Studying the Effects of Code Clone Size on Clone Evolution

Master's Thesis

Gosse Bouma

Studying the Effects of Code Clone Size on Clone Evolution

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Gosse Bouma
born in Amsterdam, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Studying the Effects of Code Clone Size on Clone Evolution

Author: Gosse Bouma
Student id: 1220217
Email: G.G.Bouma@Student.TUDelft.NL

Abstract

The practice of code cloning is something every software developer has to deal with at some point. The evolution of code clones is of particular interest, because the effects of cloning code show up later in the lifetime of a project. We research the effects a clone's properties have on its evolutionary behavior. For this purpose an approach to extract the clone size information from mined software repositories is shown. Using this approach an insight can be gained into how clone sizes evolve over time, as well as whether the size has an influence on other evolutionary patterns of a clone. We present our findings and conclude that clone size influences a clone's evolution in several ways. Finally a discussion and framework for future work on this subject is presented.

Preface

In this thesis I present the research I have conducted for my master's degree during 2011 - 2012 at the Technical University of Delft. The subject was suggested to me by my supervisor Andy Zaidman and it immediately sparked my interest. I would like to thank dr. Zaidman for suggesting the subject to me and also for his advice and guidance throughout the process of writing this thesis, which has helped me greatly. I would also like to thank my family for their support and for allowing me the opportunity to acquire a master's degree.

Gosse Bouma
Amsterdam, the Netherlands
June 26, 2012

Contents

List of Figures

Chapter 1

Introduction

The duplicating of code, also known as code cloning, is an important phenomenon in software development and engineering. There are a number of issues that arise when code clones are created during the development of a software project. This report discusses the findings of research on the evolution of code clones over a project's lifespan, done by software repository mining. To introduce the subjects discussed in this report they will first be placed in context. After that, the problem that our research focuses on is stated and a number of research questions that will be answered throughout the report is introduced. In the last section of this introduction chapter an outline for the rest of the report is given.

1.1 Context

Every software developer has done it: copy an existing piece of code and paste it to replicate a certain functionality. When this is done, a relation between the copy and pasted code segment is established and the pair of code segments becomes a so-called *code clone*. The phenomenon of code clones is not uncommon [20, 22]. In fact, it is almost inescapable to use them in any non-trivial software project. Empirical studies of software projects suggest that on average a typical software project contains - depending on certain factors such as the programming language used - 10 to 20% cloned code [30, 20, 22]. Extreme cases can even consist of more than 50% duplicated code. These figures depend also heavily on the definition of a code clone. A distinction should be made between exact clones and so-called inconsistent clones [25], which are not 100% exact clones but might have slight alterations. A precise definition used throughout this report, as well as further explanation of the terms mentioned above and a complete taxonomy of clone types, will be given in chapter 2.

Despite being widespread, the introduction of clones to software projects can cause a number of issues. Most importantly, in evolving software, that is to say software that changes over time, code clones can result in so-called inconsistent changes [16]. An inconsistent change happens when one segment of code is changed, but its cloned counterpart is not. This phenomenon can cause unexpected behavior in the software and introduce bugs. Another important issue that arises because of code clones is the fact that the cost of maintaining the code increases [16]. Code clones can increase the size of the program, making it less transparent and thus harder to understand. In addition, if the maintainers of the software - which are often not the same persons as the initial developers - are not aware of the code clones, they can introduce inconsistent changes too. Another issue with code clones is that they are often the result of violating design patterns, they are an example of so-called ‘code smells’ [10]. While this does not necessarily cause problems immediately, it can be an indicator for poorly designed software which is a warning signal for the developers of the software.

Aside from these issues, there are also advantages of using code clones [21]. For example, when a new feature is to be introduced in the software that is similar to an existing feature, it can be greatly beneficial to introduce a clone to replicate the functionality of the existing feature. The clone can then be developed independently and changed according to the requirements of the new feature.

Now that it is established that developers will sooner or later re-use an existing piece of code to, for example, quickly introduce an already implemented functionality in another place in the software. There are a number of reasons why this happens. Most importantly, it is often quicker, and thus seemingly cheaper, to simply copy and paste a code fragment, rather than writing an abstraction that creates the same functionality without having duplicate code.

Developers might introduce clones with the intention of keeping and managing this code clone permanently, or with the intention of rewriting it later. In the second case, the developer takes on what we call *technical debt* [23]. Technical debt is taken on by creating code that violates certain software engineering or development patterns, established to aid in creating understandable and maintainable code. This means he “borrows” time from the future (when he has to rewrite the code clone) to gain a productivity increase in the present. Taking on technical debt can be beneficial for software developers in certain cases, for example to quickly produce a working program to be released. However, taking on too much debt - just like with real monetary debt - is counterproductive because the software becomes unmaintainable. For this reason the debt has to be serviced and the violating code has to be refactored to adhere to patterns so that the code remains understandable and thus easier maintainable.

Another reason why a developer might introduce clones to the system is because he has the solution to a certain problem in his memory from implementing earlier. This might be done unconsciously, and the clone is introduced inadvertently [5]. Sometimes clones are introduced because of language constraints, or because the language has better performance using clones.

If the developer does not intend to rewrite the clone at a later time then it must be managed in some way. In order for the clone not to be inconsistently changed, the developer needs to be able to keep track of all the pairs of clones somehow. If there are only a few pairs it might be possible for the developer to remember them by heart, but when there are more pairs and they are scattered all over the project, it becomes necessary to help the developer automatically track these clone pairs somehow.

1.2 Research Questions

The research presented in this report aims to provide further insight into the evolution of the relation between code clones over a project's lifetime. The ultimate goal of this research is to find patterns in the evolution of code clones. With these patterns it is then possible to gain better knowledge of when code clones cause problems and whether there is a point for developers where using tools to help track clones becomes necessary. To do this research, we need a way to be able to observe the evolution of a software project. We will create a tool that will aid us with this goal. The tool will use a technique called software repository mining to extract all revisions from the project's lifespan from a repository. It will then perform clone detection on each revision to find clones and write information about them and their relations to a database. The information in this database can then be read and visualized such that it becomes understandable. Using this visualized data we can gain insight into the evolutionary patterns of the clones over several revisions.

To guide the research process we have formulated some research questions. The ultimate goal of this thesis is to shed light upon the influence that a clone's properties, most importantly and firstly a clone's size, have on the evolution of a clone. To reach this goal we propose some research questions.

"How does the size of clones evolve over time?"

The size of a clone is an important figure to gauge its complexity, and it essentially refers to the number of times a code segment is cloned. Before we can answer this question however, we must find out the state of the art techniques in the area of software repository mining. Then we can find how clone information can be extracted from the mined repositories, how this information should be stored and eventually how we can visualize this data in a meaningful way.

The main research goal of this thesis is quite broad, in the sense that there might be many aspects that can be learned from the evolution of code clones. To narrow down the aspects we want to study from the evolution of code clones, we have formulated some more specific research questions. Firstly, we ask the question:

"Is there an affinity between the clone size and its probability of being removed?"

We want to find out whether there is a limit to how large a clone a developer can keep track of before refactoring it. This can help determine whether there is a need for automated clone tracking at a certain clone size. When answering this question it is important to research not only if a clone has been removed but also the reason why. It might be possible that the removal of the clone and the size of its clone group are unrelated, for example, simply because a certain functionality is no longer required in the project.

The next sub-question is quite similar to the previous one:

"Is there an affinity between the clone size and the probability of clone fragments being removed from that clone?"

For this question we can use the same techniques as the previous one, but instead of checking whether a clone has been removed entirely, we check whether a fragment was removed. We want to know whether the size of a clone has any connection with fragments of the clone being refactored. When a fragment of a clone is being refactored it might be an indication that the developer can no longer keep track of all the fragments of a clone. Alternatively, when a clone fragment is removed it might be by accident and the other fragments were meant to be changed as well, indicating there is another problem with keeping track of the clones.

1.3 Outline

The rest of this report is structured as follows: In chapter 2 the related research will be discussed. This chapter is divided into several sections each discussing research related to a seperate subject. In chapter 3 the implementation of the tool used for our research is discussed. Then in chapter 4 we will present and explain the results of our research, allowing us to answer our research questions. Finally in chapter 5 the research questions will be answered and a final conclusion will be drawn. In this chapter we will also provide a discussion about the conclusions and potential future work that can be done in this area of research.

Chapter 2

Related Work

Code clones are a widely researched phenomenon in computer science. Particularly in the area of software evolution they play an important role. In this chapter we will discuss literature related to the research presented in this report. To do this effectively, we have divided related work in the following fields of interest: code clone detection, code clone genealogy and evolution, software repository mining. Dividing the literature in these categories is logical since the research presented in this report builds on existing research in these areas. A context for our research will be created and knowledge about the state of the art techniques will be gained.

Before the matters mentioned above can be discussed, the terminology regarding code clones needs to be defined clearly. Hence this chapter will be ordered as follows. In section 2.1 we will discuss the terminology surrounding code clones and place it in context of the study of this report. Next, in section 2.2 we will present literature about clone detection techniques and motivate a choice of technique used for the research. In section 2.3 we will talk about the evolution of code clones throughout the lifetime of projects. Finally in section 2.4 the research area of software repository mining will be discussed and the state of the art techniques presented.

2.1 Code Clone Terminology

Considering that this report relies heavily on the notion of code clones and their related terminology, it is important to have a clear understanding and definition of this terminology. In this section we will use existing literature about this terminology to define and explain relevant terms in the area of code clones.

First of all, a definition for the term 'code clone' itself needs to be established. This term has slightly different definitions in different literature [25]. The reason for this is that clones can be classified as different types, and literature considers different subsets to be defined as clones. Before going further into the different classifications, a definition of code clones regardless of these classifications can be established as provided by Baxter et al. [5]:

> 'Clones are segments of code that are similar according to some definition of similarity.'

This definition requires some further explanation and interpretation. The ‘definition of similarity’ can be defined to establish the appropriate taxonomy of clones. We interpret ‘segments of code’ to mean *all* segments with a certain similarity. So: a code clone is the collection of all segments of code that are similar according to some definition of similarity. What exactly constitutes a ‘segment of code’ will be discussed in further detail below in 2.1.2. Throughout this report we also use the term ‘clone group’ as synonymous with ‘code clone’ but used to emphasize the fact that we are referring to the entire group of similar (cloned) segments. We also use the term ‘clone fragment’ or ‘clone relation’ to refer to one of the cloned segments within a clone group.

2.1.1 Clone Taxonomy

Clones can be divided into two classes: exact clones and inconsistent clones [16]. Alternatively one can refer to these classes as textual clones and semantic clones. These classes of clones can then be further divided [25]: Exact clones can be of either purely exact clones - that is to say a directly copied and pasted segment of code - or a normalized exact clone. Normalized exact clones are the same as purely exact clones after some code normalizations, such as variable name normalization. In other words, a segment of code is copied directly but some parts that do not change the semantic behavior - such as variable names - are changed. Inconsistent clones can also be divided into two types: Clones where parts have been added or removed but semantically execute the same computation, and clones that execute the same computation but are textually completely different. Summarizing we can distinguish between the following classes and types of clones as established by Koschke et al. [25]:

- Exact/textual clones:

  Type 1 Purely exact clones

  Type 2 Normalized clones

- Inconsistent/semantic clones:

  Type 3 Clones with parts added/removed

  Type 4 Textually different clones

To illustrate these types further, below is a more detailed explanation plus an example for each type.

Type 1

The description of this type, purely exact clone, is actually somewhat of a misnomer. The segment of code does not necessarily have to be a one-on-one textual copy. This is because comments, whitespaces and formatting are discarded when deciding whether a code segment is a clone or not. This can be illustrated with the following example:

```
public void foo(a, b, c, d) {
    if (a > b) {
        c = b + d;
        a = 1;          // Reset a
    } else {
        d = a;          // Set d to a
    }
}
```

Listing 2.1: A simple code segment

A type 1 clone of this code segment could look like:

```
public void foo(a,b,c,d)
{
    if (a>b)            // Check
    {
        c=b+d;
        a=1;
    } else              // Otherwise
        d=a;
}
```

Listing 2.2: Type 1 clone

It is clear that the code segment in listing 2.2 is textually quite different from the original. Not only the comments are different and in other places, also the whitespacing is different as well as the parentheses which are even omitted completely at the else statement. Still, this is an exact clone of type 1.

Type 2

Type 2 clones can differ slightly more from the original code segment than type 1 clones. However, after normalization the code must be equivalent to the original. In practice, depending on the detection technique, this means that variable names and constant values can differ. Some detection techniques offer the possibility to add other properties that can differ. Consider the code fragment 2.1 listed above:

```
public void foo(a, b, c, d) {
    if (a > b) {
        c = b + d;
        a = 1;          // Reset a
    } else {
        d = a;          // Set d to a
    }
}
```

An example of a type 2 clone could be:

```
public void bar(x, y, i, j)
{
    if (x>y) // Check
    {
        i=y+j;
        x=2;
    } else
        j=x;
}
```

Listing 2.3: Type 2 clone

Clearly this piece of code is quite different from the original. However, the overall syntactic structure has been kept in the clone, justifying the classification as a type 2 clone.

Type 3

It can be argued whether type 3 clones are a type of exact clones rather than inconsistent clones. After all, much of the syntactic structure is maintained, however it is possible that 'gaps' are introduced. The concept of gaps is best understood by example. If we again consider the code segment listed in 2.1.

```
public void foo(a, b, c, d) {
    if (a > b) {
        c = b + d;
        a = 1;       // Reset a
    } else {
        d = a;       // Set d to a
    }
}
```

Now, a type 3 clone can be constructed:

```
public void foo(a, b, c, d) {
    if (a > b) {
        c = b + d;
        a = 1;       // Reset a
    } else {
        d = a;       // Set d to a
        c = 5;       // Added statement
    }
}
```

Listing 2.4: Type 3 clone

Here the clone is an exact copy of the original with one statement added. This one statement creates a gap of size 1 in the clone. This clone is still important and can cause all the same problems type 1 and 2 clones can cause and thus will have to be detected if possible.

Type 4

Type 4 clones are different from the other clones because the syntactic structure is broken. The only reason they can be considered clones is because the original code and the cloned code produce the same functionality. Lets take for example the computation of a number's factorial, a computation that can be done in several different ways:

```
public int fact(n) {
    int res = 1;
    for (int i = 0; i <= n; i++)
        res = res * i;
    return res;
}
```

Listing 2.5: A simple factorial algorithm

A type 4 clone of this segment of code could be:

```
public int fact(n) {
    if (n == 0)
        return 1;
    else
        return n * fact(n-1);
}
```

Listing 2.6: Type 4 clone

It is questionable whether clones of type 4 are relevant for the purpose of our research, because they are very difficult to detect. Regardless of that, it is likely that type 4 clones do not cause many of the problems other clones can cause. For example, if one were to change something in the factorial method in 2.5, this change might not be relevant for the cloned code in 2.6 because it is syntactically quite different. On the other hand, it cannot impact us negatively if clones of this type could be detected regardless and included in our research. In section 2.2 we will find out to what extent type 4 clones can be detected.

Other authors attempt to provide finer grained classifications [25, 3, 19] but for the purpose of this report the rather simple classifications mentioned above suffice. Having established the classifications of code clones, it is not yet possible to define the level of similarity by which clones are defined for this report. Ideally all the four types of clones can be considered, however this might prove challenging. It depends highly on the used detection technique which of the above mentioned types can and will be detected. In section 2.2 this will discussed further.

2.1.2 Clone Properties

Throughout this report a number of terms regarding the properties of a code clone are used. Particularly the dimensions of size and length of a clone are important notions that our research will rely upon. In the definition of a code clone established earlier there was mention of certain ‘segments of code’. One can perhaps imagine what types of segments are meant here, but truly this is a vague term. It is not clear, for example, what length a segment should be in order to be considered [29]. However, the metric used for measuring the length of the segment is then also unclear. It could be a certain number of *tokens* or certain number of *lines of code*. It turns out that this metric too - like the similarity of clones - is dependent on the detection technique. Some clone detectors use a token-based approach while others use the number of lines of code for clone detection. In the next section a closer look will be taken at clone detection techniques, so a choice regarding this matter will be made there as well. Closely related to the length of a clone is the *granularity*.

