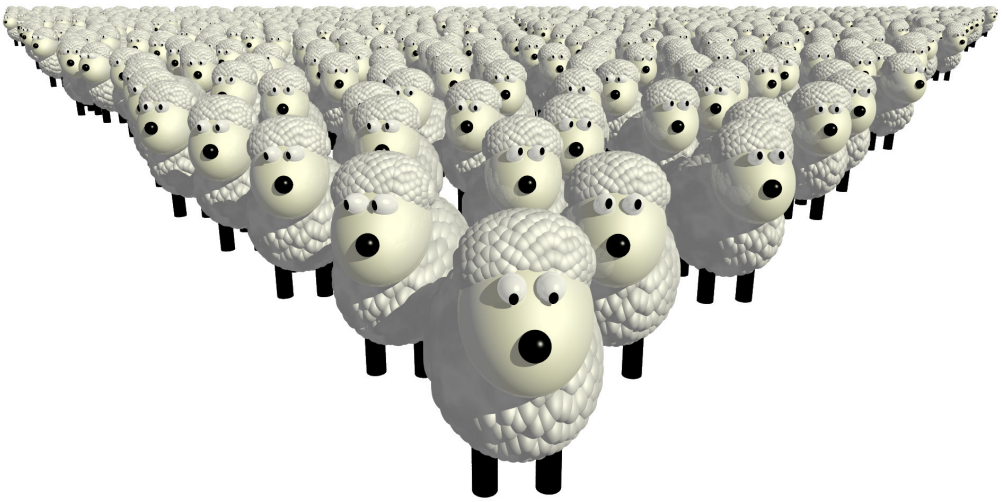
Managing Clones Using Dynamic Change Tracking and Resolution

Helping Developers to Cope with Changing Clone Fragments



Michiel de Wit

Managing Clones Using Dynamic Change Tracking and Resolution

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Michiel de Wit
born in Rotterdam, the Netherlands

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

Document was formatted using MiKTeX 2.6.
Transition diagrams created with GraphViz Dot 2.21.
UML diagrams created with UMLet 9.03.
Charts created with Microsoft Excel 2007.

Managing Clones Using Dynamic Change Tracking and Resolution

Author: Michiel de Wit
Student id: 1015877
Email: mail@MichieldeWit.nl

Abstract

By many, code cloning is nowadays recognized as a threat to the maintainability of source code. Many clone detection strategies have been proposed and a considerable number of removal strategies, mostly based on refactoring techniques, has been shown. However, recent research has showed that clones can often not be removed easily and other strategies, based on clone management need to be developed. In this thesis, a clone management strategy based on dynamic inferring of clone relations based on monitored clipboard activity is described. A tool is introduced that is able to track live changes to clones and offers several resolution strategies for inconsistently modified clones. The adequacy, usability and effectiveness of this Eclipse plug-in have been studied in an experiment, the results of which show that developers actually do see the added value of such a tool but have very strict requirements with respect to its usability.

Thesis Committee:

Chair:	Prof.dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
Committee Member:	Dr. S.O. Dulman, Faculty EEMCS, TU Delft

To my father, who involuntarily taught me the insignificance of achievements,
but who, I trust, would be very proud of what I achieved.

timber ['timbə] timmerhout *o*, (ruw) hout *o*; bomen; bos *o*; stam; balk; spant *o*; *fig* materiaal; **-ed** houten; met houten begroeid; **line** boomgrens; **-merchant** houtloper; **-yard** houtopslagplaats [31]

Preface

This thesis is the product of many, but only one of them is credited for it. That does not seem fair. In the past eleven months I have worked very hard to finally finish what I started so long ago. And this thesis would certainly never have got finished without the great care, support and insights of my dream-come-true girlfriend Judica. She has persistently supported me and motivated me to go on. And quite frankly, that was often very necessary.

Ever since I received by bachelor degree I experienced this strong cosmic resistance, holding me back from my master degree. I say 'cosmic resistance', but I am quite sure there were no force fields, clone armies or other intangible powers withholding me. It was more like a combination of bad cards and a poor poker face.

Nevertheless, after three vain attempts this time I actually made it to the end. Chances are that I drop dead just before I receive my degree, the certificate might get lost or my hard disk gets erased before I can get this thesis printed. But I did finish it and honestly, I am quite proud of it.

To get back to all these people that cooperated to this thesis. First of all, there is Andy, my daily supervisor. He has been very open minded and motivating and always gave me the impression that he had faith in me and what I was doing. Really great! Than there are all other people who voluteered for my experiment. Their input was invaluable for this thesis. I am very pleased they wanted to trade some of their invaluable time for a precious chalkboard coffee mug. Not to forget, my professor, Arie van Deursen. He has consistently supported me in the many attempts I made to graduate. And finally I need to thank my family, particularly my mother, for all support and trust they gave me.

Michiel de Wit
Delft, the Netherlands
February 4, 2009

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In the early days, humans used to scribble things down in clay. The invention of ink and paper made wordy writing a lot less cumbersome. It actually became feasible to spread knowledge by duplicating larger pieces of text, whole books eventually. Since it was first written, the Bible has been copied by hand thousands of times. In this copying, errors were made [37], sometimes leading to substantial semantical differences and inconsistencies.

One characteristic of human beings that clearly sets them apart from other lifeforms, is their ability to use advanced tools. Although other animals like chimpanzees and crows are known to use instruments too [9], *homo sapiens* surely is the only animate being that uses computers to ease its life.

Using computers, copying is a lot easier. Computers are used to copy books, to copy pictures and they are used to copy software source code as well. Copying source code is a practice that is quite common among software developers nowadays [38]. Developers copy for all sorts of reasons [36], but fact is that they copy quite a lot [5, 52]. And in this copying there is, just like in copying big books, the risk of unintentionally introducing errors and inconsistencies. As source code tends to change more often than for instance the Bible, there is the additional problem of keeping all copied fragments in sync.

Duplicate pieces of source code are often referred to as 'clones'. This master thesis is about managing such clones. As the volume of clones in a given source base grows, so will the complexity of coping with all these duplicates increase. In the following sections, the context for this master thesis is sketched and the research questions are outlined.

1.1 About code cloning

When ordinary people are asked about 'cloning', chances are that a sheep named Dolly[1] will be mentioned [65]. Some other people may relate 'clones' to contemporary SF-movies[2]

[1] Actually, her real name was 6LL3.

[2] Those of George Lucas in particular

whereas a small remainder will probably think of botanical techniques to multiply plants asexually.

Obviously, code cloning has something to do with duplication. When programmers are developing new software or revising existing code, they will likely use tools that offer some sort of clipboard functionality. A developer can put snippets of code on this clipboard, often by means of the infamous Ctrl-C key combination, and reuse these fragments later on as often as they see fit [42]. Although most developers will probably not refer to this as code cloning, this is exactly what they are doing. However, using a clipboard is only one way in which code clones can be created.

1.1.1 Defining code clones

Definitions for 'clone' are as many as there are researchers in the field, some very vague, others overly specific. One of the most useful definitions is that by Basit and Jarzabek, stating that "code clones (...) are code fragments of considerable length and significant similarity." [7] This definition identifies clones as non-trivial fragments ("of considerable length") that share sufficiently many traits to consider them similar and related.

It is easiest to see code cloning as a form of duplication, but it can be more convenient to consider it a form of redundancy [32], as not all cloning is the result of deliberate duplication. Some clones are the result of what are sometimes called 'mental macros' [8]. As Baxter *et al.* put it:

> "(...) many repeated computations (...) are simple to the point of being definitional. As a consequence, even when copying is not used, a programmer may use a mental macro to write essentially the same code each time a definitional operation needs to be carried out." — [8]

Mental macros, or 'idioms' [36] are small, relatively simple pieces of logic that are used often, but have for some reason never been abstracted into a higher language construct. Sometimes, it is mainly the language's lack of expressiveness that causes these idiomatic duplications [38, 59], but the cause may just as well be laziness or a developer's reticence to create a new routine for his mental macro.

An interesting collection of such mental macros is sketched by Van Deursen [21]: in this article, 21 real life examples of ways to express the calculation to determine leap years are shown. Two examples from this article are shown in listing 1.1.

1.1.2 Clone Typology

To better capture the differences between clones that are exact copies of previously written code, possibly created by using a digital clipboard, and clones that are the result of mental macros, various researchers have proposed clone classification schemes. Some of these taxonomies mainly define levels of similarity [10, 20, 52], where others aim to differentiate between different kinds of code fragments that can be cloned [6, 35].

Listing 1.1: Two semantic clones, the result of so called 'mental macros'

```
Leap :=
  (Y mod 4 = 0) xor
  (Y mod 100 = 0) xor
  (Y mod 400 = 0) ;

...

if Y mod 400 = 0 then
  Leap := true
else if Y mod 100 = 0 then
    Leap := false
  else if Y mod 4 = 0 then
      Leap := true
    else Leap := false ;
```

The most commonly used clone classification is that by Bellon and Koschke [10], based on previous work by Davey *et al.* [20]. In this taxonomy, three separate types of clones are distinguished:

- **Type 1.** Exact copy without modifications, with the possible exception of differences in lexically trivial tokens (e.g. spaces and comments).
- **Type 2.** Syntactically identical copy in which only variable, type, or function identifiers were changed.
- **Type 3.** Copies with more modifications; statements were changed, added, or removed.

As can be seen in the list, the type number is a direct indication for the amount of differences allowed between clone instances. Clones of the first type are basically just identical. Because most programming language grammars include some meaningless token types (such as spaces, line breaks, and comments), fragments that differ only in trivial lexemes are still considered identical.

When a code fragment is being copied to serve as a template [38], parts of the fragment will be changed after duplicating it. Depending on the level of changes, these clones will either be classified a type 2 or type 3 clone. For type 2 clones, only identifier renamings are acceptable differences, where type 3 clones can have more elaborate changes. In earlier literature, type 2 clones are often referred to as 'parameterized clones' [5], based on the idea that for such clones the names of the identifiers used are irrelevant and could be considered parameters to a clone generating function. An example of two logically identical functions that are type 2 clones is shown in listing 1.2.

A fourth type of clones is often added that is used to denote clones that are not similar in any syntactical way, but do describe the same or similar logic. Such clones are often called 'semantic clones'. Koschke *et al.* define semantic clones as a pair (A,B) such that

Listing 1.2: Two consistently renamed type II clones

```
1  public double square(double x) {
2    double square = x * x;
3    return square;
4  }
5
6  public int pow2(int value)
7  {
8    // Multiply value by itself
9    int result = value * value;
10
11   // Return resulting square
12   return result;
13 }
```

"*B* subsumes the functionality of *A*, in other words, they have 'similar' pre and post conditions." [42] The two semantic clones shown in listing 1.1 clearly both implement the same logic, thus adhering to the same pre and post conditions, but their syntactic differences make it hard to identify these clones.

1.1.3 Good or Bad?

Knowing about the existence of code clones, one may wonder whether having clones in source code is a good or a bad thing. Although no clone is the same, it is generally true that clones have a negative impact on the maintainability of a software corpus [36, 41, 55]. A recent experiment of Lozano and Wermelinger [50] shows that cloning often increases maintenance effort.

To give an impression of the sorts of problems that cloning may lead to, the following list shows some of the most common drawbacks:

- **Code growth.** Being a form a redundancy [32], duplicated code will invariably result in a larger code base, causing longer compilation times and higher storage costs than necessary.
- **Dead code.** Often, code is duplicated to prevent regression [19]. This kind of cloning can lead to dead code, needlessly complicating comprehension of a software system.
- **Increased cost of extension.** Code cloning can have a negative effect on the cost of implementing new features. Extensions will have to be applied consistently, carefully updating all instances of a clone.
- **Introducing bugs.** By copying and pasting existing code, bugs can easily be propagated [45, 38]. Furthermore, less competent developers may be altering complicated code of a coworker, without actually understanding the code, thus possibly introducing bugs, too.

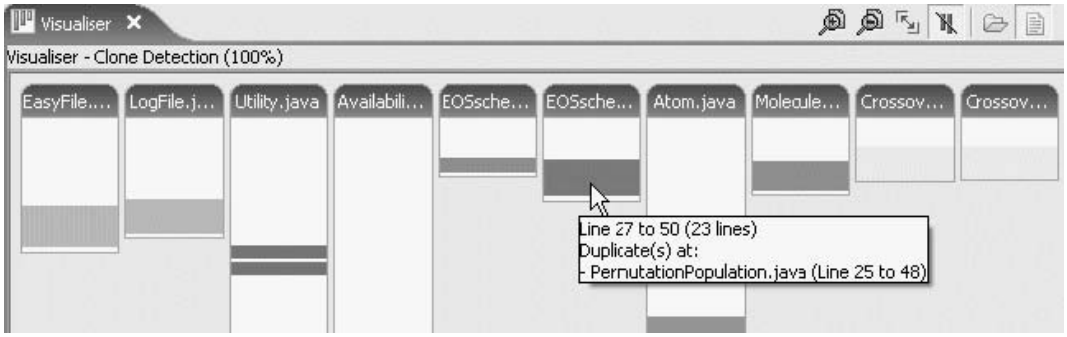


Figure 1.1: A clone visualization technique using stripes and bars to show what parts of code have been cloned.[3].

- **Bug camouflaging.** Code may become so heavily duplicated, that developers might start to consider it an idiom [36] and will no longer doubt its correctness, creating perfect hideouts for persistent bugs and inefficient code.
- **Confusion.** Developers may get confused by similar implementations of the same algorithm. Although the implementations may be equivalent, developers may still believe that there is a reason for the existence of multiple implementations that they may just have overlooked.
- **Motivating bad design.** High degrees of cloning may tempt developers to lower their standards, using bad designs instead of higher quality solutions, fearing to put a flag on a mud barge.

Apart from these drawbacks, cloning may sometimes be helpful, too. In an article with the captivating title "'Cloning Considered Harmful' Considered Harmful", Kapser and Godfrey [36] show that cloning has its benefits, too. Often, developers duplicate existing code as a starting point to develop new code. This technique, dubbed 'forking' by Kapser and Godfrey helps developers write new code more quickly. Forking happens especially often when drivers for a new model of an existing device are being developed. The practice of forking relieves developers of their responsibility to alter existing stable code to add abstractions and as such is a good way to reduce regression risks [19]. Brandt *et al.* reason that copy and pasting code helps to ease the task of writing code, as it allows developers to use rule-based rather than skill-based behavior [12].

1.1.4 Managing Clones

Studies have shown that software corpora tend to contain large amounts of code clones, ranging from 7% to 23% [40]. In one case, a bug was shown to have been propagated to twelve different places by copying and pasting before it was finally found and fixed on all twelve locations [38]. The only reason the developers were able to find all twelve instances

[3]Illustration taken from [58].

of this clone, was because of a rather specific comment included in the fragment. Obviously, a better clone management scheme could have prevented all this misery easily.

A lot of research effort has been put into detecting clones [4, 8, 34, 42], but the subject of managing clones has received relatively little attention. One reason for this discrepancy is probably due to the fact that clone detection is mainly an algorithmic problem, considered a 'true computer science' problem by most, whereas clone management is more about change management and business tactics. In the last few years, however, an increased interest in the management problem is showing, leading to all sorts of new insights.

Techniques have been proposed for tracking clones [22, 45], visualizing clones [58] (cf. figure 1.1) and refactoring code to remove clones [24, 15]. However, most of these techniques focus on either showing cloning or removing cloning. As case study results suggest that about 50% of clones found in source bases can not be refactored [39], more effort will be needed to support developers in containing code clones and helping them to update and maintain cloned fragments.

1.2 Thesis Project

In a recent publication by Zoltán Mann [51], an interesting idea is formulated that might help to gather clone information while code is written and not just in retrospective as currently is common [41]. The concept is quite simple: Mann states that by replacing the copy and paste operations supported by most development environments with a set of well-defined cloning operations, the developer's duplication intentions can be better modeled. The operations proposed by Mann add meaning to copied fragments, stating whether the copied fragments are meant to remain identical, are subject to minor changes or are copied for other reasons.

Using the information gathered by such new copy operations, new possibilities for clone containment arise. If a developer is encouraged to specify the relation between copied fragments, this relation can than later be visualized and enforced by the development environment, possibly leading to more clone-awareness and less bugs due to inconsistently altered clone fragments.

1.2.1 Research Questions

Implementing and evaluating a tool based on Mann's ideas will be the goals for this master project. Mann is quite convinced his approach will help developers to better cope with cloning at a minimal cost of having to specify some additional information during the act of copying and pasting. Based on this conception, the following questions on which to base research work arose:

- **Question #1.** Can the copy and paste replacements described by Mann be realistically implemented in a programmer's development process and coding environment?

- **Question #2.** Are developers willing to alter existing copy and paste habits to help contain code clones?
- **Question #3.** In what ways can the relations established by using Mann's operations be used to enforce consistent editing of clones?
- **Question #4.** Will Mann's operations help reduce cloning related problems?

1.2.2 Project Goals

Starting with the research questions stated above, a plan was conceived to find answers, stating several project goals. Each of the goals aims to help find the answer to one or more research questions and at the same time serves as mile stones that determine the project's schedule. Based on the proposed plan stated in section 5.2.2 of in the literature study report that precedes this thesis [66], the following will be the goals for this thesis project:

- **Implementation of Mann's operations.** The new operators, replacing the old copy and paste commands, will have to be implemented in either an existing development environment or in a prototype environment created for the experiment. Implementation will have to be such, that it would be sufficiently usable in a real production environment.
- **Implementation of relation visualization and enforcement.** Once the clone relations can be established between code fragments using the new operations, appropriate visualizations and enforcement mechanisms will have to be implemented. Visualizations can remain quite simple, for instance by adding marks in the margins of the cloned code fragments. Clone relation enforcement will be more difficult to implement, especially in the case of clone relations in which small mutations are permitted.
- **Experiment.** Once a prototype has been constructed, it should be evaluated by means of an experiment. Actual developers will have to be questioned about their experiences after a reasonable exposure to the prototype and data about the use of each of the operations needs to be logged for later analysis. A longitudinal study showing possible benefits to clone management, however, is not likely to be feasible in the scope of a master thesis project.
- **Evaluation of results.** The results gathered by means of the experiment will have to be evaluated and related to the original research questions. New questions are likely to arise and recommendations for future development of similar tools are to be listed.

1.3 Thesis Structure

The remainder of this thesis will roughly follow the same structure as the project goals stated above. Each of the following chapters will discuss a separate stage of the project and will document both the end results of each phase and the steps that led to these outcomes.

In the next chapter, the process of translating Mann's general ideas into a prototype tool useful for developers is described. Mann's operations are explained and investigated in

more detail. Using basic design techniques, a set of design specifications is drawn up to set the bounds for a prototype tool.

As experience is often a better adviser than dry theory, prototypes were built to explore the implementation options for some of the pivotal features of the prototype to be build. Chapter 3 reports on the most important findings.

In chapter 4, the implementation of an Eclipse plug-in that realizes Mann's operations will be described. This chapter will both describe how the basic operations were realized and how additional visualization and enforcement were added. Using common schema techniques, blue prints of the most interesting parts of the prototype will be given.

Chapter 5 details the experiment that was conducted to evaluate Mann's operations in a controlled setting. Both the experiment's design, including the measurement tools used, and the actual execution of the experiment are documented. A detailed report of the data produced as part of the experiment is documented as part of the appendices.

Before drawing any conclusions, chapter 6 relates the work done as part of this project to recent work by other researchers. By doing so, the relevance of this project's outcomes can be better judged and related to the rest of the relevant scientific corpus.

In the final chapter, chapter 7, the outcomes of this research project are evaluated and conclusions are drawn. The research questions stated earlier in this introduction are reconsidered and answered by referring to the outcomes of the conducted experiment.

A number of appendices is added to this thesis. In the extra chapters, the questionnaires and programming assignments used in the experiment are printed. Furthermore, the data gathered with the experiment is included in tabular form.

Chapter 2

Conceptual Design

In the previous chapter, the operators proposed by Mann [51] to replace existing copy and paste commands in code editing environments were mentioned briefly. These operations were proposed to help prevent inconsistencies that are typically caused by sloppy code duplication. In this chapter the process of designing a prototype tool based on Mann's proposed operators is discussed. Prior to a description of the prototype, the Mann operations are detailed, such as to give a deeper understanding of what the prototype is to realize. In the next chapter, the actual implementation process is detailed.

2.1 About Mann's Operators

The first question that needs to be answered about the operators proposed by Mann is: "Why are they needed?" Mann gives a clear answer to this question in his article:

> "Many software developers know the feeling of (...) debugging a program only to discover (...) that the error stemmed from copy-pasted code segments that had become inconsistent in subsequent editing. (...) Given the extensive use of copy-paste operations and their tendency to cause inconsistencies, there is clearly a pressing need to rethink current editor programs." — [51]

Surveys by LaToza *et al.* show that 59% of developers feel that finding all instances of duplicated code is a serious problem [48]. To aid developers in coping with clones and fighting bugs introduced by inconsistent clone editing, several solutions come to mind. One solution often suggested by computer scientists is to remove the code redundancy and thus prevent inconsistent editing altogether. Actually, this solution resembles the solution that most database engineers propose when redundancy issues occur: normalization. By normalizing such that redundancies are eliminated, the need to keep duplicate entities synchronous is removed. However, just like in database design, normalization is not always possible or desirable and techniques to support redundant data have to be introduced.

Mann pleads to do just the same with duplicated code: to implement techniques that help keep redundant code consistent. As it is generally quite hard to differentiate between code

duplication that is due to semantic redundancy and duplication that is inherent to the language's grammar, it is necessary to actively include the developer in the process of guarding code consistency. Mann proposed to do this as follows:

> "One solution is to replace *cut*, *copy*, and *paste* with operations that correspond directly to the intended semantics behind their use. With these operations, the user can specify semantic relationships among copied objects, and the editor program can use that information to help in the long-term support of those relationships. It would thus avoid the inconsistencies that currently arise from the use of cut, copy, and paste." — [51]

2.1.1 Copy and Paste Scenarios

In his article, Mann introduces five typical scenarios involving the clipboard. Some of these scenarios have the potential to cause later inconsistencies. Furthermore, these five scenarios help to get a better impression of the reasons developers have for using the clipboard during their programming work.

- **Cut to delete.** First of all, developers may be using the clipboard functionality of their development environment just to delete a specific code fragment. This may just be a bad habit or a way of assuring the removed code can be easily restored, in case this may be necessary. Anyhow, this type of clipboard use is harmless with respect to causing possible inconsistencies.
- **Cut and paste to move.** The intended use of the cut and paste command sequence is to move code. The clipboard is used as a generic storing place for code while it is being moved. Although in its base form this operation is harmless – after all, no redundancy is created – the moved data will stay on the clipboard and can thus potentially be duplicated later on.
- **Copy and paste to duplicate.** The riskier scenarios involve the copy and paste command sequence. Code is placed on the clipboard to be replicated at one or more locations later on. Ideally, a developer would copy and paste only to duplicate. Identifiers, like variable names, are likely candidates to be copied and pasted in this manner [38]. Essential to this scenario is, that the developer does intend to duplicate only; the copied fragment is not to be modified later on, but just serves as a form of inevitable redundancy.
- **Copy and paste for templating.** When larger code fragments are duplicated, chances are that they are not meant to be copied literally, but rather serve as a template for a similar piece of code that needs to be written. Templating is a common reason for code duplication [36] and has the potential to introduce rather persistent inconsistencies.
- **Copy and paste without a logical connection.** Other reasons may exist for duplicating code. As programming languages typically contain a lot of structure elements, chances are that a developer will be copying and pasting those, too. The logical connec-

Scenario	Risk of inconsistencies
Cut to delete	Low
Cut and paste to move	Low
Copy and paste to duplicate	Medium to high
Copy and paste to create a template	High
Copy and paste without logical connection	Medium

Table 2.1: Five common copy and paste scenarios and their intrinsic risks

tion between the source and target of a copy and paste operation on such elements may be weak or even be non-existing. Risks for code inconsistencies are much lower here.

2.1.2 Replacement Operators

The aforementioned five copy and paste scenarios are summarized in table 2.1, indicating their associated risk of introducing inconsistencies. To counter these risks, Mann introduces four operations that are to replace the standard cut, copy and paste operations.

- **Move.** With the move operator, a code fragment can be moved. It is just like the cut and paste command sequence, without the code fragment remaining on the clipboard.
- **Copy-identical.** A code fragment is duplicated and kept synchronous with the original fragment automatically.
- **Copy-and-change.** Duplication, but with the constraint that the user must change the fragment after duplication.
- **Copy-once.** This operator copies a fragment, without adding further constraints. As such, this operator emulates the original copy and paste commands.

Mann suggests that the relations established by these operations should be maintained by the development environment automatically. In practice, this means that the copy-identical and the copy-and-change operations need extra attention, whereas the other two operations basically emulate conventional clipboard use.

2.2 Prototype Concept

The copy and paste replacement operators by Mann could be implemented straight forward in a development environment. However, chances are that most developers will not use them: the copy and paste shortcuts are used so thoughtless by developers, that it will be hard to alter these ‘instincts’. Therefore, a slightly modified version of Mann’s ideas is proposed: keep the original copy and paste operations, but infer the intended operation as soon as the cloned code fragment is modified. When the clone is never modified, the *clone-identical* operation is implicitly assumed. In this way, the strain on developers is reduced, while Mann’s clone relations can still be captured and enforced.

2.2.1 Inferring the Operations

Inferring the Mann clone relations is not quite trivial, but a simple and effective strategy was conceived. This strategy is based on always inferring the most restrictive operator and loosen the constraints as required. Furthermore, the *move* operator can be easily inferred, as this is indicated by the use of the *cut* operation instead of *copy*. This leaves three operations to infer. Sorted on restrictiveness, this yields the following determination list:

- **Copy-identical.** Initially, a fragment is not changed after duplication, so the *copy-identical* constraint holds.
- **Copy-and-change.** As soon as the fragment is changed, the constraints can be weakened to *copy-and-change*.
- **Copy-once.** If more elaborate changes are performed, all constraints can be removed completely to emulate the *copy-once* operator.

2.2.2 Clone Change Resolution

Using this simple scheme to infer the Mann operations significantly reduces the effort required of the developer, thus enhancing the chance of successful adaptation. To make weakening constraints on cloned code fragments not too easy, a mild barrier was deemed necessary. Without such a noticeable 'bump', developers would presumably not have a sense of being restrained in their cloning behavior at all, thereby effectively eliminating the possible benefits of Mann's operators. Although this barrier should obviously not be too obtrusive, research indicates that cloned code is actually edited less frequent than other code [43], so that developers will not be likely to get hindered in their daily work too much.

As a solution to this issue, the concept of 'clone change resolution' was introduced. That is, as soon as a clone – created by copy and pasting – is modified, this change has to be resolved to maintain a consistent clone model. The term 'resolution' is used deliberately, as this suggests the existence of some sort of problem that needs to be resolved.

The clone model, the set of all clone relations in a source base, is defined to be consistent if all related clones are either equal or contain well defined modifications. To keep the initial prototype simple, only one type of modification is considered well defined: simple token replacement. Such replacements, which involve only substitution of single tokens, result in parameterized clones [4].

More complex modifications might be considered later on, but were deemed too complex to fit within the scope of this project. Among such more complex differences would be substitutions of one or more tokens by a different number of other tokens and variable bodies, that is clones of which only a certain prefix and postfix are equal, but that have different contents (cf. listing 2.1, where the bodies on lines 4–5 and 15–17 are variable). Such modifications are harder to describe and would require advanced clone descriptors, such as those described by Duala-Ekoko and Robillard [22]. Furthermore, allowing for more complex differences weakens the cloning relation and may arguably lead to rather arbitrary clones with insignificant similarity.

Listing 2.1: Two clones with variable bodies

```
1  public double distance(Point p) {
2    assert p != null;
3
4    double distance = Math.sqrt(Math.pow(this.x - x, 2) +
5                    Math.pow(this.y - y, 2));
6
7    return distance;
8  }
9
10 ...
11
12 public double distance(Point p) {
13   assert p != null;
14
15   double dx = thix.x - x;
16   double dy = this.y - y;
17   double distance = Math.sqrt(dx * dx + dy * dy);
18
19   return distance;
20 }
```

2.2.3 Resolutions

Based on the Mann operators and the determination list for inferring the operators described in section 2.2.1 a basic set of clone change resolutions was compiled. These resolutions are meant to deal with a clone change, resolving the inconsistency that was created by only modifying one clone.

2.2.3.1 Unmark Clones

The most basic strategy to cope with inconsistently modified clones is to just forget about the clone relation[1]. This rather trivial resolution actually emulates Mann's *copy-once* operation.

To deal with the case where a developer would want to keep part of the clone under consistency control, and only remove the clone mark from part of the fragment, two additional resolutions were conceived. These resolutions respectively unmark only the modified head or the tail of a clone fragment. The corresponding subfragments should be removed from the other clones in the clone relation, too, making these resolutions slightly more complicated.

As an illustration of the *unmark head* and *unmark tail* resolutions, consider the following example. In the code fragment below, the underlined part is assumed to have been modified.

```
double distance = Math.sqrt(dx * dx, dy * dy);
```

[1] cf. glossary in appendix A.

When this clone change would be resolved by unmarking the tail, all text starting with the modified text (underlined in the example) would be unmarked, resulting in a clone that consists only of the first few tokens. The fragment below shows this (the fragment that is still marked as a clone is boxed):

```
double distance = Math.sqrt(dx * dx, dy * dy);
```

The inverse holds for unmarking the head: only the tokens after the last modification are kept as the clone, all other tokens are unmarked:

```
double distance = Math.sqrt(dx * dx, dy * dy);
```

As is shown in the example, unmarking the head or tail of a clone fragment may result in a remaining clone fragment that does not corresponds with a syntactical unit anymore. One might consider trimming the remaining clone to the largest syntactical unit still contained in it. Koschke *et al.* show an easy way to "cut out syntactical clones" [42] by cleverly augmenting the token stream of a fragment.

2.2.3.2 Apply Changes to Clone Set

A second way to resolve inconsistent clones is by updating all clones in the clone set[2] to reflect the latest modifications, basically resynchronizing the clones. This update resolution elegantly implements Mann's *clone-identical* operator. However, though updating sounds deceptively simple, this resolution is actually quite intricate. Clones may differ in spacing (and possibly in comments too), and these differences will have to be respected when applying changes that have been made to another instance in the clone set.

Let us assume the following clone set, consisting of two clones, where the underlined fragment again indicates what has been modified:

```
double x = Math.sin(angle) * distance; //Calculate x
...
double x= Math.sin(angle)*radius;      //Get X
```

To properly update the first clone to reflect the changes made to the second clone, the differences in spacing will have to be taken into account, so that the clone pair will look as follows after they have been resynchronized:

```
double x = Math.sin(angle) * radius; //Calculate x
...
double x= Math.sin(angle)*radius; //Get X
```

Notice how spacing has been preserved and the original clone fragment has not been blindly overwritten by the old one. Things get even more complicated when synchronizing parameterized clone fragments (cf. section 2.2.3.4).

[2] cf. glossary in appendix A.

2.2.3.3 Postpone and Ignore

Two other, less favorable ways to cope with clone changes are to postpone a resolution and ignore the changes altogether. Whereas the first solution is basically a way of allowing the developer to resolve the changes himself first, to ignore the changes means the developer has to take control over the clones himself. By ignoring clone changes, a resolution is basically postponed *ad infinitum*. Postponing and ignoring both emulate different aspects of Mann's *copy-and-change* operation.

2.2.3.4 Parameterize Clones

The final resolution in the list compiled for this project is a combination of unmarking and ignoring changes: parameterization [4]. The changed parts of the clone are declared *parameters* of the clone: tokens that can be freely varied without breaking the clone relation. Basically, these parts are being unmarked, without being removed from the clone.

By declaring certain parts of a clone as parameters, the clone can later be updated to changes applied to another instance, without overwriting the parameterized parts. This form of change resolution is more advanced than the operations Mann suggests, but does help developers to deal with slight variations in clone fragments in a more intuitive way.

