Diploma Thesis

4th October 2005

Evolution Impact of Code Clones

Identification of Structural and Change Smells based on Code Clones

Reto Geiger

of Zürich, ZH, Switzerland (98-915-184)

supervised by

Prof. Dr. Harald Gall

Beat Fluri

University of Zurich

Department of Informatics

s.e.a.l.

software evolution & architecture lab

Diploma Thesis

Evolution Impact of Code Clones

Identification of Structural and Change Smells based on Code Clones

Reto Geiger

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Diploma Thesis

Author: Reto Geiger, r.geiger@enerprice.ch

Project period: 4th April 2005 - 4th October 2005

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

Many persons deserve thanks for their help in the realization of this thesis. I would first like to express my thanks to all persons who supported me during the last six months but remain unnamed in this acknowledgement.

Special thanks goes to Professor Harald Gall for giving me the opportunity of writing this thesis. It was definitely a highlight during my studies at the University of Zürich. I would like to thank Beat Fluri for his support and input during the last six months. Thanks also go to Martin Pinzger for answering many questions.

I would like to thank the CCFinder development team at the University of Osaka for providing this powerful tool for the use in this thesis.

Special thanks also goes to my family: my father Werner Geiger for the interesting discussions and his interest in this thesis; my mother Marie-Theres Geiger for proofreading this work on a subject so different from her usual interests; my brother Gaudenz Geiger for his encouragement and his insistence that I relax sometimes.

Abstract

Code clones have long been recognized as *bad smells* in software systems and are assumed to cause maintenance problems during their evolution. We can identify these problems by the examination of change coupled files. It is broadly assumed that the more clones two files share, the more often they have to be changed together. This connection between clones and change couplings has been postulated but neither demonstrated nor quantified yet. However, it simplifies the identification of code clones which are suitable candidates for refactoring.

In this thesis we examine if a correlation can be verified. Using a newly developed framework we examine the code clones and change couplings of a large software system and attempt to correlate these measurements. The results obtained are statistically ambiguous making a fully automated selection process impossible. A different amount of code clones does not necessarily lead to a related difference in the number of change couplings. A decision about if and when to refactor a code clone cannot be based on its presence or size alone. We therefore propose a set of metrics and a visualization technique allowing the software engineer to base her decisions on more sophisticated information.

Zusammenfassung

Duplizierungen im Quelltext eines Software–Systems sind schon lange als *bad smells* bekannt und man nimmt an, dass sie zu Problemen während der Evolution des Systems führen. Wir können solche Probleme daran erkennen, dass gewisse Dateien oft gemeinsam geändert werden müssen. Man nimmt an, dass je mehr Quelltext zwei Dateien gemeinsam haben, desto öfter sie auch gekoppelt geändert werden müssen. Eine solche Beziehung zwischen Klonen und Änderungskopplungen wurde bisher postuliert, aber nicht nachgewiesen. Wäre eine solche Quantifizierung jedoch möglich, so würde sie die Identifizierung von Klonen vereinfachen, welche sich als Kandidaten für ein Refactoring anbieten.

In dieser Arbeit untersuchen wir, ob eine solche Beziehung nachgewiesen werden kann. Mit Hilfe eines neuentwickelten Ansatzes untersuchen wir Klone und Änderungskopplungen und versuchen diese Messwerte in Beziehung zu setzen. Die Resultate, welche wir erhalten haben, sind statistisch nicht eindeutig. Eine Änderung der Grösse der Klone führt nicht zwingend zu einer Anpassung der Anzahl von Kopplungen. Allein aufgrund der Präsenz oder Länge eines Klons kann deshalb nicht entschieden werden, ob und wann er mittels Refactoring entfernt werden sollte. Deshalb schlagen wir eine Reihe von Metriken und eine Visualisierungstechnik vor, welche es einem Software–Ingenieur erlauben, diese Entscheidung auf eine beitere Basis an Informationen abzustützen.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> Begun the clone war has.
>
> Master Yoda, Star Wars Episode 2

Code clones are regarded as a major *bad smell* in a software system. This thesis will explore how prominent their impact on the evolution of a large software project is.

1.1 Setting the Scene

This thesis touches several fields of research in software engineering. Some of these have been known and treated for years while others are still relatively new. So far the correlations between these different fields have not been explored fully. This is attempted in this thesis.

1.1.1 Bad Smells and Code Clones

Martin Fowler has introduced the concept of a bad smell in [FBB+99]. A bad smell is a warning sign about potential problems in source code [Wak03]. These indicators suggest the possibility of refactoring if they are detected. They do, however, not give a precise set of metrics which tell the engineer just when to start refactoring a system.

Duplicated fragments of source code - referred to as code clones - are a prominent and well researched kind of a bad smell. In fact, Fowler refers to duplicated code as the "number one in the stink parade". This quote gives an idea of how important he suggests that code clones are. Several methods and tools for detecting code clones have been proposed and are discussed in detail in Chapter 2.

Code duplication is often cited as one of the major smells and a system containing a large proportion of duplicated code is considered to be difficult to maintain. There are different forms of code clones: syntactic or semantic duplication. As semantic equivalence is hard to detect, most of the available clone detection tools focus on finding syntactic duplications. It is estimated that normal industrial source code contains 5 – 20 % of duplicated fragments [MLM96].

Code clones have several unwanted consequences on the system. Generally the code is bloated unnecessarily. More code leads to more possibilities for introducing defects into the system. The introduction of dead code is not really needed for the system to perform its functionality can also be a consequence. The work necessary for changing any part of the system is increased according to the larger volume of code [LB85].

1.1.2 Software Evolution and Maintenance

The development of a software system is usually not finished when the first version is released. The evolution of a software system continues after this initial delivery. Changes and improvements are necessary during the maintenance phase of its lifetime [LB85]. There is no standard definition of software evolution but evolution in general can be defined as

> a process of continuous change from a lower, simpler, or worse to a higher, more complex, or better state. [GT03]

Software maintenance is the main driver of behind the evolution of a system after its initial development. It is defined in IEEE Standard 1219 as

> The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment. [IEE93]

The financial impact of maintenance is grave – the costs of changes carried out after delivery are estimated at 40 – 70 % of the total costs during a system's lifetime [GT03]. Any reduction of the maintenance and evolution effort of a software system is therefore highly desirable.

As bad smells are mostly indicators for maintainability problems of the source code, they lose their significance if the system remains stable and is never changed after its initial release. Unfortunately software systems which are actively being used to solve problems in the real world are never completely stable during their lifetime. Lehman postulated this already in 1985 [LB85]. He formulated a number of laws, some of which explain why software has to evolve:

- **Law of continuing change**: Systems must be continually adapted or they become progressively less satisfactory to use.
- **Law of increasing complexity**: As a system evolves, its complexity increases unless work is done to maintain or reduce it.
- **Law of continuing growth**: Functional capability must be continually increased over a system's lifetime to maintain user satisfaction.
- **Law of declining quality**: Unless rigorously adapted to meet changes in the operational environment, system quality will appear to decline.

Basically, a system has to evolve so that its users remain satisfied. Negative effects of these necessary changes include increasing complexity and decreasing understandability. To counter them, the resulting bad smells must be detected during the system's lifetime. This detection forms a first step to correct such problems through refactoring. In this case the postulated negative influence of code clones on the software's maintainability comes into play. Code duplication increases the workload for changes and cloned code drastically reduces the understandability of the code. Thus, code clones are a major factor that must be considered during the evolution of a system.

An expression of maintainability problems during the evolution of a system are change couplings. The definition of a change coupling simply implies that two files are coupled by being changed at the same time [GHJ98]. If such couplings occur only sporadically, they do not present a major problem. If on the other hand two files are changed together during the whole evolutionary process, a refactoring should probably be considered.

Code clones comprise a major bad smell. A significant correlation between code clones and change couplings has been assumed for some time now. We therefore expect that changes in the amount of duplicated code fragments should be followed by an observable change to the couplings between two files.

1.2 Goal

1.2.1 Research Goal

In this thesis, we cover the question whether a correlation between code clones and the presence of change couplings exists or not.

The detection of code clones has been a research topic for a considerable time since the presence of duplicated code was identified as one of the main bad smells a software system can contain. More recently, further smells have been described, which affect systems not only during their initial development but rather during their evolution over time.

If such a correlation can be derived from the data obtained in a large system, code clone detection could be used to predict future maintenance problems. Even if there is no systematic correlation between code clones and change couplings, problematic areas in the source code could be better identified when those two examination techniques are combined. Code duplications suitable for refactoring could be more effectively identified by using the two methods together. The historical data enriches the clone detection and helps to decide whether to refactor or not.

To find an answer to this research goal, we apply code clone detection tools to different releases of a large software system. We then compare the results of this analysis with data collected about the evolutionary and change behavior of the same system to see if the occurrence of duplicated source code and couplings between different files are correlated.

While answering the research question, we use different tools and give an evaluation of their applicability to this task.

In summary, the research goal is to develop a way of correlating code clone and change coupling data so that we can give more precise conclusions about the evolutionary behavior of a system.

It is widely accepted that code clones pose a major problem during the evolution of a system. We were therefore surprised at how inconclusive this correlation turns out be in our case study.

1.2.2 Case Study: The Mozilla Project

To validate our approach, we selected as a case study the open source Mozilla project web browser. Its entire source code is available in a CVS repository. The system has already been used in several software evolution and change coupling studies [FPG03, FGP05].

The first milestone release of this project occurred in early 1999 while the most recent release is available since September 21st, 2005.

For this thesis seven releases have been selected, starting with release 0.9.2 (June 28th, 2001) and finishing with 1.7 (June 17th, 2004). The releases considered are each separated from the next by an interval of about six months.

One advantage of the Mozilla project is the presence of clearly defined software modules allowing to reduce the complexity of the clone detection process. Furthermore, a large part of the system is written in the C/C++ programming languages supported by most clone detection tools.

Table 1.2.2 shows an overview of the seven releases considered with the respective number of C/C++ files (NOCF) and the lines of code within these files (LOCC).We do not use all of these files in the development of this thesis.

[1] NOCF: number of C/C++ files.

[2] LOCC: lines of C/C++ code including blank lines and comments as these lines are also processed by a detection tool.

Release	Date	NOCF[1]	*.h	*.c	*.cpp	LOCC[2]
0.9.2	June 28th, 2001	11,619	6,171	1,600	3,848	3,306,470
0.9.7	December 21st, 2001	11,945	6,385	1,653	3,907	3,518,502
1.0	June 5th, 2002	11,191	5,259	1,970	3,962	3,868,403
1.3a	December 13th, 2002	13,314	7,342	1,824	4,148	3,924,442
1.4	June 30th, 2003	13,646	7,599	1,849	4,198	3,986,846
1.6	January 15th, 2004	13,260	7,505	1,563	4,192	3,835,584
1.7	June 17th, 2004	13,563	7,681	1,579	4,303	3,913,042

Table 1.1: Mozilla releases considered in the case study.

1.3 Structure of the Thesis

Chapter **??** presents related work that has been done in the area of code clone detection and the impact of these duplications to the evolution of software ; Section **??** presents the fra. We then introduce the theoretical background of the framework by which we examine the Mozilla case study in Chapter 4. The application of this framework is covered in Chapter 5. In Chapter 6 we summarize the thesis by presenting its contribution, lessons learned, and we raise questions for future work.

Chapter 2

State of the Clone

In this chapter we give an introduction to the terminology and approaches to code clone detection. We describe seven different techniques and implementations. The effect of code duplications on the evolution of a software system is shortly covered. Finally we describe the two most common visualization techniques used in clone detection tools.

2.1 Definitions

The literature currently gives no precise and consistent definition of what a code clone actually is. This section therefore defines the expressions used in the context of this paper.

Definition 1 ***(Code Clone)*** *A code clone is a piece of replicated source code that offers duplicate or near duplicate functionality. [LPM+97, BB02]*

This definition is neither precise nor conclusive due to the lack of a suitable metric to measure the similarity of two functions. Whether or not two sections of a software system are considered a clone pair often depends on the point of view. The following definition can be considered as more practical – yet even less precise.

Definition 2 ***(Code Clone)*** *A code clone is a piece of replicated source code that has been detected by a clone detection tool, and that the user identifies as a clone.*

The latter definition takes into account the assessment of a software engineer using a given clone detection tool. The introduction of detection tools also implies the definition of some additional terms.

Definition 3 ***(Candidate)*** *A candidate is a possible clone reported by a detection tool.*

Definition 4 ***(False Positive)*** *A false positive is a reported candidate that is not a clone.*

Definition 5 ***(False Negative)*** *A false negative is a clone that is not reported.*

The problems caused by false negatives are more difficult to solve than those based on false positives. To identify a false negative it would be necessary to know all duplications in the source code. If that was known, a clone detection tool would not be needed. In our case study, we were therefore able to eliminate certain false positives but no false negatives.

By definition, a code clone is always involving more than one fragment of source code. This leads to

Definition 6 ***(Clone Relation)*** *A clone relation is defined as an equivalence relation on code fragments. It holds between two code sections if and only if they are the same sequences. [KKI02]*

Definition 7 ***(Clone Pair)*** *A clone pair is a pair of code fragments between which the clone relation holds. [KKI02]*

In reality, code clones often cover more than two different pieces of source code. The following definition therefore introduces the clone class.

Definition 8 ***(Clone Class)*** *A clone class is a set of code fragments that have been created by copying and possibly modifying each other. This set is not ordered. The clone relation holds between any tuple of code fragments in the set.*

Even though there are very different approaches to identify code clones, most of them use a common vocabulary defined for the most part at the First International Workshop on Detection of Software Clones in October 2002, which was held in conjunction with ICSM '02[1] and SCAM '02[2]. These terms are essential to discuss the effectiveness of the different methods for detecting clones.

Definition 9 ***(Recall)*** *The recall is the ratio between the number of real clones among the reported candidates and the number of clones that are in the system.*

$$Recall(P,T) = \frac{RealClonesReported(P,T)}{Clones(P)}$$

where:
P = Input source code and
T = Clone detection tool

Definition 10 ***(Precision)*** *The precision is the ratio between the number of real clones among the reported candidates and the total number of reported candidates.*

$$Precision(P,T) = \frac{RealClonesReported(P,T)}{Candidates(P,T)}$$

where:
P = Input source code and
T = Clone detection tool

Recall is therefore a metric for the amount of discovered duplicates while precision measures the number of false positives. There is usually a trade-off between recall and precision.

Definition 11 ***(Clone Coverage)*** *Clone coverage is defined as the ratio between the length of cloned code fragments and the total lines of code in a file, class, or other source code entity.*

This definition of coverage is not entirely unambiguous.It is not specified if the total lines of code include lines that contain no information for the compiler (*e.g.*, blank lines or comments). It is also not clear how many source code entities are affected by a given clone class. In this thesis, comments and blank lines are not included in the total lines of code. Duplications are always considered to hold between exactly two entities – in this case those entities are C or C++ files.

[1] IEEE International Conference on Software Maintenance 2002 in Montréal, Canada.
[2] Second IEEE International Workshop on Source Code Analysis and Manipulation 2002 in Montréal, Canada.

2.2 Approaches to Code Clone Detection

Detecting duplicated source code is not trivial. Cloned code fragments often undergo small changes like renaming of certain variables or the introduction of additional code into one of these fragments. Therefore simple string-matching is usually not powerful enough to be of any use. In the following sections we describe several more sophisticated approaches.

During the last years several approaches to detect code clones have been proposed. Generally speaking, these methodologies can be categorized into four different solutions:

- detection based on lexical analysis [Bak92, RD98, KKI02],
- on source code metrics [MLM96],
- on an abstract syntax tree representation of the system [BYM+98],
- on isomorphic program dependence graphs [Kri01].

2.2.1 Dup

Baker proposed her approach to clone detection in 1992 [Bak92, Bak93]. This makes it the earliest clone searching approach considered in this paper. Later approaches often share many similarities with her tool. It can therefore be considered the direct ancestor of most later approaches.

Her method has been implemented with a tool called **Dup**. Clones are most often introduced by "copy–paste–change" programming. Therefore, the subsequent code duplications are often line-by-line copies. Consequently Dup uses a line-based approach when searching for clones. White spaces and comments are eliminated and the resulting normalized lines are compared. Normalized lines are considered to be a clone if the sequence of characters is the same in both. Baker does not consider the semantics of the program being checked for clones in her approach.

The tool also allows the detection of imperfect matches by parameterizing the query. Allowed parameters are for example the names of variables, constants, or methods. A one-to-one relation between parameters is still required.

Dup creates an output of "longest match" strings, meaning that the longest possible fragments of cloned source code are returned. To avoid getting too short – and therefore uninteresting – duplications, a minimal length can be specified. The tool can visualize the results in a dot plot graph to simplify their comprehension.

2.2.2 Duploc

Duploc was first presented in 1998 [RD98, DRD99] with the intention of offering a clone detection tool supporting both visual exploration and automatic detection. Detection and visualization was often divided into several tools in earlier approaches. Duploc is available under the GNU General Public License[3].

Duploc searches for clones by using a line based approach, which makes it possible to adapt to new programming languages[4]. Thus, Duploc can for practical reasons be considered as language-independent.

Code clone detection works in three steps. First, each source code fragment is normalized by removing comments and white spaces. One line has been chosen as a suitable size because most important code duplications include several lines. It also allows to keep the algorithm as simple

[3] http://www.iam.unibe.ch/~rieger/duploc/

[4] Duploc so far supports C, C++, Cobol, Eiffel, Java, Pascal, Assembler, Perl, Python, and Smalltalk.

as possible. To further reduce the amount of irrelevant code clones found, common language-specific statements, such as `break;` or `int i;` in C – are eliminated.

In a next step, the normalized lines of source code are compared to each other by string matching. If the two lines in a pair are cloned, they are assigned a boolean `true`. These values are then stored in a matrix. The coordinates are determined by their line indices. The necessary comparisons lead to a complexity of $\Omega(n^2)$ where n is the number of compared files. In order to reduce this complexity and therefore increase the scalability of the tool, a further optimization step is introduced. After the initial normalization of the source code, the resulting strings are hashed into B buckets and only the possible tuples in one bucket are compared by string matching. Since identical lines are hashed into the same cluster, no false negatives are introduced. The complexity is reduced by the factor B.

The last step consists of the visualization of the results. As in Baker's Dup, the clone pairs are displayed in the form of a scatter plot directly derived from the matrix created in the last step. A boolean value of *true* will result in a dot in the appropriate place of the plot allowing the user to detect certain patterns. To cope with the possibility of some changed code within a clone pair (represented as a broken diagonal line in the scatter plot), a pattern matcher is run on the diagram that allows for a certain degree of change – *e.g.*, renamed variables.

2.2.3 CCFinder

CCFinder is currently considered as a state of the art clone detection tool. It was first described in [KKI02] and is still being actively developed. A new version is expected shortly after this thesis is finished[5].

The main motivation for the development of CCFinder was the need for a tool applicable to a million-line sized software system with affordable complexity of computation. Another requirement was a relatively small language dependent part making the tool adaptable to other languages.

The clone detection approach is based on the transformation of the source code to a sequence of tokens and the subsequent analysis of these tokens instead of the raw source code.

First, the source code is lexically analyzed. CCFinder breaks every line of code down into a sequence of tokens by applying language specific information about keywords and other conventions. The tokens are identifiers as well as keywords and symbols with a semantic meaning. All sequences are then concatenated. During this step, white spaces and comments are removed. These suppressed tokens however are kept in memory for the later formatting of the result.

Second, CCFinder transforms the generated token sequence according to language specific rules. For instance, the transformation rules for C++ are given in Table 2.1. These rules aim at a regularization of identifiers (rules RC1 and RC2) and an identification of structures (RC3 and RC4). Identifiers relating to types variables and constants are replaced with a symbolic token, making the detection of clones with altered naming possible.

Rule	Description
RC1 (Remove name space attribution)	**(Name1'::')+Name2** $\mapsto$ **Name2** Here, the operator+, a postfix operator of regular expressions, means a repetition of one or more times. in C++ source files, a name may belong to a name space or a class and can be spelled in full or in shorter form. The transformation is to neglect the attribution so they are considered equivalent in clone detection. Ex. `std::ios_base::hex` is transformed into `hex`.

[5] http://www.ccfinder.net

Rule	Description
RC2 (Remove template parameter)	**Name '<'ParameterList'>'** $\mapsto$ **Name** Here, ParameterList is a sequence of Name, Number, String, Operators, ',' and Expression. Expression is a sequence of tokens which starts with '(' and ends with a corresponding ')' and does not include ';'. Template arguments may be omitted because of a type estimation by the compiler or because of the scope of the template. The transformation copes with the case. Ex. `sort<int>` is transformed into `sort`.
RC3 (Remove initialization lists)	**'=' '{'InitializationList,'}'** $\mapsto$ **'=' '{'UniqueIdentifier'}'** Here, InitializationList is a sequence of Name, Number, String, Operators, ',', '('. ')', '{' and '}'. UniqueIdentifier is a unique token which never appears in another place of a token sequence. Some tables (such as character code, color code and wave tables) include a continuation of a value and regular repeats of some values. The rule eliminates such large table initialization codes.
RC4 (Separate function definitions)	**Insert UniqueIdentifier at each end of the top-level definitions and declarations.** This rule prevents extraction of clone pairs of the code portions that begin in the middle of a function and end in the middle of another function definition.
RC5 (Remove accessibility keywords)	**AccessibilityKeyword** $\mapsto \Phi$ Here, Φ is a null sequence. Ex. `protected: void foo();` is transformed to `void foo();`.
RC6 (Convert to compound block)	**Each single statement after if(), do, else, for() and while() is transformed to a compound block.** Ex. `if (a==11) b=1;` transforms to `if (a==1) {b==2;}`.

Table 2.1: CCFinder transformation rules for C++. [KKI02]

After these transformations, the actual match detection is carried out. Clone pairs are detected from all the subsequences of a minimum length defined by the user. The location of clone pairs is stored by their location in the token sequence.

Finally, the location of clones is converted back to line and column numbers of the original input source code.

In [UKKI02] a graphical user interface called **Gemini** is presented which effectually adds a fifth step to the clone detection process. The program offers further possibilities supporting software maintenance. Gemini offers visualization, metrics, and source code browsing capabilities.

2.2.4 CloneDR™

CloneDR is the only commercial clone detection tool that is considered in this thesis[6] and forms part of a larger software maintenance platform platform called DMS. The approach has first been described by Baxter *et al.* [BYM+98].

Basically, CloneDR first parses the input source code and generates an abstract syntax tree. After that, three consecutive algorithms are applied to detect clones with increasing granularity.

[6]The trademark is held by Semantic Designs, Inc. (http://www.semdesigns.com).