This metric can be explained as the nesting level of the code that is cloned: whether it is an entire class, a method or just a block.

Another property that is important when considering code clones, and one that we are most interested in researching, is the size. That is to say, the number of times a certain segment of code of a defined similarity occurs. A dimension related to this is the clone *distance* or *radius* which in turn is dependent on clone locations throughout the code [20]. Other properties (such as degree of similarity) of clones exist [25] but for the research in this report these are of little relevance.

2.2 Clone Detection

The research presented in this report depends on the ability to detect clones in code. We will be using existing techniques to detect clones, rather than implementing a new technique from scratch, therefore clone detection is an important part of the related work. To find out which techniques are the most suitable for use in this project, a look will be taken at literature of the most modern techniques available. In addition, it needs to be established exactly which clones will be detected. Does the technique offer the ability to detect all types of clones described in the previous section, or only a subset of types? This question and others will be answered in this section.

The main problem that is faced when trying to detect clones - similar segments of code - is that the detector does not know what segments will be cloned and where the cloned segments will be located. In theory the detector needs to compare every code segment against every other segment [29]. Of course this is an impossible task to do within reasonable time on large projects. In reality, clone detectors use more efficient algorithms that require less comparisons [29]. In addition, clone detectors use preprocessing techniques on the code to make it easier to handle for the comparison algorithm. Normalization of variables and identifiers which is mentioned previously is included in this preprocessing phase. Further details of this preprocessing phase are omitted in this report, for more information on this subject see Roy et al.'s work [31].

As mentioned briefly in a previous section, there are different techniques for clone detection. That is to say different techniques use different code units to reason over. The most important techniques are text based, token based and abstract syntax tree (AST) based. To get a better understanding of these techniques and provide some examples, the following subsections will take a look at each in turn.

2.2.1 Text based detection

This technique requires the least amount of pre-processing of code. The idea is that the textual representation of code is used to find recurring patterns. This can be done in several ways. One way is to generate a hash representation of segments of code and look ahead in the code to find whether the hash occurs in other places too [14]. Another way is to apply a so-called dynamic pattern-matching algorithm to the code, which for every piece of code looks at the entire rest of the code for duplicates. This approach is obviously very expensive (order $O(n^2)$) so not very widely used. In fact purely text-based approaches are generally not widely used because of various limitations. For example, line by line approaches are vulnerable to differences in use of line-breaks or parentheses [29].

An example of a text based detection tool is *Dup* [2]. This line-based tool assigns parameterized token strings to variables, so type 2 as well as type 1 clones can be detected. However, it is quite outdated and suffers from limitations such as the one mentioned above.

2.2.2 Token based detection

Token based detection techniques require the code to be transformed into a sequence of tokens. These tokens are constructed using techniques from compiler construction such as lexical analysis. The clone detector can then look for recurring patterns in this token sequence. This technique for detecting clones is especially powerful for type 1 and 2 clones. Type 3 clones can also be detected since these are basically type 1 or 2 clones, split at the clone gap (see 2.1.1). So long as the gap is not too large, that is the clone parts are not too far apart, type 3 clones can be detected effectively.

An example of a token based clone detector is *CCFinder* [18]. After the appropriate preprocessing this tool creates a so called suffix-tree on which a string matching algorithm is applied. In the end a pair of code segments (clone pair) and type information about the clone is returned. This tool is fairly widely used among researchers to detect clones in different programming languages.

2.2.3 AST based detection

AST based detection techniques use the abstract syntax or parse tree of code. This tree is constructed using a parser for the appropriate programming language rather than by the detection tool [31]. The AST contains all information about the code and thus is potentially more powerful than a sequence of tokens. Clone detection on an AST is generally done by finding subtrees that are similar. This can be done by hashing subtrees and only comparing subtrees with similar hashes.

An example of an AST based detection tool is SimScan[1]. This tool is often used by researchers to detect clones of all types efficiently in the Java programming language.

There are other detection techniques (such as dependency graph based) but these three are the most widely used and thus the most important. In order to make a choice about the technique and tool that will be used for the research, a consideration about the advantages and disadvantages needs to be made. Of the mentioned techniques both token and AST based are viable, whereas text based is somewhat outdated for use nowadays. As for tools there are a number of viable options: the mentioned CCFinder and SimScan, as well as AST based CloneDigger[2] and token based CloneDetective [15].

Extended research on these clone detection tools has been done by H. H. Mui [26]. While SimScan has the most accurate detection of all types, it is also the slowest by quite a large margin. Another problem is that since its development has halted for a while, it is fairly

[1]http://blue-edge.bg/download.html

[2]http://clonedigger.sourceforge.net/

outdated and does not support newer Java versions (1.4 and up). This makes it infeasible for use nowadays. CCFinder and CloneDigger are fairly similar in detection performance, but CCFinder is faster. Since it is also more widely used, it is perhaps better documented. For this reason, we choose CCFinder as our clone detection tool for use in our research.

2.3 Clone Evolution

The main field on which our research will build is that of code clone evolution or code clone genealogy. This field of research, as the name might suggest, entails the study of changes that code clones undergo over the lifetime of their respective software system. As clones are inevitably a part of every software system [20, 22], it is important to maintain them and identify the problems that play a role in this process. A number of questions exist in this field of research. An important problem here is mapping clones from one revision of the system to the next [27]. The main research question though in the area of evolution, is when do clones cause problems?

To look at these points in turn, this section is divided in subsections. First we will establish the terminology and context regarding clone evolution, particularly what type of changes can occur in clones. Then we will look at the related work conducted in the area of clone tracking. The final subsection will discuss research about when clones cause problems in the evolution phase.

2.3.1 Clone Genealogy

Before going further into problems regarding clone evolution, it is important to establish a clear context. Kim et al. [24] introduce a model of clone genealogies to formally describe evolutionary behavior. A clone genealogy describes all the changes a clone group undergoes over its lifetime from one revision to the next. A clone group is the collection of all cloned segments of code. To describe the changes in this genealogy at the level of clone groups, a number of evolution patterns are defined. These patterns describe the change that has occurred between two consecutive revisions of the system - though they could be applied to any two different revisions. The patterns are:

- *Same*: No change has occurred from one version to the next.
- *Add*: One or more code segments are added to the clone group in the newer version. For example, the developer copy and pasted a piece of code.
- *Subtract*: One or more code segments are removed from the clone group in the newer version. For example, a segment was refactored by the developer.
- *Consistent change*: All segments in the group are changed consistently.
- *Inconsistent change*: One or more segments are not changed consistently with the rest. For example when the developer forgets to apply the change to a cloned segment.

- *Shift*: One or more segments in the newer version overlap with segments from the older version.

These patterns for every change are not exclusive. It is quite possible for a developer to add a segment and consistently change all existing segments slightly from one revision to the next. In this case both the add and consistent change patterns apply to the clone group.

Of the identified evolutionary patterns, the inconsistent change pattern is the most interesting to research further because this is where most potential problems arise. Aversano et al. [1] acknowledged this and have divided the inconsistent change pattern further: one can distinguish between *independent development* and *late propagation*. In the case of independent development, the cloned segment that is not changed according to its original code is developed independently and the potentially different behavior of this cloned segment compared to its original code is intended. In the case of a late propagation, the change that was applied to one or more segments in the clone group was intended for all segments but the developer forgot to apply the change. When in a later revision a problem is identified regarding this segment, the change is applied (propagated) in this later revision.

2.3.2 Clone Tracking

The main problem that is faced when detecting clones in multiple revisions of a system is discovering whether a clone fragment found at a certain place in the newer revision is in fact the same fragment as the one in the previous revision. If nothing is changed in the file where a clone fragment resides, and the clone's other fragments are not changed either, the fragment is easily tracked from one revision to the next. However, as soon as something is changed, not necessarily in the clone fragment itself but in the file containing it, the clone position changes. Most clone detectors return a clone in the form of start/end line, along with other properties such as a reference to the clone's pairing fragment. Now when the file in which the clone resides is changed in a later revision, for example because some code is added above the clone, the start and end lines no longer match.

One way to solve this problem is the introduction of an abstract representation of clones that does not rely upon line numbers. Duala-Ekoko and Robillard [7] introduce such an abstract representation called *Clone Region Descriptors* (CRD) and have built a clone tracking tool that uses this representation called CloneTracker [8].

The idea behind using CRDs to locate clones is to use blocks as location markers. A CRD contains the file name and class name. If an entire method is cloned it is fairly straightforward: the method is linked in the CRD. However, if only a part of a method is cloned, it becomes harder to reference the cloned part without using line numbers. The way that CRDs refer to this cloned part is by dividing the method in blocks, with each block containing its operator or type (for, switch, etc) and an anchor. The anchor is a string that identifies the block more clearly, for example the argument function of the operator. However, even with this anchor appended, blocks are not necessarily unique because it is not uncommon for certain functions (for example for-loops) to have the same arguments several times. Hence, CRDs are appended with one more property called the corroboration metric. This is a hashtag that does uniquely identify the block.

Constructing CRDs from a system can be done efficiently and accurately [9]. The advantage of using this technique is that once the clones have been detected for the first time - say in the first revision - the CRDs can be generated and used for the remaining revisions instead of having to detect clones for all those as well.

2.3.3 Evolutionary Problems

Having established the changes that can be applied to clones over their lifetime, as well as being able to track clones and their changes, we can now look at when problems arise during this process. As stated earlier, the bulk of problems arise when an inconsistent change occurs, particularly a late propagation. Kim et al. [24] have found in an emprical study that many clones in a system are introduced with clear intent and generally have a short lifetime. Either they are introduced as a stepping stone for new functionality and from then on evolve independently from their original. Alternatively, many developers use clones to quickly gain access to a certain functionality for example to be able to implement a certain interface so that it can be tested. This clone is then quickly refactored - usually before a major new release of the system - and thus removed.

Bettenburg et al. confirm this view [6]. Their study focuses on the user-perceived effects of code clones, or bugs introduced because of cloning at the release level. The study's result showed that very few (1-4%) inconsistent changes to clones cause defects at the release level. The authors claim that because of this, clones do not slow down development very much. However, another similar study from the same year [16] that studies clones not only on the release level but also on revisions in between, claims that clones can in fact be a large factor in slowing down development. Juergens et al. claim that nearly all unintential inconsistent changes to clones cause defects in the system. These conflicting results show that no consensus in this area of research has been reached and further research is necessary.

An important question in clone evolution is whether certain - and if so, which - properties of a clone have influence on the number of inconsistent changes applied to that clone. Thummalapenta et al. [32] have researched whether the granularity and radius of a clone influence the number of late propagations applied to it. The granularity as we recall describes the nesting level of the clone: class, method or block. The radius is a measure of distance between two clones that takes into account whether the clones reside in a different file on the system. The results of the study could not prove that either clone granularity or clone radius affected the number of late propagations to the clone.

2.4 Software Repository Mining

At the heart of our research is the ability to retrieve revisions of a software system from a repository. Before clone detection can even be applied, or information about the evolution of clones can be gathered, a representation of the system's revisions needs to be in place with the source code for every revision available. Many software developers choose to use versioning software for their repository such as Concurrent Versioning System (CVS)[3], SubVersion (SVN)[4] or Git[5]. CVS is an older system that is not used very much for modern day projects. Our main focus will be on mining SVN repositories, but the theory is also applicable to Git and similar systems. Aside from versioning systems there are other software repositories that can be mined, such as bug trackers. However, we are mainly interested in extracting actual source code versions and the evolutionary data associated with those from repositories, and hence will only look at mining versioning systems.

The value of software repository mining has been talked about since the 90's [12, 4, 11]. There is a large amount of information about a system that can be extracted by using software repository mining. As Ball et al. [4] demonstrate, by using software repository mining one can not only gain access to the system's code, he can also extract large amounts of evolutionary information about how the system was developed. For example, by using certain metrics like connection strength, it is possible to see which classes are changed simultaneously. This is of course highly relevant information for code clones. Also it can be clearly illustrated which developer is responsible for which pieces of the code. Having access to this kind of information can greatly help developers finding weak spots in the system that need extra attention or perhaps even reengineering [11].

Kagdi et al. [17] have identified further evolutionary information that can be mined from versioning systems. It is possible to gain information about which classes change simultaneously, indicating the possibility of either a dependency or a clone relation. The authors provide a number of reasons why to do software repository mining. They also provide a taxonomy of what a software repository mining approach might look like, but they do not provide any tools to aid in the process.

Robbes [28] states that traditional versioning systems are not sufficient to use for studying software evolution. Instead, he proposes a change-based software repository system where the evolution of a software system is not shown as a history of versions, but rather a set of change operations that have brought the system to its current state. The program is represented as a collection of ASTs that evolve as the system evolves. The author also presents an implementation of his proposed system, however this tool has severe limitations for use in our research. The tool does not support versioning systems, instead it is supposed to be used for a software system from the beginning so that all changes can be recorded over its lifetime. Furthermore the implementation is limited to an addon for an IDE that is not widely used in industry.

Software repository mining is obviously highly valuable but the practice is still in an early stage. In a roundtable by prominent researchers in the field of software repository mining the future of the practice was discussed [13]. Software repository mining can become very useful in a corporate setting, for example for managers to gain insight into the activity of developers in the system. Repositories will also be regarded as more valuable, and they will be used more with future mining possibilities in mind. Tools for repository management will become more integrated with the rest of the system development process,

[3]http://www.nongnu.org/cvs/

[4]http://subversion.apache.org/

[5]http://git-scm.com/

and get more functionality for visualization. However, care should be taken not to have too high expectations for the future of software repository mining.

The practice of software repository mining is and will be an important area of research that can bring large gains to the field of software engineering. However, none of the presented research in this area offers a readily available solution for mining versioning systems. Therefore we have decided to implement a tool that can retrieve the source code for every revision from an SVN server. The output of this tool can then be used to run the clone detector CCFinder on. The tool will be implemented in Java using the SVNKit[6] library to access the necessary functionality to mine an SVN server. Further details of the implementation of this tool will be presented in a following chapter.

[6]http://svnkit.com/

Chapter 3

Tracking Clone Evolution

To be able to answer the research questions we will need to do a number of experiments on real world software systems. The systems to be used for our research will be listed in the next chapter. This chapter will describe the tool that is going to be used to do the experiments. The tool tracks the evolution of clones throughout their lifetime and there are several steps that have to be taken. The first step is the mining of repositories, this process is described in section 3.1. The next step is the detection of clones, which is explained in section 3.2. The final phase of the tool consists of the reasoning over data collected in the previous steps, which is explained in more detail in section 3.3.

3.1 Repository Mining

The first step in the process of tracking clone evolution is the finding and mining of a software project's repository. Since there are no suitable tools readily available for this exact purpose, this part of the tool will need to be implemented from scratch. We have chosen to mine only repositories that use the SVN versioning system. The library that provides interfaces for accessing SVN repositories in Java is called SVNKit and it is freely available as open source software.

The implementation of the SVNKit functionality will mostly revolve around the use of `SVNRepository` Java objects. This is done in a `SVNRepositoryController` class that implements an interface. This class will take as parameters the URL of the repository to be mined and the (local) path to which it will be stored on the executing machine. If the repository requires authentication to be accessed, the username and password are also provided. Once the class is initiated an `SVNRepository` object for the mined and the local repository is created. The way the checkout process of a remote repository to the local machine works in this implementation is that SVNKit requires a remote as well as local repository object to be created, where the entire remote repository is then copied over to the local one. The `SVNRepository` objects for the remote and local repositories are created in the same way, the only difference being the URL on which they are initiated: the remote repository is typically located on an http or https address and the local one is located on a local path (for example C:\repository in a Windows environment or ∼/repository in a Linux/UNIX environment). The SVNRepositoryController class then provides methods for starting the mining process. By default, the program checks the repository for the last revision and simply checks out all revisions from 1 to the last to the local machine. Alternatively, a range of revisions can be specified to narrow down the mining process. Also, it might be desirable to narrow down the number of mined revisions further so an interval can be specified, such that for example only every other revision is mined, or every 4 revisions.

