heuristic nature of our approach, however, enables to build lexical analyzers that can be adapted to a new language by changing a few simple parameters. We claim that language independence is achieved by our approach since the adaptation work consists only in configuration which can be done without in-depth knowledge of any advanced technology like parser construction. Also, the configuration can be achieved in short time frame. Moreover, the approach is robust and mistakes during the language configuration do not fatally interrupt the detection process and have at most a detrimental effect on the detection or selection performance.

[4] Available from `http://ctags.sourceforge.net` [May 15, 2005]

6.2 Assessing the Impact of Normalization

Code normalization increases the likelihood that we can detect duplication which has been edited, reducing the number of false negatives. When we normalize code, however, the probability of retrieving false positives increases as well. This section reports about an experiment with which we want to assess if the benefit of less false negatives is balanced by the price of more false positives.

The experiment consist in selecting two example systems and performing the following steps multiple times: *i)* select a set of normalization operations and transform the code, *ii)* compute the candidate clones, and *iii)* compare the candidates to a reference set to assess precision and recall. In each round, we vary the normalization parameter, increasing the number of normalization operations.

The section is structured as follows: We first present the comparative study of Bellon from which we have taken part of our reference set (Section 6.2.1). We then explain the parameters of the experiment (Section 6.2.2), the systems selected for experimentation (Section 6.2.3), the construction of the set of reference clones (Section 6.2.4), and finally the results (Section 6.2.5).

6.2.1 Bellon's Comparative Study

The large variety of clone detection techniques that have been developed in recent years has spurred interest in comparing their effectiveness. Bellon [Bel02b] has conducted a comparative case study with the goal of establishing the relative advantages and disadvantages of the different approaches. In his study representants of all main detection paradigms participated: string-based ([Bak92], [KKI02], [DRD99]), parse-tree based ([BYM+98]), metrics-based ([MLM96b]), and program dependence graph-based ([Kri01]). We explain here the setup of Bellon's experiment:

Reference Set Construction. To compare the different approaches, Bellon built a reference set by manually confirming a set of participant-submitted candidates to be clones. However, it is important to note that this reference set was (1) based on candidates identified by tools participating in the comparison, and (2) incomplete — due to time constraints, Bellon was only able to cover 2% of the candidates.

Clone Types. For his study Bellon categorized all clones into three types: Exact clones (Type 1), clones where identifiers have been changed (Type 2) and clones where whole expressions or statements have been inserted, deleted or changed (Type 3).

Mapping Clone Candidates to References. To decide which candidates correspond to a confirmed clone, Bellon defined a matching criterion based on the notion of *distance* between clones. This criterion assesses a clone pair to be a 'good enough' match of another clone pair if the overlap between the two corresponding source fragments is large enough and they are of comparable size. The OK and GOOD metrics determine how well two clone pairs overlap each other, *i.e.*, if they can be declared as *similar*.

When participating in Bellon's case study, we compared only non-normalized code. We now wish to consider the impact of normalizing source code on the effectiveness of our technique.

6.2.2 The Parameters of the Experiment

We have two independently varying parameters in the experiment. The *normalization degree* is a combination of normalization operations applied to the source code. The complete set of operations is shown in Table 6.1. The *gap size* is a parameter of the algorithm which retrieves clone candidates from the comparison matrix (Section 4.3.2 on Page 64).

Operation	*Language Element*	*Example*	*Replacement*
1	Literal String	`"Abort"`	`"..."`
2	Literal character	`'y'`	`'.'`
3	Literal Integer	`42`	`1`
4	Literal Decimal	`0.314159;`	`1.0`
5	Identifier	`counter`	`p`
6	Basic Numerical Type	**`int, short, long, double`**	`num`
7	Function Name	`main()`	`foo()`

Table 6.1: Normalization Operations on Source Code Elements.

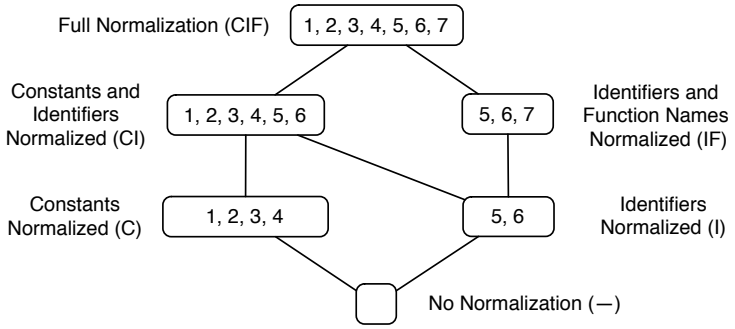


Figure 6.2: Different degrees of normalizations and their relationships.

Degrees of Normalization

We define six degrees of normalization that make use of various subsets of the normalization operations listed in Table 6.1. These six degrees form a lattice, illustrated in Figure 6.2, reflecting which normalization operations are performed. These normalization degrees correspond to the different editing operations a programmer may perform when duplicating code.

No Normalization (–). Here, only the basic noise reduction is applied to the source code, *i.e.*, removal of comments, white space, and some uninteresting lines containing only **`else`**, for example. The results gathered for this degree demonstrate the effectiveness of the basic approach.

Constants Normalized (C). In addition to the noise removal we normalize literal characters, strings, and numerical constants, *i.e.*, we map them all to a similar token (operations 1, 2, 3 and 4 of Table 6.1).

Identifiers Normalized (I). After noise removal we normalize some lexical language elements: identifiers, labels, basic numeric types (operations 5 and 6).

Identifiers and Function Names Normalized (IF). In addition to identifiers, we change all function names in declarations and invocations to `foo` (operations 5, 6, and 7).

Constants and Identifiers Normalized (CI). This includes all operations except function name normalization.

Full Normalization (CIF). Here we apply all the normalization operations.

Gap Size

The retrieval of clone candidates from the comparison matrix as described in Section 4.3.2 can be thought of as an *ad hoc* normalization mechanism. Indeed, a gap in a sequence of matching lines occurs when corresponding lines fail to match. When we allow a gap in a sequence of matching lines, we are normalizing the contents of

these lines. Because of the generality of this mechanism, the increase of noise or the loss of precision due to it is noticeable.

In this experiment we want to compare the gap mechanism with the degree of normalizations. The gaps in a comparison sequence can be controlled by the *maximum gap size* filter criterion. We have set this parameter to the values 0, 1, and 2. Based on our experience, we choose to let no more than 2 consecutive non-matching lines in a comparison sequence of minimal length 6, which is the same minimal length as was agreed upon by the participants of the Bellon study. Our experiment generates then 18 different data sets based on the six different normalization degrees and the 3 gap sizes.

6.2.3 Selection of Experimental Systems

We selected the systems from which the clones are to be retrieved according to the following criteria: (a) availability of reference data for other clone detection tools, (b) differences in coding style and line layout, (c) real applications developed by external persons, and (d) common programming language, to avoid possible influence of programming paradigms.

We chose two systems from Bellon's comparative study: the WELTAB system consisting of 39 files (9847 LOC) and the COOK system consisting of 295 files (46645 LOC) [Bel02b]. WELTAB is a relatively small application known to contain considerable amounts of duplicated code, and is therefore convenient for carrying out experiments. The COOK application adopts a code formatting approach in which function arguments and parameters are put on separate lines, posing a special challenge to line-based clone detection approaches.

6.2.4 Construction of the Reference Set

To compute recall and precision, we need to compare the candidate clones reported by our tool with a reference set of validated clones. We have constructed such a reference set from two sources: (1) the (incomplete) reference set assembled by Bellon [Bel02a] which was assembled by manually examining candidates detected by various tools, and (2) the result of a manual evaluation of the results reported by our tool as shown by the following table.

Case Study	*Retrieved Candidates*	*Evaluated Candidates*	*Confirmed Clones*
WELTAB	10,392	8411	6499
COOK	82,655	46,288	5672

To be clear: we measure recall using the confirmed clones that Bellon selected from the candidates of all tools participating in his study. Precision, on the other hand, will be established using the confirmed clones reported by our own tool.

Note that, although the reference set is not homogeneous, and not necessarily complete, one can still obtain meaningful figures for recall and precision. To assess recall, one may use an arbitrary, sufficiently large set of confirmed clones. The reference set need not be complete. To assess precision, it suffices to manually examine a representative sample of the candidate clones detected. To determine the recall rate of our tool we have mapped the retrieved candidates to the reference clones by way of the mapping function defined by Bellon [Bel02a] with the same *OK* threshold of 0.7.

6.2.5 Results

We now summarize the results of our experiments. We present the numbers of candidate clones detected, recall for different categories of clone types, and precision.

	Normalization degree	WELTAB *Candidates*	COOK *Candidates*
–	No Normalization	1467	7661
C	Constants normalized	2255	12,434
I	Identifiers normalized	2565	29,333
IF	Identifiers, function names normalized	2608	38,141
CI	Constants, identifiers normalized	5946	38,581
CIF	Full Normalization	6334	49,789

Table 6.2: Retrieved Candidates for the different normalization degrees (maximum gap size = 2).

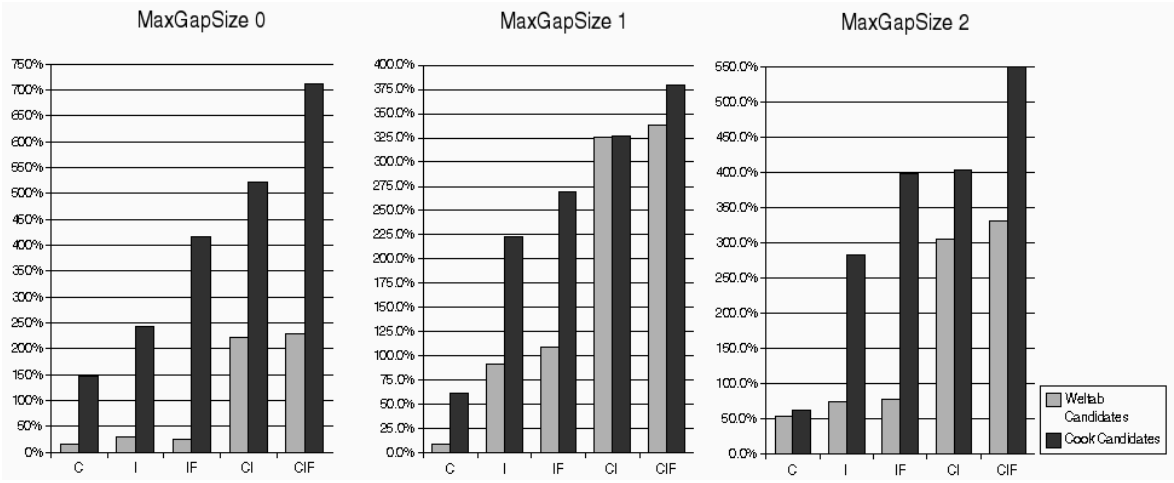


Figure 6.3: The increase in retrieved candidates, measured relative to the lowest degree of normalization.

Retrieved Candidates

Table 6.2 shows the number of the candidates identified with the largest setting of the maximum gap size. In Figure 6.3 we plot the percentage increase in the number of identified candidates that normalization brings, in comparison to clone detection without normalization. As expected, COOK shows a more-or-less steady increase in candidates identified as more normalization operations are applied. In WELTAB, however, we notice a considerable but puzzling jump when constants and identifiers are normalized jointly.

Recall by Clone Types

The effectiveness of a clone detection techniques will vary depending on how much a clone has been edited after copying. We adopt Bellon's classification of clone types in an effort to measure recall as a function of both editing operations and degrees of normalization.

In the WELTAB case study (see Figure 6.4), we see that overall recall (all types) increases from 78.2% to 95% when introducing more normalization. For Type 1 clones (identical clones), the recall rate is 100% at all degrees, as would be expected from exact string matching.

The lowest recall for non-normalized code is registered for type 2 clones (renamed identifiers). This can be explained by the observation that identifier changes are likely to occur systematically on most of the lines of a clone. Exact string matching will therefore miss every line thus modified, and consequently fail to identify the clone. With the normalization transformation, however, the recall rate rises by a remarkable 25% to a final level almost equal to that for type 1 clones. It does not reach 100% because we normalized constants and identifiers with different tokens which fails in the case when a constant parameter has been changed into a variable.

For type 3 clones (arbitrary edits), recall is initially higher than for type 2 clones. This can be explained by the fact that we take gaps (non-matching lines) into account when collecting the clones. However, since the

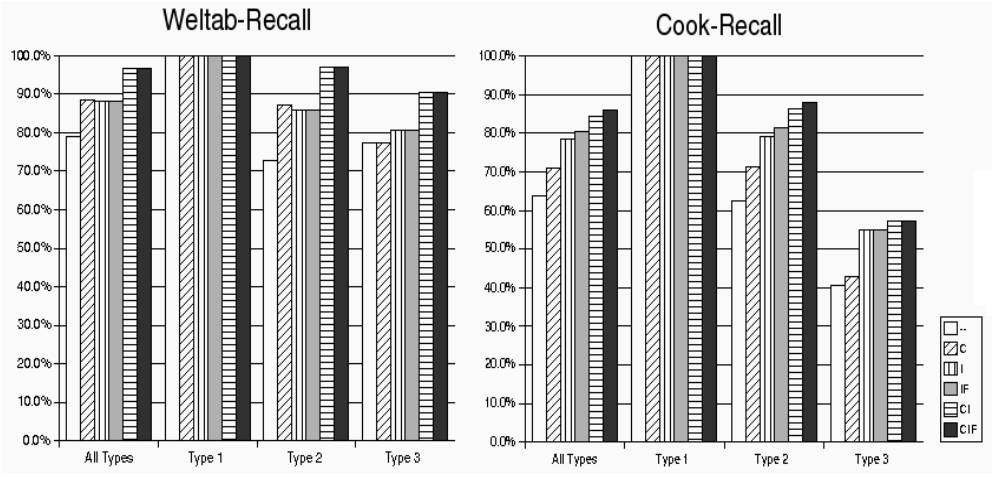


Figure 6.4: Recall for WELTAB and COOK split by clone type. Maximum gap size is 2.

normalization operations we perform are local to a single line, we cannot improve detection rates as much as for type 2 clones. Nevertheless, we achieve a recall rate of 90% at the most normalized degree.

In the COOK case study, the picture is similar. For type 1 clones, recall is 100% already at the lowest degree of normalization. For type 2 clones, recall rises from 63% to 88%, again by about 25%. Type 3 recognition is the worst for all degrees of normalization.

We investigated why the recall was so low for COOK type 3 clones and found a number of reasons:

- code that was syntactically too different to be recognized as a clone,
- altered source elements that we did not normalize, *e.g.*, type casts, pointer dereferences,
- altered formatting of source lines,
- source text inserted or deleted from the middle of a clone, and
- clones too small to be retrieved by our specification.

In Figure 6.5 we see how recall varies in response to increasing degrees of normalization. Increasing the maximum gap size from 0 to 1 improves recall significantly, whereas a maximum gap size of 2 has less impact. Normalizing constants improves recall for both WELTAB and COOK, whereas normalizing identifiers and function names is good only for COOK.

Precision

We now consider how precision varies with respect to the degrees of normalization. The studies illustrate well the common phenomenon that precision diminishes with increasing recall.

With the WELTAB case study, we observe a precision of 94% for non-normalized code, but this drops to 70% at the highest degree of normalization. The very high precision of WELTAB is consistent with the results of Bellon's experience that the confirmation rate for WELTAB candidates (coming from all the participating tools) was, at 90%, the highest among all the systems under study.

The situation is not so good in the COOK case study, where initial precision (*i.e.*, without normalization) is only 42% and drops to 11.5% for the highest degree of normalization. The latter clearly represents an unattractive level of noise for an engineer who is searching for refactoring opportunities.

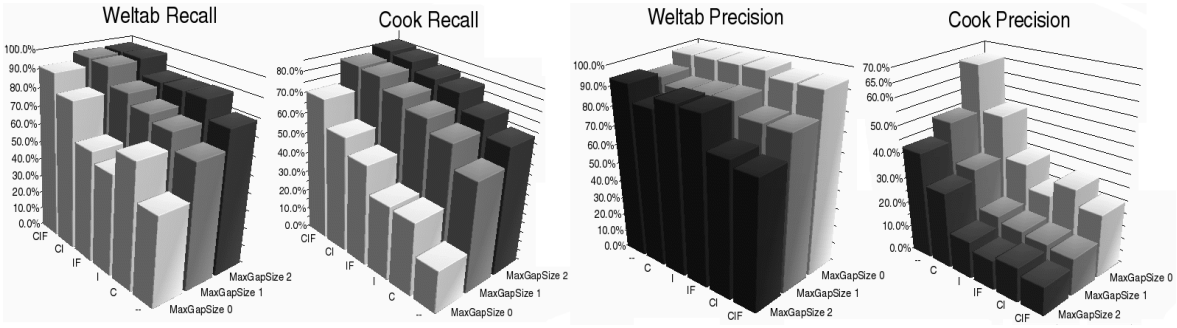


Figure 6.5: Recall and precision for WELTAB and COOK.

A similar drop in precision from the WELTAB to the COOK case studies was reported by Bellon for both our approach and that of Baker.

6.2.6 Discussion

Our evaluation shows that allowing for some variation in duplicated code is necessary to get decent recall. We were not able to conclude that gaps in clones or specific normalization of certain source elements is better. A maximum gap size of 1 yields good results, but allowing two lines as gaps can lead to an undesirable loss in precision. However, a similar drop in precision can result from aggressive normalization. In particular, normalizing function names can lead to a significant loss of precision that is not worth the minimal gain in recall.

One of the problems with the simple approach we promote is that large numbers of false positives can be generated. Multiple improvements are possible:

- Normalizing literal arrays, which are a source of many false positives in COOK, reduces the number of candidates.
- Taking into account function boundaries will remove many false positives which cross from one function into the other covering mostly boiler plate code, *i.e.*, `return`-statements followed by the header of the next function.
- Clustering all $n(n-1)/2$ clone pairs that are produced by n instances of the same source fragment into *clone classes* will reduce the number of instances that have to be investigated individually.

The strong variation between the two case studies suggests that future work should also focus on the analysis of variables that are outside the code itself such as the coding style, the programming language, the development process, or the programmer's education. Understanding these factors would help in tuning the detection for a particular system, improving the efficiency.

6.2.7 Other Systems of the Bellon Study

The Bellon case study comprised six more systems that were to be searched for clones (Figure 6.6). Whereas Bellon has reviewed manually 2% of the clone candidates found by the studys participants in all eight systems, we have used only two of the eight systems, manually assessing 80% (WELTAB) and 56% (COOK) of the clone candidates retrieved by our tool. To give an impression of how our approach performs with the other systems we present numbers that we could compute automatically without manual intervention.