The first detection algorithm is concerned with finding duplicated sub-trees. In order to find matching candidates even if they contain some variation, the sub-trees are not checked for equality but rather for similarity. The user can define the degree of similarity. A problem that arises is the scalability of this approach. It is not efficient to compare every sub-tree with every other to detect clones. Therefore, the sub-trees are first categorized using a hash function that orders similar sub-trees into the same buckets. Only the candidates contained in the same bucket now have to be compared with each other, reducing the computational complexity to the order of $\Omega(N)$, with N being the number of nodes in the AST. To solve the problem that n sub–trees containing small variations would end up in different buckets because of a good hash function, an artificially bad function is used. This function has to take into account that most clones are created by copying and pasting the relevant code fragments followed by some small local changes. Therefore, the hash function is designed to ignore small sub-trees which represent these modifications. The similarity of two trees is determined with the following formula:

$$Similarity = \frac{2 \cdot S}{2 \cdot S + L + R}$$

where:
S = number of shared nodes
L = number of different nodes in sub-tree 1
R = number of different nodes in sub-tree 2

This calculated similarity is compared with the user-specified threshold to determine the hash bucket for each sub-tree. This first algorithm is purely syntax-driven.

The second detection algorithm is designed to detect clones involving certain recurring fragments like sequences of declarations and statements. In an AST, such sequences show up as heavily left- or right-leaning trees with a common sequencing operator at their root. The algorithm used returns only the longest sequence and ignores the rest. This reduces the number of clone candidates but increases their average size.

The third algorithm attempts to generalize the clones detected in the first step and filtered in the second. This generalization is achieved by examining the parent nodes of detected sequence clones. If the parent nodes are similar enough, they are considered to be a generalized clone of the following sub-sequences and the former sequences can be replaced by their parents.

Compared to the clone detection tools based on lexical analysis described earlier, this approach sets higher requirements to the parser which constructs the initial AST. An adaption to different programming languages is therefore difficult to achieve.

CloneDR allows the visualization of the results as well as an automatic replacement of the clones found in a system. An evaluation copy with limited functionality is available for download[7].

2.2.5 Duplix

Duplix was introduced by Jens Krinke and uses an approach unlike the tools mentioned before [Kri01]. The goal of this approach was to create a clone detection tool that does not suffer from the usual trade–off between recall and precision. This goal has however not been achieved as later studies show [Bel02].

The identification of duplicated fragments of source code is based on finding similar subgraphs in a fine–grained program dependence graph (PDG). The approach therefore considers not only the syntactic structure of the input, but also the data flow. Basically, the graph's vertices represent assignment statements and control predicates in the input code. The edges show

[7] http://www.semdesigns.com/Products/Clone/

dependences between different components of the program and describe either a data or control dependency. A data dependence edge from vertex v_1 to v_2 signifies that component v_1 assigns a value to a variable used by v_2. A control dependency means that if the component represented by v_1 is evaluated to a certain value, v_2 is executed. This value is modeled as an attribute of the edge. Krinke extends this traditional definition of a PDG by adding elements of ASTs. AST vertices are mapped nearly one-to-one to their PDG counterparts. Furthermore, they get three attributes: class (statement, expression, procedure call etc.), operator (binary expression, constant etc.) and value (the exact operator). The model is further enhanced by special dependence edges between various classes of vertices.

Clones are now detected by finding similar sub-graphs of the PDGs generated from the input. Two paths are considered to be similar if there exists for every path $v_0, e_1, v_1, e_2, \ldots, e_n, v_n$ in one graph a path $v'_0, e'_1, v'_1, e'_2, \ldots, e'_n, v'_n$ in the other graph and the attributes of the vertices and edges are identical. To find these possible similar sub-graphs would require a complexity of $\Omega(V)$ where V is the number of vertices. In an attempt to reduce computational costs, only a subset of all vertices are considered as starting points. To select these vertices, Krinke uses predicate vertices in order to find clones regardless of their placement in specific functions. The constructed sub-graphs are then weighted (the criterion is the number of data dependence edges in the graph).

The prototype implementation of this approach currently supports only ANSI C. Due to the complexity of the data flow analysis, there is also a limit to the size of the input code[8]. Originally the tool has only been tested with up to 25,000 lines of code – in later experiments it processed input sizes of up to 115,000 lines.

2.2.6 CLAN

Merlo *et al.* propose an approach to clone detection based on source code metrics [MLM96, LPM+97]. The initial goal of the project was the identification of student assignments which "were sharing too much of their software projects".

The approach uses a set of defined metrics to find clones in programs written in procedural programming languages. To get the required metrics, a source code analyzer tool set – Datrix™– is used. As a first abstraction Datrix creates an AST representation of the input source code. This tree is then converted into an Intermediate Representation Language (IRL) for further processing. This two-step transformation is necessary for the support of several different input languages. The IRL contains information about architecture, static data type, control flow, and data flow of the input. Metrics generated on the last two categories of information are of particular interest for this clone detection approach.

The scope of detected clones in this approach are only complete functions with equivalent or similar functionality. The maximum amount of cloned functions in a system is therefore $\frac{n \cdot (n-1)}{2}$ where n is the number of functions in the input code.

The approach considers four points of comparison when detecting clones:

1. Name,
2. Layout,
3. Expressions, and
4. Control flow.

First the names of the functions tested for clones are compared. Two functions bearing the same name – even across component borders – are considered to be likely candidates for code duplication. This step does not yet include the use of any further source code metrics.

[8] In fact, the limit is on the number of similar sub-graphs, which is however unknown before the tool is run.

Next, the layout – here defined as the visual organization of the source code – of the two functions is compared. Various metrics are applied to comments, lines of code, and variable names.

The third step is concerned with the expressions within the function. Their number, nature, and complexity are considered and mapped to five different metrics.

In a final step, the control flows of two functions are compared. First the number of nodes and arcs in the different graphs are compared. To this comparison information about loops and decisions in the program is added. Eleven distinct metrics measure the control flow.

A total of 21 different metrics are available in the steps described above. The user specifies which combination of these metrics is used. The selected metrics are then compared for any two functions. Functions are considered equal in a certain point of comparison if the respective values of the metrics are equal. They are similar if the difference between the two values is smaller than a predefined delta value. The totality of all metrics together serves to determine the level of code duplication between two functions. This approach defines an ordinal scale of eight levels of cloning according to the four steps during clone detection:

1. ExactCopy,
2. DistinctName,
3. SimilarLayout,
4. DistinctLayout,
5. SimilarExpression,
6. DistinctExpression,
7. SimilarControlFlow, and
8. DistinctControlFlow.

In [MLM96] the application to two case studies is described. These consist of between 6,645 and 7,146 functions (485,433 and 506,823 lines of code). A definition of the metrics used in clone detection can be found in the same paper.

2.3 Code Clones and Evolution

So far, less work has been done in the research of code duplications in relation to the evolution of the corresponding system. The general understanding is that code duplications represent a major bad smell [FBB+99] and often lead to problems with the maintainability of a software system (and therefore to higher maintenance costs). For that reason, they should be removed from the system by appropriate refactoring techniques as described by Fowler.

In [KSNM05] Kim *et al.* examine the evolution of code clones over several versions of a software system. A possible connection between duplications and subsequent couplings between several files is however not discussed. They introduce a model to classify clone genealogies which is similar – but more comprehensive – than the attempt of clustering duplicates used later in this paper. They postulate that in many cases clones are not inherently bad and should therefore be left in the system. They argue that a number of duplications only remain in a system for a short time and are not worth the trouble of a major refactoring effort. On the other hand, older clones are often so well established that they are not refactorable anymore. If this train of thought is followed to its end, one could conclude that clones should never be refactored at all as they are always either too young or too old to be suitable candidates.

This thesis intends to develop a way of discerning between code clones whose removal could benefit the system, and duplications that can safely be left alone. It is not intended to provide a general answer if clones should be refactored at all.

2.4 Visualization of Code Clones

There are two main approaches to the visualization of code duplications which are being used by the majority of the clone detection tools discussed before. Most detection tools – if they offer any visualization at all – use both methods simultaneously. The two visualization techniques are source code highlighting and dot plots[9].

2.4.1 Source Code Highlighting

Source code highlighting is a very simple visualization technique. Nevertheless it conveys essential information that for example a dot plot cannot display.

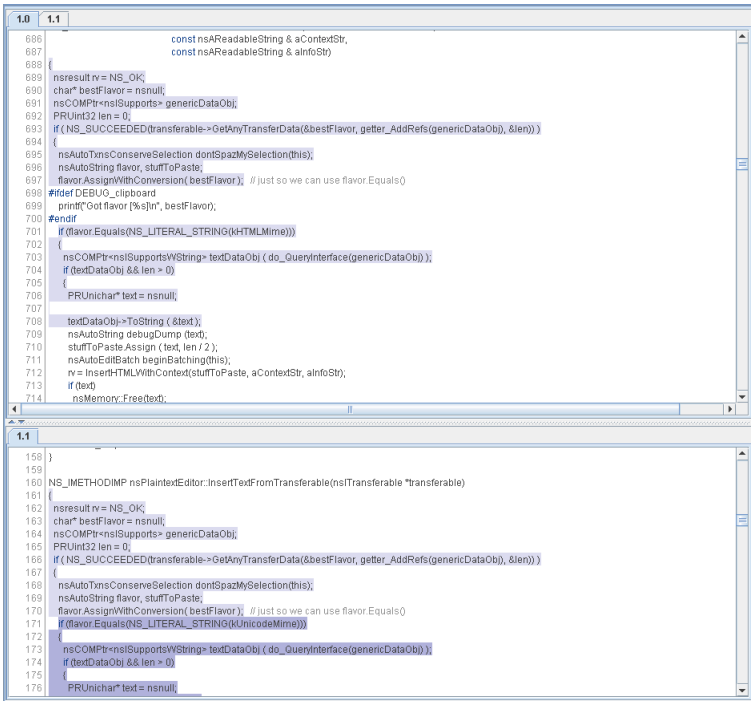


Figure 2.1: Example for visualization by source code highlighting in Gemini.

The basic principle is that the source code fragments containing clones are displayed at the same time. The cloned fragments are highlighted by the use of a different background color. Figure 2.1 shows an example of an output created by CCFinder's graphical user interface Gemini. Different hues of a color are used to indicate source code fragments that are cloned more than

[9]The term "scatter plot" is used as a synonym.

once. In the example, the darker colored piece of code in the file numbered 1.1 is cloned up to three times in the other file.

Visualization by source code highlighting is very successful in showing clearly defined fragments of clone candidates. It can be used for the classification of the candidates reported by a detection tool into real clones and false positives. If the fragment is a true duplication, the highlighted code can be further examined to determine the extent of cloning and possible changes (*e.g.,* renaming of variables or methods) made during the cloning process.

Source code highlighting has its limitations. It is not a visualization useful for giving an overview of all clones in a system. Its scope is usually limited to two files a time.

2.4.2 Dot Plots

The second major visualization technique for code clones is the dot plot[10]. This method has been used in nearly all detection tools starting with Baker's Dup [Bak92] and is since then the standard visualization in this field.

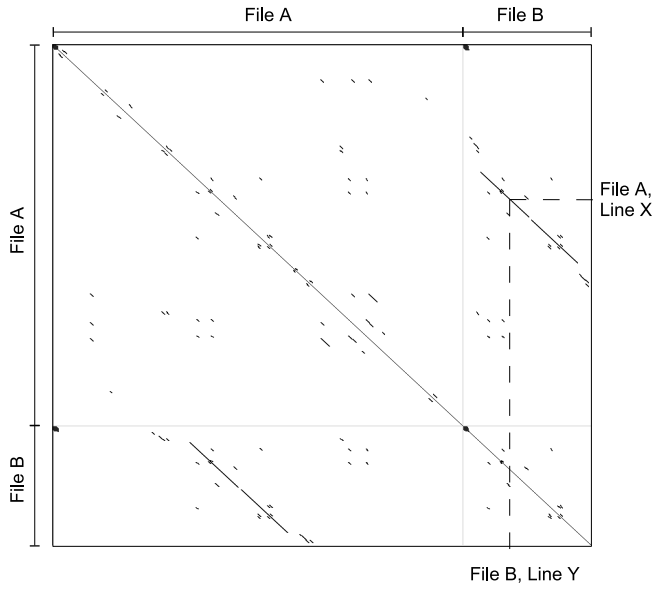


Figure 2.2: Example for visualization by dot plot in Gemini.

The basic principle of the dot plot visualization is that of a matrix. The lines of code[11] of the files that are compared form the two dimensions of the matrix. If lines X of file A and Y in file B contain a match, the dot plot contains a dot at coordinate $(A.X; B.Y)$ as indicated in Figure 2.2. The origin of the coordinate system is usually in the upper left corner (except in Baker's Dup where it is located in the lower left corner).

The dot plot visualization is very useful when the goal is to get an overview of the code clone situation in more than two files. This technique is also applicable for very large inputs. During our case study of the Mozilla web browser, it has been used to gain an overview of complete releases of Mozilla with close to 4 million lines of code in several thousand files.

[10]The term scatter plot is used as a synonym for dot plot in this thesis.
[11]Criteria other than LOC (*e.g.,* tokens or statements) can also be used. The standard in the examined tools is LOC.

The strength of the dot plot visualization lies in the occurrence of recurring and recognizable patterns signifying certain qualities of a detected clone candidate. These patterns are described in [RD98].

The simplest pattern can be observed in Figure 2.2. Longer cloned fragments are displayed as diagonal lines in the plot. When a file is compared to itself, it is by definition a perfect clone of itself. Therefore a diagonal line stretches from the upper left to the lower right corner. This, for example can be seen in Figure 2.2 where file A is compared to itself in the upper left square. The same applies to file B in the lower right corner.

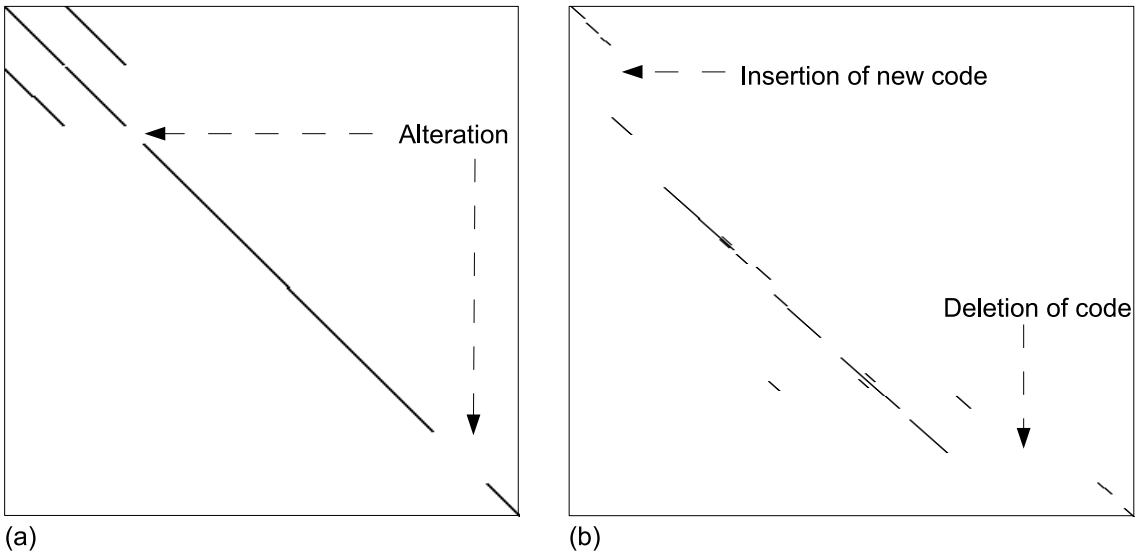


Figure 2.3: Dot plots showing clones with changes on certain lines (a) and with inserted and deleted lines (b).

Disconnected diagonal lines as shown in Figure 2.3(a) stand for cloned code fragments with alterations. As these changes are not duplicated code anymore, they show up as a hole in the diagonal lines indicating code clones in the dot plot.

Cloned sequences to which new code has been added or from which code has been deleted can be detected by diagonal lines in the dot plot that are shifted in places. Insertions lead to more lines of code – the diagonal line is therefore shifted downwards. Likewise, the deletion of source code lines shifts the diagonal to the right as indicated in Figure 2.3(a).

The final pattern commonly found in dot plots concerns repetitive code, *e.g.*, in `switch`–statements. These clones occur several times at relatively small intervals and form rectangular structures in the dot plot. Periodic occurrence of code can be limited to single statements or comprise whole similar methods of many lines or even complete files. An example for this pattern is provided in Figure 2.4. In this case the cloned fragment is a similar method of about 12 lines of code that is repeated in most of the files of a module.

All examples used in this section are actual results obtained from the clone detection runs in the case study. All described patterns are very commonly found in the code of the Mozilla system.

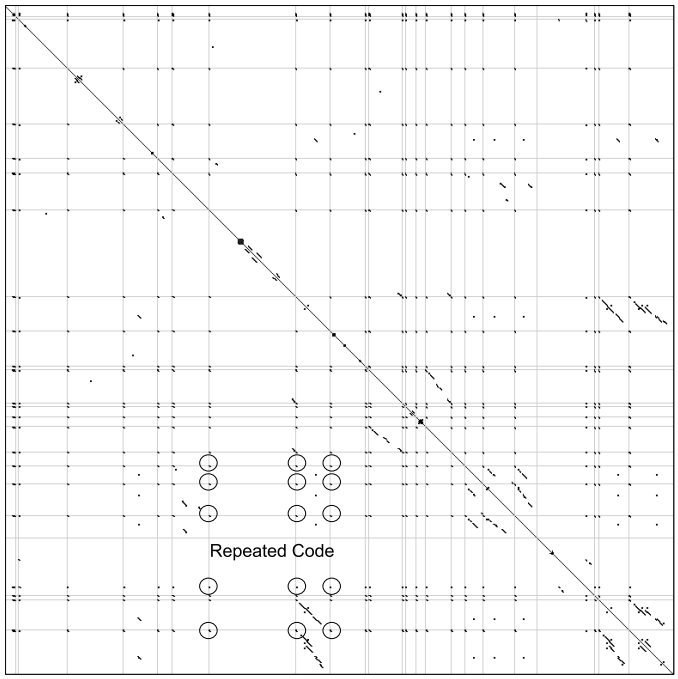


Figure 2.4: Example dot plot showing code repeated periodically in several files of a module.

Chapter 3

Evaluation of Clone Detection Tools

In Chapter 2 we have described several clone detection tools in theory. In this chapter we evaluate three of them further in order to determine the tool best suited to the Mozilla case study. We start by giving a brief summary of all seven approaches, followed by an explanation of how the candidate tools are further evaluated. Finally the results of this evaluation are presented.

3.1 Introduction

3.1.1 Recapitulation

In this section we will briefly compare some major attributes of the code clone detection approaches we have presented previously. More comprehensive comparisons – also including tools not treated in this paper can be found in [BB02] and [Bel02].

There is a consensus in literature that no single code clone detection tool is the software engineer's well known "silver bullet". Each tool has it's weaknesses and most have particular strengths. Previous surveys indicate that there are clones that cannot be detected by all of the tools. Some duplications are reported by only one approach.

We will consider several attributes with a certain importance to the applications of the tools in our case study. Table 3.1.1 gives a brief summary of the different approaches. Especially the input size should not be read as an absolute value but rather as an order of magnitude of experiments documented in the literature – there might be undocumented uses with different input sizes.

Tool	Approach	Background	Granularity	Input	Recall	Precision
Dup	Pattern matching	Academic	Line	700k	High	Low
Duploc	Pattern matching	Academic	Line	150k[1]	High	Low
CCFinder	Pattern matching	Academic	Token	2,6m	High	Low
CloneDR	Abstract syntax tree	Commercial	Line	>400k[2]	V. low	V. High
Duplix	PDG	Academic	Statement	115K	Low	Low
CLAN	Source code metrics	Academic	Function	>14,8m	Low	V. High

Table 3.1: Comparison between clone detection approaches.

3.1.2 Methodology for the Evaluation

During the initial definition of the research goals for this thesis, a pre-selection of three code clone detection tools has been made. These candidates were mostly chosen for their availability. They included Duploc (the only tool available for download under the GNU General Public License), CloneDR (free evaluation copy available) and CCFinder, which has been been obtained with permission of the authors at the Osaka University.

The three candidate tools were evaluated according to several criteria which were considered important for the applicability of the tools in the case study. Most of these criteria are not directly measurable or even depend heavily on the subjective perception of the user. The tools have been evaluated using a sample input code of about 12.000 lines of C++ source code[3]. The results and usability were then judged by the following criteria:

Language Support: The parts of the Mozilla system selected for this case study are written in C and C++. It is therefore essential for any suitable code clone detection tool to be applicable to inputs written in these programming languages. A tool using only plain text comparison without any regard to the C/C++ syntax would not be of any use as it could only detect exact duplications.

Input Size: The complete Mozilla system consists of several million lines of source code (cf. Table 1.2.2). As it was intended to compare not only one release of a module with itself but several releases at the same time, a suitable clone detection tool must be able to process at least several hundred thousand to a few million lines of input source code. If this threshold is fulfilled, the future input can be scaled accordingly. Capabilities exceeding these minimal requirements are only of secondary importance.

Comparison of the Candidates: The three clone detection tools evaluated use different approaches and detection algorithms. The detected clone candidates must be compared to see if the programs detect roughly the same sets of code duplications. If this is the case, the use of only one of these tools for the subsequent case study is sufficient. If on the other hand the resulting sets are largely disjoint, more than one of the programs should be used and the results combined to get a more complete overview of the code clones present in the case study.

User Interface: This criterion is largely based on the subjective impression of the tester. Desirable are a simple way of defining input source code both interactively and in a batch file, a clear visualization of the clone candidates with the possibility of source code browsing and the ability to export and print certain aspects of the results. Also of importance is a user interface that links the source code to the visualization. The code must be accessible so that the user can decide if a candidate is a clone or a false positive. As there are no generally accepted metrics for the assessment of user interfaces, this criterion reflects only the personal preferences of the user.

Output: The ideal clone detection tool presents the detected clone candidates in an understandable visualization as well as in a log file for later reference. The visualization is important for the user to gain an initial overview with the possibility of quickly detecting points of interest for further more in-depth study. An automatically created log file is necessary for a later automated

[1]Duploc is said to have been used on inputs of more than 1m lines of code. However, this could not be verified in the literature.

[2]It has been verified in e-mail contact with Semantic Designs, Inc. that CloneDR is able to handle input of well over 1m lines of code.

[3]The available evaluation copy of CloneDR could only process Java code, which was therefore used for this tool.

approach to process large volumes of clone data. A log file should also be formatted in a clearly and possibly well documented way to facilitate parsing.