This mining process only considers the case where the repository accepts the repeatedly checking out of revisions. Some repositories do not accept this, but in this case we can still access the repository and acquire the revisions using another option. SVN repositories allow for the mirroring of the server to be hosted by others. In this case all the information on the server is replicated to be hosted on another machine. This functionality is supported by SVNKit by using the `replay()` command. Our tool implements this functionality by providing a `replicateRepository()` method to the `SVNRepositoryController` class. In case the repository that will be mined does not allow the repeatedly checking out of revisions, this method can be used so that the repository can be hosted locally. Because the information on the server is stored differently from an actual checkout - the directory structure of the revisions is not replicated in the same way a checkout would do it - we still have to check out the revisions somehow. We do this by locally hosting the repository and checking out the revisions the same way it would be done were the repository hosted remotely but with repeatedly checking out allowed, the only difference being that we provide a local path to the repository to be mined instead of a remote URL. This way, the contents of the repository are duplicated locally, but because we require a certain directory structure for the other functionality of our tool and local storage space was not a constraint, we considered this solution to be the most efficient one. Besides this, checking out from a locally hosted repository is considerably faster than from a remote one, so the bulk of time would still be in replicating the remote repository.

We talk about the directory structure that we attain from mining a repository. For the purpose of the functionality of the other parts of our tool, we want a specific structure in which the projects and revisions are placed. The structure we use is the following: we use a `projects` directory in which all the projects to be researched will be placed. These project directories then contain a directory for every revision which includes that revision's number. In that revision directory the project's directory structure is kept and used for the later phases of our tool. So our directory structure looks like this: ~`/projects/projectx/r1` in which the project's directory structure is kept, where usually a `/src/` directory is used. Concluding, after the software repository mining phase of our tool we end up with a collection of projects which are then divided into revisions, ready to be used in the next phase of our tool.

3.2 Clone Detection

Once the repository has been mined and the project stored on a local machine, the next step is the detection of clones. As stated and motivated in a previous chapter, the tool that will be used for doing this is CCFinderX. This tool can detect clones in Java software by using token based detection.

3.2.1 Detection

The detecting of clones is done by calling an instance of the CCFinderX application and running it on all the revisions of a project. The path of the project's source directory is supplied and our tool loops over the standardized revision directories contained in the project directory. For all revisions, all .java files contained in that revision's directory will be scanned for clones. The results are then stored in another directory which can also be supplied by the user. The CCFinderX application first does a preprocessing cycle which creates a preprocessed file with the extension .ccfxprep for every Java file. This file contains the contents of the Java file in token form, with one token on every line. Next, the actual clone detection algorithm is executed on these files. Once the detection algorithm is done, a binary file with the results is generated in the results directory. This file is not readable, but CCFinderX offers postprocessing functionality that turns this binary results file into a text file. The postprocessing is done directly after the detection, such that at the end of execution a readable text file with clone information is stored in the results directory.

From this text file the clone information can be extracted. It is organized in the following manner: first there is a list of all the files in the project that contain cloned fragments, with numbers assigned to those files as identifiers. This file list also lists the number of tokens in each file. Then there is a list of clone pairs of the form `[a] [b.c-d] [e.f-g]` with *a*, *b*, *c*, *d*, *e*, *f* and *g* being integer numbers, which indicates the clone pair and its location: first the identifier *a*, then the file as listed above (*b*, *d*) and the line numbers (*c*, *d* and *f*, *g*), or rather token numbers, which indicate the location range of the cloned fragment within the preprocessed file. To illustrate this better, consider this example of a file containing CCFinderX clone detection information in listing 3.1:

```
source_files {
    1   C:\\projects\\charts4j\\r1\\src\\AbstractLineChart.java 297
    2   C:\\projects\\charts4j\\r1\\src\\BarChart.java   577
}
clone_pairs{
    651 1.119-183    2.158-222
    572 1.183-291    2.304-412
}
```

Listing 3.1: Example CCFinderX result file

This file lists two source files in the project which contain 297 and 577 tokens respectively. Then there are two clone pairs listed, which are assigned identifiers 651 and 572. The first clone pair consists of segments of code located in the AbstractLineChart.java file on token 119-183 and the cloned segment is located in the BarChart.java file on token 158-222.

Both the detection and postprocessing are done subsequently on every revision in the project's directory, with the results directory typically being located in the project's directory as well.

3.2.2 Analysis

The way in which CCFinderX presents the clones is not very efficient or clear. CCFinderX detects clones in *clone pairs*: for every clone pair CCFinderX finds, it stores the original segment as well as the cloned segment. The results file thus contains duplicate information, since CCFinderX will find all cloned pairs. To give an example, consider the case where a clone has the original segment *x*, and cloned segments *y* and *z*; CCFinderX will now find the pairs *xy*, *yx*, *xz*, *zx*, *yz* and *zy*. In the concrete example listed above in listing 3.1 we have thus omitted some detected clone pairs. The complete clone pair list looks like this:

```
...
clone_pairs{
    651 1.119-183    2.158-222
    572 1.183-291    2.304-412
    651 2.158-222    1.119-183
    572 2.304-412    1.183-291
}
```

Listing 3.2: Example CCFinderX clone pair list

Luckily these duplicate clone pairs are easily spotted because, as can be observed from listing 3.2, they are assigned the same identifier.

Because of this inefficient representation by CCFinderX we introduce a new more efficient format for storing this clone information. We create a *clone group*, implemented as a `CloneGroup` object, which contains the original fragment and every cloned fragment once. The clone group for the earlier example would hence look like this: {`x, y, z`}.

To be able to track clone fragments over several revisions, simply storing their location expressed in token numbers is not enough. As explained in the previous chapter, the location in line numbers of a code fragment can change from one revision to the next. Therefore, the tool analyzes clone fragments and stores them in a more usable format. Instead of using a number for the file in which the clone fragment is stored, we store the relative path to the file. We keep the location information, since we have access to this anyway and storing it does not cost much, but we add a way to uniquely identify and locate the clone by introducing a clone region descriptor (CRD). In the previous chapter the term clone region descriptor was explained, but to briefly reiterate: the CRD is a line of text that uniquely identifies a clone fragment by including its relative file path, class, method and block information within the method. Barring extreme situations, such as the developers changing the entire package structure of the project, CRD's will allow us to identify clones over multiple revisions.

CRD Implementation

The implementation of the generating of CRD's for code fragments is done by reading and reasoning over the code from the preprocessed file containing the token representation of the code. The first step is locating the starting point of the fragment in the preprocessed file. From here the task is to find the first block or method identifier by looking ahead. Since the preprocessed file consists of tokens, with one token per line, this process is a matter of looking ahead until the next opening curly bracket token represented by `(brace` is found (or the end of the clone, whichever occurs first). Since the code fragment can start practically anywhere, it is not known what type of block or function identifier will be found, and whether it will be found at all: it is possible that the cloned fragment is simply a statement and not a block. We look for blocks of certain types, as listed by the specification of a CRD [8]: *for, while, do, if, else, switch, try, catch, finally, synchronised.* Of course we also look for a function with its identifier.

The next step, whether a block or function specification has been found or not, is to look at the preceding tokens to find the next level of nesting and the block or function in which the code fragment is located. To do this, the algorithm reads backwards from the clone starting point in the preprocessed file until it finds the opening curly bracket token. Since it is possible that another block resides in the same nesting level above the cloned fragment, the algorithm might wrongly find the opening bracket of this other block. To counter this, the algorithm keeps a stack of closing curly brackets (represented in the preprocessed file as `)brace`) that it finds, and skips any opening brackets as long as the stack is not empty. Once the actual bracket has been found, we need to look a bit further back to find the matching block or function identifier, matching the block types or function identifier as stated earlier. This process is repeated until the end of the file is reached, so that eventually the file name, class name, method name and block identifier(s) are retrieved and put together in a string to represent the CRD.

After the CRD for clone fragments has been generated, all the gathered clone information is stored in an XML database. For every revision a list of clone groups is saved, with every clone group containing a number of clone fragments which are identifiable by the CRD but also contain the location as found by CCFinderX.

3.3 Reasoning

Now that the clones have been detected and analyzed so that their information is easily accessible, the next step in the tool is to - by lack of a better term - reason over the information of several revisions so that evolutionary data can be extracted. The clone information is stored in XML format with every revision containing a list of clone groups. From this information as it is we can extract certain data. Firstly, the size of clones can be measured. The size of all clones in a revision can be retrieved by simply measuring the number of cloned segments in the clone groups. From this data we will then generate statistics and present these in a readable manner. Our tool supports the calculating of the average clone size in a revision, as well as the minimum, first quartile, median, third quartile and maximum clone size. The tool supports exporting these statistics as an Excel file, or it can generate a graph directly. The generating of graphs is done by an interface which we have implemented using the Charts4j program, which is one of the projects we have selected for research as well. This program provides an API for generating several types of graphs from datapoints. This part of our tool is fully modifiable and extendable.

To generate statistics about the clone size of all clones over a number of revisions it is not needed to track individual clones. However, if we want to find out whether a clone has been refactored in the course of its lifetime, it becomes necessary to be able to track individual clones over multiple revisions. The CRD implemented earlier allows us to uniquely identify clone fragments, but not entire clone groups. To be able to do this, an algorithm to uniquely identify entire clone groups per revision needs to be implemented. Since clone groups consist of several clone fragments that can be tracked already through the CRD, it makes sense to use this for implementing the tracking of the clone group.

Since a clone group as well as its contents can change however, we cannot simply check whether the clone group consists of all the same fragments. However, because it is also impossible for a fragment to be in more than one clone group, we can simply check whether at least one fragment is still contained in the clone group. On the other hand, it could be possible that a fragment is removed even if the entire clone group still exists. Therefore the algorithm we implemented to compare a clone to another to decide whether it is equivalent works by checking whether one fragment is included in both clones. This means comparing all fragments of one clone against all fragments of another clone, or if there are n fragments an order of complexity of $O(n^2)$. Perhaps such a solution is not the most efficient solution to this problem, however because clones do not grow very large, this solution does in practice not increase the running time of the tool by much while being easy to implement.

The next part of our tool uses the clone tracking to generate statistics about clones being deleted. We check in every revision whether the clones from the previous revision are still present using the method described above. If the clone is no longer present we record it in its latest instance. At the end of the computation, we end up with a list of deleted clones which we can use to gather information, most importantly the size of the clone which is the figure we are interested in to answer our research questions. We can then use this information to generate statistics of the evolutionary behavior displayed in the deletion of clones. Particularly the probability of a clone of a certain size being deleted. Aside from deleted clones, we can also gather information on another type of clone that is interesting for our research, namely clones which are not entirely deleted but have fragments removed from them. To find these clones, we again compare iteratively for every previous revision all clones with those in the current revision. If the clone does not contain all the fragments it had in the previous revision that clone is recorded. Additionally if the clone is no longer present it is also recorded because it no longer contains all the fragments of the previous revision (namely none). Using the resulting list of clones which had fragments removed, we can again generate statistics towards answering our research question.

At the end of the reasoning phase we end up with a number of graphs and statistics that we can use for our research. These results and their detailed analysis are reported in the next chapter.

Chapter 4

Experiment and Results

In order to research the evolution of clones, we must do an experiment on real world software systems. In this chapter we discuss the experiment and the results gained from that experiment. In section 4.1 we outline the software systems that we are going to do the experiment on, and discuss the reasoning behind choosing these systems. In section 4.2 we present the results that the experiment has produced. In section 4.2.3 we analyze and discuss the results.

4.1 Subject Systems

In order to gain insight into the evolution of code clones, we have selected a number of software systems that we will use our tool on to gain data. These systems are open source Java systems with the entire SVN repository open for mining. We make sure that the size and age (that is to say, the number of revisions) of these systems is quite different, as to gain a wider viewpoint of the researched properties. Also, before the results are presented we look into the systems further to get an idea of what we can expect in the results. The selected systems are listed below.

4.1.1 Charts4j

Summary:

- **Number of revisions:** 279
- **Date of first revision:** September 30, 2008
- **Number of contributers:** 4
- **Lines of code:** 8494
- **Website:** http://charts4j.googlecode.com/

Charts4j is the smallest and youngest system we have selected. It is a Java library for creating charts using the Google Charts API[1]. It is developed by just a few people and the

[1] http://code.google.com/apis/chart/

activity on the SVN is not very high. Keeping this in mind, it is likely that the number of clones, as well as the size of clones will be relatively small. However, the functionality is also important to keep in mind. It generates several types of charts, which can result in relatively similar code under the hood. It is quite likely that generating a line chart results in similar code to when generating a bar chart. Looking at the code we can back up this speculative statement. The respective classes for the generation of bar and line charts in Charts4j are BarChart.java and AbstractLineChart.java. Both these classes contain a method for preparing the data to be converted into a chart, called prepareData(). This method, while not exactly the same, clearly contains very similar code, some parts directly cloned as can be seen in the code samples from these classes in listings 4.1 and 4.2.

```
for (Plot p : lines) {
  final PlotImpl line = (PlotImpl) p;
  if (hasLegend) {
    parameterManager.addLegend(line.getLegend() != null ? line.getLegend
        () : " ");
  }
  if (hasColor) {
    parameterManager.addColor(line.getColor() != null ? line.getColor() :
         BLACK);
    colors.add(line.getColor() != null ? line.getColor() : BLACK);
  }
  final ImmutableList<Marker> markers = line.getMarkers();
  for (Marker m : markers) {
    parameterManager.addMarkers(m, lineCount);
  }
  final ImmutableList<MarkedPoints> markedPointsList = line.
      getMarkedPointsList();
  for (MarkedPoints mp : markedPointsList) {
    parameterManager.addMarker(mp.getMarker(), lineCount, mp.
        getStartIndex(), mp.getEndIndex(), mp.getN());
  }
  if (line.getFillAreaColor() != null) {
    parameterManager.addFillAreaMarker(FillAreaType.FULL, line.
        getFillAreaColor(), lineCount, 0);
  }
  if (hasLineStyle) {
    parameterManager.addLineChartLineStyle(line.getLineStyle() != null ?
        line.getLineStyle() : LineStyle.newLineStyle(1, 1, 0));
    lStyles.add(line.getLineStyle() != null ? line.getLineStyle() :
        LineStyle.newLineStyle(1, 1, 0));
  }
  if (hasPriorities) {
    priorities.add(line.getPriority() != null ? line.getPriority() :
        Priority.NORMAL);
  }
  lineCount++;
}
```

