Diploma Thesis

July 23, 2006

Evolving Code Clones

An Approach towards a Fine-Grained Analysis of Code Clone Changes and Change Couplings

Emanuel Giger

of Winterthur, ZH, Switzerland (01-703-560)

supervised by

Prof. Dr. Harald Gall

Beat Fluri

University of Zurich

Department of Informatics

s.e.a.l.

software evolution & architecture lab

Diploma Thesis

Evolving Code Clones

An Approach towards a Fine-Grained Analysis of Code Clone Changes and Change Couplings

Emanuel Giger

University of Zurich
Department of Informatics

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

Diploma Thesis

Author:	Emanuel Giger, emanuel.giger@access.unizh.ch
URL:	http://seal.ifi.unizh.ch/113/
Project period:	26th January 2006 - 26th July 2006

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

Acknowledgements

Many people were involved in the realization of my thesis and deserve thanks. I shall express my deepest gratitude to all those people for their support and valuable inputs during the last six months but remain unnamed in this acknowledgment.

Special thanks go to Professor Harald Gall for giving me the chance to write my diploma thesis at his s.e.a.l.-lab. I would like to thank my supervisor Beat Fluri for his best effort and support during the whole six months I work on this thesis

I would like to thank the developers of the CCFinderX at the University of Osaka. They gave me permission to use their tool in my thesis.

Special thanks go to my sister Rebekka Giger and her boyfriend Wissam Chamssedine for proofreading my thesis on a subject so different from their usual medical and economic backgrounds.

I would like to thank my parents Marcel and Stephanie Giger who supported me not only during the last six months but during my whole studies.

Abstract

The term *code clone* in the field of software engineering refers to the fact that a software system contains duplications in its source code. Such code clones are marked as *bad smell*. They are assumed to cause problems during the evolution and maintenance of a system, because programmers and developers may need to locate code clones in the entire source code to change them consistently. This problem manifests itself in change coupling groups – group of source code files that are often changed together. It is thus of importance to have a methodology to identify and ”disarm” such critical files specifically.

A systematical correlation between code clones and change couplings has been assumed so far. Recent research activities could neither verify this correlation nor totally reject it.

In this thesis we use a new approach that combines various technologies to investigate the relation between code clones and change couplings.

We applied our approach on two case studies. The evaluation of the results could not establish a systematic correlation or an interaction between code clones and change couplings. Nevertheless the case studies pointed out certain file groups in which code clones indeed caused change couplings. The approach developed in this thesis can be used to investigate a software system, and to identify such critical file groups in a well defined process.

Zusammenfassung

Der Begriff *Klon* bezieht sich im Gebiet der Software Entwicklung auf die Tatsache, dass ein Software System Duplizierungen in seinem Quelltext aufweist. Klone haben den Ruf eines *bad smell*. Man nimmt daher an, dass sie Probleme waehrend der Evolution und Wartung eines Software Systems bereiten. Dies ruehrt daher, dass Klone oft konsistent an allen Stellen, an welchen sie im gesamten Quelltext auftreten, angepasst werden muessen. Dieses Problem aeussert sich in einer Gruppe von Dateien, die oft zusammen geaendert werden. Umso wichtiger waere eine Methodik, die es erlaubt, kritische Dateien und Klone gezielt zu identifizieren und zu ”entschaerfen”.

Es gilt seit laengerem die Vermutung, dass ein Zusammenhang zwischen solchen Quelltext-Klonen und dem Aenderungsverhalten der betroffenen Dateien besteht. Diese Vermutung konnte in frueheren Untersuchungen nicht bestaetigt, jedoch auch nicht gaenzlich verneint werden.

In dieser Diplomarbeit wird eine neue Kombination von Techniken verwendet, um die Zusammenhaenge und Wechselwirkungen zwischen Duplizierungen und Aenderungskopplungen zu untersuchen.

In unseren Fallstudien konnten wir keinen solchen systematischen Zusammenhang feststellen. Dateien mit einer hoeheren Anzahl von Klonen, werden nicht zwingend oefter miteinander geaendert. Die Untersuchungen haben jedoch gezeigt, dass in gewissen Faellen gemeinsame Aenderungen tatsaechlich auf Klone zurueck zu fuehren sind. Unser Ansatz kann verwendet werden, um diese Faelle gezielt in einem System zu suchen, und im Zuge eines Refactoring zu entfernen.

Contents

1 Introduction **1**
- 1.1 Setting the Scene - Code Clones, Change Couplings and Software Evolution 1
- 1.2 Research Goals and Envisioned Outcome 2
- 1.3 Structure of the Thesis 3

2 State of Code Clone Detection and Change Coupling Analysis **5**
- 2.1 Code Clone Detection 5
- 2.2 Extraction of Release History Information 5
- 2.3 Change Coupling Analysis 6
- 2.4 Fine-grained Analysis of Source Code Entities 6

3 Approach **9**
- 3.1 Introduction 9
- 3.2 Framework 9
  - 3.2.1 Preparation Steps 10
  - 3.2.2 Ambiguous Steps 14
  - 3.2.3 Repetitive Steps 15
- 3.3 Implementation of Step 1 - 3 17
  - 3.3.1 Coarse Idea of the Target Software System 17
  - 3.3.2 Choosing the Environment, Tools and Technologies 17
  - 3.3.3 Defining a Set of Metrics 20
- 3.4 Final Remarks 26

4 The Clone Analyzer **27**
- 4.1 Introduction 27
- 4.2 Implementing the Clone Analyzer 27
  - 4.2.1 Eclipse 27
  - 4.2.2 Design and Architecture of the Clone Analyzer 28
- 4.3 Clone Analyzer Manual 37
  - 4.3.1 Setup 37
  - 4.3.2 Clone Analyzer Properties 38
  - 4.3.3 Context Menu Integration of the Clone Analyzer 38
  - 4.3.4 Start a Clone Detection 40
  - 4.3.5 Perform Code Clone Analysis 40
  - 4.3.6 Perform Change Coupling Analysis 43
  - 4.3.7 Building a File-History 43
  - 4.3.8 Generating Log Files 45

4.4 Final Remarks 46

5 Case Studies and Evaluation 47
5.1 Environment 47
5.2 C / C++ Case Study - Mozilla 47
5.2.1 Input Selection 48
5.2.2 Obtaining and Collecting Initial Code Clone and Change Coupling Information 48
5.2.3 Identification of Possible Clone and Coupling Candidates 48
5.2.4 Tracking Clones over Time 49
5.2.5 Analyzing Data from Tracking Process 49
5.2.6 Correlation of Code Clones and Change Couplings 51
5.2.7 Conclusion 53
5.3 Java Case Study - Eclipse 53
5.3.1 Correlation of Code Clones and Change Couplings 54
5.4 Evaluation 55
5.5 Limitation of the Results 56

6 Conclusion and Final Remarks 59
6.1 Contributions 59
6.2 Lessons Learned 60
6.3 Future Work 60

A Glossary 63

B File Formats 67
B.1 Format of Positional Transformed Clone Data Output File 67
B.2 Format of Statistic File 68

C List of Used Tools 69

D Content of the Delivered CD-ROM 71

List of Figures

3.1 Framework Overview 11

4.1 Eclipse-Platform Architecture and Plug-Ins, Imitation of [Beaton 2006] 28
4.2 Workbench-Window of Eclipse 3.1.2 on Windows XP 29
4.3 Simplified Hibernate Code Clone Model 33
4.4 Context Menu of the EVOLIZERBASE 35
4.5 Context Menu of the CLONEANALYZER for Source Code Files 39
4.6 Context Menu of the CLONEANALYZER for Folders 39
4.7 CCFinderX Input Structure 41
4.8 Code Clone Browser for nsMathMLmrootFrame.cpp 41
4.9 Position Table of Code Clone 30 in nsMathMLmrootFrame.cpp and nsMathMLmsqrtFrame.cpp 42
4.10 Code Clone Overview of the Eclipse Project mathml 43
4.11 Change Coupling Browser for File nsMathMLmrootFrame.cpp with #Files = 4 and #Coccurrences = 11 44
4.12 File-History Panel for Code Clone 158 45
4.13 Temporal Appearance Matrix for Clone 158 45

5.1 Sample Plot and Regression Function (Mozilla) 53
5.2 Sample Plot and Regression Function (Eclipse) 56

List of Tables

5.1 Frequency of σ Values (Mozilla) 50
5.2 Frequency of ϱ Values (Mozilla) 52
5.3 Frequency of ϱ Values (Eclipse) 55

List of Listings

1.1 Java Code Snippet 2
3.1 Example Source Code 1 18
3.2 Example Source Code 2 with Additional Comment 19
3.3 Example Source Code 3 19
3.4 Example Source Code 4 with an Additional Indent and Blank Lines Removed . . . 19
4.1 CCFinderX Output Processed with P-Option 30
4.2 CCFinderX Output with Positional Transformed Clone Data 32
4.3 Code Clone and Source Code Entities 34
4.4 CCFinderX Output for Source Files with Revision-Number Appendix 36
5.1 Log File for Source Code Files *nsMsgProgress.cpp* and *nsMsgStatusFeedback.cpp* . . . 50

Chapter 1

Introduction

Code clones or duplicated source code fragments are regarded as *bad smell* in a software system, and are said to cause maintenance problems during its evolution. This thesis will investigate the relation and interaction between code clones and change couplings based on fine-grained analysis of source code changes, code clone data, and information about the release history of a software system.

1.1 Setting the Scene - Code Clones, Change Couplings and Software Evolution

Many research work has been made about code clones; how to detect them and about how they influence a software system (negatively). Code Clones are commonly recognized as a major bad smell [Fowler *et al.* 1999].[1] A code clone is a duplicated fragment of source code appearing more than once in a software system. Geiger gives several more formal definitions for a code clone in [Geiger 2005]. There are various reasons why source code contains duplicated code fragments: Bad design decisions in a software system, limitations of the used programming language, unexperienced developers or time pressure, because code clones are an easy way for programmers and developers to achieve reuse of source code artifacts. Practically code clones are created by pure copy-paste of source code and sometimes by slightly modifying it.

As bad smells code clones are said to lower the quality of a software system. They are often the consequences of ”ad-hoc” changes. Such changes take place regardlessly of the system architecture [Krahn and Rumpe 2006]. Generally code clones bloat a software system, because its source code is not written as compactly and as efficiently as possible. Such unnecessary or dead code decreases the readability and increases the possibility of defects. Due to their negative impact code clones lie in the center of many refactoring activities. This thesis contributes to those activities by developing a methodology to handle code clones and change couplings based upon a new combination of different technologies.

In most cases a software system is far from being stable after its first release and initial delivery. But it evolves, changes and adapts to new requirements in its environment to remain useful to its users. Lehman formulated this fact in several famous laws in [Lehman and Belady 1985]. Those laws explain why software has to evolve, and why such an evolution mostly goes hand in hand with increasing complexity of the software system. According to [Boehm 1981], costs

[1]The term *bad smell* is an indication for potential problems in the source code of a software system([Fowler *et al.* 1999], [Wake 2003]).

during this evolutionary stage after the first delivery can make up to 60% of the total costs for the entire life time of a software system – even up to 70% following [Grubb and Takang 2003]. A methodology to reduce these maintenance and evolution efforts as well as their associated costs is thus necessary.

This thesis links code clones and certain aspects of software evolution. Let us have look at an example. Listing 1.1 shows a simple Java source code snippet that opens a connection to the database `db_clients`.

```
Connection con = null;
String url = "jdbc:db2://host.domain.com/6790/db_clients";
String user = "guest";
String password = "guest";

try{
   Class.forName ("COM.ibm.db2.jdbc.net.DB2Driver");
   con = DriverManager.getConnection(url,user,password);
}
catch(Exception e){
   System.err.println(e.getMessage());
}
```

Listing 1.1: Java Code Snippet

Supposed this snippet is ”cloned” into every class of a software system that needs access to the database. Now whenever the URL of the database server changes during the evolution, all files that contain this code clone must be adapted. The negative impact on the software evolution rises as more classes are affected by this code clone. The effort of maintenance increases because all cloned code snippets must be located in the system, and then they have to be changed consistently.

An expression of maintainability problems are change couplings groups. The term *Change Coupling Group* is explained in [Fluri *et al.* 2005] and implies that two or more files are commonly changed at the same time. Coupling groups occurring relatively often during the lifetime of a system might point out some problems in the evolution of this system. In the above example change couplings are caused by the code clone because the value of the `url` field is ”hard coded” as a Java string within the duplicated source fragment. A refactoring *e.g.*, extracting the connection procedure into a single method or moving `url` as static field into a Java interface will detach the affected files.

It has been long time taken for granted that there is a (positive) correlation between code clones and change couplings. We therefore expect that an increase in the amount of duplicated code is reflected by an increase of change couplings of the affected source files.

1.2 Research Goals and Envisioned Outcome

In this thesis we investigate whether the relation between changes in code clone fragments and change couplings exhibits a correlation. If such a correlation can be deduced and expressed in a useful statement *e.g.*,”*Most changes in a code clone affect all files containing this clone.*” or ”*The more code clones a group of files shares, the more these files are change coupled.*”; this is a true incentive during the initial development of a software system to reduce the number of code clones as early as possible in order to cut the costs for later maintenance and evolution of the system.

To answer this question we use a combination of fine-grained source code change analysis, data of the release history of a software system and code clone detection technology. The advantage of our approach is that it can detect changes in code clones between different file versions. Moreover, we can tell whether a clone changed only in a single file or in all affected files at the same time, and therefore caused a change coupling. Data from this analysis can be used to judge the state of a software system in the context of code clones and change couplings. Based upon this assessment it is possible to predict the future behavior of the system.

Despite the fact that there might not be a significant correlation of code clones and change couplings, our combination of different technologies provides a good mechanism for software engineers to identify systematically possible refactoring candidates. For instance, a group of files is identified having a large number of clones, and most of their common check-ins were caused by changes in one of these cloned fragments. Then such a group can be a suitable refactoring candidate and is worth taking a closer look.

We have mainly two goals: First, answering the question if there is a correlation between code clones and change couplings. Second, providing a methodology to examine a software system and detect critical code clones. To meet these research goals, an abstract framework is developed in Chapter 3 to combine fine-grained source code change analysis, release history data, and code clone information. The relevance and benevolence of our approach is evaluated with two case studies in Chapter 5.

Evolution is an inescapable property of software systems. Thus a clearly defined software evolution process and techniques that minimize the negative side effects of software evolution are desirable. The results of this thesis are intended to contribute to such a systematic process by providing mechanisms to deal with code clones and change couplings.

1.3 Structure of the Thesis

The remainder of this thesis is organized as followed: Chapter 2 presents related work in the field of code clone detection (Section 2.1), extraction of release history information (Section 2.2), analyzing change couplings (Section 2.3), and in the field of fine-grained analysis of source code changes using abstract syntax tree differencing algorithms (Section 2.4). Chapter 3 presents an abstract approach towards a fine-grained analysis of change couplings and code clone changes. We then present a concrete implementation of this approach. The core of the implementation is the Eclipse Plug-In CloneAnalyzer. It is presented in Chapter 4. The abstract framework as well as the the exemplary implementation are then evaluated by means of a C/C++ and a Java case study. The results of these case studies are used to explain the relation between code clones and change couplings. We used the open source Mozilla web browser project for the C/C++ case study, and source code from the open source Eclipse SDK project for the Java case study.

Chapter 6 closes this thesis by presenting and summarizing its results, contributions and lessons learned and by pointing out questions and ideas for future work. The glossary in Appendix A explains the technical terms used in this thesis.

Chapter 2

State of Code Clone Detection and Change Coupling Analysis

This chapter shows what has been covered so far, what has been left out and what insights and statements were made by related work. It also highlights where this thesis can attach and continue this related work; where it can produce a delta to its related work and give new conclusions. Reading this chapter is also helpful to understand the abstract framework in Chapter 3.2 and the designs decisions made in Chapter 4 for the implementation of the CLONEANALYZER.

2.1 Code Clone Detection

When investigating the relation between code clones and change couplings, the detection of code clones plays a central role. The methodology of how to detect code clones in source code is an active research field itself.

The detection of code clones is not trivial, since code clones are seldom exact copies of their original source code fragments but experience small changes that make a code clone slightly different from its original, *e.g.,* renaming or different condition statements. Therefore sophisticated approaches taking into account such small changes are needed to detect code clones. The discussion of *code clone detection* lies not in the main focus of this thesis. A detailed introduction into this subject, an overview of several approaches to detect code clones and an evaluation of code clone detection tools by a defined set of criteria is given in [Geiger 2005]. For the rest of this thesis the results and conclusions from [Geiger 2005] are used and referred to when necessary.

2.2 Extraction of Release History Information

Most mid-size and large software projects make use of version control systems. They help managing the source code in a central repository and allow several developers to work simultaneously. When using version control systems like CVS or SVN a huge volume of historical data is collected. This data depicts interesting views of a software system, and is used for a retrospective analysis about the evolution and temporal behavior of a system. Fischer *et al.* introduce an approach for extracting data from version control systems and storing it into a SQL database in [Fischer *et al.* 2003]. This version information is then enriched by data from bug tracking systems.

A meta model for analyzing release history data is presented in [Marjanovic 2006]. A simplified version of this model was adapted for the implementation of the EVOLIZERBASE that is used in this thesis to extract release history data from CVS (cf. Section 3.3.2).

Data from version control systems can be used to calculate change couplings in the release history of a software system. Besides change couplings many other metrics can be computed with this data *e.g.*, error proneness (number of errors), change rates (number of changes within a certain amount of time). Some of these metrics are used in the evaluation case study in [Fischer *et al.* 2003].