Recall: A high recall is desirable in order to obtain as many clone candidates – and therefore data for further processing – as possible. The evaluation of the value for recall is based on literature, most prominently in [Bel02].

Precision: A high recall is desirable in order to be able to reduce the necessity to inspect all candidates manually prior to further processing the data. The evaluation of the value for precision is based on the findings by Bellon [Bel02].

3.2 Evaluation

3.2.1 CloneDR

We only had the evaluation version of CloneDR available which has a very limited functionality. It can only process either Java or COBOL source code. Full clone detection functionality is also only available for input of less than 1.000 lines of code. If this threshold is exceeded, only the first 10 clone pairs with a length of less than 50 lines are returned.

The full license of CloneDR also provides a facility to automatically remove detected clones. For our case study, this functionality has been considered irrelevant.

Language Support: The full version we were offered would accept exclusively C/C++ input code, so this lack in the evaluation copy was not marked as a point against CloneDR.

Input Size: It has been verified through e-mail contact with the manufacturer of the tool that CloneDR is able to handle input sizes in excess of 1 million lines of source code. This is enough to compare seven releases of any module within the case study at the same time.

Comparison of the Candidates: Because of the limitations on input languages mentioned above and the subsequent exploration using a different input program of similar size, the detected clones could not be compared with the other two detection tools under consideration.

User Interface: CloneDR in its evaluation version is a pure command line tool that does not come with a graphical user interface. The parameters and inputs are provided in a specially formatted file stating the options used in the detection as well as a list of every file to be tested. It is not entirely clear if the full version offered to us would include a graphical user interface or if CloneDR would have to be embedded into Semantic Design's Design Maintenance System™(DMS) [BPM04] to provide this.

Output: Output is generally written to the terminal. To make it available in a file, a piping mechanism provided by the operating system is necessarily used. CloneDR's output provide a comprehensive log of the whole detection process as well as some statistics about the detected clone candidates. The main part of the output file is taken up with the actual candidates, described in a verbose format. This has the advantage of giving all necessary information on one glance. Included are for example the exact location of the clones, the source code and the necessary parameters replaced for near-miss detection. The output is formatted for human use – it's

verbosity and lack of clearly defined separators would make it very difficult to parse for further processing the output.

Recall: CloneDR is generally attributed with a low value for recall, meaning that it does not find all clones present in a given system.

Precision: The precision of the candidates reported is generally high. The clones reported in our test input were for the most part justified. There were some cases where candidates had to be rated as false positives. Other detected cloned code fragments could not possibly be refactored (*e.g.,* exception- and event-handling).

3.2.2 Duploc

Duploc is written in Smalltalk and needs VisualWorks 3.0 to run. The source code is available for download[4] and needs to be recompiled and integrated into a VisualWorks workspace. We used VisualWorks versions 3.0 as well as 7.3 in our trials.

The tool is the only one known to be freely available under the GNU General Public License.

Language Support: Duploc allows a very wide range of programming languages as input. C and C++ are among them. Duploc also allows the comparison of files written in different programming languages.

Input Size: Duploc is supposed to have handled inputs of about 1 million lines of code. The largest number that could be verified in the literature is around 150,000 LOC, which is still enough to compare seven releases of most of the modules comprising the case study. The lower boundary of 150,000 LOC is however insufficient for the larger modules.

Comparison of the Candidates: As was to be expected from its reported recall, Duploc detected a substantial quantity of code clone candidates. Because of the approach used for clone detection, the tool reported fewer candidates than CCFinder which is described below.

User Interface: Duploc features either an interactive mode with a graphical user interface or a batch mode. In interactive mode, the results of the detection cycle are displayed as a series of scatter plots. The source code can be browsed by selecting the relevant clones in the graph. In batch mode only a log file detailing the results is generated – the candidates are not shown in graphic form and no source code browsing is possible. The log file cannot be loaded and processed to show the graph.

A problem that has surfaced frequently are Smalltalk error messages mostly concerning unhandled exceptions. Examples for their appearance were during source code browsing, deleting an old file list or when the cursor left a certain area. Whether these errors occurred because of a faulty installation or because of errors in the source code remains unknown to the user unfamiliar with Smalltalk.

[4] http://www.iam.unibe.ch/~rieger/duploc/

Output: The output provided is a dot plot showing the clone candidates graphically. This representation allows a visual recognition of patterns characteristic to certain types of clones. The graph can be saved in Postscript format. No log file is generated in interactive mode.

In batch mode, a log file is generated including information about the clone sequences found in any two files. The output generated can be parsed without much difficulty for further use outside of Duploc.

Recall: Duploc is attributed with a high recall due to the number of candidates returned.

Precision: The precision of this tool is in the literature usually considered to be low. However, the samples we have checked out of the set of clones detected in our test code were for the most part real clones (mostly of the unaltered "copy-paste" type). We consider Duploc's precision to be sufficient for our purposes.

3.2.3 CCFinder

CCFinder was obtained with the help and permission of the developers at the Osaka University. The program is written in Java. Initially, CCFinder is a command line only program.

The development team does also provide a maintenance support environment called Gemini into which CCFinder is integrated. It adds visualization capability as well as access to further information on the input code. Whenever we talk of CCFinder, we mean a combination of CCFinder and Gemini.

Language Support: CCFinder supports a wide variety of input languages including C and C++ which we need for our case study.[5]

Input Size: The highest known reported size of input used with CCFinder is around 2.6 million lines of code [KKI02]. This is a sufficiently high number to check seven releases of any single module of the Mozilla case study. It is also likely to be possible to check one complete release of Mozilla at once.

Comparison of the Candidates: Depending on the parameters set for a detection run, CCFinder does detect more candidates than Duploc giving this tool a higher recall ratio. The clones covered by Duploc are generally also found with CCFinder.

User Interface: Gemini offers an intuitive user interface presenting the candidates in scatter plot form. Additionally, several metrics are available as well as source code browsing with highlighting of the clones. Gemini provides sophisticated filtering and sorting mechanisms allowing the user to specify exactly what clones are of interest to him. Metrics, scatter plot and source code are interconnected so that the focus is always on the selected clones regardless of the view.

All parameters necessary for CCFinder can be changed directly in the graphical user interface, allowing the user to tailor the tool to his needs.

One disadvantage of the interface is that there is no possibility of printing any part of the displayed information directly out of Gemini.

[5] The tool is also able to process plain text so that in theory any given programming language can at least be examined for exact duplication.

Output: The clone candidates are displayed in scatter plot format along with additional metrics and meta-information in Gemini. One problem is that the scatter plot or clone metrics cannot be printed other than by taking a screenshot. This method is feasible but cumbersome and inadequate insofar as all information concerning the context is lost.

An additional log file can be created. This file is generated by CCFinder without any involvement by Gemini. This means that the additional information – such as metrics – which is calculated by Gemini cannot be saved. However, the log files can be loaded in Gemini to access this data.

The log file is precisely formated making it possible to parse it for further use outside the CCFinder-Gemini package.

Recall: Like Duploc, CCFinder has a high recall ratio. Depending on the parameters used during the detection run, recall can be even higher than in the tool evaluated above.

Precision: Due to the amount of candidates, CCFinder also reports a number of clones which have to be classified as false positives. This leads to a relatively low precision.

This behavior seems to be caused by the very fine-grained parametrization during the clone detection process. The candidates often have to be inspected manually to determine if they are genuine clones or not.

3.3 Conclusion

The evaluation of the three tools was used to decide which approach should chiefly be used for the Mozilla case study.

All three tools comply with the language support criterion, even though CloneDR would have to be purchased for C/C++ support.

The three criteria including precision, recall, and the comparison of the candidates are all interrelated. A high recall was favored over high precision because in any case the reported candidates would have to be evaluated manually for patterns significant to our research goal. Therefore, false positives could be eliminated later on. A higher recall would deliver more interesting candidates. CCFinder and Duploc are pretty much on the same level in these criteria with slight advantages for CCFinder.

CCFinder also has the best and most intuitive user interface among the evaluated tools. The additional clone metrics are a further advantage for this project. Duploc's interface needs more time to get used to, but is also a powerful instrument. The problem with inexplicable error message persists but does in the most cases not prevent the user from reaching his goal. The evaluation copy of CloneDR could not compete with either as far as the user interface is concerned.

While CloneDR produces the most comprehensive log file, the file generated by CCFinder is easier to parse for a prototype implementation later in the project. The clone metrics calculated by Gemini are not represented in the output log while CloneDR has a reduced set of metrics available.

After this evaluation, it was decided to use CCFinder and CloneDR as primary research tools with Duploc as backup. This however proved to be impossible after the estimated costs to purchase CloneDR were taken into account. It was therefore decided to go on with the other two tools. CCFinder was to be used first and Duploc was to help in the selection of candidates suitable for further manual inspection. It was hoped that a combination of the two tools would significantly reduce the time spent on eliminating false positives.

Chapter 4

Methodology

This chapter introduces a methodology with the goal of solving the research question posed in Chapter 1. The application of this methodology to the case study will be treated in a separate chapter.

4.1 Introduction

In this thesis we address the question if there is a connection between the presence of code clones in a software system and the occurrence of change couplings during the evolution over several subsequent releases. As this question has not yet been explicitly treated, a new methodology for finding a solution is introduced which is subsequently applied to the case study.

The approach to answering the research question can be broken down into several steps. Most of the later phases are based on further processing data obtained in earlier steps. The basic framework of the approach consists of:

1. Evaluation of code clone detection tools,
2. Application of clone detection tools,
3. Identification of suitable clone candidates,
4. Extraction of change coupling information from the RHDB,
5. Correlation of clone data with change couplings,
6. Definition of a metric to describe the impact of code clones, and
7. Implementation of a prototype tool exploiting a connection.

An in depth description of the theoretical aspects of this framework is given in this chapter and its application to the Mozilla case study is treated in the next chapter. The theory behind some of these steps is comparatively trivial and will only be described briefly.

The Mozilla project uses CVS for version control. All information about change couplings have been extracted from this repository. This process has been described in previous work [FPG03].

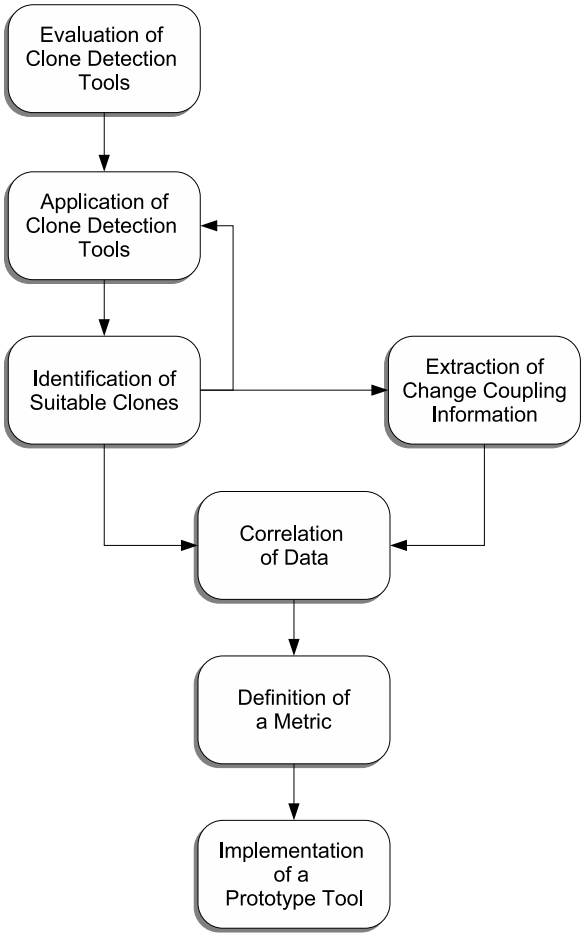


Figure 4.1: Overview of the framework for this thesis.

4.2 Framework

4.2.1 Evaluation of Code Clone Detection Tools

The selection of the code clone detection tool most suitable for the task only nominally forms part of the framework. Once a user has decided on his preferred method, this step can be skipped in later applications of this framework.

The basics for this evaluation are given in Chapter 2 and the process of selecting the most suitable tool with respect to our case study Mozilla is described in Chapter 3.

4.2.2 Application of Clone Detection Tools

The selected clone detection method must then be applied to the system being examined. This phase can be broken down into three sub–steps such as:

1. Provision of an environment,
2. Selection of input, and
3. Clone detection runs.

Provision of Environment

Considerable care must be taken to select a test environment suitable to the task. Several of the available clone detection tools have special requirements due to the way they were implemented. Often external runtime environments for either Java or Smalltalk must be provided while other tools are only available for certain operating systems.

Another major requirement which must be considered is memory consumption. Detecting and above all visualizing code duplications are resource–intensive tasks that often require more random access memory than standard workstations are equipped with.

The selection of a suitable test environment depends mostly on the tool selected in step 1 of this framework and the requirements are usually mentioned – although often not in their entirety – in the appropriate literature.

Selection of Input

There are two aspects to the selection of source code suitable as input for clone detection tools. The first is concerned with the examined software system and the second with the clone detection tool itself.

Industrial–sized software systems are not always written in only one programming language. Moreover, a certain percentage of the files in a project does not necessarily contain source code suitable for inspection. Software systems also include binaries, image files and a multitude of other formats.

Clone detection tools usually have a certain limit beyond which the display of results becomes either impossible due to constraints of the program or unmanageable by the user because of the sheer volume of data presented.

This sub–step must address both of these problems. Once it has been decided what the extent of the sample is and on what type of files the focus of the clone detection run is, the entirety of the examined system's file must be filtered accordingly. The files must then be formatted in a way that the clone detection program can use them as input. At least this usually means the creation of a meta file containing a list of the files to be compared. It can also include preprocessing the files directly prior to any detection run.

Since the focus in these examinations is always on several releases of the same system, this increased complexity does also have to be taken into account. The different releases must be compared to each other which increases the size of the compared code considerably. It will also be necessary for the user to be able to distinguish between the different releases after the result has been obtained. Most detection tools provide a facility for grouping the input and output. The user has to exploit this possibility in a way to make the distinct releases clear.

Clone Detection Runs

If the first two sub–steps have been completed successfully, the application of the clone detection tool is straight–forward. The user has only to decide on which options are used for the detection. Usually this includes at least the definition of the minimum length of a code clone but some detection programs offer more sophisticated means of configuring the tool to the needs of a certain case study.

The clone detection runs must produce an output that is suitable for further study. To achieve this goal, a multi–step approach to clone detection is used, which leads to a loop back from phase 3 (identification of suitable clone candidates) to phase 2 with ever finer granularity of results.

At first the whole system (or a part of it as large as possible) is checked for clones at a time. It is neither necessary to check all releases at the same time nor does the defined length of a code clone have to be minimal yet. However all releases must be checked and the visualizations are then compared to identify areas of interest for further study.

After that, the set of files used as input in each subsequent detection run is narrowed until at last only files containing instances of a certain clone class are compared. As the number of input files gets smaller, the minimal length of detected clones can be reduced accordingly to produce a finer grained detection result. In these later repetitions of clone detection runs, it is essential that all releases of the files are checked at the same time in order to better detect changes between different releases. After every run, the results are examined as described in Section 4.2.3.

4.2.3 Identification of Suitable Clone Candidates

Not all clone candidates reported by a detection tool are equally useful in the study of the development of code duplications during the evolution of a software system.

This phase of the framework forms a loop together with the previous phase. The suitable candidates obtained from a repetition of phase 2 are examined ever more finely.

As the goal of this framework is to define the connection between duplicated code and change couplings between files, the only interesting clone pairs are those in which the cloned code fragments appear in two or more files instead of duplications within one file only. Furthermore, clones whose length varies or that do appear or disappear during the examined period are considered more interesting than duplications that remain stable. This selection criterion is based on the long–standing assumption that there is a significant relation between code clones and change couplings. If the amount of clones is reduced, this should lead to an observable decrease of change couplings. After an increase of code duplications, we expect a related rise in change couplings between two files. In order to express the necessary attributes of code duplications, a new measure must be defined.

Classification of Code Clones over Several Versions

To facilitate argumentation, we introduce a classification of evolving code clones for this thesis. Depending mostly on the clone coverage, duplicated code examined over more than one release can be clustered in five classes or types of clones.

In [KSNM05] a model of clone genealogy is introduced. Its classes are mostly based on a qualitative examination of the evolution of duplicated code. For our approach a mere quantitative approach is sufficient and the model used is therefore simpler. During this phase of research, it is unimportant if there are changes within a cloned code fragment. What counts is the fact that a duplicated segment is either stable, increasing or decreasing over time.

Absolute numbers are inadequate when comparing different files because their lengths are hardly ever truly comparable. The same applies to the length of cloned code fragments. Even

when only two files are compared to each other, it is not necessarily the case that both contain cloned sequences of the same length – file A might contain a cloned fragment c, while file B contains fragment c more than once (or even overlapping). Therefore this model of code clone classification relies on the clone coverage in every single file. This ratio is for two files A and B defined as

$$CloneCoverage_A(A,B) = \frac{ClonedLines(A,B)}{NCLOC(A)}$$

and

$$CloneCoverage_B(A,B) = \frac{ClonedLines(B,A)}{NCLOC(B)}$$

where $ClonedLines(X,Y)$ is the number of lines in file X that are clones of lines in file Y[1] and $NCLOC(X)$ is the number of lines of source code in file X not counting comments and blank lines. A cloned line is only counted once even if it forms part of more than one clone pair or is covered multiple times by overlapping clones. It is important that this definition is always applied to exactly two files at a time. When more than two files are compared, every pair of files out of this set must be compared separately. It is however not important if A and B are files. Any suitable entity of source code can be used instead.

Due to the definition of a code clone

$$CloneCoverage_A(A,A) = CloneCoverage_B(A,A) = 1$$

holds for every file A when it is compared to itself as any file is by definition an exact clone of itself.

To apply clone coverage to a set of evolving files, it is necessary to observe the values over several versions of the files. These comparisons allow the classification of a specific file into one of five types depending on the development of its clone coverage.

For the definition of the different types, it is not important which file is observed. For this reason we shortened $CloneCoverage_X(X,Y)$ to $CloneCoverage(X,Y)$.

Two files A and B can share more than one semantically distinct clone pair. The types can be used to classify every instance of a clone pair or clone class in two files on its own. In this thesis, $CloneCoverage(X,Y)$ is however always calculated for the entirety of all code clones shared by X and Y.

- **Type 0:**
  The relative length of the cloned fragments in question remains the same between versions i and $i+1$.[2]
  $$CloneCoverage(A,B)_i = CloneCoverage(A,B)_{i+1} \neq 0$$
- **Type 1:**
  A clone is newly introduced after version i. It is present in version $i+1$ but not in i.
  $$CloneCoverage(A,B)_i = 0$$
  and
  $$CloneCoverage(A,B)_{i+1} > 0$$

[1] It is not necessarily the case that all cloned lines are of the same importance at a given time. Clone coverage can also be used if only a specific subset of cloned lines shared by A and B is observed.

[2] All files are classified separately. It is therefore not necessary to distinguish between $CloneCoverage_A(A,B)$ and $CloneCoverage_B(A,B)$ in the context of the definition of types 0 to 4.

- **Type 2:**
  A clone present in version i is subsequently deleted and not contained anymore in version $i+1$.

$$CloneCoverage(A,B)_i > 0$$

and

$$CloneCoverage(A,B)_{i+1} = 0$$

- **Type 3:**
  The clone grows in importance after version i. In version $i+1$ it is larger in proportion to the non-commented lines of source code than in i. Either the code clone itself has become longer or the total lines of code in version $i+1$ have diminished while the duplicated fragment did not get smaller proportionally.

$$CloneCoverage(A,B)_i > 0$$

and

$$CloneCoverage(A,B)_{i+1} > 0$$

and

$$CloneCoverage(A,B)_i < CloneCoverage(A,B)_{i+1}$$

- **Type 4:**
  The clone's importance is lower in version $i+1$ than it was in version i. Either the duplicated sequence has become shorter or otherwise the total number of lines of code has grown while the cloned fragment was not expanded proportionally.

$$CloneCoverage(A,B)_i > 0$$

and

$$CloneCoverage(A,B)_{i+1} > 0$$

and

$$CloneCoverage(A,B)_i > CloneCoverage(A,B)_{i+1}$$

Types 1 to 4 show some change during their life cycle. Among them, those best suited for further investigation are type 1 and 2 duplications. It is expected that the couplings between files containing cloned fragments of each other show some sort of correlation between the changing code clones and their later couplings. If this assumption is true, for example two files into which a type 1 clone is introduced after version i are expected to share more couplings in subsequent versions. Type 0 clones are also of interest: according to the hypothesis, couplings caused by code clones are expected to be stable.

A certain clone is not restricted to belong to only one of these types over it's life time of several versions. It is for example possible for a duplication to be introduced after version i and growing in importance between versions $i+1$ and $i+2$ while it is eliminated again after version $i+3$. Thus it is of type 1 between i and $i+1$, of type 3 between $i+1$ and $i+2$, of type 0 from $i+2$ to $i+3$ and finally of type 2 between $i+3$ and $i+4$ – if only versions i and $i+4$ are compared, the clone would not be detected at all as it is irrelevant regarding these two versions.

Selection of Code Clones

After the first clone detection run comparing as much of the system as possible, the metric described above is not yet applicable due to the volume of obtained data. The initial selection must there be based on the visualization of the result. Points of interest that are promising for further examination are often visible as dense patterns in the dot plot. Other than by the classification of all detected candidates, this is the only possibility of obtaining finer grained input. As the intention of this first detection run was to get an overview over the system rather than the detection of all suitable clones in one step, this rather intuitive criterion can be justified. It does however have one major drawback: if the visualized output is not ordered in any way, the patterns may not be visible. But as the input can usually be defined in a way to guarantee such an order – usually either by different source code folders or by modules – this problem can be evaded.

After later detection runs the selection can be made with the help of the clone coverage metric. As in these detections more than one release of the input files is checked at the same time, the clones can be classified into the 5 types by comparing the dot plots of the different releases. The exact clone coverage value does not have to be known at this time.

There is one more difficulty that is addressed in the course of this phase. As has been mentioned in Chapter 2, the duplication candidates reported by the detection tools are not always real clones. Since no clone detection tool with perfect precision and a substantial recall has yet been developed, the user has to decide by manual inspection if a candidate is a real clone or a false positive. This problem currently precludes a complete automation of this phase of the process. The selection of the clones for further study has to be done by a human unless the user is willing to accept the possibility that false positives influence further calculations[3].

4.2.4 Extraction of Change Coupling Information

Change Couplings

Files or other source code artifacts in a software system can be coupled in a variety of ways. These dependencies affect the maintainability of the system. Syntactic dependencies can be detected by examining the source code and include characteristics such as method calls. In systems implemented according to best practices in software development, these dependencies are often found not only in the code but also in the documentation.