2.3 Requirements Analysis

To implement the clone change resolutions described in the previous section and study their usefulness in an experiment, it is best to develop a plug-in to an existing development environment. By doing so, side-effects of an environment unknown to the experimental subjects are prevented. An unfamiliar environment might cause subjects to behave differently or may cause a negative bias caused by the unfamiliarity with the experiment's settings.

The two most suitable development environments to extend are Microsoft's *Visual Studio* and the *Eclipse IDE*. Both environments are broadly accepted and used and it would be fairly easy to find experimental subjects that are used to these particular platforms. Furthermore, these development environments have well-documented extension interfaces, making the development of a plug-in within the scope of a thesis project more feasible.

A quick survey among likely volunteers – mostly postgraduates and Ph.D. students – learned that Eclipse was the preferred development environment. After asserting that there were no major technical advantages to selecting Visual Studio, Eclipse was selected as the platform for the plug-in.

As a first step towards designing the Eclipse plug-in, a name was chosen. The name CLONEBOARD was one of the first options generated and was finally elected because it is both easy to remember and elegantly links the two concepts it relates to: the clone and the clipboard[3].

[3] *The Clone and The Clipboard* seems to be quite suitable as a name for a fairytale, actually.

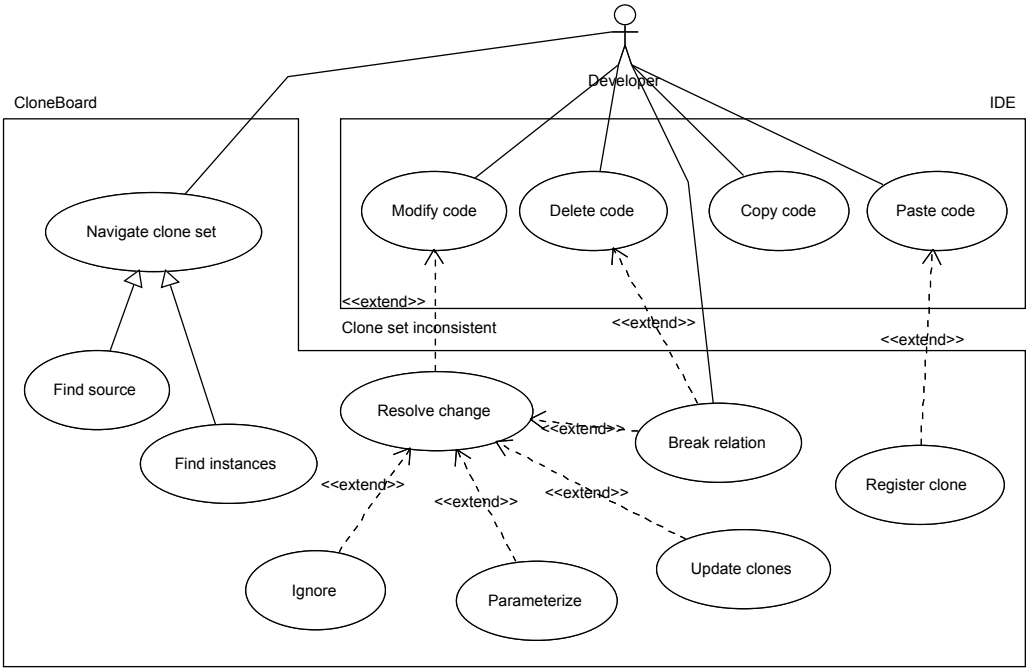


Figure 2.1: Use cases for the CLONEBOARD plug-in

2.3.1 Use Cases

Before being able to specify any further requirements to the plug-in that is to be developed, its uses need to be explored further first. Obviously, the primary use of the plug-in will be to resolve inconsistent clones using the Mann operation-based resolutions discussed earlier, but other functionality is likely to be required too, if the plug-in is to be accepted by developers.

Figure 2.1 shows an UML use case diagram in which the uses that are likely to be required are depicted. As is shown in the diagram, there are three main use cases: *Register clone*, *Navigate clone set* and *Resolve change*. The most basic of these use cases, *Register clone*, describes the creation of a new clone relation by copying and pasting code fragments. To not disturb the developer more than needed, the action of creating a clone should use existing copy and paste actions in the integrated development environment (IDE).

The *Navigate clone set* use case describes the ability to navigate the clones in a clone set. A clone generally always relates to one other clone. However, as a clone may be duplicated several times, a set of related clones will come into existence. To allow developers to trace the origin of clone fragments or find all instances of the clone set, navigation functionality would be paramount.

Finally, the most elaborate use case involves the resolution of a clone change when a developer modifies code that has been cloned. Although research shows that cloned code

tends to be modified less [43], coping with change is the primary purpose of the plug-in, so the *Resolve change* use case is paramount. The case when the code of a clone fragment is totally removed is – as a use case – not considered a code modification, but rather a clone relation break. Only when a clone's fragment is altered, this will be considered modification.

When a clone's code modification results in an inconsistent clone set, this will trigger the *Resolve change* use case, requiring the developer to make a choice out of a selection of appropriate change resolutions. Depending on the developer's choice, one of the resolution use cases will be performed. The specifics of these resolutions have been detailed in section 2.2.3.

2.3.2 Functional Requirements

To come to a system design, the use cases determined previously, need to be translated into functional requirements that describe "the actions that the product must take if it is to be useful to its users." [54] Following the use cases specified in the previous section and shown in figure 2.1, the following paragraphs describe the respective functional requirements that were drawn up for CLONEBOARD.

Use Case: Register Clone

- **Intercept clipboard manipulation.** The plug-in should intercept and register common clipboard actions (i.e. *copy*, *cut* and *paste*) without disturbing the original actions.
- **Relate copied fragments.** The plug-in should apply the clone relation to the code fragments that were duplicated using clipboard actions.
- **Filter trivial clone relations.** Trivial clone relations should be filtered out. Among such relations are cloned keywords, cloned spacing, cloned operators and cloned comments.
- **Provide visual feedback.** Established clone relations should be confirmed to the developer by means of some sort of visual feedback.

Use Case: Navigate Clone Set

- **Link clones in clone set.** The plug-in should allow a developer to find all clones in a clone set, by starting from a clone fragment in a source editor. Or as LaToza *et al.* propose: "[to discover clones] embed hyperlinks between clone instances with editor support for navigating between clone instances." [48]
- **Link to original copy.** The developer should be able to find the source of cloning, that is the original copy from which clones were created.
- **Browse clones by file.** To provide overview, the plug-in should enable clone browsing on a per file basis. The overview should be linked to the source code viewer, so that clones can be highlighted in their original context.

- **Show attributes.** If attributes are attached to clones (e.g. parameters), the plug-in should expose these for the developer to inspect.

Use Case: Break Relation

- **Remove from clone set.** When requested or required to, the plug-in should remove a clone from a clone set.
- **Remove empty clone set.** If breaking a clone relation means the clone set becomes empty (i.e. there is only one copy of the fragment left), the clone set should be removed.

Use Case: Remove Clone

- **React on deletion.** The plug-in should break the clone relation when the full code fragment of a clone is deleted by the user regardless of the deletion operation used.

Use Case: Modify Code

- **Check clone consistency.** The plug-in should react on the modification of a clone's code by redetermining its consistency. Only when the clone is still consistent with the rest of the clone set, it should be marked consistent.
- **Highlight inconsistency.** As soon as an inconsistent clone is detected, the clone should be highlight to communicate the inconsistency to the developer.
- **Trigger resolution.** When an inconsistent clone is detected, the plug-in should trigger clone resolution as soon as the developer stops modifying the clone and is available for possible queries regarding the change resolution.

Use Case: Resolve Change

- **Determine applicable resolutions.** When clone change resolution is required due to a detected inconsistency, the plug-in should determine which resolutions apply.
- **Select resolution.** To determine the appropriate resolution, the plug-in should query the developer. The developer should be able to make an informed decision, so the plug-in should supply the relevant information. The developer should be able to communicate the resolution he wants to be applied.
- **Determine preferred resolution.** To aid the developer in selecting a change resolution, the plug-in should use heuristics to determine the resolution that is most preferable and highlight this resolution.
- **Postpone resolution.** Resolution of a clone change should be postponable to cope with the possibility that the plug-in wrongly assumed the developer was done editing the clone or is available for queries.

- **Apply resolution.** The selected resolution should be applied after asserting that it is indeed applicable. Changes to clones that are part of the resolution should not lead to new resolution queries, but should rather be accepted, however without abstaining to register inconsistencies.
- **Remember selected resolution.** If the developer so wishes and it seems appropriate, the plug-in should remember what resolution was selected for a particular clone.
- **Automatically apply selected resolution.** If a resolution was remembered for a particular clone, it should be automatically applied on the next clone change resolution case, provided that the resolution applies.

2.3.3 Constraints

It is important to get the technical boundaries of a system clear before starting to develop it. System constraints limit the technical possibilities, but at the same time help to focus on possible ways to implement the required functionality. Some of the constraints for CLONEBOARD were sketched already, others still need to be made explicit. The following list shows the most important system constraints imposed on CLONEBOARD, either by external limitations or to reduce implementation complexity.

- **Java language.** Clone consistency checking and change resolution will assume the Java language. Although other languages can be used in the Eclipse IDE too, focusing on a single language simplifies the analysis routines.
- **Eclipse v3.4.** As an open source project, the Eclipse IDE tends to change quite radically from version to version. To reduce implementation complexity, the Eclipse version the plug-in is to operate on is fixed on version 3.4, also known as Ganymede[4].
- **Prototype quality.** The plug-in needs to be developed to such a quality level that it is suitable for use in an experiment. It needs not be commercial quality. As such, there is no need for installation manuals and elaborate help functionality.
- **Single user.** The clone data gathered by the plug-in does not need to be shared with other developers. The plug-in can be assumed single-user only. Adding multi-user functionality would add considerable development time and is not necessary to evaluate the concepts.

2.3.4 Non-functional Requirements

As Robertson and Robertson summarized a common observation, "non-functional properties may be the difference between an accepted, well-liked product and an unused one." [54] To prevent such a disappointment, it is wise to draw up a list of so called non-functional requirements that state which qualities a software system should have in order to be usable. The following is a list of such requirements for the CLONEBOARD system.

[4] See `http://www.eclipse.org/ganymede`

- **Blend in.** The plug-in should blend in with the development environment as much as possible, adapting its basic look and feel and not introducing any more user interface concepts than is strictly necessary.
- **Hide.** CLONEBOARD's main purpose is to help developers cope with the negative consequences of clone inconsistencies. As long as there are no problems, the plug-in should remain invisible as much as possible, not intruding on the developers professional activities when there is no strict need for it.
- **Patience.** No decisions should be forced on developers. When a choice is to be made, there should always be an option to postpone the choice or escape the matter altogether.
- **No performance penalty.** There should be no noticeable performance penalty of using CLONEBOARD. This requirement is slightly diminished by the fact that the tool will at first only be used in a controlled environment, in which data volumes can be expected to be low.
- **Stable.** Bugs in the plug-in should not have consequences for the overall system. Failure of CLONEBOARD should not bring down the system or corrupt valuable assets.
- **Fault tolerance.** As developers are no normal users, they can be expected to do things normal users are not expected to do. CLONEBOARD should – within the bounds of reason – be tolerant to such potentially faulty behavior.
- **Developer friendly.** CLONEBOARD is a tool for developers and should as such speak the language of developers and respect their customs.

2.4 Summary

In this chapter, the conceptual design of a prototype based on Mann's operators was outlined. The process of designing this tool consisted of first translating the Mann operators into concepts more suitable to be implemented in software. Change resolutions were introduced as a means to infer the Mann operations rather them have them explicitly specified by the user. Several change resolution strategies have been outlined and a deduction scheme was given by which Mann's operators could be inferred.

By means of traditional use case analysis, the core functionality for a prototype plug-in was established. Each use-case was translated into a set of functional requirements, thus preparing for the actual implementation. To guard the overall quality and usability of the prototype, a set of non-functional requirements was specified that set the overall goals for the prototype system.

Chapter 3

Technical Research and Prototyping

Developing a plug-in for Eclipse is a non-trivial endeavor [63]. Eclipse is notoriously complex due to its very flexible design. Although this flexibility allows developers great freedom in extending most aspects of the development environment, it at the same time makes for a very steep learning curve. Without relevant previous experience, starting to write a plug-in straight on is without doubt dense at best. It is certainly advisable to start experimenting with the environment first, building prototypes of key features before integrating all concepts into a final product.

In this chapter the process of experimenting to come to a final technical design is outlined. Sharing some of the design rationale may prove valuable to future researchers and developers.

3.1 Main Features

The CLONEBOARD plug-in, as specified in the previous chapter, will consist of several important features, which each pose their own challenges. These features can be broadly divided into a number of distinct concerns, each of which needed to be tested for implementation feasibility before the actual plug-in could be developed. By prototyping these important features, the risk of project failure can be significantly reduced. [11]

- **Capturing clipboard activity.** The foremost requirement for CLONEBOARD's success is to be able to capture the developer's clipboard activity. Only by doing so can clone information be gathered in the proposed way. The only alternative to this approach would be to use clone detection algorithms to find clone relations after the fact [4, 42].
- **Clone model.** A model of the registered clones is required to be able to implement a Model-View-Controller (MVC) plug-in. The MVC pattern is widely used in Eclipse and seems very appropriate for the case at hand, as multiple views on the same model are required.

- **Clone linking.** To navigate clones, i.e. finding other instances of the same clone set, it is necessary to embed some kind of 'hyperlinks' [48] in the source code. Without prior knowledge of the Eclipse API's, this feature seemed particular difficult to implement.
- **Clone browsing.** Browsing the clone model using some sort of viewer seems to be one of the less challenging aspects of the CLONEBOARD plug-in. Lots of viewers have been developed for Eclipse, so information about the process must be plenty.
- **Clone resolution.** While resolving clone changes, the modifications will sometimes have to be forwarded to other instances. Updating code may prove to be more intricate than at first anticipated, particularly because of Eclipse's highly abstracted interfaces. Furthermore, detecting a suitable moment to interrupt a developer to query him about a resolution may not be as straight forward as one might hope.

Implementing a clone model and an accompanying browser proved to be quite simple. Clayberg and Rubel actually give a very good tutorial on implementing a model and accompanying viewer [18]. A small prototype showed that the implementation of these concerns would not be likely to cause much technical difficulties.

Some of the other concerns, however, posed more of a challenge. In the following sections, the research and prototyping activities for each of these concerns are outlined, giving useful insights into the rationale behind some of the implementation choices for CLONEBOARD.

3.2 Capturing Clipboard Activity

The first and primary challenge faced was to try and capture clipboard activity in the Eclipse environment. Quite some good and thick books have been written about Eclipse plug-in design [18, 26, 29], but none of them mentions capturing clipboard activity. Eclipse features a well-designed extension API, that allows developers to extend various parts of the environment. Actually, the API is designed in such a way, that plug-ins themselves can provide new extension points. And to cap it all off, almost all functionality of Eclipse has been implemented as a plug-in.

Eclipse features a very helpful toolkit, called the Plug-in Development Environment (PDE)[1], that amongst other things allows plug-in developers to browse all available extension points. However, browsing the list, double checking every extension point, learned that there are no clipboard extension points in Eclipse. This meant that alternative solutions had to be sought for.[2]

[1] See `http://www.eclipse.org/pde`

[2] In retrospect, one easy solution, only discovered after the plug-in's completion, would have been to use the *ElectroCodeoGram* software by Schlesinger and Jekutsch [56]. This framework captures all sorts of developer activities in Eclipse, including clipboard activity.

3.2.1 Option 1: Replace the JavaEditor class

The first solution that was conceived came down to replacing the Java editor plug-in that ships with Eclipse by one that was extended with the CLONEBOARD functionality. In this modified version, the copy and paste actions would be intercepted.[3] Although this option was considered plausible, it certainly did not feel 'right': Eclipse has a highly extensible framework that should allow for more elegant alternatives.

3.2.2 Option 2: Subclass the JavaEditor class

Eclipse supports registering multiple editors for the same file type. One of these editors will be the default editor, while the other types can be selected manually, should the need arise. Using this mechanism was considered as the second way to capture clipboard activity. By implementing a so called plug-in fragment[4], it was found possible to subclass the Java editor[5] and override the default copy and paste actions in this new class. This extended Java editor could then be registered as an alternative editor for Java files[6]. But as is often stated in the Eclipse documentation, plug-in fragments are mainly meant to be used as a way to make small adjustments to existing plug-ins, not to add new functionality. So this option, too, was found undesirable.

3.2.3 Final option: Dynamically replace clipboard actions

By further researching, a prototype was eventually created that dynamically replaced the copy and paste actions of the Java editor – and actually all text-based editors – by custom actions. Although the implementation still has some drawbacks, it was found the most optimal solution. The only other plausible possibility would have been to add a low-level keyboard event filter to be alerted of copy/paste shortcuts, and this option surely did not feel 'right' at all.

The clipboard actions were replaced at system start up by registering for the startup extension point,[7] that signals the registered plug-in as soon as the Eclipse environment has been started.[8] At start up, all opened editors are queried – and a listener is registered to be informed of new editor windows – and the copy and paste actions of each editor were replaced by using the `ITextEditor.setAction` method. The new actions were looped back to the original implementations, to keep any special features particular editors may add to the standard copy and paste implementation Eclipse provides.[9]

[3] Altering `org.eclipse.jdt.internal.ui.javaeditor.createActions()` should suffice.

[4] See `http://wiki.eclipse.org/FAQ_What_is_a_plug-in_fragment%3F`

[5] Its implementation is marked internal, so that it can not be subclassed in a normal plug-in

[6] It proved impossible to set any other editor than the one that ships with Eclipse as the default Java editor.

[7] I.e. `org.eclipse.ui.startup`

[8] Actually, users can disable the start up of plug-ins manually, thus possibly disabling CLONEBOARD

[9] The Java editor, for instance, adds functionality that embeds information about imports that are required by the copied fragments, so that these imports can be automatically added when the fragment is pasted in an other source file.

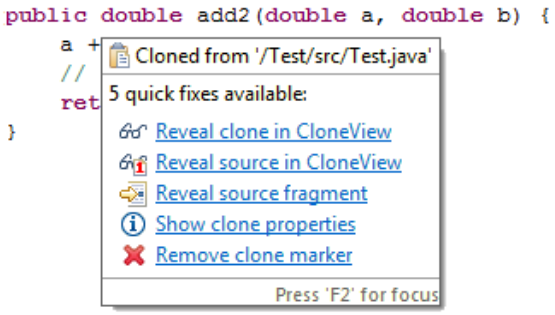


Figure 3.1: A marker resolution popup used to provide clone navigation options.

3.3 Hyperlinking Clones

To help developers find all instances of a clone, hyperlinking clone instances together would be a good solution [48]. Implementing hyperlinks in the Eclipse code editors is not necessarily a trivial operation, so some testing was done to find the best technical solution.

After inspecting all possible extension points and studying the relevant Eclipse source code sections, it was concluded rather quickly that true hyperlinking (i.e. using links embedded in the source code) is not possible in Eclipse. However, alternatives were found.

3.3.1 Marker Resolution

Eclipse supports the notions of code markers and code annotations. A marker can be put on an annotated piece of source code. These markers can be shown in the margin of most code editors Eclipse supports. Using a dedicated extension point,[10] a list of options can be shown when the user hovers his mouse over the annotated piece of code or its marker in the margin.

Experimenting with Eclipse's marker resolution mechanism, as this extension is called, proved it to be a good solution to the hyperlink problem. Developers can bring up the marker resolution popup by hovering an annotated piece of code or by using a keyboard shortcut. The window is non-obtrusive and can be easily dismissed. Furthermore, multiple hyperlinks are supported, so that several different navigation options, as well as other clone related operations, can be supported.

Figure 3.3.1 shows an example of a marker resolution popup in which several navigation options are shown, along with other clone operations. The only downside of the marker resolution mechanism is the fact that the popup window shows the caption '*x* quick fixes available', which does not quite describes the purpose of the navigational hyperlinks. How-

[10] The extension point is identified as `org.eclipse.ui.ide.markerResolution` and is primarily intended to offer the developer solutions to the 'problem' a marker indicates.

ever, this minor disadvantage will probably have to be accepted, as no other comparable solutions are available.

3.3.2 Drawback of Technique

Some further experimentation showed a major problem with the marker mechanism: for reasons not quite clear, the popup window would only show the first time the mouse is hovered over the marker code fragment. On all later occasions, the hyperlinks were not shown anymore. After arduous hours of deep investigation, the cause was shown to be in the Java editor's implementation of the popup window. Due to what must be considered a bug, the popup window would show up the first time, regardless of its contents, but would later on only show up if true 'quick fixes' were found.

A workaround for the issue was found, by applying a special 'is quick-fixable' attribute to the marker. Unfortunately, the code required to apply this attribute did not turn out to be very elegant. As such, one may consider this a rather dirty fix. But then again, most workarounds tend to get disqualified for beauty contests.

3.4 Capturing Clone Changes

Clones are usually created by using the standard copy and paste operations. These operations make it easy to detect cloning in real-time. Detecting clone changes in real-time, however, is a bit harder. The main problem in detecting changes is in grouping atomic clone changes (e.g. character insertions or deletions) into transactions. Clone change resolution only makes sense when considering change transactions and not atomic changes, as nearly all semantically relevant syntactical elements developers may alter span multiple characters and are made during a certain time interval.

3.4.1 Grouping Concerns

Correctly grouping changes into transactions is a very important factor in the final usability of the plug-in, as change resolution will occur as soon as the clone change transaction has ended. Concluding a transaction's end too early will likely lead to frustrated developers as they will be queried for a resolution when they were actually still in the middle of applying changes.

To group atomic changes, several of their properties may be considered:

- **Location.** The clone (or clones[11]) the change applies to. A change to one clone does not influence other clones' transactions.
- **Time.** The moment in time the change was applied. Changes that are distant in time are not likely to belong to the same transaction.

[11] A code fragment may be part of several partially overlapping clone fragments. A developer may for instance first copy an entire method and later copy an identifier used in this method, thus creating two clone sets.

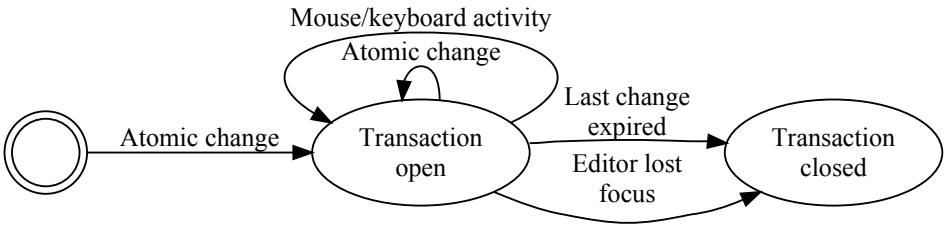


Figure 3.2: State transition diagram of clone change grouping scheme.

- **Type.** A change can be an insertion, deletion or replacement. Different types of changes may belong to sequential transactions, rather than to one single transaction.
- **Size.** Changes may span just one character, or multiple.[12] Small changes are less likely to form a transaction on their own than large modifications.

Furthermore, there are some external events that may occur in the development environment between two atomic changes that may provide help in grouping changes:

- **Focus.** A developer may change the IDE's focus to another interface component (i.e. an other editor window). Changing to another editor may be an indicator of the end of a transaction.
- **Saving.** Source files are changed periodically. When an editor saves a file manually[13], it is fair to assume he has finished editing the clones he previously modified.
- **Keyboard and mouse activity.** A developer may use his keyboard or mouse in between two atomic changes for other purposes than editing, for instance to navigate to another part within the same clone fragment or to operate code assistance features offered by the IDE.[14]

3.4.2 Final Grouping Scheme

It took quite some experimentation to come up with a grouping scheme that was able to compose meaningful change transactions. The final scheme was actually only implemented after the experiment pilot was performed (cf. section 5.6).

The basic concept of the scheme that was designed is that of expiration. For each clone that is changed, a time of last change is kept. As soon a clone's last change is a certain amount of time in the past (i.e. it has expired), all changes to that clone that have been

[12] Multiple characters may be deleted or replaced at once by means of the fragment selection functionality found in most source code editors. Multiple characters may be inserted at once by pasting a fragment stored on the clipboard.

[13] Most IDE's support automatic periodical saving to prevent data loss in case of a catastrophic failure.

[14] In the case of Eclipse, such features include code completion and in-line documentation browsing using popup windows

recorded since the last transaction on that clone are grouped into a new transaction. This design entails that there may be several open change transactions[15] at any time, possibly expiring at the same time, too.

To prevent early transaction ending, the scheme was extended by counting several external events as changes, too. All keyboard and mouse activity that could not be directly related to a clone change, was considered a change on all open change transactions. During the experiment pilot it was shown that Eclipse features such as code completion and in-line documentation were being used in the middle of a transaction quite regularly, but as they were initially not being counted as changes, clone change transactions would expire in the middle of using such a feature, often leading to irritation or misunderstanding.

Finally, changing the IDE's focus to another editor was added to the scheme as a trigger to instantly expire all clone changes in the editor that just lost focus. This feature was intended to prevent confusion and disorientation. Queries about the resolution of changes to a clone in an editor that is not visible or not in focus, may lead to wrong decisions, as the developer might not evaluate them in the right context.

The final scheme is depicted in figure 3.2. As the diagram shows, a transaction will remain open as long as atomic changes to its associated clone occur or there is some general keyboard or mouse activity. When the last change to the transaction expires, or the editor loses focus, a transaction will be closed and is ready to be resolved.

3.5 Summary

Entire books are written on the subject of developing Eclipse plug-ins. Clearly, creating plug-ins is not a trivial task. To gain experience in writing plug-ins for Eclipse and to identify potential problems early on, a number of prototypes was created for the plug-in's pivotal features.

Techniques were explored to capture clipboard activity in Eclipse. Two prototypes were built before the final solution to the seemingly obtuse task of hooking into Eclipse's clipboard handlers was found. Even though this solution was still not as elegant as was hoped for, it proved to be the optimal solution within the bounds set by the Eclipse framework.

To implement clone hyperlinking, quite a lot of research had to be conducted as well. Finally, a solution based on Eclipse's marker model was conceived. This solution proved to be the closest thing to true clone hyperlinking. However, technical issues with the Eclipse environment called for rather dirty workarounds to get the mechanism working as expected.

Rather surprisingly, tracking changes in Eclipse's editors proved to be easier than expected. Still, quite a lot of research had to be done to come up with an appropriate scheme to bundle atomic changes into transactions.

[15] A transaction is considered open when it has been created, but is not yet ended.

Chapter 4

CLONEBOARD Implementation

The previous chapters have described several preparatory steps towards the development of an Eclipse plug-in based on Mann's operations (cf. section 2.1). Based on the requirements elicited in section 2.3 and the experience gained by research and prototyping (cf. chapter 3), an Eclipse plug-in was implemented and dubbed CLONEBOARD.

This chapter outlines the implementation techniques used to build CLONEBOARD and details its structure and inner workings using standard UML diagrams.

4.1 Decomposition

CLONEBOARD was decomposed into six major components. Each of these parts fulfill a specific task. Figure 4.1 shows these components and their interactions. In the sections to follow, each of the components is detailed. To give an overview of the function of the components in the greater whole, each of them is briefly described below:

- **CloneBoard.** The prime component of the plug-in is the `CloneBoard` component. This component manages all responsibilities an Eclipse plug-in has, i.a. hooking the other components, handling errors and providing logging functionality.
- **CloneRepository.** All clone data is managed by the clone repository. This component serves as the model in the MVC architecture used by CLONEBOARD and is used by most of the other components.
- **TextEditorManager.** As clones are created using Eclipse's text editors – particularly its Java editor – quite a lot of CLONEBOARD's functionality relies on the interface with these editors. The `TextEditorManager` component manages this interface.
- **CloneManager.** The responsibility of creating clones and keeping the clone model up to date is delegated to the `CloneManager` component. Using services provided by the `TextEditorManager`, this component hooks into the text editors' copy and paste functionality.

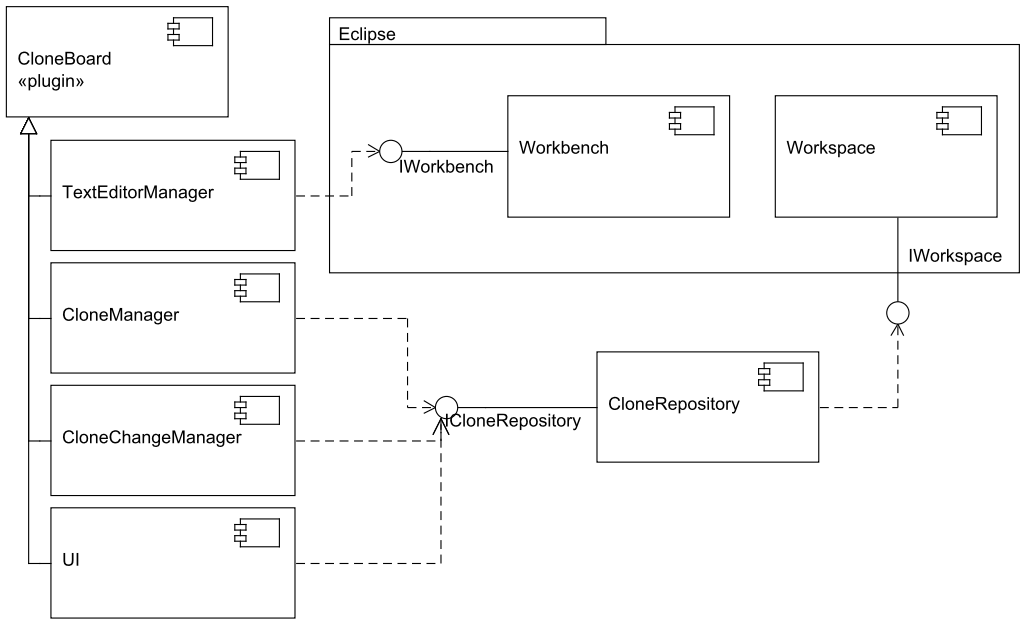


Figure 4.1: Basic decomposition of CLONEBOARD.

- **CloneChangeManager.** Changes to clones and the resolution of these changes are managed by the `CloneChangeManager`. To infer clone change transactions (cf. section 3.4) the services provided by the `TextEditorManager` are used. When the end of a transaction is detected, this component initiates the clone resolution process.
- **UI.** All user interaction is handled by the `UI` component. A number of distinct elements (i.a. a tree-view of the clone model, clone hyperlinks and queries regarding clone resolutions) jointly make up the user interface.

As can be seen from both the UML diagram and the above descriptions, a number of components (i.e. `CloneBoard`, `CloneRepository` and `TextEditorManager`) primarily fulfill a supporting role, whereas the other components provide the actual functionality.

4.2 Bootstrapping

Most Eclipse plug-ins contain a so called ‘activator’ class. This class is automatically activated by Eclipse when the plug-in is loaded, hence the name. The Eclipse framework contains a base implementation for user interface driven plug-ins, called `AbstractUIPlugin`. This class can serve as an activator and thus was a logical starting point for CLONEBOARD.