2.3 Change Coupling Analysis

While files and other source code artifacts of a software system can be coupled in different ways, change couplings are a special kind of logical couplings [Geiger 2005]. As a subset of logical couplings, change couplings can not be detected and identified by examining the source code and by program analysis alone. As change couplings refer to a temporal aspect of a software system, they can be identified by using historical data form version control systems. A definition of change couplings is given in [Fluri *et al.* 2005]: *"Two files exhibit a change coupling if they are commonly committed e.g., at the same time, by the same author, and with the same modification report*." Of special interest are those files that have a strong change coupling past and occur often throughout the release history. Such file groups might indicate weaknesses in the design of a software system. A refactoring of the affected files may reduce their change couplings.

Fluri *et al.* further use the structure comparison functionality of the Eclipse Compare Plug-In[1] to detect source code changes in [Fluri *et al.* 2005]. They evaluate their approach of *fine-grained change coupling analysis* by a Java case study to investigate the impact of structural source code changes on the amount of change couplings. That case study reveals that a significant number of change couplings are not caused by structural changes.

As stated in Section 1.2 the research goal is to examine if code clones cause files to be change coupled. There are other reasons why files can be changed and then checked-in together: All files of a change coupling group contribute to the same function and exhibit functional dependency, or there is a strong code ownership and a specific programmer is responsible for all his files.

An approach to investigate the relation between code clones and change couplings is presented in [Geiger 2005]. This approach was applied on a Mozilla case study. The results of the case study showed that a (positive) correlation cannot be statistically verified nor completely denied. A major shortage of this approach is that change couplings are calculated on the level of files. When calculating change couplings the approach in [Geiger 2005] relied only on the information provided by CVS that is used in the Mozilla project. CVS does not give information where changes took place in a source code file between two subsequent check-ins. It is therefore unverifiable whether the change couplings were caused by changes in cloned source code fragments.

2.4 Fine-grained Analysis of Source Code Entities

The shortcomings of the approach in [Geiger 2005] imply that a mechanism is necessary to extract those changes in a source code file that occurred within code clones. As shown, relying only on change data given by CVS is insufficient. This data not only lacks the information where changes took place in source code files, but most version control systems detect changes in source code on a textual basis using text differencing algorithms such as the Unix `diff`. Thus changes in such

[1] org.eclipse.compare

version control systems are detached from particular source code entities. Textual changes may be reported although the corresponding source code entity did not change syntactically.

Fluri and Gall suggest an approach to detect source code changes on a syntactical level by using differencing algorithms that compare abstract syntax trees in [Fluri and Gall 2006]. They further introduce a taxonomy to describe the significance level of source code changes based on the elementary tree edit operations *insert*, *delete*, *move* and *update*. It assigns a *significance of the change* to every detected source code change in order to be able to define its impact on other source code entities. In their ArgoUML Java case study they rise two questions:

- To what extent are *lines added/removed* taken from the CVS log indicators for the significance of the applied changes?
- Do the change significance of change coupled files behave similarly?

The results of the case study showed that *lines added/removed* are an insufficient indicator for the significance of changes. Concerning the second question, the case study concludes that change significance of change coupled files do not behave similarly.

To our knowledge the novelty of the approach in this thesis is, that it combines all mentioned approaches and techniques. By using fine-grained source code change analysis we can determine whether a code clone was affected by a structural change. Using the release history information extracted from version control systems, change couplings can be calculated. Therefore the approach is able to extract exactly those change couplings due to changes in cloned source code fragments. This allows to evolve code clones and to draw a picture of their change behavior over time.

Chapter 3

Approach

3.1 Introduction

The research goal of this thesis is to investigate the relation between code clones and change couplings. Especially we are interested in those change couplings that were caused by structural changes in code clones. As shown in Chapter 2 we combine mainly three techniques: *Code Clone Detection*, *Fine-Grained Source Code Change Analysis* and *Release History Information*. This chapter presents the approach that was developed in this thesis to meet the research goals. In a first step we will take a look at it from a abstract level, detached from any specific tool or technology. From this point of view the approach can be seen as a domain independent framework or as a recipe / guideline to answer the research questions of this thesis. Everyone interested in examining code clones and change couplings can implement a concrete solution of this framework tailored to his very own needs. In this first step we address general questions and problems concerning the framework itself and the subject of code clone and change coupling analysis. We use this abstract point of view to present the framework as a whole. We will neglect specific problems that emerge from concrete design decisions.

In a second step we switch to a concrete level by presenting our implementation of this framework. We show how the framework is implemented according to the specifications and requirements of this thesis. For every abstract step of the framework the concrete corresponding solution in our implementation is given. The focus shifts from general to specific problems and challenges that arise from our design decisions. The result of this implementation is a methodology whose core is a fully automated tool that allows to investigate the relation of code clones and change couplings, and to reason about the state of a software system.

Section 3.2 contains the abstract presentation of the framework. Section 3.3, Chapter 4, and Chapter 5 refer to the concrete implementation.

3.2 Framework

The approach can be split up in several steps: The ”*preparation steps*” are done once or only a limited number of times. They are concerned with initial design decisions about tools and technologies to be used, with the target platform for the later code clone and change coupling analysis, or with mining initial clone and coupling data. Other steps – mostly later steps within the framework – are part of the actual analysis. They are very repetitive and done every time as information about code clones and change couplings in a software system or one of its parts / modules is desired. The analysis is responsible for the computation of metrics and measures allowing the software engineer to draw conclusions about the state of his system. Furthermore the results of the analysis can be used to identify critical changed coupled file groups and code clones. The framework consists of 10 steps:

1. Coarse Idea of the Target Software System
2. Choosing the Environment, Tools and Technologies
3. Defining a Set of Metrics for the Analysis
4. Implementation of a Tool
5. Input Selection
6. Obtaining and Collecting Initial Code Clone and Change Coupling Information
7. Identification of Suitable Clone and Coupling Candidates
8. Tracking Code Clones and their Change Behavior over Time
9. Analyzing Data from the Preceding "*Tracking Process*"
10. Based on the Previous Analysis – if Required – Refactoring Measures are Considered and Implemented.

For every step a detailed, theoretical and abstract description is given in this section. Some of the background has already been covered by previous chapters. Where necessary, it is shortly described or references to related work covering the relevant background are given. The focus of this chapter lies on the illustration of every single step, its meaning inside the framework and how it might be adapted to a specific implementation. Figure 3.1 gives an overview of the framework steps in their temporal sequence.

In the following these steps are classified into *preparation steps*, *ambiguous steps* and *repetitive steps*.[1] For every classification type there is a explanation of its meaning inside the framework. The classification of each step of the framework into one these tree types is however not set in stone, actually it may vary as the case arises.

3.2.1 Preparation Steps

These steps are not part of the actual code clone and change coupling analysis of a software system. They are necessary preconditions that must be done to enable this analysis. They are mostly implemented once and remain stable as long as no major change within the target application domain of the framework takes place.

Coarse Idea of the Target Software System

The task of this step is to determine the concrete application domain for which the framework is to be implemented. This step is important because the design decisions of the later framework steps depend on the properties of the software system(s) that is / are subject to the code clone and change coupling analysis. For instance, if we want to analyze a software system written in C / C++, it would be a waste of time to evaluate code clone detection tools among other candidates

[1]Neither the classification itself is common valid nor are its types part of commonly accepted standard. They are specifically chosen in the context of this thesis and might have other or no meanings outside the thesis

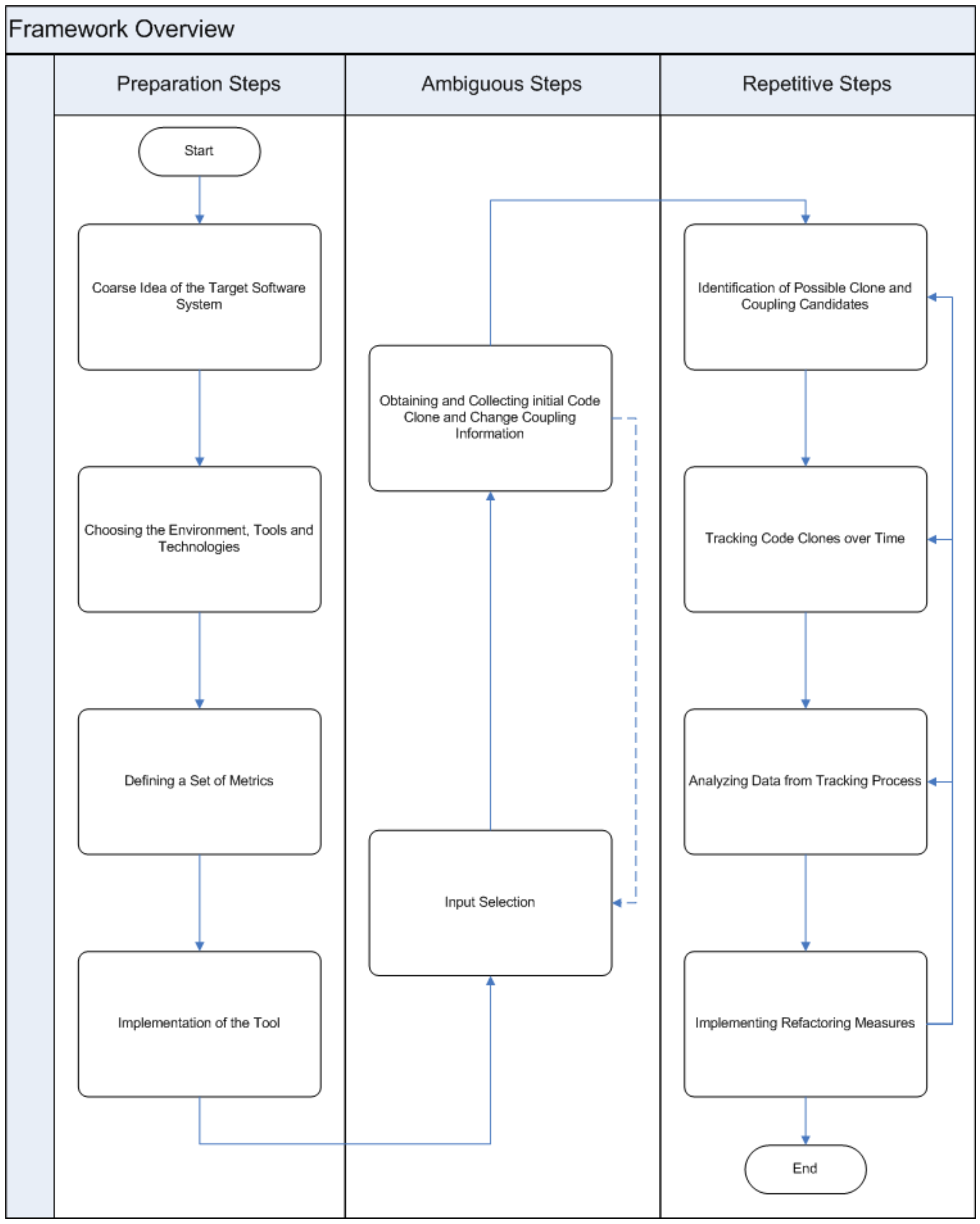

Framework Overview
Preparation Steps
Ambiguous Steps
Repetitive Steps
Start
Coarse Idea of the Target Software System
Choosing the Environment, Tools and Technologies
Defining a Set of Metrics
Implementation of the Tool
Obtaining and Collecting initial Code Clone and Change Coupling Information
Input Selection
Identification of Possible Clone and Coupling Candidates
Tracking Code Clones over Time
Analyzing Data from Tracking Process
Implementing Refactoring Measures
End

Figure 3.1: Framework Overview

that are not able to process C / C++ source code. Another question is : *"What is the amount of lines of source code that is to be analyzed?"* This ensures that one chooses a clone detection tool that is able to handle a number large enough of lines of source code. Such thoughts help to focus on the concrete initial position under which the analysis takes place.

Since step 4 covers the implementation of a tool that automates the analysis process, this step can be seen as *requirements engineering.* It is widely accepted that a adequate and systematic requirements specification reduces later costs (*e.g.*, [Glinz 05 / 06]).

It is important that this step is carried out carefully, because it sets the boundaries for any concrete implementation of the framework and therefore influences the quality of the outcomings of the following steps.

Choosing the Environment, Tools and Technologies

This is a crucial step in the whole framework, since it is indirectly responsible for the input data of the analysis (*garbage in - garbage out*-Problem). The following list shows and illustrates all components to harvest the input data:

- **Code Clone Detection**: As we take a look at source code changes in code clone fragments, we need a way to detect code clones in the source code of our target software system. The final choice of a clone detection tool as part of the implementation depends on various criteria: Such as language support, maximum input amount of source code lines the tool can process, comparison of candidates returned by the detection tool etc. A catalog of possible criteria as well as an evaluation of several code clone detection tools according to that catalog is given in [Geiger 2005]. We return to that catalog in Section 3.3.2. There are other additional criteria to consider in the context of this thesis: *"Does the clone detection tool provide an open accessible API?"* This question is of importance if the framework implementation is to be realized as a fully integrated tool. *"Is the clone detection tool itself platform independent?"*, *"Does it run only on one specific operating system?"*, *"How hardware-resource consuming is the clone detection tool?"* If the resulting implementation is intended to run on each software engineers desktop or portable device, it makes no sense to decide in favor of a clone detection tool that has hardware requirements exceeding by far the resources of an average developer station.

  As a result a clone detection tool should emerge that has been carefully evaluated according to all needs of the concrete situation. To emphasize: If the previous step *Coarse Idea of the Target Software System* has been done accurately, the evaluation of code clone detection tools leads to a faster and more satisfying choice.

- **Change Distilling**: The research goal in Section 1.2 of this thesis implies that a mechanism is needed to identify structural source code changes (in a code clone). There are two main consideration when choosing such a mechanism: First of all, the mechanism must be able to process the programming language of choice. Second, the level of granularity; to what depth of source code entities should changes be detected? For instance, if the target programming language is Java; one level - in the case of Java the most simple level of granularity - is the one that detects changes on the level of a Java-Class. For a fine-grained analysis of source code changes this is surely not the adequate choice, since code clones mostly never cover the entire class. In such a case it is not possible to tell whether changes arise within the code clone or elsewhere in the class. This level of granularity would reduce the framework more or less to the approach used in [Geiger 2005] where change couplings are calculated on the level of files.[2]

[2]If interfaces are treated the same as classes and the dogma *"one Java-Class or -Interface per file"* is strictly followed, the framework would be indeed reduced completely to the approach used in [Geiger 2005].

A deeper level of granularity is achieved where source code changes are recognized on the level of Class-Members. In the object-oriented context of Java this includes fields and methods. Again, if the code clone for example covers only a *for*-Statement and eventual nested statements but not the entire method, this may lead to identification of changes that do not meet the claim of a ”*fine grained analysis of source code changes*”. An additional facet comes into play when not only the identification of source code changes themselves, but also the extraction of the change types and their significance are desirable as described in [Fluri and Gall 2006].

The level of granularity implies that there must be some kind of a parser that is able to identify the source code entities within the string based token stream of the source code file according to the chosen level of granularity and the grammar of the target programming language.

- **Release History Tool**: Investigating the relation between source code changes in code clones and change couplings requires information about the release history of the software system. Two questions arise: ”*Where can such information be found?*” and ”*How is the information extracted from its repository or storage location?*” A mechanism or a tool is needed that accesses such information in its repository and extracts it in a suitable manner. An approach by [Fischer *et al.* 2003] using CVS and Bugzilla is mentioned in Section 2.2.

Defining a Set of Metrics

To discuss the relation between source code changes in code clones and change couplings, and to describe and qualify the severity of an eventual correlation, a certain set of metrics is necessary.

The question of what marks a good metric in general, and especially of what marks a good metric to describe a software system deserves a whole book by itself and is a strongly discussed issue. What set of metric is finally chosen for the concrete implementation of this framework is open as the case arises. Since this framework analyzes code clones and change couplings the set of metrics must of course somehow take these factors into account. For the case where change couplings are calculated on the level of files, Geiger develops a good set of metrics in [Geiger 2005]. Several properties of a good metric are given in [Glinz 2005]:

- **Validity**: Meaning, that the metrics must measure the intended issues. The relevant factors to measure in the context of this thesis are code clones and change couplings.

- **Significance**: Is it possible to draw meaningful conclusions about the relation between code clones and change couplings based on the metrics? Do the metrics allow to postulate meaningful statements about the state and the evolutionary behavior of the examined software system concerning code clones and change couplings?

- **Stability/Reproducibility**: Under the same conditions the metrics should return always the same results, at least when ceteris paribus the same modules of the same software system are analyzed.

- **Analysis**: Is it possible to perform statistical analysis based on the computed results of the metrics?

A possible set of metrics tailored for this thesis is presented in Section 3.3.3.

Implementation of a Tool

Short recapitulation: Up to now the selection process for mechanisms and tools to obtain all necessary input data has been covered. We also gave a first general glance at the definition of measures and metrics for the later analysis.

This step of the framework treats the implementation of a tool that glues all previously mentioned components together and actually performs the analysis. The purpose of such a tool is to simplify the remaining steps of the framework by integrating and automating them, and to minimize the workload for the user. This step is very open, since the resulting implementation strongly varies with general conditions under which it is developed. Nevertheless all implementations must be capable to perform a minimal set of tasks in order to be able to carry out the upcoming code clone and change coupling analysis. The list given below shows the minimal set of tasks, whereas a single task does not necessarily correspond to a certain framework step. It is possible that a task is only one part of a step, or the same task is executed in several framework steps.[3]

1. **Code Clone Detection**: Automated detection of code clones by means of the chosen clone detection tool. Furthermore the tool must provide information about the position of a code clone to facilitate an exact localization of a code clone inside its affected source code files.

2. **Find Corresponding Source Code Entity**: The tool must be able to locate code clones within a file and find its corresponding source code entities.

3. **Detect Source Code Changes**: The tool must be able to use the comparing functionality of the chance distilling tool to identify structural source code changes in code clones.

4. **Release History Information**: Automated access and extraction of relevant information about the release history of the target software system must be possible. Based on this data change couplings are then calculated.