On the other hand, there are couplings which are not detectable by program analysis alone. These are so called logical dependencies. These interrelations are usually not documented and known only implicitly to the developers. As these couplings are not represented in the source code either so that other means of detection must be applied which usually rely on data obtained from version control systems [GHJ98, GJK03, ZWDZ04].

Change couplings are a special case of logical couplings and indicate that source code artifacts have been modified together in the past. The reasons for the occurrence of such coupling can be very different and can for example be based on dependencies explicitly designed into the system's architecture or on unintentional relations such as the introduction of code clones during the implementation phase of the system.

In this framework change couplings on the level of a single file rather than between modules or subsystems are examined.

[3] When a sufficiently large amount of input is used, meaningful results may still be obtained by a completely automated process.

Release History Database

The concept of the release history database (RHDB) was first described in [FPG03]. The database was using version and bug tracking data. Currently the RHDB contains data obtained from the Mozilla open source project which uses CVS[4] as version control system and Bugzilla[5] for the organization of bug reports. The data is stored in a MySQL database.

For the purpose of this framework, not all information stored in the RHDB is of equal importance. Currently[6] there are 91 tables used in the RHDB – only three of them are needed for the extraction of change coupling data. Every file in the CVS repository has a corresponding instance in table `cvsitem` where its attributes are replicated in the database. Every check–in of a file results in an entry in the CVS log file. Information about any of these modifications is stored in the `cvsitemlog` table of the RHDB. The final table necessary for the determination of change couplings is `cvsitemloggroup`. These groups are formed by files that have been checked in together during a time slot of 15 minutes. For any file modified during this interval one entry in the `cvsitemloggroup` table is added. Any of these groups has a unique value in field `cvsitemloggroup.groupidx` and can therefore be recognized as being coupled.

The RHDB contains much information that is not needed for the purposes of this framework and is still acquiring more tables and information.

Measuring the Density of Change Couplings

The number of change couplings is like the amount of code clones a metric that is only defined for a pair of files. To make the values comparable to each other, the absolute number of change couplings must be observed in relation to the number of times a file is checked into the version control system during the time interval in question.

The number of change couplings between a pair of files or similar entities of source code during a given interval is the same for each file. The number of check–ins during the same time can however vary, giving us a distinct ratio for each file. For two files A and B the ratios are defined as

$$CouplingCoverage_A(A, B, I) = \frac{ChangeCouplings(A, B, I)}{Checkins(A, I)}$$

and

$$CouplingCoverage_B(A, B, I) = \frac{ChangeCouplings(B, A, I)}{Checkins(B, I)}$$

where $ChangeCouplings(X, Y, I)$ is the number of times file X and file Y are checked in together during time interval I and $Checkins(X, I)$ is the total number of times file X is checked in during I.

Like in the case of $CloneCoverage(X, Y)$ the index defining the file has been left out when the equation is covering not a specific pair of files but all files. $CouplingCoverage_X(X, Y, I)$ is thus abbreviated to $CouplingCoverage(X, Y, I)$. In this thesis we use the convention that the indices are left out when the equations do not refer to a specific pair of files.

In theory,

$$0 \leq CouplingCoverage(X, Y, I) \leq 1$$

should hold.

In practice there are two cases in which the above equation is not fulfilled. The first and more frequently encountered problem occurs if file X has never been checked in (and therefore not

[4] http://www.cvshome.org
[5] http://www.bugzilla.org
[6] As of September 19th 2005.

been changed) during interval I, which leads to $Checkins(X, I) = 0$. In this case a division by zero occurs and $CouplingCoverage(X, Y, I)$ is not a number. In this case, the value is artificially set to $CouplingCoverage(X, Y, I) = 0$. This decision is justified by the fact that since X has never been changed, there cannot be any change coupling between X and Y during period I.

The other anomaly that can occur is caused by the implementation of the change coupling concept in the RHDB. It is possible for two files to be coupled more often than one or both of them are changed in total. Two files are considered to share a coupling if they are checked into CVS during the same time slot spanning 15 minutes. If file X is checked in at time $0 \leq t_X < 15$ while file Y is checked in twice at $0 \leq t_{Y1} < t_X$ and at $t_X < t_{Y2} \leq 15$, this is counted as two couplings between X and Y. Since logic demands that two files cannot be coupled more often than the $Minimum(Checkins(X, I), Checkins(Y, I))$, any coupling coverage values larger than 1 are artificially set to 1. This can be justified because in the short interval between two check–ins of the same file not much can be changed – it is more likely that a small change has been forgotten before the first check–in necessitating the second.

A high coupling coverage between two files indicates a possible maintenance problem. There are various reasons why two files have to be changed together. One main reason are code clone pairs with fragments in both files, but it is not the only conceivable cause. Other reasons can include a functional dependency between two files or simply the fact that the same programmer is responsible for both files.

When combined with the clone coverage criterion, coupling coverage can serve as an indicator if the code duplications in question are dangerous – that is when the files are coupled because of the clones – or harmless – the changes to the files do not concern the cloned fragments. In the first case, the maintainability of the system could benefit from a refactoring, in the latter case, the system will most likely not suffer if the code clones are left in place.

Obtaining Coupling Coverage Data

The raw data necessary to determine coupling coverage values for the Mozilla case study can be found in the release history database. For other case studies, this data would have to be obtained previously. This process is not covered in this thesis and is treated in [FPG03]. However, since the RHDB is based on CVS data obtained from the public Mozilla repository[7], it only contains data that is generally stored by CVS or that can be derived directly from this data. One major drawback is that as of now, CVS does not provide information about where exactly the changes were made inside the file prior to check–in. Therefore this information is not contained in the RHDB and there is no way to find out from the database if the cloned fragments inside a file were affected by the changes made before a check–in. This check must therefore be done manually.

Both $ChangeCouplings(X, Y, I)$ and $Checkins(X, I)$ can be calculated directly from the entries in the database using only the three tables `cvsitem`, `cvsitemlog` and `cvsitemloggroup`. X and Y correspond to the relative paths of the files under examination. The interval I must be broken down into a starting date I_1 and an end date I_2 to be compatible with the database tables. For the purpose of the determination of coupling coverage, the actual values of the resulting tables are not important. Therefore the queries can be designed to merely count the elements of the results set.

The determination of the total numbers of check–ins of a given file X during an interval (I_1, I_2) is straight forward:

```
SELECT COUNT(*)
FROM cvsitem i, cvsitemlog l
WHERE i.rcsfile = X
```

[7] http://www.mozilla.org/cvs.html

```
AND l.cvsitemid = i.id
AND l.date BETWEEN I1 AND I2;
```

Extracting the number of change couplings between two files X and Y during interval (I_1, I_2) is also possible but requires joins over more tables:

```
SELECT COUNT(*)
FROM cvsitemloggroup g1, cvsitemloggroup g2, cvsitemlog l1,
      cvsitemlog l2, cvsitem i1, cvsitem i2
WHERE i1.rcsfile = X
AND i2.rcsfile = Y
AND l1.cvsitemid = i1.id
AND l2.cvsitemid = i2.id
AND l1.date BETWEEN I1 AND I2
AND l2.date BETWEEN I1 AND I2
AND g1.cvsitemlogid = l1.id
AND g2.cvsitemlogid = l2.id
AND g1.groupidx = g2.groupidx;
```

With these values, the coupling coverage can be determined for each pair of files during a given period. Later on, these values shall be correlated with the clone coupling data.

4.2.5 Correlation of Code Clones with Change Couplings

A correlation between code clones and change couplings has so far been taken for granted. This assumption largely depends on whether or not in the case of a coupled change the duplicated source code fragments were affected. Coupled changes must be examined manually to determine the changes that lead to a coupling.

The values obtained are displayed in a chart plotting coupling coverage values against clone coverage.

Thereafter, a regression analysis of the data obtained is attempted to quantify. This correlation is needed in order to define a metric for the impact of a code clone on the evolutionary behavior of code fragments. Two premises must be fulfilled for this regression to be significant. One is that a representative sample of all files containing code clones is used for the calculation. The second is that this random sample can be described with sufficient precision by a regression function, meaning that the correlation coefficient is sufficiently close to 1.

4.2.6 Definition of a Metric to Describe the Impact of Code Clones

An intuitive distinction between "harmless" and "dangerous" code clones has so far been used in this thesis. For the application of this classification to other case studies, a more formal approach is necessary. In the first place the factors that influence the classification of code clones in relation to change couplings are discussed. Following this, an attempt to defining a measure of the dangerousness of individual code duplications is made. Finally a way of visualizing this newly defined metric is discussed.

Influencing Factors

As has been mentioned before, it is not possible to sufficiently classify a code duplication solely on the basis of its size as measured by detection tools. The importance of a code clone is based mostly

on its context in the whole system. Even large fragments of duplicated code can be insignificant to the evolution of an examined system if they occur in library files which – even though they form a part of the project in question – are maintained externally. On the other hand there are small cloned code fragments in important parts of the system's source code that are the causing many change couplings. Code clones always have to be examined in relation to the total lines of code in their environment (usually a file) as absolute values are hard to compare.

The same considerations apply to the study of change couplings. Again these values must not be seen as absolutes but rather in proportion to their environment which in the case of change couplings are the total check–ins of – or rather changes to – the files in question.

Despite the deliberations leading to the conclusion that the clone and coupling values are examined as proportional rather than absolute values, these absolutes nevertheless have a further influence on a metric for the "danger" inherent to a code clone. Generally speaking, a longer fragment of duplicated code tends to have a larger influence than a shorter sequence and a file that is changed and checked in more often has a greater potential of presenting a problem than a file that is never touched during the evolution of a system.

Several interesting parameters are currently not obtainable automatically. Since CVS does not provide the exact position in the source code where a change has occurred, these missing parameters unfortunately include information if a change leading to a check–in really affected the code clone. One objective of the proposed metric is that it can be calculated without human intervention. Therefore, all information that would have to be obtained manually has been excluded from the calculation.

The input parameters for the calculation of the metric include:

- Clone coverage,
- Coupling coverage,
- Length of cloned fragments, and
- Absolute number of coupled check–ins.

Definition of a Metric

Any possible measurement for the impact of certain code clones must take into account the parameters mentioned above. Even so, it cannot be an absolute metric because of the factors that are also considered important but which are as of now unobtainable by automated means. A measurement could rather serve as an indicator about which clones could be good candidates for further inspection and for a possible refactoring effort.

The first premise for a metric with a certain explanatory power is that clone coverage and coupling coverage are significantly correlated. To demonstrate such a relationship, enough data must be obtained to allow the calculation of a regression function. To be significant, the result of such a regression analysis would have to have a coefficient of determination (or R^2–value) close to 1. If such a correlation is not apparent, the usefulness of any defined metric based on these parameters is highly doubtful.

The length of the cloned code fragments as well as the absolute number of couplings are of importance mostly to weigh the results further. A longer clone in a file with more couplings shows more potential for a refactoring than a smaller clone even though that might yield the same combination of clone and coupling coverage values.

If such a metric is meaningful depends largely on the statistical significance of the correlation between clone and coupling coverage values.

Visualization

Because of the difficulty to express the four dimensions making up the danger inherent to a given clone in one metric, a different light–weight approach is proposed. This method of visualization is inspired by Michele Lanza's polymetric views implemented in the CodeCrawler reverse engineering tool[8] [LD03].

In the visualization, the four dimensions listed above can be displayed in a Cartesian coordinate system enriched with additional use of color and the diameter of points in the chart. The dimensions are used as follows:

- **Clone Coverage** → x–axis
- **Coupling Coverage** → y–axis
- **Length of the Clone** → size of the point
- **Number of Couplings** → color of the point

The defined metrics are also shown in Figure 4.2.

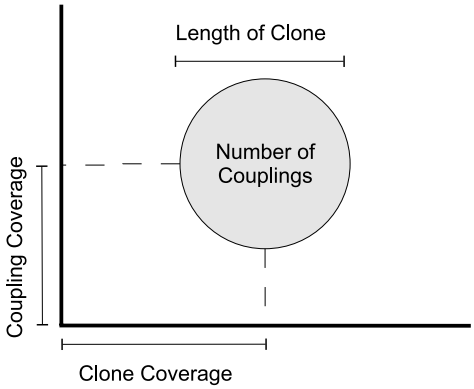


Figure 4.2: Description of the metrics used in the visualization.

The x– and y–axes run from 0 to 1 as in the diagrams used earlier in this thesis.

The size of the point is defined in proportion to the length of the clones. The maximum diameter is fixed and corresponds to the length of the longest clone. All other diameters are calculated proportionally to the length of the rest of the clones:

$$Diameter(A) = f(ClonedLines(A, B), max(ClonedLines(X, Y))$$

where A is the file the point stands for, B the file with which A is compared and $max(Cloned–Lines(X, Y))$ is the maximum length of cloned fragments encountered in the case study. More elaborate, the function leading to the diameter of any point is

$$Diameter(A) = MaxDiameter \cdot \frac{ClonedLines(A, B)}{max(ClonedLines(X, Y))}$$

where $MaxDiameter$ is a constant describing the maximal diameter of a point.

[8] http://www.iam.unibe.ch/~scg/Research/CodeCrawler/index.html

The color is defined in a way that the highest number of couplings is displayed as red (RGB–value (255,0,0)). The intermediate colors are determined by variations of the RGB value proportional to the relative number of couplings so that a gradual transition to blue is achieved (RGB–value (0,0,255)), which corresponds to zero couplings. A value precisely in the middle between the maximum number of couplings and 0 therefore corresponds to a violet color defined by RGB value (127,0,127). Two values – the value of the R–channel and of the B–channel – are changing while G–channel remains a constant 0. The differences between the R– and B–values of the maximal number of couplings and the values for any other number of couplings can be described mathematically as

$$R = \frac{ChangeCouplings(A, B, I)}{max(ChangeCouplings(X, Y, I))} \cdot 255$$

and

$$B = 255 - R$$

where R is the RGB–value for red and B the RGB–value for blue of the color of the point in the resulting chart. A and B are the specific file under consideration. $max(ChangeCouplings(X, Y, I))$ represents the maximal number of couplings between any two files X and Y during interval I.

Unlike the numerical approach described above, this visualization is not dependent on a significant regression. The user is able to see possible problems and reacts by closer inspection of the affected files.

Challenges Concerning the Metrics

The numeric metric as described above has several weaknesses. A single number cannot express everything that would be necessary for an extensive conclusion about the amount of problems a code clone poses to the maintainability of the system.

For example, clone candidates reported by detection tools are often outright false positives or otherwise of insignificant – or inferior – quality. Moreover, even if two files are connected by a high coupling coverage value, it is not necessarily the case that these couplings have been caused in any way by code duplications.

The visual approach also suffers from these problems. However, it is possible to convey a much larger amount of data than in the case where all aspects are aggregated into one single number. Therefore, the visualization is probably more useful to a user interested in detecting problematic code clones.

Both methods also rely on the availability of sufficient historic data as the coupling–related parameters are only available in hindsight. The expressiveness declines if only data for a short period of time is available.

One drawback of the visual approach is that in case of very large volumes of input data the resulting visualization gets cluttered and very hard to understand.

In any case, both approaches can only serve as indicators for possible problems. A code clone's danger of posing a maintenance problem can only be determined conclusively, if the files in question are compared manually to see if the duplication is the cause for change couplings. However, the proposed metrics can help to focus the search for problematic elements by narrowing down the possible candidates.

4.2.7 Implementation of a Prototype Tool

Intention

This final phase of the framework has one goal: to simplify the preceding steps by integrating and automating as much as possible and therefore to reduce the workload of the user.

As this thesis is part of a larger effort to create a framework for a software reengineering tool, no stand–alone application with a focus on the relationship between code clones and change couplings is planned. Instead, a number of library classes limited to certain aspects of the problem are written, which can be integrated into the larger project.

Domain and Requirements

As the goal of the implementation phase is the production of several loosely coupled library classes, the problem domain has been broken down into several distinct assignments that might not necessarily correspond to a single phase of this framework.

The problems are listed in the temporal sequence dictated by the separate phases of the framework:

1. Generation of an input file for CCFinder,
2. Parsing of a CCFinder output file,
3. Parsing of a C/C++ source code file,
4. Retrieval of coupling related data from the RHDB, and
5. Computation of metrics.

The first issue is not connected to the rest because there is currently no possibility of an integration of CCFinder into the broader framework these library classes are part of. Problems 2, 3 and 4 are all preconditions for the calculation of any metrics but are also used on their own.

Possibilities

The aspect of visualization has so far only been implemented rudimentary. If this simple visualization can be combined with the ability to jump to the respective source code entities (possibly combined with the highlighting of cloned code), a user will have more possibilities to explore the problem more effectively.

Another aspect that is not yet implemented is the integration of the CCFinder code clone detection tool. Since this program is not distributed as open source, this integration could not be realized yet.

As the classes do not define a stand-alone utility, it will be necessary to include them into a larger system. The functionality they provide can be used universally despite of its context int he larger system.

Chapter 5

Case Study

This chapter describes the proceeding during the application of the previously introduced tools and methodologies to the case study and presents the results of the experiment and the insights gained.

5.1 Evaluation of Clone Detection Tools

This phase of the framework has already been covered in detail in Chapter 3. The decision taken was to employ CCFinder for the most part and Duploc as a backup. Because of problems with Duploc encountered later in the case study, CCFinder was finally used exclusively.

5.2 Application of Clone Detection Tools

This Phase comprises the actual clone detection runs on the selected Mozilla case study. This section treats the setup used during the experiment, a definition of the files examined for code duplication and a description of the actual detection runs.

5.2.1 Environment

All clone detection tools were run on a computer under Windows XP (Service Pack 2) on a 2,8 GHz Pentium processor. Initially our machine was equipped with 512 MB of RAM, which was soon found to be inadequate. In the final configuration, memory was increased to to 3,5 GB of RAM.

For the programs that need Java, both JRE 1.4.2 and 1.5.0 were provided, but wherever possible release 1.4.2 was used. Duploc needs a VisualWorks environment for execution. We used versions 3.0 and 7.3. VisualWorks 3.0 did not run on our test computer but only on a secondary machine with much limited hardware. Therefore, version 7.3 was used during the main test runs. We did not find any difference in the behavior of Duploc.

5.2.2 Files Selected for the Experiment

We decided early in the project to examine only those parts of the Mozilla case study, that are written in C or C++. The reason for this decision is in the availability of code clone detectors for

these programming languages, whereas there are fewer options for other languages used in the Mozilla project (*e.g.*, Perl). Also, C and C++ files form the bulk of the project's source code.

The source code as obtained from the Mozilla website[1] included many files that were not of interest for this survey. These included among others makefiles, binaries, various files created by CVS and files written in a different source code language. All files except those clearly associated with C or C++ have been deleted along with empty directories. This leads to a structure containing only files ending with ".c", ".cpp" or ".h". The decision was taken in order to facilitate and speed up the definition of the input used for the following clone detection runs.

Early test runs were conducted in order to determine the size of the input most suitable for the next phase of the survey. Due to certain known limitations to the size of input code in Duploc these early tests were attempted with CCFinder using Gemini for visualization. During all detection runs, the Java heap space allocated to CCFinder was set to a maximum of 1.460 MB. If more was allocated, CCFinder would not load properly anymore.

First it was attempted to compare the seven complete releases of Mozilla in one detection run. Since this input amounted to a total of 27,353,289 lines of C/C++ code in 88,538 files, it was not expected that this run would succeed. Even after upgrading the system to 3,5 GB of RAM the test failed. The result was considered neither surprising nor disappointing since the amount of data would have been too big for a visual inspection even if the run would have succeeded.

The next attempt included only one complete release at a time. The maximum size was therefore reduced to 3,986,846 lines of code in 13,646 files (these extreme values are achieved in Mozilla release 1.4). The clone detection could be executed with a peak memory usage of 2,04 GB. The CCFinder output log files were generated without any problems. If Gemini can visualize the metrics and dot plot depends heavily on the value of the parameter defining the minimal length of the detected clone candidates. A length of 30 tokens is most often used in the known experiments and is therefore also used here as a default value. However, with larger size input, we often use longer cloned fragments (50 – 70 tokens) in order to ensure the tool's ability to visualize the results in a still readable form. In the case of a whole release, 70 tokens were considered adequate and still manageable by CCFinder. This resulted in a general overview of the clone situation in the system that could be used to select further areas of interest. The resulting dot plots are shown in Figures 5.1 to 5.4. The modules are arranged along the central diagonal as well as possible. Areas of interest for further study – *i.e.*, with a large percentage of duplicated code – can be identified by clusterings of dots.

During this phase an additional problem with Duploc has been detected. In certain circumstances, Duploc does not correctly detect line breaks in input source code files. In that case, the whole file is considered to be one long line. Since Duploc's clone detection algorithm is line-based, this means that a candidate can only occur if two files are completely equivalent which in reality is of course never the case. All input files of our case study – however not the input we used for test purposes during the first phase – were treated in this way which rendered Duploc virtually useless to us. The reason for this behavior is unknown at present.

The final decision was to test seven releases of a single module at a time. That way, the evolution of the clones could be examined over all releases while the size of the input was still small enough to be handled by CCFinder and for the most part by Duploc. The output is also not too cluttered to be understandable. We do however loose the possibility of comparing clones outside a specific module – thus each module is considered to be a separate program. To cover this shortcoming, some additional clone candidates from different modules obtained in the detection run of one complete release will also be taken into consideration.