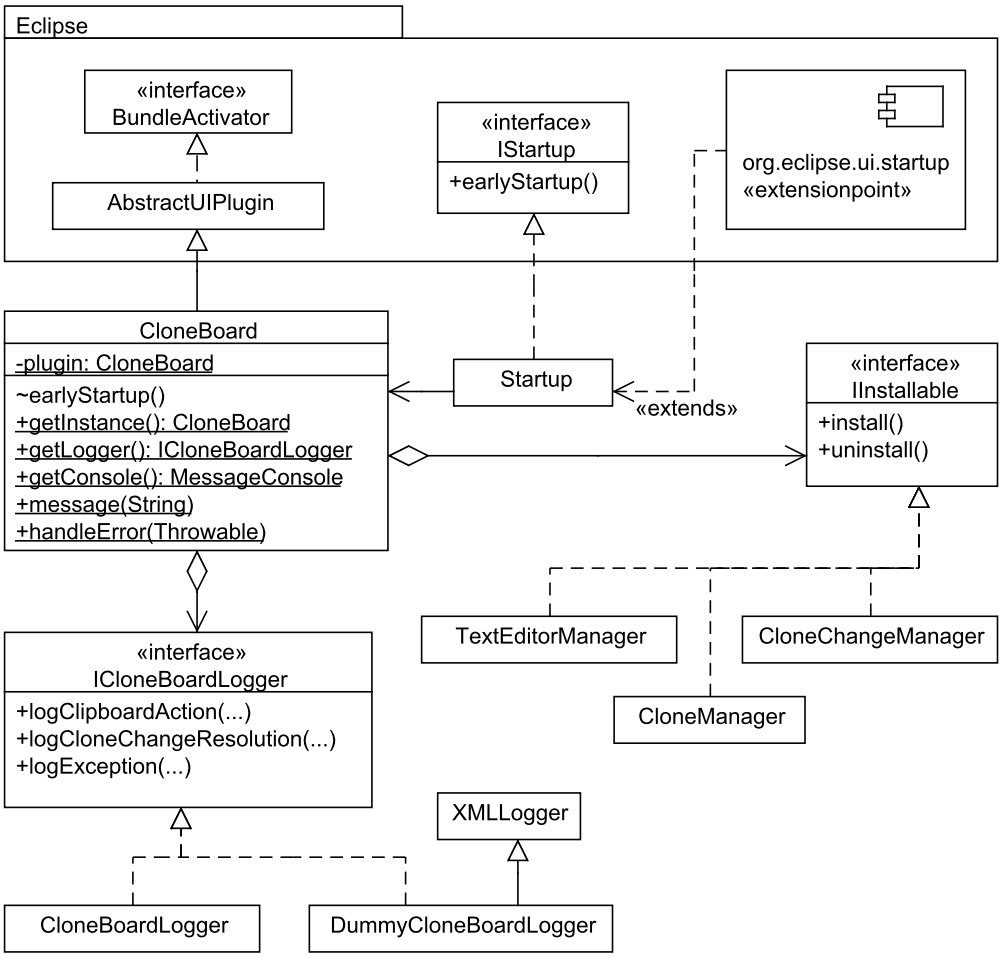


Figure 4.2: Classes and interfaces involved in the CloneBoard component.

Based on the `AbstractUIPlugin` class, the `CloneBoard` class was implemented. As shown in figure 4.2, this singleton class offers all basic functionality for the `CloneBoard` component, including logging, error handling and bootstrapping the other components.

4.2.1 Component Initialization

To initialize the other components, the plug-in makes use of two interfaces: `IStartup`, part of the Eclipse framework, and `IInstallable`. By extending Eclipse's `org.eclipse.ui.-startup` extension point, it is possible to be notified as soon as the Eclipse IDE has been initialized. As the class that implements the required `IStartup` interface a class separate from the `CloneBoard` class was created, to prevent double instantiation.[1] This class, called

[1] Double instantiation is of course undesirable for a singleton class.

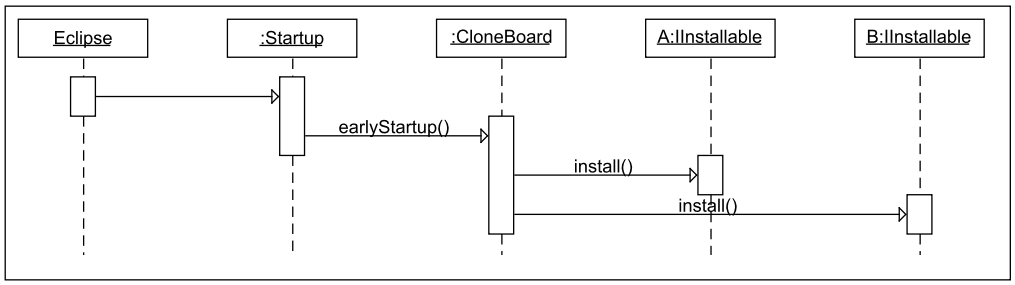


Figure 4.3: Sequence diagram of component bootstrapping process.

`Startup`, implements a single method that directly invokes the `earlyStartup()` method of `CloneBoard`.

Once notified of Eclipse's startup, the `CloneBoard` class starts to bootstrap the other components. Three of these components (cf. figure 4.2) implement the `IInstallable` interface, that defines methods for installing and uninstalling each of the components in the Eclipse framework. In this way, the bootstrapping procedure is transparent and allows for easy future expansion. Figure 4.3 shows a sequence diagram of the general bootstrapping procedure.

4.2.2 Logging

To facilitate the planned experiment, a logging mechanism was embedded in the `CloneBoard` component. Several events can be logged by using the `ICloneBoardLogger` interface, an instance of which is supplied by the `CloneBoard.getLogger()` method. Depending on circumstances[2], either a XML-based logging class will be supplied or a dummy class that discards all log messages.

The XML logs created by the `CloneBoardLogger` class do not adhere to a standard XML scheme, but use an ad-hoc scheme specific to the kind of log data being stored. Listing 4.1 shows an example of the log data that is written by the XML logger. The data is structured such, that it can later be interpreted easily using both automatic tools and by hand.

4.3 The Clone Model

To store clone data, several options were evaluated. The first thought was to create an object model to represent all clones and use either XML or a database for persistence. However, integrating this object model into the Eclipse environment, particularly in the text editors, proved to be quite difficult. Eclipse uses a model of markers and annotations (cf. section

[2]Mainly write rights on the system's disk.

Listing 4.1: Example of log data written by the `CloneBoardLogger` class.

```
<?xml version="1.0"?>
<log>
	<clipboard action="copy">
		<timestamp time="1231091935488">
			<![CDATA[4-jan-2009 18:58:55 CET]]>
		</timestamp>
		<fragmenttype type="COMMENT"/>
		<marker id="1618" type="cloneboard.ui.markers.temp">
			<attribute name="fragmentType">
				<![CDATA[COMMENT]]>
			</attribute>
			...
		</marker>
	</clipboard>
</log>
```

3.3) to associate objects with source code. To link an object model to source code, it would thus be necessary to create markers and annotations for each object in the model and keep both in sync.

Having two separate object models and keeping them in sync, did not seem to be a task sufficiently trivial to provide the required level of reliability. To cite Hoare: "The price of reliability is the pursuit of the utmost simplicity" [28]. A simpler model was thus sought after, and found. Instead of keeping two models in sync, one model – Eclipse's marker model – was wrapped with a second model.

4.3.1 Wrapping Eclipse's Marker Model

The diagram in figure 4.4 shows how the two models relate to each other. Every element in Eclipse's marker model is wrapped by a associated element in CLONEBOARD's clone model:

- **IClone $\mapsto$ IMarker.** Eclipse's markers represent a basic fragment of code. To represent a cloned piece of code, it suffices to wrap a marker in an object that represents a clone.
- **ICloneContainer $\mapsto$ IResource.** Markers are uniquely associated with resources. Resources can be files, folders, projects or other source entities. The `CloneRepository` component represents these resources by clone containers. These containers are an abstraction representing the concept of a source element containing clones.
- **ICloneRepository $\mapsto$ IWorkspaceRoot.** In Eclipse, resources form a hierarchy. At the top of this hierarchy, there is the workspace root. In CLONEBOARD's model, this root is represented by the `ICloneRepository`. In both Eclipse and CLONEBOARD, these entities are specializations of more general container elements.

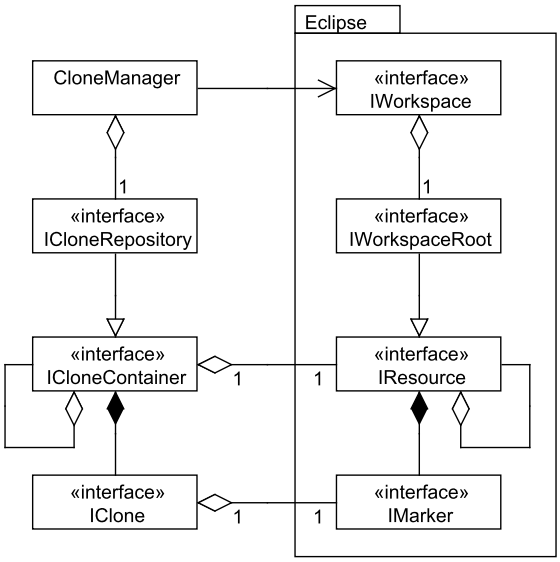


Figure 4.4: Model of the relation between the CLONEBOARD object model and Eclipse's marker model.

- **CloneManager ↦ IWorkspace.** The entity that governs all resources in Eclipse is the workspace. In CLONEBOARD, the repository of clone data is managed by the `CloneManager` class.

A deliberate choice was made to wrap each of Eclipse's model elements, rather than inherit from them, such as to hide away all the details of Eclipse's complicated model and present only those facts relevant for the cloning model.

4.3.2 Representing Clones as Markers

In the model that was thus created, each piece of cloned code is represented by an Eclipse resource marker. These markers are persisted by Eclipse automatically, so that no further persistence technology is required. Resource markers both have types and attributes. Marker types are organized hierarchically, so that there can be basic types and more specialized types.

CLONEBOARD defines five resource marker types, each serving a different purpose (cf. figure 4.5). At the root of the hierarchy, there is a basic marker. This marker type serves as a common ancestor to all other marker types. From this root, two main types are derived: a temporary marker and a clone marker. The temporary marker is used to mark a piece of code that has been copied to the clipboard, but has not yet been pasted. The clone marker is further divided into two subtypes: a clone source marker and a clone instance marker. Source markers mark the original copy of a clone, where instance markers are used to indicate all other clone instances in a clone set.

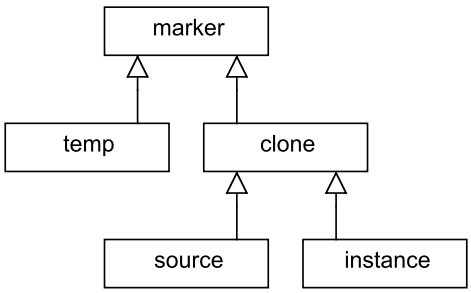


Figure 4.5: The resource marker types defined by CLONEBOARD.

4.3.3 Clone Interface Hierarchy

To handle the concept of *clone families* (or *clone sets*), two distinct types of clones were defined: a clone source and a clone instance. As shown in figure 4.6, one clone source aggregates one or more clone instances, thus effectively forming a clone family.

Each clone has a basic set of property getters (shown in the figure 4.6 as the first member box of `IClone`), which all wrap calls to getters on the `IMarker` interface. In other words: all data associated with a clone is actually stored in its respective marker object. The `IMarker` interface conveniently supports a key-value based attribute list, so that arbitrary data elements can be associated with markers easily.

To enhance the clone model's performance, some marker attributes and data that is derived of those are cached. With caching, the risk of running behind the facts is introduced. To counter this risk, an invalidation mechanism is employed. As soon as an external change to a marker is detected, the associated clone object is invalidated, effectively clearing its cache.

4.3.4 Clone Containers

The notion of a clone container provides an elegant abstraction to the diversity of Eclipse resource types that can directly or indirectly contain code clones. In CLONEBOARD's clone model, each clone container is associated with a resource. Furthermore, each container keeps a list of clones that are directly contained in it and a list of child containers. In practice, only files will contain clones, whereas other types of resources will generally have only child containers associated with them.

To benefit overall performance, a lazy loading [25] mechanism was implemented. That is, only if a container is specifically asked about its child containers or clones, it will inspect its associated resource and instantiate wrappers for all entities it finds. When a container's data is refreshed, for instance because it was signaled that the marker model of a resource has changed, all previously instantiated wrappers are maintained. In this way, memory

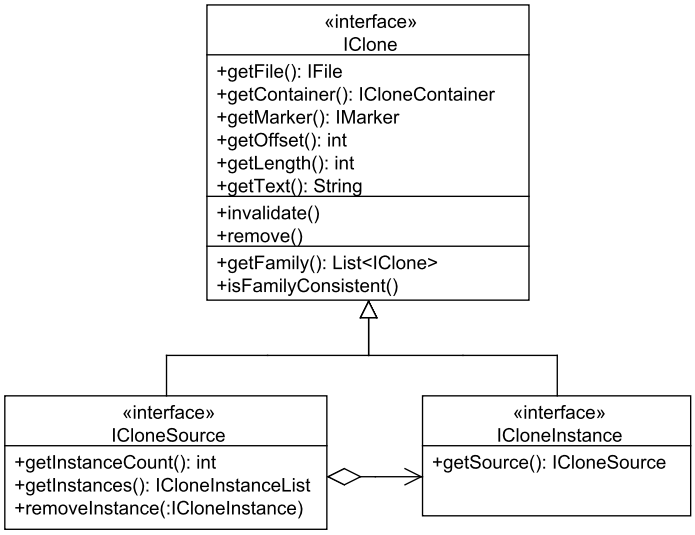


Figure 4.6: The `IClone` interface and its inheritors.

is conserved, but more importantly, a strict 1-to-1 relation between the wrappers and the entities they represent is enforced.

4.4 Interfacing with Text Editors

As discussed in sections 3.2 and 3.4, capturing clone creation and clone change events in Eclipse is not quite trivial. The Eclipse framework heavily uses the Observer design pattern [27], so the best way to get to know about events is to subscribe to all relevant observables and bring all gathered information to a central place. The `TextEditorManager` class is this junction. By employing the same Observer pattern Eclipse uses, the idTextEditorManager distributes its information to all interested components.

4.4.1 Managing Information

The `TextEditorManager` gathers its information from various sources, mainly by subscribing itself to observables using `addListener` methods. Figure 4.7 outlines the parties involved:

- **IPartService.** This service provides information about the state of all editors in the workspace. When new editors are opened, existing ones lose focus or get closed, this service sends a notification.
- **IDocument.** Every time an editor is opened, its associated document interface is retrieved. The `TextEditorManager` subscribes itself to document change events. These

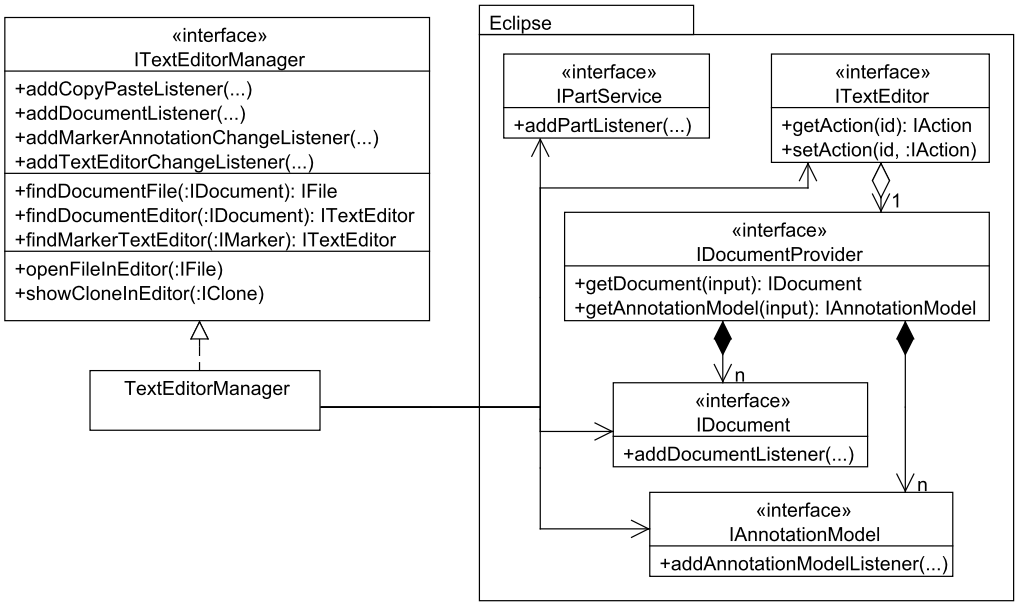


Figure 4.7: Model of the entities involved to integrate CLONEBOARD in Eclipse's text editors.

events are fired just before and just after every atomic document change. Each change is represented as a tuple: offset and length of replaced text and the replacement text. This information is sufficient to deduce the different types of document operations: insertion, deletion or replacement.

- **IAnnotationModel.** Information about changed text annotations is gathered in a way similar to the way document changes are obtained. The `IAnnotationModel` interface provides all necessary information.
- **ITextEditor.** To get notified of clipboard operations, some trickery was required. There is no notification mechanism for this sort of events in Eclipse. However, it did prove possible to retrieve and replace the `IAction` objects associated with each clipboard operation of a text editor. By wrapping the original action object in a proxy, existing functionality was maintained and a notification mechanism was established.

4.4.2 Reversing Relations

Eclipse's framework is highly normalized. This can for instance be seen in figure 4.7: documents are not directly related to text editors, but there is an intermediate party, the document provider, that links the two. These indirections make the framework very flexible, but at the same time makes it nearly impossible to navigate associations in the reverse direction. For instance, there is no direct way to relate a document to an editor.

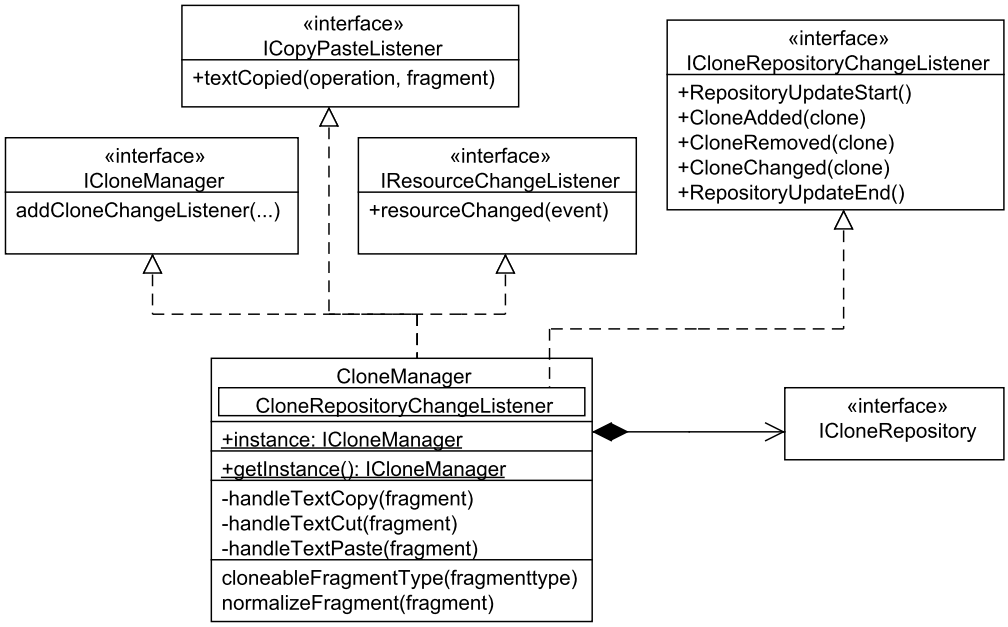


Figure 4.8: The CloneManager and the interfaces it implements.

As some features require to traverse some of the associations between documents, resources and text editors in the reverse direction, extra facilities had to be implemented for this in the `TextEditorManager`. Three methods, all prefixed with `find`, serve this purpose. The `TextEditorManager` keeps reverse indices of all relevant associations it detects, using the endpoint of each relation as the entry into the index. In this way, the associations can be reversed relatively easily, without a large computational overhead.

The alternative to this approach would be to iterate over collections of the entities that can possibly be the starting point of an association for which only the endpoint is known. Although this approach is less prone to errors,[3] it is computationally more expensive, which makes it unsuitable for inner-loop operations.

4.4.3 Opening Editors

Programmatically opening a source file in an editor is not quite a trivial task in Eclipse. Again, it are Eclipse's normalizations that complicate matters. As the task of opening an editor for a certain file, optionally highlighting a clone fragment in such a file, is required for some of CLONEBOARD's features, this functionality is embedded in the `TextEditorManager`.

[3] In contrast with the index-based approach, iterating over the actual entities guarantees up to date data. A reverse index may get out-dated and is thus less reliable.

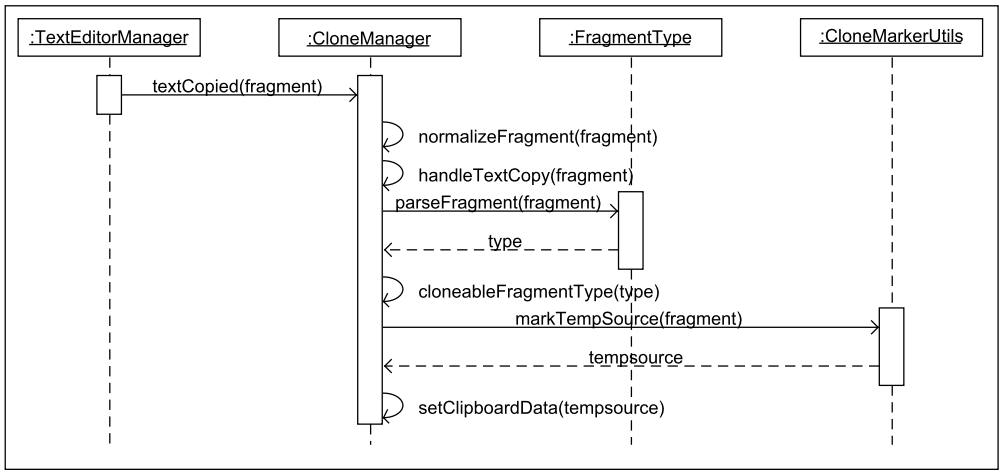


Figure 4.9: Sequence diagram of a text copy operation.

4.5 Capturing Copy and Paste Operations

One of the main tasks of the `CloneManager` component is to capture copy and paste operations and register non-trivial duplications as clones. To implement this functionality, the `CloneManager` class was conceived. The class exposes little functionality to other classes, but rather takes control of the management process and delegates tasks to other components.

4.5.1 Registering Clones

`CloneManager` registers itself as a copy and paste operation listener with the `TextEditorManager`. Every time a copy operation is detected, a dedicated handler is fired. This handler inspects the copied fragment and if it is not dismissed as a trivial clone, it is marked with a temporary marker. All necessary information to create a clone pair once the copied fragment is pasted again is added to the clipboard.[4]

Pasting a code fragment invariably leads to `CloneManager`'s intervention, too. The clone information stored on the clipboard during the copy operation is retrieved and used to link the pasted fragment to the original copy. As part of this process, the temporary marker that flagged the source copy is replaced by a definitive clone source marker. The process of copying and pasting is detailed in the sequence diagrams of figures 4.9 and 4.10.

Temporary markers are used in the process as an easy means of describing a code fragment. Other methods, such as a description based on offset and length of the fragment might

[4]Adding information to the clipboard actually did not prove to be possible. The only operations on the clipboard supported in Eclipse's SWT framework are getting and setting. The *add* operation had to be simulated by getting all clipboard data first, decoding as much of it as possible and slipping the additional data in before re-encoding the data and putting it back on the clipboard. In rare cases, this process may lead to data loss due to unrecognized encoding schemes.

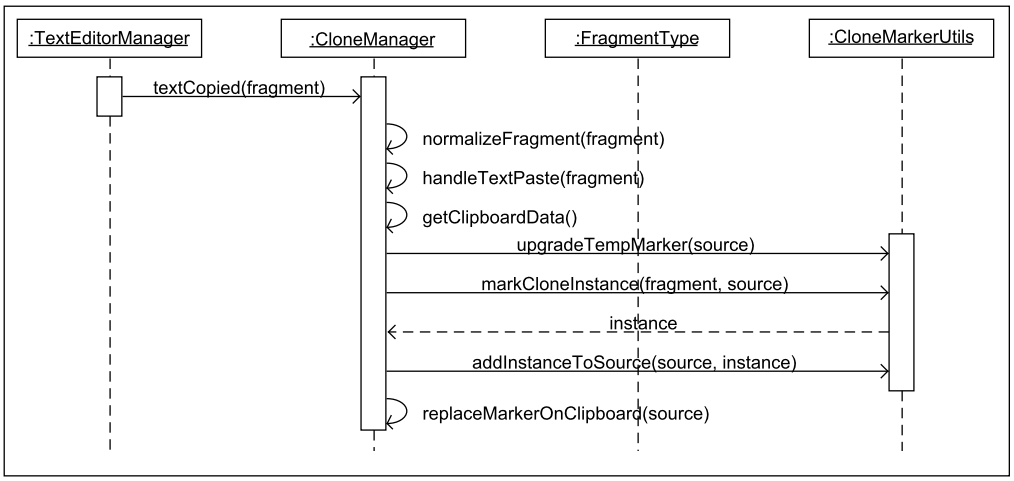


Figure 4.10: Sequence diagram of a text paste operation.

have been used, too, but these often have the disadvantage of being absolute and not robust against shifts of the fragment due to insertions and deletions in the code before the fragment. The temporary markers used are automatically updated by Eclipse to reflect changes of the environment. Furthermore, the markers are not persisted between sessions and are automatically removed by CLONEBOARD when a new fragment is copied.

4.5.2 Normalization and Classification

As can be seen in the sequence diagram of figure 4.9, copied fragments are being normalized and classified before they are being processed further. Only fragments that are considered relevant are treated as clone source. Other fragments are dismissed as spurious clones.

CLONEBOARD first normalizes all copied fragments. Normalization is a rather simple process that removes the leading and trailing whitespace from the fragment. These spaces are irrelevant for Java code, as they only determine the layout of the code. Eclipse's Java editor automatically adjusts the indentation of a code fragment when it is pasted, making it even less useful to consider white space.

Classification of potential clone fragments is done using a simple, heuristics based code analyzer, implemented by the `FragmentType` class. This analyzer detects the kind of code region that was cloned [35]. Regular expressions are used to analyze code fragments. The determination diagram in figure 4.11 shows the types differentiated by the analyzer and the heuristics used. A regular expressions-based heuristic analyzer is used rather than a island grammar-based parser [53] for performance reasons. Furthermore, the analyzer does not need to be very accurate and no information is extracted from the fragment other than its type, so that a full parser really would be overkill.

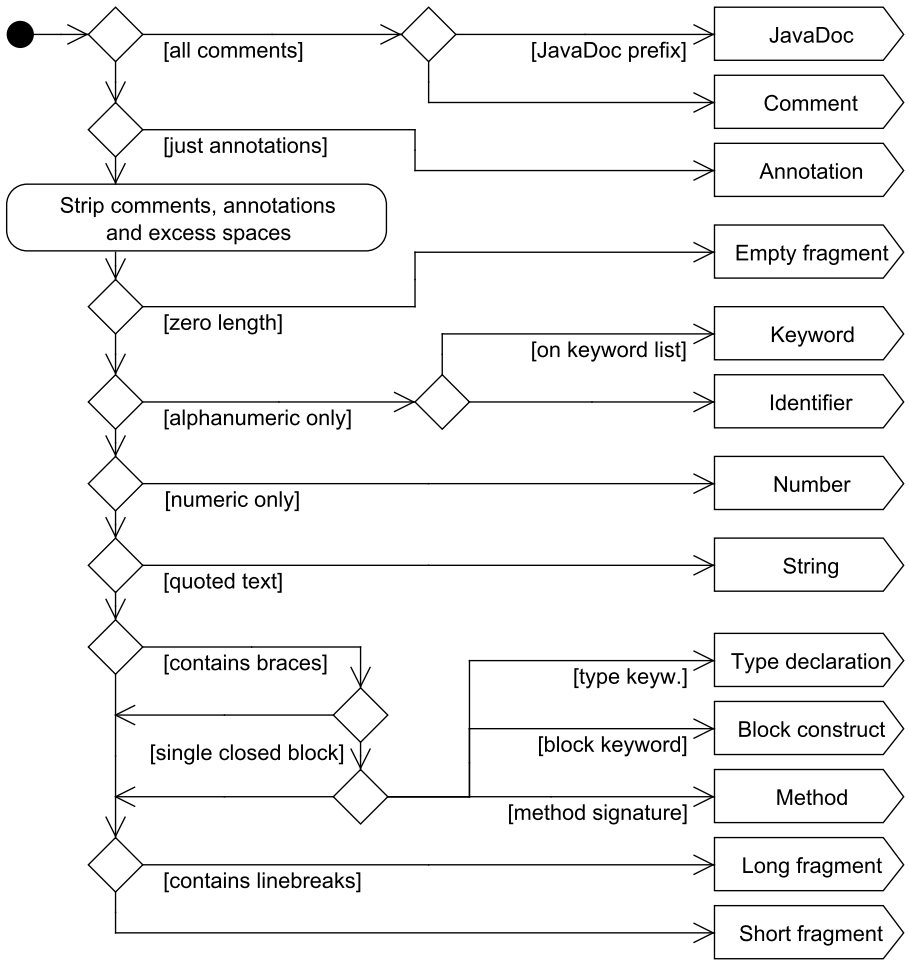


Figure 4.11: Determination scheme for heuristic fragment type detection.

Based on reason and experience, a number of fragment types have been declared trivial and are automatically dismissed by the `CloneManager` when such fragments are copied:

- **Empty fragments.** Trivially, fragments of zero length are filtered.
- **Comments and JavaDoc.** Fragments consisting of comments only pose no code inconsistency risks should they be copied and modified inconsistently.
- **Single keywords.** Language keywords (for instance access specifiers) are copied for convenience reasons, but can not be considered clones both because they are too small and because they cary no information. Furthermore, such fragments are never changed, but only replaced entirely or deleted.

- **Annotations.** Just like keywords, annotations[5] add little information. Arguably, some kinds of annotations, particularly custom and more complex annotations, might need to be excluded if the need arises.
- **Number.** Numeric constants are filtered, mainly because they are too small to be real clone candidates and generally carry too little information to be a likely source of inconsistent modification bugs.

4.5.3 Cutted and External Fragments

Two special situations may occur when the `CloneManager` handles a paste operation: the original fragment was cut rather than copied or the fragment was put on the clipboard by another application. CLONEBOARD handles both situation in the same, elegant way. When a pasted fragment, for whatever reasons, can not be linked to a source fragment, the pasted fragment is marked as the source of any future paste operations. In other words, the paste operation is interpreted as a copy operation. Should the same fragment be pasted again, then it will be automatically linked to the fragment that was first pasted.

4.6 Detecting and Handling Clone Changes

Thus far, components have been described that work together to register clone relations. The component that adds the actual clone change resolution functionality is the `CloneChangeManager`. In this component, detecting changes, grouping atomic changes into change transactions and resolving these changes is handled. One single class, by the same name as the component, implements all this behavior.

4.6.1 Detecting and Grouping Changes

To detect changes, the `CloneChangeManager` relies on the information provided to it by the `TextEditorManager`. Source code changes, or *document changes* as they are called in Eclipse lingo, are reported to the `CloneChangeManager` by means of the `IDocumentListener` interface (cf. section 4.4).

The `CloneChangeManager` responds to each reported document change by doing two things: a logical clock [46] is incremented and a document change handler is triggered. The logical clock is used to impose a total ordering on clone changes. This ordering is particularly useful in deciding which changes occurred after a document was last saved and before the document is discarded. Eclipse's marker persistence engine does not undo changes to marker when a document is discarded, so that the `CloneChangeManager` will have to do this itself.

[5] A Java annotation is an interface that is associated with a Java element to add special properties. One such – built-in – annotations is the `@override` annotation, that indicates an explicit method override.