5. **Track Code Clones over Time**: As the research goal is to investigate the relation of changes in code clones and change couplings, the tool must allow to track code clones in the release history of a system and detect changes between two subsequent file revisions. This task mainly consists of the combination of the previous tasks 1 - 4.

6. **Computation and Presentation of Metrics**: At the very end, the tool must compute the defined set of metrics, and present them to the user in a way that meaningful conclusions about the state of a software system can be drawn.

3.2.2 Ambiguous Steps

The steps explained in Section 3.2.1 provide the groundwork for all later steps of the framework. Those earlier steps are mostly processed once. They might be repeated if a major change in the application domain of the framework occurs *e.g.,* one wants to analyze a new software system written in a different programming language. A careful and systematic carrying out of step "*Coarse idea of the target software system*" in Section 3.2.1 reduces the possibility of such a major change.

The definite character of the steps in this section mainly depends on the decisions of users. As the case arises, they can be more like a preparation step or have the nature of a repetitive step.

[3]The order of the task enumeration does not necessarily reflect the temporal sequence of their execution

Input Selection

This section covers the question whether an entire software system or only certain parts of it are subject to the code clone and change coupling analysis. There are two aspects to consider: First a large industrial software project is often not written in only one programming language, moreover it is possible that a certain number of files does not even contain source code. The source code of a software system often includes images, makefiles, manifest files, documentary files, interface definition files etc. Second the size of a software system. The analysis of an entire large software system results in an amount of data not analyzable due to limitations of used technologies and tools, or they become unmanageable by the user.

This step must address both of this problems. After the degree of the input and the programming language has been decided, the entire source code must be narrowed down. For a small or medium software system, it might be possible and reasonable to take a look at the entire system at once. In this case the selection of the input happens once. For a lager system it is wise to analyze only certain parts of it at once. So the input selection happens every time one decides to analyze a specific part of the software system. This step must achieve a balance in the way that the input does not exceed a specific size, but it must not lose its expressiveness by narrowing it down too much.

Obtaining and Collecting Initial Code Clone and Change Coupling Information

The previous section addressed the problem which part of a software system should be analyzed. The goal was to minimize the number of files to a reasonable amount in a first step. The problem emerging now is that it is normally not known beforehand which files share code clones and are change coupled. When analyzing the change behavior of code clones and change couplings in the context of this thesis, the focus lies on file groups that share at least one clone *and* were checked-in at least two times together in the release history. To get an initial overview which files share code clones and are change coupled, it is necessary to apply the clone detection tool and the release history tool on the entire previously selected input. The resulting file groups must be narrowed down further, since not every code clone or change coupling group is equally adequate for an analysis as explained in Section 3.2.3.

3.2.3 Repetitive Steps

The framework steps explained in this section are all part of the analysis that leads finally to the metrics computation. These steps are all highly repetitive because they have to be performed for every selected candidate that is investigated.

Identification of Possible Clone and Coupling Candidates

As mentioned in Section 3.2.2, in the context of this framework only files that share at least one code clone *and* are checked-in at least two times together are of any interest. The question now arising is whether in a first phase files that are change coupled should be taken and then checked if these couplings were caused by code clones; or whether files sharing code clones should be chosen and then checked if these clones caused change couplings.

Nonetheless not every change coupling group or code clone is suitable for the study of the development of code clones over the evolution of a software systems:

- **Code Clone Detection**: Since change coupling is a phenomenon between different files, only those code clones are interesting that occur not only in one source file but at least in two or more files.[4] Contrary to the identification of suitable clone candidates used in [Geiger 2005] where code clones whose length varies strongly over the examined period are more interesting,[5] the length of a code clone fragment in this framework is secondary. A change of the length of a clone is not necessarily caused by a structural change in the cloned source code fragment. An example where a change in the length of a code clone does not reflect a structural change is given in Listing 3.1 and 3.2. This framework focuses on identifying structural changes using fine-grained source code change analysis.

- **Change Coupling Groups**: Fluri *et al.* defines change couplings in [Fluri *et al.* 2005] as:

  > *Two files exhibit a change coupling if they are commonly committed, e.g., at the same time, by the same author, and with the same modification description.*

  Of special interest are coupling groups that recur often, because coupling groups appearing only once are probably accidental. Fluri *et al.* introduces a more formal concept to detect frequent change coupling groups in [Fluri *et al.* 2005].[6] The essence of their concept for this framework is: First different coupling groups can occur different times. Second a file can appear in more than one coupling group. The presence of change coupling sub-groups[7] forms a loop that leads to more and more fine-grained analysis of change coupling groups.

Tracking Code Clones over Time

After having identified a group of files as a suitable clone or coupling candidate, those files are to be tested for structural changes within code clone fragments. This rises the question which time frame of file revisions will be considered when evolving code clone changes. One can be interested in a complete overview and include every revision of each file. Another possibility would be to just look at the most recent file revisions to get an idea of the newer system evolution. There is no finite answer to this question. The choice depends on the purpose of the clone change behavior analysis. In regard with the research goals of the thesis we must include at least those revisions that represent change couplings to decide if a certain change coupling was due to changes in a cloned source code fragment. A larger time frame to track code clones gives more detailed insights of the state of a system, but it also increases the volume of data to examine.

Analyzing Data from Tracking Process

In the previous step all data of changes in code clones and change couplings were obtained. That data is used in this step as input to statistical analysis, *e.g.*, simple linear regression analysis to test whether code clones and change couplings exhibit a correlation. Such a correlation has been long time assumed, and its investigation was at the core of [Geiger 2005]: ”*The more code clones a group of files shares, the more they are change coupled.*” In a second step we use the data to compare, assess, and identify critical file groups and code clones.

Implement Refactoring Measures

The research goal of this thesis is not only to examine relation between code clones and change couplings generally , but also to provide a mechanism to programmers, developers and software

[4]Note: In general it can be useful to also look at code clones that appear only in one file to reason about the state of a software system.

[5]Cf. Section 4.2.3 *Identification of Suitable Clone Candidates* in [Geiger 2005]

[6]Section 3.5 *Detecting Frequent Change Coupling Groups* in [Fluri *et al.* 2005]

[7]According to [Fluri *et al.* 2005]: A change coupling group g is a change coupling sub-group if $g \subseteq h$.

engineers to identify problematic files (cf. Section 1.2). If the previous section reveals a problematic correlation, then it would be necessary to examine files that show a high rate of change couplings due to changes in the code clone fragment and – if necessary – eliminate the code clones. In best case such a refactoring leads to decrease of change couplings, and thus reduces the effort and costs of future maintenance of a software system.

3.3 Implementation of Step 1 - 3

In Section 3.2 the approach developed in this thesis was presented on an abstract level. The ambition was to introduce it as a whole, and to give an idea about the meaning and the contribution of each step within the framework.

The next task is to implement and adapt the framework to the specifications and requirements of this thesis. It is explained how each step was designed, and why it was designed that way. Moreover we see, that depending on each design decision new questions and problems arise. The presentation of our implementation is split on the rest of the thesis as followed:

- This section covers step 1 - 3 and is concerned with the selection of all necessary tools and technologies.
- Section 4 presents the implementation of the tool CLONEANALYZER according to step 4.
- Section 5 elaborates steps 5 - 10 by the means of two case studies.

3.3.1 Coarse Idea of the Target Software System

Following the "*Diploma Thesis Specification*" the approach must be evaluated by means of a C / C++ and a Java open source project [Giger 2006]. This implies that the resulting tool must be able to investigate code clone changes and change couplings for source code written in C / C++ and Java. Another constraint is that the tool in step 4 of the framework is to be realized as a Plug-In for Eclipse.

3.3.2 Choosing the Environment, Tools and Technologies

Following step 2 in Section 3.2.1 a *Code Clone Detection Tool*, a *Source Code Change Distilling Tool* and *Release History Tool* are necessary:

Code Clone Detection Tool : The detection of code clones is a widely discussed topic and subject to many research activities. The primary goal of this thesis is neither the investigation of the various approaches to code clone detection nor the evaluation of the existing code clone detection tools. A detailed evaluation of several approaches to code clone detection and detection tools is presented in [Geiger 2005]; the tools were evaluated and judged by 7 criteria:

- Language Support: What languages are supported for code clone detection by the tool?
- Input Size: What maximum input size is the tool capable to process?
- Comparison of the Candidates: When comparing the clone candidates of different tools do they roughly detect the same sets of code duplications? If not and the returned sets are largely disjoint, more than one detection tools should be used and the results then combined to get a more complete overview of the code clones in the target software system.

- User Interface: Does the tool provide a simple way to access its functionality?
- Output: Are the detected clone candidates displayed in a understandable visualization? Is the output provided in a log file for later reference and automated processing?
- Recall: Does the detection tool have a high recall value? (see appendix A for *Recall*)
- Precision: Does the detection tool have a high precision value? (see appendix A for *Precision*)

Other relevant criteria in the context of this thesis such as an open accessible API or platform independence were already discussed in Section 3.2.1.

Relying on the evaluation results in [Geiger 2005] the CCFinder was chosen for the implementation and obtained with the permission of the developers at Osaka University.[8] Contrary to [Geiger 2005] the newer version of the CCFinder *CCFinderX* is used. The CCFinderX is shipped along with a powerful graphical user interface (GUI) front end called GemX. GemX is written in SWT and aimed to become a Plug-In of Eclipse in the future. Whenever talking of the *CCFinder* in the remainder of the thesis, this refers to the new version CCFinderX.

Using the CCFinder as the code clone detection component for the implementation is not without problems:

- The CCFinder runs only on Windows XP 32-bit. Thus the implementation of a fully integrated and platform independent tool is not possible anymore.
- Neither the source code nor an API to access the clone detection functionality directly is provided. Since the output log file of the CCFinder is formatted in a clearly and well documented way, the clone detection results can be obtained by parsing this log file.

Despite all these problems the CCFinder is a very good choice, since it supports C/C++ as well as Java, and it is able to process a input size up to 2.6 million lines of code [Geiger 2005]. Another advantage is that knowledge about the CCFinder and some software artifacts of [Geiger 2005] can be reused.

Change Distilling : As shown in Section 3.2.1 a way to detect source code changes that meets the claim of *fine-grained analysis of source code changes* is necessary. Most mechanisms use some kind of textual differencing algorithms to detect changes in source code files. Since textual differencing algorithms are limited to line-level granularity, several limitations arise. An important one in our context is, that the changed source code entity, *e.g.*, method, loop statement, declaration etc. is not determined. The identification of the changed source code entity is necessary when distinguishing between syntactical and non syntactical changes.

Listings 3.1 and 3.2 show a textual change in a simple code fragment. The change in the source code is caused by some additional comment. Although the change takes place within the textual source code range of the method *addValues*, it reflects not a syntactical change as the source code entity itself, the method, did not change.

```
public int addValues(int a, int b){
      return a+b;
}
```

Listing 3.1: Example Source Code 1

[8]CCFinder official Site: http://www.ccfinder.net/

```
public int addValues(int a, int b){
      //add input values and return sum
      return a+b;
}
```

Listing 3.2: Example Source Code 2 with Additional Comment

Even *pretty print* source code changes such as indents or added / removed blank lines state a textual change and are reported when detecting source code changes on pure textual basis. Source code Listings 3.3 and 3.4 clarify the problem: The blank lines 2 and 4 are removed from Listing 3.3 to 3.4, whereas Listing 3.4 contains an additional indent on line 2. As in the above example the change happens within the textual range of the method *addValues*. Since the method itself has not been changed, this is not a syntactical change and should therefore not be considered in *fine-grained analysis of source code changes* – or it must be at least possible to filter such non-structural changes.

```
1 public int addValues(int a, int b){
2
3 return a+b;
4
5 }
```

Listing 3.3: Example Source Code 3

```
1 public int addValues(int a, int b){
2       return a+b;
3 }
```

Listing 3.4: Example Source Code 4 with an Additional Indent and Blank Lines Removed

These limitations can be addressed when source code changes are mapped to changes in abstract syntax tree (AST) representations of source code. Fluri and Gall give a detailed approach for comparing AST in their work [Fluri and Gall 2006]. The smallest entities used in this approach are normal statements.[9] They develop the Eclipse Plug-In ChangeDistiller implementing a source code change detection algorithm based on abstract syntax tree comparison. The ChangeDistiller compares regular, labeled, language independent trees.

We chose the ChangeDistiller to detect source code changes in our implementation because it is language independent and detects changes to a deep level of granularity; realized as Eclipse Plug-In the ChangeDistiller fits perfectly in this thesis for which the envisioned outcome must be an Eclipse Plug-In.

The ChangeDistiller only performs the tree comparison, but it does not offer a functionality to obtain the abstract syntax tree of source code. The ChangeDistiller uses the parser of the *Java Development Tool* (JDT) to construct the AST of a Java source file. The AST for source code files written in C / C++ is generated by the *C/C++ Development Tool* (CDT).

[9]More coarse-grained structure statements such as loop or control statements are treated separately.

Release History Tool : Version control systems, *e.g.,* CVS contain a large volume of historical data about a software system collected during its lifetime. Such information can be used to calculate change couplings as shown in [Fluri *et al.* 2005]. The concept of extracting information from version control and bug tracking systems is presented in [Fischer *et al.* 2003]. Based on this idea the Eclipse Plug-In EVOLIZERBASE was implemented by the s.e.a.l.-group at the Department of Informatics at the University of Zurich.[10] The EVOLIZERBASE accesses the information in CVS repository of a system and imports it into a MySQL- or PostgreSQL database using the additional object-oriented layer Hibernate[11] in between. The EVOLIZERBASE has already a well defined API for querying the version data stored in the database and is thus easy to use.

For the rest of this implementation we use theEVOLIZERBASE to gain information about the release history and change couplings. A disadvantage of EVOLIZERBASE is that it is currently only implemented for the import of data out of CVS. The import of data contained in other version control systems[12] is not yet possible.

3.3.3 Defining a Set of Metrics

This section provides a definition of measures and metrics. It brings out a concrete elaboration of the framework step 3 presented in Section 3.2.1. Defining metrics is crucial for the quality and usability of any analysis process, because they form the basis of any calculated results that are ultimately presented to the user. Based on the metrics computation the user can reason about the state of his software system and may implement some necessary refactoring measures. Saying that even the most sophisticated approach for fine-grained analysis of source code changes and code clones itself is useless if its results are not presented in a informative and meaningful way.

When choosing measures and metrics this section focuses on the envisioned outcomes and goals. As stated out in Section 1.2 this thesis aims mainly at two goals: First investigating the relation between change couplings and code clones, and whether they exhibit a correlation. Such a (positive) correlation has been taken for granted so far. This assumption emerges because of the *duplicated character* of code clones. The reuse effect achieved by introducing code clones is not grounded on artful and smart design of a software artifact but on a pure textual basis. Suitable source code is cloned mostly by copy-paste – and slightly adapted to its new domain.

Therefore a correlation tries to quantify the severity of the impact of code clones. This thesis uses statistical regression analysis to investigate the relation between code clones and change couplings. The purpose of a regression analysis is to test existing measures for a correlation and dependencies. This fact imposes three preconditions to a regression analysis:

1. No statistical method or analysis is able to prove causality directly. To interpret a statistical correlation as a real causal correlation, logical and domain related considerations are necessary. While the first step – the statistical correlation analysis – is relatively unproblematic, the second step requires extensive knowledge of the research field. Such an interpretation must be carried out systematically and carefully. A strong correlation should therefore be seen as an indicator of a possible dependency between two measures. Interpreting it causally is a delicate challenge and may lead to a sham-correlation that is – concerning the contents – not justified [Fahrmeir *et al.* 2001].[13] The assumption at the beginning of this

[10] Department of Informatics, University of Zurich: http://www.ifi.unizh.ch

[11] http://www.hibernate.org

[12] Overview of several version control systems is given on: http://better-scm.berlios.de/comparison/comparison.html

[13] Some examples for sham-correlations and logical fallacies can be found on http://en.wikipedia.org/wiki/Correlation_implies_causation_%28logical_fallacy%29. Especially read the funny example from *The Simpsons*!

section that a change in a source code fragment automatically leads to changes in its duplications because of its scruffy design can be a reasonable logical thought.

2. As it is impossible to test the entire population of code clones and change couplings, the input data for the regression analysis must picture a random and representative sample of all code clones affected and change coupled files.

3. The case studies will use simple linear regression for the statistical analysis. A simple linear regression uses a linear function $y = a \times x + b$ to describe the correlation between two measures. Where x is the independent measure and y depends on x. An input sample is described precisely enough if it has a correlation coefficient close to 1.

If the statistical regression analysis can indeed verify a (positive) correlation, this would fortify the *bad smell* character of code clones. This is were the second goal comes into play. As mentioned before the metrics then must be visualized and presented in a meaningful way to allow users to identify critical code clones and possible refactoring candidates. The extraction of all relevant data, the computation of measures and metrics, and the visualization should only require a minimum of user intervention. Section 4.2 therefore implements a tool called CLONEANALYZER to automate this process.

Measures and Metrics

Since the investigation of code clone changes and change couplings is a matter of time, the term *File-History* is introduced: *A File-History is a collection of revision vectors for every file that is a member of this File-History*. For instance, the files *X*, *Y* and *Z* belong to the same File-History. Assuming for *X*: Revisions 1.96, 1.97, 1.98 were chosen; for *Y*: Revisions 1.2, 1.3 and 1.4; for *Z*: Revisions 1.11 and 1.12, the File-History can be represented as the following matrix:

$$\begin{array}{c} X \\ Y \\ Z \end{array} \begin{bmatrix} 1.96 & 1.97 & 1.98 \\ 1.2 & 1.3 & 1.4 \\ 1.11 & 1.12 & \end{bmatrix} \begin{array}{l} \text{revision vector 1} \\ \text{revision vector 2} \\ \text{revision vector 3} \end{array}$$

Or in a more formal way:

$$H = \{F_{1_1 \ldots 1_x}, F_{2_1 \ldots 2_y}, \ldots, F_{n_1 \ldots n_z}\}$$

whereby *H* is a File-History and F_n is the *nth* file in this File-History *H*. The subscript indexes $n_1, ..., n_x$ represent the revision vector of file F_n. The revision vectors are temporally ordered in ascending manner. Their subscripts thus reflect the temporal sequence of the revision numbers. For two subscripts n_i and n_j of file n and with $i < j$ this means that the revision number at position i represents an earlier revision of file n than the revision represented by the revision number at position j. The cardinality $|\cdot|$ of a File-History expresses the number of files forming the File-History. In the above example the cardinality is 3 for *X*, *Y* and *Z*.