It was also decided against using the header files (*.h) for comparison purposes. The reason for this lies in the fact that these files generally only contain declarations and no functionality (with the exception of a small proportion of inline functions). Furthermore, due to the rules used

[1] http://www.mozilla.org/releases/

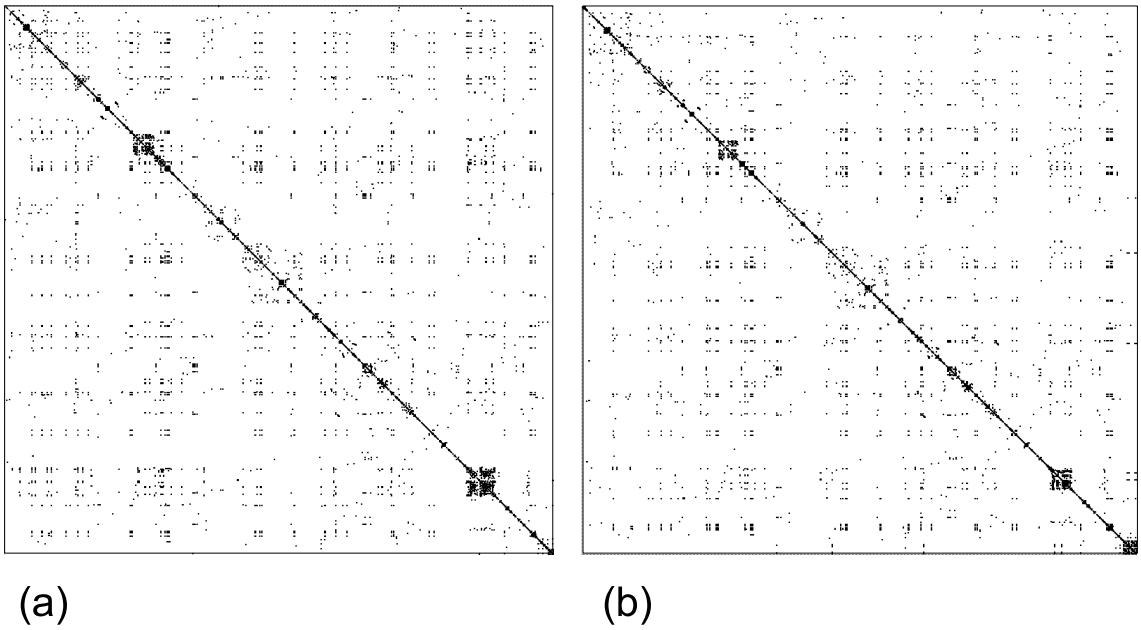


Figure 5.1: Dot plots for Mozilla releases 0.9.2 (a) and 0.9.7 (b) (70 tokens).

to parametrize the input in CCFinder, most declarations would be reported as candidates and would have to be eliminated manually later on.

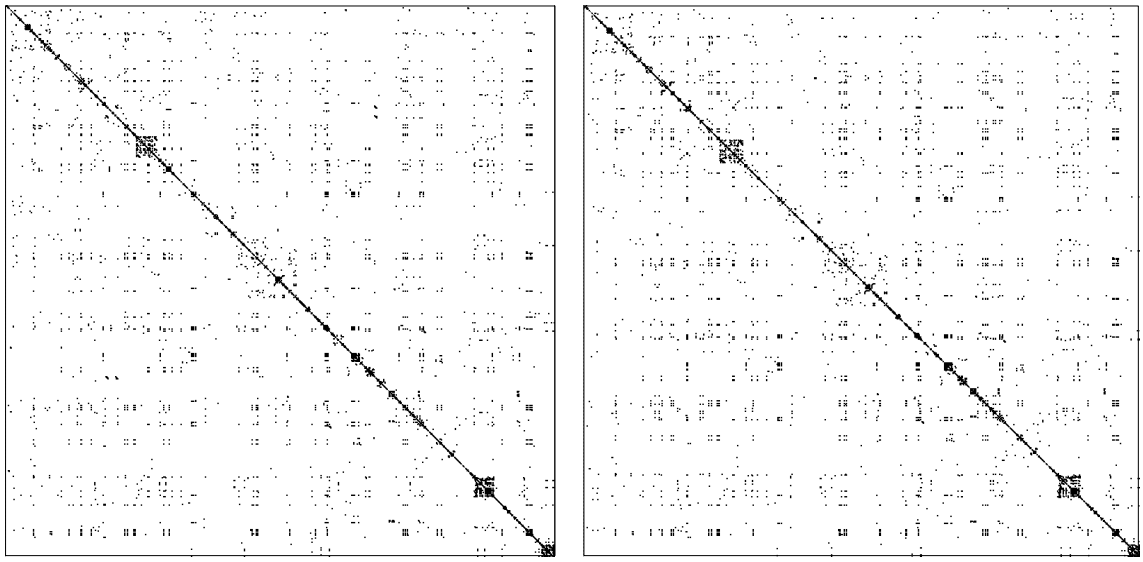
5.2.3 Clone Detection Runs

After determining the input size best suited for our needs – seven releases of one module at a time – the clone detection phase was started by using all seven releases of Mozilla separately as input for CCFinder. Unlike in later detection runs, a minimal clone size of 70 tokens was used, which seems to be a lower boundary for inputs of this order of magnitude. Attempts with a shorter length of clones invariably failed because of memory issues. The results obtained serve only to create a first overview of how many clones are expected. It also shows if it is possible to detect any significant change during the time interval under consideration – a determined effort at refactoring could for example lead to a system-wide decline of code clones. The scatter plots of the results of this phase are shown in Figures 5.1 to 5.4. These plots served to identify modules containing a high amount of code clones. Which modules the visible concentrations of code duplication belong to can be inferred from the affected directories. The concentration found on the lower right end of the diagonal line is for example made up of directories belonging to module "GFX and Widget – Mac".

Out of a total of 95 modules, 15 were considered to contain a sufficient amount of cloned code to be of interest for studying intra-module clones. The concrete selection of code clones for further study started with the plots described above.

The dot plots obtained through code clone detection runs on entire releases of Mozilla were also used to determine if there are inter-module clone pairs worth considering in a later phase. Here, unlike in the selection of candidate modules, the focus was on individual files rather than whole folders and modules. The description of the selection process is continued in Section 5.3.1.

The modules that were considered suitable for further study were then subjected to a more thorough code clone detection run. All seven releases of each module were examined together to



(c) (d)

Figure 5.2: Dot plots for Mozilla releases 1.0 (c) and 1.3a (d) (70 tokens).

get an overview over both the code clones in one version as well as the changes inside a certain cloned fragment introduced from one version to the next. Several inter-module candidates were also subjected to these clone detection runs. As with the modules, all seven releases of a pair of files were tested together. For both of these detections, the minimal clone size was lowered from 70 tokens to 30 tokens. This threshold is widely used in papers relying on CCFinder to detect clones and can therefore be considered a quasi-standard (*e.g.*, [KKI02, KSNM05]). As a lower boundary of 30 tokens eliminates code clones that are too short to be considered important, such a limit is reasonable. In our case study, a sequence of 30 CCFinder tokens represent about 2.4 lines of C or C++ source code [2].

5.3 Identification of Suitable Clone Candidates

The role of this phase is basically as an intermediate between the detection runs described above. Therefore the results obtained during the repeated cycle of the detection and this selection step is covered in Section 5.2.

Work in this phase consisted in the comparison of dot plots generated by the clone detection runs and the examination of the source code of the files involved to determine the validity and interest of a given duplication. Insights gained during these examinations that are not relevant to the application of the framework to the Mozilla case study are described in Section 5.3.3.

[2]The lines of code here include comments and blank lines even though these are usually not considered to be code clones. When they are ignored, the length of code fragments rises to around 2.9 – 3.6 NCLOC.

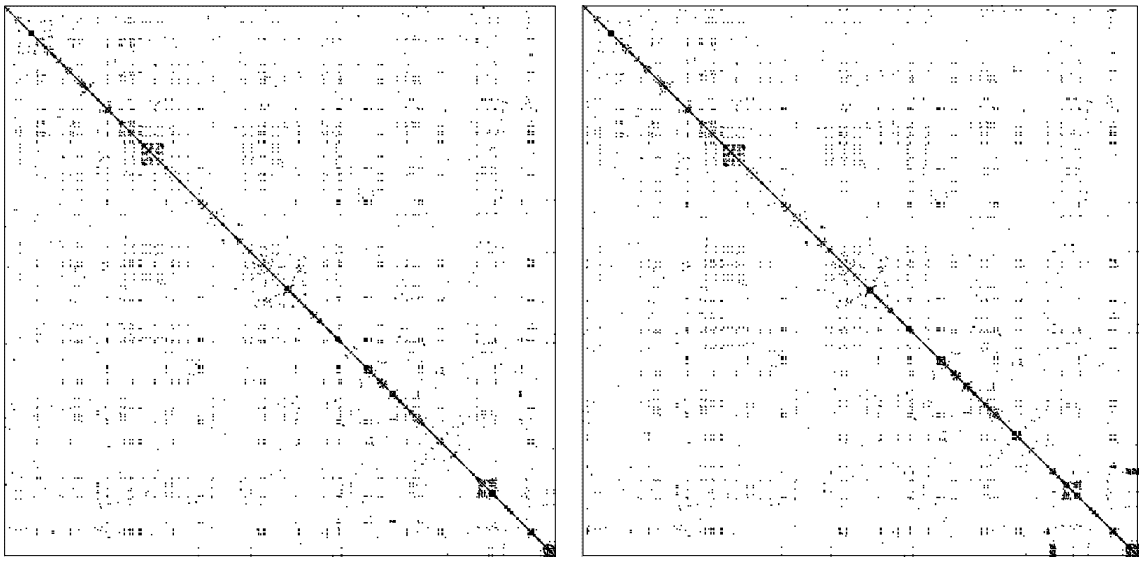


Figure 5.3: Dot plots for Mozilla releases 1.4 (e) and 1.6 (f) (70 tokens).

5.3.1 Selection of Code Clones

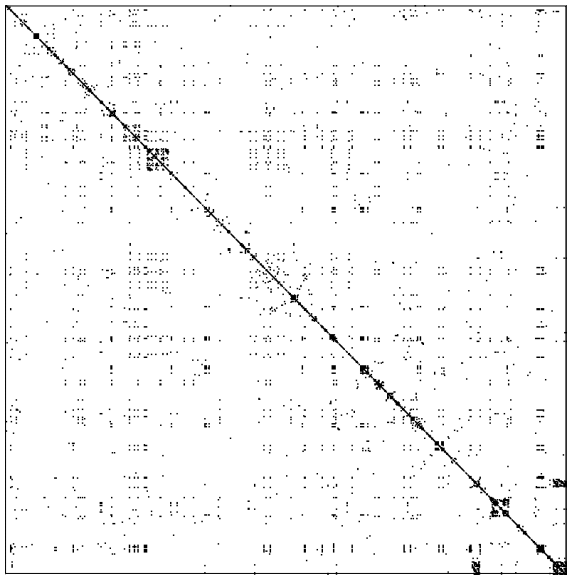
Not all code clone candidates that are detected by CCFinder can be used for the purpose of this case study. One obvious problem are false positives. These were eliminated by manual inspection before proceeding. Of the remaining clones, those of types 1 to 4 as described in Section 4.2.3 were selected as the most interesting because their relative lengths change during the evolution.

Another kind of candidate that has been considered unsuitable is a true clone that has more structural than functional implications. Due to the parametrization taking place during every CCFinder detection run, several candidates usually consist purely of sequences of similar statements without much semantic content. Examples for this are sequences of `#include`–statements, declarations of variables or `switch`–statements.

Because this case study covers seven subsequent releases of Mozilla in parts with substantial changes, there are files that only appear in one or two releases altogether. Since clones in such files do not provide much evolutionary information, a lower limit of four releases was set in which any file containing a clone should appear.

To determine the pairwise clone coverage values for each pair of the evaluated files, a simple tool was developed and applied. It extracts the number of cloned lines of code from a given CCFinder output file and calculates the non-commented lines of code directly from the source files. In some phases during these calculations a problem with the way CCFinder logs its output was detected. Despite the fact that CCFinder is supposed to eliminate blank lines and comments, there are sometimes remnants of comments apparent in the representation of the code clones. This seems to happen mainly if there are lines consisting of pure comments in the middle of a cloned sequence. Instead of reporting two clones, the comment lines are also included. In the calculations for this case study this leads in several cases to a clone coverage value larger than 1, which should not be possible according to the definition. Since this value will only get larger than 1 if the clone coverage is anyways exceedingly high, these values were reduced to 1.[3]

[3] Another possibility would be very long sections of comments embedded in code clones. It was verified manually that



(g)

Figure 5.4: Dot plots for Mozilla release 1.7 (g) (70 tokens).

5.3.2 Clone Coverage in the Case Study

A final criterion was the wish to include clones of as many different evolutionary classes as possible to achieve a sample of data that reflects the entire source code of Mozilla. These classes have been defined in Section 4.2.3.

Altogether a sample of 31 files from 15 disjoint clone classes forming 21 clone pairs have been selected and examined individually. The results of this examination can be found in Appendix B. Hardly any of these pairs are of a single evolutionary type over the whole examined interval. Most stable phases have during their lifetime. This leads to type 0 clones being present in 13 clone classes. Nearly as numerous are type 3 and 4 clones which are represented in 11 respectively 12 classes. Not surprisingly type 1 – in two classes – and type 2 clones – present in only one class – form a minority.

The sample of files selected feature a wide range of clone coverage values. As the files have mostly been chosen because the cloned sections within the pairs changes over the seven releases of this case study, there are interesting combinations of clone coverage values. However, the sample is for precisely the same reason not a random sample that can be used for statistical and regression analysis.

There are files that consist nearly entirely of clones of another file over all seven releases. The most obvious example for this is a file whose clone coverage value starts at 0.82 in release 0.9.2 and rises to 0.97 until release 1.7. Other pairs' clones remain more or less stable at very different levels ranging down to 0.1 and smaller.

Other clones show more interesting behavior like a clone coverage value sinking from 0.8 to 0.27 over the course of seven releases. The largest observed difference between two subsequent releases is from 0.51 to 0.07 – in about six months the size of the mutual clones has in this case been reduced by about 86%, probably a sign of a major reengineering effort.

this is not the case where the value has been adjusted.

Clones completely disappearing are rare. The largest clone coverage value completely disappearing in a subsequent release of the same file is at 0.04. Clones being introduced into two files that did not contain any earlier is also not a common occurrence. However changes of the clone coverage values by as much as 0.44 have been observed between two releases where there were no clones present at first.

5.3.3 Additional Insights

The results described in this section are not directly related to the research questions treated in this thesis. However, they might otherwise be of interest.

When a module is compared to a different release of itself, it is possible to detect the effects of certain refactorings directly through patterns in the dot plots generated from the clone candidates. One common refactoring technique is to move functionality from one or more files into a new file (*e.g.*, in the refactoring known as "Extract Class" [FBB+99]). This modification leads to an identifiable pattern in the dot plots when one version of the module which is shown in Figure 5.5. Basically the clones disappear from several files and re-appear in a new or different existing file in the next version. The effect is only visible when two versions are compared to each other, not when the two subsequent versions are only compared each to themselves. In the figure, (a) is the row associated with file A in version 1.0. Column (c) represents file A in release 1.3a. The two versions of the same file do not contain any clones. Instead, some functionality which in release 1.0 was in file A has been moved to a file B, which is new in release 1.3a. This file is associated with column (b) and shows the moved fragment formerly of file A as a clone.

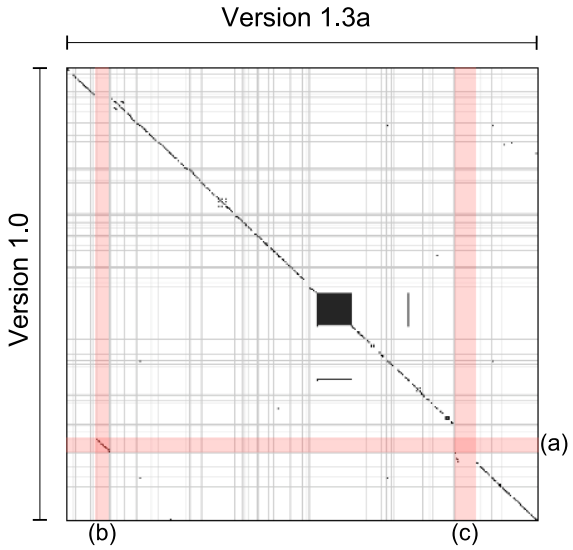


Figure 5.5: Example dot plot showing moved functionality.

5.4 Extraction of Change Coupling Information

During this phase the concept of change coupling is examined for the case study. It is so far independent of the previous three steps except for the fact that the same sample files are used.

Phases 3 and 4 together produce the input needed for further research in the following steps.

There is one major difference between the examination of code clones and that of change couplings: the former involves the exploration of files at a given moment – the date of each new release of Mozilla – while the latter must be investigated over a given interval. As Table 5.1 shows, the intervals coupled with a certain release are starting at the date where the previous version was released and end the day prior to the release of the version in question. The reason for this partitioning is that during the specified interval those changes have been implemented, that lead to the current release. In the case of release 0.9.2 – the first release examined – an earlier release (0.7) has been selected so that the interval leading to 0.9.2 is about the same length as that leading to the other releases.

Release	Date of Release	Interval
0.9.2	June 28th, 2001	Jan 9th, 2001 – Jun 27th, 2001
0.9.7	December 21st, 2001	Jun 28th, 2001 – Dec 20th, 2001
1.0	June 5th, 2002	Dec 21st, 2001 – Jun 4th, 2002
1.3a	December 13th, 2002	Jun 4th, 2002 – Dec 12th, 2002
1.4	June 30th, 2003	Dec 13th, 2002 – Jun 29th, 2002
1.6	January 15th, 2004	Jun 30th, 2003 – Jan 14th, 2004
1.7	June 17th, 2004	Jan 15th, 2004 – Jun 16th, 2004

Table 5.1: Mozilla releases with the time intervals relevant for the changes reflected therein.

A parallel to the evaluation of the code clones is the fact that the absolute value of the number of change couplings is not sufficient. A similar metric as for the code duplications has therefore been created.

5.4.1 Coupling Coverage in the Case Study

The coupling coverage values have been calculated for the same sample of files as the clone coverage. Other than the clone coverage, the coupling coverage is observed during an interval. In a first step, the values for each file have been calculated for any of the seven releases composing the case study. The relevant intervals can be found in Table 5.1. Following this examination, the coupling coverage has then been considered over the whole life cycle of the system until release 1.7.

Value	Absolute Frequency	Frequency in Percent
< 0.1	35	12.24 %
0.1 – 0.2	19	6.64 %
0.2 – 0.3	12	4.2 %
0.3 – 0.4	14	4.9 %
0.4 – 0.5	22	7.69 %
0.5 – 0.6	8	2.8 %
0.6 – 0.7	22	7.69 %
0.7 – 0.8	13	4.55 %
0.8 – 0.9	13	4.55 %
0.9 – 1.0	128	44.76 %
< 1.0	286	100 %

Table 5.2: Frequency of change coupling coverage values.

There are some instances where one or both of two files sharing a cloned code fragment are never checked in during a given interval. In this case, a division by zero would occur. To avoid this problem, the value of the coupling coverage for those two files is for the interval in question set to 0.

The evaluated files provided a total of 286 values for change coupling coverage. There is a remarkable concentration on either end of the scale as Table 5.2 shows. One explanation for the high percentage of values between 0.9 and 1.0 is the fact that most files in the sample have been selected for their relatively high clone coverage. There are two examples of clone classes that show coupling coverage of 1.0 in almost all cases. However, the sample also contains clone pairs with a very low clone coverage. However, the sample cannot be considered randomly chosen.

5.5 Correlation of Clone Data and Change Couplings

We attempt to determine the impact of code clones on change couplings by correlating the values gathered during the last two phases of this case study. The coupling coverage is interpreted as a function of clone coverage.

Three adjustments have been made to the raw data obtained during the previous phases:

1. If the clone coverage is larger than 1.0 as described in Section 5.3.1, the value in question has been reduced to 1.0 exactly.

2. If the coupling coverage results in a division by zero as described in Section 4.2.4, the value is set to 0.0.

3. If the coupling coverage results in a value larger than 1.0 as described in Section 4.2.4, the value is set to 1.0.

The first step in correlating the two different series of data was an attempt to assess the importance of the code clones in relation to the change couplings.

5.5.1 Classification of Results

Code clones are considered one of the major bad smells [FBB+99]. It is therefore expected that the bigger the amount of duplication in a pair of files is, the more those files are coupled during the subsequent evolution of the system. Likewise, a code clone that later on spawns change couplings is considered more harmful than a clone that is hardly ever modified at the same time wherever it is duplicated.

Any coupling represents changes in both files at the same time. What has actually been changed is so far not determined. It was expected that mostly the duplicated source code fragments were subject to change when the files are checked in together. Since CVS does not yet automatically provide this information – the log only contains data about how many lines were deleted or added, not however which lines – the changes were examined manually using `cvs diff`.

To determine a scale to classify the impact of code clones on change couplings, the changes that lead to the related check–ins were therefore examined manually and clustered into four types:

1. **Random Changes**: The changes made to the two files are not related through any code clones – they might however concern methods with similar purposes that do not share duplicated code fragments.

2. **Comments / Includes**: The changes only affect either comments in both files or `#include`–statements. Neither of these types of code fragments is considered a true code clone even though they might be the same in both files.

3. **Methods / Names**: The changes in both files only affect either names that have been changed or changed method calls. Usually the definition of the methods in question is found in a different class so that no further changes are made to the code of the files being examined.

4. **Changed Clones**: The changes occur in the duplicated fragments of both files at the same time. These changes go further than changes covered by the previous classes and often change the program logic.

A fair amount of indications for a connection between code clones and change couplings can be found in the sample of files examined during this case study. The most obvious of these are pairs of files that have a clone coverage value greater than 0.9 and that are coupled in every single case when they were checked in. A good example, since clone coverage for one of the files is continually between 0.9 and 0.97 and for the other between 0.83 and 0.93 are `mathml/base/src/nsMathMLmunderFrame.cpp` and `mathml/base/src/nsMathMLmoverFrame.cpp`. During their evolution until Mozilla release 1.7 they are coupled in every single case of a check–in. Considering the very high clone coverage values for both files, it is not surprising that the changes leading to any coupled check–in were never independent. Of the 42 examined check–ins, 7 were caused by changes to either the comments or to `#include` statements. 10 changes concerned method signatures or the renaming of variables or methods. The remaining 25 couplings were all caused by changes to the cloned fragments – 16 of these were exactly equivalent in both files (except for the different names of certain functions – usually an "over" in one file was replaced with an "under" in the other). These two files are together with the remaining member of their clone class a good example of a "copy–paste–modify" approach to programming. It is expected that their maintainability would benefit from a major refactoring effort. Since a very high percentage of change couplings can be directly linked to existing code clones, these duplications must be classified as dangerous to the evolution of the Mozilla project.

Oppositely to the case treated in the previous paragraph, it is expected to see relatively low values of coupling coverage in files that do not share much duplicated code. One example for such a connection is the file `gfx/src/windows/nsRenderingContextWin.cpp` when it is compared to `gfx/src/xlib/nsRenderingContextXlib.cpp`. Because of the different number of lines of code and check–ins in the two files, it is necessary to calculate both coupling coverage and clone coverage separately for each. This results in 4 values for the two files. The clone coverage of the first file turns out to be between 0 and 0.21 while the latter has values between 0 and 0.23. The coupling coverages are 0.17 and 0.35 respectively. It is however noteworthy that these values can reach up to 0.8 and 1 when the focus is set to the developments in only one single release. The files are coupled 39 times. Only 2 of these couplings concern real cloned code fragments while a majority of 26 change couplings are caused by code sections that are not cloned (although coupled changes are often made to methods with the same function). 5 couplings are caused by changes to the comments or `#include`–statements and 6 concern modified method signatures. Despite at times high coupling coverage values between two subsequent releases of Mozilla, the relatively small percentage of code clones does not seem to cause major change couplings. The impact of the code clones can in this case be considered harmless.