We let our detector run on all six systems, setting the normalization degrees to the extremes (–) and (CIF) and fixing gap size at 2. We give the time the comparison took (in seconds), the number of clone candidates

CaseStudy	*Normalization Degree*	*Comparison*	*# of Candidates*	*Recall*
NETBEANS-JAVADOC	–	3s	198	49.1%
	CIF	15s	2804	87.3%
ECLIPSE-ANT	–	5s	201	56.7%
	CIF	30s	2099	90.0%
SWING	–	180s	5561	69.9%
	CIF	700s	34,659	85.3%
ECLIPSE-JDTCORE	–	240s	18,438	54.9%
	CIF	1380s	60,314	74.6%
POSTGRESQL	–	300s	20,294	66.3%
	CIF	3000s	41,083	71.5%
SNNS	–	180s	29,767	50.3%
	CIF	5400s	88,565	80.8%

Figure 6.6: Other Systems of the Bellon Study

retrieved, and the recall ratio with regard to Bellon's reference set for that system, computed again using the mapping function with the *OK* threshold of 0.7. Since Bellon's reference sets are not complete, precision can not be evaluated without manually investigating the clone candidates, which we refrained from due to time constraints.

Recall for unnormalized code is around 50% only in four cases, lower than all values for WELTAB and COOK. POSTGRESQL is an interesting case, as even with full normalization recall increases only by 5% and stays at a low 70%. Many of the clones from this system seem to have evolved quite for from each other.

6.3 Comparison with Other String-Based Approaches

When searching the literature for string-based clone detectors, we find at least the approaches of Johnson [Joh94a], Baker [Bak95a], Kamiya et al. [KKI02], and Cordy et al. [CDS04], which are closely comparable to our own. In this section we compare our approach with the ones of Baker and Kamiya et al. which have both been participating in the Bellon comparative study [Bel02b], thus providing data for a comparison. The main differences amongst these approaches are the selection of the *comparison granularity*, and the choices regarding *code normalization*:

- Baker [Bak95a] selects a single line as comparison unit, just as we do. She describes a mechanism to normalize identifiers and literal constants which respects the local context in which identifier names are changed. Two code sections are jointly recognized as duplicates if their identifiers have been systematically replaced, *i.e.*, `x` for `width` and `y` for `height`. This prevents some of the false positives that our more simple approach produces. She uses a suffix tree algorithm for comparison.
- Kamiya et al. [KKI02] work at the granularity level of individual tokens. They perform a subset of the code normalizations we have proposed in Section 4.2.3 and employ a suffix tree comparison algorithm similar to Baker's.

This section is structured as follows: We first report about the experiences with the string-based detectors that Bellon has made in his comparative study (Section 6.3.1). We then try to find the normalization degree in our case which comes closest to Baker's and Kamiya's results (Section 6.3.2). We finally mimic some of the unique features of the other two approaches, systematic identifier mapping in Baker's case (Section 6.3.3), and token- instead of line-based comparison in the case of Kamiya et al. (Section 6.3.4) and evaluate their influence on the results.

6.3.1 String Matching as Evaluated by Bellon

To evaluate our approach, we provided Bellon with results obtained from non-normalized source code, allowing for a gap of 1 line between matching lines. He categorizes our approach with the other string- and token-based approaches of Baker [Bak92] and Kamiya [KKI02] as having high recall but low precision.

In his summary, Bellon writes (page 123): *"We can put the tools into two coarse categories: the ones which have a high recall but a low precision and the ones that accept low recall to achieve high precision. (...) Together with Baker, Rieger reports about the same number of candidates. The rate of rejected candidates is also the same for these two participants."*

More detailed results from Bellon's study are shown in the next table. We list the differences of Baker and Kamiya to our own, setting our values as 100%.

Differences to Rieger	WELTAB		COOK	
	Baker	*Kamiya*	*Baker*	*Kamiya*
Retrieved Candidates	+988	+2144	-113	-6318
References matched with OK	+26	-66	-2	-105
References matched with GOOD	+50	-97	-1	-42
Precision	-0.003	-0.008	0	+0.03

Regarding the number of returned clone candidates, our approach was closest to Baker's which is normal since both approaches make line breaks a factor of comparison.

6.3.2 Comparison Against Different Normalization Degrees

We now wish to investigate how the approaches of Baker and Kamiya compare against the different degrees of normalization that we have introduced in Section 6.2.2. We chose the best result of Kamiya, *i.e.*, the voluntary

submission where some noise was removed by them. We did not remove any clone candidates from the results retrieved by our own tool.

To obtain a meaningful comparison, we first determine which choice of minimum gap size and normalization degree returns a comparable number of candidates. Then we analyze recall and precision[5] for this specific configuration.

Case Study	*Data*	*Baker*	*Rieger*		*Kamiya*	*Rieger*
WELTAB	# Cands	**2742**	2378 (IF, 1)	2608 (IF, 2)	**3898**	3761 (CIF, 0)
	Recall	80%	86%	88%	93%	92%
	Prec.	80%	90%	90%	99%	91%
COOK	# Cands	**8593**	9043 (CIF, 0)	7661 (–, 2)	**2388**	2764 (C, 0)
	Recall	70%	71%	64%	43%	36%
	Prec.	29%	26%	42%	42%	49%

Contrary to expectations, Baker exhibits a somewhat worse precision for WELTAB. For COOK, her precision is slightly better than ours, though we can significantly improve precision at a 7% cost in recall. We partly attribute Baker's loss in precision to some noise (`#include` statements) that she does not remove.

Kamiya on the other hand exhibits a better precision than we for comparable recall in WELTAB. In COOK, his recall is better but our precision is better for a comparable number of candidates.

In a second analysis, we identify configurations of our tool that exhibit similar precision to the other approaches, and then we compare recall and the number of retrieved candidates.

Case Study	*Data*	*Baker*	*Rieger*	*Kamiya*	*Rieger*	
WELTAB	Precision	**80**%	82% (CIF, 1)	**99**%	98% (IF, 0)	
	Recall	80%	96%	93%	61%	
	Candidates	2742	4973	3898	1414	
COOK	Precision	**29**%	30% (C, 2)	**45**%	42% (–, 2)	49% (C, 0)
	Recall	70%	89%	43%	64%	36%
	Candidates	8593	2255	2388	7601	2764

From these numbers a consistent ranking cannot be derived. We see that our simple approach can achieve results similar to the other two in all cases. For WELTAB, normalizing identifiers and function names seems to be important to obtain similar results. For COOK, however, normalizing identifiers results in too many candidates. We must therefore restrict ourselves to normalizing constants only, or setting the maximum gap size to 0. The set of applicable normalization operations is thus shown to depend on the system under study.

6.3.3 Impact of Systematic Identifier Normalization

Baker does not replace names of identifiers indiscriminately by one and the same token, but makes the consistent replacement of identifiers one criterion of the comparison. This avoids clone candidates which have the same syntactic structure but differing identifier usage and is an early filter against false positives.

By filtering out candidate clones where identifiers are mapped inconsistently (according to Baker's description in [Bak95a]) we can derive how much precision is lost when uniformly normalizing identifier names. At gap size 0 we get the following percentages of candidates which exhibit inconsistent identifier mapping:

[5] We evaluated Baker's and Kamiya's candidates in the same manner as our own (see Section 6.2.4).

Normalization Degree	WELTAB	COOK
–	0.0%	0.0%
C	0.0%	0.1%
I	1.0%	5.8%
IF	1.1%	7.3%
CI	3.8%	7.3%
CIF	3.8%	9.6%

The more we normalize identifiers the more inconsistency is naturally found. Filtering ten percent of the retrieved candidates could certainly be interesting if they all would prove to be false positives. The merit of using this characteristic as filtering criterion is however less clear for clones where the two copied fragments are more distant, as can be found among the results of, for example, metrics based methods. Finding inconsistently mapped identifiers is then no longer very effective at spotting false positives. From the confirmed clones of Bellon's study we can flag as having inconsistent identifier mappings 20.2% of WELTAB and 20.5% of COOK references.

6.3.4 Impact of Token Based Comparison

Kamiya, rather than using source lines, compares the code on the granularity level of tokens. This avoids problems with *line break relocation* where only the layout of the code is changed. The smaller granularity however also means that more entities must be processed.

When investigating the references that were detected only by Kamiya but not by us, there was only one example of a clone where layout changes prevented the line-based comparison from detection. This fact can however not be generalized, since it is influenced by the particular construction of the reference set and the characteristics of the case studies. In their own investigations Kamiya et al. [KKI02] have reported that as much as 23% of the clones found by the token-based comparison exhibited line break relocation. A detailed investigation of the impact of line break relocation on our results is presented in Section 6.4.

6.4 Assessing the Impact of Pretty-Printing

A clone detection approach that takes the lines of code as they appear in the original source text is vulnerable to *line break relocation*, *i.e.*, programmers formatting the same code in different ways. This can occur if developers with different programming styles reuse each others code. Also, source lines are usually broken after 80 characters to maintain readability in editor windows. The lengths of identifier names, string constants, or the depth of indentation levels can therefore become responsible for layout differences between copied code. The following two fragments from the ECLIPSE-ANT system which are the same except for identifier names and linebreaks are an example:

```
public void setProperty(String name, String value) {
 if (null != userProperties.get(name)) {
  log("Override ignored for user prop." + name, MSG_VERB);
 }
 log("Setting project property: " + name + " -> " +
     value, MSG_DBG);
 properties.put(name, value); }
```

```
public void addReference(String name, Object value) {
 if (null != references.get(name)) {
  log("Overriding previous def. of reference to " + name,
      MSG_WARN); }
 log("Adding reference: " + name + " -> "+value, MSG_DBG);
 references.put(name,value); }
```

An obvious solution to the problem of line break relocation is to transform the source code into a normalized layout before the comparison by a *pretty printer* in the widest sense of the word. Pretty printers usually strive to make the code readable for a human. This means that in addition to enforce a common layout for all instances of a language constructs, tricky issues like indentation or the aforementioned problem of maximum line length have to be considered as well. These finer aspects can be neglected if we want to just bring the code in a normal form for line-by-line comparison.

In order to get an impression of the importance of the problem of missing duplication because of line break relocation, we conduct an experiment detecting duplication in layout normalized code. We want to answer the following two questions:

1. How many clones we detect contain line break relocations between their two source fragments?
2. How many of the clones which contain line break relocations are (partially) detected in the original code?

We do not expect to derive a common percentage valid for all the different systems that will be participating in the experiment, since everything from layout style to average identifier length to layout discipline differs from programmer to programmer. What we expect is an indication of the prevalence of the problem. We want to assess if it is advisable to apply the layout normalization to all systems before comparison.

The structure of this section is the following: We explain the experimental setup in the next section, then present the systems we have used in the study, then characterize the results using a few statistics.

6.4.1 Experimental Setup

We take a system and apply a number of normalizations. This version is called the *original* code, the set of clones detected in this code is called *OC*. A pretty-printer is then applied to the *original* code, transforming it into *layout-normalized* code. We map the clones detected in the *layout-normalized* code back to the line numbers of the *original* and are then able to count how many of these clones contain misaligned line breaks, a set we call *LBC*. The size of *LBC* is the answer to the first question formulated above. To determine how many

```
if(line.indexOf("--") >= 0) sql += "\n";
if(dType.equals(DelimType.NORMAL) && sql.endsWith(del) ||
   dType.equals(DelimType.ROW) && line.equals(del)) {
 log("SQL: " + sql, Project.MSG_VERBOSE);
 execSQL(sql.substring(0,sql.length()-del.length()),out);
 sql = ""; }
```

```
if(p.indexOf(p) >= p)
   p += p;
if(
      p.equals(T.T)
   &&
      p.endsWith(p)
   ||
      p.equals(T.T)
   &&
      p.equals(p)) {
   log(
        p + p
       ,
        T.T);
   execSQL(
             p.substring(
                           p
                         ,
                           p.length() - p.length())
           ,
             p);
   p = p;
}
```

Figure 6.7: A fragment of ECLIPSE-ANT code before and after the application of code normalization and layout normalization (the indentation in the layout-normalized version is done by hand to enhance readability).

clones from *LBC* we can also detect in the *original* code, we determine the recall rate when mapping *OC* to *LBC* with the same mapping function used in Section 6.2.4. The clones that are not in the recall of the mapping function will show us how much duplication we loose when not applying layout-normalization to the source code.

Code Normalization

We have chosen to compare normalized code in order to get a reasonable amount of duplication, *i.e.*, to increase the chances for clones which contain relocated line breaks to be detected at all. We normalize the following elements of the code:

- Names of variables (*not* function names).
- Literal constants (numbers, characters, strings).
- Labels in C code.
- Literal Arrays (their content is completely removed).

The decision to remove literal arrays was taken since, first, copies of literal arrays, especially in normalized code, are uninteresting. Second, in some systems the copies of literal arrays exhibited most of the relocated line breaks, thereby disturbing the results.

Layout Normalization

For layout normalization we first remove all line breaks from the original code. Line breaks are then reinserted according to a few simple rules. The most important rule is that a line break only appears after the end of a statement or a block. There are three exceptions to this rule:

i) Conditionals of **`if`**-statements, **`for`**- and **`while`**-loops appear alone on a line.

ii) Boolean expressions with `&&` or `||` operators separate their subexpressions over multiple lines.

iii) Parameter lists which contain subexpressions instead of only variable names are spread out over multiple lines (the token '`,`' acting as separator).

The goal of rules *ii)* and *iii)* is less layout normalization and more the increase of the detection rate, to keep a change in a subexpression from 'polluting' the entire statement. For an example see Figure 6.7. The layout normalization process is performed by a simple parser which can be configured for many languages by giving the tokens acting as statement-, block-, expression-, and list element delimiters, as well as some of the operators mentioned in the list of rules above.

Clone Characteristics

We retrieve clones of the standard length 6 line or longer, and we allow for gaps of at most size one. Allowing gaps in the retrieved clones means that more duplication is being reported, increasing the chances of encountering some with relocated line breaks. To assess an eventual influence of the gap we also retrieve clones without gaps.

Selected Systems

We select a range of small- to medium sized systems, written in JAVA and C, from different sources and with different layout styles.

System	*Language*	*Origin*
NETBEANS-JAVADOC	JAVA	Open Source
WELTAB	C	Industry
AGREP	C	Academia
ECLIPSE-ANT	JAVA	Open Source
COOK	C	Open Source
APACHE	C	Open Source

6.4.2 Results

The results of the case study will be presented under two aspects: *i)* the effect that the layout normalization has on the original code in terms of number of line breaks to give an impressions of the amount of change the code undergoes in the normalization, *ii)* the number of clones that we detect that contain line break relocations (the set *LBC* mentioned above), and *iii)* the number of clones from *LBC* which we cannot detect in the *original* source code.

The increase of number of lines by the layout normalization ranges from 30% to 65%. The increase of atomic matches in the comparison matrix is significant, more than doubling in three cases. For COOK the small increase of matches is due to the fact that the original layout of the code is already very loose, and that in order to speed up comparison the term containing a single variable name (`'p'`) has been ignored in the comparisons, making the brunt of the matches increase invisible.

System	*No Gaps*	*Gap Size 1*
NETBEANS-JAVADOC	0	0
WELTAB	12 (0.1%)	55 (0.4%)
AGREP	7 (0.2%)	9 (0.2%)
ECLIPSE-ANT	15 (6.4%)	31 (8.8%)
COOK	8 (0.7%)	28 (1.2%)
APACHE	17 (0.4%)	48 (1.1%)

Table 6.3: The number of clone candidates that contain line break relocation (*LBC*).

System	*No Gaps*	*Gap Size 1*
WELTAB	12 (100%)	22 (39.7%)
AGREP	6 (85.7%)	9 (100%)
ECLIPSE-ANT	5 (32.3%)	10 (32.3%)
COOK	4 (50.0%)	20 (71.4%)
APACHE	10 (58.8%)	22 (45.8%)

Table 6.4: The number and percentage of *LBC* candidates which are also detected in the original code.

System	*Original Lines*	*Lines Increase*	*Matches Increase*
NETBEANS-JAVADOC	9000	41%	46%
WELTAB	10000	54%	106%
AGREP	12000	42%	166%
ECLIPSE-ANT	16000	64%	123%
COOK	43000	29%	6%
APACHE	63000	39%	84%

The number of clones that contain line break relocations (size of *LBC*) as shown in Table 6.3 is the answer to the first question of our experiment. The percentage number is relative to the set of all retrieved candidates, a set which includes an unknown but potentially large number of false positives. We did not remove the false positives from this set and neither from *LBC*. In most of the investigated systems, the percentage of clones having line break relocated source fragments is very low. Only in ECLIPSE-ANT we find an elevated percentage, which is due to the low number of clones detected all in all. We see a tendency of increased number of *LBC* when allowing gaps in the clones. This is to be expected when more difference is allowed between the source fragments.

Table 6.4 shows the recall rates when mapping *OC* to *LBC*. This means, for example in the AGREP system, we detect 6 out of 7 clones containing line break relocations in the *original* code as well. The generally high recall rates indicate that we can get hints at many of the (especially longer) clones from *LBC* by looking at the *original* code, that line-break relocation does not hide many clones from our view.

6.4.3 Discussion

In every system that we have investigated for this experiment there are clones which contain line break relocation. The percentage of these clones is however small, rarely rising over 1% of all candidates. We have also seen that on average 65.4% (clones without gaps) and 57.8% (clones with gapsize 1) of the line break relocation affected candidates will still be caught in the original code. Most of the clones which we do not detect are short ones, as larger clones offer enough surrounding context that does not contain relocated line breaks. We therefore conclude that even though line based detection misses some duplication, line break relocation is not preventing us from detecting the major part of duplication using a line-based comparison. And even if we feel that there is a significant portion of duplication withheld from detection by the line break relocation, we have the option to use pretty printing techniques for source code normalization.

6.5 Validation of Clone Ranking

In this section we report on an initial experiment to assess some of the clone ranking measures proposed in Section 4.4. We want to assess if the measures improve the handling of clones for the user by providing a ranked list, letting the user investigate the most likely clone candidates first.

We will first explain the experimental setup, and then perform the experiment on different candidate clone sets, some specially evaluated for this experiment, and some evaluated by different evaluators.

6.5.1 Experimental Setup

We take sets of clone candidates which have been evaluated to determine the relevance, manually sorting them into 'confirmed clones' and 'confirmed non-clones'. We then rank the set using one of the metrics proposed below. Ideally, the ranking should put every confirmed clone before every confirmed non-clone in the list. In practice, however, both categories will be intermixed, due to failure of the metrics in measuring the aspects which determines the relevance of a clone, and in some cases also due to faulty evaluations of the candidates.

As the characterizing number for a ranking we take the median for each of the two categories. The median of the confirmed clones, for example, indicates that 50% of the clones lie above this mark. From the number of confirmed clones and non-clones we can derive the optimal positions of the two medians. The ranking which pushes the actual medians as close as possible to the optimal medians is the most successful.

Ranking Measures

We use three measures to rank the clone candidates:

LOC: This measures the length of the clone, regardless of gaps.

LOC is used as a baseline since it has been used frequently to rank clones. That it has some merit is obvious if one considers that a clone of 50 lines is most likely relevant, whereas a clone of only four lines will be a false positive with high probability. Since this metric is insensitive to many other aspects which determine clone relevance, it is also easy to improve upon it.