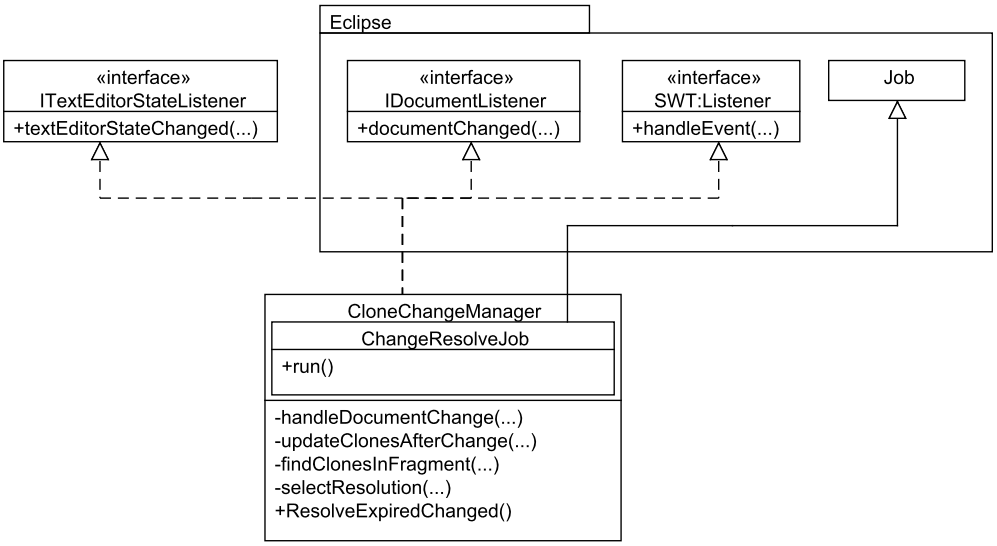


Figure 4.12: The CloneChangeManager and the functionality it implements.

To handle document changes, the `CloneChangeManager` performs a number of steps, some of which are directly related to the clone change management process, others are mainly intended to work around limitations in Eclipse's marker model:

- **Update marker positions.** Although Eclipse's marker model provides an automatic way to update the positions of markers when code is inserted or deleted before them, this algorithm did not prove to conform to the particular demands of the clone model. Parts of Eclipse's update algorithm were therefore overridden, mainly to be able to handle the correct interpretation of insertions just before or just after a clone.

  In CLONEBOARD's implementation, insertions right before or after a clone fragment are added to it when they contain just alphanumeric characters. During alpha testing, it was discovered that in certain boundary cases, such characters should be added to the related clone fragment.

- **Find affected clones.** For every document change, a list of affected clones is compiled. Every clone that overlaps with the changed fragment is said to be affected. Usually, this will be just one clone, but especially in large delete operations, this might be more than one clone instance.

- **Create and update transactions.** As soon as it is known which clones are affected by the change, new change transactions are opened for them or current transactions are updated, thus effectively implementing a grouping strategy. The last change timestamp of each transaction is updated to keep the transaction open.[6]

[6]The system clock rather than the logical clock described before is used for this, as time-outs measured in milliseconds are used to determine the end of a transaction.

Listing 4.2: Production rules used by CLONEBOARD's Java tokenizer in EBNF.

```
fragment         = { token } ;
token            = spaces | number | string | char |
                   complex identifier | eqop | comment | other ;
space char       = white space | tab | newline | linefeed | formfeed ;
spaces           = space char , { space char } ;
number           = digit , { digit } , [ "." { digit } ] |
                   "." { digit } ;
string           = '"' , { any - '"' } , '"' ;
char             = "'" , any - "'" , "'" ;
identifier char  = alphabetic char | digit | underscore ;
identifier       = identifier char , { identifier char } , [ "()" ] ;
complex identifier = identifier , { "." , identifier } ;
eqop             = ( "-" | "<" | ">" | "+" | "/" | "|" | "&" | "=" ) , "=" ;
comment          = "/*" , { any } , "*/" |
                   "//" , { any - newline } , newline ;
other            = any ;
```

- **Update clone model.** Finally, the clone model is updated to reflect the changes. Each clone marker stores its exact associated code fragment and this needs to be updated to the changed situation. Furthermore, clones that have become of zero length due to an extensive delete operation are removed.[7] This update process of course triggers notifications that viewers of the clone model can use to update their displays.

4.6.2 Calculate Differences

To be able to accurately judge what parts of a clone have been changed – this is relevant for determining applicable change resolutions – CLONEBOARD uses a comparison algorithm based on Hunt and McIlroy's famous *diff* program [64]. Instead of comparing lines for differences, tokens are compared. The result of this comparison is the shortest series of add and delete operations that transform the unchanged clone fragment into its altered form.

Clone fragments are decomposed into a stream of tokens by means of a very simple lexer that is part of CLONEBOARD. The lexer recognizes only a few terminal symbols, some of which are more complex than those typically found in a Java tokenizer. The production rules for this tokenizer are shown in listing 4.2. Most of the rules need no further explanation, but one is special: the `complex identifier`. A complex identifier is a series of one or more identifiers, possibly with an empty parameter list appended, separated by dots. These complex identifiers can thus identify fields and function calls without arguments. This special token is of particular use when it comes to parameterizing clones (cf. section 4.6.4).

[7] In the process of removing clones, other clones in the same family will have to be updated. Possibly, an entire clone family is discarded, should it otherwise end up with only one family member.

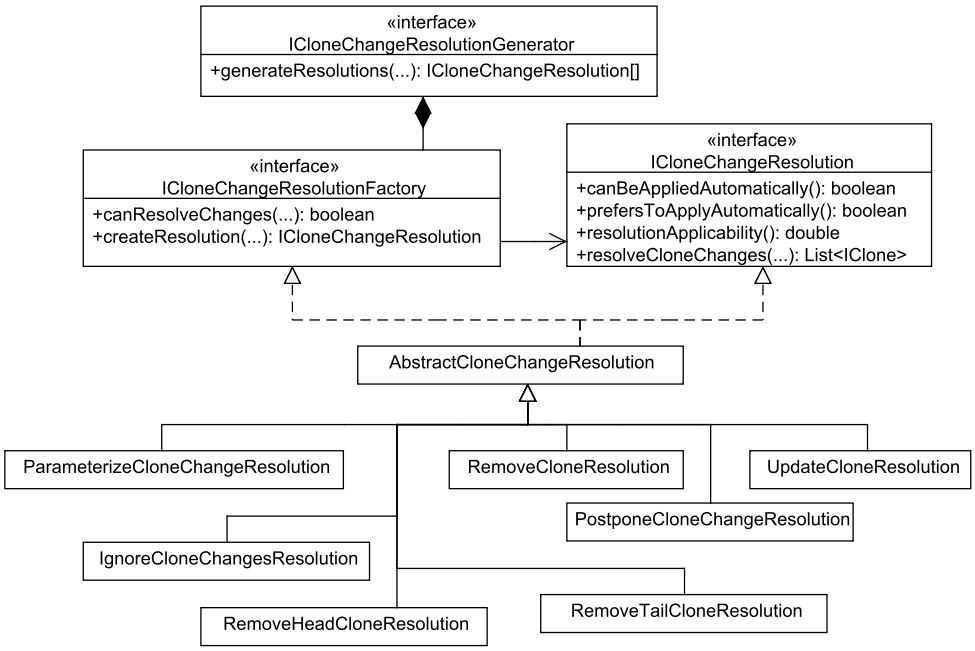


Figure 4.13: Implementation hierarchy of clone resolution classes.

4.6.3 Resolving Changes

The `CloneChangeManager` contains an innerclass that extends Eclipse's `Job` class. This inner class is used to run a scheduled job twice a second to check for expired clone change transactions. A transaction is said to have expired when it has not been changed for a certain amount of time. After this time, it will be automatically picked up by the job and handed to `CloneChangeManager`'s expiration handler.

Each expired change transaction is handled in the same manner. Before initiating a clone change resolution, a number of steps is performed:

- **Filter trivial changes.** Trivial changes, that is changes that only affect spaces and comments, are filtered out. For such changes, no clone change resolution is required as they do not affect clone family consistency.
- **Consistent again.** Changes that make an inconsistent clone family consistent again are excepted from change resolution, too. After all, such changes are actually very desirable from a clone management perspective.
- **Find apt resolutions.** Not all change resolution strategies (cf. section 2.2.3) always apply. Therefore a resolution generator is queried for applicable instances before the user is addressed. The resulting list is then sorted based on the applicability of each resolution (cf. section 4.6.6).

- **Query user.** When a list of suitable resolutions has been compiled, this list is handed over to a user interface broker with the request to query the user for the most appropriate course of action (cf. section 4.7.4). If the user has previously indicated that a certain resolution has to be applied by default, the querying will be skipped automatically.
- **Apply resolution.** If a resolution has been selected, it will be applied. Each resolution is designed as a separate object that implements a common interface (cf. figure 4.13). Among others, this interface provides a method `resolveCloneChanges` that resolves the specified changes and returns a list of affected clones. This list is used to automatically discard the resolution processes for clone changes that have been resolved as part of another clone's change resolution.

4.6.4 Parameterizing Clones

The *parameterize clone* change resolution can only be used when the number of tokens in a clone fragment has remained unchanged during the modification. This is a limitation of the resolution, caused by the design decision to assign parameters only on a per token basis. This is best explained using an example. Consider the original fragment below:

```
subtotal = quantity * price ;
```

If the boxed token ('`price`') would be replaced by multiple tokens, as shown below, multiple parameters would need to be defined for the new fragment (i.e. for every changed token one), whereas just one parameter would have to be defined for the other members of the family (that still have the old token).

```
subtotal = quantity * (price * 1.2) ;
```

Solving this discrepancy is not trivial, as it requires advanced techniques to keep track of parameters. These techniques are being actively researched and developed [22], but implementation of such algorithms was deemed to fall outside the scope of this thesis project.

CLONEBOARD transforms every changed token into a parameter, when parameterization is selected. Information about the tokens that are made parameters is stored in the clone marker as a list of token indices. When tokenizing a clone fragment that contains parameters, parameterized tokens are replaced by special markers that will match any token when compared in the *diff* algorithm. In this way, future changes to a parameter will not be considered an inconsistent clone change anymore.

4.6.5 Applying Changes to Clone Family

To apply the changes made to one clone to its family members, as required for the *Apply changes to all clones* change resolution, a simple and elegant solution was found. The output generated by the *diff* algorithm (i.e. a sequential list of addition and deletions) actually

formed the recipe to update other clones to the new, modified state. As parameterized tokens are never identified as being modified, no special measures needed to be taken to avoid updating these special tokens, resulting in a simple yet effective synchronization algorithm.

4.6.6 Determining Clone Change Resolution Applicability

As most computer users tend not to read on-screen messages completely, it is important to sort option lists presented to a user in such a way that the most relevant options occur first. In this way, one can be sure that at least the most relevant options will be considered.

This principle was used to aid the developer in selecting a clone resolution. Prior to querying the user for his preferences, each resolution object is asked to give an assessment of its applicability to the changes that are being dealt with. Resolutions get to rate their applicability on a scale from 0.0 to 1.0, where higher values correspond with a higher relevance. Table 4.1 lists the heuristics used to determine the applicabilities of the resolutions currently offered by CLONEBOARD. Most of the weights associated with the heuristic rules were determined by experimentation and reasoning. Field testing will have to be done to be able to fine tune these rules.

4.7 User Interface

The final part of the CLONEBOARD plug-in, the part were all other components come together, is the user interface. For CLONEBOARD, the most challenging part of the user interface is in providing a visualization overlay in the Java editor, including the ability to 'hyperlink' clone instances together (cf. section 3.3). In the following paragraphs, the implementation of this feature and other user interface elements is glanced over.

4.7.1 Visualizing and Hyperlinking Clones in Code

As previously discussed in chapter 3, Eclipse's marker and annotation model is the best candidate to be used for in-code clone visualization. Plug-ins have the opportunity to specify the presentation style for each marker type separately (cf. section 4.3.2 for marker types). Among the properties that can be set are background color of the marked code fragment, border styles and optionally a marker icon in the left margin.

After experimenting with the various visualization options, a very subtle scheme was chosen. In this scheme, clones are highlighted by a thin grey box around the duplicated code. All other presentation settings were deemed too prominent or obtrusive. As an alternative, a ruler bar was added to the margin of each code editor. This so called *CloneBar* shows grey bars before lines that contain clones. Lines containing multiple clones get multiple bars. To warn developers of inconsistently modified clones, the *CloneBar* shows red bars in the margin of inconsistent clone families. Clones the cursor is currently placed on, light up blue to help developers recognize the extent of the clone they are editing. Figure 4.14 shows an example of the *CloneBar* at work.

Table 4.1: Heuristics used to determine clone change resolution applicability.

Resolution	Applicability	Heuristic
Ignore changes	1.0	Clone family inconsistent
	0.5	Comment or JavaDoc
	0	Otherwise
Parameterize Clone	0.6 – 1.0	The more tokens replaced, the less applicable
	0	Changes too complex for algorithm
Postpone	0.8	Comment or JavaDoc
	0.7	Block construct or method declaration
	0.7	Uncategorized long fragment
	0.3	Otherwise
Unmark clone	1.0	Identifier, keyword or number
	0.9	Clone family inconsistent
	0.9	Less than 5 tokens in original fragment
	0.7	Number of changes exceeds 9
	0.3	Otherwise
Unmark head	0.8	First tokens of fragment were changed
	0.8	Method declaration
	0.6	Uncategorized short or long fragment
	0.5	Block construct
	0.3	The number of changes exceeds 4
	0.3	String literal
	0.1	Otherwise
Unmark tail	0.9	Last tokens of fragment were changed
	0.8	All changes occur in second half of fragment
	0.6	Uncategorized short or long fragment
	0.4	Block construct or a method declaration
	0.3	The number of changes exceeds 4
	0.3	String literal
	0.1	Otherwise
Apply to all	0.9	Fragment is an identifier
	0.4	A maximum of one change has been applied
	0.2	The number of changes exceeds 3
	0.1	Otherwise

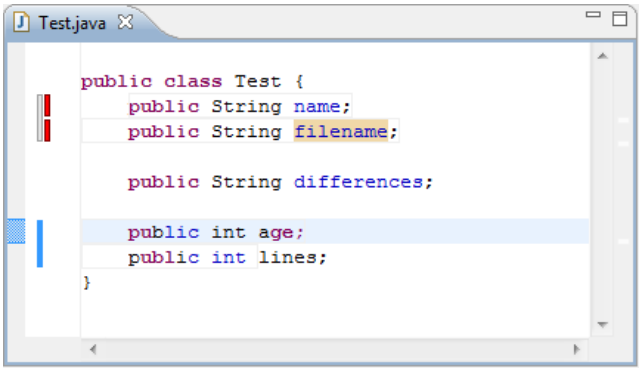


Figure 4.14: The CloneBar is used to subtly indicate the presence of clones.

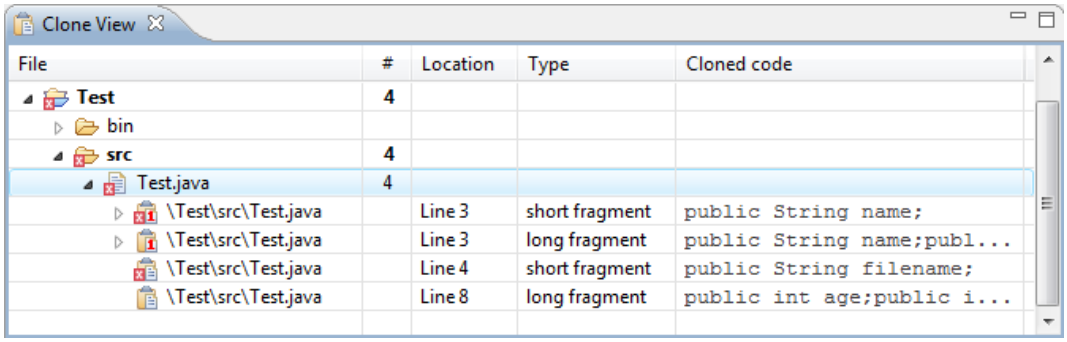
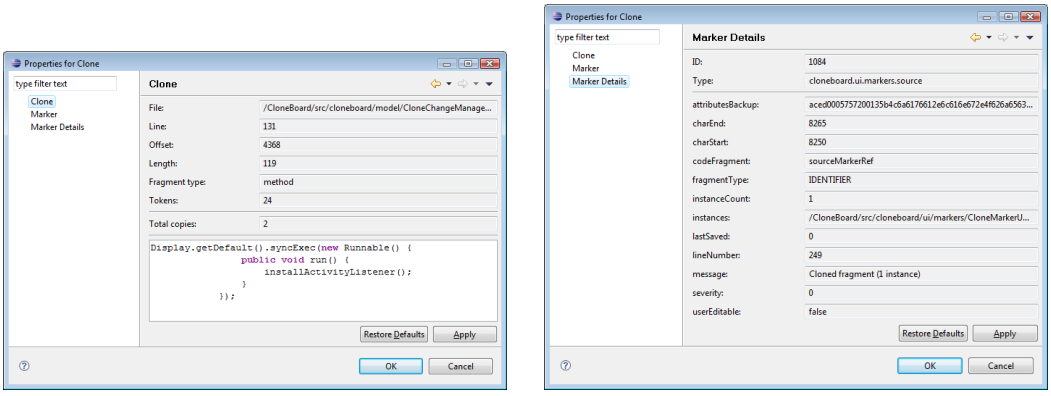


Figure 4.15: The CloneView gives a quick overview of a source base's clone fragments.

Finally, inter-clone hyperlinks were implemented using the technique described in section 3.3. Based on Eclipse's marker resolution mechanism, a popup window was realized that is shown when the user hovers his mouse sufficiently long over a clone fragment. Aside from hyperlinks to other clone family members, some other useful tools are offered as well (cf. figure 3.3.1). Among these options, the option to manually remove the clone fragment is offered. Furthermore, a user can view some basic properties of a clone fragment using a link present in the popup window (cf. section 4.7.3).

4.7.2 CloneView

To accommodate easy clone browsing and give developers a quick and easy overview of the clones in their source base, a hierarchical clone model viewer was implemented. In this view, all clones in the repository are shown, sorted by location. To prevent an information overload, a tree control was used to filter irrelevant data. The tree represents the structure of projects, folders and files the developer is already familiar with and adds to this entries for clones.



(a) Clone property page. (b) Detailed marker property page.

Figure 4.16: The Generic Elipse properties dialog is used to show clone properties.

As shown in figure 4.15, apart from each clone's location, its type and associated code fragment are shown in the view as well. To give developers an impression of the amount of clones in a particular project, folder or file, an extra column was added showing the total amount of clones for each container. Finally, a red overlay icon is applied to entries that either contain inconsistent clone families or represent members of such families. In this way, developers will be able to quickly see where the 'pain' is.

A context menu is provided for every entry in the CloneView, giving developers the option to directly navigate to the code fragment that contains the clone or highlight the source of the clone family.

The view was implemented based on the excellent explanation of Eclipse views provided in a book on plug-in development by Clayberg and Rubel [18].

In Eclipse, every view is implemented as a descendant of `ViewPart`. The contents of such a part are free to be determined by the developer, but in this case a standard treeview component was selected as its primary contents. All that was needed to hook the viewer to the clone model, was to implement a so called content provider: `ITreeContentProvider`. This interface specifies a number of methods the viewer can use to determine the root of a treeview and find each node's children.

Further presentation of each entry in the view was realised by implementing a series of interfaces used to handle layout. Among these, the most important is the `ITableLabelProvider` interface, that is used to determine the contents of each of the views contents.

4.7.3 Clone Properties Window

To provide some extra information about clones and ease the process of debugging CLONEBOARD, a property page for Eclipse's standard properties dialog was implemented (cf. figure (a)). A property dialog for a clone can be opened both using a hyperlink in the marker popup window or by right-clicking an entry in the CloneView.

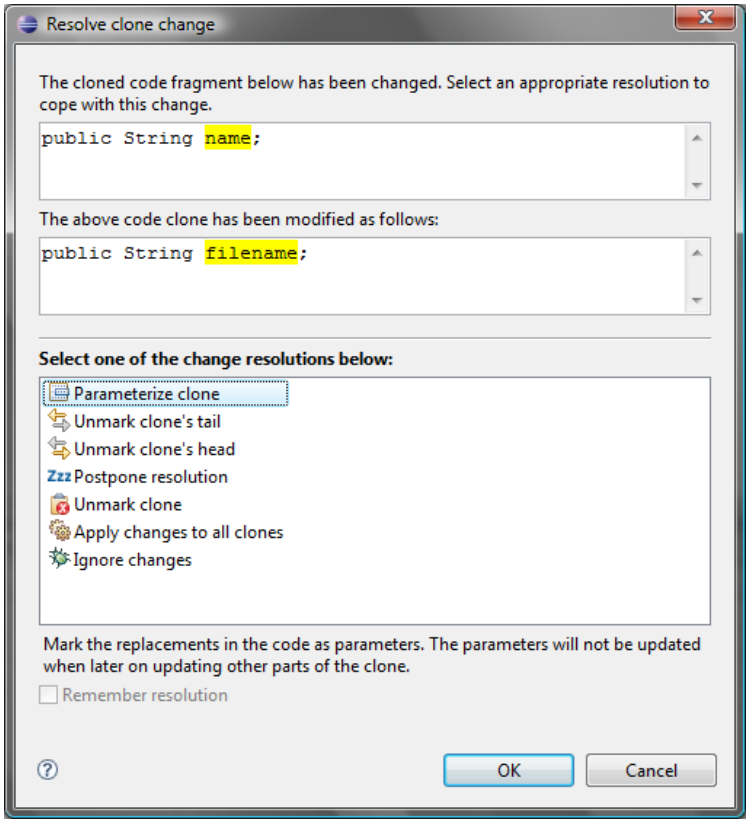


Figure 4.17: The clone change resolution window.

A property page is easily created by extending the `PropertyPage` class. Eclipse does not force a particular layout for property pages, but there are some conventions that provide some support.

In total two property pages were implemented. The first page shows information about a clone fragment itself. Some basic information is shown, including location and type of the fragment, as well as information regarding the clone family. Furthermore, the clone fragment itself is displayed outside of its context. Special highlighting is applied to tokens that were parameterized (cf. section 4.6.4).

The second property pages shows detailed information about the Eclipse marker used to demarcate the cloned fragment. This page shows additional information, not shown by Eclipse's default marker property page. As shown in figure (b), the property page mainly shows marker attributes. As such, it is a good way to inspect the inner workings of CLONEBOARD's hybrid clone model.

4.7.4 Resolution Window

As detailed in section 4.6.3, the `CloneChangeManager` uses a user interface broker to get a user interaction object it can use to query the user. Currently, CLONEBOARD contains only one such interaction object, which is the clone change resolution window. This window is the part of the user interface that will probably attract most attention, as it will pop up quite often.

The window is implemented as a simple dialog with an *OK* and *Cancel* button. Apart from a list of available resolutions the user can choose from, both the original and changed clone fragments are displayed. A simple highlighting scheme is used to show which parts of the clone have been changed. The data obtained from the *diff* comparison (cf. section 4.6.2) is used as a basis for the presentation. Each recorded deletion is highlighted in the original fragment, whereas each addition gets marked in the field that displays the modified clone text.

4.8 Considerations

To keep the thesis project on schedule as much as possible, a feature freeze was decided on two weeks before the planned experiment. There were at least a dozen more small and larger features still on the wish list when this decision was made. In this final section of the chapter, some of the features missing in the first version of CLONEBOARD that are really worth to be discussed are touched upon.

4.8.1 CloneBoard Extension Points

Most design decisions made during the initial decomposition of CLONEBOARD were based on the assumption that the design should allow for easy extension. The *de facto* way of extending Eclipse is by writing new plug-ins that subscribe themselves to predefined extension points. Some parts of CLONEBOARD lend themselves for this sort of extension, too. One of the wishes therefore was to define a set of new extension points that could be used by other plug-in developers later on to add new resolutions and resolution querying interfaces.

Having these new extension points, however, was considered a bit of a luxury problem, as currently there are no likely candidates to extend CLONEBOARD. Implementing extension points for internal use only, seemed to be a rather pretentious *tour de force*, and was thus postponed, hoping that time would remain for such a feat of strength.

4.8.2 Support for Code Repositories

CLONEBOARD's clone data is stored in Eclipse markers. These markers are persisted in proprietary binary files on a per-project basis. When working on a software project in a team, central code repositories are used to share sources and merge the results of each developer's individual labour. The binary format used by Eclipse is not suitable for use

with code repositories, as merging of binary files is generally considered impossible without detailed knowledge of the file format.

In the early stages of CLONEBOARD's development, an alternative persistence engine was considered. Particularly, an engine more suitable for use in a multi-developer environment. Implementing or integrating such technology into CLONEBOARD was, however, considered too much effort, considering that it had no added value for the experiment. As can be read in chapter 5, the experiment was to be performed on individual developers rather than teams of developers, making support for code repository futile.

4.8.3 Alternative Resolution Querying

During alpha-testing and the pilot experiment, CLONEBOARD's resolution querying mechanism (cf. section 4.7.4) was found to be more obtrusive than originally expected. Some slight modifications were made to the `CloneChangeManager` to counter part of this effect, but the need for a more subtle querying mechanism remained irrefutable.

Alternative methods to query a user for his preferred clone change resolution were considered. Among those was a techniques similar to that used by Eclipse to report code errors: underlining erroneous code fragments with a distinctive squiggly line. Such markers clearly indicate to developers that something is wrong, but leaves the choice whether and when to solve the issue to their own preference. Unfortunately, no more time was available to implement this alternative querying style as a feature freeze had already been effectuated.

4.9 Summary

This chapter has been a rather technical report on the implementation details of CLONEBOARD. With its various UML diagrams it serves mainly to outline the techniques that were used to bring all required technology together in one consistent Eclipse plug-in.

Among other things, the technology used to integrate a clone model in Eclipse's marker model was described, just as it was shown what logic was required to be able to dynamically deduce clone's from copy and paste operations. Furthermore, it was shown how the various user-interface components of CLONEBOARD came to be.

In a separate section, some reflective considerations with regard to CLONEBOARD's implementation were made explicit. Given the relatively short time for a thesis project, it is only logical that not all wishes and second thoughts can be resolved within the project's scope.

Chapter 5

Experiment

Tinkering with a technician's tools is dangerous. Tools tend to become an inseparable part of those who work with them. And just as one would not be greeted with gratitude for tweaking a worker's wrist joint, no unconditional appreciation is to be expected for cordially made adjustments to his tools either. All the more reason to be very careful when altering a software engineer's development environment. And as CLONEBOARD does change parts of a developer's work process, putting it to a test in a controlled setting before praising its benefits is a definite recommendation.

To measure CLONEBOARD's impact on the development process and see whether it would be able to effectively assist in the clone management problem, it was tested in an experiment. A number of subjects were asked to develop code in an Eclipse environment that was extended with the CLONEBOARD plug-in. By carefully recording their findings, a better evaluation of CLONEBOARD and the principles it is based on was made possible.

In the remained of this chapter, the conducted experiment is described and its results are outlined. Its first section will deal with the experimental design. Before the experiment was carried out, its design was first evaluated by means of a pilot. After some fine tuning and sharpening, the actual experiment was conducted. All of these steps, as well as an after-the-fact evaluation of the experiment and a report on the validity of its outcomes are described in the sections to come.

5.1 Experimental Design

Experimental design is an important aspect of any experiment. Only when an experiment is designed well, can its results be interpreted and validly generalized. The first step in experimental design is to get two important matters straight: firstly, what concepts are to be examined, and based on that, what type of experiment is best suited to investigate these variables.

5.1.1 Variables

The subject of the experiment to be conducted will be CLONEBOARD, a tool developed as part of this thesis project, that is implemented based on concepts introduced by Mann [51] (cf. section 2.1). The CLONEBOARD tool itself will be the independent variable in the experiment, as the goal is to see what change the tool can make to the development process.

The research questions posed in chapter 1 offer a good starting point to determine the dependent variables. Two of these questions relate to technical possibilities (i.e. #1 and #3), whereas the other two relate to more measurable concepts. Only these latter two are thus suitable to be examined by means of an experiment. Paraphrased in the context of the proposed experiment, these questions would become:

- **Question #2.** Is CLONEBOARD sufficiently useful and usable for developers?
- **Question #4.** Will CLONEBOARD help reduce cloning related problems?

The dependent variables to be examined can be deduced from these questions relatively easily. Question #2 maps to two variables: adequacy and usability. The other question yields a third dependent variable, namely effectiveness.

- **Adequacy.** A tool is said to be adequate when it is sufficiently able to perform the tasks its users expect it to do.
- **Usability.** When a tool can be operated in a comprehensible way, without causing grave discomfort to its user, it is said to be easy to use.
- **Effectiveness.** An effective tool is one that is able to yield the intended results and achieves the goals that it is used for.

5.1.2 Experiment Type

Given the conditions for this experiment (i.e. testing a new tool on a number of points), some types of experiments are better suited than others. As a new tool is tested, conducting a historic case study is of little use. Given the relatively short amount of time, introducing the tool with a number of subjects and studying them for a longer period of time (i.e. a longitudinal study) is ruled out, too. The most reliable and achievable type of experiment is probably the controlled experiment, as it allows the concepts to be examined in a controlled environment in a relatively short amount of time. Sjoeberg *et al.* 's definition of a controlled experiment (in a software engineering context) underlines this suspicion:

> "A randomized experiment or a quasi-experiment in which individuals or teams conduct one or more software engineering tasks for the sake of comparing different populations, processes, methods, techniques, languages, or tools." – [57]

Ideally, a controlled experiment would be performed by using two sets of test subjects: a control group and an experimental group. The first group would perform a series of tasks without the new tool, and the experimental group gets to do the same work with the new

tool. However, the nature of the dependent variables is such that it is hard to measure them without using the tool. In other words, it is only possible to measure the effects of the tool's presence, not of its absence.

The only dependent variable that could be tested both with and without the tool, would be the tool's effectiveness: if the subjects in the test group experience less clone related problems than the subjects of the control group, this effect might be attributed to the tool's effectiveness. Measuring the amount of clone related problems subjects experience objectively is quite difficult in the context of a controlled experiment. Only in a longitudinal study could such effects be measured with some reliability.

These considerations led to the conclusion that a classic controlled experiment would not be feasible. As a second-best option, the *one-group pretest-posttest preexperimental design* [16] was chosen. This type of designs is called preexperimental to indicate that "[it does] not meet the scientific standards of experimental design" [3]. In other words, an experiment by this design can not be used to report real findings, but rather just findings. The difference between these two is clearly defined by Brooks[1]:

> "*Findings* will be those results properly established by soundly-designed experiments, and stated in terms of the domain for which generalization is valid.
>
> *Observations* will be reports of facts of real user-behavior, even those observed in under-controlled, limited-sample experiences." – [13]

So, although an experiment like this will not be able to provide hard facts, it still has its use. Or as Brooks puts it: "Any data are better than none." Valuable information about the value of both Mann's proposed operators and CLONEBOARD's implementation of them can be obtained, and only through experimentation.

5.1.3 One-group Pretest-posttest Design

In a one-group pretest-posttest preexperimental design, only one group is tested. There is no control group. Instead of having a control group, the experimental group is subjected to an extra test before the experiment is conducted. This test serves as a base measurement to which the measurements gathered after the experiment can be compared. Although this approach has an inherently higher risk of invalidity (cf. section 5.10), it can still render useful observations when executed carefully.

During pretesting and posttesting, the subjects are measured in terms of the dependent variables. Usually, the same questionnaire is used both before and after varying the independent variable (i.e. introducing the tool). By using the same questions, the pretest and posttest results can be compared easier. However, when asking the same series of questions twice, subjects may tend to 'clean up' there answers in the second round, as they realize what purpose the questions serve and want to show themselves off by answering in a more desirable way. These risks will have to be considered carefully, both during and after the experiment.

[1] Brooks also wrote a famous book on software engineering, called "The Mythical Man-Month" [14]

5.1.4 Selecting a Case

The independent variable being exanined is CLONEBOARD. To test its influence on the dependent variables, subjects will have to be exposed to CLONEBOARD, preferably in a way that maximizes generalizability of the final results. One way to do this, is by simulating a development task in an environment that includes CLONEBOARD. In this way, subjects are exposed to the tool in a way that resembles the conditions of its intended use.