The matrix of a File-History reminds of the matrix of coupling clusters in [Fluri *et al.* 2005]. Both matrices describe similar circumstances. But there is an important difference. While a revision vector of a coupling cluster exhibits a common check-in of certain files and is interpreted vertically in the matrix, the revision vector in a File-History matrix illustrates an excerpt of the release history of its file. Thus revision vectors are interpreted horizontally on the level of files.

This implies that revision vectors of a File-History matrix do not have necessarily the same size, since files usually have different numbers of revisions.

Having set up the term *File-History*, the *TemporalCloneGroup* can be created. Meaning all revisions of the files in a File-History are subject to a code clone detection run:

$$TemporalCloneGroup(H) = \{c_1, c_2, \ldots, c_j, \ldots, c_{n-1}, c_n\} = DetectionRun\,(H)$$

whereby a $TemporalCloneGroup$ (TCG) is a set of code clones c_j resulting from applying a clone detection run on a File-History H. $DetectionRun$ is an abstract synonym of a function to detect code clones in a given set of source files.

Based on TCG and File-History the *Temporal Appearance Matrix* – designated by the greek letter τ – for each code clone can be drawn. For instance, tcg is a concrete TCG, code clone $c_j \in tcg$, and *h* is a concrete File-History. The *Temporal Appearance Matrix* τ for c_j looks as following:[14]

$$\tau(c_j) \begin{bmatrix} F_{1_1} & F_{1_2} & \cdots & F_{1_x} \\ F_{2_1} & F_{2_2} & \cdots & F_{2_y} \\ \vdots & \vdots & \ddots & \vdots \\ F_{n_1} & F_{n_2} & \cdots & F_{n_z} \end{bmatrix} \begin{array}{c} \text{appearance revision vector 1} \\ \text{appearance revision vector 2} \\ \vdots \\ \text{appearance revision vector n} \end{array}$$

whereby $F_n \in h$. The cell F_{n_i} means that a code clone c_j was detected in the revision number i in the revision vector of file n of its File-History h. Using such matrices we can exactly locate a specific code clone in different revisions of source code files.

In a next step we include possible structural changes of a code clone between subsequent revisions of an affected file

$$StructuralChanges(c_j) = SourceCodeDifferencer(\tau(c_j))$$

is introduced. Whereby *StructuralChanges* (SC) is the set of all structural changes of a code clone c_j. *SourceCodeDifferencer* is an abstract synonym of a function to detect source code changes between different file revisions. The structural changes are obtained by applying the source code differencing function on the τ of a code clone. Based on SC a temporal appearance matrix can be enriched with information about the structural changes of a code clone. For instance, *sc* is a concrete SC for code clone c_j. $sc_{n_i} \in sc$ then maps to the cell at position (n, i) in the temporal appearance matrix $\tau(c_j)$. Meaning, that for sc_{n_i} there has been a structural change in code clone c_j in file n of a File-History between the revision number at the position i and its predecessor $i-1$ in the revision vector of file n.

[14]This step implies that the results of the detection run not only include all code clones but must also offer a way to determine in which versions of a file a certain code clone appears. The CLONEANALYZER uses the unique clone ID of the CCFinder to track code clones (cf. Section 4.2.2).

Let us clarify this situation with an example using the above File-History with the source code files *X*, *Y* and *Z* and name it *fh*. Again its representation as matrix looks as following:

$$\underbrace{\begin{array}{c} X \\ Y \\ Z \end{array}\begin{bmatrix} 1.96 & 1.97 & 1.98 \\ 1.2 & 1.3 & 1.4 \\ 1.11 & 1.12 & \end{bmatrix}}_{\text{Matrix Representation for } fh}$$

Then fh is subject to a clone detection resulting in the temporal clone group t containing three code clones:

$$t(fh) = \{c_1, c_2, c_3\} = DetectionRun(fh)$$

Based on these results the temporal appearance matrices can be constructed for all code clones of t. Assuming the clone detection shows that code clone c_1 occurs in revisions 1.96, 1.97 and 1.98 of file *X*; in revisions 1.11 and 1.12 of file *Z* the temporal matrix for clone c_1 looks as following:

$$\underbrace{\begin{bmatrix} 1.1 & 1.97 & 1.98 \\ 1.10 & 1.12 & \end{bmatrix}\begin{array}{c} X \\ Z \end{array}}_{\tau \text{ for } c_1}$$

To indicate changes in the source code fragment of code clone c_1 a differencing algorithm is applied on $\tau(c_1)$:

$$sc = \underbrace{\{sc_{1_3}, sc_{2_2}\}}_{\text{StructuralChanges for } c_1} = SourceCodeDifferencer(\tau(c_1))$$

sc contains two structural changes for clone c_j; one at the cell $(1,3)$ and the other at the cell $(2,2)$ in $\tau(c_j)$. This leads to the conclusion that code clone c_1 exhibits structural changes from revision 1.97 to 1.98 in file X and from revision 1.11 to 1.12 in file Z. Using now change coupling information one could tell whether the revision step 1.97 to 1.98 in file *X* and the revision step 1.11 to 1.12 in file *Z* also represent a change coupling of these two files. If so, this change coupling was caused by a change in the source code fragment of code clone c_1.

For a short recapitulation: Up to this point a formal way has been showed to describe the change behavior of a code clone by the means of a temporal appearance matrix τ enriched with information from a source code differencing algorithm and data from change coupling calculation. Based on this approach, measures and metrics can finally be set up that will be useful for the statistical regression analysis:

- γ is defined as the cardinality of a *TemporalCloneGroup(H)*

$$\gamma = |TCG|$$

and refers to the fact of how many code clones the files of a File-History H share over time.

- δ expresses the number of times the files of a File-History were committed together and is defined as

  $$\delta(H) = CouplingCalculator(H)$$

  whereby *H* is a File-History. *CouplingCalculator* is an abstract synonym of a function to calculate the change couplings of File-History H.

- ε expresses the number of change couplings caused by changes in code clone c_j and is defined as

  $$\varepsilon(c_j) = |CouplingCalculator(SourceCodeDifferencer(\tau(c_j)))|$$

  whereby $c_j \in tcg$ and *tcg* is a TCG.

- σ expresses the total sum of change couplings caused by all code clones of a temporal clone group tcg and is defined as

  $$\sigma = \sum_{i=1}^{\gamma} \varepsilon\left(c_i\right)$$

  whereby $c_i \in tcg$ and *tcg* is a TCG.

- ϱ expresses the ratio between the number of change couplings caused by all code clones and the total number of change couplings and is defined as

  $$\varrho = \frac{\sigma}{\delta}$$

  For example a value $\varrho > 0.5$ indicates that more than 50% of the change couplings of the File-History for which σ and δ have been calculated were caused by changes in code clones.

Classification of Code Clone Change Behavior

When investigating the relation between change couplings and code clones, a classification to describe the change behavior of a clone over time is desirable. An approach to classify code clones over several revisions by means of five types (*Type 0 - Type 4*) was introduced in [Geiger 2005].[15] The classification uses the ordinal comparison of the *CloneCoverage* value for two files between subsequent file versions (cf. appendix A for *CloneCoverage*). A shortcoming of this approach is its focus on file pairs. It uses a metric that describes a property between two files. However a code clone is not necessarily a matter between only two files, but in many cases more than two files are affected. This thesis shifts its focus on the code clone itself to introduce a classification. Its goal is to describe the changes of a code clone in the context of the release history and change couplings of the affected files using *Clone Change Types* (CCT). There are four different CCT to describe the circumstances under which a code clone changes (*CCT 0 - CCT 3*).

As shown below the advantage of this approach is that it classifies each single change of a code clone. The impact or severity of a code clone can be judged by an aggregate overview of its change types. Since assigning source code changes of a code clone to a certain change type is a

[15] Cf. section 4.2.3.1 *Classification of Code Clones over Several Versions* in [Geiger 2005]

matter between two subsequent revisions of a file, a code clone can have different Clone Change Types over time.[16]

Let c be a detected clone, $F(c) = \{f_1, \ldots, f_n\}$ the set of all files that are affected by c and $f_j \in F$ is a single file affected by *c*. $CTS(c, f_j) = \{cct_1, \ldots, cct_x\}$ is then the set of all *Clone Change Types* of clone *c* over the release history of file f_j. $CC(F(c))$ is the set of change couplings of all files affected by clone *c*. Thereafter a *Clone Change Type* is a set of three properties p_{0-2}:

$$CCT = \{\underbrace{i}_{p_0}, \underbrace{i+1}_{p_1}, \underbrace{hasChanged}_{p_2}\} \in CTS(c, f_j)$$

where i and $i+1$ are the revisions i and $i+1$ of file f_j.

The property $hasChanged \in \{TRUE, FALSE\}$ indicates whether code clone *c* has structurally changed (*hasChanged = TRUE*) between version *i* and *i+1* of file f_j or not (*hasChanged = FALSE*). Finally based on this set up, the four *Clone Change Type* can be characterized by concrete parameter values of the properties set:

- **Change Type 0** $= \{i, i+1, hasChanged\} \wedge i+1 \notin CC(F(c)) \wedge hasChanged = FALSE$
  Meaning the step from version *i* to *i+1* does not represent a change coupling of the affected files, and the code clone has not structurally changed.
- **Change Type 1** $= \{i, i+1, hasChanged\} \wedge i+1 \in CC(F(c)) \wedge hasChanged = FALSE$
  Meaning the step from version *i* to *i+1* represents a change coupling of the affected files, but there has not been a structural change in the code clone.
- **Change Type 2** $= \{i, i+1, hasChanged\} \wedge i+1 \notin CC(F(c)) \wedge hasChanged = TRUE$
  Meaning the code clone structurally changed, but the step from version *i* to *i+1* does not represent a change coupling.
- **Change Type 3** $= \{i, i+1, hasChanged\} \wedge i+1 \in CC(F(c)) \wedge hasChanged = TRUE$
  Meaning the code clone has changed in a file during a change coupling.

A *change coupling due to changes in a cloned source code fragment* occurs when a code clone changes in *all* files at the same time. More formally: For a given code clone c there is a $F(c)$ with the file index j and a $CC(F(c))$ with the change coupling index k.

$$\forall j : \exists i : cct_i \in CTS(c, f_j) \wedge cct_i = ChangeType3 \wedge p_1 \mapsto cc_k \wedge cc_k \in CC(F(c))$$

Note: p_1 is the property at index 1 in the CCT property set describing the version number $i+1$ of file j.

Visualization

This section defines measures and metrics to investigate the relation between code clones and change couplings as well as a classification to describe the change behavior of a code clone. As mentioned at the beginning not only metrics are important but also their visualization. They must be presented to the user in a meaningful way allowing to draw conclusions about the evolutionary state of a software system in the context of code clones and change couplings. The

[16]The same is true for the classification introduced in [Geiger 2005].

CLONEANALYZER presented in Section 4.2 implements the metrics developed in this section as well as a visualization.

Limitations of the Metrics

The significance and expressiveness of these metrics and measures rely on the available amount of data, because evolving code clones over time is a temporal process. For a relatively young software project the period of release history is short, and the expressiveness decreases.

Another limitation is the comparability. The measures and metrics do not account for the differences between software project such as programming languages, workflow processes, project size etc. Using solely these metrics to compare the quality of several software systems or projects in the context of code clones and change couplings is problematic.

3.4 Final Remarks

In this chapter we presented a framework to meet the research goals. In a first step in Section 3.2 the framework was described as an abstract recipe for everyone interested to develop his own implementation. This recipe however is not set in stone and it may be modified and adapted as the case arises. It does not focus on any concrete tool, technique or mechanism. It is rather a guideline that shows all necessary steps, defines a minimum set of requirements, names possible problems and solutions and gives advices.

Section 3.3 then showed concrete implementations for steps 1 - 3 of the framework: *Coarse Idea of the Target Software System*, *Choosing the Environment*, *Tools and Technologies*, and *Defining a Set of Metrics*. The focus of that section therefore shifted to a more detailed level of view by referring to concrete solutions, tools and technologies: In Section 3.3.2 the CCFinder was chosen as the code clone detection tool; the CHANGEDISTILLER is used to detect source code changes in a code clone, and the EVOLIZERBASE retrieves version history data from the CVS repository a software system. Section 3.3.3 introduced the term *File-History* to describe a set of certain files and revision numbers associated to those files. The *Temporal Appearance Matrix* τ allows to determine exactly in a formal way in which revision of a file a code clone appears throughout the entire File-History. Based on τ and the File-History a set of metrics was defined allowing to investigate the relation of code clones and change couplings. A system consisting of 4 *Clone Change Types* is used to classify the change behavior of code clones over time.

Chapter 4 covers step 4 *Implementing the Tool* and presents the CLONEANALYZER. It provides an automated way to process steps 5 - 10 of the framework. We will use the CLONEANALYZER in the case studies of Chapter 5.

Chapter 4

The Clone Analyzer

4.1 Introduction

This section covers the implementation of an Eclipse Plug-In according to step 4 of the framework. It uses code clone data, release history information from the EVOLIZERBASE, and results from the AST differencing algorithm of the CHANGEDISTILLER to calculate the set of metrics defined in Section 3.3.3. In the remainder of this thesis the tool is called CLONEANALYZER. It serves as starting point for Chapter 5 were steps 5 - 10 of the framework are applied to concrete case studies. Because of its realization as Eclipse Plug-In many Eclipse specific concepts and terms are used in this section. For a better comprehensibility a short description of the Eclipse-Platform and its Plug-In architecture is presented, before the details of the CLONEANALYZER are worked out. Section 4.3 at the end of this Chapter gives a detailed manual how to use the CLONEANALYZER.

4.2 Implementing the Clone Analyzer

4.2.1 Eclipse

Since version 3.0 Eclipse itself does not constitute anymore an IDE[1] for Java or any other programming language, but is a collection of components forming together the extensible Eclipse-Platform [Beaton 2006]. The Eclipse-Platform is used as an integration point for other tools and applications. By adding Java (*e.g.,* JDT) or C / C++ (*e.g.,* CDT) development components, the Eclipse-Platform turns into a Java respectively a C / C++ IDE. The basic Eclipse-Platform, the JDT and the Plug-In Development Environment (PDE) build the Eclipse Software Development Kit (Eclipse SDK) initially shipped when "downloading Eclipse". The functionality is thus not provided by Eclipse itself but by specialized Plug-Ins built on top of the Eclipse-Platform. The Eclipse-Platform manages the integration of individual tools into one single environment providing a powerful, consistent and extensible experience for its users. The Eclipse-Platform is responsible for loading, executing and unloading the Plug-Ins. Figure 4.1 gives an overview of the Eclipse Platform and its main components. Also the "*glue*"-character of the CLONEANALYZER is clearly outlined: By plugging into the Eclipse-Platform it accesses the functionalities of other installed Plug-Ins and uses them for its own services.

- **Runtime**: The Runtime-Component of the Eclipse-Platform is responsible for detecting and loading available Plug-Ins and managing the Plug-In infrastructure.

[1]Up to version 2.1 Eclipse itself was designed as an IDE.