CCGW (Combined Code and Gap Weight): This computes the weight of matched code and subtracts the weight of the code in the gaps (see Section 4.4.3). The measure determines relevance from the point of view of how likely and easy it is to remove the duplication by a refactoring.

CCGW': This is a variant of CCGW which is more sensitive to elements that are hindering immediate refactoring of a clone. The measure tracks the occurrence of some specific elements on only one side of a gap and reacts with a larger reduction of the weight. For example, finding one of the keywords **`if`**, **`else`**, **`do`**, **`for`**, or **`while`** on only one side of a gap spells of a rather important difference between the two fragments, making refactoring potentially harder. Function invocations[6] that occur on only one of the corresponding lines also speak of a considerable difference and are therefore cause for weight reduction.

Figure 6.8 gives an initial impression of the improvement that the proposed measures have on the baseline LOC measure. Whereas for LOC the number of clones and non-clones is similar for every percentile of the ranking with only a slight surplus of clones in the three first percentiles, the CCGW' measures clearly pushes the clones to the front and the non-clones to the back of the ranking.

[6] We employ a simple nGram-based fuzzy string comparison for function names. For example, the names `printf` and `fprintf` are considered to be the same.

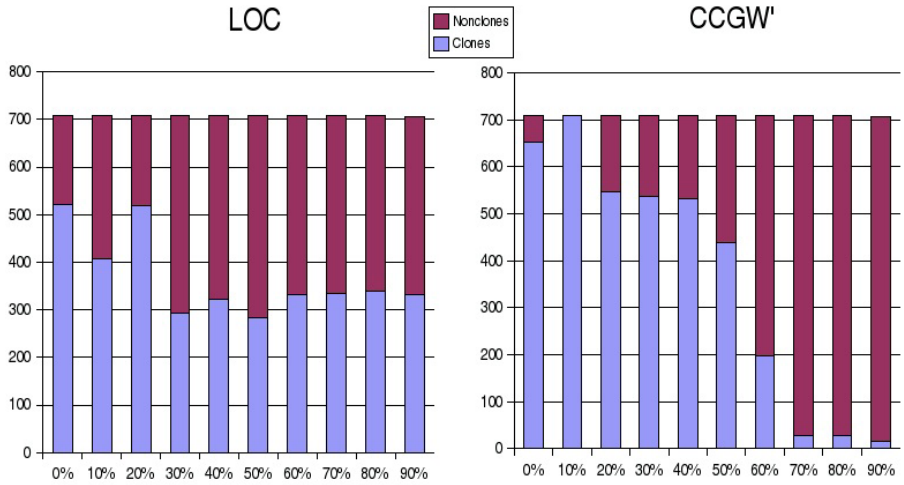


Figure 6.8: Two ranked lists of WELTAB candidates shown as frequency distributions of confirmed clones and non-clones in ten percentiles. The ability to distinguish clones from non-clones is clearly superior for the CCGW' measure.

Candidate Sets

Two candidate sets, one from the WELTAB system and the other from COOK, have been evaluated specially for this experiment. We have normalized the following syntax element before the comparison: variable names, literal constants, labels, and literal arrays. For COOK we did not normalize certain type names and removed all code that was not contained within a function body in order to reduce the number of candidates. For WELTAB we selected all clones with minimal length 6 and maximal gap size 2. For COOK we selected all clones with minimal length 7 lines and maximal gap size 2.

Candidate Set	*Size*	*Confirmed Clones*		$\tilde{c}$	$\tilde{nc}$
WELTAB	7087	3684	(52%)	1842	5385
COOK	4583	2153	(47%)	1076	3368

The optimal median of the confirmed clones $\tilde{c}$ and the optimal median of the confirmed non-clones $\tilde{nc}$ is directly derived from the number of confirmed clones. The actual medians resulting from the rankings are compared to these optimal medians below.

6.5.2 Results

We describe the ranking of the candidate sets by the three measures with the distance from the actual medians $\tilde{c}^*$ and $\tilde{nc}^*$ to the optimal medians. The accumulated distance gives a single number characterizing the accurateness of the measure.

Candidates	*Measure*	$\lvert\tilde{c}^* - \tilde{c}\rvert$		$\lvert\tilde{nc}^* - \tilde{nc}\rvert$		*Total Distance*	
WELTAB	LOC	1097	(15.5%)	1444	(20.4%)	2541	(35.9%)
	CCGW	422	(6.0%)	217	(3.1%)	639	(9.0%)
	CCGW'	219	(3.1%)	58	(0.8%)	277	(3.9%)
COOK	LOC	570	(12.4%)	334	(7.3%)	904	(19.7%)
	CCGW	132	(2.9%)	16	(0.4%)	148	(3.2%)
	CCGW'	115	(2.5%)	62	(1.4%)	177	(3.9%)

The CCGW ranking improves markedly on the simple LOC ranking in all cases. For WELTAB, the CCGW' variant improves the ranking yet again for clones and non-clones. For COOK, the non-clones median $\tilde{nc}^*$ is worse for CCGW' than for CCGW. When looking at the clones which have been pushed back by CCGW' but not by CCGW, we find that many of the confirmed clones have been declared so based on a semantic similarity which is expressed in idiomatic structures and therefore recognized by the detector. The central part of the idiom, consisting in an assignment or function call, is however frequently so different as to make the clone candidates unfit for refactoring. The CCGW' measure is able to point out these differences. Another frequent problem is also that code that is structurally the same differs in nested function calls, as can be seen in the following example:

```
if(ewhite)
 while (cp > cp1 && wc_find(white, cp[-1]))
   cp--;
```

```
if(ewhite)
  while (cp > cp1 && isspace(cp[-1]))
    cp--;
```

It is a possibility to change the CCGW' measure in order to weigh nested invocations differently from top level invocations.

Finally, it happens that some copied code which was confirmed to be a clone has, before or after the bulk of the copied code, a number of spuriously matching lines which drag the weight down.

6.5.3 Investigation of Fixed-Length Clones

As we have seen above, the length of a clone is a predictor of clone relevance, albeit not a very good one. The CCGW and CCGW' measures also profit from the additional weight of more lines, *i.e.*, they too have a tendency to push longer clones towards the front of the ranking. An influence of the clone length on all of the measures used here can therefore not be denied. To remove this influence we repeat the experiment on a number of sets which contain candidates that are all of the same length. All of these sets are extracted from the main candidate sets introduced above.

Candidate Set	*Size*	*Confirmed Clones*		$\tilde{c}$	$\tilde{nc}$
WELTAB Length 6	1972	918	(46.6%)	459	1445
WELTAB Length 7	1502	679	(45.2%)	340	1090
WELTAB Length 8	1086	512	(47.1%)	255	799
COOK Length 7	964	283	(29.4%)	142	623
COOK Length 8	775	92	(11.9%)	46	433
COOK Length 9	848	497	(58.6%)	249	672

Note that the ranking using LOC as measure is basically random for clones of the same length. We therefore do not show the values for the LOC measure anymore in the tables below.

Candidates	*Measure*	$\lvert\tilde{c}^* - \tilde{c}\rvert$		$\lvert\tilde{nc}^* - \tilde{nc}\rvert$		*Total Distance*	
WELTAB L6	CCGW	135	(6.9%)	131	(6.6%)	266	(13.5%)
	CCGW'	223	(11.3%)	28	(1.4%)	251	(12.7%)
WELTAB L7	CCGW	18	(1.2%)	9	(0.6%)	27	(1.8%)
	CCGW'	17	(1.1%)	9	(0.6%)	26	(1.7%)
WELTAB L8	CCGW	0	(0.0%)	28	(2.6%)	28	(2.6%)
	CCGW'	0	(0.0%)	7	(0.7%)	7	(0.7%)

For WELTAB, the CCGW measures are quite accurate for length 7 and 8, and are as good or better as the numbers for the entire candidate set. For length 6, the CCGW' measure, while improving the accuracy for the

non-clone median $\tilde{nc}^*$, reduces the accuracy of the clone median $\tilde{c}^*$ by 5% with respect to CCGW. Upon close investigation of the ranked clones, we find that a large number of confirmed clones with an **`if`** statement in only one of the two fragments are pushed back by the CCGW' measure, rightfully indicating an impedance for immediate refactoring. The ranks thus emptied are filled with clone candidates which were evaluated as non-clones due to a subtlety not recognized by the measures. Rather than the evaluation casting doubt on the measure, the CCGW' measure thus puts the evaluation in question.

Candidates	*Measure*	$\lvert\tilde{c}^* - \tilde{c}\rvert$		$\lvert\tilde{nc}^* - \tilde{nc}\rvert$		*Total Distance*	
COOK L7	CCGW	2	(0.2%)	5	(0.5%)	7	(0.7%)
	CCGW'	5	(0.5%)	30	(3.1%)	35	(3.6%)
COOK L8	CCGW	15	(1.9%)	1	(0.1%)	16	(2.1%)
	CCGW'	13	(1.7%)	4	(0.5%)	17	(2.2%)
COOK L9	CCGW	53	(6.3%)	6	(0.7%)	59	(7.0%)
	CCGW'	32	(3.8%)	4	(0.4%)	36	(4.3%)

The measures CCGW and CCGW' are very accurate for the given lengths, almost reaching the optimal clone and non-clone medians. Except for Length 9, the values are as good or better than for the entire candidate set. Investigating the problem with the Length 9 candidate set, we could identify a number of features found in many candidates where the automatically derived weight was much higher than the weight the evaluator gave the code. This pushed confirmed non-clones up in the ranking. Explicitely removing these features from the ranking algorithm we were able to align the ranking better with the human evaluation (reduce the problematic difference from 1.4% to 0.5%)

Another slight anomaly can be seen for the non-clone median at Length 7 where CCGW' reduces accuracy by 2.6%. We find that the confirmed clones which are pushed down by CCGW' actually have significant differences and should probably be declared non-clones. This would move up the non-clone median $\tilde{nc}$, making CCGW' more accurate than CCGW.

6.5.4 Influence of Evaluation Bias

Manual evaluation of clones is difficult and subject to many biases. It is, for example, true that knowledge of the measure which is used for ranking the clones can guide the evaluation process unconsciously. Since the evaluations of the WELTAB and COOK candidate sets were performed by a knowledgeable evaluator, the influence cannot be ruled out. It is also in general not easy to reach an agreement on the status of a given candidate among a number of evaluators [WJL+03]. Different evaluators have different sets of criteria for clone relevance. Chances are that a ranking measure models the criteria of one evaluator better than the criteria of another evaluator.

To get a broader impression of the performance of ranking measures we rank candidate sets which have been evaluated by a number of evaluators that did not have any knowledge of the CCGW and CCGW' measures. The candidate sets of each evaluator are not identical, making a direct comparison impossible. However, the evaluated sets were randomly assembled from a large set of candidates containing many clones from the same class. Each set therefore contains a comparable selection of candidates.

Per-Evaluator Results for the WELTAB System

The set of evaluated clone candidates from the WELTAB system were comparably small with around 250 candidates each. The evaluations resulted in elevated ratios for confirmed clones for every evaluator.

Evaluator	*Size*	*Confirmed Clones*		*Total Distance* CCGW	CCGW'
EVALUATOR A	247	138	(55.9%)	16.2%	10.9%
EVALUATOR B	247	135	(54.7%)	4.5%	3.2%
EVALUATOR C	250	141	(56.4%)	5.2%	4.0%
EVALUATOR D	250	182	(72.8%)	2.4%	4.0%
EVALUATOR E	253	162	(64.0%)	1.2%	2.4%
EVALUATOR F	249	109	(43.8%)	2.4%	1.6%
EVALUATOR G	245	125	(51.0%)	5.7%	6.1%

The CCGW is effective at predicting the ranking of every evaluator, with the exception of EVALUATOR A. When looking at the details of EVALUATOR A's evaluation we found that the optimal clone median $\tilde{c}$ is hit accurately by both measures CCGW and CCGW'. The inaccuracy is due to a large difference $|\tilde{nc}^* - \tilde{nc}|$. Upon closer investigation we find that about two thirds of EVALUATOR A's confirmed clones that are placed below $\tilde{nc}^*$ do not allow easy refactoring, vindicating the ranking of CCGW and CCGW'.

The difference between CCGW and CCGW' is mostly insignificant, whether CCGW' is improving upon CCGW or not (in EVALUATOR D's case, the difference is only three candidates on the side of the non-clone median).

Per-Evaluator Results for the COOK System

The candidate sets from the COOK system were of a medium size with a bit under 2000 clone candidates. The evaluations show a consistently low percentage of confirmed clones for every evaluator.

Evaluator	*Size*	*Confirmed Clones*		*Total Distance* CCGW	CCGW'
EVALUATOR A	1811	286	(15.8%)	12.5%	12.5%
EVALUATOR B	1806	106	(5.7%)	11.2%	10.1%
EVALUATOR C	1815	171	(9.4%)	9.9%	8.5%
EVALUATOR D	1805	230	(12.7%)	22.2%	24.3%
EVALUATOR E	1818	326	(17.9%)	7.1%	7.8%
EVALUATOR F	1810	89	(4.9%)	12.6%	11.0%
EVALUATOR G	1807	166	(9.2%)	13.5%	12.5%

The accumulated distances from actual to optimal medians for the CCGW and CCGW' measures are unusually high for the COOK candidates. Some reasons for the misalignment between ranking and evaluation are:

- Evaluators dismissed copied fragments of densely repetitive code (two or three consecutive lines repeated with minimal changes), copied within the same function.
- Evaluators dismissed candidates consisting mostly in variable declarations, being only the header of a function, or consisting only in elements of a literal array.
- In EVALUATOR D's case, where the largest differences are found, a third of the confirmed non-clones above $\tilde{c}^*$ could be also declared as clones.

This shows that the measures are insensitive to a number of criteria that evaluators apply in their decisions.

6.5.5 Discussion

The experiment has shown that CCGW and CCGW', a simple set of complexity measures constructed with only the knowledge about keywords and operators of a programming language, are able to predict the aspect of refactorability to a good extent and even in cases correct a superficial human evaluation.

The measures are shown to significantly improve upon the simple line count. The difference between CCGW and the variant CCGW' is however not that big. In most of the cases CCGW' improves upon the results of CCGW, sometimes considerably. The reverse effect is however also possible. The justification to use CCGW' is that it is sensitive to some obvious impediments for refactoring measures

In this experiment the weights have all been derived from the frequency of the individual elements or features. We have in only one case tuned the weights manually to adjust the measure to some specific relevance criteria used by an evaluator.

The experiment conducted here is only a preliminary assessment of the measures as it has the following problems:

- The accuracy of the candidate evaluations is not assessed.
- The performance of the reviewers is not uniform. There was no common agreement upon what a clone consists in. The evaluators were also experiencing a learning process but did not revise their earlier decisions.
- The construction of the ranking measures was known before some of the evaluations, making evaluations biased in favor of the relevance criteria that were actually assessed by the measures.
- There are many influences that are not controlled, mostly pertaining to the systems used as sources of clone candidates.

6.6 Investigation of Scalability

Recalling the description of Section 4.3.2 we briefly list the contributions of each step to time and space requirements of the algorithm:

1. The construction of the index is linear in input size.
2. The construction of the comparison matrix is quadratic in the number of duplicated comparison units.
3. The analysis of the comparison matrix is linear in the number of atomic matches.
4. Any analysis of the extracted clones, like building clone classes or ranking, are linear in the number of clones.

By far the largest contributor is the second step. This is mainly due to the number of atomic matches which dwarfs every other size in the algorithm. It is also due to the complicated operations of memory allocation and establishing multiple links between cells of a sparse matrix which only happen during the construction of the comparison matrix, and not during the extraction of the clones from the matrix.

We first illustrate the time and space behavior of the algorithm with data gathered during an experiment and the discuss some aspects of the comparison matrix.

6.6.1 Experimental Data

The time and space measurements we report were taken in the course of the normalization experiment (see Section 6.2). Table 6.5 shows values for the different normalization degrees of the WELTAB and COOK system.[7] The data not only describes how the approach handles systems of different sizes, but also how more normalization results in less specific code and a less diverse vocabulary. This increases the number of matches significantly.

The COOK system especially illustrates how the greater uniformity of highly normalized code increases the number of atomic matches. As stated, the comparison and filtering phases are mostly determined by the number of atomic matches.

[7] The platform used for the experiment was a 2.1 GHz AMD Athlon with 550 MB of memory running Linux 2.4 and VisualWorks SMALLTALK 7.1.

System	*Normalization Degree*	*Atomic Matches*	*Time*		*Space*
			Comparison	*Filtering*	
WELTAB	–	260,710	5.8s	0.7s	2.2MB
	C	388,732	6.4s	0.9s	2.4MB
	I	365,289	7.0s	0.8s	3.2MB
	IF	429,226	7.3s	0.8s	3.4MB
	CI	649,817	8.8s	1.1s	3.8MB
	CIF	726,470	9.5s	1.2s	4.2MB
COOK	–	1,485,630	150s	15.8s	59MB
	C	2,166,832	183s	22.8s	88MB
	I	5,771,343	469s	61.3s	252MB
	IF	6,825,959	690s	64.3s	330MB
	CI	6,629,632	546s	63.7s	289MB
	CIF	7,703,927	750s	88.8s	380MB

Table 6.5: Experimental samples of time and space requirements.

System	*Granularity*	*Input Size*	*Distinct Terms*	*Atomic Matches*
WELTAB	Line	9847	2251	260,673
	Token	80,601	1230	146,255,494
COOK	Line	46,645	21288	1,485,630
	Token	278,390	11567	1,297,446,919

Table 6.6: Vocabulary size decreases, input size and number of matches increase when using tokens instead of lines as comparison unit.

System	*Atomic Matches*	*% Matches by n Most Frequent Terms*		
		$n = 2$	$n = 5$	$n = 10$
POSTGRESQL	18,900,000	76%	88%	93%
ECLIPSE-JDTCORE	5,800,000	61%	77%	84%
SNNS	3,400,000	59%	70%	77%
SWING	2,800,000	45%	69%	78%
COOK	1,700,000	48%	62%	72%
AGREP	480,000	45%	67%	79%
WELTAB	140,000	15%	26%	38%
ECLIPSE-ANT	70,000	22%	34%	48%
NETBEANS-JAVADOC	40,000	25%	40%	53%

Table 6.7: Number of atomic matches generated by the most frequent terms.

6.6.2 Impact of Comparison Unit Granularity

If the granularity of the comparison is reduced (from lines to tokens to characters), not only does the size of the comparison matrix grow for a given system, but more importantly the comparison generates more atomic matches. The reason for this is the reduced variability of the vocabulary (shorter terms mean less distinct terms) and the therefore increased frequency.

The effects of using tokens instead of lines can be observed in Table 6.6. For WELTAB the increased input size (8.1 times more) and the reduced term variability (46% less terms) result in a 560 times increased match count. For COOK, input size increases 6 times, the vocabulary shrinks by 46% and the number of matches increases 873 times. It is thus clear that using tokens instead of lines will lead instantly to a multiplication of any existing scalability problems.