There are also interesting combinations of values that seem to contradict too close a relationship between code duplication and change couplings. On one end of the scale there are files containing a large proportion of mutual code clones, but are almost never changed at all and even then only coupled rarely. Examples for such couplings are the files `jpeg/cjpeg.c` and

`jpeg/djpeg.c` which form part of the Image Handling: JPEG module[4] present in all releases of Mozilla. In every release, they contain about 60 % of mutually shared code, but their coupling coverage is still low – during the development, each of these files is checked in 5 times but only once they are coupled. On closer inspection, these files and essentially the whole module turn out to form part of a project independent of Mozilla. The files are maintained by the Independent JPEG Group[5] and the check–ins during the Mozilla project are mere updates of library files modified elsewhere. Since the files still are part of the Mozilla release, they are considered relevant. It is possible that the duplicated code fragments cause couplings during their development, but those clones are not relevant in the context of Mozilla. This kind of code duplication would therefore be considered harmless in respect to the evolution of this case study.

The other extreme position is formed by files that show a high coupling coverage either in one specific release or in total yet share only relatively few mutual code clones.

If only one release at a time is examined, it is possible to find examples where files with a clone coverage values of less than 0.1 are coupled every time they are checked in. However, total check–ins between two releases often amount to only a very small number. An example for this behavior is the file `intl/uconv/src/nsWinCharset.cpp` which in the time intervals relevant for releases 1.6 and 1.7 is coupled with `intl/uconv/src/nsMacCharset.cpp` every single time it is checked in. Total coupled check–ins only amount to 4 during this period. Of these 4 couplings, one was caused by a change to comments, one by a change in a method call, one by totally unrelated changes and only one by the cloned fragment itself. Despite the very high coupling coverage, such clones should in this context be considered harmless.

The second related anomaly mentioned concerns files with a perpetually large change coupling value and relatively low clone coverage. The most obvious example for this is `layout/mathml/base/src/nsMathMLmsubFrame.cpp` of the MathML module which is coupled with `layout/mathml/base/src/lnsMathMLmsubsupFrame.cpp` in 97 % of its check–ins over the whole observed evolution. The clone coverage never exceeds 0.45 and drops to 0.32 by release 1.7. These two files are coupled 30 times during the period covered by the observed releases of Mozilla. 12 of these couplings were connected directly to the duplicated code in the files and 5 were completely independent changes made to the files at the same time. The rest of the couplings were either caused by simultaneous changes to comments and `#include`–statements or to method calls when the method's signature had been changed – these cases were not considered to be caused by code clones. Only 40 % of the change couplings were therefore caused by code duplication. The clones in these files are not considered particularly dangerous.

5.5.2 Correlation

When the clone and coupling coverage values are plotted against each other, the first impression is that the resulting distribution is rather random and hard to approximate by any regression. Figure 5.6 shows the aggregation of the calculated values for the seven releases of Mozilla. The clone coverage values are calculated for each release. The coupling coverage values corresponding to a release are calculated during the time period before the release. The periods are defined in Table 5.1.

Because the exact time of the Mozilla release in which the clones and couplings occur are not relevant, all calculated values can be aggregated into one chart to clarify the distribution. This irrelevance of the exact time is based on the fact that the duration of the intervals are roughly equal and the method of calculation is insensitive to the exact point on the timeline on which the values are calculated. The aggregate chart is shown in Figure 5.7.

[4] http://www.mozilla.org/owners.html
[5] http://www.ijg.org

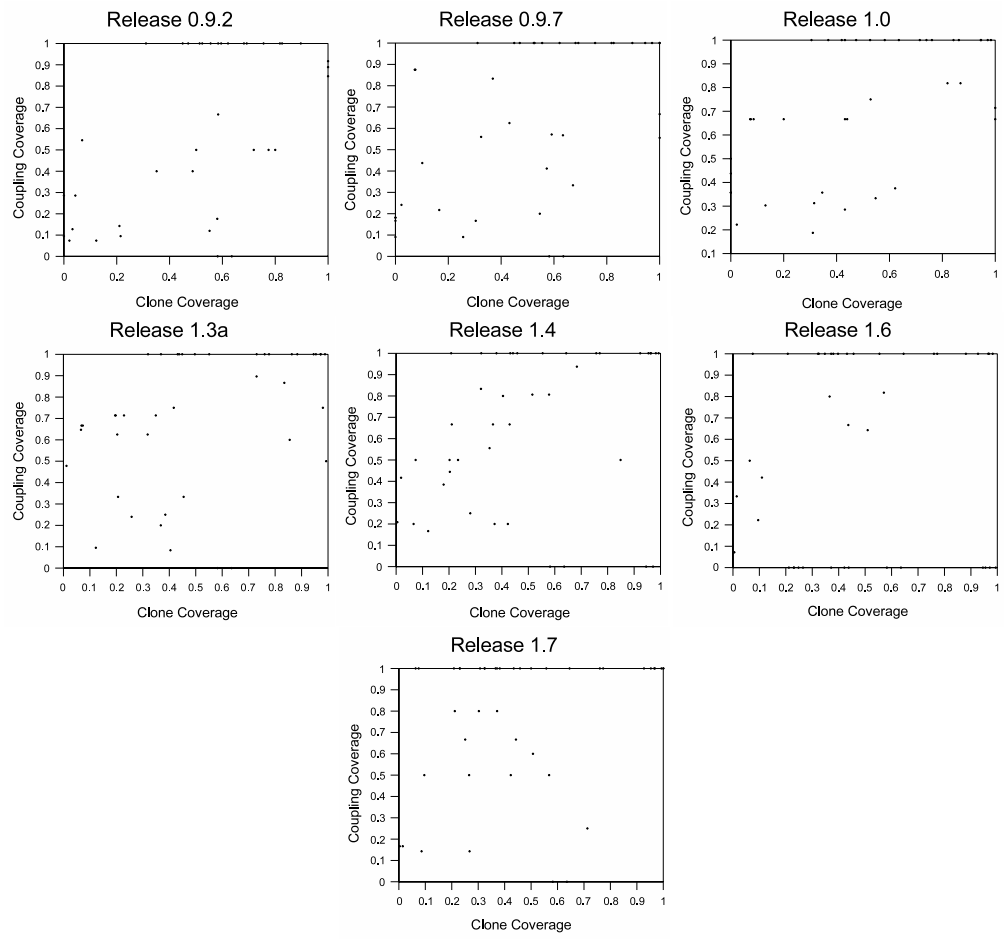


Figure 5.6: Clone and coupling coverage values for each release of Mozilla.

In the Charts 5.6 and 5.7, all coupling coverage values have been calculated for the interval of roughly half a year between subsequent releases of Mozilla. During these intervals, the absolute number of check–ins and couplings for individual files is generally low, often leading to extreme coupling coverage values of 0 or 1. In order to obtain values that are less reliant on very small numbers, the interval for the last release of Mozilla has been extended. In this case, all changes prior to the release date of Mozilla 1.7 have been considered to be in this interval because ultimately all of these changes lead to release 1.7 (the same reasoning has been applied previously to the shorter intervals leading to each release). The resulting chart is shown in Figure 5.8. These first observations show two results. The first is an impression of randomness – there is no obvious relation between code and coupling coverage visible in the data processed. The second conclusion is a consequence of the first. The amount of data obtained from the hand-picked sample of code clones is neither big enough to lead to meaningful conclusions nor is the sample statistically independent as the clones have been selected because of certain attributes.

To obtain a sample that is more suitable it was decided to obtain clone and coupling coverage data for a complete release of Mozilla. The release selected for the detection of clones was once

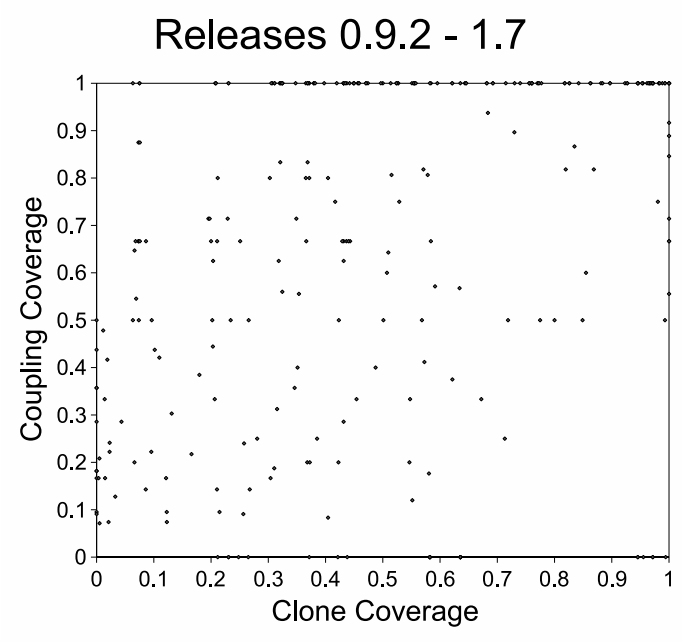


Figure 5.7: Aggregate clone and coupling coverages in Mozilla releases 0.9.2 – 1.7.

again 1.7. CCFinder has been configured to detect clones of a minimal length of 30 tokens which was in this case possible as no visualization in Gemini was necessary. Because of the large size of clone data, false positives could not be sorted out manually. But since long clone candidates tend to be real clones and the amount of data obtained is large, false positives should not have a major influence on the final outcome. The coupling coverage values were calculated for the whole period prior to the release date of Mozilla 1.7. Files that contain no clone candidates were not processed.

The calculations required a computation time of a little more than 15 hours and resulted in 139,523 clone coverage / coupling coverage tuples. The resulting chart is shown in Figure 5.9. The visible horizontal "bands" of values correspond with often obtained ratios like 0.5 or 0.2.

Of these combinations, 109,090 tuples or 78.19 % have clone coverage values of less than 0.2 while 30,433 or 21.81 % have values ranging from 0.2 to 1.0. As has been determined earlier, clone coverage values below 0.2 do not seem to have much influence on the change coupling behavior during the evolution of the system. It is expected that these relatively low level values of clone coverage will dominate any regression on the whole set of data. A visual inspection of the chart also implies that these values are associated more or less randomly with coupling coverage values.

The number of input combinations of X– and Y–values for a regression analysis is in this case study limited to 65,536. Therefore, a regression analysis will be carried out using first a random sample of the complete set of values and second all values with a clone coverage larger than 0.2 as input. This procedure will result in a conclusion about the expected behavior in the whole system and about the behavior of larger code clones (which are usually considered to be more dangerous). The criterion to determine the significance of a regression analysis is the coefficient of determination or R^2–value, which is a measure for the percentage of statistical spread that is explained by the regression function. The regression analysis is calculated using the least squares method.

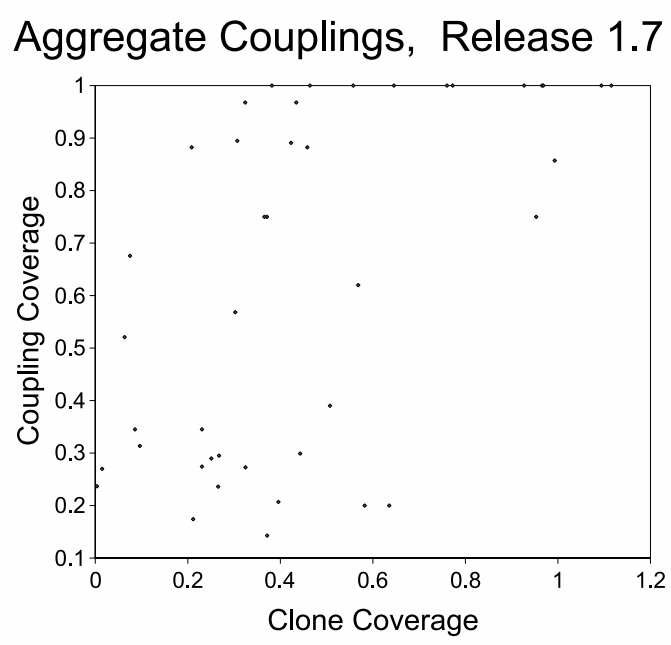


Figure 5.8: Clone and coupling coverage combinations for release 1.7 with aggregate couplings.

The distribution of data points in the generated chart 5.9 seems to indicate that a linear or logarithmic regression might be most appropriate although there are in both cases many values that deviate from such a calculated function. The distribution does not raise expectations of a very clearly defined regression function as the clone / coupling coverage combinations are seemingly random.

In total, regression analysis was conducted over three different random samples of 65,536 values each. In any case, the R^2 value was better for linear than for logarithmic regression over the same sample. A linear regression resulted in the best fitting function with the highest coefficient of determination of 0.702:

$$CouplingCoverage(A, B, I) = 1.038 \cdot CloneCoverage(A, B) + 0.097$$

The sample data used for this regression is shown in Figure 5.10. The resulting equation explains 70.2 % of the scattering visible in the chart. The other attempts at regression analysis yielded even lower R^2 values.

Similar regression analyses were computed for the 30,433 instances of data where the clone coverage was above the 0.2 threshold described above. In this case, no random sampling was necessary as all values could be evaluated. The best fit for these clone–intensive input combination was reached with a logarithmic regression. However, the R^2 value even for this instance was only 0.248 meaning that not even a quarter of the scattering is explained by the regression function. A linear regression with a coefficient of determination of 0.2088 resulted in a straight line with the equation

$$CouplingCoverage(A, B, I) = 0.512 \cdot CloneCoverage(A, B) + 0.4781$$

which is not a very close fit to the results obtained by a sample of all input values.

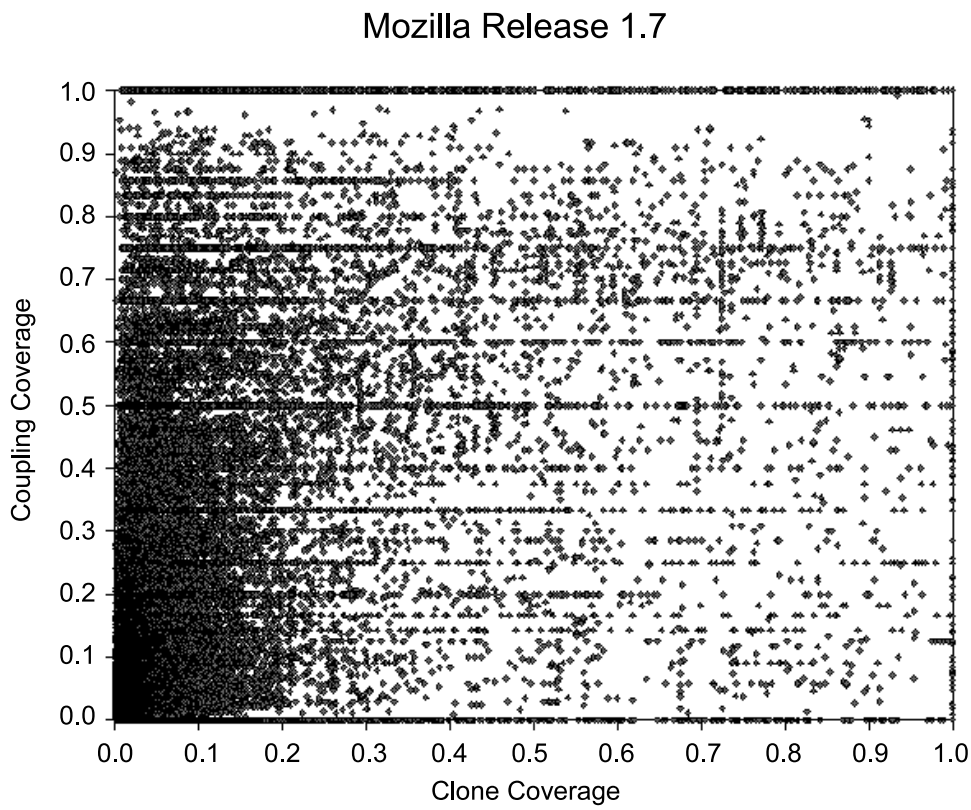


Figure 5.9: Clone and coupling coverage for the complete Mozilla release 1.7.

5.5.3 Conclusion

The findings during this case study seem to indicate a certain correlation between cloned fragments of source code and change couplings during the evolution of a software system. This connection was expected from previous work starting with [FBB+99]. Usually the larger the clone coverage between two files is, the more often these files are coupled. It is however neither possible to conclude that code duplications are always reflected in high change coupling coverage values nor is the opposite always true. From the results of this case study it is impossible to definitely exclude the possibility that there is in fact no statistically relevant correlation between code duplications and change couplings.

In this case study, a clear mathematical relation between code clones and change couplings cannot be established with any certainty. Even though for the entirety of all clone / coupling coverage tuples a mathematical relation with a reasonable significance can be established, in the case of files with a high percentage of code clones, this correlation becomes more and more insignificant.

Change couplings can have causes other than code clones. Files fulfilling similar roles in the system often are changed together even though they might not contain many duplicated code fragments. On the other hand, there are also code clones that can be classified as “harmless” since they are almost never changed at all and therefore very rarely coupled.

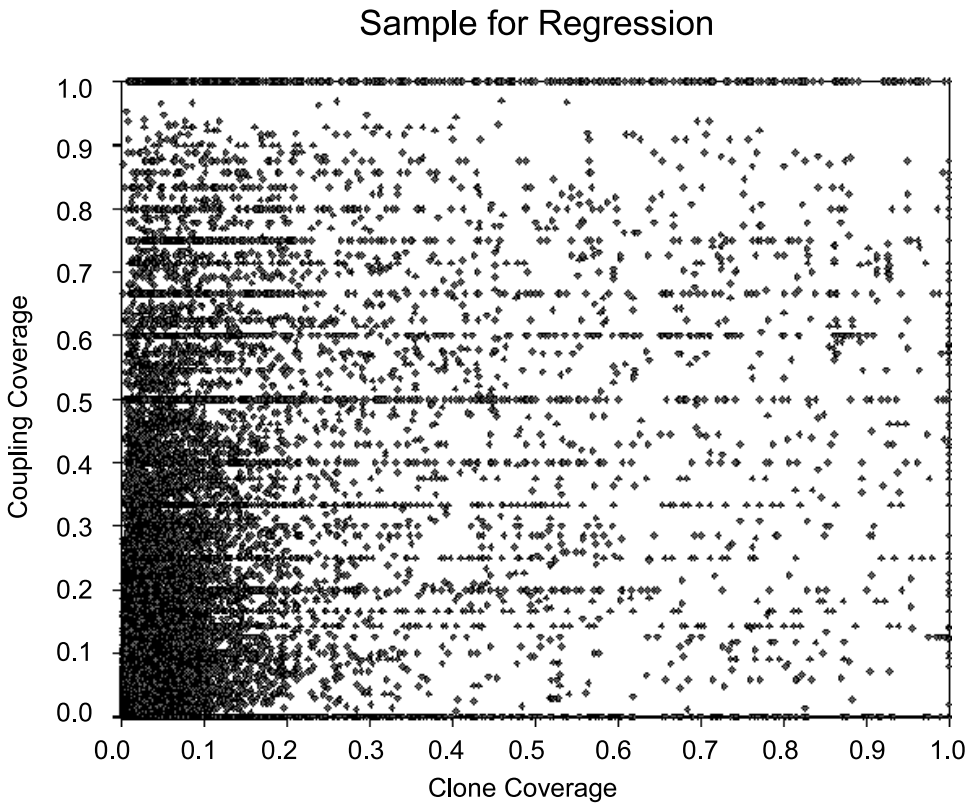


Figure 5.10: Sample used for regression.

Despite these exceptions, the general tendency for files with a high clone coverage value is to be coupled more often than files with a lower percentage of duplications despite the fact that this relation cannot be expressed with a simple mathematical equation.

An examination of clone and coupling coverage can be used to identify groups of files that would benefit from a determined refactoring effort. In this case study, several groups of files with medium to high clone coverage values have been found that are coupled nearly every single time the files are checked in to CVS. With modern refactoring techniques and paradigms of object–oriented software development, such “dangerous” fragments of cloned source code can be addressed.

It is not possible to distinguish harmless from dangerous code duplications simply by looking at the results of a code clone detection run on only one release of a software system. There are more factors to be considered to make a prediction of the future behavior of a code duplication.

It is however save to say that the larger a cloned code fragment is in percentage of its environment, the higher is the probability of it becoming “dangerous” during the evolution of a system.

5.6 Towards a Metric for the Impact of Code Clones

5.6.1 Definition of a Metric

Because of the difficulties of establishing a mathematical relation between code duplications and change couplings described above, the definition of a significant metric is almost impossible. The following equations are therefore not considered as a proven mathematical approach, but as an attempt at classifying the possible danger a code clone might pose to the future evolution of a system.

The metric that is developed in this section is not intended to provide an absolute ranking of the danger inherent to a number of clones. It is only possible to assign to a file a value in relation to the other clones of a sample input. The statement made is therefore along the lines of "File A contains clones of file B. File A also contains clones of C. But because of the coupling history and the size of the clones, the probability of file A to be coupled with B in the future is higher than the probability of A being coupled with C".

As has been established in Section 4.2.6, factors influencing this decision are not only clone and coupling coverage, but also the absolute length and number of couplings of a pair of files. Because of the relatively unreliable relations between the two coverage values obtained through regression analysis, the use of historical data can be considered safer than the exploitation of these equations. The premises for further development are therefore:

1. Files that have been coupled in the past are likely to be coupled in the future,
2. Files containing long cloned fragments of each other are likely to be coupled,
3. Increasing clones lead to increased coupling, and
4. Decreasing clones lead to decreased coupling.

First it can be assumed that all parameters remaining unchanged, the "danger" of a change coupling during an interval I_f in the future remains the same as during a similar interval I_p in the past:

$$Danger(A, B, I_f) = CouplingCoverage(A, B, I_p)$$

As has been explained earlier, change couplings can occur for many reasons that are not directly connected with code clones. Since the intention of this metric is to assess the danger inherent to a clone, the presence of clones is considered more important than the occurrence of couplings. Therefore the introduction of clone coverage leads to

$$Danger(A, B, I_f) = a \cdot CouplingCoverage(A, B, I_p) + b \cdot CloneCoverage_n(A, B)$$

where n is the current version and a and b are factors so that $a < b$ and $a + b = 1$.