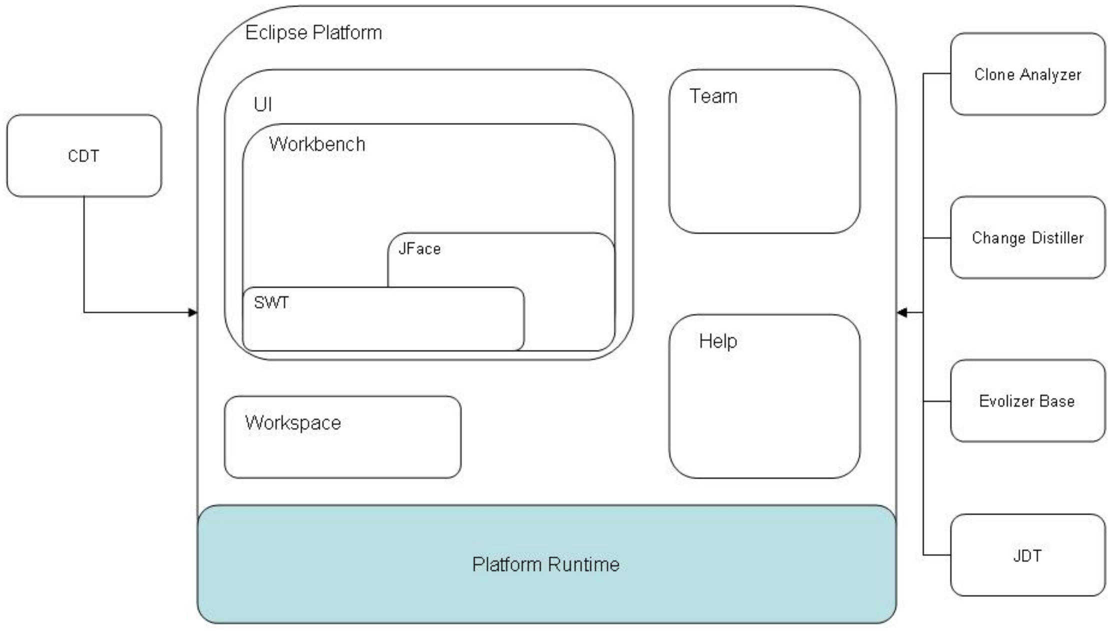


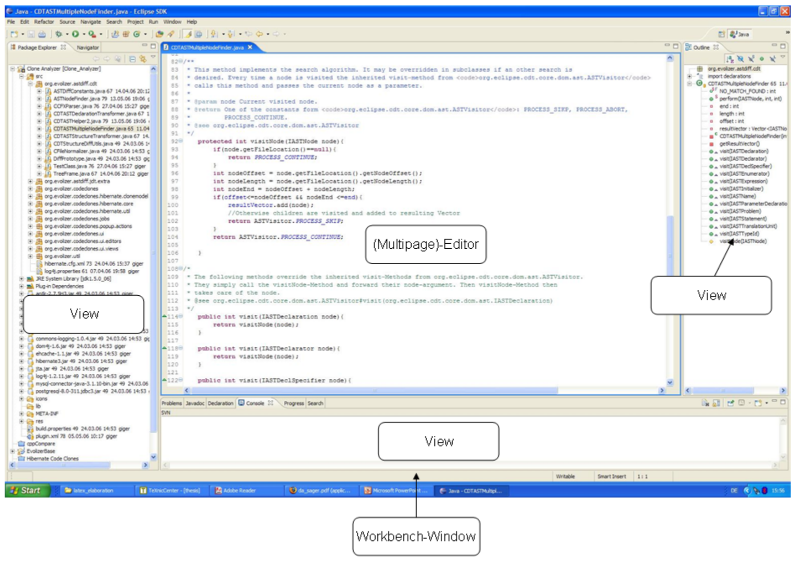
Figure 4.1: Eclipse-Platform Architecture and Plug-Ins, Imitation of [Beaton 2006]

- **Workspace**: A workspace instance manages the top-level projects. It is possible to have several workspaces.
- **SWT**: The Standard Widget Toolkit (SWT) provides a common OS-independent API for widgets and graphics. Its implementation allows a tight integration with the underlying native window system.
- **JFace**: JFace is part of the Platform UI components. Its design aims at window system independence and cooperation with SWT.
- **Workbench**: The Workbench defines the UI paradigm of Eclipse. The Workbench represents the part of the interface that allows a user ”*to work on his projects*”. Figure 4.2 shows the Workbench window and some of its inner components.[2]

4.2.2 Design and Architecture of the Clone Analyzer

This sections explains the design and the architecture of the CLONEANALYZER. The tasks described in Section 3.2.1 are divided into specialized classes as members of several Java-Packages. In the following these packages and their responsibilities are shown. Where useful for a better understanding, some classes as well as their interfaces and functionality are explained in a more exhaustive way. For a complete overview of all classes and their functionality reference to the Javadoc is recommended. All classes are written in Java 5.0 (*Tiger*-Release 1.5) and use the Generics feature. Compiling and running the CLONEANALYZER requires a Java 5.0 compatible platform.

[2][Springgay *et al.* 2002] gives a more detailed introduction to the of Eclipse UI Guidelines.


Java - CDTASTMultipleNodeFinder.java - Eclipse SDK
CDTASTMultipleNodeFinder.java
(Multipage)-Editor
View
View
View
Workbench-Window

```
/**
 * This method implements the search algorithm. It may be overridden in subclasses if an other search is
 * desired. Every time a node is visited the inherited visit-method from <code>org.eclipse.cdt.core.dom.ast.ASTVisitor</code>
 * calls this method and passes the current node as a parameter.
 *
 * @param node Current visited node.
 * @return One of the constants form <code>org.eclipse.cdt.core.dom.ast.ASTVisitor</code>: PROCESS_SIKP, PROCESS_ABORT,
 *         PROCESS_CONTINUE.
 * @see org.eclipse.cdt.core.dom.ast.ASTVisitor
 */
    protected int visitNode(IASTNode node){
        if(node.getFileLocation()==null){
            return PROCESS_CONTINUE;
        }
        int nodeOffset = node.getFileLocation().getNodeOffset();
        int nodeLength = node.getFileLocation().getNodeLength();
        int nodeEnd = nodeOffset + nodeLength;
        if(offset<=nodeOffset && nodeEnd <=end){
            resultVector.add(node);
            //Otherwise children are visited and added to resulting Vector
            return ASTVisitor.PROCESS_SKIP;
        }
        return ASTVisitor.PROCESS_CONTINUE;

    }

/*
 * The following methods override the inherited visit-Methods from org.eclipse.cdt.core.dom.ast.ASTVisitor.
 * They simply call the visitNode-Method and forward their node-argument. Then visitNode-Method then
 * takes care of the node.
 * @see org.eclipse.cdt.core.dom.ast.ASTVisitor#visit(org.eclipse.cdt.core.dom.ast.IASTDeclaration)
 */
    public int visit(IASTDeclaration node){
        return visitNode(node);
    }

    public int visit(IASTDeclarator node){
        return visitNode(node);
    }

    public int visit(IASTDeclSpecifier node){
```

Figure 4.2: Workbench-Window of Eclipse 3.1.2 on Windows XP

Clone Analyzer Package Structure :

- `org.evolizer.astdiff.cdt`: Classes in this package are responsible for implementing the existing abstract syntax tree differencing algorithm for C / C++. They parse C / C++ source code using the AST-Builder from the CDT and construct regular labeled trees as input to the differencing algorithm of the CHANGEDISTILLER. The package includes classes for finding nodes in the abstract syntax tree for a given offset and length within a C / C++ source code file.
- `org.evolizer.astdiff.jdt.extra`: Package containing classes that provide additional functionality for the Java differencing algorithm of the CHANGEDISTILLER specially needed in this project.[3]
- `org.evolizer.codeclones`: This package and its classes make it possible to detect and evolve code clones over time and extract changes in the source code fragments of code clones between subsequent file revisions.
- `org.evolizer.codeclones.hibernate.clonemodel`: Because the detection of code clones is a very time and resource consuming task, it was decided to store the results of a clone detection run, processed by the CCFinder, in a MySQL-database using the object-oriented layer Hibernate. This package includes the Java-Bean classes for Hibernate as well as their corresponding Hibernate-Mapping `xml` files.
- `org.evolizer.codeclones.hibernate.core`: Package providing functionality to retrieve and insert code clone information from and into the MySQL-database via Hibernate.
- `org.evolizer.codeclones.hibernate.util`: Auxiliary classes for the use of Hibernate and additional Query-API for the EVOLIZERBASE.
- `org.evolizer.codeclones.jobs`: Detecting code clones and identifying source code changes in code clones over a release history are time consuming tasks. Such tasks

[3]The CHANGEDISTILLER was originally only implemented to detect changes in Java source code.

are processed in subclasses of `org.eclipse.core.runtime.jobs.Job` to avoid the current thread to be blocked by those tasks while their are done. The Job-Classes themselves do not provide any functionality, the mainly wrap classes with time consuming tasks in their `run`-Method.

- `org.evolizer.codeclones.popup.actions`: Classes for the context menu actions when *right-clicking* on resource entities in the package or resource explorer of the Eclipse-Workbench window.
- `org.evolizer.codeclones.ui.editors`: The user interface of the CloneAnalyzer is integrated in the Eclipse Workbench by Multipage-Editors and Views. This package contains the classes for creating the Multipage-Editors and rendering their content.
- `org.evolizer.codeclones.ui.views`: This package contains the classes for creating the Views and rendering their content.
- `org.evolizer.codeclones.util`: Contains auxiliary classes used throughout the entire project.

As defined in Section 3.2.1 the CloneAnalyzer must be able to perform a minimal set of tasks to serve as a fully integrated tool and to minimize the manual workload. Following the realizations of those tasks in the implementation of the CloneAnalyzer are shown. If necessary and useful for understanding, concrete Java-Classes are referred to.

Code Clone Detection : An automated code clone detection is complicated, because there is no API to access the detection functionality of the CCFinder. It is possible to start the CCFinder, save its output manually in a log file, and then parse this file every time code clone information is needed. Still this is not a very convenient way for the user. To overcome this problem the CloneAnalyzer uses a workaround via the Java-API. It generates a Batch-File for starting the CCFinder with all necessary input arguments. The Batch-File is then executed by invoking the method `exec` of the class `java.lang.Runtime` that allows to start external processes via Java. The same workaround via external processes is used when converting the binary output file of the CCFinder into a text formated file.[4] Listing 4.1 shows an exemplary excerpt of the textual CCFinder output. Essential for the automated code clone detection in the CloneAnalyzer is the class `CCFXParser` in the package `org.evolizer.codeclones`. Its API provides functionality to create the batch file, start the CCFinder and transform the binary output of the CCFinder into a text file with the format as described in the Appendix B.1.

```
version: 10.1.5
format: pair_diploid
//CCFinderX command options
preprocess_script: cpp
source_files {
1 nsMathMLContainerFrame_1.cpp 4957
2 nsMathMLContainerFrame_2.cpp 4957
//more source files......
}
clone_pairs {
```

[4]The initial output of a code clone detection run is saved in a binary file. By starting the CCFinder with the *P*-Option in the commando line environment, the CCFinder reads the binary output file and transforms it into textual output [Kamiya 2006].

```
12 1.3750-3812 1.3798-3860
13 1.3750-3828 1.3782-3860
12 1.3750-3812 2.3798-3860
14 1.3766-3860 1.3750-3844
//more clone pairs.......
}
```

Listing 4.1: CCFinderX Output Processed with P-Option

Of special interest are the lines in the *source-file* section. Entries of this section denote all source code files that were subject to the clone detection by a unique ID, their file names and token lengths:

$$\underbrace{1}_{\text{File-ID}} \quad \overbrace{nsMathMLContainerFrame_1.cpp}^{\text{Filename}} \quad \underbrace{4957}_{\text{Token-Filelength}}$$

Lines within the *clone-pair* section show the positions of each code clone within all affected source code files. The CCFinder displays the detected clones using file pairs. Two files together with information about the position of code clones form a so called *Clone-Pair*:

$$\underbrace{12}_{\text{Clone-ID}} \quad \underbrace{\overbrace{1}^{\text{File-ID}} \; . \; \overbrace{3750}^{\text{Begin-Token}} \; - \; \overbrace{3812}^{\text{End-Token}}}_{\text{File 1 of Clone Pair}} \quad \underbrace{\overbrace{2}^{\text{File-ID}} \; . \; \overbrace{3798}^{\text{Begin-Token}} \; - \; \overbrace{3860}^{\text{End-Token}}}_{\text{File 2 of Clone Pair}}$$

A fully formal definition of the textual output of the CCFinder is given in [Kamiya 2005]. One problem of this format is its position information about code clone inside a source code file. The CCFinder uses a token stream based approach for detecting code clones instead of pure textual source code matching. The CCFinder transforms every input line of source code into a normalized stream / sequence of tokens. In a further sub step programming language specific rules are applied to the token stream. A token therefore is not a simple textual reflection of its origin, but a keyword, an identifier or some semantic meaning symbol. After the transformation process the definite token stream is investigated for code clones. A detailed description of how the token stream based approach in the CCFinder works and an overview of the language specific transformation rules for C++ are presented in [Geiger 2005].

The transformed token sequence of a source code file is then saved in a *Preprocessed File* with the file extension `CCFXPREP` in the same directory as the input source file.

For example: For `nsMathMLContainerFrame_1.cpp` the corresponding preprocessed file `nsMathMLContainerFrame_1.cpp.cpp.CCFXPREP` containing the token stream is created in the same directory.

Due to the token based approach the position of code clones is given by the begin and end token of its appearance within the token stream of an affected source code file. Positional data of code clones in terms of tokens is CCFinder specific and generally useless in combination with other tools. For the most cases values like *Begin-Line, End-Line* or *Offset, Length* are used to describe a certain range inside a source code file. The CLONEANALYZER addresses this problem by parsing the mentioned preprocessed files. Those files contain the exact location of every token in a source file.[5] This additional position data from the preprocessed files in combination with the original textual output is then transformed into a format where *Begin-Line* and *End-Line* or *Offset* and *Length* are used to describe the positions of code clones in a source file. Listing 4.2 shows an excerpt of such a positional transformed output:

```
preprocess_script: cpp
#begin{file description}
1 695 nsMathMLmoFrame.cpp
2 478 nsMathMLmtableFrame.cpp
//more source files....
#end{file description}
#begin{clone}
9 1 196,1 202,1 2 235,1 243,1
5 2 405,1 418,2 2 419,1 432,2
//more clone pairs....
#end{clone}
```

Listing 4.2: CCFinderX Output with Positional Transformed Clone Data

The `preprocess_script` line is taken over from the original output in Listing 4.1 and denotes the programming language of the files that were subject to the clone detection. The CCFinder uses this option to apply the right transforming rules when creating the normalized token stream of a source code file.

Of special interest are the lines of the *file-description* section. They have the same meaning as lines of the *source-file* section in the original CCFinder output (cf. Listing 4.1), except the length of a source file is specified by lines rather than by tokens:

$$\underbrace{1}_{\text{File-ID}} \qquad \overbrace{695}^{\text{Length by Lines}} \qquad \underbrace{nsMathMLmoFrame.cpp}_{\text{Filename}}$$

Lines within the *clone-pair* section display code clones and their occurrences in source files by means of a *Clone-Pair*:

$$\underbrace{9}_{\text{Clone-ID}} \; \underbrace{\overbrace{1}^{\text{File-ID}} \; \overbrace{196,1}^{\text{Beginline \& Lineoffset}} \; \overbrace{202,1}^{\text{Endline \&Lineoffset}}}_{\text{File 1 of Clone Pair}} \; \underbrace{\overbrace{2}^{\text{File-ID}} \; \overbrace{235,1}^{\text{Beginline \& Lineoffset}} \; \overbrace{243,1}^{\text{Endline \&Lineoffset}}}_{\text{File 2 of Clone Pair}}$$

Appendix B.1 gives a semi formal description of the format for such positional transformed files.

As shown in the package overview description, the CLONEANALYZER stores the results of code clone detection runs in a MySQL database using the object-oriented layer Hibernate in between. Figure 4.3 shows the Hibernate Java-Bean classes of the *Hibernate Code Clone Model* and their most important attributes. The model is used to make the results of a detection run persistent.[6] The goal was to map the data of the output log files of the CCFinder. The

[5]See [Kamiya 2005] for a fully formal description of the preprocessed file format.
[6]For a complete overview refer to the Javadoc of the Java-Classes forming the *Hibernate Code Clone Model*.

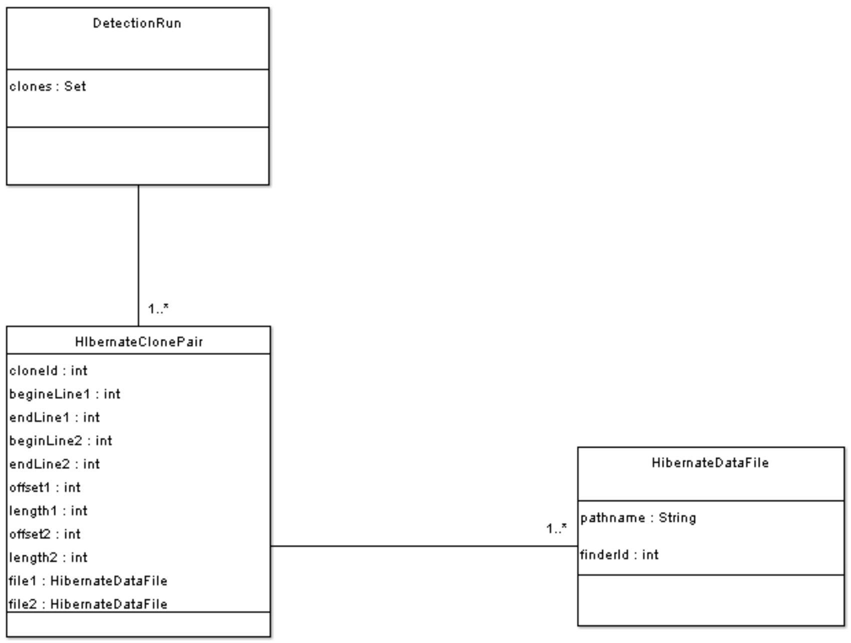


Figure 4.3: Simplified Hibernate Code Clone Model

model gives the possibility to easily query and browse through the results of code clone detection runs. It is therefore kept consciously simple and reflects the format of the positional transformed output log file as described in Appendix B.1 .The introduction of the class `DetectionRun` is necessary to clearly identify code clones among several detection runs in the database. An instance of this class can been seen as the logical equivalent of an output log file, since the CCFinder creates a log file for every detection run.

Finding the Corresponding Source Code Entity : Even though using some language specific meanings during the transformation process, code clones detected by the CCFinder are returned on a pure textual basis. A code clone is a character string whose normalized token sequence appears several times throughout the input source files. As seen in Section 3.3.2 it must be possible to break down a code clone to the source code entities that are covered by its textual range. As the CHANGEDISTILLER uses abstract syntax trees for its differencing algorithm the term *Finding the Corresponding Source Code Entity* turns into *Finding the Corresponding Abstract Syntax Tree Node*.

Because the CHANGEDISTILLER only performs the comparison of abstract syntax trees, the *CloneAnalyzer* and the CHANGEDISTILLER both rely on the parsers of the JDT and CDT for creating abstract syntax trees. Both the API of the CDT as well as the API of the JDT provide functionality to determine the exact textual position in a source code file for each node of the abstract syntax tree. Contrary to the CCFinder those APIs use *Offset* and *Length* to specify the position of such nodes. This aspect clearly points out why describing the position of code clones by the means of tokens is useless. The CLONEANALYZER transforms the original output of Listing 4.1 into the format of Listing 4.2.