6.6.3 Impact of Frequent Terms

The distribution of frequencies for the terms in the vocabulary is highly skewed: on the one hand, around 70% to 80% of the vocabulary terms occur only once, and on the other hand most of the matches are generated by a few very frequent terms. Table 6.7 shows the total number of matches for a number of systems (without any normalizations applied to the code) and the percentages of matches that are generated by the 2, 5, and 10 most frequent terms. The tendency we can observe for this sample is that the larger the total number of matches is the more skewed the frequency distribution gets.

The advantage of such a distribution is that we are able to significantly reduce the number of atomic matches that need to be handled by removing a small number of terms from the vocabulary [CH93]. The most frequent terms are usually uninteresting in themselves, *e.g.*, '`return false;`', '`break;`', or '`int i;`'. What is lost by removing such terms is therefore mostly boilerplate duplication.

6.6.4 Discussion

Creating and storing atomic matches in a comparison matrix is of quadratic complexity and the major problem in terms of scalability. The size and diversity of the vocabulary has a large impact on the number of matches that have to be handled. One of the justifications to use lines as comparison unit granularity is therefore the relative diversity of its vocabulary with respect to the more uniform token vocabulary. Knowing which are the most frequent terms and their weight we are able to remove noise prior to the comparison. The separation of source representation (the vocabulary) and duplication representation (the comparison matrix) is also advantageous if we want to implement incremental strategies as discussed in Section 4.3.2. After a batch of source code has been compared and the clones have been extracted, we can delete the comparison matrix again.

6.7 Conclusions

We list the conclusions on the different aspects presented in this chapter separately.

Language Independence

We can implement the techniques for a wide variety for languages. We can make most of the implementation configurable, so no new implementation is needed when switching to a different language.

Some older languages with non-standard features like allowing keywords as variable names foil the simplicity of lexical analysis.

Code Normalization

Code Normalization is necessary to get relevant duplication. It is however important to find a balance, since too much normalization lets the precision plummet. Allowing gaps of mismatching code in the middle of a duplicated fragment is a good way to cope with changes that exceed the scope of the normalizations. A gap size of 2, however, is already allowing too much difference which makes many smaller clone candidates invalid.

Comparison with Other Approaches

String-based clone detection can be categorized as having "high recall and low precision" [Bel02a] in general. We have shown that our approach is close to other well known string-based detectors, and closest to Baker's line-oriented detection method.

Assessment of Line-Orientation

We have seen that line-break relocation does not affect recall significantly for all 6 sample systems under investigation. Also, it is again very simple to build a generic pretty printer which can be configured easily for the majority of programming languages and which produces a normalized code layout.

Ranking of Clone Candidates

Using a more sophisticated code representation in the ranking phase than in the comparison phase has proven to significantly improve the ability to present the reengineer with a list of the relevant clones.

Scalability

The chosen granularity of a single line of code lets us treat small to moderate systems easily. For very large systems the number of atomic matches that is generated during the comparison can grow too large to keep it in memory all at once. It becomes necessary to selectively remove overly frequent but unimportant lines, and eventually employ incremental strategies.

Chapter 7

Conclusions

With this thesis we set out to develop a lightweight approach to code duplication detection which is applicable to systems of different sizes and languages. In this chapter we summarize the contributions that we have made towards this goal. We finally list some ideas for future work.

7.1 Contributions

Duplication of source code is a problem that arises in every software project, mostly due to the pressures of delivering a working system in a certain time span. Since it can have grave consequences—increased maintenance costs and error risks—duplicated code must be dealt with in every phase of the software engineering life cycle: during development when we want to be warned of introducing duplicated functionality, during maintenance when we want to make sure that we update all occurrences of an irregular source fragment, and during reengineering phases when we want to remove accumulated duplication.

We have presented the field of duplication detection in detail. We have shown the ties between code duplication detection and the old field of information retrieval and emphasized the importance of the notion of *relevance*. We have explained the different conceptual entities which describe the field of clone detection: How the programs source text is split into *source units*, the independent entities which are not ordered but contain sequences of *comparison units* which are the object of the comparison function. We have presented the difference between *free* and *fixed* granularity clones: Whereas free granularity clones will show the duplication situation in (sometimes confusing) detail, fixed granularity clones will reduce the duplication to a fixed set of potential relations between source units. We have also emphasized that clone pairs should not be considered in isolation, but should however be aggregated, not only via the (occasionally) transitive clone relation, but also via the neighborhood of their constituting source fragments.

Of the multitude of clone detection approaches which we classify using the goals of *Scalability*, *Adaptability*, *Duplication Sensitivity*, and *Reengineering Support*, we present a lightweight solution that emphasizes *Scalability* and *Adaptability* as important aspects of a clone detector that belongs in the toolbox of every engineer.

Following the selection of goals, we have chosen to use string matching as the comparison function. To compare source code using string matching is inherently favorable to both goals: A natural format into which source code does not have to be transformed, and a simple comparison function which does not need much overhead. A functioning detector can be set into practice immediately.

One of the big problems for clone detection is that copied source fragments are usually not similar on a textual level: Many smaller or larger differences conceal an underlying similarity between fragments. To level the superficial differences between fragments so that they do not hinder the comparison function in detecting similarity, we employ a number of techniques which bring the source code into a *normal form*: A pretty printer normalizes the layout, and a number of normalization and filtering techniques remove volatile syntax elements from the code. We split the source text into lines and compare them, exploiting the fact that exact

string matching defines a set of equivalence classes to speed up comparison. By integrating gaps at arbitrary locations in a matching sequence of code, we allow for unforeseen variations of the duplicated code which are not caught in the normalization. The gaps and the normalization measures, while one the one hand increasing recall, naturally induce an increase of the number of false positives on the other hand. We counter this effect with a second analysis phase, this time performing more detailed analysis of the clone candidates. We present a number of measures which give weight to a clone candidate based on fine-grained similarity analysis of the gaps, and size, frequency, and distance of the matching code. Combining these measures we form ranking strategies for a number of different reengineering tasks. The parsing that we have to do for both the normalization of the source code as well as the analysis of the clone candidates is kept minimal, such as to enable a change of programming language via configuration.

A minimal parsing approach does not, however, gather information necessary for deep code analysis. To cope with this need and instead of supporting the reengineer with automated proposals for clone refactoring, we have investigated the possibilities for an improved presentation of the clone data. Apart from creating aggregations of clones into clone classes and clone class families, we have proposed enhancements for clone browsers and code editors. This low level presentation is complemented with, on a local level, the well known dotplot visualization, useful for analyzing rich duplication situations in a close neighborhood. On the global level we propose a set of visualizations of quantitative aspects of the detected clones. An engineer thus gets insights into the duplication situation on the system level.

From our validation of some aspects of the proposed techniques we derive the following insights.

String based clone detection is generally characterized as having high recall but low precision, *i.e.*, the engineer is confronted with a large number of clones, many of which are false positives. We have demonstrated that a quite simple ranking measure is able to prioritize the list of clone candidates in accordance to human evaluators, reducing the amount of time that reengineers loose sifting through false positives.

We have also shown that the choice of a single source line as the granularity of the comparison unit is justifiable: Only a very small percentage of clone candidates from a number of example systems could not be detected due to relocated line breaks. We have additionally shown that a configurable pretty printer is able to normalize the layout of source code should it be suspected that many clones were missed due to layout differences.

In summary, we have provided evidence that it is possible to construct efficient clone detectors based on simple, adaptable techniques. We were able to reduce the impact of drawbacks inherent to string based duplication detection, increasing recall on the on hand, but also controlling the loss of precision one the other hand. We have balanced a lack of automation potential with increased support for clone presentation and visualization. This work is an example of the usefulness of program analysis based on incomplete information [Mur96][vDK99]. We have shown that there is a middle ground between shallow and deep code analysis, where information extraction is affordable and its results make duplication detection more efficient.

7.2 Future Work

This section is split in a *specific* part which lists ideas that grew out of the techniques and methods presented in the thesis, and a *general* part which addresses the wider research field.

Thesis Follow Up

Direct extensions of the thesis work are:

A Methodology: This thesis presents a toolbox of techniques for clone detection, but it provides very little guidance in how and when to apply them. One of the most important questions is probably which normalizations should one apply to keep the number of candidates acceptable. Another question is how to deal with large systems. Such a handbook of best practices, along the lines of the reengineering patterns of Demeyer et al. [DDN02], is needed to support practitioners in the field.

Appropriate Clone Granularity: It is a reasonable assumption to make that programmers are copying fragments of source code which represent a conceptual entity, performing a single conceptual task. Such a program *chunk* has for example been defined as *a sequence of software instructions that achieves a coherent purpose and that can be understood outside of the context in which it is used* [BRS+97]. Program *plans*, or *clichés*, and how they can be identified have been investigated for program recognition tools. Duplication detection approaches which use fixed clone granularity do so to reduce the search space by constraining the number of potential clone relations between program entities. Functions are commonly chosen as the fixed granularity level which is reasonable because development heuristics suggest that functions should encapsulate a single conceptual operation. In real world software, however, functions more often than not assemble multiple responsibilities. Similarities between responsibilities buried in overloaded functions may go undetected for a detector operating at the function level. It could therefore be interesting to use the results from program plan identification to define more appropriate potential candidates for clone instances.

Creation of Search Patterns from Source Code: Following an approach used in the domain of DNA sequence analysis by the homology detector BLAST [AGM+90], we can attempt to find all duplicates of a given code fragment by defining a pattern which describes the set of potential clones of the original fragment (like a regular expression describes a set of strings) and start a targeted search. This will be helpful for maintainers who have changed a piece of code and want to know if the change has to be applied somewhere else in the system too.

Exploiting Dotplot Patterns: If we use free clone granularity, relations between copied fragments can be manifold. For example, smaller fragments can be contained within larger fragments or can overlap with other fragments. What could be interesting to investigate is what these structures reveal about the source code. Especially structures that are repeated regularly may offer insight into interesting characteristics of a program.

If we visualize duplicated code with dotplots, we see the relations between copied fragments appear as a multitude of dot configurations in 2D [Hel94][Hel95]. Many of these configurations are unique, but some are found over and over again, so that we can identify them as *repetition patterns*. We currently retrieve only few of these patterns. If we find patterns that represent interesting information we can give the software maintainer a few tools for understanding.

General Ideas

These ideas encompass the wider context of duplication detection:

Causes for Duplication: Understanding which process and organizational properties as well as project history events can influence the creation of clones can help clone detection. The data collected by version control systems, for example, could be used to direct detection endeavors.

High Level Duplication: We need to find ways to detect clones within other representations than source code. If we are able to spot potential duplication on the design level, we could apply preventive measures early on, for example to propose a library function for tasks needed in several subsystems.

Influence of Source Code Properties on Detector Performance: From the experience of looking at various systems and also from literature [Bel02a] we have learned that detection approaches behave differently in terms of recall and precision depending on the system. What we do not know is which properties this behavior is depending on. A better understanding would allow us to tune detectors for specific systems.

Relevance is Key: As we have discussed in Chapter 3 the relevance of a clone is dependent on the tasks of maintainers. To advance research, we need examples of tasks that industrial users need clone detection for, as

well as examples of detected clones that are not suitable for any refactoring measures. Such examples should be collected in a *clone library* which could be part of a benchmark suite.

Consequences for Language Design: Following the dictum of Krueger [Kru92] that to date, higher level programming languages are the most efficient mechanism for reuse, we wonder how much insight into language design can be gained from code duplication research.

Some concepts that are currently developed in object-oriented programming aim to improve the reuse mechanisms for classes to reduce the number of situations where programmers are forced to implement code twice. An example for such a mechanism that reduce duplication are traits [BSD03]. However, as Jarzabek&Shubiao [JS03] have shown, sometimes the duplication is so fine grained that even with the most sophisticated language mechanisms we cannot avoid duplication. Jarzabek&Shubiao therefore explore meta descriptions of source code which is free of duplication. Clone detectors could create the meta descriptions automatically.

Appendix A

Approaches to Code Duplication Detection

This section strives to give an overview of the code duplication detection approaches that are discussed in the scientific literature. Recently, companies have been building tools, *e.g.*, SIMSCAN [1] and SIMIAN [2], for which the details of the approaches are not disclosed. We therefore exclude these from the presentation in this section. The full list of detection approaches presented in this overview is given in Table A.1. The presentation in this table adopts a broad classification according to the general level of detail the pre-comparison code transformation, *e.g.*, *lexical*, *syntactic* or *semantic*. This attribute is a good overall characterization because it influences the code transformation, comparison technique and analysis capacity of the approach. The next subsection gives an overview of the structure of the appendix.

Reference	*Level*	*Code Representation*	*Comparison Technique*
[Joh94a]	Textual	Substrings	String-Matching
[DRD99]	Textual	Normalized Strings	String-Matching
[DBF+95]	Textual	Metric Tuples	Neural Networks
[Bak92]	Lexical	Parameterized Strings	String-Matching
[KKI02]	Lexical	Normalized Strings	String-Matching
[MM01]	Lexical	Word in Context	Latent Semantic Analysis
[Jan88]	Syntactical	Call Graph, Metrics	Hybrid
[MLM96b]	Syntactical	Metric Tuples	Discrete Comparison
[Kon97]	Syntactical	Metric Tuples	Euclidian Distance
[BYM+98]	Syntactical	Abstract Syntax Tree	Tree-Matching
[BMD+99]	Syntactical	Abstract Syntax Tree	Tree-Matching
[CDS04]	Syntactical	Strings	String-Matching
[Kri01]	Semantical	Program Dependence Graph	Graph-Matching
[KH01a]	Semantical	Program Dependence Graph	Backward slicing
[Lei03]	Semantical	Abstract Syntax Tree	Hybrid, Syntax-Driven

Table A.1: List of approaches for detecting duplication of code

[1] Available from `http://www.blue-edge.bg/download.html` [May 15, 2005]
[2] Available from `http://www.redhillconsulting.com.au/products/simian/` [May 15, 2005]

Overview of Clone Detectors Aspects

To structure the overview of the related work we define a number of aspects which characterize the detection approaches. This will allow us to report concisely about the noteworthy characteristics of the individual approaches. Organizing the presentation in this way reduces the amount of repetition compared with a detailed explanation of each individual approach.

We present the following aspects:

Code Features to be Compared: A detection approach extracts attributes from the source code and computes a profile for each comparison unit. Clones are found by comparing these profiles. The selection of the comparable features determines what kind of code transformations and which comparison techniques are needed by the approach.

Comparison Techniques: Apart from the selection of the comparable source code attributes, the way in which these properties are represented and ultimately compared determines the nature and complexity of a detection approach.

Clone Granularity: The choice between fixed or free clone granularity (see Section 3.6.1 on Page 35) determines the amount of data that must be compared and analyzed.

Groups of Clones: The clone pair is the simplest form of clone representation. Grouping clone pairs reduces the amount of data that must be investigated by a reengineer.

Ranking and Filtering: Detectors use different strategies to distinguish relevant clone candidates from false positives.

Reengineering Support: Collecting additional information which is used for the classification of the found clones helps assess the opportunities for clone-based reengineering measures.

There exist, of course, many more properties by which we could categorize the detection approaches, for example scalability, language independence, space and time complexity, precision and recall behavior. We have chosen the set of properties above because they can be gained from the literature for all the approaches and can therefore be compared.

A.1 Source Code Features

As we have stated in Chapter 3 clones must be defined based on the similarity between some objective features of the source text. We can therefore characterize all possible detection approaches by the kind of features that they have chosen to base their comparison on. We can distinguish three different types of source code features that are being used to assess similarity between source fragments:

Textual Features: These features are on the level of the textual representation of the program. They mostly determine the layout of the source code. Basing duplication detection on these more 'ephemeral' features makes sense when we assume that copying and pasting the source code in an editor transfers these features from the old to the new location.

Examples are: *the source text itself, indentation, number of non-blank lines, line length, identifier length, number of comments.*

Structural Features: These features describe structural properties of the code. Most source code features fall into this broad category, which is why we define two sub-categories:

Syntactic Features: Syntactic features are derived from the results of parsing the source code such as an abstract syntax tree. Most of the myriads of source code metrics can be used for comparison purposes.

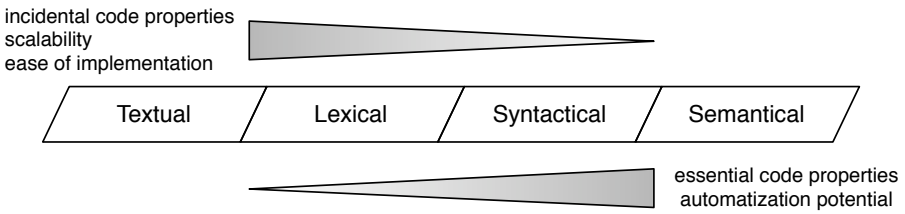


Figure A.1: The range of analysis techniques which extract comparable features from source code. Towards the left, more superficial features need only lightweight analysis techniques which are scalable. Towards the right, essential features require more complex analysis techniques, but provide enough information for improved automation.

Examples are: *number of statements, nesting level, keywords used, operators used, number of parameters passed by reference/by value, number of exit points, number of defined/used/updated variables.*

Semantic Features: Semantic features are derived from detailed code representations or the results of advanced code analysis.

Examples are: *call-graph features, program dependence graph features like value and reference dependencies, features of program slices, pre- and post-conditions.*

Symbolic Features: These features come from the elements of a program that stem from natural language. They are the terms that make the program readable for the programmer.

Examples are: *identifier-, function-, and type-names, comments.*

The properties used for comparison can be positioned along an axis from shallow to deep code analysis that is needed to extract the properties from the source text (see Figure A.1). The passage from text- to semantics-based analysis is a progressive abstraction of incidental properties of source code towards its essential *meaning*. This brings more opportunities for the comparison and filtering phases, but the cost lies in increased dependence on difficult-to-adapt parsing technology and increased computational complexity which reduces scalability.

How the selection of the features to be compared influences each step of the detection is described in the following paragraphs.

Transformation Step: The type of transformation is the basis for the entire approach and from this decision almost everything else is determined. The transformation itself is of linear time complexity (input program size) and thus the cheapest of the entire process. Space requirements however depend much on the format. An abstract syntax tree needs about ten nodes for each line of code [BYM+98]. If control- and data-flow edges are added to the graph space requirements increase further.

Comparison Step: The more a comparison method is aware of semantic properties of the entities it compares, the better it can uncover similarities among ever more varied and changed source fragments. For example, if a comparison algorithm knows that + is a commutative operator, it will find that `a+b` and `b+a` are the same. It is at this stage where deep knowledge of the source code has its greatest impact, but it is also this stage—which is characterized by an $O(n^2)$ time complexity—where complicated approaches pay most of their price.

Ranking and Filtering Step: Ranking tries to determine which detected clone pairs are relevant. Not all relevance criteria can be made into an algorithmic filter, however. Detailed knowledge about the copied source code and how it relates to its context increases chances to express such a criterion in a form that can be used in an automatic filter.