Recent changes in the clone situation must be considered and are as of now not yet reflected separately in the equation. If the clone coverage increases, the danger inherent to the clone should likewise increase:

$$Danger(A, B, I_f) = a \cdot CouplingCoverage(A, B, I_p) + b \cdot C$$

where

$$C = CloneCoverage_n(A, B) + \Delta CloneCoverage(A, B)$$

and

$$\Delta CloneCoverage(A, B) = CloneCoverage_n(A, B) - CloneCoverage_{n-1}(A, B)$$

$Danger(A, B, I_f)$ has so far been defined in a way that $0 \leq Danger(A, B, I_f) \leq 1$ always holds in practice. $CloneCoverage(A, B)$ can, despite the addition of $\Delta CloneCoverage(A, B)$, never get larger than 1 nor smaller than 0 because of the initial definition of clone coverage.

Finally, the absolute length of a code clone must be introduced. The reason for this is the fact that larger code clones are usually difficult to maintain while on the other hand they often offer more possibilities for refactorings than small clones. To keep the value of $Danger(A, B, I_f)$ between 0 and 1, the length of the code clone again has to be set in proportion. Since the goal of the metric is to compare only the relative danger of the input sample, it is reasonable to calculate a value

$$RelativeLength(A, B) = \frac{ClonedLines(A, B)}{max(ClonedLines(X, Y))}$$

where $max(ClonedLinesX, Y))$ is the maximal length of a cloned fragment in the input sample of files. On inclusion into the equation, $RelativeLength(A, B)$ is allocated a parameter c so that

$$Danger(A, B, I_f) = a \cdot CouplingCoverage(A, B, I_p) + b \cdot C + c \cdot RelativeLength(A, B)$$

so that $a + b + c = 1$ and $a < b + c$ to keep the focus on the code duplications.

The values of a, b and c are not fixed but should be constant to produce comparable results. They reflect the emphasis that is being placed on the different aspects. It makes sense to assign values of about 0.3 – 0.4 for each of these parameters with the value of each b and c – which are the clone related parameters – being a bit higher than the value of the coupling related parameter a. Factors a, b and c cannot be universally fixed. They must instead be determined separately for every project.

When the $Danger$–values for a certain set of files has been calculated, it is possible to rank these files in the order of problems they are likely to cause during the future evolution of the system. This metric is however strictly limited to the comparison of files for which the $Danger$–values have been calculated together. This ranking can only give a general idea which files could be of interest for further study. To gain further insights, the input data should at the same time be visualized as this technique shows more clearly the scale of the metrics that have been used to create the $Danger$–value.

On its own, the $Danger$–value is inadequate to come to a definite decision about a certain clone. Like other metrics that condense complex circumstances into a single number, it can merely serve as an indicator or starting point for further investigation.

To evaluate the validity of this proposed metric, it can be calculated for a certain release of a case study. Then, the forecast is compared to the actual change coupling values during the later evolution of the system. For this examination, clones of types 1 to 4 as described in Section 4.2.3 should be used as they should show a marked difference. In the context of Mozilla, possible candidate files for this survey are *e.g.*, `gfx/src/os2/nsRenderingContextOS2.cpp` and `gfx/src/xlib/nsRenderingContextXlib.cpp`. These files contain no shared clones in Mozilla releases 0.9.7 and 1.0 but in 1.3a they share about 20 % of their source code. Other examples can be found in Appendix B.

5.6.2 Visualization

The reduction of the various factors influencing the impact of code clones on change couplings is problematic. This is shown in the previous section. The visualization of the same factors is more meaningful to the human observer. The visualization technique described in 4.2.6 has been applied to several modules of Mozilla.

The scope of the colors and sizes of dots used in a chart is limited to the chart they appear in. Because of that the figures should not be compared among each other – the results would not be meaningful.

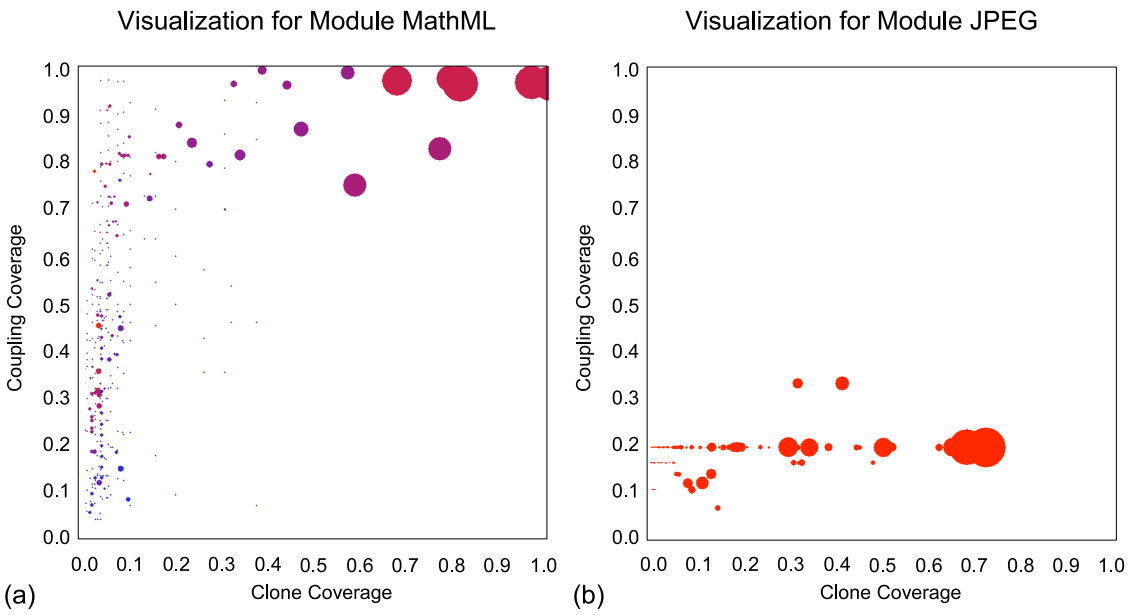


Figure 5.11: Visualization of Mozilla modules MathML and JPEG (Release 1.7).

Figure 5.11(a) shows the situation for the MathML module of Mozilla release 1.7. Of particular interest for a reengineering effort would be the 5 files shown in the upper right corner. They are very often coupled with other files and share large fragments of duplicated code. The module consists of 26 C++ files between which 470 distinct pairs of files sharing code exist.

The situation shown in Figure 5.11(b) is very different. The 52 C files of module JPEG form 230 distinct pairs which share code. The dots are all equally red because every file of the module was coupled exactly once during the period covered in this chart. In this case, the selection of candidates for a refactoring would have to rely on the length of code clones and the clone coverage alone.

These visualizations do not allow to automatically rank the code clones by the degree of maintenance problems they are likely to cause. However, a software engineer can get an overview and decide which files are likely candidates for a refactoring. To facilitate this decision further, the charts would need to be annotated with additional information about the file a dot represents. Also, additional charts showing what color and size corresponds to which amount of couplings respectively length of clones.

5.7 Implementation of a Prototype Tool

It has been decided that the implementation should not provide a stand–alone tool. A collection of library classes implementing certain aspects of the framework has been considered more useful.

5.7.1 Structure of the Library Classes

The implementation of the framework upon which this case study is based consists of several classes considered to be library classes for inclusion into other larger tools. The five functions

presented in Section 4.2.7 are divided into specialized classes implemented with Java 5.0. Several additional classes are provided in order to model the environment or to provide support.

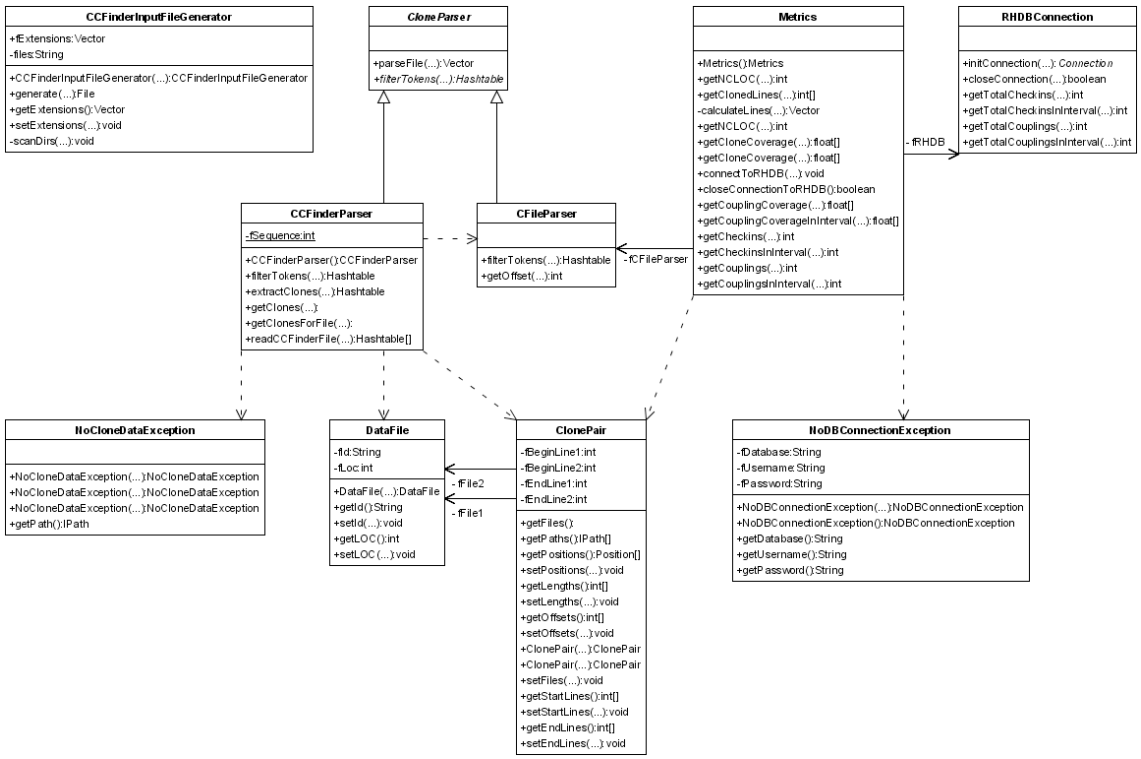


Figure 5.12: UML class diagram showing the implemented library classes.

The classes are:

- **CCFinderInputFileGenerator** offers functionality to create an input file for CCFinder,
- **CloneParser** is an abstract class to read in and tokenize a textfile,
- **CCFinderParser** is an implementation of `CloneParser` to read in a CCFinder output file and extract information about code clones and the affected files,
- **CFileParser** is an implementation of `CloneParser` to read in a C or C++ file and provide functionality to filter it for irrelevant lines of code,
- **Metrics** provides methods to calculate the various metrics described in the framework,
- **RHDBConnection** manages the connection and queries to the RHDB MySQL database,
- **ClonePair** provides an internal implementation of a clone pair,
- **DataFile** models a file with additional information added,
- **NoCloneDataException** is an `Exception` to handle problems caused by the processing of code clone data, and
- **NoDBConnectionException** is an `Exception` to be thrown if no connection to the database has been established.

The dependencies of the various classes are shown in UML diagram 5.12. For the sake of clarity, the method parameters have been omitted in the diagram.

5.7.2 Usage

The classes have been developed using Java 5.0 and can be used like any other Java classes. Their interfaces are described in standard Javadoc format and are therefore not described in detail in this document.

Two requirements exist that are currently not implemented in the library files. The first is the possible need for an SSH–tunnel in case the release history database is accessed from outside its network. This connection will have to be established prior to connecting to the RHDB by an external program.

The second requirement concerns CCFinder. Because the source code of this clone detection tool is not available, it could not be integrated to generate information about duplications automatically. It is therefore required to generate a standard CCFinder output file for further processing the code clone data. This file may not be altered manually because otherwise, the correct work of the library classes cannot be guaranteed.

For larger calculations, the classes are very memory and time consuming. We recommend to allocate a Java heap space of more than 500 MB.

Chapter 6

Conclusion

6.1 Contribution

This thesis presents a survey of the impact of code clones to change couplings and therefore the maintainability of a industrial–scale software system. That such a relationship exists has been postulated for several years but it has not been quantified yet.

The following points are concrete results of the development of the framework and of its application to the case study:

- A framework for the correlation of code clones and change couplings has been developed,
- Different code clone detection approaches and tools have been evaluated,
- Metrics for the evaluation of code clones in relation to change couplings have been proposed,
- A visualization technique for these metrics has been proposed, and
- A set of library classes implementing certain aspects of the framework have been produced.

An approach to finding a correlation between code clones and change couplings has been developed that is not limited in its use to the case study to which it has been applied – it is essentially independent of the type of system or of the programming language in which the system is written. It defines a framework that can be used in other cases with a much reduced amount of effort and time.

The evaluation of the code clone detection tools was conducted with respect to availability, scalability and above all usability. These three parameters are instrumental for the applicability of these tools to large scale software systems and the subsequent further processing of the generated output data.

The metrics defined are relatively simple and all of them need access to the system's source code and to a database containing historical release and modification data based for example on a version control system. A mathematical and quantifiable relation between code clones and change couplings could not be established because of the random character of this supposed connection in the case study. Therefore the final attempt at assessing the danger inherent to a code clone based on historical data and information generated by detection tools is rudimentary at best and possibly inadequate for other applications.

The visualization technique that has been developed offers a different and probably more practicable approach to the detection of problematic code clones. If this method could be imple-

mented and embedded into a software reengineering environment, it could potentially offer a useful guidance in the decision which clones are to be refactored.

The implemented library classes offer the possibility to automate large portions of the framework presented in this thesis. They were kept general enough to be useful in different environments.

The final result of this thesis is the implication that at least in this case study, the correlation between code clones and change couplings is too complex to be expressed trivially. For a significant distinction between clones that are irrelevant to the evolution of a system and clones that are harmful, more information is needed than what can be obtained automatically. Despite sophisticated tools that are available, the judgment of the software engineer is still needed.

6.2 Lessons Learned

Important insights have been gained in different areas: during the application of different tools and in a general overview of the relationship between clones and change couplings.

6.2.1 Tools

Clone Detection Tools

The evaluation of different code clone detection tools resulted in the selection of CCFinder and its visualization interface Gemini as the main program used for the case study. CCFinder was considered the most powerful of the available tools. It is also still being supported and development continues – during the six months of the case study, two new versions have been released. Shortly after the end of the period of this thesis, in October 2005 a major update named CCFinderX is expected to be released[1]. This update should further increase the usefulness of CCFinder.

CCFinder does also have some slight shortcomings. The different transformation rules described in Chapter 2 can supposedly be activated or deactivated individually for a detection run [Bel02]. The documentation does not cover this point and there does not seem to be a way to influence these rules from the Gemini user interface. The format of the output file is also not completely explained. Entries that seem to concern the aforementioned rules – this has been deduced that they concern CCFinder's mysterious `-r` option – are not mentioned.

Another problem concerns the visual output presented in Gemini. The tool does not offer a way to either export or print these graphs and metrics directly. Especially the metrics calculated by Gemini are for that reason lost for further processing.

These shortcomings should in no way diminish the fact, that CCFinder has been an excellent asset and a very well usable tool in the code clone detection domain. With the upcoming CCFinderX the tool also has the potential to become even better.

During the application of CCFinder to the case study, the standard options were found to be adequate. As in most other case studies using this detection tool, a minimal clone length of 30 tokens has been determined to give the best balance between fine granularity and understandability of the output. For very large input sizes (like entire releases of Mozilla), this threshold had to be raised to up to 70 tokens. The problem lies not in the actual detection process but in the visualization. Visualization of large input sizes needs a lot of RAM – even though the computer used for these evaluations was well supplied with memory, Gemini still runs out of Java heap space. If the allocated heap space exceeded 1,460 MB, the tool would not load properly anymore. Hopefully this shortcoming will be addressed in later releases. It has to be noted that it is still

[1] http://www.ccfinder.net

possible to run detections with a 30 token threshold and have to the output saved for further external processing – only the visualization with Gemini is impossible.

The other code detection tool selected for application to the case study was Duploc. However, this program developed a problem processing the Mozilla files: the linebreaks were ignored resulting in every file having a length of exactly one line. Because of the line–based approach to detection used in Duploc, no clones at all were detected in Mozilla. Duploc was also apt to terminate certain processes with Smalltalk error messages.

Duploc has the undeniable advantage that its source code is available and that a user proficient in Smalltalk can therefore adapt the tool to his needs or correct certain problems himself.

Release History Database

The use of the release history database was unproblematic because of the availability of expert advisers.

One challenge with the RHDB is that it has been developed further since the publication of the last description. Therefore, there is in some cases no explicit documentation of the meaning of certain fields (including for example exact definitions of time slots used in various tables).

A question has also arisen about the `cvsitemcoupling` table. Not all couplings that are expected after the examination of the corresponding `cvsitemlog` entries are present.

The last finding might offer an explanation of the former question, although this has not been examined. It is possible for two files to be coupled more than once during a certain time slot if one of them is checked in twice. If it is desirable for files to be potentially able to have more couplings than check–ins depends on the intention of the database and is not explored further within this thesis.

6.2.2 Correlation between Clones and Change Couplings

The long suspected relation between the amount of code clones present in a system and the subsequent change couplings during the evolution of the system could neither be proved nor disproved as a result of this case study. The results are too ambiguous and seemingly random.

For several instances, proof for a connection could be provided by manual inspection of all the changes. The problem with this approach is a matter of time. For a large system it is certainly not possible to do such a manual survey economically. If however suitable techniques – *e.g.*, visualization – are applied, the candidates for further manual inspection and eventual refactorings can be reduced to a more manageable size.

A final observation concerning code clones has been made during the implementation of the library classes: despite the fact that code duplication and all their problems have been in our mind permanently for half a year, it seems impossible to write even simple source code completely devoid of code clones.

6.3 Future Work

This thesis has proposed a framework for the evaluation of the impact of code clones on change couplings and software evolution. It has been applied to the Mozilla project but it remains to be seen if it is also applicable to other case studies developed in a different way. Possibly such case studies produce more generally applicable predictions for the danger inherent to a certain clone.

To completely automate the examination of the relationship between code duplications and change couplings it should be possible to access information about what part of a system has been changed in order to determine if the coupling was indeed caused by a clone. This information is

not obtainable from CVS but would greatly simplify the maintenance of a system. Therefore an implementation of such a feature into CVS would be desirable.

The metric for the danger inherent to a clone has only been defined sketchily in this thesis. It should be tested in larger case studies in order to determine its significance and the best values of the parameters not defined here.

Finally the embedding of the library classes implemented in course of this thesis could enhance the efficiency of making decisions about where to apply refactoring techniques to a system during its evolution. To achieve this goal, it would be necessary to automate the framework as far as possible to relieve the maintenance engineer's workload. A combination of the described visualization technique with the possibility of source code browsing implemented to work with a major integrated development environment could provide a powerful tool do assess the potential problems source code clones can cause in a system.

The development and maintenance of software is a difficult and very important field of engineering due to the continued computerization of the world. If this thesis could trigger some train of thought that ultimately helps to make it a little bit less complex, its goal has been achieved.

Appendix A

List of Required Tools

- **CCFinder**: http://www.ccfinder.net
- **Gemini**: http://www.ccfinder.net
- **Duploc**: http://www.iam.unibe.ch/ rieger/duploc/ (as of July 2005)
- **CVS**: http://www.cvshome.org
- **VisualWorks 3.0**: http://smalltalk.cincom.com
- **Java 5.0**: http://java.sun.com