The following example clarifies this situation: Supposing the CCFinder reports a code clone in the exemplary Java source code Listing 4.3 starting at Line 2 and ending at Line 9. Source code entities (and their nested entities) covered by the code clone are highlighted in light gray and consist of one *for*-Statement and two *Method-Calls*. The problem now emerging is to find the abstract syntax (sub-)tree that represents the best textual range of the code clone. The CloneAnalyzer addresses this problem by two approaches: First, it offers functionality to find the best fitting node within the abstract syntax tree. In Listing 4.3 the best fitting node is the *for-Statement* (Lines 2 - 7). The advantage is that this node can already be used as input for the ChangeDistiller, but the two nodes representing the Method-Calls (Line 8, 9) are lost. This approach is only an approximation and may lead to biased results, when code clones cover many nodes with a small textual range in the source code.

The second approach finds all nodes within the textual range of the code clone. Referring to Listing 4.3 this includes the *for*-Statement (Line 2 - 7) and the two Method-Calls (Line 8, 9). Comparing to the first approach *looking for the best fitting node* the Method-Calls do not get lost this time. But the result are several nodes that are not tied together by a common root. They are therefore not suitable as input for the ChangeDistiller. The CloneAnalyzer overcomes this problem by putting those nodes together by a *dummy-root*-Node.

```
 1  public void someMethod(int a, int b){
 2        for (int i = 0;i<10;i++){
 3           //for-loop body
 4           if(a<b){
 5              //if-statement body
 6           }
 7        }
 8        otherMethod(b);
 9        otherMethod(a);
10        if(a==0){
11           System.err.println("Fatal Error: Abort");
12           System.exit(1);
13        }
14  }
```

Listing 4.3: Code Clone and Source Code Entities

Responsible for finding the corresponding AST nodes for a given textual range are:

- `org.evolizer.astdiff.cdt.ASTNodeFinder`: Finds the best fitting node covered by a given textual range inside the abstract syntax tree for C / C++ obtained by the CDT-Parser.
- `org.evolizer.astdiff.cdt.CDTASTMultipleNodeFinder`: Finds all nodes covered by a given textual range inside the abstract syntax tree for C / C++ obtained by the CDT-Parser.
- `org.evolizer.astdiff.jdt.extra.JDTASTNodeFinder`: Finds the best fitting node covered by a given textual range inside the abstract syntax tree for Java-Code obtained by the JDT-Parser.
- `org.evolizer.astdiff.jdt.extra.JDTASTNodeFinder`: Finds all nodes covered by a given textual range inside the abstract syntax tree for Java-Code obtained by the JDT-Parser.

Detect Source Code Changes : The CloneAnalyzer uses the abstract syntax tree differencing algorithm of the ChangeDistiller to detect source code changes. AST nodes constructed

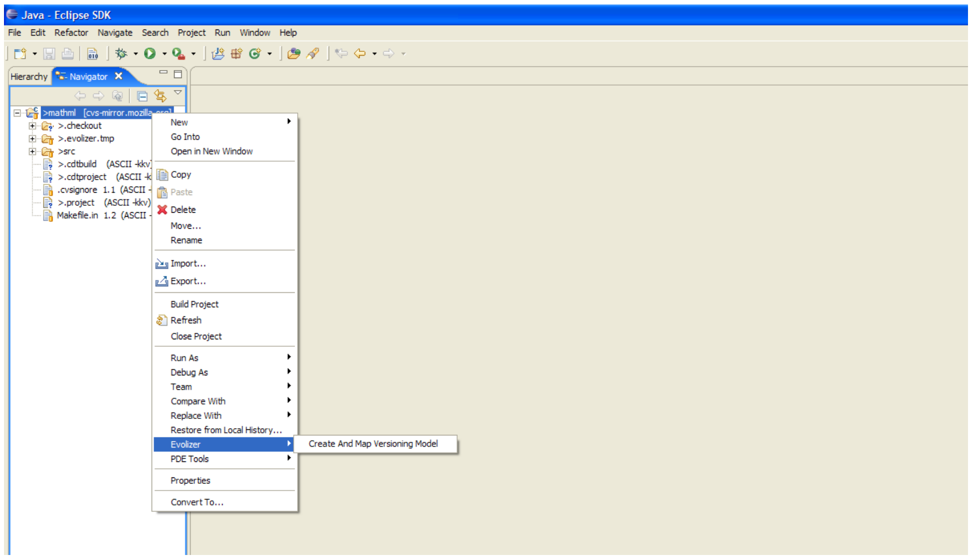


Figure 4.4: Context Menu of the EvolizerBase

by JDT or CDT must be transformed into language independent regular trees, before we can use them with the ChangeDistiller. These transformations are done by classes of the packages

`org.evolizer.astdiff.cdt`

for abstract syntax trees of C / C++ source code and by

`org.evolizer.astdiff.jdt`

for abstract syntax trees of Java source code.

Release History Information : For acquiring the release history data the EvolizerBase is used (cf. Section 3.3.2). The EvolizerBase is implemented as Eclipse-Plug-In and integrated in the context menu when right-clicking on an Eclipse-Project in the Exploring-View (cf. Figure 4.4). Extracting release data from large software projects such as Mozilla results in a huge amount of information. Since the EvolizerBase puts the obtained data in a MySQL- or Postgre-Database, a sufficient upstream connection to the server hosting the database is necessary to complete the extraction process in a reasonable time frame.

The EvolizerBase uses a simplified versioning meta model of [Marjanovic 2006]. It currently consists of 12 Java-Classes - only three of them are necessary to calculate change coupling groups [Jetter and Wuersch 2005]. Every check-in of a file is reflected by an entry in the CVS log file. To distinguish between two or more versions of a file CVS distributes *revision numbers* [Bartels and Jaehne 2000].

The relevant class for calculating change coupling groups is `Transaction` of the package `org.evolizer.base.versioning.model`. An instance of this class contains a set of revisions of files that have been checked-in together during a specified time slot. Every file in the CVS repository has its corresponding instance of `File` in the EvolizerBase. Since every revision number can be uniquely related to a file in the CVS repository, such a transaction instance and its set of revision numbers allows to identify change coupled files.

Further information provided by the EVOLIZERBASE, *e.g.*, modifications reports, authors of revisions etc. are not of any importance so far.

The class `CouplingCalculator` is responsible for calculating change coupling groups in the CLONEANALYZER. As seen not every change coupling group is of the same interest. The `CouplingCalculator` allows to filter coupling groups by means of $\#Occurrences$ and $\#NumberofFiles$.

Track Code Clones over Time : Tracking code clones over time is necessary to detect whether a code clone has changed between two subsequent revisions. The CCFinder assigns a unique ID to every code clone (cf. Section B.1 and Figure 4.3). This ID allows to locate a code clone in different Clone Pairs of an output file of the CCFinder. The CLONEANALYZER uses this ID to achieve a mechanism to identify a specific code clone throughout several revisions of a source file. It checks out the revisions of each involved source code file to the local system and appends the corresponding revision number to the file name. For instance, file *AbstractCalculator.cpp* with revisions 1.1, 1.2 and 1.3 and file *NumericCalculator.cpp* with revisions 1.7, 1.8 and 1.9 are subject to a clone tracking process. The local file system looks then as followed after the check out:

$$\begin{matrix} AbstractCalculator\_1.1.cpp & AbstractCalculator\_1.2.cpp & AbstractCalculator\_1.3.cpp \\ NumericCalculator\_1.7.cpp & NumericCalculator\_1.8.cpp & NumericCalculator\_1.9.cpp \end{matrix}$$

These files are then subject to a code clone detection run. Listing 4.4 shows an excerpt of the output file for this run.

```
preprocess_script: cpp
#begin{file description}
1 502 AbstractCalculator_1.1.cpp
2 510 AbstractCalculator_1.2.cpp
3 302 NumericCalculator_1.7.cpp
4 307 NumericCalculator_1.8.cpp
//more source files....
#end{file description}
#begin{clone}
9 1 196,1 202,1 2 235,1 243,1
9 3 50,1 55,2 4 52,1 58,2
9 1 196,1 202,1 3 50,1 55,2
9 1 196,1 202,1 4 52,1 58,2
//more clone pairs....
#end{clone}
```

Listing 4.4: CCFinderX Output for Source Files with Revision-Number Appendix

One can see that the clone number 9 appears in revisions 1.1 and 1.2 of file *AbstractCalculator.cpp* and in revisions 1.7 and 1.8 of file *NumericCalculator.cpp*. The position information for clone 9 in file *AbstractCalculator.cpp* is then used to find the abstract syntax tree covered by clone 9 in revision 1.1 and 1.2. These syntax trees are compared to identify whether clone 9 has changed between revision 1.1 and 1.2 in *AbstractCalculator.cpp*. This approach reflects the term *File-History* introduced in Section 3.3.3. The class `FileHistory` in the package `org.evolizer.codeclones` implements this concept.

The method `FileHistory#trackCloneInHistory(int cloneId)` shows in which revisions of a file the specified clone `cloneId` appears.

Computation and Presentation of Metrics : As outlined in Sections 4.2 and 3.3.1 the CLONEANALYZER has to be realized as an Eclipse Plug-In. This constraint implies for logical reason to provide an User-Interface (UI) that is fully integrated in the Eclipse-Workbench paradigm (cf. Figure 4.2). The UI of the CLONEANALYZER is implemented in SWT and is integrated into the Eclipse-Workbench by using JFace-Components. The step by step walk-through guide in Section 4.3 explains the use of the UI in detail.

4.3 Clone Analyzer Manual

Section 4.2 addressed the implementation of the CLONEANALYZER. The goal is to provide an automated way for the analysis part (step 5-10) of the framework. Reading this manual is not necessary to understand the scientific part of this thesis. If one is not interested in the use of the CLONEANALYZER, he can skip this section without missing relevant findings concerning code clones and change couplings. It is possible to come back to the manual anytime when actually working with the CLONEANALYZER. Because the implementation of the CLONEANALYZER constitutes an important part of this thesis, it was decided to present the CLONEANALYZER in a more detailed manner. Many of the design decisions, properties and features of the CLONEANALYZER have already been covered by the previous Section 4.2.

4.3.1 Setup

As seen in section 3.2 the CLONEANALYZER requires other tools and technologies to work with:

- **CCFinderX** for code clone detection
- **CHANGEDISTILLER** for detecting source code changes
- **EVOLIZERBASE** for building the release history database (RHDB) from log files of CVS
- An **Eclipse SDK** distribution; CLONE ANALYZER, CHANGEDISTILLER and EVOLIZERBASE are plugged into the Eclipse-Platform.
- A **JDT** Plug-In distribution for constructing the abstract syntax trees from Java source code. The JDT is usually included in the Eclipse SDK.
- A **CDT** Plug-In distribution for constructing the abstract syntax trees from C/C++ source code.
- An external **C/C++ compiler** is necessary to make use of the CDT.[7]
- A **MySQL-database** to store code clone data and release history information.
- **Java 5.0** (*Tiger-Release* 1.5) platform

Because of the CCFinder the CLONEANALYZER only runs with Windows XP 32-bit. Other required libraries such as Hibernate are included in the `lib` directory. One problem when using the CLONEANALYZER and its related tools is the huge volume of data to be processed. Detecting code clones, evolving their change behavior and the extraction of release history data are resource

[7]For example *Cygwin* in a Windows environment: http://www.cygwin.com/

consuming tasks. It is necessary to allocate the possible maximum memory to the Java heap which is somewhere around 1.5 GB. At least 3 GB RAM must be available to keep the host system of the CLONEANALYZER running and prevent it from crashing. If the MySQL database does not run on the same physical device as the CLONEANALYZER a sufficient upstream connection is needed.

4.3.2 Clone Analyzer Properties

The corresponding folder of the package `org.evolizer.codeclones` includes the property file `analyzer.properties`.[8] The property entries in this file must be adapted to the actual running environment before using the CLONEANALYZER. A property is a key-value pair separated by the equal sign: `<key>=<value>`. There are currently three property entries:

- `CCFinder`: The CLONEANALYZER starts the CCFinder as an external process. It is necessary to know where the CCFinder binaries can be found. The value of this pair denotes the absolute system path to the *bin* directory of the CCFinder distribution.
- `statsFolder`: The CLONEANALYZER creates statistics log files as described in Section 4.3.8. The value of this pair denotes the absolute system path to the folder where the log files are to be created.
- `languages`: This property denotes the programming languages for which the CLONEANALYZER is implemented to evolve the changes of code clones. The languages are described by the extensions of the related source code files: `*.cpp`[9] for C / C++ and `*.java` for Java.

For example the `CCFinder}` property entry of the system setup used for the case studies in Chapter 5 is `CCFinder=C:\Programme\ccfinder\ccfx-win32-en\bin`.

4.3.3 Context Menu Integration of the Clone Analyzer

Figure 4.5 shows the context menu of the CLONEANALYZER when right-clicking on source files in the *Project Explorer*. Depending on the selected perspective in the Eclipse-Workbench the functionality of the CLONEANALYZER is available for `*.java`, `*.cpp`, `*.c` and `*.h` files.[10] The opened context menu offers three submenus: *Start a Clone Detection*, *Perform Code Clone Analysis*, and *Perform Coupling Analysis*. The first two submenus are also available when right-clicking on Eclipse projects and folders. As source code files, projects and folders in the resource tree of the Eclipse-Workspace have different semantic meanings depending on the selected perspective. The CLONEANALYZER works with raw Eclipse projects, *Java-Projects* (of the Java-Perspective) and *C/C++-Projects* (of the C / C++-Perspective). Figure 4.6 shows the context menu when selecting a folder with source files in the C / C++-Perspective.

In Section 3.2.3 we raised the question whether to start the analysis with files sharing code clones and then examining their change behavior in respect to change couplings; or whether to start with change coupling groups and then investigating the changes of cloned source code fragments. The CLONEANALYZER gives consideration to this thought by offering both ways: Browsing by code clones as well as navigating through change coupling groups. The three submenus form the origin for the code clone and change coupling analysis and are explained in detail in the succeeding. In the reminder of this manual `nsMathMLmrootFrame.cpp` and `nsMathMLmsqrtFrame.cpp` from the Mozilla module `mozilla/layout/mathml` are used for demonstration purposes of the CLONEANALYZER.

[8]It is important that the property file lies in the same folder as *org.evolizer.codeclones.PropertyReader.java*. Otherwise the CLONEANALYZER can not find the file and aborts.

[9]*cpp* is representative for all other extensions of C / C++ source code files such as **.c*, **.h* or **.t*.

[10]For the installation of Plug-Ins into the Eclipse-Platform please refer to the Eclipse-Help

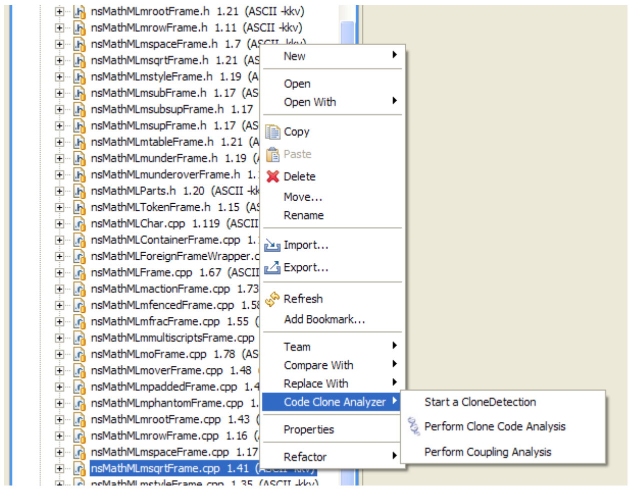


Figure 4.5: Context Menu of the CLONEANALYZER for Source Code Files

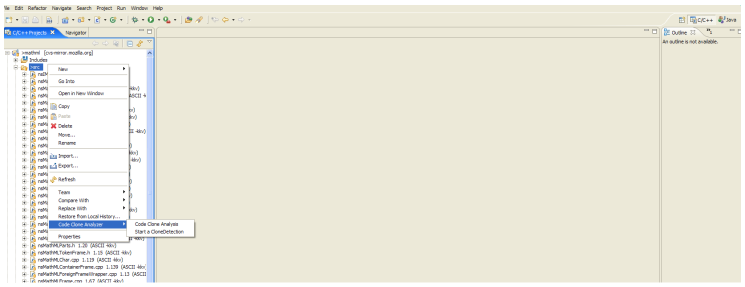


Figure 4.6: Context Menu of the CLONEANALYZER for Folders

4.3.4 Start a Clone Detection

When starting the examination of a software system with code clones, initial clone data is required as a basis. The submenu *Start a Clone Detection* starts the CCFinder. The input parameters for the detection run depend on the selected resource entity:

- **Source Code File**: The absolute system path to the parent directory of the selected source code file is passed as input parameter to the CCFinder. For instance, if source code file i is selected in Figure 4.7, the path of `Root Directory` is passed to the CCFinder. Since the CCFinder scans the input root folder in a recursive manner, all other source files $1 - n$ in `Root Directory` are subject to the detection run; as well as all child directories j and their source code files $1 - n$. The extension of the selected source code file is used as `preprocessed_script` parameter for the CCFinder.

- **Folder**: When selecting a folder in the resource tree of the Eclipse-Workspace, a dialog is opened to choose the programming language – currently `cpp` and `java` are available. The selected entry is used as `preprocessed_script` parameter for the CCFinder. The CCFinder uses the path of the selected folder as input argument.

- **Java-Package**: The same principle as for selected folders is applied since a Java-Package corresponds to a folder in the Eclipse-Workspace of the local system.

- **Eclipse-Project**: The same principle as for selected folders is applied since an Eclipse-Project corresponds to a folder in the Eclipse-Workspace of the local system.

The recursive scanning of the CCFinder facilitates narrowing down the code clone detection. Selecting `Root Directory` in Figure 4.7 results in a general analysis. Selecting a child directory j in Figure 4.7 specializes the analysis on that specific directory.