Simulating a development task is not easy. For this experiment, the choice was made to pick a certain existing software system as a case and give subjects a number of programming assignments that involve modifying existing source code. For this approach to be successful, a number of important conditions have to be met:

- **Realistic case.** First of all, the selected case should resemble a real-life software system. Even though the system may be smaller and itself not relate to corporate systems, it should have all the properties of such a system.
- **Sufficiently complex.** A case that is too simple does not allow the experiment's findings to be generalized to the complex systems developers typically encounter in a commercial or scientific context.
- **Sufficiently interesting.** To get the test subjects involved in the case, which will help them forget the research setting and hopefully show more natural behavior, the case should be sufficiently interesting. A case that is 'fun' to work on is more likely to draw subjects back into their normal development routines than a tedious task would.
- **Equally familiar to all.** To eliminate the potential number of external variables that influence the dependents, conditions should be as similar as possible for all subjects. One likely candidate to influence the results is a subject's degree of familiarity with the case. A developer that is familiar with a system is likely to show other behavior than one that is still new to it.
- **Easy to learn.** When a case is selected that is unknown to all test subjects, it should be sufficiently easy to learn. As the time for the experiment is limited, the amount of time spent on getting to know the system should be limited as much as possible.

After some careful considerations, two candidate cases were selected: CheckStyle[2] and RoboCode[3]. Both systems are open source, relatively small, easy to learn, sufficiently complex and written in Java. CheckStyle is the more serious of the two, being an extensible 'spell checker' for Java code. RoboCode, on the other hand, is an AI programming puzzle and as such is a little more playful. Finally, on the grounds of being the least known and most interesting of the two candidates, RoboCode was selected as the case.

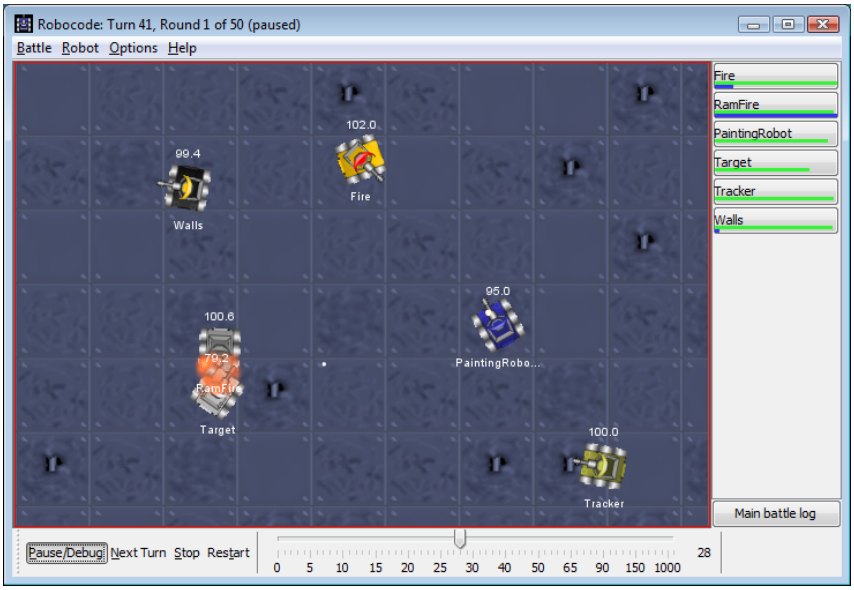


Figure 5.1: In RoboCode, artificially intelligent agents struggle for survival.

5.1.5 About RoboCode

The selected case, i.e. RoboCode, can essentially be described as a programming puzzle. RoboCode implements a simulation framework in which several artificially intelligent agents are confronted to each other to find the one with the most superior logic (cf. figure 5.1). Each of the agents represents a robot, a kind of a modern style gladiator, that is put into an arena with other robots in a struggle for victory. The robot that lasts longest is proclaimed the winner. Usually a great number of simulations (or 'rounds') is run to get statistical evidence and rule out shear luck.

To compete with each other, each robot can perform three different types of actions: moving, scanning and firing. The most intelligent combination of these three actions determines the winner.

- **Moving.** The arena in which the robots compete has preset dimensions. Each robot is free to move around in this arena, within the bounds of physics. Each robot has a maximum speed and acceleration and moving costs energy.
- **Scanning.** Using a device that most resembles a conventional radar, each agent can scan its surroundings, looking for opponents. Only by scanning, information about the opposition can be acquired.
- **Firing.** To reduce the opposing robots' energy, bullets can be fired. The amount of destructive energy of a bullet can be varied, and firing each bullet will drain the firing agent's energy resources.

[2]See `http://checkstyle.sourceforge.net/`
[3]See `http://robocode.sourceforge.net/`

New agents are created by subclassing a base robot class provided by the RoboCode framework. Each robot runs its own thread, issuing commands to perform its basic actions. Each command given will block the threads execution for a time interval that corresponds with the time it would physically take to perform the action. Information about the arena and the other contestants is presented to the robot by means of events.

5.2 Pretest and Posttest

As the dependent variables that are studied (i.e. adequacy, usability and effectiveness of CLONEBOARD) are all rather subjective qualities, gauging them using some sort of a measurement instrument is not possible. Instead, the experimental subjects will have to be *asked* about their opinions regarding these variables. The most suitable and reproducible way to do this, is by means of a questionnaire, preferably with closed-ended questions only. Open-ended questions are harder to process and more difficult to interpret. Closed-ended questions, on the other hand induce the risk of limiting the subject's responses too much, thereby not measuring some potentially important variables [3].

A common approach to gauge subject's opinions about certain concepts is to use matrix questions in which respondents can rate a number of statements on a 1 to 5 scale, ranging from 'strongly disagree' to 'strongly agree' (the so called *Likert scale* [49]). Such matrices are easy for subjects to fill out and give them a manageable amount of freedom to express their opinions. The drawback of this method, on the other hand, is that subjects may start seeing a certain pattern in their answers and fill out the rest of the questions in line with this pattern instead of their opinion [3]. A way to remedy this effect is to mix the orientations of questions[4], such that the likeliness that subjects 'discover' trivial patterns is reduced.

5.2.1 Pretest design

For the pretest, a total number of five themes were chosen. Each theme relates to a different aspect of the experiment. Most themes are intended to determine possible external variables that might influence the dependent variables, other than the independent variable that is being examined (i.e. CLONEBOARD).

- **Personal background.** First of all, some personal details of the experimental subjects were asked. Relevant details were thought to be age, education level and current professional occupation. Gender was not included in the questions, as this could be easily deduced from both seeing the subject and realizing that the chance of finding a female respondent on a computer science faculty is next to zero anyhow.
- **Development experience.** To get an impression of the subject's experience in developing software, a number of statements were included that relate to programming experience. As a control question, participants were asked whether they had any experience with RoboCode.

[4] By orientation is meant, whether agreement to the key concept means having to answer the question postively or negatively.

- **Attitude towards code quality.** As developers young and old tend to have different attitudes towards the importance of good quality code, a number of statements were added that attempt to gauge each developer's standpoint.
- **Attitude towards cloning.** A subject's attitude towards cloning might influence its expectations with respect to a tool that supports clone management. To assess the extent of this effect, a number of statements that gauge each subject's familiarity with the concept of cloning was added.
- **Expectations for a tool like CLONEBOARD.** Finally, a series of statements was conceived to measure the actual dependent variables. The subjects were given a brief description of CLONEBOARD but in abstract terms as they were not yet familiar with the tool, followed by statements about such a tool. The description given is printed below.

  > "With a clone management tool, one should be able to see what parts of code have been cloned at any time. Such a tool should give a developer the opportunity to inspect cloning on a per file basis. Furthermore, the tool should alert a developer whenever he is changing a cloned fragment, offering several resolution strategies to cope with the changes. Among such strategies should be the options to update all clone instances."

The exact list of questions and statements used in the pretest is printed in appendices B and C.

5.2.2 Posttest Design

After the subjects have completed their assignments, or reached the end of the predesignated amount of time, they had to fill out a second questionnaire serving as a posttest. This test's primary intent was to measure whether the subject's expectations with regard to a clone management system have been fulfilled and CLONEBOARD is found to be a useful example of such a tool.

In the posttest, a number of different issues were addressed. First of all, a number of checks were performed to see that the experiment went well and to assess the amount of interference by external variables. For this purpose, two Likert scale matrix questions were formulated.

- **Assignments Experience.** The first set of statements relates to the subject's experience of the assignments. Were they too hard? Did the subject feel any time pressure that might have a negative result on his performance or perception? Were the assignments sufficiently inspiring?
- **Development Style.** Subjects were asked to indicate their coding habits in question 3 of the pretest. To see whether the subjects followed these principles during the execution of the assignments, a number of similar questions were asked in the posttest. Major differences in style may indicate a Hawthorne effect [47].[5]

[5] Landsberger defined the Hawthorne effect as "a short-term improvement caused by observing worker performance."

Two more matrix questions were added to assess the subject's experiences with the CLONEBOARD user-interface. These questions are intended to measure to what extent the subject noticed the presence of CLONEBOARD and actively used it. Furthermore, questions to gauge the subject's appreciation of the features offered were included.

- **UI Experience.** The first questions measures some general aspects of the subject's interaction with CLONEBOARD.
- **Resolution Window Experience.** More statements relating specifically to the clone change resolution window are grouped into a second question.

To get an impression of the subject's perception of the change resolutions, two questions directly relating to the seven resolution options implemented in CLONEBOARD were added:

- **Resolution Frequency.** In a matrix question, all seven resolutions were listed with the request to give a rough indication of the number of times the subject used each of the resolutions during the assignments. As a scale, five different values were chosen: never, once, 2–5 times, 6–10 times and more than 10 times. Given the relatively short time given for the assignments and existing figures about developer copy and paste usage [38], this scale seemed to be fair.
- **Resolution Value.** The second matrix questions asked subjects to rate the usefulness of each of the resolution on a Likert scale. If a resolution was never used, the participants were asked to indicate how useful they thought the resolution would be.

The seventh question of the questionnaire addresses the dependent variables. Subjects were given the same seven statements as used in question 5 of the pretest, only changed to feature the name of the clone management tool, namely CLONEBOARD. The answers to these questions can be directly compared to those given in the pretest, to see to what extent CLONEBOARD meets up with the subjects expectations for a tool like it.

To further measure the participants perception of the CLONEBOARD user interface and to see to what extent problems in the usability of the tool hindered its use, an extra series of Likert-scale statements was added.

As a means to assert that problems with the case, the execution of the experiment, its documentation or the questionnaires did not have a negative effect on its validity, subjects were asked to rate a final series of statements. Instead of a Likert scale a scale from 1 to 9 was used to enable participants to give a more fine-grained answer. A ten point scale was deliberately abstrained from, as especially academically educated tend to find *10* an ireal rating.

On the back of the posttest questionnaire, participants were given space to write down some comments or suggestions they might have.

A full list of the questions on the posttest questionnaires is given in appendix C.

5.3 Programming Assignments

Considering the amount of time a volunteer would maximally want to spend on an experiment, the duration for the programming assignments was fixed at two hours. Ideally, one would track participants much longer, but this would be nearly impossible without compensating subjects for their expenses. And as is widely known, a software developer's time is not cheap.

Thus, the programming assignments to be given to the participants should be such that completing them in two hours is possible. Furthermore, as time is short subjects should not have to spend too much time to get to know the case, before starting their programming work. The RoboCode case chosen for the assignments is very suitable to match these requirements, as it is easy to get acquainted with and a robot can be developed in a relatively short amount of time.

5.3.1 Initial Design

At first, a series of five assignments was drawn up, leading the experimental participants through all stages of developing a robot. These stages include processing radar data into a 2D representation of the other robots' positions and implementing routines to effectively move to a predetermined position in the arena. Both tasks involve a non-trivial amount of trigonometry. Furthermore, AI code would have to be implemented in order to effectively hunt the robot's enemies.

During the experiment's pilot, the relatively large amount of trigonometry involved in building a robot proved to be an obstacle. As quite a lot of the subject's time went into figuring out the math, less cloning-related data was gathered. To counter this negative effect, the assignments were redesigned.

5.3.2 Final Assignments

For the final assignments, a basic robot implementation was created in which all trigonometry was embedded. Basic operations, such as moving, scanning, aiming and firing were all abstracted from and more convenient methods to drive the robot were implemented. As a direct consequence, the programming assignments could be much simplified, while still offering the subject's the rather motivating opportunity to develop their own, intelligent agent.

Attempts were made to include tasks that would provoke code cloning. Among developers' motives for cloning described in literature are reuse of complicated control structures [38, 36] and the lack of language support for secondary concerns [59]. In other words, asking the experimental subjects to implement variations of a given algorithm and letting them implement a secondary concern (e.g. logging) are likely ways to provoke cloning behavior.

Based on these observations, a set of five programming assignments was drawn up. The last assignment was deliberately designed open-ended, so that the occasional participant that

would complete the other assignments ahead of time would still be able to fill the remainder of the two hours with the final assignment.

In the first two assignments, the subjects were asked to implement logging functionality. These assignments both help participants to get to know the software better and are likely to give rise to code cloning. The third assignment requires the developers to implement a series of variations on an already implemented target selection algorithm. The required variations were designed not to be to big, making them excellent candidates for code duplication.

Having inspired the experimental subjects by the design patterns of the third assignment, they are requested to implement a different set of algorithms for which the same design pattern could be used. The assignment's description explicitly hints at these similarities, by that both aiding the subjects in finding a solution and increasing the chances of more cloning.

Finally, the fifth assignment more or less gives participants *carte blanche* to extend and alter the robot code as they set fit. Some possible directions for improvements were hinted on, but not made too explicit.

The final list of assignments used in the experiment is detailed in appendix D.

5.3.3 Additional Documentation

To give each experimental subject an easy introduction to both RoboCode and CLONEBOARD, introductory documentation on these systems was handed out to them. This documentation (printed out in appendix F from page 117 onwards) outlines the structure of the case and CLONEBOARD's user-interface and gives some usful background information.

In addition to two pages of documentation text, a laminated reference sheet was handed out to participants. This sheet shows screenshots of the main parts of both RoboCode and CLONEBOARD and explained some important details, including the available clone change resolutions and RoboCode action commands. A facsimile of this reference chart is printed in appendix F from page 125 onwards.

5.4 Selection of Subjects

Nearly as important as the experimental design are the experimental subjects. Finding volunteers for experiments is a notoriously difficult task, especially when participants need to have some prior knowledge or skills. To properly test CLONEBOARD, the experimental subjects had to be moderately skilled Eclipse-based Java developers. Given the current tendency of the market for IT personnel, finding skilled Java developers that are actually willing to sacrifice some three ours of their valuable time is hard.

The most likely source of volunteers was found to be the university itself. On Delft's faculty of computer science, Java is the *de facto* programming standard, Eclipse being a popular IDE choice among students. To recruit volunteers, leaflets were spread on the student labs and pinned to notice boards spread over the faculty. Furthermore, invitations were sent to existing mailing groups and informal inquiries were made.

As an incentive to volunteers, two goodies were promised: a free lunch and a nice surprise. Whether the incentive worked or not is hard to say in retrospect, but finally a total number of 7 volunteers for the experiment and 1 volunteer for the pilot were recruited.

5.5 Experiment Setup

In an attempt to exclude as many external variables as possible, the experiment was setup on a virtual machine. On a virtual machine, a predetermined set of hardware components is simulated. By simulating hardware, it was easier to reset the experiment equipment after each run and exclude hardware related influences.

After a process of research, trial and error, VMWare's virtualization solution[6] was selected. VMWare provides a free set of tools that can be used to create virtual machines (VMWare Server) and play the simulations (VMWare Player). The virtualization software and the machine images were installed on a PC in the student lab of the Delft University of Technology's Software Technology department. The host system was a true workhorse, featuring multiple processing cores, multiple gigabytes of internal memory and a 27 inch WUXGA[7] TFT display. The student lab was relatively quiet, with good furniture and lighting.

On the virtual machine, a stripped down version of Windows XP was installed. The only applications made available to the test participants were Eclipse 3.4, RoboCode 1.6.2, Internet Explorer 6 (for access to online Java documentation) and Notepad. An Eclipse Java project was setup with all the required files and settings.

5.6 Pilot

To test run the experiment and locate any problems in its design, a pilot study was performed prior to the actual experiment. As the subject of this pilot, an acquaintance with a degree in software engineering was selected. The initial set of assignments (cf. section 5.3.1) was used in this pilot, along with the pretest and posttest questionnaires described in section 5.2.

Running a pilot of an experiment is generally considered to be a good way to fine tune the experiment's design. In the pilot conducted for this experiment, a number of issues showed up. These insights were used to adjust some parts of the experiment, particularly the programming assignments (cf. section 5.3.2).

Furthermore, some bugs in CLONEBOARD were discovered and fixed and some small adjustments to the tool's parameters were applied:

- **Tokenizer Adjustments.** In the clone comparison routine implemented by CLONEBOARD, code fragments are compared on a token level. The exact definition of what constitutes a token showed to be inadequate during the pilot. String constants were not

[6] See `http://www.vmware.com`

[7] 1920×1200 display resolution.

properly recognized and the initial definition of identifiers[8] proved to be too narrow. Changes were applied, resulting in the tokenization scheme described in section 4.6.2.

- **Enhanced Navigation.** Some minor navigational options were found missing in the initial version of CLONEBOARD. These options were added to conform with their description in section 4.7.1.
- **Resolution Window Timing.** The timing of the change resolution window's pop-up was adjusted, as it was found to be too obtrusive.

The pretest and posttest questionnaires were found to be adequate and no changes were made in these. Furthermore, the experiment's timing proved to be quite accurate and no changes were made on this matter either.

5.7 Experiment Execution

The experiment was conducted on five days in January 2009, both in the morning and the afternoon, testing one subject at a time. By not testing groups of subjects, more attention could be given to each participants, making it easier to create the informal atmosphere necessary to emulate normal development conditions.

Several techniques were used to make the experimental subjects at ease, including the use of a soft, low-pitched voice while addressing subjects and an optimistic and inspiring attitude while briefing the subjects on the experiment. All participants seemed to react well to this approach, all working on the assignments in a relaxed way.

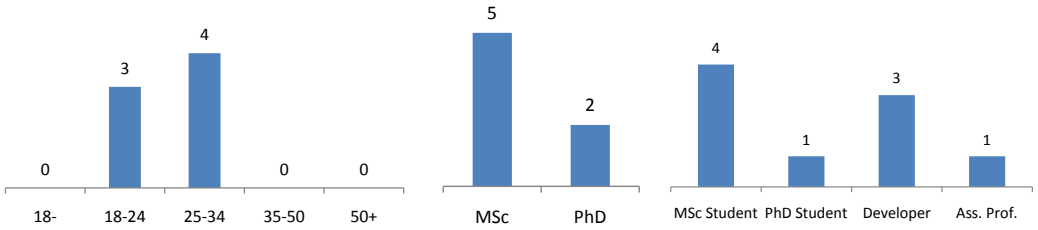
With regular intervals, the test participants were addressed during their programming work to ask them about their work. Drinks were offered and advice was given when subjects were experiencing difficulties with the assignments. Fifteen minutes before the end of the two-hour experiment window, the participants were notified, enabling them to finish their work. Some participants specifically asked for more time to finish their work. Never more than 10 minutes of extra time were granted.

Overall, the experiment went well with no noteworthy problems or irregularities. One bug that had the potential to invalidate the logs generated by CLONEBOARD was fixed halfway the experiment. The bug fix was trivial and did in no way alter the functionality of CLONEBOARD.

5.8 Results

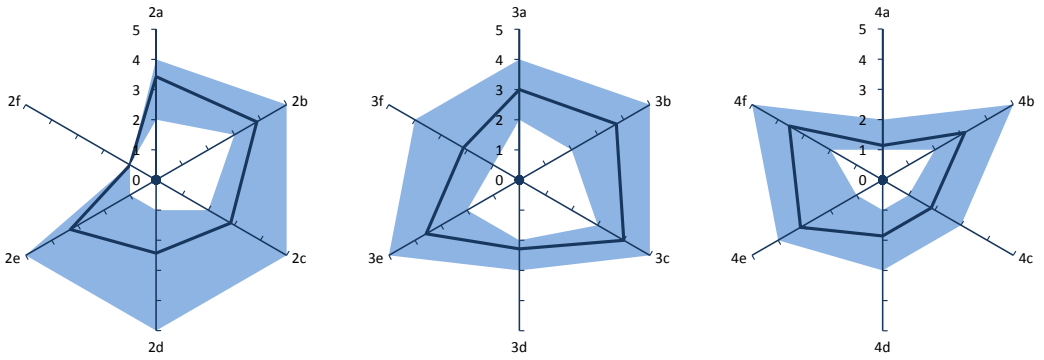
The experiment resulted in three sets of results: the answers from the pretest, the posttest replies and the cloning activity logged by CLONEBOARD. In this section, the data from all three sources are discussed. Analysis on these results is deferred to a discussion in section 5.9, whereas conclusions are reported only in chapter 7.

[8] Before complex identifiers were introduced, only regular identifiers were recognized.



(a) Age distribution (b) Education level (c) Current occupation

Figure 5.2: Personal background of experimental subjects.



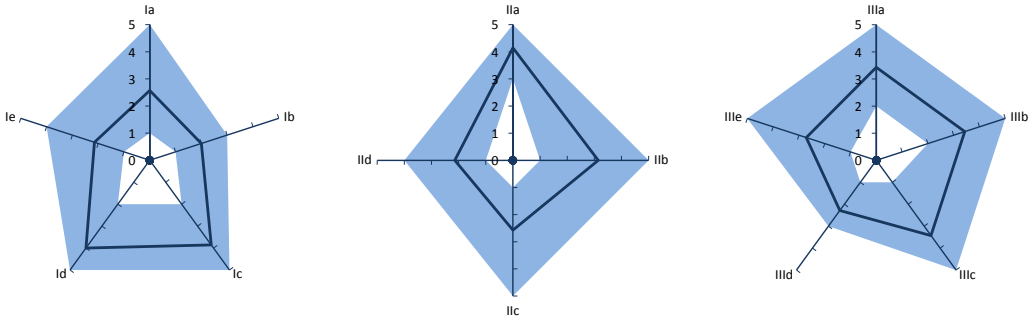
(a) Developer experience (b) Attitude towards code quality (c) Attitude towards cloning

Figure 5.3: Developer profile of experimental subjects.

5.8.1 Subject Profile

Personal Background In the pretest questionnaire, the first question asked to every subject is about their personal background. As all volunteers were recruited on the computer science faculty, the subjects all had either a MSc degree, PhD degree or were very close to one. Furthermore, all participants were male and had ages between 22 and 29. The occupations of the subjects varied, where some reported more than one current occupation. These results are summarized in figure 5.2.

Developer Profile Several questions were asked to get an impression of the subject's development skills. The radar diagrams in figure 5.3 shows which answers have been given on questions 2 to 4 of the pretest. The colored surface in these diagrams indicates the range of answers given, whereas the bold line shows the average answer. Clearly, most subjects consider themselves an averagely experienced (*2a*, average score 3.4) and rather proficient (*2b*, average score 3.9) Java developer. The contexts in which the participants have developed varies greatly (*2c* and *2d*), where some have worked in teams or in a commercial environment and others have not. Furthermore, not all subjects consider Eclipse to be their 'native' development environment, but most do (*2e*, average 3.3, median 4). However, none of the



(a) Assignments experience (b) Development style (c) UI experience

Figure 5.4: Subjects' experiences with CLONEBOARD and the assignments.

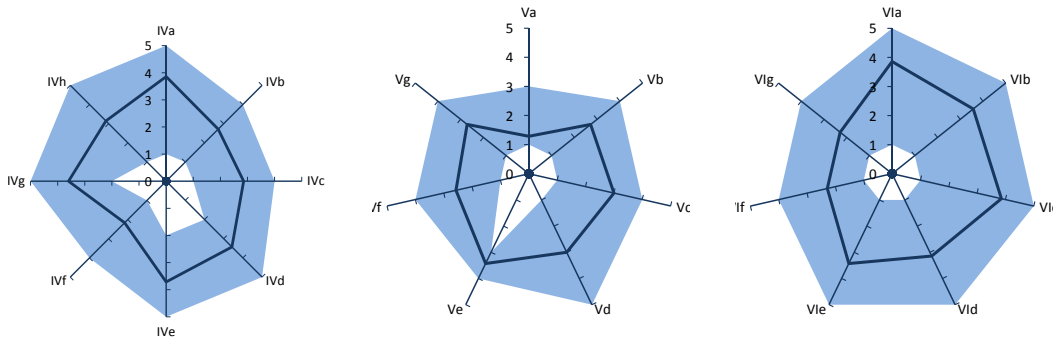
participants claimed prior knowledge of RoboCode (*2f*), which confirms the experimental design's assumptions with this regard.

The subjects' attitude towards code quality is more consistent (cf. figure 5.3b). Writing clean code is valued about equally much as writing functional code (*3a*) and most respondents hesitatingly agree that bugs are often the result of programmer sloppiness (*3b*, average score 3.7). With a score ranging between 3 and 5, all developers endorse a focus on writing good quality code (*3c*, average score 4.0). The statement about commenting behavior was answered by all subjects with a score of 2 or 3, indicating that they generally keep a fair balance between the amount of code and comments they write, with a slight bias towards code (*3d*, average 2.3). With only one rating the statement lower than 3, respondents seemed to somewhat believe that better tools can actually prevent bugs (*3e*, average 3.6). As indicated by the last statement of pretest question 3, only few of the subjects were often using Eclipse's code refactoring facilities (*3f*, average 2.1).

Of all three tested aspects of code development, the respondents are most unanimous in their attitude towards cloning. With the exception of one respondent, all were very familiar with the concept of cloning (*4a*) and agree that copy and pasting is not the best reuse strategy, not in general (*4c*) and not when it comes to crosscutting concerns (*4d*). Copy and paste habits seem to differ quite a bit along the test subjects. Some indicate to copy and paste a lot while programming, whereas others are more reluctant copiers (*4b*, average 3.1, median 2). Most respondents have come across inconsistent clones, but none did so very often (*4e*, average 3.1). Apart from two subjects, all agreed that cloning can lead to bugs (*4f*, average 3.6).

5.8.2 Working with CLONEBOARD

The first three questions of the posttest questionnaire are used to get a basic impression of the subject's experiences with the assignments and CLONEBOARD. These questions help to assert that the assignments were adequate to provoke the desired kind of behavior. The results (cf. figure 5.4) show that the participants generally did not find the assignments



(a) Resolution experience (b) Resolution frequency (c) Resolution value

Figure 5.5: Subjects' experiences with the clone change resolutions.

too hard (*Ia*, average 2.6, median 2). None of the subjects experienced time pressure and actually even tended towards the inverse (*Ib*). Generally, the respondents reported that they found the assignments interesting to do (*Ic* and *Id*). With the exception of one, all subjects were totally satisfied with the level of guidance provided (*Ie*, average 2.1).

With regard to their development style, the respondents all gave rather different answers. Although all subjects confirmed that their programming work reflected their usual habits (*IIa*), some reported to have focused on functional code more than others (*IIb*) and the reported degree of comments written varies greatly, too (*IIc*). Most participants did not indicate to have copied and pasted more than they would normally do, some even reported to have copied slightly less (*IId*).

When asked about the subjects' general experiences with CLONEBOARD, some minor patterns emerged in the resulting answers. Nearly all respondents reported to have encountered CLONEBOARD during their assignments, although not all indicate they did so a lot (*IIIa*). Interesting to note is that the respondents seem to identify the change resolution window with CLONEBOARD, as all of them replied the same to both statements *IIIa* and *IIIb*, the latter of which asks about encounters with the resolution window.

Quite some of the subjects reported to have often quickly dismissed the resolution window by cancelling it (*IIIc*, average 3.4, median 4).[9] Both the CloneView (*IIId*) and CloneBar (*IIIe*) were not rated very high: with the exception of one respondent, all indicated to find little use for these two navigational elements. Comments of some respondents did however show that the clone hyperlinking feature (cf. section 3.3) actually was appreciated.

5.8.3 Resolutions

During the experiment, all subjects were confronted with the clone change resolution window, that would pop-up after a clone had been changed inconsistently. To most subjects

[9] One respondent denied cancelling the window, but rather indicated that he had confirmed it blindly, most of the time. His original rating to *IIIc* was changed from 1 to 5, as canceling and blindly confirming can be considered the same in the context of the question.

Resolution	Usage	Value
Apply changes to all clones	Almost never	Very useful
Ignore changes	Fairly often	Useful
Parameterize clone	Quite often	Very useful
Postpone resolution	Quite often	Fairly useful
Unmark clone	Quite often	Useful
Unmark clone's head	Fairly often	Not so very useful
Unmark clone's tail	Fairly often	Not so very useful

Table 5.1: The experimental subjects' opinions about the 7 clone change resolutions.

(with the exception of one) it was clear most of the times why the window appeared (*IVa*, average 3.9). The window clearly did not always show at convenient moments (*IVb*, average 2.7). For some of the participants, the *before* and *after* views of the changed clone fragment were not sufficiently clear (*IVc*, average 2.9), probably because they were too small at times, as commented by one of the respondents.

Most of the developers more or less agreed that the window showed sufficient information (*IVd*, average 3.4, median 4) and that the order of the resolutions as they were shown was quite logical (*IVe*, average 3.7, median 4). The *Remember resolution* option of the window was not valued well (*IVf*), but it was clear to most participants why the option was not always available (*IVg*, average 3.6).[10] Some of the participants missed change resolutions (*IVg*), most notably more advanced parameterization and resolutions that refactor the affected code.

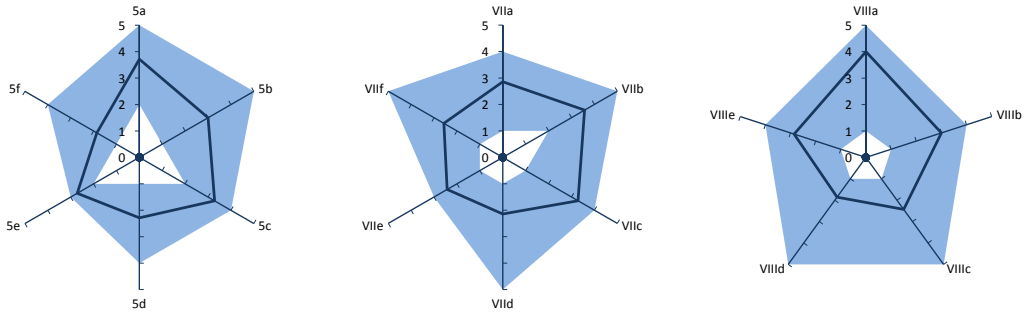
The respondents were asked how often they used each of the resolutions and how they valued each of them (*V* and *VI*). One subject rated all resolutions with a 1, but others answered slightly more differentiated. Table 5.1 summarizes the results, showing what according to the test subjects are the most useful resolutions. One striking, somewhat paradoxal result shown in this table is the fact that apart from one, none of the subjects actually used the *Apply changes to all clones* resolution, but is valued highest of all. One respondent comments about this, stating that the case did provoke cloning, but gave little reason to update clones.

5.8.4 Tool Evaluation

The most important questions of the experiment are question 5 of the pretest and question VII of the posttest. In these questions, CLONEBOARD is compared to the expectations the respondents had of a hypothetical clone management tool with CLONEBOARD's functionality. These questions, together with posttest question VIII measure the dependent variables of the experiment.

In figure 5.6, the radar charts show some differences between the participants original expectations and their perception of CLONEBOARD. These differences can be observed

[10] Two of the respondents actually did not rate these statements as they had not noticed the option at all.



(a) Expectations (b) Perception (c) UI problems

Figure 5.6: Subjects' evaluation of CLONEBOARDas a clone management tool.