Aggregation Step: At this stage, clones having in common some properties are aggregated into groups. A useful criterion is to collect all clones which copy the same original source fragment. Deep code analysis additionally enables categories which do not rely on the transitivity of the clone relation. These advanced properties can be applicability of a specific kind of refactoring, or ease and impact of the refactorings.

A.2 Comparison Techniques

The detection approaches are using a wide range of comparison techniques. We can categorize the most common ones in these categories:

String Matching: Matching strings is done using the basic `strcmp` function or the UNIX DIFF tool [CDS04], as well as suffix trees [Bak92][KKI02], and dynamic programming (edit distance) [BMLK99].

Tree- and Graph-Matching: If the source code is represented as an abstract syntax tree [BYM+98] or a more elaborated program dependence graph [Kri01][KH01b], the matching function seeks similar subgraphs. This can be done by looking for isomorphic program slices.

Vector Comparison: Tuples of attribute measures are compared using distance metrics like the Euclidian distance for vectors. Machine learning with neural networks has also been used to classify similar tuples together [DBF+95].

A special comparison technique is used by Leitão [Lei03]: his comparison follows the call graph, selecting comparators according to the syntactic elements that he finds on the way. This approach could be described as some sort of parallel execution of two source fragments. Earlier, Jankowitz in a plagiarism detector [Jan88] used the call graph as well to map procedures of two compared programs onto each other. The procedures are then compared via a number of code metrics.

A.3 Clone Granularity

The approaches are equally distributed over the two possible choices of clone granularity:

Granularity	*References*
Fixed	[Jan88][DBF+95][MLM96b][Kon97][BMD+99][MM01][Lei03] [CDS04]
Free	[Bak92][Joh94a][BYM+98][DRD99][Kri01][KH01a][KKI02]

We can relate the clone granularity to the comparison method by counting the number of approaches that employ a given method:

Granularity	*String*	*Tree&Graph*	*Metrics Vector*	*Other*
Fixed	1	0	4	3
Free	4	3	0	0

String- and tree/graph-based comparison are methods which lend themselves for free clone granularities. Metrics-based approaches, however, need a fixed fragment for which they compute a matrics vector which is then compared.

A.4 Grouping of Clones

Clone groups reduce the amount of data that must be investigated by the reengineer. Clones are grouped using either the clone relation or the closeness of neighboring fragments in the source code.

Grouping by Clone Relation: A clone relation is usually reflexive and symmetric, only few are also transitive. Kamiya et al. [KKI02], who use exact string matching for comparison, are able to define clone equivalence classes from their transitive relation.

If the clone relation does not have the transitivity property, clone groups can not be equivalence classes. Baxter et al. build groups by a *first-fit* strategy: a clone is put into the first group in which it is a copy of all instances already in the group [BYM+98]. Giesecke defines sets of source fragments $M(x)$ that consist of all fragments that are clones of the given fragment x [Gie03]. These sets are not disjoint in general. Davey et al. [DBF+95] use a more refined representation of such sets: they cluster feature vectors by their Euclidian distance and represent the similar vectors in a dendrogram, similar to a phylogenetic tree. Such a tree does not only contain all related fragments but also shows which fragments are closer and which are farther apart.

Grouping by Neighborhood: Johnson uses the file as the focal point of duplication investigation. For a set of files he groups all strings duplicated in each of the files. The size of this shared code is set in relation with the size of shared code between other file sets and the size of the files. This enables reasoning about the similarity of files [Joh94b].[3]

Kapser&Godfrey identify *regions* in the source code on a granularity level below 'file': type definitions, prototypes, and variables; individual macros, structs, unions, enumerations, and functions [KG04]. All clones in the same region form a group that can be understood as a whole, taking into account the properties of the region.

A.5 Ranking and Filtering

A detection usually retrieves many clones which turn out to be false positives. Detection approaches try to reduce these numbers by different means.

- String-based comparisons which normalize the names of parameters evaluate the mapping between the parameter names in the two fragments. An inconsistent mapping might indicate a false positives [Bak93b].

  The token-based string matching of Kamiya et al. reduces the number of likely less useful clones by allowing clones only to begin with a token out of a set of *leading tokens*, *e.g.*, keywords which initiate statements, block delimiters, and tokens following statement delimiters [KKI02].

- Metrics-based approaches compute a set of metrics which do not correlate to make the comparison dependent on as much information as possible [Kon97]. Smaller instances of the chosen source unit type, *e.g.*, functions of only a few lines, do not, however, distinguish themselves very well by their metric values and will produce many false positives [DMLP98].

- Kapser&Godfrey use the characteristics of the region in which the clones are found to determine their likely importance. Clones between structure definitions, for example, often turn out to be false positives [KG04].

A.6 Reengineering Support

Clone management only starts with the detection of the clones. Further analysis of the found clones is performed to gather information for the reengineering actions which will ameliorate the duplication situation.

- Computation of the amount of code that could theoretically be removed if all duplication was eliminated [Bak95a][KKI02].

[3] Yamamoto et al. [YMKI02] condense this approach into the definition of a metric S_{line} to measure the distance between entire software systems, *e.g.*, different versions of the BSD UNIX operating system.

- Documentation of the presence of all clones [Bak95a].
- Analysis of the differences between clone instances to find clone categorizations which contain useful information for the reengineer [BMLK99][KN01]. This includes the analysis of the dependencies between the cloned fragments and their context to gauge the ease with which such fragments could for example be extracted and replaced by a function call [BMLK99].
- Creation of macros abstracting the duplicated code [BYM+98].
- Reengineering an object oriented systems using design patterns like *Template method* and *Strategy* [BMD+00].
- Extracting copied fragments into procedures in the presence of reordered as well as inserted and deleted statements [KH02].

A.7 Related Field: Compiler Optimization

For optimization purposes, compiler research has been trying to solve a problem similar to duplication detection since the 1970s. Compiler optimizers want to know which variables or expressions have the same values during program execution [AWZ88][RKS99]. This knowledge is used for various code motion activities which reduce the number of operations that are executed [MR79][Sim96]. The analysis must be provably correct and is thus of semantic nature. The most important differences to code duplication detection can be stated as follows:

- The detection and subsequent transformation *must* be fully automatic. A "human in the loop" is not feasible for efficiency reasons.
- Only transformations which provably do not change the *observational equivalence* of the code are applied.
- The algorithms work on code generated by a compiler front end which is very restricted (three address code) and does not, for example, contain procedure calls. The formal properties of this kind of code make semantic analysis feasible.
- The optimization pass must be efficient in order to not slow down the compilation. Due to the complexity of the analysis, it is not feasible to extend it beyond the limits of a single procedure.

Whereas most of the work on compiler optimization is too much restricted and expensive to be applied to code duplication detection, some of the ideas have been fueling research on duplication detection. The techniques of Komondoor&Horwitz [KH01b] and Krinke [Kri01] both use variants of the program representation graphs that are proposed for program analysis in compiler optimization. Krinke however explicitely refrains from complete equivalence, a strong prerequisite for automatic code transformations.

Horwitz et al. also use these techniques to automatically integrate different versions of the same program [HPR89].

A.8 Related Field: Plagiarism Detection

Plagiarism detection is a field that is important mostly in the area of education where an accurate grading of assignments is only possible if one is sure that the solution is original. Educational institutions have an interest in only handing out good marks to students who have earned them rightfully.

The research on plagiarism actually predates the source code duplication research by a number of years (the earliest papers we are aware of have been published in the 1970s). The reason for this might be that plagiarism is a problem that educators have faced for centuries. Code duplication has only surfaced as an important problem when the life cycles of the first very large software systems that were created under industrial conditions entered the maintenance phase.

Plagiarism is an increasingly real problem in all educational institutions and particularly in programming courses since it is very easy to copy the solution of a programming assignment and alter a few details to make it look like original work without having any understanding of the problem or the solution. Plagiarism is rampant not only in programming courses but also in classes where written essays have to be composed. The world wide web being the largest collection of publicly accessible information lures many a student into plagiarizing documents found on the net [WW02]. Much work to uncover plagiarism therefore is focusing on natural language texts. Natural language differs from programming languages by being much more complex. Whereas we have complete grammars for our programming languages such a thing does not exist for common English. Whereas the vocabulary of a program consist in at most 50 keywords (up to 300 for COBOL) and quite a constrained set of user defined identifiers which all have clearly defined, comparatively simple semantics, the vocabulary of common newspaper articles alone is about 3000 words which have immensely more difficult and ambiguous semantics. In summary, plagiarism detection for natural language texts is more difficult and uses a host of techniques which are different from software plagiarism detectors. In our short overview we will therefore concentrate on the work being done on uncovering plagiarism in source code.[4]

Differences in Detecting Duplication and Plagiarism

Since plagiarizing essentially means copying, one would expect a close relation between plagiarism and duplication detectors. There exist however marked differences between the two problems. To describe this relationship we will first list the common characteristics, then some essential differences which influence the matching strategies, and finally some practical differences where the requirements for clone detectors can be relaxed for plagiarism detection. We finally look at the kind of code transformations a plagiarist employs to hide his activity from the eyes of a detector.

Common Characteristics: In duplication as well as in plagiarism detection the "human in the loop" is of similar importance. Plagiarism is a serious verdict which eventually leads to dire consequences for the culprit. One has to be convinced beyond reasonable doubt about the reasons for perceived plagiarism before raising the flag. The relevance of similarities can also depend on the context of the exercise, *i.e.*, if the assignment called for the solution to a narrowly defined problem, the solutions by different people can easily become very similar without being the result of a forbidden collaboration. The confirmation of a perpetration is thus tricky and cannot be automated. Plagiarism detectors can only provide a preliminary selection of likely candidates which constrains the number of cases that has to be vetted by the instructor.

Essential Differences: These differences between the two fields have an influence on the design of the comparison algorithms:

No Self-Similarity: Plagiarism is strictly a phenomenon *across* programs. Since nobody plagiarizes himself, duplicated code within the same program is never plagiarism. The detectors can thus skip intra-file comparisons completely. Each fragment of the original then has at most one corresponding fragment

[4] For an overview over plagiarism detection in natural languages see Clough [Clo00].

in the plagiary. The consequence is that the detection process for plagiarism is computationally less expensive since for every match between original and plagiary the matching fragments can be removed from the pattern and the text. When matching progresses the search space is thus more and more pruned. The algorithm of Wise [Wis96] exploits this condition explicitly.

Whole Program Structure: Early plagiarism detectors were considering the structure of the entire program instead of only looking at a set of unordered functions. Jankowitz [Jan88] and Cunningham&Mikoyan [CM93], for example, use the call graph as a first similarity criterion, and deepen the analysis by an investigation of individual functions. Code duplication on the other hand is normally not interested in code structure beyond the borders of a function. Later plagiarism detectors have relaxed this requirement as partial plagiarism has entered into focus.

Semantic Equality: Plagiarized code must behave exactly as the original. In clone detection, on the other hand, we are also interested in structural similarities of semantically diverse fragments. Leitão mentions an instructive example to illustrate this difference [Lei03]: Two functions, one to sum up all integers in a list and the other to concatenate all sublists of a list, cannot be result of plagiarism but can very well be considered duplication and refactored using the `fold`-operator in LISP.

Quality of Matches: Since the final verdict must result from a manual comparison of two candidates, match results with low confidence are acceptable. The cost of a false positive is much lower than that of a false negative for a small population such as the exercises handed in by a computer science class (see for example Prechelt et al. [PMP00] who measure the performance of their tool with the formula $\mathit{Precision} + 3 \cdot \mathit{Recall}$).

Practical Differences: A few differences have to do with factors which are important for clone detectors but which plagiarism detectors do not have to take into account.

Scalability: Checking for plagiarism among the weekly assignments of a computer science class will burden a system with a few dozen programs of small to moderate size. Never will the detector have to tackle millions of lines of code.

For detectors that must verify the originality of natural language essays written for school or university assignments, the scalability issue is however a relevant one. The text database that must be searched for possibly plagiarized documents is either the world wide web or a large collection of documents covering all possible topics.

Language Variety: Rarely will the plagiarism detector be confronted with systems written in different languages. Also, the range of languages used for education is quite narrow and constrained to mainstream languages. An institution will not change their course language frequently.

Refactoring Support: Found plagiarism will not have to be remedied. Additional information to support the refactoring process which duplication detectors collect is not needed.

Attacks on Plagiarism Detectors: Another difference between plagiarism- and duplication detectors is that the former ones are participating in an arms race. The foremost goal of every plagiarist is always to hide the origin of his work, *i.e.*, to confuse the an eventual detector, be it human or automated. This leads to a number of code transformations which are *attacking* the correct functioning of the detector. To be winning the arms race the detector must continuously analyze which attacks are effective against it and develop new defenses. There are no fundamental differences between transformations applied by plagiarists and by code duplicators. All transformations (see Appendix B) that are applicable in one case could also be found in the other case, the only difference being the intention behind them. The plagiarist is however constrained by the following conditions:

- The strongest of all constraints is that the overall semantics of the code are not changed.[5] Even if additional statements are inserted they will be redundant with respect to the already existing code.

[5] Normally, that is. Prechelt et al. [PMP00] report about plagiarists who inserted program errors on purpose.

- Changes should achieve maximal efficiency against a (human or automatic) detector. This implies (real or imagined) knowledge on what kind of changes are able to fool the detector.
- Changes should be minimal. The effort for plagiarizing must be smaller than the effort for doing the work from scratch. The smaller the assignment, the less effort economically still makes sense.
- Changes must be applied to all of the code to avoid that the detector picks up partial plagiarism and the summoned human investigator is able to complete the analysis.

Against an advanced detection algorithm like the one of Wise [Wis96], only a small number of disguising techniques are effective. All of them change the sequence of statements in order to confuse the detector. Examples are:

- repetition of side-effect free assignments.
- redundant **`if`** statements with identical `then` and **`else`** clauses.
- frequent invocations of dummy functions.

The requirement to change code everywhere and the nonsensical nature of the inserted statements make the disguised code immediately suspicious to a human inspector. It might even be the case that certain disguising strategies create patterns that can be picked up by other program analysis tools, exposing a plagiarist through exactly the means behind which he wanted to hide.

In summary: duplication detectors could be employed as plagiarism detectors. Plagiarism detectors, however, which exploit the one-to-one correspondence between pattern and text, are not useful for duplication detection.

Appendix B

Post-Copy Editing of Code

One of the central problems of code duplication detection is that instances of copied code are not identical. A common duplication scenario is that a programmer working on implementing some functionality remembers the place of a fragment of code which serves this same purpose already. The size of the fragment can range from single expressions to whole subsystems consisting in multiple source files. The programmer copies the fragment and inserts it at the new place. In many cases the programmer then has to adapt the copy to the new location. These immediate changes are, in the longer run, followed by maintenance cycles that can have diverse effects on the two fragments. All of these adaptions are of varying magnitudes. They can roughly be characterized as either *essential* or *superficial* according to if they affect the purpose of the code or not. The changes progressively disconnect the original from the copy, essential changes putting more distance between them than superficial ones.

The challenge for a detection mechanism is to see through the changes, to bridge the distance between the fragments and to recognize the original clone relation. For fixed granularity clone detectors this simply means that a threshold must be established which draws the line between what is still considered a clone and what a false positive. A detector of free granularity clones, which is the kind we are going to discuss in this chapter, has two ways to handle changes:

- Normalize certain elements of the source code and compare the deeper code structure that is revealed. The more *essential* a change, the greater the normalization that is necessary to bridge it. Greater normalization always means deeper program analysis, starting from lexical, then syntactic, and finally semantical analysis.

  Only smaller changes which do not destroy the structure of the clone can be handled in this manner.

- Aggregate separate adjacent sub-clones that result from larger changes in the middle of a copied fragment. Sub-clone aggregation is an additional analysis step after the initial clone candidates have been assembled from the atomic matches.

  This type of handling changes between copied source fragments is necessary if the changes are disruptive, *e.g.*, insertion, deletion or moves of code, which cannot be normalized.

The clone relation can be broken for each pair of copied fragments if the accumulation of changes becomes too large. The breaking point is reached when the two fragments no longer fulfill the same purpose. The ideal detector would recognize each clone pair exactly up to this “breaking point”. Since the purpose of a fragment is difficult to capture current clone detectors will miss clones which would still be acknowledged by human experts.

Structure of the Appendix

This appendix presents a list of changes which programmers typically apply to copied code. We will list edit operations in increasing severity, from superficial to essential to disruptive. We can not propose a more relevant order without having data about the frequency of each of the operations. For each transformation we will discuss the influences which are conducive to these changes, the evolution of the code in the course of its life cycle not being mentioned especially. We then show small examples, and finally investigate the type of change from the point of view of how it can be dealt with by a clone detector, putting emphasis more on normalization and less on sub-clone aggregation. For the sake of simplicity of the discussion we will assume that only one transformation has been applied to the code. Nothing will be said on how the techniques react when multiple transformations have been applied at once.

For changes which can be dealt with by code normalizations we are especially interested in the question of how much program analysis is needed to perform the normalization, or concretely: Can we build a configurable lexical analyzer or parser that is able to normalize this kind of changes?

B.1 Superficial Edit Operations

Edit operations which change the code superficially do not alter the purpose of copied fragment at all. A clone detector can handle these kinds of changes exclusively by using normalizations.

Editing White Space

White space normally does not carry any other semantics than acting as a delimiter between tokens (languages like PYTHON are exceptional as they use tabulators as block and scope delimiters). Although of no great importance for the purpose of a source fragment, white space determines the human readability of the source code.

Editing Incentives: The code layout is a usually subject to personal preferences of a programmer and is only likely to be changed if another programmer adjusts a copied fragment to his or her own liking (in the absence of a coding style guide that must be adhered to).

Example: The only type of white space changes that affects clone detectors is the *line break relocation* which can throw detectors off course that are line-oriented. The following examples differ only in line-breaks:

```
for (i=0;i<TABSIZE;i++)
  {
    bp = symtab[i];
    while (bp)
      {
        bptmp = bp->link;
        XFREE (bp);
        bp = bptmp;
      }
  }
```

```
for (i = 0;
     i < TABSIZE;
     i++)
 {bp = symtab[i];
  while(bp) {
   bptmp = bp->link;
   XFREE(bp); bp=bptmp;}}
```

Coding styles which are followed by development teams usually are only guidelines and can be easily broken by a programmer. Line breaks can also be forced incidentally, *e.g.*, when long identifier names push the line length over a given character threshold: The following example demonstrates this:

```
for (i=0; i<MAX; i++) memset(d[i], '\0', MAX);
```

```
for (counter=0; counter<MAXNUM_PAT; counter++)
  memset(tc_aduplicates[counter], '\0', MAXNUM_PAT);
```

Normalization Measures: For detectors which are sensitive to line-break relocation, pretty printing is a way to normalize the source code and remove all the effects of these kind of changes. We can build a pretty printer which is configurable for a number of languages (see Section 4.2.2 on Page 48).

Editing Comments

Comments are not executed at runtime of the program and therefore have only a marginal importance (and, unfortunately, are often treated that way by programmers). If they are present and well written, however, they should explain the purpose of the code on a more abstract level than the source code. This ideally would make them robust against changes in a new context and they would therefore be good indicators of copy and paste. In practice, however, programmers often do not add comments to their code at all, or rarely at the right abstraction level.