Listing 4.1: AbstractLine.java code sample

```
for (Plot p : barChartPlots) {
  final PlotImpl plot = (PlotImpl) p;
  parameterManager.addData(plot.getData());
  if (hasLegend) {
    parameterManager.addLegend(plot.getLegend() != null ? plot.getLegend
        () : " ");
  }
  if (hasColor) {
    if (plot.getBarColors().isEmpty()) {
      parameterManager.addColor(plot.getColor() != null ? plot.getColor()
          : BLACK);
    } else {
      final List<ImmutableList<Color>> colors = Lists.newArrayList();
      final List<Color> colorList = Lists.newLinkedList();
      for (int i = 0; i < plot.getData().getSize(); i++) {
        colorList.add(plot.getColor() != null ? plot.getColor() : BLACK);
      }
      for (BarColor bColor : plot.getBarColors()) {
        if (bColor.getIndex() < plot.getData().getSize()) {
          colorList.set(bColor.getIndex(), bColor.getColor());
        }
      }
      colors.add(Lists.copyOf(colorList));
      parameterManager.addColors(Lists.copyOf(colors));
    }
  }
  if (hasZeroLine) {
    parameterManager.setBarChartZeroLineParameter(plot.getZeroLine() /
        Data.MAX_VALUE);
  }
  if (plot.getDataLine() != null) {
    parameterManager.addLineStyleMarker(plot.getDataLine().getColor(),
        lineCount, 0, plot.getDataLine().getSize(), plot.getDataLine().
        getPriority());
  }
  for (Marker m : plot.getMarkers()) {
    parameterManager.addMarkers(m, lineCount);
  }
  for (MarkedPoints mp : plot.getMarkedPointsList()) {
    parameterManager.addMarker(mp.getMarker(), lineCount, mp.
        getStartIndex(), mp.getEndIndex(), mp.getN());
  }
  if (plot.getFillAreaColor() != null) {
    parameterManager.addFillAreaMarker(FillAreaType.FULL, plot.
        getFillAreaColor(), lineCount, 0);
  }
  lineCount++;
}
```

Listing 4.2: BarChart.java cloned sample

From inspecting the rest of the code of the Charts4j system we can also see there are certain abstractions that the developers have failed to implement. There are a number of classes in the parameters package that all have the same getValue() method implemented. This method is declared in an abstract class AbstractParameter.java that all these classes extend, but the method is not implemented here. Not all classes that extend this abstract class implement this method in the same way, but a large part of the classes do: 16 out of 26 classes. These duplicate methods are likely to be detected as clones. Two samples of the cloned getValue() method are shown in listings 4.3 and 4.4.

```java
public String getValue() {
  final StringBuilder sb = new StringBuilder();
  int cnt = 0;
  for (AxisLabels aLabels : axisLabels) {
    sb.append(cnt++ > 0 ? "|" : "").append(aLabels);
  }
  return !axisLabels.isEmpty() ? sb.toString() : "";
}
```

Listing 4.3: AxisLabelParameter.java getValue() method

```java
public String getValue() {
  final StringBuilder sb = new StringBuilder();
  int cnt = 0;
  for (LineStyleWrapper l : lineStyles) {
    sb.append(cnt++ > 0 ? "|" : "").append(l);
  }
  return !lineStyles.isEmpty() ? sb.toString() : "";
}
```

Listing 4.4: LineChartLineStylesParameter.java cloned getValue() method

Aside from this numerously cloned piece of code, we have not been able to find clones with a large number of clone relations in the Charts4j system. Therefore we can predict the scale of clone sizes likely to be detected will be within tens of clone relations.

4.1.2 Google Guice

Summary:

- **Number of revisions:** 1597
- **Date of first revision:** August 23, 2006
- **Number of contributers:** 11
- **Lines of code:** 42368
- **Website:** http://google-guice.googlecode.com/

Google Guice is a Java framework that provides functionality for doing dependency injection in Java code. Dependency injection is a design pattern in object oriented software engineering and this tool helps in automatically adhering to this pattern when developing Java software. The dependency injection pattern is specifically aimed at improving a system's testability. By injecting dependencies into classes where needed, the developer can test highly dependent classes more easily. The system is fairly young and considering that its purpose is to enable developers to create cleaner and more testable code, it is to be expected that the developers of this system have employed abstraction where possible. Also, because this system is specifically designed to help the testing of code, and Google is known for its rigorous testing of software, this system contains a lot of test code.

This test code is particularly interesting to look at as it is in fact larger than the actual core software. Because of the high code coverage that Google wants to achieve with its testing, there are numerous clones in the testing part of the system. Two examples are listed below.

```java
providesInjector = Guice.createInjector(new AbstractModule() {
  protected void configure() {
    install(ThrowingProviderBinder.forModule(this));
  }

  @SuppressWarnings("unused")
  @CheckedProvides(RemoteProvider.class)
  String foo() throws AccessException {
    throw new AccessException("boo!");
  }
});
```

Listing 4.5: ThrowingProviderTest.java sample test code fragment

```java
providesInjector = Guice.createInjector(new AbstractModule() {
  protected void configure() {
    install(ThrowingProviderBinder.forModule(this));
  }

  @SuppressWarnings("unused")
  @CheckedProvides(RemoteProvider.class)
  String foo() throws RuntimeException {
    throw new RuntimeException("boo!");
  }
});
```

Listing 4.6: ThrowingProviderTest.java sample test code cloned fragment

From what we can see from a quick inspection of the code the most clones are located in the testing part of the code. It will be interesting to see whether the results will confirm whether the testing part of the system contains the largest clones as well.

4.1.3 Subclipse

Summary:

- **Number of revisions:** 5254
- **Date of first revision:** June 19, 2003
- **Number of contributers:** 25
- **Lines of code:** 11379
- **Website:** http://subclipse.tigris.org/

Subclipse is a plugin for the Eclipse platform that integrates Subversion (SVN) functionality into the IDE. This project is quite popular and the SVN server sees quite high activity. The system has been extended since its first release, now encompassing several separated systems (for example, the core and ui are split projects). Our experiment is done on all the separate projects included in the Subclipse repository. Since the core project is still limited in scope, it is possible for us to manually inspect the core code to get an idea to what extent there are clones in the system. A class that stands out in terms of cloned code is the SVNProviderPlugin.java class, which contains various methods that are very similar to each other. These methods contain function calls to library functions, with just the parameters differing among them. One method, which broadcasts information about file changes in the subversion repository, is cloned five times. The information that this methods broadcasts is slightly different each time, for example file changes or meta file changes. Two instances of this cloned piece of code are listed in figure 4.7 and 4.8.

```java
public static void broadcastSyncInfoChanges(final IResource[] resources,
    final boolean initializeListeners) {
  IResourceStateChangeListener[] toNotify;
  synchronized(listeners) {
    toNotify = (IResourceStateChangeListener[])listeners.toArray(new
        IResourceStateChangeListener[listeners.size()]);
  }

  for (int i = 0; i < toNotify.length; ++i) {
    final IResourceStateChangeListener listener = toNotify[i];
    ISafeRunnable code = new ISafeRunnable() {
      public void run() throws Exception {
        if (initializeListeners) listener.initialize();
        listener.resourceSyncInfoChanged(resources);
      }
      public void handleException(Throwable e) {
        // don't log the exception....it is already being logged in
        // Platform#run
      }
    };
    SafeRunner.run(code);
  }
}
```

Listing 4.7: SVNProviderPlugin.java sample code fragment

```java
public static void broadcastModificationStateChanges(final IResource[]
    resources) {
  IResourceStateChangeListener[] toNotify;
  synchronized(listeners) {
    toNotify = (IResourceStateChangeListener[])listeners.toArray(new
        IResourceStateChangeListener[listeners.size()]);
  }

  for (int i = 0; i < toNotify.length; ++i) {
    final IResourceStateChangeListener listener = toNotify[i];
    ISafeRunnable code = new ISafeRunnable() {
      public void run() throws Exception {
        listener.resourceModified(resources);
      }
      public void handleException(Throwable e) {
        // don't log the exception....it is already being logged in
        // Platform#run
      }
    };
    SafeRunner.run(code);
  }
}
```

Listing 4.8: SVNProviderPlugin.java sample cloned code fragment

As can be observed from these listings, the code is highly similar, but a few subtle differences make it so that it is difficult to create abstractions or generalizations for this code. From a quick look over the core code of Subclipse, we have not been able to find another clone with a larger number of relations. This would lead us to believe that - barring an oversight - there are no clones of very large size in the Subclipse project, and the maximum clone size would be within single digits.

4.1.4 ArgoUML

Summary:

- **Number of revisions:** 19796
- **Date of first revision:** Jan 26, 1998
- **Number of contributers:** 52
- **Lines of code:** 121615
- **Website:** http://argouml.tigris.com/

ArgoUML is the largest and oldest system that we have researched. It is a system used for creating UML documents. Considering its code base is quite large, and has grown quite significantly over the timespan of its development, we are interested in seeing whether this results in an increase in the number of clones and the size of clones as well. Because of the scope of this project, a manual inspection of the code to look for potentially significant clones is unrealistic.

4.2 Clone Size Evolution Experiment

The first thing we want to research on the selected systems is the evolutionary behavior of clones, in particular their size. Before we show the results we will briefly describe the experiment that will be executed on the selected systems.

4.2.1 Experiment

Our tool can mine revisions of a system from its repository. It can then detect clones in these revisions, save the results and output information about these results. For the experiment, we have mined the first 1500 revisions for each project if possible. This number is chosen because the program as it is implemented now is such that larger amounts of revisions are not supported because of machine memory limitations. The Charts4j project does not have 1500 revisions as of yet, so all available revisions were mined (238).

The next step in the program is to detect the clones using CCFinderX as the clone detection program. The information on the detected clones is then analyzed and stored in XML format. The final step is the reading and reasoning over the XML data and to generate a form of presenting the information that are more easily readable, such as graphs. In our results we will show several graphs which represent different data. In all the graphs - unless specified otherwise - the horizontal axis represents time, with earlier revisions to later revisions represented respectively from left to right.

The question that we want to answer by conducting this experiment is the following:

- **How does the size of clones evolve over time?**
  To be able to answer this question we will need to measure the size of the clones detected in the systems. The size of a clone, as defined in an earlier chapter, is the number of cloned fragments it has. Because the question mentions clones in general, we only need to look at statistics generated on all clones within a revision. Our tool supports the calculation of average, minimum, first quartile, median, third quartile and maximum clone sizes of all clones within a revision. If we calculate these statistics for every mined revision, the resulting clone sizes can be plotted against time (revisions) in a graph.

4.2.2 Results

The results from running the experiment on the selected systems are listed below. For each system, we present some graphs regarding the size of clones within that system. The sizes are plotted against time on the horizontal axis. The graphs are explained in more detail and if the graph has interesting properties, these properties are investigated further.

Charts4j

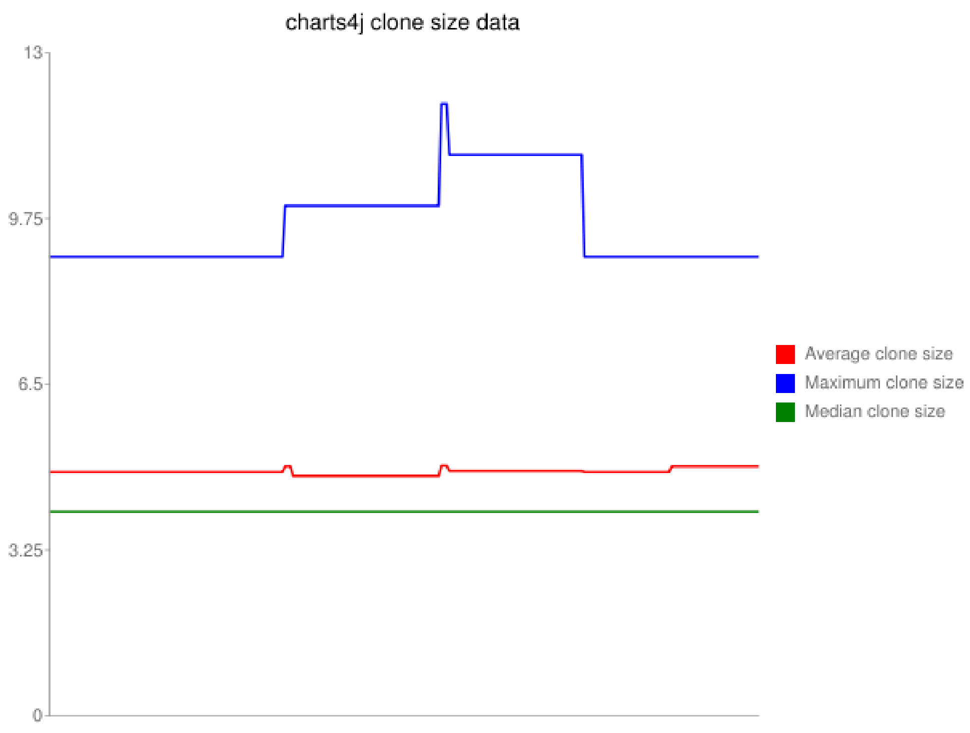

Figure 4.1: Clone size evolution Charts4j

In the first graph we see that the maximum increases, then decreases. The average clone size remains mostly stable. It should be noted that clone pairs (clones of size 2) are filtered out of this graph. These are in fact the most numerous and the median size would be 2 were they included. It is worth taking a closer look at the largest clone in this project and see what happened to it where its size changed. There are four instances where the maximum clone size changes. From the graph the exact revisions where this happens cannot be discerned, but with our tool we can find out exactly in which revision the maximum clone size changed. In this case, this is in revisions 99, 158, 161 and 212. If we take a look at the subversion history log, in none of these revisions a mention was made about the refactoring of clones, so a further inspection of the actual code is needed.

First of all, it should be noted that the clone with the maximum number of clone relations is in fact the same clone over the entire history. Also, the observation we made at the introduction of the Charts4j system was accurate, as the mentioned cloned getValue() method is in fact the largest clone in the system. Apparently though, CCFinderX has not detected the same number of occurances of this clone as we did during our manual code inspection (9 in the latest revision, instead of our 16, see 4.1.1). The reason behind this is unknown to us. In fact, in the last instance, revision 212, where the number of clone relations is reduced from 11 to 9, all the cloned fragments are changed slightly. The two fragments that are removed however, seem to have been changed in the same way but are no longer considered clones. This might be an inaccuracy in the CCFinderX algorithm, or an oversight on our part. The other instances where the maximum clone size changes are down to new classes being introduced that implement the same method in the case of increases, or a change in the implementation of one of the cloned fragments in the case of a decrease.

Another thing to note from the graph is that the median is stable and lower than the average. This suggests that there are a large number of small clones and only a few larger ones.

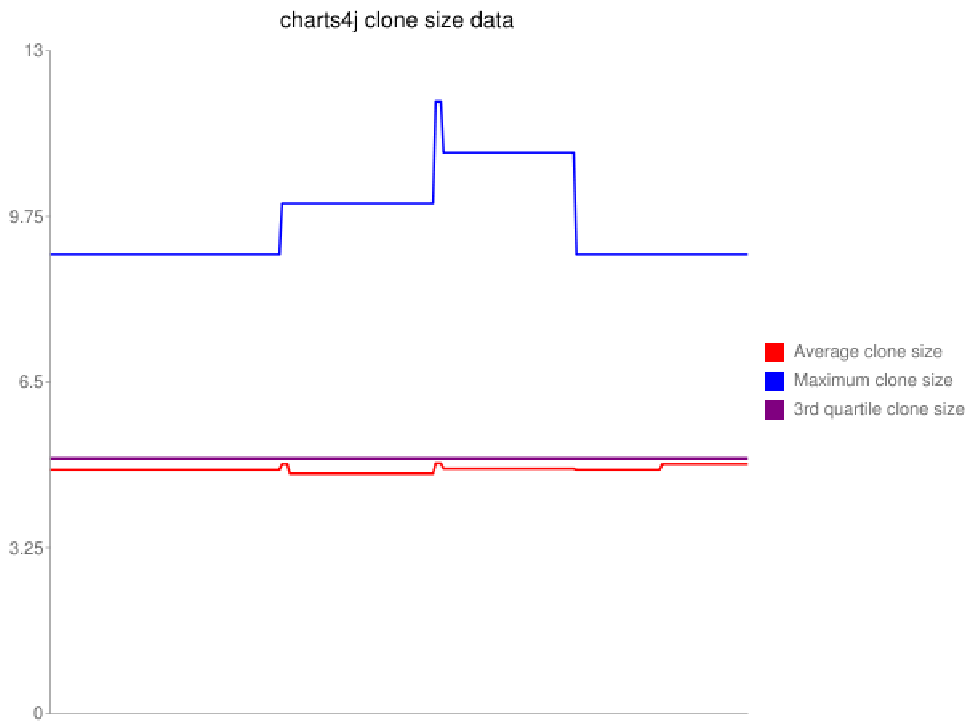

Figure 4.2: Clone size evolution Charts4j, more detail

Figure 4.2 shows the third quartile instead of the median. This is higher than the median but still stable at 5, whereas the median is at 4. The evolution of these third quartile clones is not very interesting as their size hardly changes.