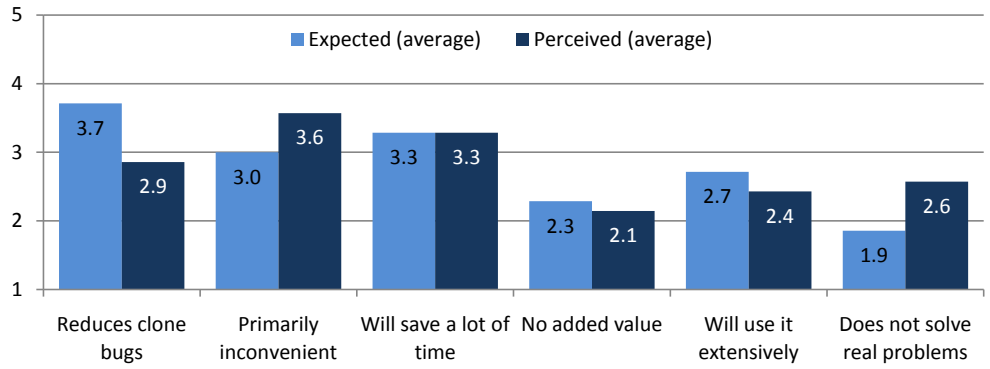
better in figure 5.7, in which the averages and medians of the ratings are shown. Although participants are still not very convinced a clone management tool would save them time (*5c* and *VIIc*), CLONEBOARD apparently does offer slightly more added value than the respondents expected (*5d* and *VIId*).

The test subjects seem to be somewhat disappointed with CLONEBOARD, in the sense that they had expected that it would help reduce clone related bugs better (*5a* and *VIIa*). This outcome might be slightly colored by two of the subjects, who changed their opinions rather radically. The higher than average medians of the ratings illustrate this fact. Furthermore, the respondents do not seem so very convinced anymore that CLONEBOARD as a clone management tool might solve any real problems (*5f* and *VIIf*). This insight is shared rather broadly among the test subjects. Only one respondent indicates that CLONEBOARD is more likely to solve problems than he had expected.

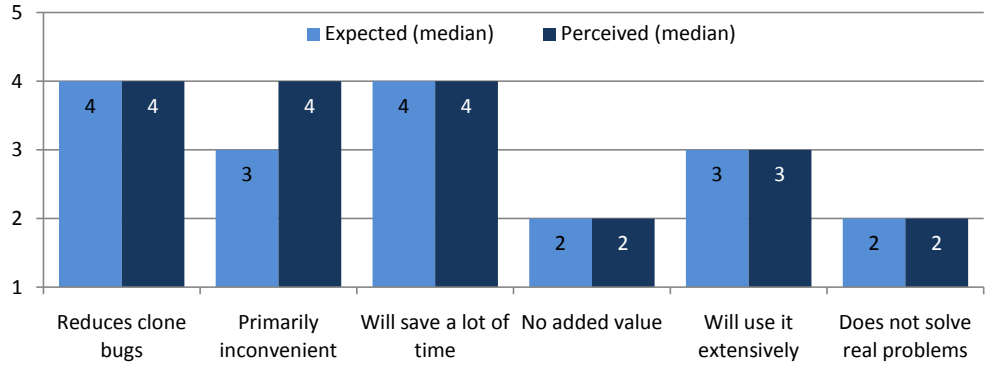
Apparently, subjects are slightly less convinced of CLONEBOARD's ability to save them time (*5a* and *VIIa*), but are more confident it will help them solve real problems (*5b* and *VIIb*). CLONEBOARD's ability to significantly help reduce clone related bugs is rated slightly lower (*5c* and *VIIc*), with the median rate still being 4, indicating a number of negative outliers.

Striking is the rather large difference between the expected and perceived inconvenience of CLONEBOARD(*5b* and *VIIb*). Whereas participants where rather mild about this in the pretest, after using CLONEBOARD, their attitude has changed significantly. This perceived inconvenience probably let them to indicate a lower chance of actually using the tool in practice (*5e* and *VIIe*).

CLONEBOARD's user-interface elements do not seem to be the reason of the subjects' disappointment. In general, participants found the UI easy to use (*VIIIa*, average 4.0, median 5) and are fairly neutral when it comes to assessing their ability to get used to CLONEBOARD as part of their IDE (*VIIIb*). The respondents reported some errors (*VIIIc*, average 2.4, median 2), but these apparently did not hinder CLONEBOARD's functionality significantly (*VIIId*, average 1.9, median 1).



(a) Averages of ratings



(b) Medians of ratings

Figure 5.7: Expectations for and perceptions of CLONEBOARD.

When asked whether CLONEBOARD would need a better user-interface, opinions seem to differ. Some are more negative than others, but on average, subjects are fairly neutral about this (*VIIIe*, average 2.9, median 3). One subject commented that the user-interface could have been less intrusive.

5.8.5 Experiment Rating

In the final matrix question of the posttest questionnaire, respondents were given the opportunity to give feedback on the way the experiment was conducted. As can be seen in figure 5.8, ratings in general were high.

Apparently, the subjects found the experiment 'fun' to do and had no problems deciphering the assignments and the associated documentation. The selected case (i.e. RoboCode) was not found suitable by all participants. Two respondents rated 'suitability' with a 7 and a 5 (out of 9) respectively.

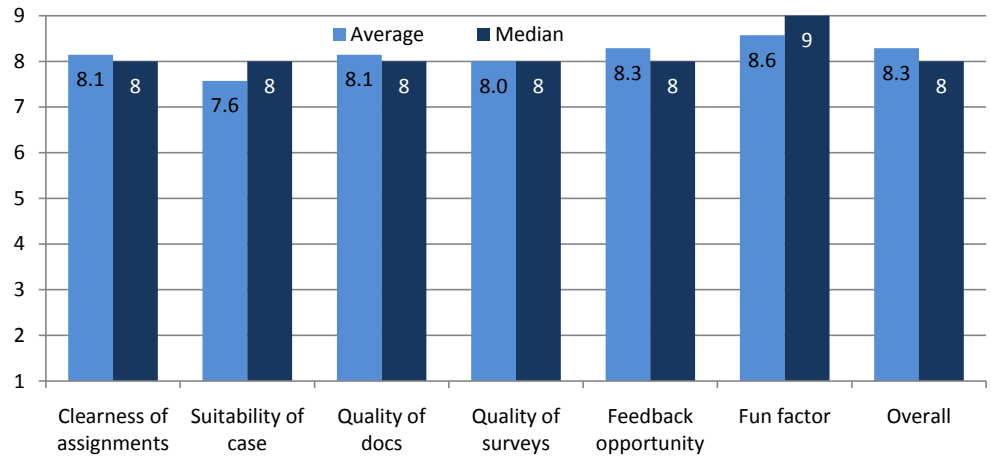
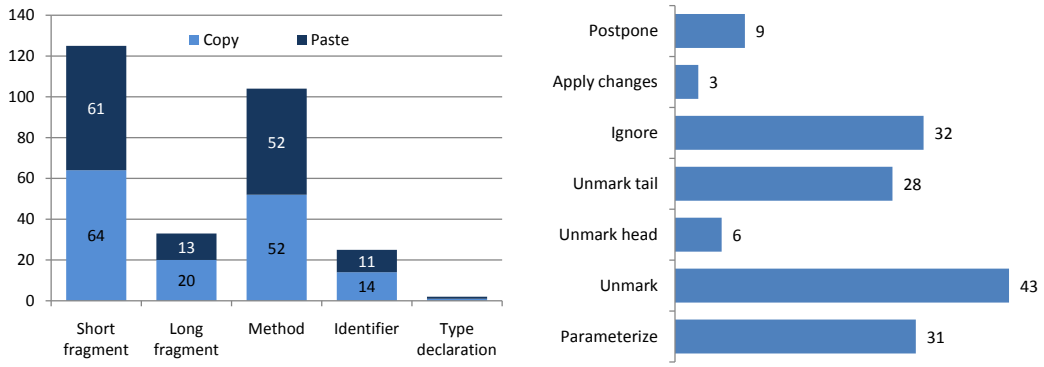


Figure 5.8: The subjects' rating of the experiment



(a) Fragment types copied and pasted

(b) Resolutions applied

Figure 5.9: CLONEBOARD usage statistics extracted from log files.

5.8.6 Log Data

Apart from the two questionnaires, a third source of information was available in the form of CLONEBOARD's XML logs. These logs contain information about non-trivial code fragment copy and paste operations, applied resolutions and possible error conditions. The logs have been analyzed to extract some basic information about the frequency different fragment types were copied and pasted (cf. figure 5.9a) and the clone change resolutions that were used most (cf. figure 5.9).

5.9 Analysis

The purpose of the conducted experiment was to investigate three variables relating to CLONEBOARD: adequacy, usability and effectiveness (cf. section 5.1.1). In the quesionnaires of the pretest and posttest, specific questions were asked to measure these variables. In the following paragraphs, the results of these and other questions will be analyzed to determine to what extent CLONEBOARD is adequate in managing clone changes, whether it is a user friendly tool and in how far it is an effective solution against clone related bugs. Furthermore, the usefulness of the implemented clone change resolution strategies will be considered, based on respondents' input.

5.9.1 Adequacy

To find out whether CLONEBOARD as a tool is sufficiently adequate for developers to use as a clone management solution in their daily work, a number of specific statements was added to the pretest and posttest questionnaires. Specifically, three pairs of statements in pretest question 5 and posttest question VII gauge CLONEBOARD's adequacy, as well as one statement in posttest question VIII:

- ***5c & VIIc.*** CLONEBOARD will help me save a lot of time.
- ***5d & VIId.*** I don't see the added value of CLONEBOARD.
- ***5e & VIIe.*** I expect that I would be making use of CLONEBOARD quite extensively.
- ***VIIIb.*** I will be able to get used to using CLONEBOARD in everyday coding.

The results of the experiment show that the respondents do see some added value for a tool like CLONEBOARD, actually even more than they initially expected. Although the difference is rather insignificant, it at least does show that CLONEBOARD lives up to the respondents expectations with regard to its potential to add value to the development process. Actually, only one respondent (#1) was heavily disappointed: he had great expectations, but apparently perceived the reverse. Excluding this subject's input, the average rating for CLONEBOARD's added value becomes 1.7[11], showing a more significant improvement.

Before the experiment, the test subjects were not convinced of a clone management tool's ability to save them time, and this opinion did not greatly change. Some respondents were more optimistic than others, but apparently CLONEBOARD either is not able to save developers time or the experiment was too short to accurately assess time savings. Considering that clone management tools will typically save time mainly by reducing inconsistencies and helping to fix bugs in duplicated code fragments more easily, it is likely that the experiment actually was too short to observe any such positive effects.

Considering these results and the fact that respondents were not overly optimistic about their ability to get used to CLONEBOARD in their daily practice (or a similar tool, for that matter), it seems fair to doubt CLONEBOARD's adequacy. Although it clearly does add

[11] Please bear in mind that statements *5d* and *VIId* were formulated negatively, so that a lower rating actually means a higher appreciation.

value, its adequacy as a tool has not been shown. A longitudinal study would be required to draw real conclusions about CLONEBOARD's adequacy on the long term.

5.9.2 Usability

Nearly all respondents either formally or informally reported that they found CLONEBOARD slightly too obtrusive. Apart from being a bit annoying, CLONEBOARD was reported to be easy to use. The answers to statement *VIIIa* clearly illustrate this. Only half of the respondents is slightly convinced that CLONEBOARD would need a better user-interface to be useful (statement *VIIIf*).

The clone change resolution window seems to be the main cause of irritation. Some of the respondents reported to have mainly dismissed the resolution window by pressing the *cancel* button or hitting *escape* (*IIIc*). The window did not always pop-up at convenient moments (*IVb*), despite all technical attempts made to choose the most appropriate moment. Comments given by some of the respondents learned that improving the timing will probably not help, as it is just the concept of a pop-up window that annoys developers: a less obtrusive query mode (e.g. a warning icon or message in a sidebar, cf. section 4.8.3) would probably lead to less disturbance.

About the resolution window itself, respondents were quite positive. The dialog window shows sufficient information to make its purpose clear (*IVa* and *IVd*) and the presentation order of the resolutions was found logical to most respondents (*IVe*).

Other parts of the CLONEBOARD user-interface (i.e. the CloneView and the CloneBar) were not found to be especially useful (*IIId* and *IIIe*). The size of the selected case and the time developers were asked to work on it probably did not require the use of such controls to navigate the clone model.

In short, the experimental subjects found CLONEBOARD to be usable, but too obtrusive. This outcome underlines the premise that a clone management tool should be as unobtrusive as possible. The barrier discussed in section 2.2.2 that should make developers more clone-aware apparently is a bit to high in CLONEBOARD. On the other hands, the techniques used to aid in resolving clone changes easily, such as the heuristic sorting of strategies and the smart highlighting of changes, were well appreciated by respondents.

5.9.3 Effectiveness

Measuring the effectiveness on a tool that is expected to be mainly beneficial when used over longer period of time is difficult within the scope of a short experiment. To nevertheless get an impression of CLONEBOARD's effectiveness, subject's were asked about their opinion: will CLONEBOARD solve problems (*5f* and *VIIf*)? In the pretest, the respondents were quite optimistic about this. Two hours with CLONEBOARD later, however, the subjects were less so. Although they still believed CLONEBOARD could solve real problems, their conviction had diminished considerably.

A similar but more dramatic decline of optimism is shown by the answers subjects gave when asked about the potential of CLONEBOARD to reduce clone related bugs. Expecta-

tions about this were rather high (*5a*), but these got crumbled during the experiment. Two respondents in particular radically changed their mind from being very optimistic to rather clowdy about the potential of CLONEBOARD. Apparently, more is needed to solve clone related issues according to the test participants.

So, is CLONEBOARD effective? Although it was hard for the experimental subjects to judge, the initial results are not very promising. The developers were rather neutral about CLONEBOARD's effectiveness, suggesting that a more elaborate evaluation would be required for them to give a conclusive answer. This impression is supported by one of the subjects, who actually suggested it quite literally in his comments.

5.9.4 Usefulness of the Resolution Mechanism

Although the primary objective of the experiment was to assess CLONEBOARD's value as a clone management tool, the questionnaire results that relate to the resolution strategies used were such that they justify further analysis.

It was interesting to see that the change resolutions that were used less were valued most (cf. table 5.1). Most notably, the *Apply changes to all clones* resolution (cf. section 2.2.3.2) was applied only three times, but rated highest. Quite in contrast with this, another highly valued resolution, *Parameterize clone*, was actually used very often. The other resolution strategies were valued less, indicating that the primary interest of developers in a clone management tool is to be able to keep similar code fragments in sync and patch them whenever a bug requires so.

This observation adds to the impression that a clone management tool should primarily guard and enforce clone integrity, but should not bother developers with the details of what exactly defines a clone and what parts should be included. The experimental subjects showed to be quite willing to define parameterized clones, but often escaped the hassle of other administrative tasks by dismissing CLONEBOARD's queries or removing clone markers altogether to prevent being disturbed. One of the participants admitted to this frankly: "I stopped using ctrl-c to prevent the resolution window from popping up."

5.10 Threats to Validity

Mathematicians are often skeptical about the value of experimenting. "Experiments don't prove a thing", they often exclaim [60]. And this is true. Theories can only be falsified by means of experiments. Absolute proof of a hypothesis is never possible through means of experimentation. However, this does not mean experimentation is a waste of time. As long as one is aware of the conditions under which an experiment has been conducted, it can be quite safe to deduce causal relations and generalize findings.

In this section, the validity of the observations resulting from the experiment is discussed. Traditionally, two different sides of 'validity' are considered. First of all, the cause-effect inferences made during the analysis will have to be tested for validity. This kind of validity is called 'internal validity'. The external validity of an experiment relates to its gen-

eralizability. It is only when an experiment's results can be justly extrapolated to a larger population that they make sense.

5.10.1 Internal Validity

Analysis of the experiment's outcomes relies on the assumption that the only factor influencing the dependent variables is CLONEBOARD itself. However, several factors relating to the subjects and the circumstances may have interfered. First of all, some subjects may have felt emotional pressure to answer positively. The one-to-one setting of the experiment may have incited this behavior. Attempts were made, however, to make clear to all subjects that they did not have to please anybody, as only sincere answers were of value.

All subjects were selected based on a presupposed experience in Java development. This assumption was never tested as such, but was confirmed by the subjects' own responses. Furthermore, inspections of their work did in fact indicate all participants were able. Still, the group of subjects was very diverse and there backgrounds undoubtedly had some influence on their answers. During analysis of the data, no obvious relations were found, however.

A rather real threat to internal validity is in the simulated circumstances. The assignments that were given to the participants may have induced certain types of behavior more than others. More importantly, the selected case may have affected the mood of the participants, indirectly affecting their judgment. Control questions have been added to the posttest to test for these effects. The selected case and assignments were found suitable by the subjects, and no negative emotions were observed.

The duration of the experiment may have influenced the validity of the results, too. A duration of two hours was considered somewhat short by some of the participants, especially as it is hard to appreciate long term benefits of clone management software in such a short time. To counter this effect, participants were deliberately asked to try and consider how the software would work for them in practice.

Finally, all external conditions of the experiment were kept as constant as possible. All participants were handed out the same documentation, used the same work station in the same lab, worked on exactly the same case and received the same verbal introduction. The subjects were made at ease to create an informal atmosphere and were not addressed too informally during the experiment, to prevent the interaction from becoming too jovial.

5.10.2 External Validity

Generalizability of the experiment's results depends mainly on three major aspects: representativeness of the subjects, suitability of the selected case and the degree of similarity the programming assignments bear to real-world development tasks.

Although all subjects were academics, their background were different to such a degree, that it seems safe to assume they represent average developers quite accurately. Some of the subjects had significant commercial development experience, whereas others had a more

theoretical background. The limited age range of the subjects, however, may have hampered the experiment's validity.

The case that was selected, was chosen to be easily comprehensible. A clear disadvantage of this choice is that it also meant the case would be not so very complex. It is safe to assume most real-life systems are more complex than the selected one. CLONEBOARD might prove to be more useful in a more complex system. However, all participants were asked to what extent they expected the tool to be of use in their daily practice. Given the different background of the subjects, this formulation should have resulted in more generalizable answers.

One of the subjects remarked that the programming assignments were more focused on inducing clone creation than on clone modification. Given the rather limited time frame, it is hard to simulate the effect of modifying clones other developers created. This shortcoming may have had a significant influence on the outcomes of the experiment. However, log traces show that participants did in fact face sufficient clone change resolution situations to be able to judge CLONEBOARD's value.

5.11 Summary

In this chapter, the whole process from design to evaluation has been detailed for the experiment that was conducted. A one-group pretest-posttest preexerimental design was chosen as it best fitted the requirements of the experiment. Although this is a valid experiment design, its results are generally not considered to be truly scientific. It was argued, however, that given the goals of this project, truly scientific results were less important than getting a good impression of the practical use of the techniques proposed.

The experiment was conducted with seven subjects and an additional subject was involved in a pilot. Each subject was tested individually in a 150 minute session. Although all subjects were academics, their background were quite dissimilar and their ages were distributed rather evenly.

After the experiment's results had been analyzed, a number of observations was made. First of all, CLONEBOARD's adequacy as a clone management tool is questionable, although subjects clearly did indicate some added value. The tool's user-interface was well designed, but its resolution window was found a too obtrusive way of querying. Respondents were not clear about CLONEBOARD's effectiveness in countering the negative effects of cloning. According to some, the experiment should have been longer to determine the plug-in's true effectiveness.

Only a small number of the clone change resolution strategies was really valued by the test subjects. Apparently, the ability to cascade changes made to one clone and the templating made possible by parameterizing clones were valued most. The other resolution strategies are to be considered superfluous mainly and should preferably be handled behind the curtains.

Chapter 6

Related Work

The clone management question has been undervalued for quite some time. A lot of effort has been put in developing static clone detectors that analyze source code and mark all similar code fragments [4, 8, 20, 34]. It is only more recently that effective ways to actually manage code cloning have been researched [2, 44, 22, 62].

6.1 Linked Editing

An interesting technique to apply changes to a set of clones was introduced by Toomin *et al.* [61]. They show that the concept of *Linked Editing* can prove useful in simultaneously updating clone fragments. By linking two code fragments together, they can be edited simultaneously by means of a visual editor. Part of what Toomim *et al.* have established is included in CLONEBOARD as well: using similar techniques, a change to one clone can be cascaded to the rest of the clone set.

The version of the linked editing prototype Toomim *et al.* have reported on does not support automatic linking of cloned fragments: users will have to select the fragments manually to start editing them. An interesting finding of the researchers is that in an experiment they conducted it showed that linked editing can save a lot of time when compared to the more traditional approach of functional abstraction to refactor redundant duplicates.

6.2 Linking Copied Identifiers

Several others have implemented plug-ins that integrate clone management into a development environment. Jablonski and Hou, for instance, have developed a framework that captures copy and paste activity and uses this information to dynamically track clones [30]. Their software, dubbed *CnP*, automatically links identifiers, so that rename operations can be guaranteed to be performed consistently. In essence, this approach is very similar to CLONEBOARD's. The main difference is in the way changes are handled: whereas *CnP*

only assists in rename operations, CLONEBOARD tries to approach changes in a broader sense.

What is interesting about Jablonski's approach is the way in which she analyzes copied clone fragments. Instead of analyzing the raw text of a code fragment, as CLONEBOARD does, Jablonski uses the abstract syntax tree representation that underlies it. Using this model, the scope of identifiers can be determined and references to the same entity can be tracked. For the purpose of enforcing consistent renamings, this probably is the best approach, but for the purpose CLONEBOARD serves, this would probably be overkill.

6.3 Tracking Clones

Duala-Ekoko and Robillard bring several clone management techniques together in a tool called *CloneTracker* [23]. This Eclipse plug-in maintains a model of all clones in a source base. The data for this model is gathered using a third-party clone detector. The way in which *ClontTracker* visualizes clones and allows navigation is similar to the techniques employed by CLONEBOARD. When it comes to handling clone changes, however, *CloneTracker* rather resorts to using linked editing, whereas CLONEBOARD introduces the concept of automatic change resolutions.

The plug-in developed by Duala-Ekoko and Robillard is rather complete and offers a broad range of funtionality. The way in which clones are tracked is especially interesting. Instead of recording the offset and length of each clone fragment in its source file, specially designed clone region descriptors are used. These descriptors use the syntactical context of a clone fragment as an anchor and are as such more robust against unsupervised file changes. In contrast, CLONEBOARDfully relies on Eclipse's marker functionality and as such is not robust against source changes made outside of the Eclipse environment.

6.4 Dynamic Clone Detection

Two tools that are very similar to CLONEBOARD in terms of technology are *Clonescape* by Chiu and Hirtle [17] and *CPC* by Weckerle [63]. Both tools are Eclipse plug-ins that monitor clipboard activity to infer clone relations, similar to the way CLONEBOARD does. *Clonescape*, however, focuses more on providing clone navigation tools. *CPC*, on the other hand was implemented to be a framework for others to base clone management technology on. As such it is technologically superior to CLONEBOARD, but it does not go beyond the point of notifying users of possible clone inconsistencies. No attempts are made by *CPC* to resolve these issues automatically.

It is interesting to see that both *Clonescape* and *CPC* rely on Eclipse's marker model to store clone information. Weckerle reports on similar problems with implementing clone capturing as were experienced during the implementation of CLONEBOARD. Whereas CLONEBOARD is only capable of adding clone marks automatically, in *Clonescape* the option was added to manually register clones. This option may prove useful to register preexisting clones, but introduces the risk of registering spurious clones.

The work of Weckerle seems particularly solid. Both the resulting *CPC* framework and the research he conducted are of admirable quality. The *CPC* tool was tested on a number of developers, and data about their cloning behavior were recorded for a period totaling two months. Weckerle has analyzed these data and was able to generate some quite interesting statistics about programmer's cloning behavior.

6.5 Other Work

Among the many other research projects that have been conducted in the field of code cloning, there are some interesting ones worth mentioning. Juergens *et al.* for instance have explored ways to relate inconsistently changed clones to potential bugs [33]. Recent research by Krinke shows that cloned code actually tends to be more stable than other code: clones are less likely to be edited once they have been created [43].

As current clone detectors tend to produce a lot of output, often containing a significant amount of spurious clones, the techniques proposed by Zhang *et al.* to filter and visualize clone data is worth mentioning [67]. The visualization techniques used are actually not quite unlike those used by CLONEBOARD. Finally, the visualization techniques used by Adar and Kim to help explore clone families and the way they have evolved can be considered quite remarkable [1]. Using various types of graphs, these researchers managed to give insight in to a whole new concept which they call *clone genealogy*.

Chapter 7

Conclusions and Future Work

In the time it took to complete this thesis project, quite a lot of work has been done. More than 150 publications were examined and over 7,500 lines of code were written to implement the more than 100 classes CLONEBOARD consists of. Furthermore, the 20 hours of experimentation lead to more than 600 data points and nearly a megabyte of XML logs. But what for?

7.1 Conclusions

At the start of this thesis project, a number of research questions have been formulated (cf. section 1.2.1). These questions formed the basis for all further actions taken. Looking back on the implemented tool, CLONEBOARD and the experiment that has been conducted with it, is it possible to formulate answers to these questions? In the following paragraphs, answers to each of the research questions (as shown below) will be proposed.

- **Question #1.** Can the copy and paste replacements described by Mann be realistically implemented in a programmer's development process and coding environment?
- **Question #2.** Are developers willing to alter existing copy and paste habits to help contain code clones?
- **Question #3.** In what ways can the relations established by using Mann's operations be used to enforce consistent editing of clones?
- **Question #4.** Will Mann's operations help reduce cloning related problems?

7.1.1 Implementing the Mann Operations

The first question that was posed relates to the feasibility of Mann's proposition to replace existing copy and paste operations in development tools with a set of cloning operations. In an attempt to prevent disappointments, Mann's proposals were interpreted rather flexibly. It was suspected early on that there would be little support for a tool that replaces two of the

most solidly anchored shortcuts in the development world. Instead, an alternative way of implementing the operations proposed by Mann was chosen.

By inferring the Mann operations rather than implementing them directly, developers will not have to be asked to adapt to new habits. Instead, their current habits could be translated transparently to the desired new behavior. In CLONEBOARD, this was achieved by assuming the strictest relation by default and querying the developer every time the relation might need to be relaxed.

The results of the experiment conducted to test CLONEBOARD show that this interactive approach was not an undivided success. Although the participating developers did rather unanimously agree that CLONEBOARD in fact does add value to the development process, the tool was found to interfere too much with their primary activities. The experiment clearly showed that a clone management tool will have to be more like a watch-dog than a traffic policeman, only barking when something is wrong and not standing in the way when one is in a hurry.

So, can Mann's operations be realistically implemented in the development process? Yes, but a strong emphasis should be on usability, requiring as little of a developer's attention as possible.

7.1.2 Change Developer Habits to Better Contain Clones

Tinkering with development tools is a delicate matter. Changing their daily work routine is not an undertaking to think to lightly of. Or is that just a overly negative presumption? What can be concluded about the second research questions? Are developers willing to alter their clipboard habits to help contain code clones?

Apart from the fact that CLONEBOARD did not always interfere at convenient moments, most of the respondents were actually rather optimistic about the concept of clone change resolutions. It was clear to most that automated support in the process of patching clones and reusing existing code as templates for new code can save them time. This seems to suggest that developers do seem to be willing to alter their copy and paste habits when it saves them time. Important is, however, that they will not have to invest more time to feed a tool with data than the time the thing will help them save.

In the experiment's pretest, subjects were asked about their expectations for a clone management tool. Their answers showed that they did believe such a tool would have them reduce the amount of clone related bugs. Furthermore, respondents did not expect to leave such a tool unused. Apparently, CLONEBOARD's user-interface was not implemented sufficiently well to provide the level of user-friendliness required to have developers accept it in their development environment. This is indicated by the fact that after they had worked with CLONEBOARD for two hours, the mild reticence respondents showed about the possible inconvenience of a clone management tool changed into a more pronounced rejection of its usability.

To answer the question about developer willingness to change habits to aid in clone containment: the developers that participated in the experiment mostly recognized the cloning

problem and seemed willing to change some of their copying habits to lessen the cloning problem. So, yes, developers are willing to change their habits, but only if the changes will not interfere with their productivity too much. CLONEBOARD did not succeed in keeping a sufficiently low profile to be accepted by developers: it was found too obtrusive.

7.1.3 Mann Operations Used to Enforce Clone Consistency

In what ways can the relations Mann's operations establish be used to enforce consistent editing of clones? In this study, the resolution mechanism has been proposed to offer developers several ways to enforce clone consistency.

Apparently, respondents highly appreciated the possibility to forward changes made in one clone to all other instances. The possibility to mark wildcards (i.e. parameters) in clones so that they could be used as templates more easily enhanced this appreciation. These observations show that the clone relations established using Mann's operations can be enforced by using the proposed change resolution mechanism.

Alternative ways to force consistent editing of clones, such as simultaneous editing, do exist. However, *post factum* restoral of clone inconsistencies shows to be a good and less invasive alternative. Other than with simultaneous editing techniques, a developer will not have to consent to anything before changing a clone. Only when the changes made a clone family inconsistent, the developer needs to give its approval to some sort of resolution.

In short, several ways to enforce consistent clone editing by using the relations established by Mann's operations do exist. One particular way to use them was designed for and implemented in CLONEBOARD: clone change resolutions. This mechanism was found to be useful and seems to be less invasive than alternative solutions.

7.1.4 Effectiveness of Mann Operations in Reducing Clone-related Problems

One of the most important questions of this research project is of course whether Mann's operations can help reduce problems related to cloning. Based on the respondents' assessment of this matter, CLONEBOARD's implementation of the operations will only be of limited use. Although the respondents were not pessimistic about its ability to solve real problems, there is still some more that needs to be done to make CLONEBOARD as effective as it should be. Respondents indicated that the current set of resolutions did not always suite their needs. Some more advanced resolution strategies were proposed.

As for Mann's operations: the fact that CLONEBOARD did not seem to be very effective does not mean that Mann's operations are of no use. The opposite might well be true. Usability aspects play a very important role in the successful implementation of Mann's operations. As CLONEBOARD was found to be insufficiently usable, this fact may have hindered its effectiveness. It is therefore impossible to conclude than Mann's operations are ineffective.

All in all, insufficient evidence was found to give a conclusive answer to the question whether Mann's operations will help to reduce cloning related problems. More research is

necessary for this, but usability issues need to be resolved first, as these apparently influence effectiveness considerably.

7.2 Contributions

This thesis project has shown that clone management tools can actually add value to the development process, but only if their usability is carefully considered and designed to be as unobtrusive as possible. The work in this thesis makes the following contributions to the field of clone management research:

- **CLONEBOARD.** The plug-in introduced in this thesis is a major contribution. Its design can be used as a starting point for other tools and its underlying concepts can be reused in other software.
- **Dynamic Clone Tracking.** Although dynamic clone tracking by means of clipboard activity monitoring has been shown by others before, this thesis has added a more theoretic backbone to the technology by introducing the concepts of atomic clone changes and clone change transactions.
- **Dynamic Change Resolution.** When it comes to its unique dynamic change resolution approach, CLONEBOARD goes beyond other similar tools. The concept of a clone change resolution adds a valuable alternative to the currently rather limited set of clone containment options of removal and refactoring. Actually, the concept of change resolution can be seen as an abstraction of these conventional solutions.
- **Resolution Strategies.** CLONEBOARD adds three new clone management strategies in the form of parameterization, change forwarding and selective adaptation of clone boundaries.
- **Experiment.** The experiment conducted with CLONEBOARD contributes some interesting insights about the perception of clone management tools. The outcomes of this result clearly shows the importance of usability factors for successful development tool introductions.

7.3 Future Work

More work is needed to find out which strategies are most fruitful in the struggle for clone containment. Based on the research done on CLONEBOARD, a number of recommendations for future work can be made:

7.3.1 Longitudinal Study

To properly assess the value and effectiveness of a clone management solution, it should be tested in the context of a longitudinal study. In the field of clone management, some longitudinal studies have been performed, but mostly in retrospect by analyzing code versioning

systems. Such retrospective studies do not lend themselves for tool evaluation as they lack the so much needed control groups and can not be repeated easily by other researchers.

A longitudinal study with CLONEBOARD may learn whether it has long-term benefits. The two hour experiment conducted with CLONEBOARD was not sufficiently long to allow participants to get used to the tool and learn how to best put it to their use.