Comments are usually disregarded by clone detectors. We are only aware of the approach of Maletic&Marcus [MM00] that uses the information in comments to detect code fragments with similar functionality.

Editing Incentives: Inadequate commentary is likely to be removed or improved by a programmer. During an attempt to understand the intricacies of a foreign fragment, programmers might annotate pieces of the code.

Normalization Measures: Commentary is removed previous to comparison by almost all detection approaches. This filtering can be implemented in a lexical analyzer which can be configured with the comment delimiter signs of the programming language in question.

Editing Redundant Syntax Elements

We call syntax elements *redundant* if they only have little semantic content, *i.e.*, if their absence does not affect the behavior of the code. Changing them will therefore not alter the purpose of a fragment much or not at all.

Examples of tokens which do not change the semantic at all are:

- Block delimiters for blocks consisting of only a single statement.
- Statement delimiters for the last statement in a block.

Example of syntactic elements which control the semantics of a piece of code slightly (from the local perspective of a copied fragment) are:

- Type modifiers in variable and parameter declarations, *e.g.*, **`const`**, **`struct`**, **`static`**, and **`extern`**.
- Access specifiers in C++ or JAVA: **`public`**, **`protected`**, and **`private`**.
- Namespace indicators.

- Labels as targets of **goto** statements.
- Type casts. Strictly speaking a type cast may alter the behavior of the code due to late-binding in object-oriented code or the invocation of complicated conversion functions.

Editing Incentives: Changing block and statement delimiters is dependent on coding style. Type modifiers are dependent on the context of the code. Labels depend on control flow.

Example: This example shows two completely equal fragments of code where block delimiters for the `then` and **else** blocks have been omitted on one side:

```
if (t == '\n')
  complain("Skip to \n");
else
  complain("Skip to %c",t);
```

```
if (t == '\n') {
  complain("Skip to \n");
} else {
  complain("Skip to %c",t);
}
```

The full package specification of a JAVA class is optional if the package as been properly imported and no name clashes occur.

```
import java.lang.reflect.*;
Method m = myClass.getMethod(mname,null);
```

```
java.lang.reflect.Method m = myClass.getMethod(mname,null);
```

Normalization Measures: These syntax elements are all tokens or keywords which can be found and removed with regular expressions searches. Configuration is done by listing all necessary tokens and keywords of a programming language.

Editing Parameters

A parameter is either a literal constant (number or string) or a variable name. These are probably the most frequently changed elements in a copied fragment.

Editing Incentives: The names of variables and the use of constants are determined by the context into which the copied fragment is inserted. Only fragments that are large enough so that they carry their own context with them will preserve their identifier names.

Example: This example from the AGREP system shows *systematic replacement of identifiers*. The uses of identifiers `r2` and `r3` are switched but the two fragments behave exactly the same.

```
if(CMask != 0) {
  r1 = Init1 & r3;
  r2 = ((Next[r3>>hh] |
         Next1[r3&LL]) &
        CMask) | r1;
} else  {
  r2 = r3 & Init1;
}
```

```
if(CMask != 0) {
  r1 = Init1 & r2;
  r3 = ((Next[r2>>hh] |
         Next1[r2&LL]) &
         CMask) | r1;
} else {
  r3 = r2 & Init1;
}
```

Normalization Measures: Names of identifiers as well as literal strings or numbers can be detected with regular expressions (see Appendix C). Normalizing all variables to the same token may produce clones which cannot immediately be refactored due to differences in variables usage. The *systematic* replacement of identifiers can be made an integrated criterion of the comparison [Bak95b] or can be made a filter for the clone filtering phase.

Editing Value Access

By *value access* we mean the way a memory location is accessed. This can be done either by using a direct name, or by dereferencing aliases. *Access chains* are formed when multiple interconnected records are dereferenced. Changing the value access in a copied fragment is behavior preserving for the fragment.

Editing Incentives: Like the change of variable names, value access is also dependent on the context of the copied source fragment.

Example: This clone from the APACHE system shows how the `config` data is accessed over three stations in the first case and over two stations in the second case.

```
void *sconf = r->server->module_config;
alias_server_conf *serverconf =
  (alias_server_conf*) ap_get_module_config(sconf,&alias_module);
```

```
void *dconf = r->per_dir_config;
alias_dir_conf *dirconf =
  (alias_dir_conf *) ap_get_module_config(dconf, &alias_module);
```

Normalization Measures: All but the last name in an access chain can be removed with regular expressions.

Editing Types

If only the type of a variable is changed but the purpose of the code is preserved, *i.e.*, the operations that are applied to the variables remain the same, the new type can be considered similar to the original type. The change is therefore only superficial.

Editing Incentives: Choosing a type for a variable is dependent on the context in which the code is executed.

Normalization Measures: Types only occur in variable or parameter declarations. Since declarations are essentially a help for static error checking and allocation optimization by a compiler, they have less semantic weight than processing logic. If we take this stance we could remove types completely from the source code, provided our lexical analysis is able to distinguish between variable and type names. Less radically, we can normalize all type names to the same token just like we do with parameter names. We can also be a bit more discriminating and form equivalence classes, *e.g.*, normalize all builtin numeric types to the same token.

B.2 Essential Edit Operations

Essential edits change the purpose of the code in smaller or larger ways, *i.e.*, the copied and edited fragment behaves differently from the original. These changes can still be handled through code normalizations, however the amount of parsing information is significantly greater.

Editing Expressions

An expression is the smallest program fragment which carries logic, fulfills a purpose and can therefore be the target of a copy operation. Any larger fragment of code contains multiple expressions.

Expressions can be changed in three ways which we are going to look at in more detail: *i)* replacement of a parameter by the result of a function invocation, *ii)* change of operators, or operand order, and *iii)* change of invoked functions.

Editing Expression Structure

An expression can consist in a simple term like a precomputed value represented as a literal constant or it can be a complex term, *e.g.*, the invocation of a function which computes the value at runtime. Editing expressions will most likely change the semantics of the source code.

Editing Incentives: If we keep in mind that a source fragment is copied because it contains a certain amount of logic ("fulfills a purpose") then we can say that an expression is more likely to be edited if the change does not affect the purpose of the copied fragment. A simple expression, *e.g.*, a variable reference, in a fragment of a dozen lines is therefore much more likely to be replaced than a complicated term in a two-line fragment.

If a fragment of code is copied with the intention to variate its purpose in some way, any expression is likely to be changed to accommodate the variation. It is however impossible to find out if a given expression is part of a large or small copied fragment, or if it is not duplicated at all.

Example: This example stems from the APACHE system. It shows how a constant variable is changed into a function invocation:

```
client_mm = mm_create(SHMEM_SIZE, tmpnam(NULL));
```

```
opaque_mm = mm_create(sizeof(*opaque_cntr), tmpnam(NULL));
```

This example from the APACHE system shows a changed condition:

```
if(apisdigit(PEEK())) {
  count = p_count(p);
  REQUIRE(c<=count,REG_BADBR)
}
```

```
if(MORE() && apisdigit(PEEK())){
  count = p_count(p);
  REQUIRE(c<=count,REG_BADBR)
}
```

Normalization Measures: Since we have no way of telling which expression the programmer is going to change, it is difficult to normalize any of them. One could only define a certain threshold of expression complexity and replace any expression below this threshold with the simplest expression: a variable name or a literal constant.

Editing Operators

Changed operators affect the behavior of the code. If only one operator is changed, the variant computation may still fulfill the same purpose.

Example: Similar computations with different operators occur for example in situations where symmetry plays a role. Such code, albeit functionally different, is conceptually the same:

```
lowerBound = pos - range/2;
```

```
upperBound = pos + range/2;
```

Normalization Measures: We can normalize operators by using single representatives for operator equivalence classes, *e.g.*, from the following set of C operators:

Equivalence Class	*Equivalent Operators*
+	+, -, *, /, %, ^, <<
++	++, --
+=	+=, -=, *=, /=

Editing Operand Order

The order of operands for commutative operators like + can be re-arranged without affecting the result of the operation.

Editing Incentives: Changing operand order is probably only the result of a "mental macro" implementation [BYM+98], a memorized piece of logic where irrelevant details such as operand order are likely to be forgotten and randomly changed.

Example: This conditional was found in the COOK system:

```
if (sp == last_shift)
  last_shift = sp2;
```

```
if (last_shift == sp)
  last_shift = sp2;
```

The following fragments stem from two files in the AGREP system:

```
alloc_buf(text, &buffer,
  BlockSize+Max_record+1);
```

```
alloc_buf(text, &buffer,
  Max_record+BlockSize+1);
```

Normalization Measures: In order to normalize operand order one has to know which operators allow commutation. Normalization here requires therefore deep code analysis and detailed knowledge of the operators. If normalization is to be done before the comparison, one would have to define a universal order for operands to map all similar terms to the same representation. Lakhotia&Mohammed [LM04] have defined normal forms for expressions. If the normalization is done during comparison we can enumerate all possible permutations and compare each combination, a potentially time consuming tasks which needs some heuristic constraints (see Leitão [Lei03] for a discussion).

Editing Function Invocations

A function invocation is the best example of code duplication that is avoided. A copied fragment consisting only in a function invocation is therefore not considered duplicated code. Function calls copied along with

larger fragment are however subject to edits just as any other part of copied code. By changing an invoked function the semantics to the source code are very probably changed.

Editing Incentives: Since functions usually represent a fair amount of programming logic, changing them is likely to alter the purpose of the copied fragment. This would support the opinion that editing functions calls does not happen often and it is more likely that the parameters of the function invocations are edited since they are semantically more lightweight and less prone to change the purpose of the code. If there exists a function `foo_b` implementing a variation of `foo`, replacing `foo` by `foo_b` will only marginally alter the purpose of the overall code, making such an edit more likely.

Example: The following example was found in the AGREP system. Three other copies of this fragment, each with a different function but the same parameter list, can be found at the same location.

```
  if (-1 == monkey(pat, m, text+start, text+end, oldpat, oldm)) {
    free_buf(fd, text);
    memcpy(text+end+1, tempbuf, m); /* restore */
```

```
  if (-1 == bm(pat, m, text+start, text+end, oldpat, oldm)) {
    free_buf(fd, text);
    memcpy(text+end+1, tempbuf, m); /* restore */
```

The following example was found in the AGREP system as well. Almost every line contains not only different variables but also different function calls. The `tc_` prefix by which the function names in the second fragment differ from the names of the first fragment indicates that the functions are probably only marginally different from each other so that the purpose of the code is the same in both fragments.

```
  for(i=1; i<=num_pat; i++) f_prep(i, patt[i]);
  accumulate();
  memset(pat_indices, '\0', sizeof(int) * (num_pat + 1));
  for(i=1; i<=num_pat; i++) f_prep1(i, patt[i]);
```

```
  for(i=1; i<=tc_num_pat; i++) tc_f_prep(i, tc_patt[i]);
  tc_accumulate();
  memset(tc_pat_indices, '\0', sizeof(int) * (tc_num_pat + 1));
  for(i=1; i<=tc_num_pat; i++) tc_f_prep1(i, tc_patt[i]);
```

This example comes from the AGREP system as well. `FirstPos()` and `Lastpos()` are macros and will be interpreted as functions only by detection approaches which do not parse the code.

```
  pos = Lastpos(e);
  while (pos != NULL) {
       tpos = pos;
       pos = pos->nextpos;
       free(tpos);
  }
Lastpos(e) = NULL;
```

```
  pos = Firstpos(e);
  while (pos != NULL) {
       tpos = pos;
       pos = pos->nextpos;
       free(tpos);
  }
Firstpos(e) = NULL;
```

The plagiarism detector of Wise maps variant function names like `strncmp` to the common form `strcmp` [Wis96].

Normalization Measures: The names of functions can be normalized with lexical means just as parameter names. Knowledge about conceptual similarity of some functions could be used to form equivalence classes and normalize more specifically.

B.3 Disruptive Edit Operations

Disruptive changes break the copied fragment apart by inserting or deleting statements, or moving code around. These severe changes can be covered up only by code normalizations based on deep source analysis methods, if at all. If the deep analysis fails or is deemed to costly, the change must be dealt with by assembling partial clones.

Insertion or Deletion of Statements

Inserting or deleting statements can change the behavior of a fragment of source code in a drastic manner and render the clone relation between copies not only unrecognizable, but also make it very difficult to reconcile the two fragments in an eventual refactoring operation.

Editing Incentives: Inserting or deleting statements is most likely done during the continued evolution of the system when new requirements make the overhaul of some parts of the code necessary.

Since statement deletions and insertions have the potential to heavily alter the purpose of the code they will be applied only to copies where the change is small relative to the entire fragment.

Example: These fragments stem from the regular expression parser of the APACHE system, they interpret the same command in two different contexts. The fragments are found in two functions about 600 lines apart.

```
INSERT(OCH_, pos);

ASTERN(OOR1, pos);
AHEAD(pos);
EMIT(OOR2, 0);
AHEAD(THERE());
ASTERN(O_CH, THERETHERE());
```

```
INSERT(OCH_, start);
repeat(p, start+1, 1, to);
ASTERN(OOR1, start);
AHEAD(start);
EMIT(OOR2, 0);
AHEAD(THERE());
ASTERN(O_CH, THERETHERE());
```

Normalization Measures: Davey et al. mention that by reducing source lines to their indentation level only, the resulting representation is relatively robust against the insertion or deletion of a line in the middle of a block [DBF+95].

In general, however, insertion or deletion of statements cannot be normalized since such changes may affect the code on every structural level. Control flow and even data flow dependencies can be disrupted by the insertion of additional code which means that there is no structural level any more which is not affected by such a change, *i.e.*, where similarity between the fragments could still be recognized.

However, if statement insertions alter the code at such a fundamental level as the data-flow, chances are that the clone relation between the fragments will be broken and consequentially detectors should not retrieve them any more.

Editing Control Flow

Editing control flow means to add new (or delete existing) potential execution paths through the code. Control flow is an essential structure in the code which makes changes to it hard to cope with if we use only normalizations.

Editing Incentives: This kind of change is usually the result of broadened requirements. A new use case for which the control flow of the copied fragment must be extended is either the result of a different context, into which the fragment is being inserted, or is introduced during the evolution of the system.

Example: The following examples show how conditionals change the control flow of an original fragment:

```
foo();
bar();
```

```
foo();
if(cond) bar();
```

```
foo();
if(cond)
  baz();
else
  bar();
```

Normalization Measures: When only control flow is changed it is possible that data flow dependencies are not affected. A data flow representation is then able to capture the commonality which still exists between the two fragments.

Reordering Statements

When altering the sequential order of statements their set remains the same, in contrast to statement insertion or deletion. By moving the existing statements around the behavior of the code may be or may not be affected. Changing the order of the cases in a selection or **`switch`**-statement, for example, has a good chance of preserving the behavior.

Editing Incentives: A behaviorally neutral statement reordering is rare, since programmers will not bother to edit code without a concrete difference. Reorderings are likely to occur when programmers implement *mental macros* where they might accidentally interchange statements which are independent of each other. Statement reorderings that alter the code's behavior are also possible whenever a slight variation of the computation is necessary.

Example: This PYTHON example stems from the ZOPE system. The array access `l = sktl[k][:]` has moved into the **`if`** statement on the right side.

```
  otest = oktl.has_key
  for k in sktl.keys():
    l = sktl[k][:]
    if otest(k):
      ol = oktl[k]
      for x in ol:
```

```
  otest = oktl.has_key
  for k in sktl.keys():
    if otest(k):
      l = sktl[k]
      ol = oktl[k]
      for x in l:
```

The following example has been taken from [KH02]. The move of the statement '`base = BasePay[emp];`' is behaviorally neutral:

```
overPay = 0;
if(hours > 40) {
  oRate = OvRate[emp];
  excess = hours - 40;
  overPay = excess*oRate;
}
base = BasePay[emp];
Pay[emp] = base+overPay;
```

```
base = BasePay[emp];
overPay = 0;
if(hours > 40) {
  excess = hours - 40;
  oRate = OvRate[emp];
  overPay = excess*oRate;
}
Pay[emp] = base+overPay;
```

Normalization Measures: If control- or data-flow are unaffected by the reordering a control- or data-flow representation of the code will normalize the changes.

Appendix C

Implementation of Normalization

For the purpose of hiding superficial differences of copied source code from the comparison we need to identify certain elements of a low level source model, like comments and literal strings, or names of variables and functions. These elements will be either rewritten (normalized) or removed (filtered). This section discusses the strategy we use to recognize the elements of the source model and shows some example implementations for recognizers.

For our purposes we need to understand the syntax of the source code on a coarse level only. We mostly have to identify high level structures which are delimited by a set of (usually) unambiguous tokens. The text between the delimiters is mostly uninteresting for our purposes and can often be ignored completely (in the manner of island parsing). A grammar to identify such elements of the source code model can be kept shallow, which reduces its complexity.

Grammar definitions come from two different sources:

Traditional Specifications: The specifications for literal numbers, for example, can be taken from the grammar specification of any programming language. These specifications are in regular expressions format and can just be copied.

Special Analysis: To identify syntax elements we often need to analyze their context:

- Is there a unique token which unambiguously identifies the syntax element?
  As an example, the scope operator `::` in C++ uniquely identifies namespace indicators.
- Is the potential position of syntax elements on the source line fixed?
  The labels in C, for example, can occur only at the beginning of a line and are delimited by a colon.
- Can short sequences of tokens identify a syntax element?
  An identifier which is not a keyword and is followed by the token '`(`' indicates a function invocation in most languages.

Regarding the language independence of parsers it can be generally stated that on the coarse level of source code understanding that we operate, the various programming languages mostly differ by the tokens used for the delimiters. An adaption of the regular expressions for a new programming language is therefore straightforward and can be automated in many cases. Some normalizations are however more complicated to adapt for another language, some others are even particular to a language. Since the parsers are small they are written easily.

We are going to specify separate partial parsers, or *recognizers*, for each element of the source model that we want to filter or normalize. Assembling a set of partial parsers which are triggered by some anchor tokens (for example the keyword **`class`** in C++ triggers a recognizer for member functions and variables) has been called *fuzzy parsing* in [Kop96]. Not integrating the different recognizers into a single grammar has a set of advantages and disadvantages as discussed in Table C.1.

Advantages	*Disadvantages*
We do not have to write and maintain a complete grammar which recognizes a program in its entirety. Partial parsers are therefore more robust and can deal, for example, with unpreprocessed code.	A full grammar which recognizes the entire structure of a program has better disambiguation capabilities.
Recognizers for coarse entities can be written concisely, especially since the more complicated syntactic entities are mostly skipped. Small specifications are easy to understand.	Reuse of grammar elements is more difficult. This may lead, for example, to duplication occurring in the specifications.
The areas of *interest* and *ignorance* of different recognizers will not align. For example, whereas one recognizer detects function calls, another skips everything within an array reference, even if it contains function calls. The recognizer specifications are easier to write if they can be formulated without being aware of such overlaps.	Each recognizer needs its own pass over the code. This is alleviated by the fact that a single recognizer usually skips more than 90% of the code.
Since the specification of a recognizer is easy to understand and change, it is possible to tune it *ad hoc* for the specific system that is investigated, without fearing to negatively affect other recognizing tasks.	