The results of the detection run are automatically stored in the MySQL database. According to the Hibernate mapping of Figure 4.3, three tables – `T_DETECTIONRUN`, `T_DATAFILES` and `T_CLONEPAIRS` – are created by Hibernate in the MySQL-database.

4.3.5 Perform Code Clone Analysis

In this manual the submenu *Perform Code Clone Analysis* for `nsMathMLmrootFrame.cpp` is selected. If there are any code clones found in the database for this source file a *Multi-Page Editor* with the *Code Clone Browser* tab opens up in the Eclipse-Workbench window as shown in Figure 4.8.

The Clone Browser is divided up into two sections. On the left there is a drop down menu containing all detection runs exhibiting code clones for `nsMathMLmrootFrame.cpp`. Choosing a detection run from the drop down menu automatically causes an update of the browser window. Below the drop down menu there is a list of measures for the currently selected run:

- **#Clone-Files**: Describes the number of files that are affected by code clones.

- **#Clones**: Describes the number of code clones.

- **#Intra-File Clones**: Describes the number of code clones that appear exclusively in one single file. Meaning such a clone refers to a source code fragment that has been cloned within the same file.

- **#Inter-File Clones**: Describes the number of code clones that appear in different files. Meaning such a clone refers to a source code fragment that has been cloned across different files.

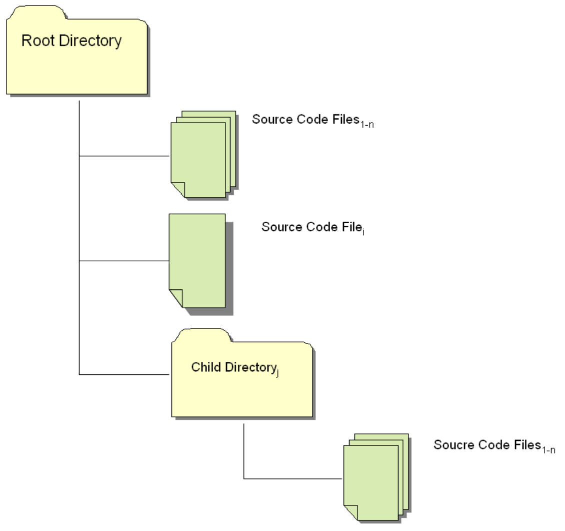
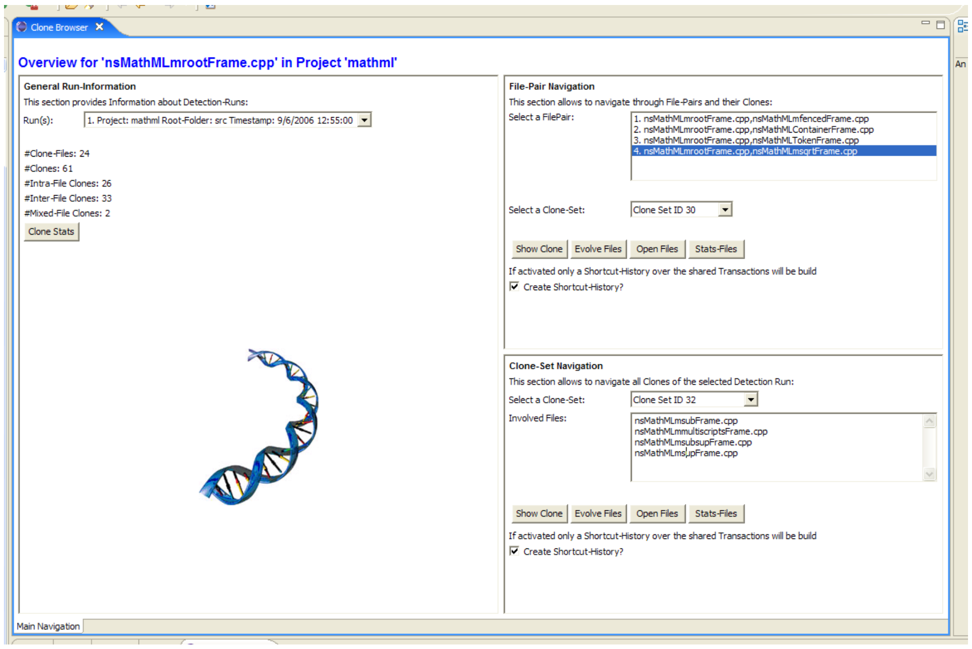


Figure 4.7: CCFinderX Input Structure



Overview for 'nsMathMLmrootFrame.cpp' in Project 'mathml'

General Run-Information

This section provides Information about Detection-Runs:

Run(s): 1. Project: mathml Root-Folder: src Timestamp: 9/6/2006 12:55:00

#Clone-Files: 24
#Clones: 61
#Intra-File Clones: 26
#Inter-File Clones: 33
#Mixed-File Clones: 2

Clone Stats

File-Pair Navigation

This section allows to navigate through File-Pairs and their Clones:

Select a FilePair:
1. nsMathMLmrootFrame.cpp,nsMathMLmfencedFrame.cpp
2. nsMathMLmrootFrame.cpp,nsMathMLContainerFrame.cpp
3. nsMathMLmrootFrame.cpp,nsMathMLTokenFrame.cpp
4. nsMathMLmrootFrame.cpp,nsMathMLmsqrtFrame.cpp

Select a Clone-Set: Clone Set ID 30

Show Clone | Evolve Files | Open Files | Stats-Files

If activated only a Shortcut-History over the shared Transactions will be build

☑ Create Shortcut-History?

Clone-Set Navigation

This section allows to navigate all Clones of the selected Detection Run:

Select a Clone-Set: Clone Set ID 32

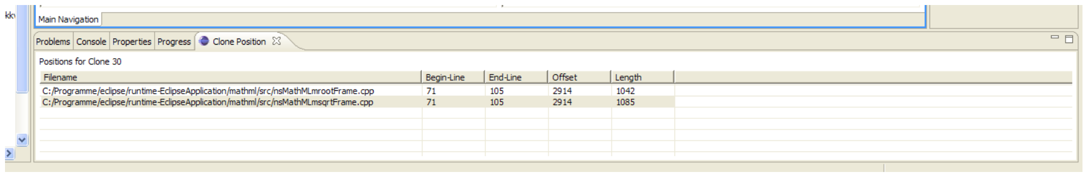
Involved Files:
nsMathMLmsubFrame.cpp
nsMathMLmmultiscriptsFrame.cpp
nsMathMLmsubsupFrame.cpp
nsMathMLmsupFrame.cpp

Show Clone | Evolve Files | Open Files | Stats-Files

If activated only a Shortcut-History over the shared Transactions will be build

☑ Create Shortcut-History?

Figure 4.8: Code Clone Browser for nsMathMLmrootFrame.cpp



Positions for Clone 30

Filename	Begin-Line	End-Line	Offset	Length
C:/Programme/eclipse/runtime-EclipseApplication/mathml/src/nsMathMLmrootFrame.cpp	71	105	2914	1042
C:/Programme/eclipse/runtime-EclipseApplication/mathml/src/nsMathMLmsqrtFrame.cpp	71	105	2914	1085

Figure 4.9: Position Table of Code Clone 30 in nsMathMLmrootFrame.cpp and nsMathMLmsqrtFrame.cpp

- **#Mixed-File Clones**: Describes the number of code clones that appear in the same file as well as in other files. Meaning such a clone refers to a source code fragment that has been cloned within the same files as well as across other source code files.

The `Clone-Stats`-Button below these measures generates the statistic log file for this run (cf. section 4.3.8 *Generating Log Files*).

The panels on the right side of the Code Clone Browser allow users to navigate through the code clones of the selected detection run. The upper half *File-Pair Navigation* depicts the clone pair approach of the CCFinder output. After having selected a detection run, the list displays all clone pairs for `nsMathMLmrootFrame.cpp`. Selecting *e.g.*, the *4th* file pair in the list – `nsMathMLmrootFrame.cpp` and `nsMathMLmsqrtFrame.cpp` – activates the clone set drop down menu and the panel of buttons. The drop down menu displays all code clones that affect the selected file pair. The meanings of the buttons below are:

- `Show Clone`-Button: Opens an view in the Eclipse-Workbench window showing the positions of the selected code clones in a given file pair. Figure 4.9 shows a view displaying the positions of code clone 30 in `nsMathMLmrootFrame.cpp` and `nsMathMLmsqrtFrame.cpp`. Clicking on a row in the position table opens the source code file in its default Eclipse editor. The textual range of the code clone is then highlighted on the left of the editor.

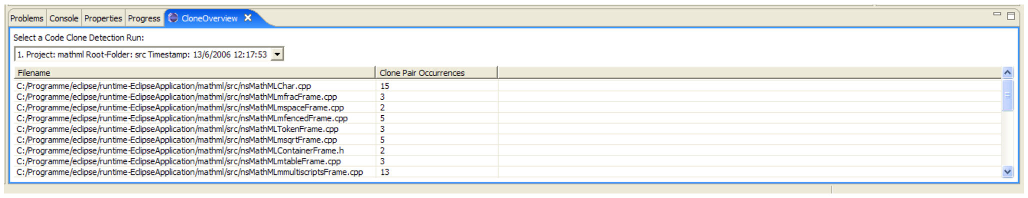
- `Evolve Files`-Button: Builds a File-History for the selected files and adds a new tab to the Multi-Page Editor (cf. Section 4.3.7 *Building a File-History*).

- `Open Files`-Button: Opens the selected files in their default Eclipse editor for further working.

- `Stats Files`-Button: Generates the statistic log file for the selected files (cf. Section 4.3.8 *Generating Log Files*).

The lower left half of the Clone Browser *Clone-Set Navigation* has a drop down menu containing all code clones of the selected detection run. Contrary to the *File-Pair Navigation* the focus lies on the code clone itself. It reflects therefore the characteristic that a cloned source fragment appears in more than just two files in many occasions. Selecting a code clone makes the CLONE-ANALYZER display all affected files. For instance selecting code clone 32 in the example names the four affected files

```
nsMathMLmsubFrame.cpp
naMathMLmmultiscriptsFrame.cpp
naMathMLsubsupFrame.cpp
nsMathMLsupFrame.cpp
```



Select a Code Clone Detection Run:

1. Project: mathml Root-Folder: src Timestamp: 13/6/2006 12:17:53

Filename	Clone Pair Occurrences
C:/Programme/eclipse/runtime-EclipseApplication/mathml/src/nsMathMLChar.cpp	15
C:/Programme/eclipse/runtime-EclipseApplication/mathml/src/nsMathMLmfracFrame.cpp	3
C:/Programme/eclipse/runtime-EclipseApplication/mathml/src/nsMathMLmspaceFrame.cpp	2
C:/Programme/eclipse/runtime-EclipseApplication/mathml/src/nsMathMLmfencedFrame.cpp	5
C:/Programme/eclipse/runtime-EclipseApplication/mathml/src/nsMathMLTokenFrame.cpp	3
C:/Programme/eclipse/runtime-EclipseApplication/mathml/src/nsMathMLmsqrtFrame.cpp	5
C:/Programme/eclipse/runtime-EclipseApplication/mathml/src/nsMathMLContainerFrame.h	2
C:/Programme/eclipse/runtime-EclipseApplication/mathml/src/nsMathMLmtableFrame.cpp	3
C:/Programme/eclipse/runtime-EclipseApplication/mathml/src/nsMathMLmmultiscriptsFrame.cpp	13

Figure 4.10: Code Clone Overview of the Eclipse Project mathml

as shown in Figure 4.8. The buttons for the *Clone-Set Navigation* have the same meaning as the buttons for the *File-Pair Navigation* approach, except for the fact that all files affected by the selected code clone – instead of only one certain file pair – are involved in the triggered actions.

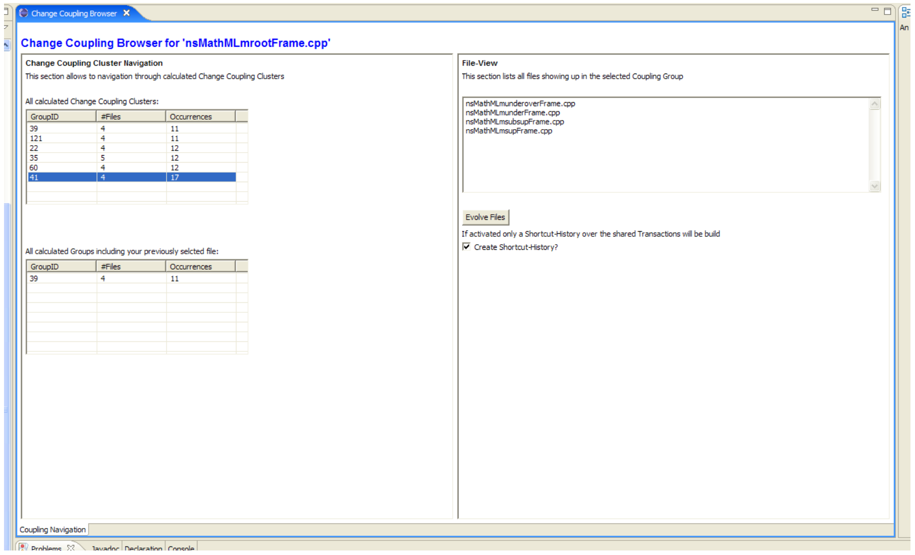
Performing a code clone analysis by a folder or an Eclipse project is detached from a specific file and causes the CLONEANALYZER to open a new view in the Eclipse-Workbench as shown in Figure 4.10. This view contains a drop down menu listing all detection runs in the database for the selected folder respectively project. Selecting a detection run entry from the menu updates the table below. The first table row shows all files of the selected run that are affected by a code clone. The second row contains information on how many times each file occurs in a clone pair.

4.3.6 Perform Change Coupling Analysis

The third submenu point starts with change coupling groups when analyzing code clones and their change behavior. Since not every change coupling group is of interest, the CLONEANALYZER allows to filter change coupling groups by $\#Files$ and $\#Occurrences$. For instance, if the user specified $\#Files = 4$ and $\#Occurrences = 10$ only those change coupling groups are returned which consist of at least 4 files and occur at least 10 times over the entire release history. After having calculated all coupling groups according to the given parameters, the CLONEANALYZER opens a Multi-Page Editor in the Eclipse-Workbench window and adds the *Change Coupling Browser* tab. Figure 4.11 shows the change coupling browser for the parameters $\#Files = 4$ and $\#Coccurrences = 11$ calculated for the release history database for module `mozilla/layout/mathml`. On the left half there are two tables. The upper one shows all calculated coupling groups. The lower table shows those coupling groups out of the couplings groups from the upper table that contain the selected file. In the example only group 39 contains the selected file `nsMathMLmrootFrame.cpp`. When clicking on a coupling group in one of the two tables all files forming that group are listed then on the right. The `Evolve Files`-Button creates a File-History of the files of the selected change coupling group (cf. Section 4.3.7 *Building a File-History*).

4.3.7 Building a File-History

Section 3.3.3 introduced *File-History* and *Temporal Appearance Matrix* τ. Both terms are used to evolve code clones and their affected files over time. The CLONEANALYZER implements this concept (cf. section 4.2.2 *Clone Analyzer Design and Architecture*). As mentioned, the range of revisions of a source file that are examined in a File-History is open as the case arises. The CLONEANALYZER offers two ways to build a File-History. First a *full range* File-History. In this case every revision of each file is considered. But when analyzing several files of a relatively old software system, building a full range File-History leads to heavy workload. Since this thesis focuses primarily on code clones and change couplings, the CLONEANALYZER has the option to build a *shortcut* File-History, which includes only the change coupled revisions (including the corresponding previous



Change Coupling Browser for 'nsMathMLmrootFrame.cpp'

Change Coupling Cluster Navigation

This section allows to navigation through calculated Change Coupling Clusters

All calculated Change Coupling Clusters:

GroupID	#Files	Occurrences
39	4	11
121	4	11
22	4	12
35	5	12
60	4	12
41	4	17

All calculated Groups including your previously selcted file:

GroupID	#Files	Occurrences
39	4	11

File-View

This section lists all files showing up in the selected Coupling Group

nsMathMLmunderoverFrame.cpp
nsMathMLmunderFrame.cpp
nsMathMLmsubsupFrame.cpp
nsMathMLmsupFrame.cpp

If activated only a Shortcut-History over the shared Transactions will be build

[✓] Create Shortcut-History?

Figure 4.11: Change Coupling Browser for File nsMathMLmrootFrame.cpp with #Files = 4 and #Coccurrences = 11

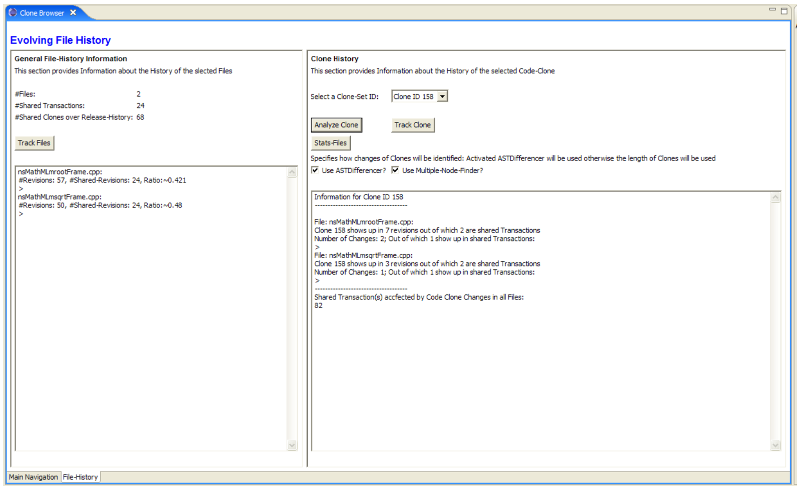
revisions) of each file. Using such a shortcut File-History reduces the workload significantly.