7.3.2 Further Development of CLONEBOARD

Although CLONEBOARD did not prove to be a golden bullet, it does represent an interesting new way to generalize clone consistency management. It surely seems worth the effort to further develop CLONEBOARD. Leading the list of improvements should be a less intrusive resolution querying mode. Possibly, problem markers can be used in combination with Eclipse's *quick fix* framework to offer developers the chance to reconcile clone inconsistencies at a time it better suits them.

Parts of CLONEBOARD may have to be rewritten to make use of some of Eclipse's more advanced features. Project natures, for instance, might be a good way to hook clone management into Eclipse projects. Natures have not been considered as an implementation option, but might be an interesting starting point for further development. With project natures, so called builders can be attached to a project. Such builders are informed about every change in source code, so that they can update a particular code representation model, in this case a clone model.

Extending the set of change resolutions CLONEBOARD supports is another desirable direction for further development. As suggested by one of the experimental subjects, including certain refactoring options in the resolution set might prove very useful. Furthermore, the clone parameterization resolution needs to be implemented such that it can handle more complex cases.

7.3.3 Study Clone Change Patterns

In order to better predict the type of relation that should be enforced on a certain clone set, it is necessary to gain a deeper understanding of clone change patterns. Quite some general work has been done in this field [36, 39, 43, 48], but only by compiling a full catalog of change patterns will it become possible to create clone management software that is able to infer the desired relations with reasonable success.

Techniques from the artificial intelligence field may prove useful in this matter, too. Association rule mining algorithms may be used to find out what kind of clone relations fit best for specific types of clones.

7.3.4 Implement Mann's Operations

Instead of the inference mechanism used by CLONEBOARD, Mann's operations could have been implemented directly. By building a prototype development environment that features

Mann's operations (and appropriate keyboard shortcut to facilitate frequent use), experiments can be done to see whether these operations are actually usable in their original form. Possibly, the operations may need to be altered or new operations may be added to the set, to make them usable to developers.

Bibliography

[1] Eytan Adar and Miryung Kim. Softguess: Visualization and exploration of code clones in context. In *I*CSE '07: Proceedings of the 29th International Conference on Software Engineering, pages 762–766, Washington, DC, USA, 2007. IEEE Computer Society. – *Cited on p. 81*

[2] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. How clones are maintained: An empirical study. In *C*SMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering, pages 81–90, Washington, DC, USA, 2007. IEEE Computer Society. – *Cited on p. 79*

[3] E.R. Babbie. *T*he practice of social research. Wadsworth Belmont, 11th edition, 2007. – *Cited on pp. 57 and 60*

[4] Brenda S. Baker. A program for identifying duplicated code. *C*omputing Science and Statistics, 24:49–57, 1992. – *Cited on pp. 6, 12, 15, 21, and 79*

[5] B.S. Baker. On finding duplication and near-duplication in large software systems. In *P*roceedings of 2nd Working Conference on Reverse Engineering, pages 86–95, Los Alamitos, CA, USA, 1995. IEEE Computer Society. – *Cited on pp. 1 and 3*

[6] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Measuring clone based reengineering opportunities. In *M*ETRICS '99: Proceedings of the 6th International Symposium on Software Metrics, page 292, Washington, DC, USA, 1999. IEEE Computer Society. – *Cited on p. 2*

[7] Hamid Abdul Basit and Stan Jarzabek. Efficient token based clone detection with flexible tokenization. In *E*SEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pages 513–516, New York, NY, USA, 2007. ACM. – *Cited on pp. 2 and 95*

[8] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In A. Yahin, editor, *P*roc. International Conference on Software Maintenance, pages 368–377, 1998. – *Cited on pp. 2, 6, and 79*

[9] B.B. Beck. *A*nimal Tool Behavior: The Use and Manufacture of Tools by Animals. New York: Garland STPM Press, 1980. – *Cited on p. 1*

[10] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *I*EEE Trans. Software Eng., 33(9):577–591, 2007. – *Cited on pp. 2 and 3*

[11] Barry W. Boehm. Software risk management: Principles and practices. *I*EEE Softw., 8(1):32–41, 1991. – *Cited on p. 21*

[12] J. Brandt, P.J. Guo, J. Lewenstein, and S.R. Klemmer. Opportunistic programming: how rapid ideation and prototyping occur in practice. In *P*roceedings of the 4th international workshop on End-user software engineering, pages 1–5. ACM New York, NY, USA, 2008. – *Cited on p. 5*

[13] F. P. Brooks Jr. Grasping reality through illusion—interactive graphics serving science. In *C*HI '88: Proceedings of the SIGCHI conference on Human factors in computing systems, pages 1–11, New York, NY, USA, 1988. ACM. – *Cited on p. 57*

[14] F.P. Brooks Jr. *T*he Mythical Man-Month: Essays on Software Engineering. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1978. – *Cited on p. 57*

[15] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe. On the use of clone detection for identifying crosscutting concern code. *S*oftware Engineering, IEEE Transactions on, 31(10):804–818, 2005. – *Cited on p. 6*

[16] D.T. Campbell, J.C. Stanley, and N.L. Gage. *E*xperimental and quasi-experimental designs for research. Rand McNally Chicago, 1963. – *Cited on p. 57*

[17] Andy Chiu and David Hirtle. Beyond clone detection. Technical report, University of Waterloo, 2007. – *Cited on p. 80*

[18] E. Clayberg and D. Rubel. *E*clipse: Building Commercial-Quality Plug-Ins. Addison-Wesley, 2004. – *Cited on pp. 22 and 50*

[19] James R. Cordy. Comprehending reality – practical barriers to industrial adoption of software maintenance automation. In *1*1th IEEE International Workshop on Program Comprehension, pages 196–205, 2003. – *Cited on pp. 4 and 5*

[20] N. Davey, P. C. Barson, S. D. H. Field, R. J. Frank, and D. S. W. Tansley. The development of a software clone detector. *I*nternational Journal of Applied Software Technology, 1(3–4):219–36, 1995. – *Cited on pp. 2, 3, and 79*

[21] Arie van Deursen. The leap year problem. *Y*ear/2000 Journal, 2(4):65–70, July/August 1998. – *Cited on p. 2*

[22] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking code clones in evolving software. In *I*CSE '07: Proceedings of the 29th International Conference on Software Engineering, pages 158–167, Washington, DC, USA, 2007. IEEE Computer Society. – *Cited on pp. 6, 12, 46, and 79*

[23] Ekwa Duala-Ekoko and Martin P. Robillard. Clonetracker: Tool support for code clone management. In *3*0th ACM/IEEE International Conference on Software Engineering, 2008. – *Cited on p. 80*

[24] Richard Fanta and Václav Rajlich. Removing clones from the code. *J*ournal of Software Maintenance, 11(4):223–243, 1999. – *Cited on p. 6*

[25] Martin Fowler. *P*atterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. – *Cited on p. 35*

[26] E. Gamma and K. Beck. *C*ontributing to Eclipse: Principles, Patterns, and Plug-Ins. Addison-Wesley, 2003. – *Cited on p. 22*

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *D*esign patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. – *Cited on p. 36*

[28] C.A.R. Hoare. The emperor's old clothes. *C*ommunications of the ACM, 24(2):75–83, 1981. – *Cited on p. 33*

[29] S. Holzner. *E*clipse Cookbook. O'Reilly, 2004. – *Cited on p. 22*

[30] Patricia Jablonski and Daqing Hou. Cren: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide. In *e*clipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange, pages 16–20, New York, NY, USA, 2007. ACM. – *Cited on p. 79*

[31] J.A. Jockin-La Bastide and G. van Kooten, editors. *K*ramers woordenboek Engels: Engels-Nederlands / Nederlands-Engels. Elsevier / Meulenhoff Educatief, 1987. – *Cited on p. vii*

[32] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *C*ASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research, pages 171–183. IBM Press, 1993. – *Cited on pp. 2 and 4*

[33] Elmar Juergens, Benjamin Hummel, Florian Deissenboeck, and Martin Feilkas. Static bug detection through analysis of inconsistent clones. In Walid Maalej and Bernd Brgge, editors, *S*oftware Engineering (Workshops), volume 122 of *L*NI, pages 443–446. GI, 2008. – *Cited on p. 81*

[34] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *I*EEE Trans. Software Eng., 28(7):654–670, 2002. – *Cited on pp. 6 and 79*

[35] Cory Kapser and Michael W. Godfrey. Aiding comprehension of cloning through categorization. In *I*WPSE '04: Proceedings of the Principles of Software Evolution, 7th International Workshop, pages 85–94, Washington, DC, USA, 2004. IEEE Computer Society. – *Cited on pp. 2 and 40*

[36] Cory Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful. In *W*CRE '06: Proceedings of the 13th Working Conference on Reverse Engineering, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society. – *Cited on pp. 1, 2, 4, 5, 10, 63, and 87*

[37] F.G. Kenyon. *O*ur Bible and the Ancient Manuscripts: Being a History of the Text and Its Translations. Eyre and Spottiswoode, 1898. – *Cited on p. 1*

[38] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in oopl. In *I*SESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society. – *Cited on pp. 1, 2, 3, 4, 5, 10, 62, and 63*

[39] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. *S*IGSOFT Softw. Eng. Notes, 30(5):187–196, 2005. – *Cited on pp. 6 and 87*

[40] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *S*AS '01: Proceedings of the 8th International Symposium on Static Analysis, pages 40–56, London, UK, 2001. Springer-Verlag. – *Cited on p. 5*

[41] Rainer Koschke. *I*dentifying and Removing Software Clones, chapter 2, pages 15–36. Springer, 2008. – *Cited on pp. 4 and 6*

[42] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *W*CRE, pages 253–262, 2006. – *Cited on pp. 2, 4, 6, 14, and 21*

[43] J. Krinke. Is cloned code more stable than non-cloned code? In *E*ighth IEEE International Working Conference on Source Code Analysis and Manipulation, pages 57–66, Sept. 2008. – *Cited on pp. 12, 17, 81, and 87*

[44] Jens Krinke. A study of consistent and inconsistent changes to code clones. In *1*4th Working Conference on Reverse Engineering, pages 170–178, 2007. – *Cited on p. 79*

[45] B. Lague, D. Proulx, J. Mayrand, E.M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *I*nternational Conference on Software Maintenance, pages 314–321, 1997. – *Cited on pp. 4 and 6*

[46] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *C*ommun. ACM, 21(7):558–565, 1978. – *Cited on p. 42*

[47] H.A. Landsberger. *H*awthorne revisited. Cornell Univ., 1968. – *Cited on p. 61*

[48] T.D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *P*roceedings of the 28th international conference on Software engineering, pages 492–501. ACM New York, NY, USA, 2006. – *Cited on pp. 9, 17, 22, 24, and 87*

[49] Rensis Likert. A technique for the measurement of attitudes. *r*chives of Psychology, 140:1–55, 1932. – *Cited on p. 60*

[50] Angela Lozano and Michel Wermelinger. Assessing the effect of clones on changeability. In *I*EEE International Conference on Software Maintenance, pages 227–236, 28 2008-Oct. 4 2008. – *Cited on p. 4*

[51] Z.A. Mann. Three public enemies: cut, copy, and paste. *C*omputer, 39(7):31–35, 2006. – *Cited on pp. 6, 9, 10, and 56*

[52] J. Mayrand, C. Leblanc, and E.M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *I*nternational Conference on Software Maintenance, pages 244–253, 1996. – *Cited on pp. 1 and 2*

[53] Leon Moonen. Generating robust parsers using island grammars. In *P*roceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01), page 13, Washington, DC, USA, 2001. IEEE Computer Society. – *Cited on p. 40*

[54] Suzanne Robertson and James Robertson. *M*astering the requirements process. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999. – *Cited on pp. 17 and 19*

[55] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. Technical report, School of Computing, Queen's University at Kingston, 2007. – *Cited on p. 4*

[56] Frank Schlesinger and Sebastian Jekutsch. Electrocodeogram: An environment for studying programming. In *W*orkshop on "Ethnographies of Code", 2006. – *Cited on p. 22*

[57] D.I.K. Sjoeberg, J.E. Hannay, O. Hansen, V.B. Kampenes, A. Karahasanovic, N.-K. Liborg, and A.C. Rekdal. A survey of controlled experiments in software engineering. In *I*EEE Transactions on Software Engineering, volume 31, pages 733–753, Sept. 2005. – *Cited on p. 56*

[58] Robert Tairas, Jeff Gray, and Ira Baxter. Visualization of clone detection results. In *e*clipse '06: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange, pages 50–54, New York, NY, USA, 2006. ACM. – *Cited on pp. 5 and 6*

[59] P. Tarr, H. Ossher, W. Harrison, and Jr. Sutton, S.M. N degrees of separation: multi-dimensional separation of concerns. In *P*roceedings of the 1999 International Conference onSoftware Engineering, pages 107–119, 1999. – *Cited on pp. 2 and 63*

[60] Walter F. Tichy. Should computer scientists experiment more. *I*EEE Computer, 31:32–40, 1998. – *Cited on p. 76*

[61] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In *V*LHCC '04: Proceedings of the 2004 IEEE Symposium on

Visual Languages - Human Centric Computing, pages 173–180, Washington, DC, USA, 2004. IEEE Computer Society. – *Cited on p. 79*

[62] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: maintenance support environment based on code clone analysis. In *I*EEE Symposium on Software Metrics, pages 67–76, 2002. – *Cited on p. 79*

[63] Valentin Weckerle. Cpc an eclipse framework for automated clone life cycle tracking and update anomaly detection. Master's thesis, Freie Universität Berlin, January 2008. – *Cited on pp. 21 and 80*

[64] P. Weiner. Linear pattern matching algorithms. In *P*roceedings of the 14th IEEE Symposium on Switching and Automata Theory, pages 1–11, 1973. – *Cited on p. 44*

[65] I. Wilmut, A.E. Schnieke, J. McWhir, A.J. Kind, and K.H.S. Campbell. Viable offspring derived from fetal and adult mammalian cells. *N*ature, 385:810–813, 1997. – *Cited on p. 1*

[66] Michiel Corneliszoon de Wit. Managing code cloning – a literature study, 2008. – *Cited on p. 7*

[67] Yali Zhang, Hamid Abdul Basit, Stan Jarzabek, Dang Anh, and Melvin Low. Query-based filtering and graphical view generation for clone analysis. In *I*EEE International Conference on Software Maintenance, pages 376–385, 28 2008-Oct. 4 2008. – *Cited on p. 81*

Appendix A

Glossary

Clone "Code clones (...) are code fragments of considerable length and significant similarity." [7] In other words, two fragments of code are considered clones if a certain *clone relation* holds. Most often, this relation is defined based on similarity or a shared origin.

Clone change resolution Strategy to cope with a change that would break one or more clone relations in a clone set, ideally with the result that the threatened clone relations are restored.

Clone family Synonymous with *clone set*.

Clone model The combined set of clone sets in a certain clone base.

Clone set The transitive closure of the clone relation. Put differently: the set of clones that are all similar to each other or share the same origin, depending on the definition of the clone relation.

Clone relation Relation between clones, mostly binary but sometimes n-ary, that links code fragments either based on similarity or shared origin. Clone relations can be strict in the sense that they only allow minimal or no differences between clones or loose, allowing for more elaborate changes.

Parameterized clone A clone of which certain parts are marked as parameters. The parameter parts of a clone are not considered when evaluating the clone relation, so that clone fragments that differ only in their parameters can still be clones.

Appendix B

Pretest Questionnaire

In the experiment conducted as part of this thesis project two questionnaires were used. In this appendix the pretest questionnaire used to get a zero-measurement before the participants started their programming assignments is printed. The following pages contain the original questions. Facsimiles of the original forms used can be found in appendix F from page 115 onwards. A more detailed explanation of each question's purpose is given in section* 5.2.1.

B.1 Personal Background

"This first question is about you. Please answer the following questions with regard to your age, education and current occupation. Your answers will be kept private and only serve to put your other answers in context."

1a. What is your age?
1b. Please sketch your educational background.
1c. What is your current occupation?

B.2 Development Experience

"Below a number of general statements about software development is shown. Please rate each of the statements on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you."

2a. I consider myself an experienced Java developer
2b. I consider myself a good/proficient Java developer
2c. I often develop software as part of a team
2d. I am often involved in developing commercial class applications
2e. Eclipse is the environment I use most for writing Java code
2f. I am familiar with Robocode

B.3 Attitude towards Code Quality

"Some more statements are shown below. These statements are about software quality. Please rate each of the statements on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you."

`3a.` Writing functional code is more important than writing clean code
`3b.` Bugs are often the result of programmer sloppiness
`3c.` When programming, I focus on writing good quality code
`3d.` I tend to write more comments than actual code
`3e.` A lot of bugs can be prevented by using better development tools
`3f.` I often use the automated refactoring tools offered by Eclipse

B.4 Attitude towards Cloning

"The following series of statements is about copy/pasting and code cloning. Please rate each of the statements on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you."

`4a.` Before this experiment I had never heard of 'code cloning'
`4b.` I use copy/paste operations often when writing code
`4c.` Copy/paste is often a good code reuse strategy
`4d.` Copy/paste is the best reuse strategy for cross-cutting concern code[1]
`4e.` I often come across inconsistently modified code copies
`4f.` I strongly believe cloning can lead to difficult to solve bugs

B.5 Expectations for a Tool like CLONEBOARD

"The final question of this survey is about your expectations for a tool that helps to maintain code clones. Please read the description below and then rate each of the following statements on a scale from 1 (totally disagree) to 5 (totally agree)."

> *"With a clone management tool, one should be able to see what parts of code have been cloned at any time. Such a tool should give a developer the opportunity to inspect cloning on a per file basis. Furthermore, the tool should alert a developer whenever he is changing a cloned fragment, offering several resolution strategies to cope with the changes. Among such strategies should be the options to update all clone instances."*

`5a.` Such a tool would significantly help to reduce clone related bugs

[1] Cross-cutting concerns are secondary aspects of a program that are orthogonal to the main logic. Examples of such concerns include logging, error handling, event triggering and code to facilitate debugging.

`5b`. Interference by such a tool would primarily be inconvenient
`5c`. A clone management tool will save me a lot of time
`5d`. I don't see the added value of such a tool
`5e`. I expect to be making use of this tool quite extensively
`5f`. The tool will not be able to solve real problems

Appendix C

Posttest Questionnaire

In the experiment conducted as part of this thesis project two questionnaires were used. In this appendix the posttest questionnaire used to evaluate the experiment and gauge the participants' experiences is printed. Facsimiles of the original forms used can be found in appendix F from page 121 onwards. A more detailed explanation of each question's purpose is given in section 5.2.1.

C.1 Assignments Experience

"The first question of this evaluation is about your overall experiences in performing the programming assignments. Please rate each of the statements below on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you."

`Ia.` The assignments were too hard for me
`Ib.` I felt a lot of time pressure
`Ic.` The assignments were very interesting to do
`Id.` I feel enthusiastic about the assignments
`Ie.` I would have needed more guidance in completing the assignments

C.2 Development Style

"In this question, we will consider your programming style during the assignments. Please rate each of the statements below on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you."

`IIa.` The programming work I did reflects my usual coding habits
`IIb.` I focused on writing functional code over clean code
`IIc.` I have written more comments than I usually do
`IId.` In the assignments I copy/pasted more often than I usually do

C.3 UI Experience

"The following statements each relate to your experiences with CLONEBOARD during the experiment. Please rate each of the statements below on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you."

`IIIa.` I encountered CLONEBOARD a lot while working on the assignments
`IIIb.` I often encountered the clone resolution window of CLONEBOARD
`IIIc.` I mostly dismissed the clone resolution window by pressing Cancel
`IIId.` The CloneView was of much use in navigating clone fragments
`IIIe.` The CloneBar was of much use in locating clone fragments

C.4 Resolution Window Experience

"In this question, the Clone Change Resolution window will be considered more closely. Please rate each of the statements below on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you."

`IVa.` It was always clear to me why the resolution window appeared
`IVb.` The resolution window always showed at convenient moments
`IVc.` The *before* and *after* fragments shown were very useful
`IVd.` The window offered sufficient information to resolve clone changes
`IVe.` The order of the available resolutions always seemed logical to me
`IVf.` I found the *Remember resolution* option very useful
`IVg.` It was clear why *Remember resolution* was not always available
`IVh.` I missed some essential change resolutions (please specify which)

C.5 Resolution Frequency

"Below all available clone change resolutions are listed. Please indicate for each of them whether you used the resolution during the experiment and if so, approximately how often you used them."

`Va.` Apply changes to all clones
`Vb.` Ignore changes
`Vc.` Parameterize clone
`Vd.` Postpone resolution
`Ve.` Unmark clone
`Vf.` Unmark clone's head
`Vg.` Unmark clone's tail

Respondents were given the choice between five answers: never, once, 2–5, 6–10, 10+.

C.6 Resolution Value

"Below, the same list of resolutions is shown again. Can you please indicate how useful you found each of the resolutions on a scale from 1 (totally useless) to 5 (very useful). If you never used the resolution, please indicate how useful you think it might be."

VIa. Apply changes to all clones
VIb. Ignore changes
VIc. Parameterize clone
VId. Postpone resolution
VIe. Unmark clone
VIf. Unmark clone's head
VIg. Unmark clone's tail

C.7 CLONEBOARD Perception

"In the general survey you were asked about your expectations for a code clone management tool. Please rate each of the statements below on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you."

VIIa. CLONEBOARD will significantly help to reduce clone related bugs
VIIb. Interference by CLONEBOARD will primarily be inconvenient
VIIc. CLONEBOARD will help me save a lot of time
VIId. I don't see the added value of CLONEBOARD
VIIe. I expect that I would be making use of CLONEBOARD quite extensively
VIIf. CLONEBOARD will not be able to solve real problems

The questions were actually arranged in a different order, to prevent respondents from recognizing the questions as the same ones asked in pretest question number 5.

C.8 UI Problems

"The last series of statements relates to CLONEBOARD's usability. Please rate each of the statements below on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you."

VIIIa. I found CLONEBOARD's Eclipse extensions easy to use
VIIIb. I will be able to get used to using CLONEBOARD in everyday coding
VIIIc. I often received an error message while using CLONEBOARD
VIIId. Bugs in CLONEBOARD severely hindered its usefulness
VIIIe. In essence CLONEBOARD is useful, but it needs a better user interface

C.9 Experiment Rating

"Please rate the following aspects of this experiment on a scale from 1 (very bad) to 9 (excellent). If you would like to comment on some of your ratings, please do so in the comment box on the next page."

`IXa.` Clearness of assignments
`IXb.` Suitability of the selected case (i.e. Robocode)
`IXc.` Quality of case documentation
`IXd.` Quality of the questionnaires
`IXe.` Opportunity to give feedback
`IXf.` 'Fun factor'
`IXg.` Overall impression

C.10 Comments

"Thank you very much for your participation in this experiment! If you would like to receive a copy of the experiment's outcomes, please write down the email address you would like to receive them on below. If you have any further remarks, findings, suggestions or other input, please write those down, too!"

The respondent was given ample space to write down his comments.

Appendix D

Programming Assignments

A set of five programming assignments was handed out to the participants in the experiment conducted as part of the thesis project. The full text of these assignments is printed in this appendix. Facsimiles of the original programming assignment pages used can be found in appendix F from page 119 onwards. Please refer to chapter 5 for an in-depth description of the assignments.

D.1 Exploring your robot

To get you more familiar with your robot, you will be adding some logging code to your robot first. This allows you to explore the existing code and see how things work. Add logging lines (using the log method implemented in `BaseCloneBot`) to CloneBot's `nextInstruction` method. Log to which position the robot is moving, which target it has chosen and with what power it fires at it.

Hints

- By pressing the CloneBot's button next to the playing field of RoboCode, you open the robot's console window. This is the window log messages are written to.
- Press the *Paint* button in the robot's console window to see what its current target and direction are. These are visualized using a red and green circle respectively.

D.2 Extend the Enemy's `toString` method

To make the information logged about the selected target more useful, you will have to extend the `toString` method of the `Enemy` class. In its original implementation, this only shows the enemy's name. Add some more information to it, such as its current location, energy level and direction.

Hints

- You can use the `StringBuilder`already created for you to add more logging information.
- Feel free to browse the `Enemy` class to look for useful information to include.

D.3 Implement better targeting routines

The CloneBot chooses its targets by calling the `selectTarget` method. This method is implemented by calling a rather silly targeting routine: `findRandomTarget`. This routine just picks one of the enemies as a target, without considering whether it would be a good candidate.

Using the `findRandomTarget` as an example, add the following similar targeting routines:

- **`findNearestTarget`.** finds the enemy that is closest. You can use the Enemy's `getDistance` method to determine its distance from your robot.
- **`findWeakestTarget`.** finds the enemy that is weakest. Use the Enemy's `getEnergy` method to see which robot has the least amount of energy left and thus is the weakest.
- **`findFittestTarget`.** finds the healthiest enemy (the one that is your best competitor). Use the `getEnergy` method to find out which enemy is healthiest.
- **`findSlowestTarget`.** finds the enemy that is moving the least (and thus is easy to aim at). Use the `getVelocity` method of the enemy to find out its speed.

Hints

- Test each of the targeting routines by inserting them into the `selectTarget` method.
- If you like, you can alter the `selectTarget` method to randomly pick one of the targeting routines, for instance by using a *switch* statement.

D.4 Getting closer to your enemy

The CloneBot is still running around like crazy, picking random locations every turn. It probably would be better to try and move your robot somewhat closer to its target enemy. Getting closer enhances the chance of hitting the enemy. Try and implement several different movement strategies. Use a similar approach to the one you used for targeting to create multiple targeting routines and slot them into the idselectNewPosition method.

Some suggestions

- **`getCloserToTarget.`** get a little closer to your target, by reducing the distance between you and your enemy. You might try and get 20% closer each time.
- **`circleTarget.`** try and move around your enemy in a circular motion. Can you figure out the math? Consider using the Point.heading and Point.move methods.
- **`confuseTarget.`** try not to be too predictable and confuse your enemy. Ideas?
- **`findBestPosition.`** pick some 100 random points in the arena and use a rating function just like was done for picking a target to see which position is best.

D.5 The final round

Your final assignment will be the most important one: actually trying to defeat your enemies. This is where you can decide the faith of your creation. You are free to alter you robot as you see fit. Your only mission is to defeat the other robots.

Hints

- Override the `onHitWall`, `onHitRobot`, `onBulletHit` and `onBulletMiss` methods to improve your robot. These events will help you to get better.
- Consider varying fire power. Less powerful bullets travel faster and are thus more accurate.

Appendix E

Experiment Results

In this chapter, tables are printed containing the coded results gathered with the questionnaires used in the pretest and posttest of the experiment, as well as data gathered by CLONEBOARD's logging facility. For reasons of privacy, the respondents' names were replaced with numerals. A total number of 7 subjects were tested. The results of the pilot experiment are not included. A more in-depth analysis of this data can be found in section 5.8. Rather than citing the full questions, only their codes are used. A full list of all questions and statements can be found in appendices B and C (for pretest and posttest questions respectively).

E.1 Pretest

Question 1

	#1	#2	#3	#4	#5	#6	#7
a	22	29	27	23	24	28	29
b	M	M	M	M	M	P	P
c	S	D	S	SD	SD	P	A

ad b. *M*: (near) MSc in computer science. *P*: (near) PhD in computer science.

ad c. *S*: student. *D*: software developer. *P*: PhD student. *A*: assistant professor.

Question 2

	#1	#2	#3	#4	#5	#6	#7
a	4	3	4	4	3	2	4
b	4	3	4	4	5	4	3
c	4	2	3	2	5	2	2
d	2	1	3	3	5	2	1
e	4	4	2	5	1	3	4
f	1	1	1	1	1	1	1

Question 3

	#1	#2	#3	#4	#5	#6	#7
a	4	2	3	2	2	4	4
b	3	4	4	2	4	4	5
c	4	4	4	5	5	3	3
d	2	2	3	2	3	2	2
e	3	3	4	4	2	4	5
f	4	2	1	3	1	1	3

Question 4

	#1	#2	#3	#4	#5	#6	#7
a	1	1	1	1	2	1	1
b	2	4	2	5	2	5	2
c	1	2	2	1	1	3	3
d	1	3	2	2	2	1	2
e	4	1	4	4	2	3	4
f	4	2	5	4	2	4	4

Question 5

	#1	#2	#3	#4	#5	#6	#7
a	4	2	3	4	4	4	5
b	2	5	4	1	3	3	3
c	4	2	4	4	2	3	4
d	1	4	2	2	4	2	1
e	2	3	3	3	3	2	3
f	1	2	2	1	2	4	1

E.2 Posttest

Question I

	#1	#2	#3	#4	#5	#6	#7
a	2	5	1	1	2	3	4
b	2	2	1	2	1	3	3
c	3	2	4	5	4	4	5
d	3	2	4	5	5	4	5
e	2	4	2	1	1	3	2

Respondent #7 commented on *a* that he found the `circleTarget` suggestion of assignment 4 too difficult.

Question II

	#1	#2	#3	#4	#5	#6	#7
a	4	4	3	4	5	4	5
b	3	2	3	4	1	4	5
c	2	5	2	2	1	2	4
d	2	2	4	1	2	1	3

Question III

	#1	#2	#3	#4	#5	#6	#7
a	3	2	4	5	3	3	4
b	3	2	4	5	3	3	4
c	1	4	3	5	4	2	1
d	1	3	2	3	2	3	2
e	1	3	2	5	2	3	3

Respondent #1 commented on *c* that he mostly dismissed the resolution window by pressing *OK*. He filled in option 1 for this question, but this should be interpreted as a 5. Respondent #4 commented on *e* that he found the pop-up window, showing clone hyperlinks particularly useful.

Question IV

	#1	#2	#3	#4	#5	#6	#7
a	3	1	4	4	5	5	5
b	1	3	2	2	4	4	3
c	1	3	4	4	2	2	4
d	2	2	5	3	4	4	4
e	3	2	4	3	4	5	5
f		3	4	2	1	2	1
g		4		2	2	5	5
h	5	4	1	5	2	1	4

Respondent #1 did not answer *f* and *g*, stating that these did not apply. Apparently, the respondent did not use the respective option.

Respondent #4 commented that he did not understand why the *parameterize clone* resolution was not always available.

Respondent #5 commented that he found the *parameterize clone* resolution very useful, allowing him to use clones as templates.

Respondent #7 commented on *c* that he found the *before* and *after* clone fragment fields too small.

Suggested change resolutions:

- Pilot Resolution that allows a method body to be parameterized as a whole.
- #1 Resolution that automatically applies a refactoring.
- #7 A variant of the *parameterize clone* resolution that can be applied to multiple changed clones at once.

Question V

	#1	#2	#3	#4	#5	#6	#7
a	1	1	1	1	3	1	1
b	3	2	3	4	4	1	2
c	3	2	1	4	3	4	4
d	3	2	2	5	1	3	5
e	3	3	3	4	3	4	4
f	3	3	1	2	1	4	4
g	3	4	1	2	1	4	4

Values 1–5 are to be interpreted as follows:

1 Never
2 Once
3 2–5 times
4 6–10 times
5 More than 10 times

Respondent #1 commented that he had picked the first resolution in the list every time, making his assessments of resolution application frequency unreliable.

Question VI

	#1	#2	#3	#4	#5	#6	#7
a	1	2	4	5	5	5	5
b	1	4	3	5	4	3	5
c	1	4	3	5	5	4	5
d	1	2	3	5	2	4	5
e	1	2	3	4	4	5	5
f	1	2	2	2	2	3	4
g	1	2	2	2	2	3	4

Question VII

	#1	#2	#3	#4	#5	#6	#7
a	1	2	4	4	4	4	1
b	5	4	3	4	4	3	2
c	1	3	4	3	4	4	4
d	5	3	1	2	2	1	1
e	1	3	3	2	3	2	3
f	5	2	2	2	4	2	1

Question VIII

	#1	#2	#3	#4	#5	#6	#7
a	1	2	5	5	5	5	5
b	1	2	4	3	4	3	4
c	1	5	1	4	2	2	2
d	1	5	1	2	2	1	1
e	4	2	3	4	4	2	1

Respondent #4 commented on *e* that the user interface should be less intrusive.