Table C.1: Advantages and disadvantages of having a set of small, independent recognizers instead of an integrated grammar for normalization purposes.

As the tool to implement the recognizers we have chosen the regular expressions (*RE*) found in PERL 5. We assume that many maintenance engineers already have a working knowledge of regular expressions, GREP being still one of the most popular tools for software maintenance. Modern RE engines integrate many features which overcome some of the shortcomings of traditional regular expressions: The PERL RE can be named and therefore easily reused which makes the specification of entire grammars feasible. Also, recursive definitions can now be written which makes it possible to recognize nested and balanced structures using REs. Filtering—removing completely—repetitive or nested structures, can often be achieved by re-applying a simple RE multiple times, *i.e.,* expressing the repetition in the programming language and not in the RE itself. Thanks to the tight integration between REs and PERL, recursion which is not permitted in the RE can be performed in the programming language. This removes more complexity from the RE than it adds to the program. This advantage is the benefit of the RE engine being embedded in a full programming language like PERL, where every kind of supporting task can be programmed easily.

A recognizer implemented using a regular expression engine combines lexical and syntactic analysis. We apply each of the recognizers individually to the source text, obeying some constraints for the order: comments and literal strings have to be removed from the source text prior to identifier normalization.

An important topic when using a rewriting approach is to make sure that the transformed code can be mapped back to the original code, the format in which we want to present the found clones to the engineers. The clones are reported on the granularity of lines. If we retain all line breaks in the rewriting, the mapping is trivial.

We are going to present four different examples of recognizers: *i)* delimited text, *e.g.,* commentary and literal strings, *ii)* recognizers for nested structures, *iii)* recognizers for identifiers and function names, and for *iv)* type casts.

Normalizing Delimited Text

Just as in a normal parsing pass, removing comments from the source text is the first step.

The following substitution recognizes comments from C/C++/JAVA programs. To be able to deal with special cases like comment delimiters within literal strings, the scanner also recognizes strings. The variable `$sourceText` holds the entire contents of a source file.

```
$sourceText =~ m{
        (
          /\*.*?\*/  # recognize multiline comments
         |
          //.*?\n    # recognize C++-style comments
         |
          "          # recognize literal strings
           (
             \\.     # do not choke on "\"" etc.
            |
             [^\\]   # recognizes all unescaped characters
           )*?
          "
         |
          '          # recognize literal characters,
           \\?       # do not choke on '\n' etc.
           .
          '
        )
              }xgs
```

Normalizing Nested Structures

Traditional regular expressions are unable to match structures which are nested or require a balanced number of delimiters [ASU86]. Current implementations of RE engines are however offering limited recursive capabilities. In PERL 5 we can for example recognize nested expressions with the following RE. The matching text is guaranteed to have balanced delimiters. The RE is given the name `$expression` so that it can be reused in other grammar productions.

```
$expression = qr{
              \(      # opening parenthesis
                (?:
                 (?>  # Non-capture group w/o backtracking
                  [^()]+
                 )
                |
                 (??{$expression}) #group w/ matching parens
                )*
              \)      # closing parenthesis
              }x;
```

The `$expression` RE illustrates how shallow the required syntax understanding is: all the complex expressions which potentially are found between the parentheses are skipped.

With the same principle we can identify nested blocks:

```
$nestedblock = qr(
                  {
                   (?:
                     (?> [^{}]+)
                    |
                     (??{$nestedblock})
                   )*
                  }
                 )x;
```

With the `$nestedblock` RE we can identify and filter away literal arrays, which when normalized usually result in much uninteresting duplication:

```
$sourceText =~ s/
                  (
                   \]\s*               # end of array typespec
                   (?:$identifier\s*)? # name of array variable
                   (?:=\s*)?           # assignment
                  )
                  ($nestedblock)  # (multidimensional) array
                  (\s*[;,)}])     # end of statement, parameter,
                                  # parameter list, or block
                /
                  "$1\{  \}$3"    # filter array contents
                /xeomg;
```

Using this RE we transform the left fragment into the right:

```
static unsigned long
      crc_32_tab[] = {
   0x00000000, 0x04c11db7,
   0x09823b6e, 0x0d4326d9,
   ...
};
```

```
static unsigned long
      crc_32_tab[] = { };
```

The `$nestedblock` RE can also be used to filter all code that lies outside of function bodies[1], a quite drastic normalization. The following RE removes everything outside of the top-level blocks.

```
$sourceText =~ s/
                  \G             # where the last match left of
                  .+?            # uninteresting  code
                  ($nestedblock) # a function body or type
                 /               # specification
                  $2
                 /xeomg;
```

With the `$expression` RE defined above (and its cousin `$arrayaccess`) we can define a pattern that matches any value which is represented as a single variable name, an array reference, or a function invocation:

```
$aValue = qr{
             $identifier      # the user defined name
              (
                $arrayaccess  # an array reference
               |
                $expression   # a function invocation
              )?
            }x;
```

[1] Note that problems may arise if block delimiters are not balanced due to conditional compilation.

With this library of REs we can recognize access chains and implement a filter which only retains the final function or variable. The code on the left is then, for example, transformed into the code on the right:

```
thing.fetch()->text;
list[i]->next->message(&p);
```

```
text;
message(&p);
```

This can be achieved with the following PERL substitution statement:

```
$sourceText =~ s/
                $aValue            # leftmost name

                (?:
                  \s*
                  (?:\.|->)        # pointer or ref
                  \s*
                  $aValue
                )+                 # can be a chain
              /
               getLastId($&)
              /xeomg;
```

The function `getLastId` removes all but the last instance of `$aValue` in the chain:

```
sub getLastId {
  my ($sourceText) = @_;
  $sourceText =~ s/$aValue(\.|\s*->\s*)//omg;
  return $sourceText
}
```

Normalizing Identifiers

One of the most important elements of the source model are names of functions and variables. Distinguishing the names of variables from the names of functions is useful since function names are much less volatile than variable names and might therefore not be normalized. For many programming languages a function name can be recognized by the parameter list that it is adjacent to. The first token of a parameter list is usually the '`(`'. The RE which recognizes names of functions is the following:

```
$sourceText =~ s/
                  $identifier  # normal identifier

                  (?=          # look-ahead assertion, does not
                               # include matched text in $&

                     \s*       # optional white space
                     \(        # start of parameter list

                  )
                /
                 foo           # generic function name
                /xeomg;
```

For a language like FORTRAN, where the token '`(`' after an identifier introduces a parameter list as well as access to an array member, we need a different expression however. In FORTRAN, function invocations are introduced with the keyword **`CALL`**. They can therefore be recognized with the following expression:

```
$sourceText =~ s/
                  (?i:         # match 'call' and 'CALL'
                   (?<CALL)    # look-behind assertion, does not
```

```
                                # include matched text in $&
                  )
                   (\s+)         # some white space

                  $identifier   # normal identifier
                 /
                  $1.foo        # generic function name
                 /xmeog;
```

The names of variables are then the strings which are not keywords and are not followed by a '(' token. The regular expression presented below implements a scanner with lookahead one non-whitespace character which recognizes identifiers in C code and replaces them with a generic token. A list of all the reserved words of the programming language is needed to distinguish identifiers from keywords (the implementation of `replaceIfNotKeyword()`, being only a comparison of strings with the keyword list, is not shown).

```
$sourceText =~ s/
              ((?>$identifier)) # no backtracking once matched

              (                  # Identifiers are followed
                \s*              # by white space
              )
            /
              # disambiguation between function
              # and identifier names
              if(length $' && substr($',0,1) eq '(') {
                $1.$2
              } else {
                replaceIfNotKeyword($1).$2
            /xeomg;
```

Filtering Typecasts

Filtering the typecasts in a statically typed language like C is difficult because there is no unique token that is involved in a typecast (to address this problem a recent C++ standard has introduced keywords for typecast operators, *e.g.*, **`dynamic_cast`**`<>` for navigating the class hierarchy). The general appearance of a typecast can be specified like this (where `$builtintype` is a list of all types that the programming language provides of its own).

```
$typecast = qr{
                \(\s*
                  (?:$builtintype|$identifier)
                  (?:\s|\*|&|\[\])*
                \s*\)
              }x;
```

Since this pattern also recognizes C code like **`sizeof`**`(`**`char`**` *)` which is not a typecast, we have to insert a separate disambiguation step after the match. The routine `hasIdentifierSuffix($string)` returns true if the token before the typecast is not an identifier, which would indicate that the entire syntax element is a function invocation[2].

```
$sourceText =~ s/
                 $typecast
                /
                 hasIdentifierSuffix($') ? '' : $&
                /xeomg;
```

[2] The exception from the exception is that a preceding **`return`** indicates again a typecast, *e.g.*, **`return`**` (`**`char`**` *) -1;`

Since type casts can frequently be enclosed in parentheses which are not removed by the above RE, the rewritten code still differs from code which would have been written if no typecast was involved. On the left we have the original code, on the right the code with type casts removed:

```
((MyThing &) list[i]).doit();
p = ((Widget *) getit())->var
```

```
(list[i]).doit();
p = (getit())->var
```

With another rewriting step we remove the superfluous parentheses. We take advantage of the regular expressions `$aValue` defined above which matches any value, be it the result of an array reference or a function invocation. We again have to make sure that no identifier precedes the match to avoid changing code like `foo(p)`.

```
$sourceText =~ s/
                \(\s*($aValue)\s*\)
               /
                hasIdentifierSuffix($') ? $1 : $&
               /xeomg;
```

Appendix D

Source Locations

A software system consists of a collection of documents which contain source code in the form of a character sequence. In order to delimit parts of the documents to identify source fragments, we need the notion of a *source location* which is the "address" of a single character in a given source document.

Definition D.0.1 (Source Location). *A* source location p *is a tuple* (ln, cn) *consisting in the line number ln and the column number cn. The set of all source locations is* $\mathcal{S}$ *which can be totally ordered by first ln and secondly cn.*

Usually, source code detection techniques have a coarser granularity than the individual character as the definition above could indicate. Most approaches work on the level of tokens, expressions, or statements, and some on the level of lines.

The notion of the *source fragment* designates a piece of source code which is an extract from a single source document, a consecutive sequence of characters including all of the source text.

Definition D.0.2 (Continuous Source Fragment). *A* source fragment f *is a continuous sequence of characters*[1] *between a start and stop location. It has the following attributes:*

- *Filename* $doc(f)$ *locates the fragment in the set of source documents that make up the system.*
- *The location* $beg(f)$ *designates where its text starts in the document.*
- *The location* $end(f)$ *designates where its text ends in* $doc(f)$.

It always holds that $beg(f) \leq end(f)$. *The characters at* $beg(f)$ *and* $end(f)$ *belong to the fragment.*

Source fragments can *contain* and *overlap* each other. The weaker of the two relations is *overlap*:

Definition D.0.3 (Overlap between Source Fragments) *Two source fragments* f_1 *and* f_2 *overlap, written*

$$overlap(f_1, f_2)$$

if either

$$beg(f_1) < beg(f_2) < end(f_1)$$

or

$$beg(f_1) < end(f_2) < end(f_1)$$

The overlap relation is reflexive, symmetric, but not transitive.

[1] For approaches that work with token or line granularity the start and stop locations can be projected onto the token or line granularity.

The stronger relation, thanks to transitivity, is *containment*:

Definition D.0.4 (Containment between Source Fragments) *The source fragment f_1 contains the fragment f_2, written*

$$\mathit{contains}(f_1, f_2)$$

if

$$\mathit{beg}(f_1) \leq \mathit{beg}(f_2) \leq \mathit{end}(f_1)$$

and

$$\mathit{beg}(f_1) \leq \mathit{end}(f_2) \leq \mathit{end}(f_1)$$

The contains relation is reflexive, transitive, but not symmetric.

Trivially, $\mathit{contains}(f_1, f_2) \Rightarrow \mathit{overlap}(f_1, f_2) \wedge \mathit{overlap}(f_2, f_1)$.

If copied code is changed, *i.e.*, statements are inserted and deleted, clones will consist of matching fragments interspersed with non-matching parts. To be able to describe such clones as a collection of non-overlapping but non-continuous fragments, we define the general source fragment:

Definition D.0.5 (General Source Fragment). *A* general source fragment g *is a tuple of continuous source fragments*

$$(x_1, \ \ldots \ , x_n)$$

where

$$\neg \ \mathit{overlap}(x_i, x_j), \ 1 \leq i, j \leq n$$

Note that source fragments are arbitrarily delimited pieces, they do not have to align with syntactic borders like blocks, statements or expressions. Such constraints can be established, if needed, later in the partitioning of the system's code or in the comparison process.

Appendix E

Example Systems

These tables show the sizes of the systems that we have used for our experiments. The last eight systems in the following table were used in the Bellon comparative study.[1]

System	*Size*		*Language*	*Origin*
	Files	LOC		
MAIL SORTING	101	39,000	C++	Industry
MFC 4.2	245	107,000	C++	Industry
ACCOUNTING	336	22,000	COBOL	Industry
APACHE 1.3.20	141	65,000	C	Open Source
JBOSS 2.3 BETA	403	35,000	JAVA	Open Source
BISON	54	14,000	C	Open Source
DIFFUTILS	24	10,000	C	Open Source
ZOPE 2.4.0	417	74,000	PYTHON	Open Source
AGREP 2.04	22	12,000	C	Academia
MULTIMARKE	70	7,000	JAVA	Stud.Proj.
ECLIPSE-ANT	178	15,000	JAVA	Open Source
ECLIPSE-JDTCORE	741	90,000	JAVA	Open Source
SWING	538	97,000	JAVA	Industry
NETBEANS-JAVADOC	101	8,000	JAVA	Industry
WELTAB	39	9,000	C	Industry
COOK	295	36,000	C	Open Source
SNNS	141	61,000	C	Academia
POSTGRESQL	322	127,000	C	Open Source

[1] Available from `http://www.bauhaus-stuttgart.de/clones/` [May 15, 2005]

Bibliography

[Aeb03] Tobias Aebi. Extracting Architectural Information using Different Levels of Collaboration. Diploma Thesis, University of Bern, September 2003.

[AGM+90] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215:403–410, 1990.

[AL98] Nicolas Anquetil and Timothy Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of CASCON'98, Toronto, Ontario, Canada*, pages 213–222, 1998.

[And02] Keith Andrews. *Visualizing Information Structures. Aspects of Information Visualization*. Professorial thesis, Technische Universität Graz, November 2002.

[AOK02] Mohamed Ibrahim Abouelhoda, Enno Ohlebusch, and Stefan Kurtz. Optimal exact string matching based on suffix arrays. In *Proc. of the Ninth Int. Symposium on String Processing and Information Retrieval*, LNCS. Springer Verlag, 2002.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, Reading, Mass., 1986.

[AWZ88] Bowen Alpern, Mark. N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 1–11, January 1988.

[Bak92] Brenda S. Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, 24:49–57, 1992.

[Bak93a] Brenda S. Baker. A Theory of Parameterized Pattern Matching: Algorithms and Applications (Extended Abstract). In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 71–80, May 1993.

[Bak93b] Brenda S. Baker. On Finding Duplication in Strings and Software. *Journal of Algorithms*, 1993. To appear.

[Bak95a] Brenda S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the Second IEEE Working Conference on Reverse Engineering (WCRE)*, pages 86–95, July 1995.

[Bak95b] Brenda S. Baker. Parameterized Pattern Matching by Boyer-Moore Type Algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 541–550, January 1995.

[Bal99] Magdalena Balazinska. Reconception de systèmes orientés-objet basée sur l'analyse des clones. Master's thesis, École Polytechnique de Montréal, November 1999.

[BCHK99] Brenda S. Baker, Kenneth W. Church, Jonathan I. Helfman, and Brian W. Kernighan. Methods and apparatus for detecting and displaying similarities in large data sets. United States Patent 5,953,006, September 1999.

[Bel02a] Detection of software clones: Tool comparison experiment, 2002. http://www.bauhaus-stuttgart.de/clones/.

[Bel02b] Stefan Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Master's thesis, Universität Stuttgart, September 2002.

[BEW95] Richard A. Becker, Stephen G. Eick, and Allan R. Wilks. Visualizing network data. *IEEE Transaction on Visualization and Computer Graphics*, 1(1):16–21, March 1995.

[BM97] Elizabeth Burd and Malcolm Munro. Investigating the maintenance implications of the replication of code. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, September 1997.

[BMD+99] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Laguë, and Kostas Kontogiannis. Measuring clone based reengineering opportunities. In *Metrics '99*, pages 292–303, 1999.

[BMD+00] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Laguë, and Kostas Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In Françoise Balmas and Kostas Kontogiannis, editors, *Proceedings Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 98–107. IEEE Computer Society, October 2000.

[BMLK99] Magdalena Balazinska, Ettore Merlo, Bruno Laguë, and Kostas Kontogiannis. Measuring clone based reengineering opportunities. In *Proceedings Sixth IEEE International Symposium on Software Metrics*, pages 292–303. IEEE Computer Society, November 1999.

[Bro95] Frederick P. Brooks. *The Mythical Man-Month*. Addison Wesley, Reading, Mass., 2nd edition, 1995.

[BRS+97] Ilene Burnstein, Katherine Roberson, Floyd Saner, Abdul Mirza, and Abdallah Tubaishat. A role for chunking and fuzzy reasoning in a program comprehension and debugging tool. In *Proceedings of the 9th International Conference on Tools with Artificial Intelligence (TAI-97)*. IEEE Press, November 1997.

[BSD03] Andrew P. Black, Nathanael Schärli, and Stéphane Ducasse. Applying traits to the Smalltalk collection hierarchy. In *Proceedings OOPSLA'03 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, volume 38, pages 47–64, October 2003.

[BYG99] Ricardo A. Baeza-Yates and Gaston H. Gonnet. A fast algorithm on average for all-against-all sequence matching. In *Proceedings of the String Processing and Information Retrieval Symposion (SPIRE)*, pages 16–23. IEEE, 1999.

[BYM+98] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant' Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of ICSM*. IEEE, 1998.

[BYN00] Ricardo A. Baeza-Yates and Gonzalo Navarro. Block addressing indices for approximate text retrieval. *Journal of the American society of Information Sciences*, 51(1):69–82, 2000.

[CC97] Jason S. Chang and Mathis H. Chen. An alignment method for noisy parallel corpora based on image processing techniques. In *Proceedings 35th Annual Meeting of the Association for Computational Linguistics*, 1997.

[CDS04] James R. Cordy, Thomas R. Dean, and Nikita Synytskyy. Practical language-independent detection of near-miss clones. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research (CASCON 04)*, pages 1–12, Markham, Ontario, Canada, 2004. IBM Press.

[CH93] Kenneth Ward Church and Jonathan Isaac Helfman. Dotplot: A program for exploring self-similarity in millions of lines for text and code. *J. Computational and Graphical Statistics*, 2(2):153–174, June 1993.