Google Guice

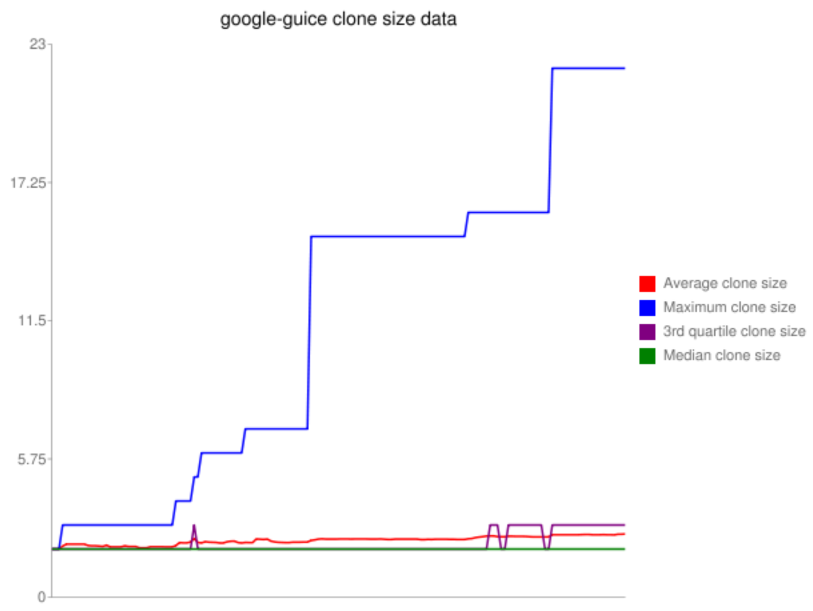

Figure 4.3: Clone size evolution Google Guice

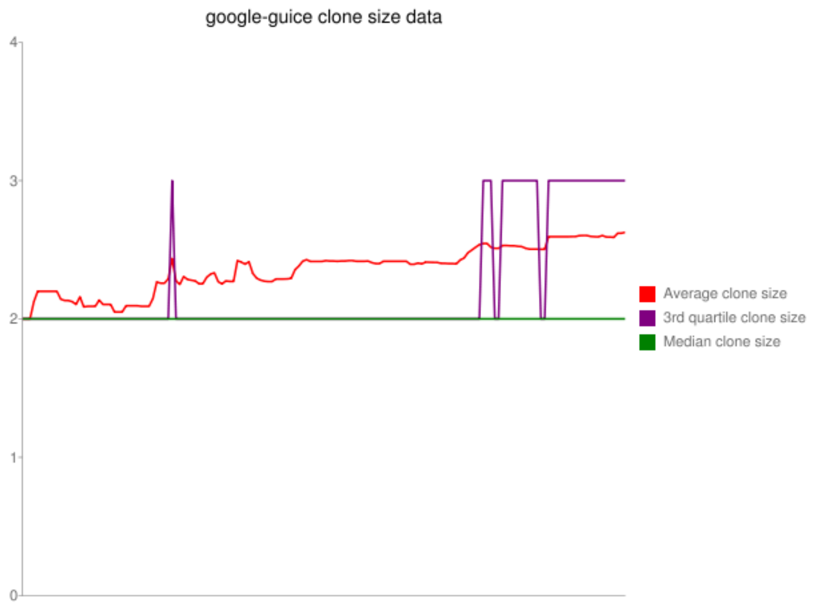

Figure 4.4: Clone size evolution Google Guice, more detail

In Google Guice, the most eye-catching feature of the graph is that of the maximum, which keeps growing larger over time. Because clone pairs are included in these graphs, it can be seen that these are clearly the most prevalent since the median size is 2 at all times, and even the third quartile is 2 for most of the time.

The maximum clone size in this system is the most interesting line in the graph. In particular the large jump the line makes from 7 to 15 and from 16 to 22. These changes take place in revision 720 and 1380 respectively. Further inspection of these changes show us some interesting information. Firstly, these clones come from the testing part of the software. The Google Guice system is organized such that all the testing code is included in the project's source directory, hence the testing code is included in the clone detection phase of our tool by default. The testing of the code is done very thoroughly, with as much code coverage as possible. The result is that many tests are very similar and will result in a clone being detected by our tool. Another interesting thing to note from the changes in maximum clone size is that the largest clone group is not always the same. In the last change in size at revision 1380 a new batch of tests is introduced to test a new functionality. The result is that the clone group that had the largest number of relations previously still exists with the same size, but is no longer the largest.

In the graph it can also be seen that while the maximum clone size increases over time, the average clone size stays stable. This while the maximum usually has at least some influence on the average. This can be explained by the number of smaller clones being introduced to offset the increased maximum clone size: at the time where the maximum clone size is 7, there are a total of 115 clones in the system; by the time the maximum has reached 22, there are 333 clones in the system. This causes the average clone size to be virtually the same.

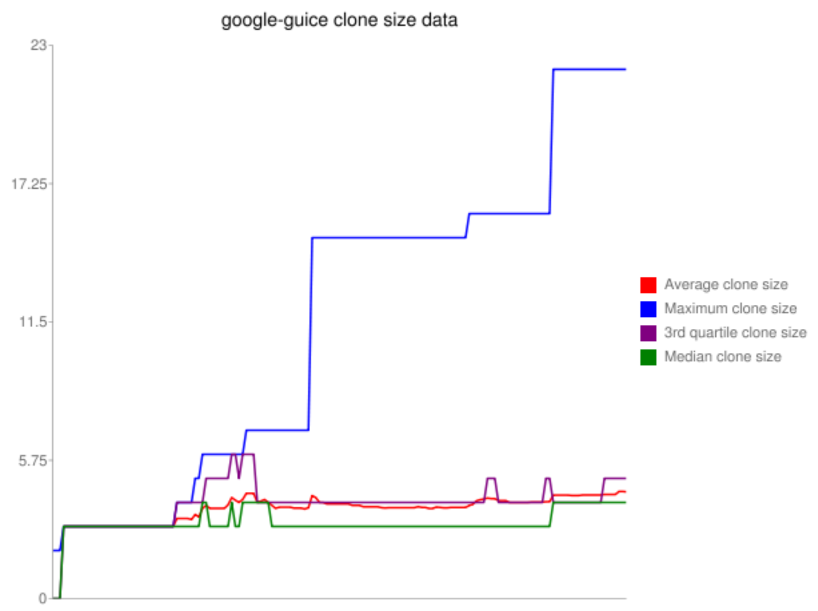

Figure 4.5: Clone size evolution Google Guice minus clone pairs

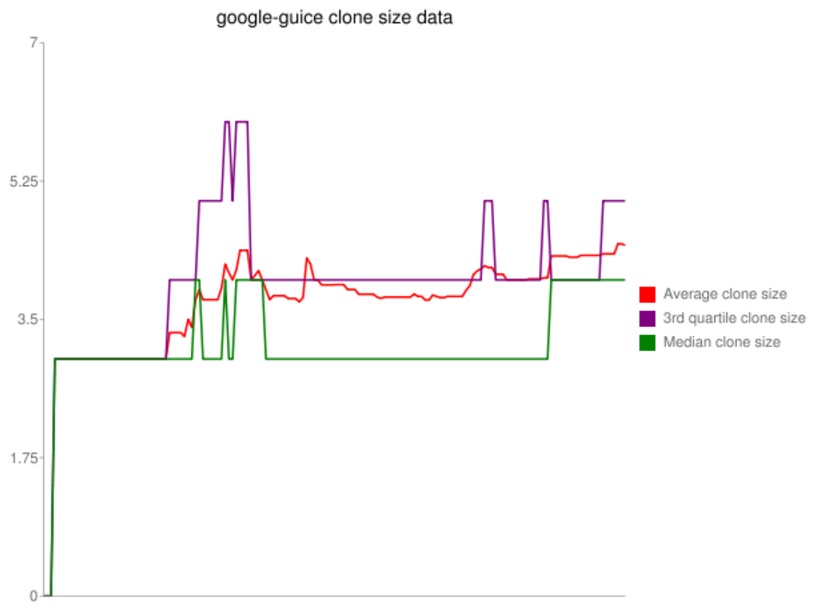

Figure 4.6: Clone size evolution Google Guice minus clone pairs, more detail

When the clone pairs are not included in the calculations, as in figures 4.5 and 4.6, again it can be observed that there are a large number of small clones of size 3 or 4. However, the graph of the third quartile suggests there are also a number of relatively large clones. The evolution of these larger clones is erratic, as there is a spike in the third quartile graph early on, after which it stabilises on 4.

Subclipse

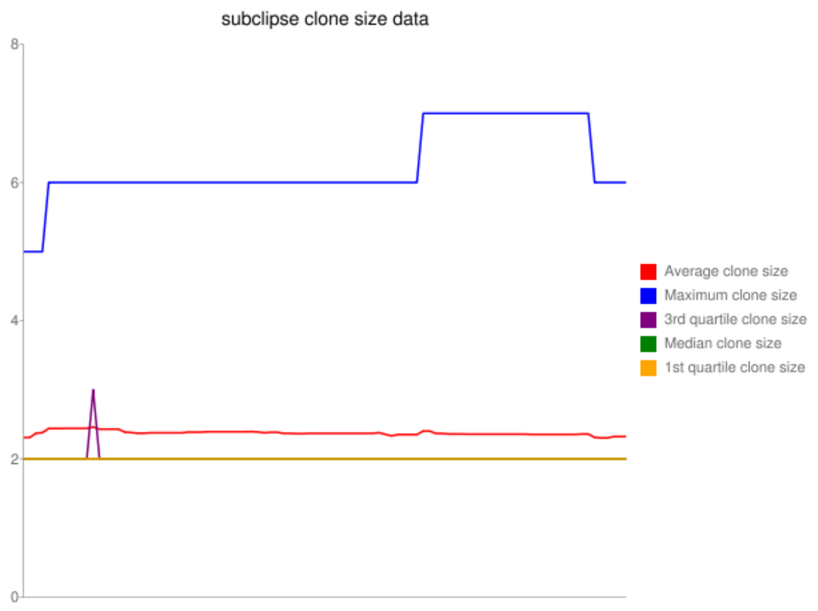

Figure 4.7: Clone size evolution Subclipse

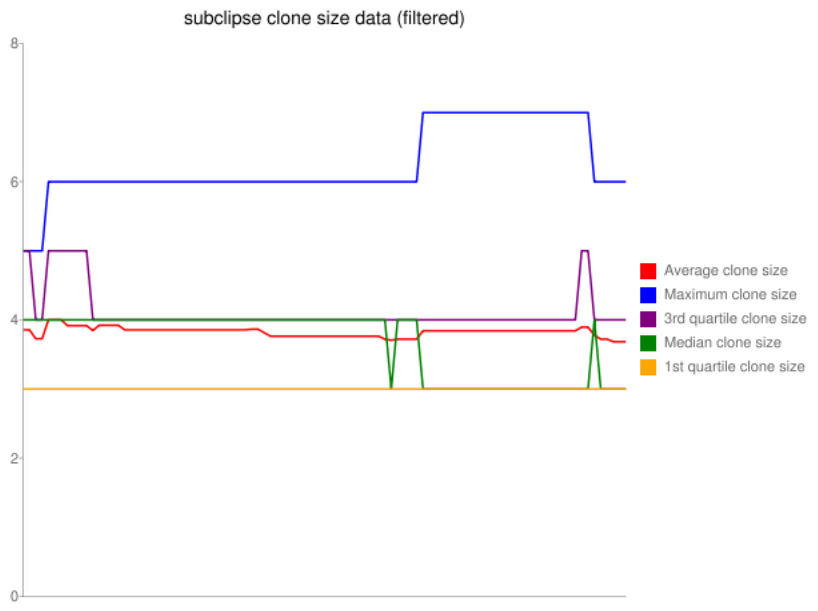

Figure 4.8: Clone size evolution Subclipse, no clone pairs

From figure 4.7 where the clone pairs are included, it becomes clear that clone pairs are again by far the most occurring type of clone, even more so than in the previous systems. Seeing as the third quartile most of the time is 2, it stands to reason that more than 75 per cent of all clones are clone pairs. This also results in a situation where the average clone size is greater than the third quartile. Another thing to note from these graphs is that the maximum is fairly stagnant, not increasing as in Google Guice. The maximum is also fairly small, with a size of at most 7 cloned fragments, compared to the maxima found in previous systems. Figure 4.8 shows that if clone pairs, the majority of clones, are left out of the calculations, there is more disparity between the quartiles of clone size.

Further inspection of the largest clones does not yield particularly interesting results: when it grows a number of cloned fragments are introduced, when the maximum decreases a bug was found in one of the cloned fragments and changed so that it was no longer part of the clone. The other cloned fragments apparently did not have this bug.

ArgoUML

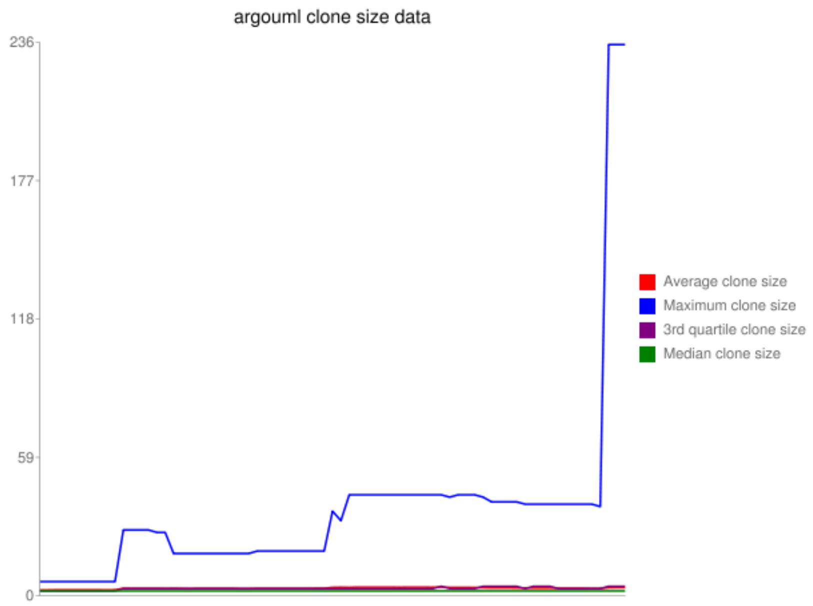

Figure 4.9: Clone size evolution ArgoUML

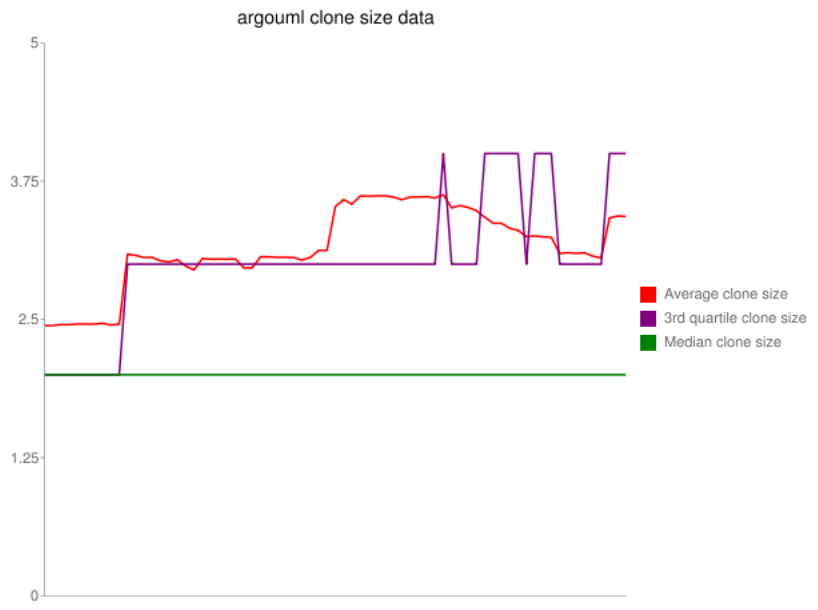

Figure 4.10: Clone size evolution ArgoUML, more detail

The graphs for ArgoUML show us yet another phenomenon. The maximum clone size is very large, while the median and third quartile are again very small. The growing pattern of the maximum is also of interest: it seems as though the maximum size first decreases before taking large jumps, after which it decreases again. The very large jump at the end of the measured lifecycle might be an area of interest to research further. This change is brought about by the introduction of a new functionality wherein a very large number of if statements are being evaluated with similar function calls being done if the case is true, see listing 4.9

```
if (n.equals("XMI")) handleXMI(e);
else if (n.equals("XMI.header")) handleXMIHeader(e);
else if (n.equals("XMI.documentation")) handleXMIDocumentation(e);
else if (n.equals("XMI.owner")) handleXMIOwner(e);
else if (n.equals("XMI.contact")) handleXMIContact(e);
... etc
```

Listing 4.9: XMIParserIBM.java largest clone in ArgoUML sample

This code shows perhaps a shortcoming of the Java language; there are languages where function calls can be dynamic such as PHP, allowing this piece of code to be written in a much more compact manner. However, each if statement cannot really be considered a seperate clone. The CCFinderX program in this case produced a false positive. However, before this spike in maximum clone size, it was already fairly large at 38. In fact, the maximum decreases in several occasions, and it might be interesting to look at why this happens.