Question IX

	#1	#2	#3	#4	#5	#6	#7
a	9	8	9	8	8	8	7
b	9	8	7	9	5	6	9
c	9	8	9	8	7	8	8
d	9	8	8	8	7	8	8
e	9	8	9	8	7	8	9
f	9	8	8	9	9	8	9
g	9	8	8	9	8	8	8

Question X

A selection of the remarks added by respondents (some remarks have been reformulated or paraphrased):

#1 "The resolution window interrupts the workflow. Use a pop-up window or sidebar instead."
"Allow manual clone marking."
"Clones created by typing are not captured."
"I stopped using ctrl-c to prevent the resolution window from popping up."

#2 "Perhaps the assignment should have been bigger, allowing participants to take them home."
"The Eclipse line-copy hotkey ctrl-alt-down is not captured."

#3 "It would be useful if the before and after fragments displayed in the resolution window would scroll in parallel."
"For larger clone fragments, the ability to investigate their context while deciding on a change resolution would be useful."
"The experiment tested the creation of clones, but not their modification."

#5 "Useful tool. However, the user interface could have been a little more discrete. Without pop-ups, for instance."

E.3 Log Data

Using CLONEBOARD's built-in XML logger objective information was gathered about the subjects' copy and paste behavior. The following tables show the most important data extracted from these logs.

Copy and Paste Operations

The table below shows the frequency fragments of specific types (cf. section 4.5.2) were copied and pasted by the subjects.

Type	#1	#2	#3	#4	#5	#6	#7
COPIED FRAGMENTS							
Short	10	10	7	20	6	7	4
Long		1	2	10	1	1	5
Method	5	8	4	10	4	13	8
Identifier		1		12			1
Type				1			
PASTED FRAGMENTS							
Short	7	15	10	12	3	8	6
Long		1	1	4	1	1	5
Method	4	10	4	8	8	12	6
Identifier				10			1
Type				1			

Applied Change Resolutions

In the following table, usage statistics for the clone change resolutions are reflected. The data in the table do not include implicit *postpone* resolutions invoked by dismissing the resolution window.

Resolution	#1	#2	#3	#4	#5	#6	#7
Parameterize	4	2		4	6	6	9
Unmark	5	4	6	4	7	11	6
Unmark head		1				2	3
Unmark tail	3	10		2		7	6
Ignore	2	4	6	13	4		3
Update all		1			2		
Postpone		1		2		2	4

Appendix F

Experiment Documentation Facsimiles

For purposes of reproducibility and transparency of experimental methods, facsimiles of all documentation handed out to the participants in the experiment are printed out on the next pages. To avoid confusion with the numbering of this thesis document, page numbering has been removed from the facsimiles. Furthermore, the second page of the documentation has been removed, as this page was blank. The original pages were scaled by a factor 0.665 to fit in the page layout of this document.

This documentation contains an introductory page, a two-page pretest questionnaire, two pages of RoboCode and CLONEBOARD documentation, a further two pages of assignments and finally a four-page posttest questionaire.

The reference sheet, printed here in portrait to fit the page layout, was originally handed out to the participants as a single sheet of laminated paper, printed on both sides.

All documentation used in the experiment was laid out using Microsoft Word 2007.

F.1 Introduction

Supervised Cloning with CLONEBOARD

A Controlled Experiment

The duplication of source code is often referred to as 'code cloning'. Most programmers make intensive use of the clipboard facilities offered by modern development environments. Copying and pasting can, however, often lead to inaccuracies, inconsistencies and even bugs. A lot of research has been put into finding duplicated pieces of code, but relatively little was done to track and manage clones as they are being created.

In this research experiment, a tool dubbed CLONEBOARD is being put to the test. CLONEBOARD can be best regarded as an agent sticking to your Ctrl-C and Ctrl-V keys. Each time code is copied or pasted, CLONEBOARD will take note, so that later on, when potential bugs are being introduced, it can advise and assist to maintain consistency.

Experiment Agenda

- *Brief welcome* *± 5 min.*
- *General survey and introduction* *± 15 min.*
- *5 Programming assignments* *120 min.*
- *Evaluative survey* *± 10 min.*

Total estimated duration ***± 150 min.***

The Experiment

In this experiment, you will be using Eclipse with the CLONEBOARD plug-in installed to perform some elementary programming assignments. Before and after these assignments, you will be asked a number of questions by means of a short survey. These surveys will not take long and are meant to get an impression of your skill level and your experiences with CLONEBOARD.

The programming assignments will all be performed in an Eclipse project that has been prepared for you. All software you need is installed on a virtual machine running Windows XP, so all you will have to do is boot the machine and get going.

About the Programming Assignments

The programming assignments that are part of this experiment relate to an open source project called Robocode. Originally started as a way to keep IBM developers sharp, this game has since grown into a worldwide AI challenge. The game appeals to developer ingenuity and is often very exciting to play.

The details of Robocode are explained further on, but to give you a hint: in Robocode miniature robotic tanks wage war against each other, struggling for survival. With no more than a set of caterpillars, a gun and a simple radar, the battle is more about inventiveness than it is about brute force. The programmer's task in all this of course is to design a robot that is wittier than all other robots, using every AI and statistics technique he can think of.

Does this sound too difficult or aggressive to you? Rest assured that the assignments will all be relatively easy and peaceful.

F.2 Pretest Questionnaire

General Survey

In this short survey a number of questions regarding your experience and attitude towards cloning and clone management will be asked to get an impression of your skills and expectations.

1 **This first question is about you. Please answer the following questions with regard to your age, education and current occupation. Your answers will be kept private and only server to put your other answers in context.**

- What is your age? .....
- Please sketch your educational background: .....

  .....
- What is your current occupation? .....

  .....

2 **Below a number of general statements about software development is shown. Please rate each of the statements on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you.**

	1	2	3	4	5
I consider myself an experienced Java developer	❍	❍	❍	❍	❍
I consider myself a good/proficient Java developer	❍	❍	❍	❍	❍
I often develop software as part of a team	❍	❍	❍	❍	❍
I am often involved in developing commercial class applications	❍	❍	❍	❍	❍
Eclipse is the environment I use most for writing Java code	❍	❍	❍	❍	❍
I am familiar with Robocode.	❍	❍	❍	❍	❍

3 **Some more statements are shown below. These statements are about software quality. Please rate each of the statements on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you.**

	1	2	3	4	5
Writing functional code is more important than writing clean code	❍	❍	❍	❍	❍
Bugs are often the result of programmer sloppiness	❍	❍	❍	❍	❍
When programming, I focus on writing good quality code	❍	❍	❍	❍	❍
I tend to write more comments than actual code	❍	❍	❍	❍	❍
A lot of bugs can be prevented by using better development tools	❍	❍	❍	❍	❍
I often use the automated refactoring tools offered by Eclipse	❍	❍	❍	❍	❍

4 **The following series of statements is about copy/pasting and code cloning. Please rate each of the statements on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you.**

	1	2	3	4	5
Before this experiment I had never heard of 'code cloning'	❍	❍	❍	❍	❍
I use copy/paste operations often when writing code	❍	❍	❍	❍	❍
Copy/paste is often a good code reuse strategy	❍	❍	❍	❍	❍
Copy/paste is the best reuse strategy for cross-cutting concern code[1]	❍	❍	❍	❍	❍
I often come across inconsistently modified code copies	❍	❍	❍	❍	❍
I strongly believe cloning can lead to difficult to solve bugs	❍	❍	❍	❍	❍

5 **The final question of this survey is about your expectations for a tool that helps to maintain code clones. Please read the description below and then rate each of the following statements on a scale from 1 (totally disagree) to 5 (totally agree).**

"With a clone management tool, one should be able to see what parts of code have been cloned at any time. Such a tool should give a developer the opportunity to inspect cloning on a per file basis. Furthermore, the tool should alert a developer whenever he is changing a cloned fragment, offering several resolution strategies to cope with the changes. Among such strategies should be the options to update all clone instances."

	1	2	3	4	5
Such a tool would significantly help to reduce clone related bugs	❍	❍	❍	❍	❍
Interference by such a tool would primarily be inconvenient	❍	❍	❍	❍	❍
A clone management tool will save me a lot of time	❍	❍	❍	❍	❍
I don't see the added value of such a tool	❍	❍	❍	❍	❍
I expect to be making use of this tool quite extensively	❍	❍	❍	❍	❍
The tool will not be able to solve real problems	❍	❍	❍	❍	❍

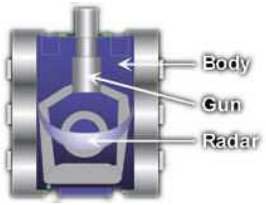
[1] Cross-cutting concerns are secondary aspects of a program that are orthogonal to the main logic. Examples of such concerns include logging, error handling, event triggering and code to facilitate debugging.

F.3 Case Documentation

Introduction to the Assignments

The programming assignments that make up the major part of this experiment all relate to Robocode. This open source project will be used as a case to test the usefulness of CLONEBOARD, an Eclipse plug-in designed to assist developers in maintaining code clones. Before we start with the actual assignments, some background information about both Robocode and CLONEBOARD is provided. A condensed version of this information is handed to you in the form of a reference sheet.

About Robocode



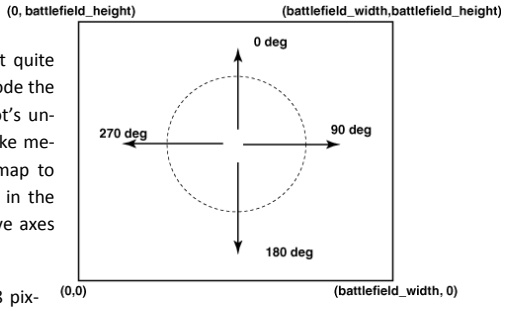
In Robocode, robots fight each other to find out which robot has the best AI. The concept is very simple, as is implementing robots. Each robot is a subclass of `Robocode.Robot` and is identified by a unique name. To make developing a robot in two hours easier and more fun, a subclass named `BaseCloneBot` was created that adds a lot of convenience methods. In the assignment, you will be developing the `CloneBot`, that is based on the `BaseCloneBot`.

Commanding the robot

To give commands and get to know more about your enemies, the `BaseCloneBot` class contains a number of convenience methods. The table below lists the available methods:

Method	Description
`moveTo(Point)`	Moves your robot to the specified location.
`aimAt(Point)`	Makes the robot aim its gun at the specified location.
`fireAt(Point, double)`	Fires a bullet with some power to the location you specify.
`fiteAt(Enemy, double)`	Carefully aims at the enemy and fires a bullet to it.
`getEnemies()`	Returns a collection of all known enemies in the arena.
`log(String)`	Writes a message to the robot's console window.

The robot's universe



In the Robocode universe, the physics are not quite the same as in the real world. That is, in Robocode the laws are a lot simpler. To start with, the robot's universe is only 800 x 600 pixels big. Pixels are like meters in the real world, just like robot turns map to seconds. The coordinate system has its origin in the lower left of the playing field, with the positive axes extending to the top and the right.

Robots can move with a maximum speed of 8 pixels/turn, accelerating with 1 pixel/turn^2. Rotation of the robot, gun and radar has different speed limit for each. These details are handled for you by the `BaseCloneBot` class, so you need not worry about them.

Testing a robot

On the virtual machine set up for this experiment, an Eclipse project has been created for you. A robot class, called `clone.CloneBot`, has been created. The robot is programmed with some random behavior. To run the robot, just run the project by pressing the ▶ button on the toolbar. The Robocode arena will be shown, preloaded with a battle in which the `CloneBot` is one of four contestants. The other three are relatively simple, yet effective bots. The goal will be to create a robot that is able to defy all three enemies at least once.

Debugging your robot

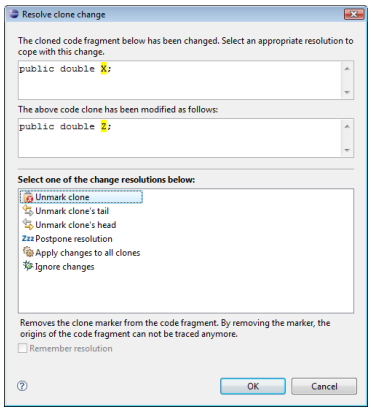
To debug your robot, you can use Eclipse's debug facilities. However, as each robot runs in its own thread, only your robot will be paused during debugging. The other robots will continue to fire and chances are that your robot will be killed before it is bug free. Mostly, it is best to either pause the battle while debugging, or use the logging facilities. By calling `CloneBot.log(String)`, you can write `String` messages to the robot's console.

Using CloneBoard

The plug-in being tested in this experiment, CLONEBOARD as it is called, has been loaded into your Eclipse environment. CLONEBOARD monitors your copy and paste actions and registers copied fragments as clones. Clones can be recognized by a thin rectangle around the cloned text, a marker in the left margin of the code and by a registration in the *Clone View*.

Clone change resolution

When a clone, created by a copy/paste operation, is modified, CLONEBOARD will intervene. A popup dialog is shown, asking you what you want to do with the modified clone. Often, when one clone is changed, the other instances will have to be updated as well. However, CLONEBOARD offers other resolutions, too. These resolutions are detailed on the reference sheet. By canceling the dialog you tell CLONEBOARD to wait for some 20 seconds. After this interval, the same dialog will popup, asking you again what to do with the changes.



Navigating clones

By hovering over fragments of cloned text, you get a number of options to handle and navigate clones. A list of all clones in your project is shown in the *Clone View* that you find at the bottom of the Eclipse window.

F.4 Programming Assignments

Programming Assignments

The following programming assignments need to be performed in the order they are presented. Don't try to write more beautiful code than you are used to. The quality of your code will not be assessed. Please try to develop in your usual way, rather than the way you think you are expected to do. In this experiment it is the plug-in that is being scrutinized, not the programmer!

> *"In this experiment it is the plug-in that is being scrutinized, not the programmer!"*

All programming assignments are to be performed in the same Java project that has been prepared for you. This project contains the following classes:

- `CloneBot`: This is the robot that you will be building on.
- `BaseCloneBot`: This is the base class of your robot, containing convenience methods that abstract away a lot of less important Robocode details.
- `Point`: A utility class used to register 2D coordinates.
- `Enemy`: A class that represents your opponents.
- `IEnemyRater`: An interface that can be used to rate enemies, such as to find the best target.

Just run the project to test your robot. A battle has already been setup for your. Among your contestants is the foul `Tracker` robot and an easy target called `SittingDuck`.

1 Exploring your robot

To get you more familiar with your robot, you will be adding some logging code to your robot first. This allows you to explore the existing code and see how things work. Add logging lines (using the `log` method implemented in `BaseCloneBot`) to `CloneBot`'s `nextInstruction` method. Log to which position the robot is moving, which target it has chosen and with what power it fires at it.

Hints:

- By pressing the CloneBot's button next to the playing field of Robocode, you open the robot's console window. This is the window log messages are written to.
- Press the *Paint* button in the robot's console window to see what its current target and direction are. These are visualized using a red and green circle respectively.

2 Extend the Enemy's *toString* method

To make the information logged about the selected target more useful, you will have to extend the `toString` method of the `Enemy` class. In its original implementation, this only shows the enemy's name. Add some more information to it, such as its current location, energy level and direction.

Hints:

- You can use the `StringBuilder` already created for you to add more logging information.
- Feel free to browse the `Enemy` class to look for useful information to include.

3 Implement better targeting routines

The `CloneBot` chooses its targets by calling the `selectTarget` method. This method is implemented by calling a rather silly targeting routine: `findRandomTarget`. This routine just picks one of the enemies as a target, without considering whether it would be a good candidate.

Using the `findRandomTarget` as an example, add the following similar targeting routines:

- `findNearestTarget`: finds the enemy that is closest. You can use the Enemy's `getDistance` method to determine its distance from your robot.
- `findWeakestTarget`: finds the enemy that is weakest. Use the Enemy's `getEnergy` method to see which robot has the least amount of energy left and thus is the weakest.
- `findFittestTarget`: finds the healthiest enemy (the one that is your best competitor). Use the `getEnergy` method to find out which enemy is healthiest.
- `findSlowestTarget`: finds the enemy that is moving the least (and thus is easy to aim at). Use the `getVelocity` method of the enemy to find out its speed.

Hints:

- Test each of the targeting routines by inserting them into the `selectTarget` method.
- If you like, you can alter the `selectTarget` method to randomly pick one of the targeting routines, for instance by using a `switch` statement.

4 Getting closer to your enemy

The CloneBot is still running around like crazy, picking random locations every turn. It probably would be better to try and move your robot somewhat closer to its target enemy. Getting closer enhances the chance of hitting the enemy. Try and implement several different movement strategies.

Use a similar approach to the one you used for targeting to create multiple targeting routines and slot them into the `selectNewPosition` method. Some suggestions:

- `getCloserToTarget`: get a little closer to your target, by reducing the distance between you and your enemy. You might try and get 20% closer each time.
- `circleTarget`: try and move around your enemy in a circular motion. Can you figure out the math? Consider using the `Point.heading` and `Point.move` methods.
- `confuseTarget`: try not to be too predictable and confuse your enemy. Ideas?
- `findBestPosition`: pick some 100 random points in the arena and use a rating function just like was done for picking a target to see which position is best.

5 The final round

Your final assignment will be the most important one: actually trying to defeat your enemies. This is where you can decide the faith of your creation. You are free to alter you robot as you see fit. Your only mission is to defeat the other robots.

Hints:

- Override the `onHitWall`, `onHitRobot`, `onBulletHit` and `onBulletMiss` methods to improve your robot. These events will help you to get better.
- Consider varying fire power. Less powerful bullets travel faster and are thus more accurate.

F.5 Posttest Questionnaire

Experiment Evaluation

Thanks for completing the programming assignments! To get an impression of your experiences with CLONEBOARD and to allow you to give your comments, please fill in the following further questions.

1 **The first question of this evaluation is about your overall experiences in performing the programming assignments. Please rate each of the statements below on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you.**

	1	2	3	4	5
The assignments were too hard for me	❍	❍	❍	❍	❍
I felt a lot of time pressure	❍	❍	❍	❍	❍
The assignments were very interesting to do	❍	❍	❍	❍	❍
I feel enthusiastic about the assignments	❍	❍	❍	❍	❍
I would have needed more guidance in completing the assignments	❍	❍	❍	❍	❍

2 **In this question, we will consider your programming style during the assignments. Please rate each of the statements below on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you.**

	1	2	3	4	5
The programming work I did reflects my usual coding habits	❍	❍	❍	❍	❍
I focused on writing functional code over clean code	❍	❍	❍	❍	❍
I have written more comments than I usually do	❍	❍	❍	❍	❍
In the assignments I copy/pasted more often than I usually do	❍	❍	❍	❍	❍

3 The following statements each relate to your experiences with CLONEBOARD during the experiment. Please rate each of the statements below on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you.

	1	2	3	4	5
I encountered CLONEBOARD a lot while working on the assignments	❍	❍	❍	❍	❍
I often encountered the clone resolution window of CLONEBOARD	❍	❍	❍	❍	❍
I mostly dismissed the clone resolution window by pressing *Cancel*	❍	❍	❍	❍	❍
The Clone View was of much use in navigating clone fragments	❍	❍	❍	❍	❍
The CloneBar[2] was of much use in locating clone fragments	❍	❍	❍	❍	❍

[2] The CloneBar is the ruler appearing in the left margin of the Java editor, highlighting lines containing cloned code with gray and blue bars.

4 **In this question, the Clone Change Resolution window will be considered more closely. Please rate each of the statements below on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you.**

	1	2	3	4	5
It was always clear to me why the resolution window appeared	❍	❍	❍	❍	❍
The resolution window always showed at convenient moments	❍	❍	❍	❍	❍
The *before* and *after* fragments shown were very useful	❍	❍	❍	❍	❍
The window offered sufficient information to resolve clone changes	❍	❍	❍	❍	❍
The order of the available resolutions always seemed logical to me	❍	❍	❍	❍	❍
I found the *Remember resolution* option very useful	❍	❍	❍	❍	❍
It was clear why *Remember resolution* was not always available	❍	❍	❍	❍	❍
I missed some essential change resolutions (please specify which)	❍	❍	❍	❍	❍

.....

5 **Below all available clone change resolutions are listed. Please indicate for each of them whether you used the resolution during the experiment and if so, approximately how often you used them.**

	Never	Once	2 – 5	6 - 10	10+
Apply changes to all clones	❍	❍	❍	❍	❍
Ignore changes	❍	❍	❍	❍	❍
Parameterize clone	❍	❍	❍	❍	❍
Postpone resolution	❍	❍	❍	❍	❍
Unmark clone	❍	❍	❍	❍	❍
Unmark clone's head	❍	❍	❍	❍	❍
Unmark clone's tail	❍	❍	❍	❍	❍

6 **Below, the same list of resolutions is shown again. Can you please indicate how useful you found each of the resolutions on a scale from 1 (totally useless) to 5 (very useful). If you never used the resolution, please indicate how useful you think it might be.**

	1	2	3	4	5
Apply changes to all clones	❍	❍	❍	❍	❍
Ignore changes	❍	❍	❍	❍	❍
Parameterize clone	❍	❍	❍	❍	❍
Postpone resolution	❍	❍	❍	❍	❍
Unmark clone	❍	❍	❍	❍	❍
Unmark clone's head	❍	❍	❍	❍	❍
Unmark clone's tail	❍	❍	❍	❍	❍

7 **In the general survey you were asked about your expectations for a code clone management tool. Please rate each of the statements below on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you.**

	1	2	3	4	5
CLONEBOARD will help me save a lot of time	❍	❍	❍	❍	❍
CLONEBOARD will not be able to solve real problems	❍	❍	❍	❍	❍
CLONEBOARD will significantly help to reduce clone related bugs	❍	❍	❍	❍	❍
I don't see the added value of CLONEBOARD	❍	❍	❍	❍	❍
I expect that I would be making use of CLONEBOARD quite extensively	❍	❍	❍	❍	❍
Interference by CLONEBOARD will primarily be inconvenient	❍	❍	❍	❍	❍

8 The last series of statements relates to CLONEBOARD's usability. Please rate each of the statements below on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you.

	1	2	3	4	5
I found CloneBoard's Eclipse extensions easy to use	❍	❍	❍	❍	❍
I will be able to get used to using CloneBoard in everyday coding	❍	❍	❍	❍	❍
I often received an error message while using CloneBoard	❍	❍	❍	❍	❍
Bugs in CloneBoard severely hindered its usefulness	❍	❍	❍	❍	❍
In essence CloneBoard is useful, but it needs a better user interface	❍	❍	❍	❍	❍

9 **Please rate the following aspects of this experiment on a scale from 1 (very bad) to 9 (excellent). If you would like to comment on some of your ratings, please do so in the comment box on the next page.**

	1	2	3	4	5	6	7	8	9
Clearness of assignments	❍	❍	❍	❍	❍	❍	❍	❍	❍
Suitability of the selected case (i.e. Robocode)	❍	❍	❍	❍	❍	❍	❍	❍	❍
Quality of case documentation	❍	❍	❍	❍	❍	❍	❍	❍	❍
Quality of the questionnaires	❍	❍	❍	❍	❍	❍	❍	❍	❍
Opportunity to give feedback	❍	❍	❍	❍	❍	❍	❍	❍	❍
'Fun factor'	❍	❍	❍	❍	❍	❍	❍	❍	❍
Overall impression	❍	❍	❍	❍	❍	❍	❍	❍	❍

10 **Thank you very much for your participation in this experiment! If you would like to receive a copy of the experiment's outcomes, please write down the email address you would like to receive them on below. If you have any further remarks, findings, suggestions or other input, please write those down, too!**

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

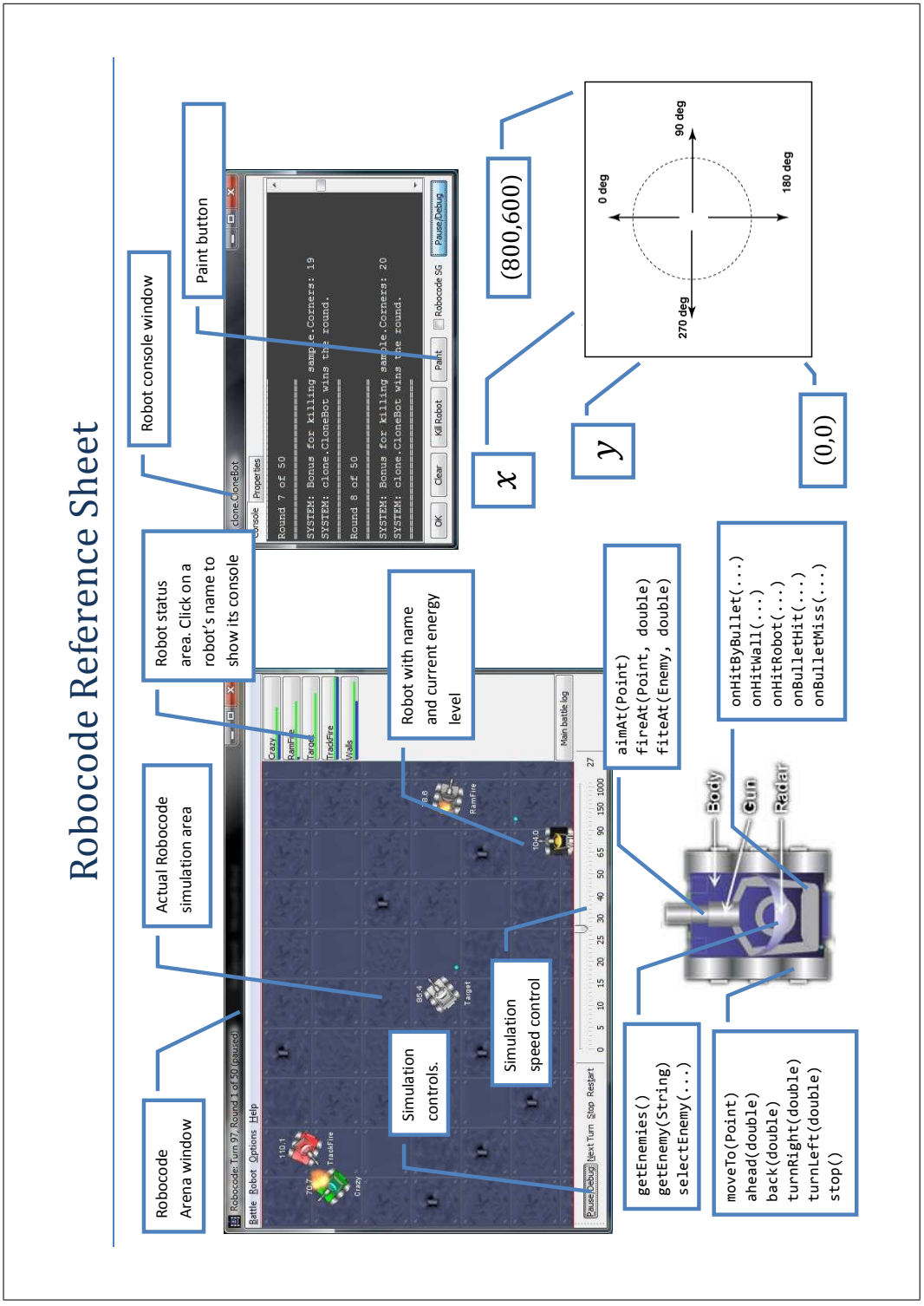
.....

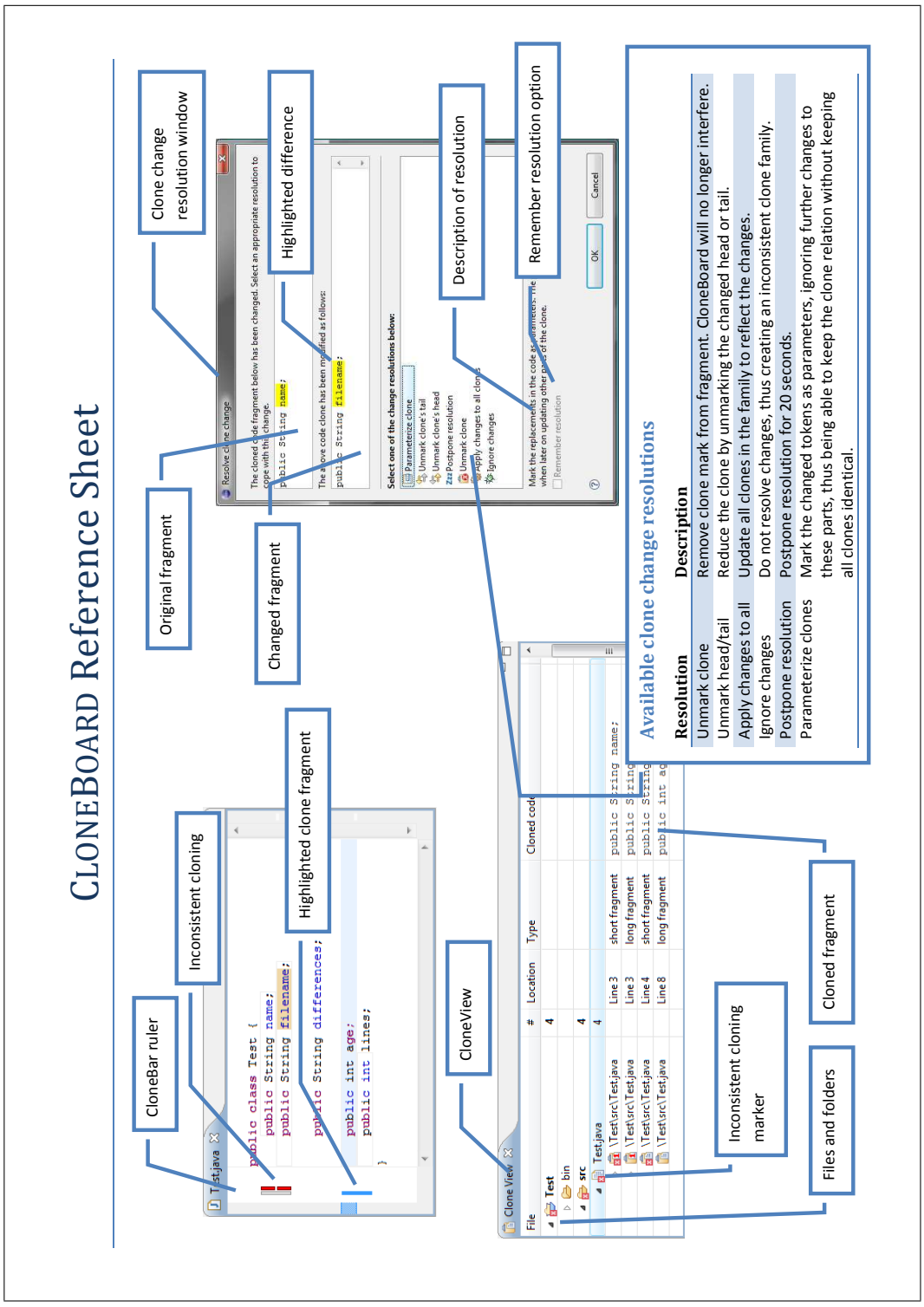
.....

.....

Thanks again!

F.6 Reference Sheet




CLONEBOARD Reference Sheet
CloneBar ruler
Inconsistent cloning
Highlighted clone fragment
CloneView
Inconsistent cloning marker
Files and folders
Cloned fragment
Original fragment
Changed fragment
Clone change resolution window
Highlighted difference
Description of resolution
Remember resolution option
Available clone change resolutions
Resolution
Description
Unmark clone
Remove clone mark from fragment. CloneBoard will no longer interfere.
Unmark head/tail
Reduce the clone by unmarking the changed head or tail.
Apply changes to all
Update all clones in the family to reflect the changes.
Ignore changes
Do not resolve changes, thus creating an inconsistent clone family.
Postpone resolution
Postpone resolution for 20 seconds.
Parameterize clones
Mark the changed tokens as parameters, ignoring further changes to these parts, thus being able to keep the clone relation without keeping all clones identical.

Index