[Chu93] Kenneth Ward Church. Char_align: A program for aligning parallel texts at the character level. In *31st Annual Meeting of the Association for Computational Linguistics*, pages 1–8. Association for Computational Linguistics, June 1993.

[CIQW+95] James H. Cross II, Alex Quilici, Linda Wills, Philip Newcomb, and Elliot Chikofsky. Second working conference on reverse engineering summary report. *SIGSoft Software Engineering Notes*, 20(5), December 1995.

[Clo00] Paul Clough. Plagiarism in natural and programming languages: An overview of current tools and technologies. Technical report, University of Sheffield, Department of Computer Science, June 2000.

[CM93] Pádraig Cunningham and Alexander N. Mikoyan. Using CBR techniques to detect plagiarism in computing assignments. Technical Report TCD-CS-93-22, 1993.

[CMS99] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman, editors. *Readings in Information Visualization — Using Vision to Think*. Morgan Kaufmann, 1999.

[Cor03] James R. Cordy. Comprehending reality—practical barriers to industrial adoption of software maintenance automation. In *Proc. 11th Int. Workshop on Program Comprehension (IWPC'03)*, pages 196–205, Portland, Oregon, USA, May 2003. IEEE.

[DBF+95] N. Davey, P. Barson, S.D.H. Field, R.J. Frank, and D.S.W. Tansley. The development of a software clone detector. *International Journal of Applied Software Technology*, 1(3/4):219–236, 1995.

[DCG93] Ido Dagan, Kenneth W. Church, and William A. Gale. Robust bilingual word alignment for machine aided translation. In *Proceedings of the Workshop on Very Large Corpora: Academic and Industrial Perspectives*, pages 1–8, Columbus, OH, 1993.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[dJ00] Merijn de Jonge. A pretty printer for every occasion. In *Workshop Proceedings 2nd International Symposion on Constructing Software Engineering Tools CoSET 2000*, pages 68–77. IEEE, June 2000.

[DMLP98] Michel Dagenais, Ettore Merlo, Bruno Laguë, and Daniel Proulx. Clones occurrence in large object oriented software packages. In *Proceedings of CASCON 1998*, pages 192–200, 1998.

[DNR05] Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the effectiveness of clone detection by string matching. *International Journal on Software Maintenance: Research and Practice*, pages 00–00, 2005. To appear.

[DRD99] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings ICSM '99 (International Conference on Software Maintenance)*, pages 109–118. IEEE, September 1999.

[ESEE92] Stephen G. Eick, Joseph L. Steffen, and Sumner Eric E., Jr. SeeSoft—A Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[FG99] Paolo Ferragina and Roberto Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[Fit69] Walter M. Fitch. Locating gaps in amino acid sequences to optimize the homology between two proteins. *Biochemical Genetics*, 3:99–108, 1969.

[FZMS02] Raphael A. Finkel, Arkady Zaslavsky, Krisztian Monostori, and Heinz Schmidt. Signature extraction for overlap detection in documents. In Michael J. Oudshoorn, editor, *Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, Melbourne, Australia, 2002. ACS.

[GFC04] Mohammad Ghoniem, Jean-Daniel Fekete, and Philippe Castagliola. A comparison of the readability of graphs using node-link and matrix-based representations. In *Proceedings of the 10th IEEE Symposium on Information Visualization (InfoVis'04)*, pages 17–24, Austin, TX, October 2004. IEEE Press.

[Gie03] Simon Giesecke. Clone–based Reengineering für Java auf der Eclipse–Plattform. Master's thesis, Carl von Ossietzky Universität Oldenburg, Germany, September 2003.

[GIHK+97] James E. Goodnow II, Jonathan I. Helfman, Thaddeus J. Kowalski, John J. Puttress, James R. Rowland, and Carl R. Seaquist. Method of identifying similarities in code segments. United States Patent 5,699,507, December 1997.

[GJK01] William G. Griswold, Jimmy J. Juan, and Yoshikiyo Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 2001 International Conference on Software Engineering (ICSE 2001)*. IEEE Computer Society, March 2001.

[Gla02] Leon Glass. Looking at dots. *The Mathematical Intelligencer*, 24(4):37–43, 2002.

[GLW+95] Nahum Gershon, Joshua LeVasseur, Joel Winstead, James Croall, Ari Pernick, and William Ruh. Case study visualizing internet resources. In *Proceedings of the Conference on Information Visualization (INFOVIZ '95)*, pages 122–128. IEEE, 1995.

[GM70] A.J. Gibbs and G.A. McIntyre. The diagram: a method for comparing sequences. its use with amino acid and nucleotide sequences. *Eur. J. Biochem.*, 16:1–11, 1970.

[Gre00] Howard Greisdorf. Relevance: An interdisciplinary and information science perspective. *Informing Science*, 3(2), 2000. Special Issue on Information Science Research.

[Gri98] William G. Griswold. Coping with software change using software transparency. Technical Report CS98-585, University of California, San Diego, Department of Computer Science and Engineering, April 1998.

[GT00] Michael Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, pages 131–142. IEEE Computer Society, 2000.

[Hel94] Jonathan I. Helfman. Similarity patterns in language. In *Proceedings of IEEE Symposium on Visual Languages*, pages 173–175, 1994.

[Hel95] Jonathan Helfman. Dotplot Patterns: a Literal Look at Pattern Languages. *TAPOS*, 2(1):31–41, 1995.

[HHHW98] Beth Hetzler, W. Michelle Harris, Susan Havre, and Paul Whitney. Visualizing the full spectrum of document relationships. In *Structures and Relations in Knowledge Organization. Proceedings 5th Int. ISKO Conference*, pages 168–175, Würzburg, 1998. ERGON Verlag.

[Hor90] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 234–245, White Plains, NY, June 1990.

[HPR89] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non-interfering versions of programs. *ACM Trans. Programming Languages and Systems*, 11(3):345–387, July 1989.

[HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison Wesley, 2000.

[HUK+02] Yoshiki Higo, Yasushi Ueda, Toshihro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On software maintenance process improvement based on code clone analysis. In *Proceedings 4th International Conference on Product Focused Software Process Improvement (Profes 2002)*, December 2002.

[Ing92] Peter Ingwersen. *Information Retrieval Interaction*. Taylor Graham, London, 1992.

[Jan88] Hugo T. Jankowitz. Detecting Plagiarism in Student PASCAL Programs. *Computer Journal*, 1(31):1–8, 1988.

[Joh93] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of CASCON 93*, pages 171–183, 1993.

[Joh94a] J. Howard Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintence*, pages 120–126, 1994.

[Joh94b] J. Howard Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of CASCON '94*, pages 9–18, 1994.

[Joh96] J. Howard Johnson. Navigating the textual redundancy web in legacy source. In *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative Research*. IBM Centre for Advanced Studies, IBM Press, 1996.

[JS96] Dean F. Jerding and John T. Stasko. The Information Mural: Increasing Information Bandwidth in Visualizations. Technical Report GIT-GVU-96-25, Georgia Institute of Technology, October 1996.

[JS03] Stan Jarzabek and Li Shubiao. Eliminating redundancies with a 'composition with adaptation' meta-programming technique. In *Proceedings ESEC-FSE'03, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 237–246. ACM Press, September 2003.

[KCS02] Cory Kapser, Jack Chi, and Maher Shinouda. A project on real world cloning: Cloning in linux file systems. Class project, School of Computer Science, University of Waterloo, Ontario, Canada, November 2002.

[KG03] Cory Kapser and Michael W. Godfrey. Toward a taxonomy of clones in source code: A case study. In *Proceedings of the First International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA)*. IEEE, September 2003.

[KG04] Cory Kapser and Michael W. Godfrey. Aiding comprehension of cloning through categorization. In *Proceedings of 2004 International Workshop on Software Evolution (IWPSE-04)*, Kyoto, Japan, September 2004.

[KH01a] Raghavan Komondoor and Susan Horwitz. Tool demonstration: Finding duplicated code using program dependences. In *Proc. European Symposium on Programming (ESOP)*, volume 2028 of *LNCS*, pages 383–386. Springer-Verlag, 2001.

[KH01b] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis (SAS)*, Paris, France, July 2001. Springer-Verlag.

[KH02] Raghavan Komondoor and Susan Horwitz. Eliminating duplication in source code via procedure extraction. Technical Report 1461, UW-Madison Dept. of Computer Sciences, December 2002.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(6):654–670, 2002.

[KN01] Georges Golomingi Koni-N'sapu. A scenario based approach for refactoring duplicated code in object oriented systems. Diploma thesis, University of Bern, June 2001.

[Kon97] Kostas Kontogiannis. Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics. In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 44 — 54. IEEE Computer Society, 1997.

[Kop96] Rainer Koppler. A systematic approach to fuzzy parsing. *Software: Practice and Experience*, 27(6):637–649, 1996.

[Kri01] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eigth Working Conference on Reverse Engineering (WCRE'01)*, pages 301–309. IEEE Computer Society, October 2001.

[Kru92] Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, September 2003.

[Lei03] António M. Leitão. Detection of redundant code using R2D2. In *Proc. Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 183–192. IEEE, September 2003.

[Lin98] Dekang Lin. An information-theoretic definition of similarity. In *Proceedings of the 15th ICML*, pages 296–304, Madison WI, 1998.

[LLWY03] Arun Lakhotia, Junwei Li, Andrew Walenstein, and Yun Yang. Towards a clone detection benchmark suite and results archive. In *Proc. of the 11th International IEEE Workshop on Program Comprehension (IWPC'03)*, pages 285–286. IEEE, May 2003.

[LM04] Arun Lakhotia and Moinuddin Mohammed. Imposing order on program statements to assist anti-virus scanners. In *Proceedings of Eleventh Working Conference on Reverse Engineering (WCRE'04)*, pages 161–170, Delft, the Netherlands, November 2004. IEEE Computer Society.

[LYW03] Junwei Li, Yun Yang, and Andrew Walenstein. Clone detector benchmark suite and results archive. In *Proceedings IWPC 2003*, Portland, Oregon, May 2003.

[MAK] Ettore Merlo, Giulio Antoniol, and Jens Krinke. Identifying similar code with metrics and program dependence graphs. To appear.

[Mal99] Pietro Malorgio. An information mural visualization for duploc. Informatikprojekt, University of Bern, July 1999.

[McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *JACM*, 23(2):262–272, April 1976.

[McL71] A.D. McLachlan. Tests for comparing related amino-acid sequences. cytochrome c and cytochrome c_{551}. *J. Mol. Biol.*, 61:409–424, 1971.

[Meh04] Michael Mehlich. Transformation systems for real programming languages. preprocessing directives everywhere. In Ying Zou and James R. Cordy, editors, *Proceedings of the First International Workshop on Software Evolution Transformations (SET)*, pages 25–28, Delft, the Netherlands, November 2004.

[Mel96a] I. Dan Melamed. Automatic detection of omissions in translations. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING'96)*, Copenhagen, Denmark, 1996.

[Mel96b] I. Dan Melamed. A geometric approach to mapping bitext correspondence. In Eric Brill and Kenneth Church, editors, *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 1–12. Association for Computational Linguistics, Somerset, New Jersey, 1996.

[Mel99] I. Dan Melamed. Bitext maps and alignment via pattern recognition. *Computational Linguistics*, 25(1):107–130, 1999.

[MLH96] Jean Mayrand, Bruno Laguë, and John Hudepohl. Evaluating the benefits of clone detection in the software maintenance activities in large scale systems. In *Workshop on Empirical Software Studies, Monterey, California, USA*, November 1996.

[MLM96a] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Automatic detection of function clones in a software system using metrics. In *Proceedings of ICSM (International Conference on Software Maintenance)*, 1996.

[MLM96b] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software Maintenance (ICSM)*, pages 244–253, 1996.

[MM90] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *1st ACM SIAM Symposium on Discrete Algorithms*, pages 319–327, San Francisco, January 1990.

[MM00] Jonathan I. Maletic and Andrian Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *Proceedings of the 12th International Conference on Tools with Artificial Intelligences (ICTAI 2000)*, pages 46–53. IEEE, November 2000.

[MM01] Jonathan I. Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *International Conference on Software Engineering*, pages 103–112, 2001.

[Moo01] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings WCRE 2001*, pages 13–22. IEEE Computer Society, October 2001.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *CACM*, 22(2):96–103, February 1979.

[Mur96] Gail C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington, 1996.

[NSTT00] Gonzalo Navarro, Erkki Sutinen, Jani Tanninen, and Jorma Tarhio. Indexing text with approximate q-grams. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, number 1848 in LNCS, pages 350–363. Springer Verlag, London, 2000.

[NW70] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarity in the amino acid sequences of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.

[Opp80] Derek C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(4):465–483, October 1980.

[PHKV93] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings OOPSLA '93*, pages 326–337, October 1993.

[PK82] J. Pustell and F. Kafatos. A high speed, high capacity homology matrix: Zooming through sv40 and polyoma. *Nucleid Acids Research*, 10(15):4765–4782, 1982.

[PMDL99] Jean-Francois Patenaude, Ettore Merlo, Michel Dagenais, and Bruno Laguë. Extending software quality assessment techniques to java systems. In *Proceedings Seventh International Workshop on Program Comprehension*, page 49, May 1999.

[PMP00] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. JPlag: Finding plagiarism among a set of programs. Technical Report 2000-1, Universität Karlsruhe, Fakultät für Informatik, March 2000.

[RB88] A.H. Reisner and C.A. Bucholtz. The use of various properties of amino acids in color and monochrome dot-matrix analyses for protein homologies. *CABIOS*, 4(3):395–402, 1988.

[RC93] Mary Beth Rosson and John M. Carroll. Active programming strategies in reuse. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 4–20, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[RD98] Matthias Rieger and Stéphane Ducasse. Visual detection of duplicated code. In Stéphane Ducasse and Joachim Weisbrod, editors, *Proceedings ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, number 6/7/98 in FZI Report. Forschungszentrum Informatik Karlsruhe, 1998.

[RDL04] Matthias Rieger, Stéphane Ducasse, and Michele Lanza. Insights into system-wide code duplication. In *Proceedings of WCRE 2004 (11th Working Conference on Reverse Engineering)*. IEEE Computer Society Press, November 2004.

[Rey94] Jeffrey C. Reynar. An automatic method of finding topic boundaries. In *Proceedings of the 32. Meeting of the Association for Computational Linguistics*, pages 331–333, 1994.

[Rey98] Jeffrey C. Reynar. *Topic Segmentation: Algorithms and Applications*. PhD thesis, University of Pennsylvania, 1998.

[RKS99] O. Rüthing, J. Knoop, and B. Steffen. Detecting equalities of variables: Combining efficiency with precision. In A. Cortesi and G. File, editors, *Proceedings of the 6th Static Analysis Symposium (SAS'99), Venice (Italy)*, volume 1694 of *LNCS*, pages 232–247. Springer-Verlag, September 1999.

[SD95] Erik L.L. Sonnhammer and Richard Durbin. A dot-matrix program with dynamic threshold control suited for genomic DNA and protein sequence analysis. *Gene*, 167:1–10, October 1995.

[SDBP98] John T. Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization — Programming as a Multimedia Experience*. The MIT Press, 1998.

[SEH03] Susan E. Sim, Steve M. Easterbrook, and Richard C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *Proceedings, 25th International Conference on Software Engineering (ICSE'03*, Portland, Oregon, May 2003.

[Sim76] Charles Simonyi. *Meta-Programming: A Software Production Method*. PhD thesis, XEROX PARC, December 1976.

[Sim96] Loren Taylor Simpson. *Value-Driven Redundancy Elimination*. PhD thesis, Rice University, May 1996.

[TUL71] Ignacio Tinoco, jun., Olke C. Uhlenbeck, and Mark D. Levine. Estimation of secondary structure in ribonucleic acids. *Nature*, 230:362–367, April 1971.

[UKKI02a] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proc. of the 8th IEEE Symposium on Software Metrics (METRICS2002)*, pages 67–76, Ottawa, Canada, June 2002.

[UKKI02b] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On detection of gapped code clones using gap locations. In *Proceedings Ninth Asia-Pacific Software Engineering Conference (APSEC'02)*, pages 327–336, Gold Coast, Australia, December 2002. IEEE.

[vDK99] Arie van Deursen and Tobias Kuipers. Building document generators. In Hongji Yang and Lee White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 40–49. IEEE, September 1999.

[vH03] Frank van Ham. Using multilevel call matrices in large software projects. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 29–34, Seattle, Washington, October 2003. IEEE.

[vR79] C. v. Rijsbergen. *Information Retrieval*. Butterworths, London, 1979.

[vRD03] Filip van Rysselberghe and Serge Demeyer. Evaluating clone detection techniques. In *Proceedings of the International Workshop on Evolution of Large Scale Industrial Applications (ELISA)*, pages 25–36, 2003.

[vRD04] Filip van Rysselberghe and Serge Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *Proc. 19. Intl. Conference on Automated Software Engineering (ASE'04)*. IEEE, September 2004.

[War00] Colin Ware. *Information Visualization*. Morgan Kaufmann, 2000.

[WCO00] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly & Associates, Inc., 3rd edition, 2000.

[WHHE84] C. Thomas White, Stephen C. Hardies, Clyde A. Hutchison, III, and Marshall H. Edgell. The diagonal-traverse homology search algorithm for locating similarities between two sequences. *Nucleic Acids Research*, 12(1):751–766, 1984.

[Wis96] Michael J. Wise. YAP3: Improved detection of similarities in computer program and other texts. *SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 28, 1996.

[WJL+03] Andrew Walenstein, Nitin Jyoti, Junwei Li, Yun Yang, and Arun Lakhotia. Problems creating task-relevant clone detection reference data. In *Proceedings 10th Working Conference on Reverse Engineering (WCRE'03)*, pages 285–295, Victoria, B.C., Canada, November 2003. IEEE.

[WL03] Andrew Walenstein and Arun Lakhotia. Clone detector evaluation can be improved: Ideas from information retrieval. In *Proceedings of the 2nd International Workshop on Detection of Software Clones (IWDSC'03)*, November 2003.

[WLK04] Andrew Walenstein, Arun Lakhotia, and Rainer Koschke. The second international workshop on detection of software clones: workshop report. *SIGSOFT Softw. Eng. Notes*, 29(2):1–5, 2004.

[WW02] Debora Weber-Wulff. Schummeln mit dem Internet. *c't Magazin für Computertechnik*, (1):64–69, January 2002.

[YMKI02] Tetsuo Yamamoto, Makoto Matsushita, Toshihiro Kamiya, and Katsuro Inoue. Measuring similarity of large software systems based on source code correspondence. Technical Report IIP-03-03-02, Osaka University, Department of Information and Computer Scineces, IIP Lab, March 2002.

[ZG03] Lijie Zou and Michael W. Godfrey. Detecting merging and splitting using origin analysis. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 146–154, November 2003.

[ZHW01] Justin Zobel, Steffen Heinz, and Hugh E. Williams. In-memory hash tables for accumulating text vocabularies. *Inf. Process. Lett.*, 80(6):271–277, 2001.