A relatively steep decrease in the maximum clone size happens early on, from 27 to 18. The subversion history log messages are not very helpful in the ArgoUML project, most are left empty. Therefore we have to look at the code what exactly happens in this instance. It seems however, that in this case the clone size is not reduced because of refactoring. Rather it is reduced because several parts of the clone were changed to add functionality, for example the additional throwing of an exception, so that they were no longer detected as clones. Whether this is justified and that the rest of the cloned parts would also need to be changed in the future is outside the scope of this thesis.

The average clone size is fairly stable but decreases slowly and spikes in relative unison with the maximum. The number of small clones also increases over time in this system, hence the relative stability.

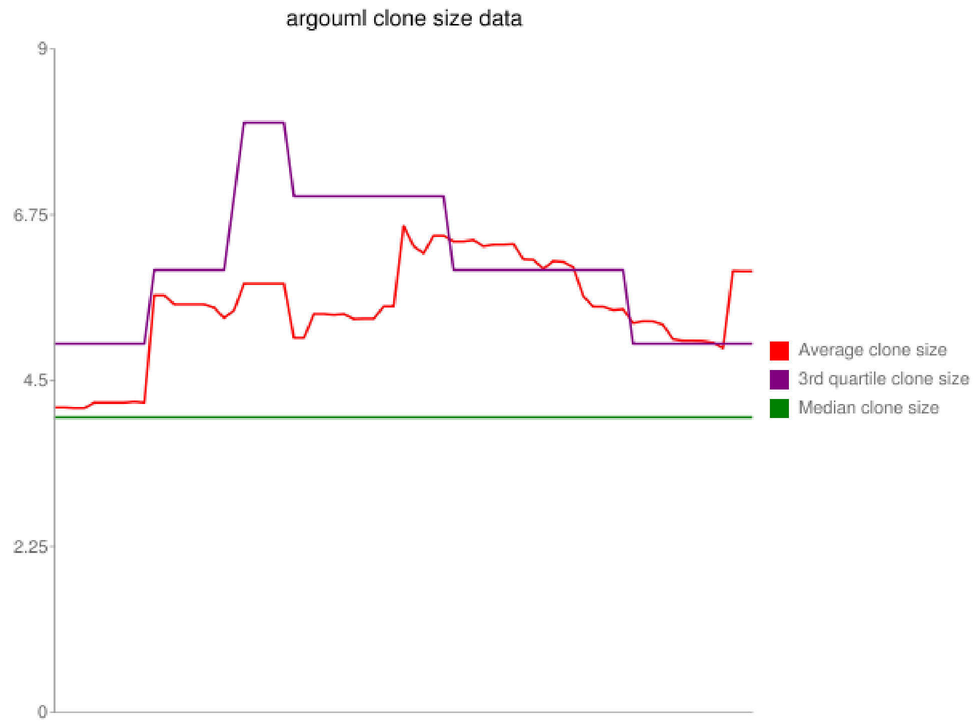

Figure 4.11: Clone size evolution ArgoUML minus clone pairs, more detail

A closer look at the clones of size greater than 2 within the ArgoUML project in figure 4.11 shows that there are indeed clones of moderate size, but even the third quartile does not come close to the maximum. Perhaps it might be worthwhile to take a closer look at a selection of the largest clones in future work, to see whether the second-to-largest clone comes close to the maximum, or is in fact closer to the third quartile in size.

4.2.3 Result analysis and discussion

Of the results obtained by the experiment we can make interesting observations. First of all, it is clear that in all software systems the number of clone pairs, or clones of size 2, are by far the most occurring type of clone. Clones in all systems that we have looked at consist for more than 50 per cent out of clone pairs. And for the clones of size larger than 2, the majority are still small of only size 3 or 4. Only a small group of clones can be considered relatively large, with more than 4 cloned fragments.

The maximum clone size as measured by the experiment also shows interesting patterns. Projects that can be considered small in scale, Charts4j and Subclipse over the timeframe that we have conducted the experiment on, have very differently evolving maximum clone sizes than larger projects. The number of developers on Charts4j is small, and for the first 1000 revisions of Subclipse the number of developers actively committing code was also small. Perhaps this is the reason that the maximum for both projects is smaller than the other projects. Having to keep track of all code as just one or a couple of persons can be more difficult than when a larger group of people works on the code. Aside from that, fewer people can simply not produce code as large and complex. Likely for these reasons, maxima of clone sizes in Charts4j and Subclipse do not grow very large, and can actually decrease relatively sharply in size as developers try to keep the size of large clones managable.

In general we can answer the question of how the size of code clones evolves within a software system as follows. There is no evidence that the average clone size increases significantly over the lifespan of a software system. The maximum clone size on the other hand, especially on larger projects with more developers, clearly grows steadily over time. A question that arises from these observations is why there is a disconnect between the maximum clone size and average clone size. Usually the largest number in a set weighs quite heavily on the average. This question can be answered by looking at the total number of clones. This number increases just as fast as the maximum increases, and because a large part of the clones are small in size, this drags down the average clone size. This in combination with the increasing maximum clone size pulling the average up, the end result is that the average remains relatively stable over the lifetime of a project.

4.3 Clone Deletion Experiment

The next experiment we execute looks closer at the deletion of clones in the selected systems.

4.3.1 Experiment

A deletion occurs when the entire clone group is removed, or in practice it is no longer detected by CCFinderX from one revision of the project to the next. We do not take into consideration the possibility of a clone being removed for one or more revisions and returning at a later stage. Such a situation might occur when a clone was accidentally changed, but usually when this happens it is just one fragment of the clone and the clone should still be detected in the next revision if just one fragment is removed. There are reasons why a clone might be deleted in one revision and return in the next, but we consider this event relatively unlikely. Aside from that, there is no reason to assume that a clone of one certain size might be more likely to be deleted and return in a later revision than a clone of another size. Since we mainly focus on the differences in evolutionary behavior between different sizes, and for the reasons stated above, we have chosen not to consider the special case of a returning clone. So when this situation occurs, a deletion will be detected by our system and taken into our calculations.

A clone might be removed for various reasons. The clone might be refactored because the developer no longer finds it practical to keep track of several cloned fragments of code. This is an interesting reason to investigate further because it assumes the developer is consciously aware that there is a clone and keeps track of its fragments. Other reasons a clone might be removed include changes in functionality in certain fragments of the clones, removal of cloned functionality, or CCFinderX no longer detecting the clone for other reasons (for example it becoming too short in length so that the algorithm no longer detects it).

We are interested in the size of clones, and in relation to the deletion of clones we are particularly interested in seeing whether the size of a clone has an influence on whether the clone will be deleted. If a clone has many fragments - its size is large - it might seem logical to assume it is harder to keep track of all the fragments. Thus, it might be more probable for the clone to be refactored. To research whether this is indeed the case, we ask the following research question:

- **Is there an affinity between the size of a clone and its probability of being removed?**
  To be able to answer this question, the term 'probability of being removed' must first be properly defined. There are more ways to look at this term. Of course, we cannot calculate the actual definite probability of any clone being removed at any time. We can only look at the past and see how many clones were removed in absolute as well as relative sense. The size of the clones that we look at also needs to be defined. Having established in the previous experiment that the number of large clones is very small, it is not useful to take into account every single different clone size existing in a project.

Now that we have stated the research question for this part of our research, we need to define the terms in which it is posed. We establish the probability of a clone of a certain size being removed as follows: for each revision in which a clone is removed we calculate the number of clones of a certain size that are removed, divided by the total number of clones of that size. We then calculate the average of all these instances to produce the probability over the entire lifetime of the project. The above definition uses the term 'of a certain size'. We do not distinguish between all possible different sizes. Instead, we divide clones in the following categories:

- **Small clones of size 2**
  The vast majority of clones in every project that we have selected, as has become clear while conducting the previous experiment, is of size 2. For this reason, and because 2 is the smallest size a clone can have, we label these small clones.

- **Medium size clones of size 3-4**
  As was also established during the previous experiment, the majority of clones aside from those of size 2, was size 3. And the third largest set was of size 4. The reason we distinguish these from small clones is that we want to find out whether the size

difference also translates into a difference into the probability of being removed. For this reason, we label clones of these sizes as medium size.

- **Large clones of size > 4**
  Clones larger than size 4 are in all projects a relatively small group. It is still possible to find a handful of clones of size 5 and 6, but larger than that it is quite uncommon to find more than one clone of the same size. In fact, it is quite possible to have one clone of size 10, then no clones of size 11, 12 or 13 an one more of 14 for example. Because we want a reasonable sample size, we take all clones of size 5 and larger and label them large clones.

Having established the terms of our research question, we can now describe the experiment. For all the selected projects we will calculate for clones of the defined size categories their probability of being removed. As defined, this is done by calculating for every revision their chance of being removed and averaging that chance over all revisions.

4.3.2 Results

The results of this experiment are presented below. For each project we show a table with the results of the calculated probability of a clone being deleted, rounded to 4 decimals, with explanations where appropriate.

Charts4j

Table 4.1: Clone deletion probability for Charts4j

Clone size category	Deletion probability
Small clones	0.0068
Medium clones	0.0074
Large clones	0.0052

The probability of each size category being removed for the Charts4j project is very similar and quite small, as can be seen from table 4.1. The Charts4j is not very large and the number of developers is low. Also, as we have seen in the previous experiment, the number of large clones is relatively small. This might mean that the developers can keep track of the existing clones quite well and do not see the need to refactor them completely. Also, because the project is relatively young, it means that more time is dedicated to expanding functionality, rather than refactoring existing code.

Google Guice

The first thing of note in the Google Guice table 4.2 is that the probabilities for all size categories are substantially larger than in the previous project. This can be explained by

Table 4.2: Clone deletion probability for Google Guice

Clone size category	Deletion probability
Small clones	0.0534
Medium clones	0.0545
Large clones	0.0063

the fact that there are revisions in which the entire packaging of classes was overhauled, prompting a deletion for 100% of the clones in our deletion detection algorithm for that revision, spiking the average probability of removal higher. The reason this happens has been explained in an earlier chapter, but comes down to the fact that because the CRD of clone fragments are no longer similar, the clone can no longer be detected as equivalent to the one in the previous revision. While these deletions are essentially false positives in the deletion detection algorithm, this should not affect the conclusion drawn from these results because it is the difference between probabilities that matters. Because this false positive affects all categories equally, it does not affect the difference between the categories and hence does not influence the conclusion.

Another thing that stands out from the table is that the probability of large clones being removed is substantially lower than the other size categories. Whereas the probability of a clone of size 2 to be removed is over 5%, the probability of a large clone being removed is only 0.6%. We can speculate, but from our investigation of this project we cannot see an obvious reason for this phenomenon. It might be the case that large clones simply have a lower probability of being removed.

Subclipse

Table 4.3: Clone deletion probability for Subclipse

Clone size category	Deletion probability
Small clones	0.0082
Medium clones	0.0046
Large clones	0.0011

Table 4.3 for Subclipse shows us yet another phenomenon. The probabilities for small and medium size clones to be removed lie closer to that of Charts4j. The probability of large clones being removed is in turn much smaller than those of small and medium size clones. The reason for this might have to do with the fact that Subclipse, as we found out in the previous experiment, has very few clones that we consider to be ‘large’ and the clones do not grow as large as in other projects. This might be due to the fact that the developers

consciously avoid creating large clones. This means that the large clones they do have are absolutely unavoidable and hence will likely not be refactored. Thus, while the fact that there are few large clones in itself does not necessarily lower the probability of large clones being removed directly, it might indirectly lower it.

ArgoUML

Table 4.4: Clone deletion probability for ArgoUML

Clone size category	Deletion probability
Small clones	0.0344
Medium clones	0.0281
Large clones	0.0125

The ArgoUML project is closer to Google Guice when it comes to clone removal probabilities, as can be seen from the results in table 4.4. In this project too the entire package structure was changed, causing false positives on the deletion detection algorithm. The difference between large clones and smaller clones is less pronounced than in Google Guice, but it is clear that clones of larger size seem still less likely to be removed.

4.3.3 Analysis

All projects except Charts4j show us that the probability of a large clone being removed seems actually lower than that of smaller clones. This is against our expectation that larger clones would be more likely to be removed. It is still questionable whether this is an anomaly in our data or whether this shows that large clones are always less likely to be removed. We can use statistics to answer this question.

Based on our findings, we can use statistical hypothesis testing to find out whether our results can answer the research question posed earlier: Is there an affinity between the size of a clone and its probability of being removed? Our findings suggest that if this is the case, the connection would be inversed, that is to say the size of the clone would inversely impact the probability of it being removed, or ‘the larger the clone in size, the less likely it is to be removed’. Therefore we can formulate an hypothesis that would support these findings, and then use statistical hypothesis testing to find out whether this hypothesis can or cannot be rejected. The hypothesis that we have formulated is as follows:

H_1: The size of a clone inversely impacts its probability of being removed from a system.

To test whether this hypothesis can be accepted as true beyond a reasonable doubt we make this the alternative hypothesis. We also state a null hypothesis which assumes the H_1 to be false. We assume the null hypothesis is true unless we can reject is in favor of the alternative hypothesis. The null hypothesis is formulated as follows:

H_0: The size of a clone does not impact its probability of being removed from a system.

Using statistical hypothesis testing we test whether we reject H_0 in favor of H_1 by using our findings of the experiment. We assume the outcome of those experiments follows a normal distribution with the outcome of the calculated probability for the *small* category as the expectation μ and half the difference between the probabilities for the *small* and *medium* size categories as the variance σ^2. Now, if the outcome of the probability for the *large* clone size category lies within the critical section of this distribution, we have proven beyond a reasonable doubt that the null hypothesis H_0 is false and we reject it in favor of H_1. The critical section of a distribution is those numbers that have a very small chance of occurring randomly. For the significance level we choose $\alpha = 0.05$, so the critical sections of the distribution are the top and bottom 2.5% of the possible outcomes. We can now set up the following formula to solve for one of the result sets:

$$P(X = x|H_0) = P\left(\frac{X-\mu}{\sigma} \geq \frac{\mu-x}{\sigma}\right)$$

In this formula, X is the distribution of our results and x our result for the large category. For example, with our results from Google Guice we would use the parameters $\mu = 0.0534$ and $\sigma = 0.00055(0.5(0.0545-0.0534))$, resulting in the following formula ($Z_{0.05} = 1.645$):

$$P(X = 0.0063|H_0) = P\left(\frac{X-0.0534}{\sqrt{0.00055}} \geq \frac{0.0534-0.0063}{\sqrt{0.00055}}\right) \approx P(Z \geq 2.01) = 0.0222$$

This result we multiply by two because we are calculating a two-tailed p-value, which is 0.0444, within the $\alpha = 0.05$ we set as significance level. Therefore, the result we found for the probability of large clones being removed from a project falls in the critical section and we have enough reason to reject H_0 in favor of H_1.