Appendix B

Calculated Clone and Coupling Values

	0.9.2	0.9.7	1	1.3a	1.4	1.6	1.7	0.9.2	0.9.7	1	1.3a	1.4	1.6	1.7	<1.7
	Cloned Lines / NCLOC							Couplings / Total Checkins							
widget/src/os2/nsFilePicker.cpp	0.8	0.59	0.43	0.37	0.28	0.27	0.27	0.5	0.57	0.29	0.2	0.25	0	0.14	0.3
widget/src/windows/nsFilePicker.cpp	0.72	0.67	0.62	0.4	0.42	0.42	0.4	0.5	0.33	0.38	0.08	0.2	0	0.25	0.21
	Cloned Lines							Couplings							
widget/src/os2/nsFilePicker.cpp	204	204	227	205	172	172	172	3	4	6	1	1	0	1	18
widget/src/windows/nsFilePicker.cpp	207	207	225	239	169	169	169	3	4	6	1	1	0	1	18
	NCLOC							Total Checkins							
widget/src/os2/nsFilePicker.cpp	255	345	526	557	613	649	643	6	7	21	5	4	4	7	61
widget/src/windows/nsFilePicker.cpp	288	308	362	591	400	401	427	6	12	16	12	5	5	4	87
	Cloned Lines / NCLOC							Couplings / Total Checkins							
intl/ctl/src/nsUnicodetoTIS620.cpp	-	-	-	0.85	0.85	0.88	0.37	-	-	-	0.6	0.5	1	1	0.75
intl/ctl/src/nsUnicodeToSunIndic.cpp	-	-	-	0.99	0.98	0.98	0.5	-	-	-	1	1	1	1	1
	Cloned Lines							Couplings							
intl/ctl/src/nsUnicodetoTIS620.cpp	0	0	0	165	163	163	36	0	0	0	3	1	3	2	9
intl/ctl/src/nsUnicodeToSunIndic.cpp	0	0	0	170	168	168	45	0	0	0	3	1	3	2	9
	NCLOC							Total Checkins							
intl/ctl/src/nsUnicodetoTIS620.cpp	0	0	0	193	192	185	97	0	0	0	5	2	3	2	12
intl/ctl/src/nsUnicodeToSunIndic.cpp	0	0	0	172	171	171	90	0	0	0	3	1	3	2	9
	Cloned Lines / NCLOC							Couplings / Total Checkins							
intl/uconv/src/nsWinCharset.cpp	0.51	0.07	0.07	0.07	0.07	0.08	0.08	1	0.88	0.67	0.67	0.5	1	1	0.68
intl/uconv/src/nsMacCharset.cpp	0.49	0.08	0.08	0.07	0.07	0.06	0.06	0.4	0.88	0.67	0.67	0.2	0.5	1	0.52
	Cloned Lines							Couplings							
intl/uconv/src/nsWinCharset.cpp	56	9	9	9	9	9	9	2	7	2	2	1	3	1	25
intl/uconv/src/nsMacCharset.cpp	58	9	9	9	9	9	9	2	7	2	2	1	3	1	25
	NCLOC							Total Checkins							
intl/uconv/src/nsWinCharset.cpp	109	123	123	123	122	120	120	2	8	3	3	2	3	1	37
intl/uconv/src/nsMacCharset.cpp	119	119	119	132	136	142	142	5	8	3	3	5	6	1	48
	Cloned Lines / NCLOC							Couplings / Total Checkins							
gfx/src/os2/nsRenderingContextOS2.cpp	0.21	0.3	0.35	0.35	0.35	0.44	0.44	0.1	0.17	0.36	0.71	0.56	0	0.67	0.3
gfx/src/windows/nsRenderingContextWin.cpp	0.12	0.26	0.32	0.32	0.32	0.37	0.37	0.07	0.09	0.31	0.63	0.83	0	0.8	0.14
	Cloned Lines							Couplings							
gfx/src/os2/nsRenderingContextOS2.cpp	291	558	678	684	668	775	775	2	2	5	5	5	0	4	32
gfx/src/windows/nsRenderingContextWin.cpp	322	555	688	694	678	769	769	2	2	5	5	5	0	4	32
	NCLOC							Total Checkins							
gfx/src/os2/nsRenderingContextOS2.cpp	1355	1836	1960	1960	1890	1770	1750	21	12	14	7	9	1	6	107
gfx/src/windows/nsRenderingContextWin.cpp	2623	2165	2182	2181	2113	2070	2070	27	22	16	8	6	5	5	224
	Cloned Lines / NCLOC							Couplings / Total Checkins							
gfx/src/windows/nsRenderingContextWin.cpp	0.02	0	0	0.2	0.21	0.21	0.21	0.07	0.18	0.44	0.63	0.67	0	0.8	0.17
gfx/src/xlib/nsRenderingContextXlib.cpp	0.04	0	0	0.23	0.23	0.23	0.23	0.29	0.18	0.5	0.71	0.5	0	1	0.35
	Cloned Lines							Couplings							
gfx/src/windows/nsRenderingContextWin.cpp	55	0	0	444	445	438	438	2	4	7	5	4	0	4	39
gfx/src/xlib/nsRenderingContextXlib.cpp	57	0	0	444	445	438	438	2	4	7	5	4	0	4	39
	NCLOC							Total Checkins							
gfx/src/windows/nsRenderingContextWin.cpp	2623	2165	2182	2181	2113	2070	2070	27	22	16	8	6	5	5	224
gfx/src/xlib/nsRenderingContextXlib.cpp	1310	1398	1587	1939	1900	1900	1900	7	22	14	7	8	1	4	113
	Cloned Lines / NCLOC							Couplings / Total Checkins							
gfx/src/os2/nsRenderingContextOS2.cpp	0	0	0	0.2	0.2	0.25	0.25	0.1	0.17	0.36	0.71	0.44	0	0.67	0.29
gfx/src/xlib/nsRenderingContextXlib.cpp	0	0	0	0.2	0.2	0.23	0.23	0.29	0.09	0.36	0.71	0.5	0	1	0.27
	Cloned Lines							Couplings							
gfx/src/os2/nsRenderingContextOS2.cpp	0	0	0	383	384	439	439	2	2	5	5	4	0	4	31
gfx/src/xlib/nsRenderingContextXlib.cpp	0	0	0	383	384	438	438	2	2	5	5	4	0	4	31
	NCLOC							Total Checkins							
gfx/src/os2/nsRenderingContextOS2.cpp	1355	1836	1960	1960	1890	1770	1750	21	12	14	7	9	1	6	107
gfx/src/xlib/nsRenderingContextXlib.cpp	1310	1398	1587	1939	1900	1900	1900	7	22	14	7	8	1	4	113
	Cloned Lines / NCLOC							Couplings / Total Checkins							
content/html/style/src/nsCSSStyleSheet.cpp	0.03	0.02	0.02	0.01	0	0	0	0.13	0.24	0.22	0.48	0.21	0.07	0.17	0.24
content/html/style/src/nsHTMLStyleSheet.cpp	0.07	0.1	0.09	0.07	0.02	0.01	0.01	0.55	0.44	0.67	0.65	0.42	0.33	0.17	0.27
	Cloned Lines							Couplings							
content/html/style/src/nsCSSStyleSheet.cpp	103	80	80	42	17	12	12	6	7	8	11	5	2	1	81
content/html/style/src/nsHTMLStyleSheet.cpp	76	107	107	77	19	14	14	6	7	8	11	5	2	1	81
	NCLOC							Total Checkins							
content/html/style/src/nsCSSStyleSheet.cpp	3150	3502	3519	3622	3426	3386	3391	47	29	36	23	24	28	6	342
content/html/style/src/nsHTMLStyleSheet.cpp	1099	1053	1240	1161	1009	988	943	11	16	12	17	12	6	6	300

	0.9.2	0.9.7	1	1.3a	1.4	1.6	1.7	0.9.2	0.9.7	1	1.3a	1.4	1.6	1.7	<1.7
	Cloned Lines / NCLOC							Couplings / Total Checkins							
content/base/src/nsGeneratedIterator.cpp	0.6	0.53	0.53	0.45	0.44	0.4	0.31	1	1	1	1	1	1	1	0.89
content/base/src/nsContentIterator.cpp	0.77	0.55	0.55	0.42	0.4	0.37	0.27	0.5	0.2	0.33	0.75	0.8	1	0.5	0.24
	Cloned Lines							Couplings							
content/base/src/nsGeneratedIterator.cpp	556	492	492	403	400	357	268	1	1	1	3	4	2	2	17
content/base/src/nsContentIterator.cpp	602	523	523	441	433	387	267	1	1	1	3	4	2	2	17
	NCLOC							Total Checkins							
content/base/src/nsGeneratedIterator.cpp	934	934	934	898	905	898	873	1	1	1	3	4	2	2	19
content/base/src/nsContentIterator.cpp	777	957	955	1058	1071	1042	1005	2	5	3	4	5	2	4	72
	Cloned Lines / NCLOC							Couplings / Total Checkins							
layout/xul/base/src/nsMenuPopupFrame.cpp	0.21	0.17	0.13	0.12	0.12	0.11	0.1	0.14	0.22	0.3	0.1	0.17	0.42	0.5	0.31
layout/xul/base/src/nsMenuBarFrame.cpp	0.5	0.5	0.43	0.39	0.37	0.35	0.3	0.5	0.56	0.67	0.25	0.2	1	0.8	0.57
	Cloned Lines							Couplings							
layout/xul/base/src/nsMenuPopupFrame.cpp	224	224	196	196	196	168	145	2	5	10	2	2	8	4	79
layout/xul/base/src/nsMenuBarFrame.cpp	245	245	218	213	213	194	164	2	5	10	2	2	8	4	79
	NCLOC							Total Checkins							
layout/xul/base/src/nsMenuPopupFrame.cpp	1065	1351	1494	1599	1615	1532	1505	14	23	33	21	12	19	8	252
layout/xul/base/src/nsMenuBarFrame.cpp	489	490	505	553	572	558	542	4	9	15	8	10	8	5	139
	Cloned Lines / NCLOC							Couplings / Total Checkins							
widget/src/mac/nsMenuItemX.cpp	0.58	0.43	0.43	0.43	0.43	0.37	0.37	0.67	0.63	1	1	0.67	0.8	1	0.75
widget/src/mac/nsMenuItem.cpp	0.58	0.37	0.37	0.37	0.37	0.32	0.32	1	0.83	1	1	0.67	1	1	0.27
	Cloned Lines							Couplings							
widget/src/mac/nsMenuItemX.cpp	160	101	101	101	100	87	87	2	5	2	1	2	4	1	18
widget/src/mac/nsMenuItem.cpp	160	101	101	101	100	87	87	2	5	2	1	2	4	1	18
	NCLOC							Total Checkins							
widget/src/mac/nsMenuItemX.cpp	274	234	234	234	233	238	238	3	8	2	1	3	5	1	24
widget/src/mac/nsMenuItem.cpp	274	274	274	274	273	268	268	2	6	2	1	3	4	1	66
	Cloned Lines / NCLOC							Couplings / Total Checkins							
jpeg/djpeg.c	0.58	0.58	0.58	0.58	0.58	0.58	0.58	-	-	1	-	-	-	-	0.2
jpeg/cjpeg.c	0.64	0.64	0.64	0.64	0.64	0.64	0.64	-	-	1	-	-	-	-	0.2
	Cloned Lines							Couplings							
jpeg/djpeg.c	223	223	223	223	223	223	223	0	0	1	0	0	0	0	1
jpeg/cjpeg.c	258	258	258	258	258	258	258	0	0	1	0	0	0	0	1
	NCLOC							Total Checkins							
jpeg/djpeg.c	383	383	383	383	383	383	383	0	0	1	0	0	0	0	5
jpeg/cjpeg.c	406	406	406	406	406	406	406	0	0	1	0	0	0	0	5
	Cloned Lines / NCLOC							Couplings / Total Checkins							
editor/libeditor/html/nsHTMLDataTransfer.cpp	0.35	0.32	0.31	0.26	0.18	0.1	0.09	0.4	0.56	0.19	0.24	0.38	0.22	0.14	0.35
editor/libeditor/text/nsPlaintextDataTransfer.cpp	1	1.03	0.98	0.98	0.68	0.44	0.42	0.89	1	1	0.75	0.94	0.67	0.5	0.89
	Cloned Lines							Couplings							
editor/libeditor/html/nsHTMLDataTransfer.cpp	474	498	474	474	345	183	183	8	14	3	6	15	2	1	49
editor/libeditor/text/nsPlaintextDataTransfer.cpp	408	445	421	403	283	168	168	8	14	3	6	15	2	1	49
	NCLOC							Total Checkins							
editor/libeditor/html/nsHTMLDataTransfer.cpp	1350	1535	1527	1840	1921	1914	2130	20	25	16	25	39	9	7	142
editor/libeditor/text/nsPlaintextDataTransfer.cpp	408	430	428	411	414	385	397	9	14	3	8	16	3	2	55
	Cloned Lines / NCLOC							Couplings / Total Checkins							
gfx/src/xlib/nsFontMetricsXlib.cpp	0.58	0.63	0.82	0.83	0.58	0.57	0.57	0.18	0.57	0.82	0.87	0.81	0.82	0.5	0.62
gfx/src/gtk/nsFontMetricsGTK.cpp	0.55	0.57	0.87	0.73	0.52	0.51	0.51	0.12	0.41	0.82	0.9	0.81	0.64	0.6	0.39
	Cloned Lines							Couplings							
gfx/src/xlib/nsFontMetricsXlib.cpp	1761	2095	3433	3715	2640	2616	2617	3	21	18	26	25	9	3	119
gfx/src/gtk/nsFontMetricsGTK.cpp	1771	2076	3397	3609	2565	2549	2549	3	21	18	26	25	9	3	119
	NCLOC							Total Checkins							
gfx/src/xlib/nsFontMetricsXlib.cpp	3033	3304	4189	4450	4564	4582	4605	17	37	22	30	31	11	6	192
gfx/src/gtk/nsFontMetricsGTK.cpp	3211	3625	3911	4945	4978	5002	5025	25	51	22	29	31	14	5	305
	Cloned Lines / NCLOC							Couplings / Total Checkins							
layout/mathml/base/src/nsMathMLmunderFrame.cpp	0.68	0.68	0.74	0.76	0.76	0.76	0.76	1	1	1	1	1	1	1	1
layout/mathml/base/src/nsMathMLMunderoverFrame.cpp	0.69	0.69	0.71	0.73	0.64	0.65	0.65	1	1	1	1	1	1	1	1
	Cloned Lines							Couplings							
layout/mathml/base/src/nsMathMLmunderFrame.cpp	152	152	165	165	168	168	168	3	3	12	3	3	2	2	43
layout/mathml/base/src/nsMathMLMunderoverFrame.cpp	207	207	203	203	193	193	193	3	3	12	3	3	2	2	43
	NCLOC							Total Checkins							
layout/mathml/base/src/nsMathMLmunderFrame.cpp	223	223	223	217	222	221	221	3	3	12	3	3	2	2	43
layout/mathml/base/src/nsMathMLMunderoverFrame.cpp	299	299	284	278	300	299	299	3	3	12	3	3	2	2	43
	Cloned Lines / NCLOC							Couplings / Total Checkins							
layout/mathml/base/src/nsMathMLMoverFrame.cpp	0.82	0.82	0.84	0.86	0.96	0.97	0.97	1	1	1	1	1	1	1	1
layout/mathml/base/src/nsMathMLMunderoverFrame.cpp	0.76	0.76	0.76	0.78	0.77	0.77	0.77	1	1	1	1	1	1	1	1
	Cloned Lines							Couplings							
layout/mathml/base/src/nsMathMLMoverFrame.cpp	202	202	208	208	225	225	225	3	3	12	3	3	2	2	43
layout/mathml/base/src/nsMathMLMunderoverFrame.cpp	226	226	216	216	231	231	231	3	3	12	3	3	2	2	43
	NCLOC							Total Checkins							
layout/mathml/base/src/nsMathMLMoverFrame.cpp	247	247	247	241	234	233	233	3	3	12	3	3	2	2	43
layout/mathml/base/src/nsMathMLMunderoverFrame.cpp	299	299	284	278	300	299	299	3	3	12	3	3	2	2	43
	Cloned Lines / NCLOC							Couplings / Total Checkins							
layout/mathml/base/src/nsMathMLmunderFrame.cpp	0.9	0.9	0.95	0.97	0.96	0.97	0.97	1	1	1	1	1	1	1	1
layout/mathml/base/src/nsMathMLMoverFrame.cpp	0.83	0.83	0.86	0.88	0.92	0.93	0.93	1	1	1	1	1	1	1	1
	Cloned Lines							Couplings							
layout/mathml/base/src/nsMathMLmunderFrame.cpp	200	200	211	211	214	214	214	3	3	12	3	3	2	2	43
layout/mathml/base/src/nsMathMLMoverFrame.cpp	204	204	213	213	216	216	216	3	3	12	3	3	2	2	43
	NCLOC							Total Checkins							
layout/mathml/base/src/nsMathMLmunderFrame.cpp	223	223	223	217	222	221	221	3	3	12	3	3	2	2	43
layout/mathml/base/src/nsMathMLMoverFrame.cpp	247	247	247	241	234	233	233	3	3	12	3	3	2	2	43

	0.9.2	0.9.7	1	1.3a	1.4	1.6	1.7	0.9.2	0.9.7	1	1.3a	1.4	1.6	1.7	<1.7
	Cloned Lines / NCLOC							Couplings / Total Checkins							
mailnews/import/outlook/src/nsOutlookCompose.cpp	1.12	1.11	1.06	0.99	0.99	0.99	0.99	0.85	1	0.71	0.5	1	-	1	0.86
mailnews/import/eudora/src/nsEudoraCompose.cpp	1.05	1.03	1.02	0.95	0.95	0.96	0.95	0.92	0.67	0.67	1	1	-	1	0.75
	Cloned Lines							Couplings							
mailnews/import/outlook/src/nsOutlookCompose.cpp	912	911	902	841	840	841	841	11	8	10	2	2	0	3	48
mailnews/import/eudora/src/nsEudoraCompose.cpp	912	909	913	852	851	852	852	11	8	10	2	2	0	3	48
	NCLOC							Total Checkins							
mailnews/import/outlook/src/nsOutlookCompose.cpp	817	822	848	847	846	846	847	13	8	14	4	2	0	3	56
mailnews/import/eudora/src/nsEudoraCompose.cpp	866	880	895	893	892	892	894	12	12	15	2	2	0	3	64
	Cloned Lines / NCLOC							Couplings / Total Checkins							
	0.9.2	0.9.7	1	1.3a	1.4	1.6	1.7	0.9.2	0.9.7	1	1.3a	1.4	1.6	1.7	<1.7
layout/mathml/base/src/nsMathMLmsupFrame.cpp	0.56	0.56	0.42	0.44	0.38	0.38	0.38	1	1	1	1	1	1	1	1
layout/mathml/base/src/nsMathMLmsubFrame.cpp	0.62	0.62	0.47	0.5	0.43	0.43	0.43	1	1	1	1	1	1	1	0.97
	Cloned Lines							Couplings							
layout/mathml/base/src/nsMathMLmsupFrame.cpp	105	105	73	73	63	63	63	3	1	4	1	2	2	1	30
layout/mathml/base/src/nsMathMLmsubFrame.cpp	105	105	73	73	63	63	63	3	1	4	1	2	2	1	30
	NCLOC							Total Checkins							
layout/mathml/base/src/nsMathMLmsupFrame.cpp	189	189	174	167	166	166	165	3	1	4	1	2	2	1	30
layout/mathml/base/src/nsMathMLmsubFrame.cpp	169	169	154	147	146	146	145	3	1	4	1	2	2	1	31
	Cloned Lines / NCLOC							Couplings / Total Checkins							
layout/mathml/base/src/nsMathMLmsubsupFrame.cpp	0.31	0.31	0.2	0.21	0.21	0.21	0.21	1	1	0.67	0.33	1	1	1	0.88
layout/mathml/base/src/nsMathMLmsubFrame.cpp	0.45	0.45	0.31	0.32	0.32	0.32	0.32	1	1	1	1	1	1	1	0.97
	Cloned Lines							Couplings							
layout/mathml/base/src/nsMathMLmsubsupFrame.cpp	74	74	45	45	45	45	45	3	1	4	1	2	2	1	30
layout/mathml/base/src/nsMathMLmsubFrame.cpp	76	76	47	47	47	47	47	3	1	4	1	2	2	1	30
	NCLOC							Total Checkins							
layout/mathml/base/src/nsMathMLmsubsupFrame.cpp	238	238	225	218	216	217	216	3	1	6	3	2	2	1	34
layout/mathml/base/src/nsMathMLmsubFrame.cpp	169	169	154	147	146	146	145	3	1	4	1	2	2	1	31
	Cloned Lines / NCLOC							Couplings / Total Checkins							
layout/mathml/base/src/nsMathMLmsupFrame.cpp	0.52	0.52	0.53	0.55	0.55	0.55	0.56	1	1	1	1	1	1	1	1
layout/mathml/base/src/nsMathMLmsubsupFrame.cpp	0.47	0.47	0.44	0.45	0.46	0.46	0.46	1	1	0.67	0.33	1	1	1	0.88
	Cloned Lines							Couplings							
layout/mathml/base/src/nsMathMLmsupFrame.cpp	99	99	92	92	92	92	92	3	1	4	1	2	2	1	30
layout/mathml/base/src/nsMathMLmsubsupFrame.cpp	112	112	99	99	99	99	99	3	1	4	1	2	2	1	30
	NCLOC							Total Checkins							
layout/mathml/base/src/nsMathMLmsupFrame.cpp	189	189	174	167	166	166	165	3	1	4	1	2	2	1	30
layout/mathml/base/src/nsMathMLmsubsupFrame.cpp	238	238	225	218	216	217	216	3	1	6	3	2	2	1	34
	Cloned Lines / NCLOC							Couplings / Total Checkins							
content/svg/content/src/nsSVGLineElement.cpp	-	0.97	0.97	0.97	0.97	0.97	1.12	-	1	1	1	-	-	1	1
content/svg/content/src/nsSVGRectElement.cpp	-	0.95	0.95	0.95	0.95	0.95	1.09	-	1	1	1	-	-	1	1
	Cloned Lines							Couplings							
content/svg/content/src/nsSVGLineElement.cpp	0	171	171	171	171	171	232	0	5	1	1	0	0	6	13
content/svg/content/src/nsSVGRectElement.cpp	0	208	208	208	208	208	291	0	5	1	1	0	0	6	13
	NCLOC							Total Checkins							
content/svg/content/src/nsSVGLineElement.cpp	0	176	176	176	176	176	208	0	5	1	1	0	0	6	13
content/svg/content/src/nsSVGRectElement.cpp	0	220	220	220	220	220	266	0	5	1	1	0	0	6	13

References

[Bak92] Brenda S. Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, 24:49–57, 1992.

[Bak93] Brenda S. Baker. A Theory of Parameterized Pattern Matching: Algorithms and Applications. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 71–80, New York, NY, USA, 1993. ACM Press.

[BB02] Elizabeth Burd and John Bailey. Evaluating Clone Detection Tools for Use during Preventative Maintenance. In *SCAM '02: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, page 36, Washington, DC, USA, 2002. IEEE Computer Society.

[Bel02] Stefan Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Master's thesis, Universitaet Stuttgart, 2002.

[BPM04] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 625–634, Washington, DC, USA, 2004. IEEE Computer Society.

[BYM+98] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

[DRD99] Stphane Ducasse, Matthias Rieger, and Serge Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 109, Washington, DC, USA, 1999. IEEE Computer Society.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[FGP05] Beat Fluri, Harald C. Gall, and Martin Pinzger. Fine-grained analysis of change couplings. In *Proceedings of the 5th International Workshop on Source Code Analysis and Manipulation*, Budapest, Hungary, September 2005. IEEE Computer Society Press.

[FPG03] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM)*, pages 23–32, Amsterdam, The Netherlands, September 2003. IEEE, IEEE Computer Society.

[GHJ98] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of Logical Coupling Based on Product Release History. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 190, Washington, DC, USA, 1998. IEEE Computer Society.

[GJK03] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. CVS Release History Data for Detecting Logical Couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.

[GT03] Penny Grubb and Armstrong A. Takang. *Software Maintenance – Concepts and Practice*. World Scientific, 2nd edition, 2003.

[IEE93] IEEE Computer Society Press, Los Alamitos CA., USA. *IEEE Std. 1219: Standard for Software Maintenance*, 1993.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[Kri01] Jens Krinke. Identifying Similar Code with Program Dependence Graphs. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 301, Washington, DC, USA, 2001. IEEE Computer Society.

[KSNM05] Miryung Kim, Vibha Sazaval, David Notkin, and Gail C. Murphy. An Empirical Study of Code Clone Genealogies. To be released for ESEC/FSE, 2005.

[LB85] Meir M. Lehman and Laszlo Belady. *Program Evolution Processes of Software Change*. Academic Press, 1985.

[LD03] Michele Lanza and Stphane Ducasse. Polymetric Views – A Lightweight Visual Approach to Reverse Engineering. *IEEE Trans. Softw. Eng.*, 29(9):782–795, 2003.

[LPM+97] Bruno Lague, Daniel Proulx, Jean Mayrand, Ettore M. Merlo, and John Hudepohl. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, page 314, Washington, DC, USA, 1997. IEEE Computer Society.

[MLM96] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 244, Washington, DC, USA, 1996. IEEE Computer Society.

[RD98] Matthias Rieger and Stephane Ducasse. Visual Detection of Duplicated Code. In *ECOOP '98: Workshop on Object-Oriented Technology*, pages 75–76, London, UK, 1998. Springer-Verlag.

[UKKI02] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Maintenance Support Environment Based on Code Clone Analysis. In *METRICS '02: Proceedings of the 8th International Symposium on Software Metrics*, page 67, Washington, DC, USA, 2002. IEEE Computer Society.

[Wak03] William C. Wake. *Refactoring Workbook*. Addison-Wesley, 2003.

[ZWDZ04] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining Version Histories to Guide Software Changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.