For the results gathered from the Google Guice project, the null hypothesis can thus be rejected. For the other projects, we can apply the same method to decide whether to accept or reject the same null hypothesis with the following calculations. For Charts4j we have $\mu = 0.0068$, $x = 0.0052$ en $\sigma^2 = (0.0074-0.0068)/2 = 0.0003$:

$$P(X = 0.0052|H_0) = P\left(\frac{X-0.0068}{\sqrt{0.0003}} \geq \frac{0.0068-0.0052}{\sqrt{0.0003}}\right) \approx P(Z \geq 0.09) = 0.4641$$

For Subclipse we the values $\mu = 0.0082$, $x = 0.0011$ and $\sigma^2 = (0.0082-0.0046)/2 = 0.0018$, resulting in the following equation:

$$P(X = 0.0011|H_0) = P\left(\frac{X-0.0082}{\sqrt{0.0018}} \geq \frac{0.0082-0.0011}{\sqrt{0.0018}}\right) \approx P(Z \geq 0.17) = 0.4325$$

Finally we fill in the values for AgroUML with $\mu = 0.0344$, $x = 0.0125$ and $\sigma^2 = (0.0344-0.0281)/2 = 0.0032$:

$$P(X = 0.0125|H_0) = P\left(\frac{X-0.0344}{\sqrt{0.0032}} \geq \frac{0.0344-0.0125}{\sqrt{0.0032}}\right) \approx P(Z \geq 0.39) = 0.3483$$

As we can see, none of these values fall in the critical section of distribution, leaving us to conclude that in these projects, there is no connection between the size of a clone and its chance of being removed. These results are not surprising: the results gathered from the Charts4j project for example are clearly much closer together than those of Google Guice. The results for ArgoUML are not quite as close, but because the difference between small and medium clones is larger, the variance is also larger resulting in a higher value to be divided by. Now, because of these differing results, one might wonder whether a definite conclusion can be drawn. However, we can conclude that for at least one of our researched projects we have proven beyond reasonable doubt that the size of the clone impacts its probability of being removed. This does not mean that the affinity is not there in the other projects, it only means that we could not prove it to be there beyond a reasonable doubt. The research question we posed earlier, *Is there an affinity between the size of a clone and its probability of being removed?*, can thus be answered. Yes, in at least one of the researched projects we can observe there to be an undeniable affinity between the size of a clone and its probability of being removed. This affinity is observed to be inversed, so the larger the clone in size, the smaller the probability of being removed. While this affinity might not exist in all projects, there is certainly a good chance it does.

4.4 Clone Size Reduction Experiment

Aside from being deleted, clones can also have their size reduced. Below we describe the experiment in further detail and present the results.

4.4.1 Experiment

This experiment researches another occurrence that can happen to clones, namely having their size reduced as opposed to being completely deleted. As we recall, the size of a clone is the number of fragments in a clone group. This number can be lowered by fragments being refactored and no longer being detected as cloned segments. An obvious example of when this might happen is when the developer introduces cloned code to set up a placeholder for new functionality. He then changes this code to implement the new functionality that would be different from the cloned code. Hence the fragment is no longer detected as a clone. Other examples of cases when a clone's fragments might be refactored are of course to deliberately remove the clone, or to remove functionality that the cloned code provided that is no longer needed. Throughout this section we use the phrases 'clone size reduction' and 'clone fragments removed' interchangeably, and these phrases refer to the *subtract* pattern described in section 2.3.

It should be noted that when we measure a clone size being reduced, we do not keep in mind the amount by which the size is reduced. Therefore, when a clone is removed we also consider this as a case of its size being reduced - namely to one or zero. In fact when a clone pair (size two) has its size reduced, the only possible outcome is the entire clone being removed.

We want to find out whether the size of a clone has an impact on the probability of it being reduced in size. In the previous section we found out that larger clones are not more likely to be removed than smaller clones, in fact the opposite turned out to be the case. Here we want to find out whether a larger clone has a larger chance of having its size reduced. Or to word it differently, we want to answer the following research question:

- **Is there an affinity between the size of a clone and the probability of clone fragments being removed from that clone?** The terminology 'probability of clone fragments being removed' needs some clarification in this context. Similar to our last experiment, we define this as the number of clones of a certain size category that have at least one fragment removed, divided by the total number of clones of that size category. We calculate this for every mined revision of our selected projects, and then average this probability out over the lifespan of the project by adding all the probabilities and deviding by the total number of revisions.

The same size categories small, medium and large apply as in the previous experiment. For a complete definition of these categories, see the previous section. Before going into the results, we cannot really predict the results, but we can point out some things with certainty. Since we do not make a distinction between one or more cloned fragments being removed, we automatically include all the cases where the entire clone is removed. Therefore, the probability that a clone of the category *small* will have a fragment removed is equal to its probability of being removed. Also, we can state with certainty that the probability of clones of the *medium* and *large* categories having fragments removed will be higher than or equal to their chance of being removed, since this case is included in the calculations. Therefore, we are really looking at whether a large clone has a significantly larger chance of being reduced in size than a small clone, or whether the probabilities do not differ much.

4.4.2 Results

Now that the research question and experiment parameters are established, we can present the results before analysing them and drawing a conclusion. The results are presented in a similar fashion as in the previous section, with probabilities being rounded to 4 decimals. Also, behind the probability we have added the amount this probability has increased from the equivalent result of the previous experiment in parentheses.

Charts4j

Table 4.5: Clone size reduction probability for Charts4j

Clone size category	Reduction probability
Small clones	0.0068 (-)
Medium clones	0.0074 (-)
Large clones	0.0059 (+0.0007)

The results for Charts4j in table 4.5 show us very similar results as in the previous experiment, with the probability of large clone losing fragments very slightly higher than their probability of being removed completely. This might be down to the fact that the probabilities were already fairly similar. The fact that this project has few developers could explain why the refactoring of clones (either fragments or entire clone groups) does not have a high priority. Hence we would find a small number of occurrences of removal of clone fragments or entire clones.

Google Guice

Table 4.6: Clone size reduction probability for Google Guice

Clone size category	Reduction probability
Small clones	0.0534 (-)
Medium clones	0.1154 (+0.0609)
Large clones	0.0097 (+0.0034)

The probabilities for with which clones are reduced in size in Google Guice, outlined in table 4.6, show us yet another phenomenon. The medium size category has a significantly higher chance of having their size reduced than the small clones. Also, this probability is much higher than just their chance of being deleted alltogether. It seems clear that something is making clones of this size more likely to be refactored. A possible explanation would be that the developers often introduce a handful of cloned fragments as a placeholder for new code and then work on these fragments refactoring them to implement the new functionality. Another explanation might be that the clones of this size category are simply more difficult for developers to keep track of and they deliberately refactor them to make them more manageable.

The large clone category probability of being reduced in size also slightly increased over its probability of being removed completely. However, this probability is still quite low compared to the other size categories. It is interesting to see that this number increased much less than that of medium sized clones.

Subclipse

The results in table 4.7 for Subclipse show very little difference in the probabilities from the previous experiment. This is down to the fact that the project has very few clones to begin with, especially clones that we classify as large. The handful of clones that are large hardly get touched at all, except a few cases where an entire clone group is removed. The developers clearly have a good understanding of the code and see no need to change much in the cloned code, indicating that the code is stable and there is not much need to change functionality or fix problems.

Table 4.7: Clone size reduction probability for Subclipse

Clone size category	Reduction probability
Small clones	0.0082 (-)
Medium clones	0.0059 (+0.0013)
Large clones	0.0011 (-)

ArgoUML

Table 4.8: Clone size reduction probability for ArgoUML

Clone size category	Reduction probability
Small clones	0.0344 (-)
Medium clones	0.0448 (+0.0167)
Large clones	0.0384 (+0.0259)

Table 4.8 with ArgoUML's probabilities of being reduced in size is more like the result we expected beforehand. Both the medium and large clones' probabilities are much closer to that of the small clones, and they both increased significantly over the probabilities reported in the previous experiment. As a detail, it is interesting that both the medium and large clones have a slightly higher probability to have their size reduced than small clones. This might be down to the fact that because they have more cloned fragments, they are more likely to be hit by random refactorings or unintentional changes. But then the larger clones would have a higher probability than the medium clones. However, because there are more medium sized clones, it is perhaps more likely that a fragment that is refactored belonged to a medium sized clone than a large one.

4.4.3 Analysis

The results of the experiment give us new insights into the evolution of clones. Particularly the Google Guice and ArgoUML projects show us results that need some further analysis.

Table 4.9: Clone evolution data

Size	Charts4j			Guice			Subclipse			ArgoUML		
	Del	Red	Diff	Del	Red	Diff	Del	Red	Diff	Del	Red	Diff
Small	0.0068	0.0068	-	0.0534	0.0534	-	0.0082	0.0082	-	0.0344	0.0344	-
Medium	0.0074	0.0074	-	0.0545	0.1154	+0.0609	0.0046	0.0059	+0.0013	0.0281	0.0448	+0.0167
Large	0.0052	0.0059	+0.0007	0.0063	0.0097	+0.0034	0.0011	0.0011	-	0.0125	0.0384	+0.0259

For reference, all the results from the previous two experiments are compactly presented in table 4.9. The shorthands used in table 4.9 to keep it compact are *Del* for probability for a clone to be deleted, *Red* for the probability for a clone to be reduced in size and *Diff* for the difference between the two probabilities.

The fact that Charts4j and Subclipse have less different results as compared to the previous experiment is also noteworthy though. This might be down to the lower levels of activity on their repositories and generally fewer clones. Also, the fact that these systems are maintained by fewer developers might result in there being less changing of clones. After all, when there are few developers, they usually are able to keep track of all the fragments in a clone better and if changes are made in one fragment they would also apply these changes in other fragments. The Subclipse project in particular seems to have been developed with the intent of keeping clones few and consistent, that is why we see very little activity in clones in this project.

The Googe Guice and ArgoUML results are quite different, both from the previous experiment and from each other. If we take a closer look at the results that we have gathered from this experiment, we can see for both the medium and large sizes a substantial increase from the deletion experiment. We can present this difference in terms of percentual changes, so that it is more understandable and easier to explain further. Note that these percentages do not represent the actual chance a clone will be reduced in size, merely the percentage of change from the chance of it being removed entirely. For Google Guice the chance for a medium sized clone to be removed entirely was 0.0545 and the chance of a clone being reduced in size is 0.1154 for a difference of 0.0609. This is a difference of 111% of the original value, in other words the chance for a clone of medium size to be reduced in size is more than twice as high as being removed completely. For large clones in the Google Guice project this difference in probabilities is 54%. For the ArgoUML project these differences for medium and large clones respectively are 59% and 207%. The question is whether we can conclude anything from these figures.

At first glance, the fact that the increase in medium clones is larger than the increase in large clones in the Google Guice project, while in the ArgoUML project it is the other way around does not seem to support this claim. However, some further inspection might be necessary before we can conclude anything. The thing that might have an influence on the gathered results would be the overall number of clones of the appropriate categories. To get an idea of this number, or rather the difference between the number of large and medium clones in a project, we can count all the clones of these specific size categories per revision. We can then graph these amounts over the lifetime of the project. Here are these graphs for the Google Guice and ArgoUML projects, with time on the x-axis and the number of clones on the y-axis:

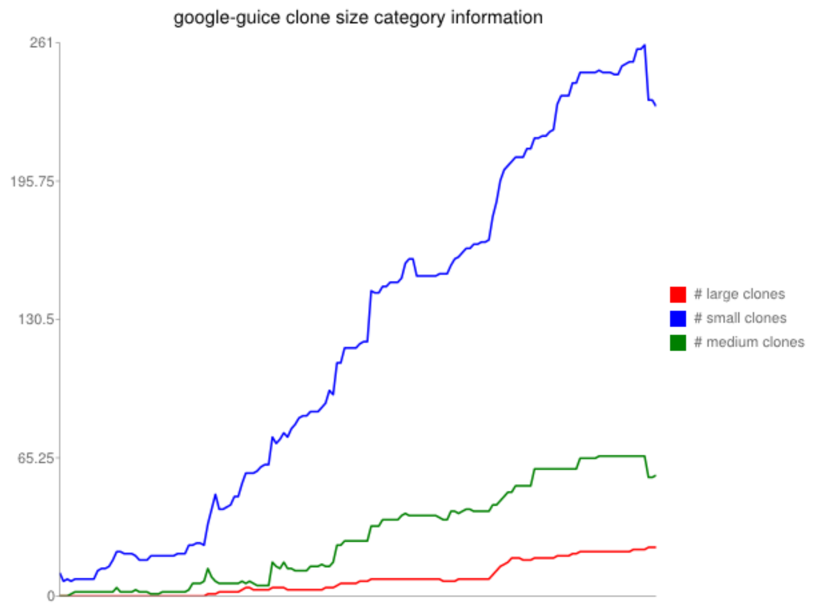

Figure 4.12: Number of clones per size category for Google Guice

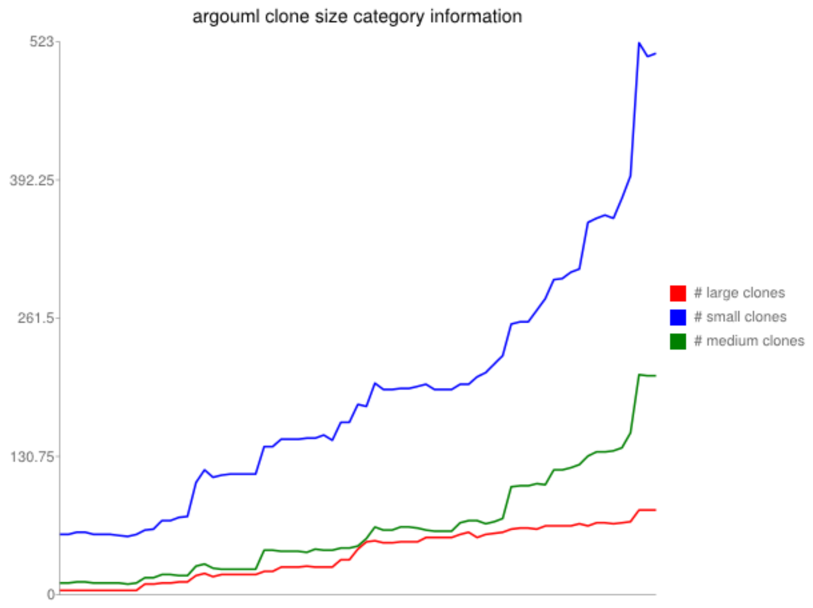

Figure 4.13: Number of clones per size category for ArgoUML

From these graphs we can see that the number of large clones in the ArgoUML project compared to the number of medium clones is larger than is the case in the Google Guice project. Therefore, we must keep in mind that a reduction in the size of a clone might be caused by a ‘random’ change in a piece of code, in which case a large clone is more likely to be affected than a smaller clone, especially if large clones are more numerous. This would account for the fact that the large clones in ArgoUML see a large increase in their chance of being reduced in size. In the Google Guice project however, the chance of a medium size being reduced in size is greater than that of larger clones, and the increase over their chance of being removed completely is also substantial. The fact that there are more medium sized clones as seen in graph 4.13 might partially explain this. Still, this could point towards a theory where medium sized clones are more likely to have fragments removed. By using the same type of statistical hypothesis testing we used in the previous section, we can decide whether or not this result supports this theory. We first set up a null hypothesis:

H_0: The size of a clone does not impact the probability of having fragments removed from it.

And the alternative hypothesis:

H_1: Medium sized clones have a higher probability of having fragments removed from them than clones of other sizes.

Now, because we are testing clones of medium size, we set the expectation μ and the variance σ^2 accordingly: for the expectation we still choose the the result of the probability for *small* clones, for the variance we choose half the difference between the probabilities between the *small* and *large* clones. Because we are still using the same normal distribution and formula as in the previous section, this results in the following computation:

$$P(X = 0.1154|H_0) = P\left(\frac{0.0534-X}{\sqrt{0.02185}} \geq \frac{0.1154-0.0534}{\sqrt{0.02185}}\right) \approx P(Z \geq 0.421) = 0.3372$$

This result is not significant enough to conclusively support the alternative hypothesis, therefore we must reject it and accept the null hypothesis for this project. Because the results we have gathered from Google Guice are the most differentiated of all the results, and because these results are not significant enough to conclusively accept an alternative hypothesis, we do not need to do the computation for the other results. However, we can now draw a conclusion from these results.

The question we asked at the beginning of this section was whether *"Is there an affinity between the size of a clone and the probability of clone fragments being removed from that clone?"*. The results we have gathered with regards to these probabilities for the three different clone size categories we have defined are so close together that we observe no conclusive evidence towards a conclusion in support of an affinity between the size of a clone and the probability of fragments being removed from it. While this does not necessarily mean there absolutely is no affinity for every project, we still have to conclude that no, our results do not conclusively support that there is an affinity between the size of a clone and the probability of fragments being removed from that clone.

Chapter 5

Conclusions and Future Work

In this chapter we will first give an overview of the contributions our experiments and this thesis provides in section 5.1. Next in section 5.2 we go over the results once more and draw conclusions from them. Finally in section 5.3 we discuss the findings of this thesis.

5.1 Contributions

The contributions of this thesis in the field of evolution of code clones are twofold: there is the tool that we have created that provides functionality not previously available, and there are the results gained from running experiments with this tool.

The tool we have created presents a number of novel contributions to the field of software evolution. The repository mining part is perhaps not novel, it was necessary to develop this part of the tool in such a way that it would be easy for the rest of the tool to interface with the mined repositories. The clone detection done by CCFinderX is not unique, but the analysis of the files as detected by CCFinderX is. We analyze the data produced by CCFinderX and in turn produce an abstracted version of the clones in Java objects that are easy to interface and potentially expand or alter. To our knowledge, the tool we have created is also the first to be able to track clones detected by CCFinderX over multiple revisions. Another contribution the tool has provided is the ability to reason over the analyzed clones to be able to obtain a great deal of evolutionary information from this data. We have provided a framework for extracting information regarding the size of clones over time, but the reasoning happens over clearly interfaced abstract clone models so that the functionality could easily be expanded. Therefore, overall the tool provides novel functionality with regards to clone tracking over several revision of a software project and contributes as well in the sense that it provides a framework which can be built on further.

In terms of the results gathered from our experiments, we have also made contributions to the field of software evolution with regards to code clones. The results show us important new findings about the evolution of clones. First of all, we have gained an insight into the distribution of clone populations in terms of their size. In other words, we have shown such information as the median, 3rd quartile and maximum size of clones over the lifetime of a project. From this data a better understanting of the evolution of clones is achieved.

Furthermore, we have researched the probability of a clone being removed or a clone losing a fragment. The fact that clones with larger number of fragments are not necessarily more likely to be removed sheds new light on code clones and how they are maintained.

Concluding, this thesis has provided novel contributions in the forms of a tool that can be used to research code clones and results from experiments providing new insights into the evolution of clones.

5.2 Conclusions

The research done in this thesis allows us to draw several conclusions. Most importantly, we can answer the research questions of this thesis. As we recall, the main goal of this thesis was to study the influence that clone properties, particularly clone size, have on their evolution. To this end, we asked several research questions in our introduction. To start, we wish to research the general evolutionary patterns of clone sizes, for which we asked and answered the following research question:

- **How does the size of clones evolve over time?**
  As we have seen from the graphs presented in section 4.2, there are several things that can be said about the evolution of the size of clones. The first thing to note is the fact that in all projects the most occurring type of clone is a clone with just two fragments, or a clone pair. Clone pairs account for more than half of all the clones we have detected in our four projects over their entire lifetime. The next most numerously occurring clone type would be that of what we consider a medium sized clone: a clone with 3 or 4 fragments. Only a small amount of clones existing in a project are of size 4 or larger. In projects that are generally smaller in scale, with fewer developers and fewer revisions, the maximum clone size does not increase over the lifetime of a project. It seems as though the developers consciously try to keep the clones small, so as not to loose track of them. In larger projects though, we have seen that the maximum clone size will increase over time, and can grow to significant sizes. For example, as we can see in the case of ArgoUML, which at some point has a maximum clone size of 236. However, at the same time we observe that the average clone size does not significantly increase over the lifetime of projects. We argue that this phenomenon takes place because the increasing maximum size, which would normally weigh heavily on an average, is offset by an increase in the number of smaller clones.

The next part of the conclusions we can draw from our research comes from the research on the deletion of clones. To be able to research this area, we asked and answered the research question:

- **Is there an affinity between the size of a clone and its probability of being removed?**
  Doing the experiment regarding the size of a clone and its probability of being removed completely from the project has shown us some interesting information. A

clone is considered to be removed when it is detected in one revision of the project and no longer in the next. Our tool detects a clone deletion by iterating over the revisions of a project and detecting clone deletions by checking whether each clone it detects exists in the previous revisions. We then generate statistics out of this by dividing all clones in size categories (small, medium and large), and checking how many clones are deleted per category for each revision. By adding together the figures for all revisions, we establish for every size category the deletion probability of clones.

The results gathered from this process show us several things. First, for every project but one, the probability of large clones to be deleted is the smallest of the defined size categories. This suggests that the larger the clone is, the less likely it is to be removed. However, it is conceivable that these results are by chance, that in reality there is no connection between the size of a clone and its probability of being removed. To find this out, we have applied statistical hypothesis testing to our results. This tests whether or not a significantly differing result could be likely happen by chance in a normally distributed set of results. The result of this statistical hypothesis testing was that for one project, namely Google Guice, the collected result was not likely to happen at random. While the results of other projects are not so significant that they could not have happened with a high probability by chance, they still reinforce the theory that there is indeed a connection between clone size and probability of being removed. Therefore we can conclude that yes, for some projects there is an affinity between the size of a clone and its probability of being removed at some point of its lifetime in a project. The larger the clone is, the less likely it is to be removed.

The last research question we ask is:

- **Is there an affinity between the size of a clone and the probability of clone fragments being removed from that clone?**
  The next question we ask is related to the previous question. The difference between a clone being deleted entirely or merely being reduced in size, is that in the case of a deletion all the fragments are removed, in the case of a size reduction one or more but not all fragments are removed. Removed in this sense really means that the fragment is no longer detected as a cloned fragment, not necessarily literally removed. The detection of size reductions is done in a way similar to the method of the previous experiment. The results of this experiment show us an interesting phenomenon. One might think that because the chance for a clone to be removed is lower when it is larger, it would also be less likely that a large clone is reduced in size. However, because of the way that the reduction of size of a clone is calculated, this chance actually includes the chance that the clone is removed entirely. After all, when a clone is removed, a size reduction did take place, namely to zero. Therefore, small clones which are clones of size 2, only have one way in which they can be reduced in size, and that is to be removed completely.

  The result is instead that the probabilities for medium and large clones are higher, most noticeably in two of the projects. Statistical hypothesis testing of the most differentiated result however, showed us that this result is not significant enough to support the conclusive existance of an affinity. Hence, we conclude that this research question can be answered negatively. No, we do not observe an affinity between the size of a clone and the probability of a clone fragment being removed from it.

The main conclusion we can draw from our research is that small clones, particularly clone pairs, play a very important role in software evolution. Not only are they by far the most numerous, they are also most likely to be candidates for refactorings. Large clones weigh less heavily in projects, because while they might grow ever larger size, small clones grow faster in number. The effect of the size of a clone on its evolution is therefore such that a smaller clone will see more rapid evolution. It is therefore advisable that developers keep in mind that creating even small clones requires alertness when it comes to software evolution.

5.3 Discussion/Reflection

The research presented in this report provides new insights on the evolution of clones. The distribution in which clones of different sizes exist over the lifetime of a project and the affinity between size of a clone and its probability of being removed both show patterns of smaller clones being more important in the observed evolution of clones. This pattern is less pronounced in the experiment on clone size reduction, and while the results of this experiment are interesting, they are not as conclusive. The underlying cause of these observed patterns can be explained in more than one way. Developers might be inclined to keep clones small if at all possible. Only if it is absolutely necessary will a developer create and keep a large clone, and because it is necessary it is unlikely to be removed. Therefore, larger clones do not experience as rapid evolution as do smaller ones. Also, because developers usually work on one thing at a time, they might look for existing functionality to help them implement the new functionality, in which case a small clone would be created. It is difficult to create many functionalities at once, and the same goes for the evolution side. It is unlikely that a developer would change all fragments of a large clone at once, it would be more likely that he changes the fragments one by one, which also explains why the size reduction is more probable that an entire removal.

Overall, the research presented in this report sheds new light on the impact of size on a clones evolutionary behavior. Interesting patterns were observed that can help us understand and explain the underlying causes of developers behavior in software evolution.

5.4 Future Work

The research presented in this report builds on existing work in the field of software evolution with regards to code clones. However, with this research the work in this field is not yet done, and there are a number of points we have identified for future work.

Firstly, with regards to our tool, there are certain things that can be improved. The way our tool is implemented, it is limited in its ability to handle very many revisions of large projects. This is because all the clone data from the detection phase is loaded in memory for every revision, with the purpose of being able to access it easily for reasoning. This method is very fast and easy to work with, but runs into problems when dealing with very large numbers of revisions. We found that for large projects, 1000 revisions was about the limit of what the tool could handle. This limitation would be an area that could be improved with future work. An idea for a solution to this would be to implement it such that the tool does not store the data in memory but writes it to files and then accesses those files when needed. This could be difficult though, and might make the entire process of clone reasoning considerably slower. Another option would be to change the reasoning phase such that instead of accessing clone data located in memory, it would read the data from the detection iteratively, just keeping the current and previous two revisions in memory. The previous two, because when we track a clone over the revisions of a project, we look whether it is in one the previous two revisions (in case it was removed accidentally in a revision and placed back in the next). This would keep the functionality of the tool intact while making the memory footprint smaller so that larger numbers of revisions could be researched.

The next idea for future work expanding on this thesis would be to research more software projects. Right now we have researched 4 projects, two of which are relatively small in scale (Charts4j and Subclipse), one could be considered medium in scale (Google Guice) and the other large (ArgoUML). It would be worthwhile to research more projects of medium and larger scales, both to compare the results to the ones we have gathered and to solidify or disprove our conclusions. Particularly for the third research question we asked, the answer to which was relatively inconclusive, additional research and more results would be worthwhile to either be able to answer this question more conclusively. In addition, were the above mentioned point included as well, perhaps larger numbers of revisions could be taken in to account and a larger part of the lifespan of a project could provide more detailed results.

Yet another area where future work can be done is the exploration of effects that other properties have on clone evolution. In this report we focus solely on the effects of size on a clone's evolution. It is conceivable that the length, that is to say the number of cloned lines of code, can have an influence on a clone's evolution as well. For example, questions such as whether longer clones are more or less likely to be removed or have fragments removed can give new insights into the evolution of code clones. Another property that could have an influence on a clone's evolution is the clone distance. This term is explained in chapter 2, but essentially refers to the number of lines between code fragments. Again, researching whether this property has an influence on a clone's chance of being deleted or having fragments removed might give additional insights into the evolution of clones.

Overall, there is still work to be done in the area of code clones and their evolution and the research we have presented can serve as a stepping stone.

Bibliography

[1] L. Aversano, L. Cerulo, and M. Di Penta. How Clones are Maintained: An Empirical Study. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 81–90. IEEE Computer Society, 2007.

[2] B.S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, pages 49–49, 1993.

[3] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *metrics*, page 292. Published by the IEEE Computer Society, 1999.

[4] T. Ball, J.M. Kim, A.A. Porter, and H.P. Siy. If your version control system could talk. In *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*. Citeseer, 1997.

[5] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings. International Conference on*, pages 368–377. IEEE, 1998.

[6] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A.E. Hassan. An Empirical Study on Inconsistent Changes to Code Clones at Release Level. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 85–94. IEEE, 2009.

[7] E. Duala-Ekoko and M.P. Robillard. Tracking code clones in evolving software. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 158–167. IEEE, 2007.

[8] E. Duala-Ekoko and M.P. Robillard. Clonetracker: tool support for code clone management. In *Proceedings of the 30th international conference on Software engineering*, pages 843–846. ACM, 2008.

[9] E. Duala-Ekoko and M.P. Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(1):1–31, 2010.

[10] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[11] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Software Maintenance, 1998. Proceedings. International Conference on*, pages 190–198. IEEE, 1998.

[12] H. Gall, M. Jazayeri, R. Kl
"osch, and G. Trausmuth. Software evolution observations based on product release history. In *icsm*, page 160. Published by the IEEE Computer Society, 1997.

[13] M.W. Godfrey, A.E. Hassan, J. Herbsleb, G.C. Murphy, M. Robillard, P. Devanbu, A. Mockus, D.E. Perry, and D. Notkin. Future of mining software archives: A roundtable. *IEEE Software*, pages 67–70, 2009.

[14] J.H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*, pages 171–183. IBM Press, 1993.

[15] E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective-A workbench for clone detection research. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 603–606. IEEE, 2009.

[16] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 485–495. IEEE, 2009.

[17] H. Kagdi, M.L. Collard, and J.I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007.

[18] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, pages 654–670, 2002.

[19] C. Kapser and M.W. Godfrey. Toward a taxonomy of clones in source code: A case study. In *ELISA workshop*, page 67. Citeseer, 2003.

[20] C. Kapser and M.W. Godfrey. Aiding Comprehension of Cloning Through Categorization. In *Proceedings of the Principles of Software Evolution, 7th International Workshop*, pages 85–94. IEEE Computer Society, 2004.

[21] C. Kapser and M.W. Godfrey. "Cloning Considered Harmful" Considered Harmful. In *13th Working Conference on Reverse Engineering, 2006. WCRE'06*, pages 19–28, 2006.

[22] C.J. Kapser and M.W. Godfrey. Supporting the analysis of clones in software systems: Research Articles. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):61–82, 2006.

[23] J. Kerievsky. *Refactoring to patterns*. Pearson Education, 2005.

[24] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196. ACM, 2005.

[25] R. Koschke, E. Merlo, and A. Walenstein. Survey of Research on Software Clones. *Duplication, Redundancy, and Similarity in Software*, 2006.

[26] H.H. Mui. Studying Late Propagations Using Software Repository Mining. 2010.

[27] J.R. Pate, R. Tairas, and N.A. Kraft. Clone evolution: a systematic review. Technical Report SERG-2010-01, Department of Computer Science, The University of Alabama, December 2010.

[28] R. Robbes. Mining a Change-Based Software Repository. In *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on*, pages 15–15. IEEE, 2007.

[29] C.K. Roy and J.R. Cordy. A survey on software clone detection research. *Queens School of Computing TR*, 541:115, 2007.

[30] C.K. Roy and J.R. Cordy. An empirical study of function clones in open source software. In *2008 15th Working Conference on Reverse Engineering*, pages 81–90. IEEE, 2008.

[31] C.K. Roy, J.R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[32] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, 2010.