After having build the File-History and its clone data has been put into the MySQL-database, a new tab *File-History* is added to the Multi-Page Editor. Figure 4.12 shows a File-history tab for `nsMathMLmrootFrame.cpp` and `nsMathMLmsqrtFrame.cpp`. Initially on the left of the tab there is some information about the files of the File-History; such as the number of change couplings (*#Shared Transactions*) and the number of code clones detected in this File-History (*#Shared Clones over Release-History*) etc. These code clones with their assigned CCFinder ID are listed in the drop down menu on the right.

The meanings of the buttons in a File-History tab are:

- `Track Files`-Button: Opens a table in the view part of the Eclipse-Workbench window showing the revision numbers of each files that correspond to their common check-ins.
- `Stats Files`-Button: Generates the statistic log file for the files of the File-History (cf. Section 4.3.8 *Generating Log Files*).
- `Analyze Clone`-Button: Calculates a textual representation of the *Temporal Appearance Matrix* for the code clone selected in the drop down menu. The results are displayed in the text area in the right half of the File-History tab. In Figure 4.12 clone 158 was selected. One can see that it caused 1 change couplings – Hibernate Transaction ID 82 – for the files `nsMathMLmrootFrame.cpp` and `nsMathMLmsqrtFrame.cpp`.[11]
- `Track Clone`-Button: Opens a new table in the view part of the Eclipse-Workbench window. Figure 4.13 shows such a table for clone 158. This table is a visualization of the temporal appearance matrix. It shows all revisions of each file where the selected code clone occurs. Furthermore the cells are colored. The colors describe the *Clone Change Types* as introduced in Section 3.3.3: $RED = CCT3$, $GREEN = CCT1$, $BLUE = CCT2$ and $WHITE = CCT0$. The table shows that clone 158 occurs in revision 1.3, 1.4, 1.5, 1.6, 1.7, 1.8 and 1.9 of file `nsMathMLmrootFrame.cpp`. It appears in revision 1.3, 1.4, 1.5 of source

[11]The numbers denoting the shared transactions that were caused by changes in code clones correspond to the unique ID of the Hibernate bean class *Transaction* in the *Versioning Meta Model* used for the EVOLIZERBASE [Jetter and Wuersch 2005].



Evolving File History

General File-History Information

This section provides Information about the History of the slected Files

#Files:	2
#Shared Transactions:	24
#Shared Clones over Release-History:	68

```
nsMathMLmrootFrame.cpp:
 #Revisions: 57, #Shared-Revisions: 24, Ratio:~0.421
>
nsMathMLmsqrtFrame.cpp:
 #Revisions: 50, #Shared-Revisions: 24, Ratio:~0.48
>
```

Clone History

This section provides Information about the History of the selected Code-Clone

Select a Clone-Set ID: Clone ID 158

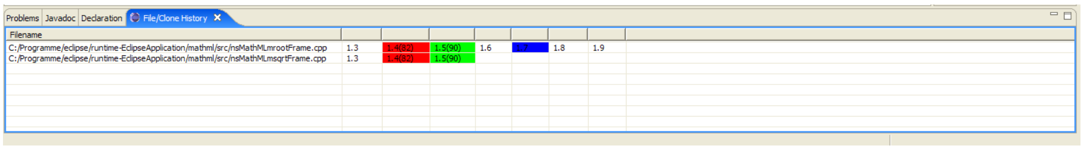
Specifies how changes of Clones will be identified: Activated ASTDifferencer will be used otherwise the length of Clones will be used

[✓] Use ASTDifferencer? [✓] Use Multiple-Node-Finder?

```
Information for Clone ID 158
-----------------------------------

File: nsMathMLmrootFrame.cpp:
Clone 158 shows up in 7 revisions out of which 2 are shared Transactions
Number of Changes: 2; Out of which 1 show up in shared Transactions:
>
File: nsMathMLmsqrtFrame.cpp:
Clone 158 shows up in 3 revisions out of which 2 are shared Transactions
Number of Changes: 1; Out of which 1 show up in shared Transactions:
>
-----------------------------------
Shared Transaction(s) accfected by Code Clone Changes in all Files:
82
```

Figure 4.12: File-History Panel for Code Clone 158



Filename							
C:/Programme/eclipse/runtime-EclipseApplication/mathml/src/nsMathMLmrootFrame.cpp	1.3	🟥 1.4(82)	🟩 1.5(90)	1.6	🟦 1.7	1.8	1.9
C:/Programme/eclipse/runtime-EclipseApplication/mathml/src/nsMathMLmsqrtFrame.cpp	1.3	🟥 1.4(82)	🟩 1.5(90)				

Figure 4.13: Temporal Appearance Matrix for Clone 158

file `nsMathMLsqrtFrame.cpp`. The red cells indicate that clone 158 changed between revision 1.3 and 1.4 in both files, and that these revision steps represent a change coupling. Both red cells have the same Hibernate transaction ID 82. Therefore this change coupling was caused by clone 158. The green cells with the transaction ID 90 show that clone 158 occurs in further change couplings, but it did not change. Another change occurred from 1.6 to 1.7 in file `nsMathMLmrootFrame.cpp`. Since this revision step does not correspond to a change coupling, the cell is colored blue.

The colored table view corresponds to a temporal appearance matrix enriched with information about structural change as described in Section 3.3.3.

4.3.8 Generating Log Files

Generating Log or *Stat(istic)s* files is an automated routine specially tailored and implemented for the case studies carried out in Chapter 5. For a given file group or a set of file groups all File-Histories are built in a first step. In a second step the CLONEANALYZER calculates the following metrics for **every** clone detected in the examined File-History. The meaning of the metrics are explained in Section 3.3.3:

- **#Files**: Cardinality of the File-History of the code clone.
- **CloneId**: Unique ID assigned by the CCFinder. This ID is also displayed in the drop down menu in the right half of a File-History tab (cf. Figure 4.12).
- γ: Number of code clones detected in the File-History.

- δ: Number of change couplings of the files forming the File-History
- ε: Number of change couplings that were caused by changes in the code clone `cloneID`.

The calculation results are saved in a **.txt*-File. A formal description of the stats file is given in Appendix B.2.

4.4 Final Remarks

This chapter covered the implementation of the Eclipse Plug-In CLONEANALYZER; a tool that provides an automated mechanism for users in order to evolve code clone changes in the context of change couplings. We use the CLONEANALYZER to process two case studies in Chapter 5.

Chapter 5

Case Studies and Evaluation

Chapter 4 presented the implementation of the CloneAnalyzer; a tool to process steps 5 - 10 of the framework as automated as possible. In this chapter we use the CloneAnalyzer to apply these remaining steps to case studies and present the results. The experiment will hopefully give new insights in the relation of code clones and change couplings, and can be used to investigate the state of a software system. The case studies also constitute an evaluation to judge the benevolence and usefulness of the framework and methodology developed in this thesis.

We will evaluate a C / C++ case study where we take a closer look at the Mozilla project, and a Java case study where source code from the Eclipse project is analyzed. Each case case study is first examined individually. In a second step we try to give an aggregate overview of all results.

The manual inspection of the results during the evaluation of the case studies showed that there is some blurring in these results due to shortcomings and problems in the technologies used of our implementation. We will address these problems in Section 5.5 and discuss their biasing and the negative impact on the results.

5.1 Environment

All case studies were run on a computer with a 3.46 GHz Intel Pentium Extreme Edition CUP using Windows XP Service Pack 2 as operating system. The system was stocked with 4 GB of RAM. According to the manual 1.5 GB of RAM was allocated to the Java heap. The manual in Section 4.3 describes the necessary setup to run the CloneAnalyzer. Appendix C lists all required tools for the case studies in this chapter.

5.2 C/C++ Case Study - Mozilla

Section 3.3.3 defines the measures and metrics, which we use in our case studies. As described, these metrics make use of file groups and the term File-History in order to calculate concrete values for the metrics set. The question that emerges now, is how to extract such file groups out of the CVS repository of the Mozilla project. We already presented some general thoughts on this question in Sections 3.2.2 and 3.2.3 when discussing it on a more abstract level. The first three Sections 5.2.1, 5.2.2 and 5.2.3 show the procedure of how to get suitable files from the Mozilla source code.

5.2.1 Input Selection

Only source files written in C / C++ are of interest for our case study. The reason for this decision is evident: The CLONEANALYZER is only implemented for C / C++ (and Java); other files like binaries, makefiles, manifest files, images, CVS log files etc. in the CVS repository of Mozilla are not important in the context of code clones and change couplings. The CCFinder automatically filters out those files in the input folders that do not match to the *preprocessed language* option.[1] It is therefore not stringently necessary to delete those non C / C++ matching files and their directories. However this is strongly recommended if the input folders contain a lot of empty child folders or child folders containing files other than C / C++ source code files. Deleting those folders by hand can increase the performance, since it prevents the CCFinder from needlessly scanning them. The decision whether to delete such files and folders depends on the size of the examined Mozilla module. This leads to an input structure of folders that contain files with the extensions `*.h`, `*.c` and `*.cpp`. We then filtered out those modules of the CVS repository that include only small number of source code files or mostly header files (`*.h`). The reason for this decision was, that such modules do not exhibit a sufficient amount of code clones to evolve. Header files usually contain only declarations of various entities but no functionality except of inline functions. The number of such inline functions is limited and thus not significant for this case study. The remaining modules were then subject to the next step *Obtaining and Collecting Initial Code Clone and Change Coupling Information* of the framework.

Early tests showed that tracking the change behavior of code clones is very time and resource consuming. We decided not to investigate the entire source code of the Mozilla project but only certain (sub-) modules at one time. This guarantees that the volume of data and the results are still manageable by the CLONEANALYZER and can be processed in a reasonable time frame.

5.2.2 Obtaining and Collecting Initial Code Clone and Change Coupling Information

Because we normally do not know whether a group of files has code clones or was change coupled at certain point of time in the release history of a system, some initial data is needed. We checked out the latest revisions of the modules / files that were selected in the previous section. They were then subject to a code clone detection run, and to the extraction process of the EVOLIZERBASE. We used this initial data to filter the files further. Unlike during the previous selection process this time the focus lies on selecting specific files rather than modules and folders.

5.2.3 Identification of Possible Clone and Coupling Candidates

Based on the initial code clone and change coupling data, this phase extracts those file groups that will be subject to the code clone and change coupling analysis. A file group must fulfill two preconditions to be suitable for evolving code clones: First, the files of such a group must share at least one code clone (the more the better). Second, they must be change coupled (the more the better). The data from Section 5.2.2 suffers from the problem, since it did not consider these preconditions. The clone data is based only on a snapshot of the latest file revisions. It does not reflect the temporal aspect of change couplings and version information. The change coupling groups were elicited solely on the release history data extracted with the EVOLIZERBASE. Such data lacks the information about code clones. We do not know whether change coupling groups

[1] The *preprocessed language* option is a property that must be defined when using the CCFinder. It denotes the programming language that is subject to the code clone detection process.

share code clones; or whether file groups that are affected by code clones were change coupled during release history.

We reduced to the entire Mozilla source code to file groups either being change coupled or affected by code clones. We then decided to use the files affected by a code clone out these remaining file groups as input for the analysis. The reason for this decision was, that we just could overtake them from the initial clone data. This way we did not have to filter change coupling groups another time by specifying values for the parameters $\#Files$ and $\#Occurrences$.

5.2.4 Tracking Clones over Time

In this step we track code clones over different file revisions, and see whether they changed and caused their affected files to be change coupled. In Section 3.3.3 the term *File-History* was introduced to describe changes in code clones between different revisions of a file. A File-History consists of a group of files and a specific number of revisions for each file. As shown in the preceding step, files that exhibit a code clone are used to build up a File-History. The last question to decide is what revisions are going to be incorporated in the change analysis. We decided to consider only those revisions of each file that correspond to change couplings.

In this case study we investigated 136 file groups from 13 (sub-)modules of the Mozilla project. These 136 file groups lead to a total sum of 324 investigated source files. We tracked the change behavior of 1188 code clones over 1655 change couplings in these source code files.

We calculated the following values for **every investigated file group** respectively its associated File-History using the metrics of Chapter 3.3.3 and the log file functionality of the CLONEANALYZER:[2]

1. Number of files of the file group
2. Number of code clones: γ
3. Number of change couplings: δ
4. Total number of change couplings that were caused by code clones: σ
5. Ratio between change couplings caused by code clones and total number of change coupling: ϱ

5.2.5 Analyzing Data from Tracking Process

In this section we use the calculated data and try to describe the relation between code clones and change couplings according to the research goals. Table 5.1 shows the frequency of σ values of the investigated file groups.

We can see that nearly 70% of the investigated file groups do not have any change couplings caused by code clones. Still approximately 30% of all analyzed file groups have change couplings due to changes in duplicated source code fragments. Most file groups have – if at all – a σ value between 1 - 10. There are some single outliners that have a significant high number of change couplings caused by code clones in our results.

We are anxious not only for an explanation of code clones and change couplings but also to provide a mechanism that helps software engineers to identify critical file groups. With this goal in mind looking only at σ – the absolute number of change couplings caused by code clones in a file group – is not adequate.

[2]The purpose of log files in the CLONEANALYZER is presented in Section 4.3.8 *Generating Log Files*.

Value	Absolute Frequency	Frequency in Percentage
0	93	68.38%
1 - 5	31	22.79%
6 - 10	7	5.14%
11 - 15	1	0.74%
16 - 20	1	0.74%
21 - 30	2	1.47%
31 - 40	0	0%
41 - 50	1	0.74%
	136 Groups	100%

Table 5.1: Frequency of σ Values (Mozilla)

Let have us a look at two file groups of our case study. The first file group consists of two files: `nsMsgProgress.cpp` and `nsMsgStatusFeedback.cpp`. Both files are part of the modul `mozilla/mailnews/base/src`. The second file group is formed by `nsMenuBarFrame.cpp` and `nsMenuPopupFrame.cpp` from the modul `mozilla/layout/xul/base/src`. Both file groups were 7 times change coupled due to structural changes in code clones. What is the conclusion for developers when trying to assess them? Are both file groups of "equal danger" in the context of code clones and change couplings? We see that the expressiveness of this sole fact σ is limited and not of any great use. It serves as best as an initial overview to see with how much clone related couplings we are dealing in our case study.

The situation is completely different when looking at ϱ – the ratio of those change couplings caused by code clones and the total number of change couplings. The files in the first file group were change coupled 11 times. Thus nearly 64% of all change couplings of this file group occurred because of code clones. If the developers eliminate the code clones in a refactoring effort from these two files, the refactoring is reflected in a reduce of change couplings in the future , this is a great benefit. The files of the second file group were checked-in 42 times toghether. This leads to a ratio of approximately 17%. Meaning that the impact of code clones on change couplings is much less severe in the second file group. This information is much more useful to compare and assess different file groups. We have to be careful with the ϱ value of 64% in the first file group, because we only rely on a short change coupling history of 11 change couplings compared to the second file group. Listing 5.1 shows the log file for the first file group that was generated with the CLONEANALYZER.[3]

```
beginFileHistory{
beginFiles{
nsMsgProgress.cpp
nsMsgStatusFeedback.cpp
}
beginChangeAnalysis{
#Files,#sharedClones,CloneSetId,#SharedTransactions,#AffectedSharedTransactions
2,3,19,11,0
2,3,20,11,3
2,3,34,11,4
}
}
```

[3]The use and meaning of log files are discribed in CLONEANALYZER manual in Section 4.3.8. The formal descibtion of the log file format is given in Appendix B.2.

Listing 5.1: Log File for Source Code Files *nsMsgProgress.cpp* and *nsMsgStatusFeedback.cpp*

By the means of this listing we can see that the file group has only 3 code clones. Two of them – clone 20 and 34 – caused change couplings. Because we only have 11 change couplings contrary to 42 change couplings for the second file group, is it more difficult to predict how the code clones will behave. If they remain stable and do not cause any change couplings in the future their relevance will decrease. In such a case it is not strongly necessary to remove the code clones at any cost, although a clone related coupling coverage of 64% seems quite high and critical. The situation is different if clones 20 and 34 continue to cause change couplings, or even new clones are introduced and start to cause change couplings. In this case measures to eliminate the code clones must be considered.

Although ϱ is an important measure to compare and investigate file groups, we can use the other information calculated by the CLONEANALYZER to draw a sharper picture of the state of code clones and change couplings: The *Number of Files* in a file group. In our case study 74.26% of all investigated file groups consist of only two files.[4] For instance, there are two file groups. Both were change coupled five time due to code clone changes. The first file group consists of two files, and the second one consists of four files. The impact of code clones on the second file group is more intense, since twice as much change couplings are affected as in the first file group. The benefit of eliminating those code clones is larger, because more files can be decoupled from each other.

Other useful information in this analyzing process is provided by the visualization of temporal appearance matrices in the CLONEANALYZER (cf. Figure 4.13). It shows the change behavior of a code clone during the release history of its affected files of the underlying File-History: A code clone that caused change couplings only in the early check-ins of the release history but remained stable in the recent past, is less harmful than a clone that constantly caused change couplings or recently became ”active”. The larger the amount of historical data, the more this increases the expressiveness of this fact.

Conclusion

In this section we have showed how the results of the metrics calculated by the CLONEANALYZER are used to investigate file groups in the context of change couplings and code clones. One can use them to identify critical file groups. As seen, it is important that this assessment is not based on the examination of single values but takes into account the different metrics provided by the CLONEANALYZER. This aspect is of particular importance when comparing different file groups amongst each other instead of looking only at single ones.

5.2.6 Correlation of Code Clones and Change Couplings

We have taken a first look at the results of the Mozilla case study in the previous section and used them to examine and compare file groups. This section covers the investigation of the relation between code clones and change couplings based on simple linear regression analysis. Because of the *bad smell* reputation of code clones, we expect that a higher number of code clones in a file group is reflected by an increasing value of ϱ. Table 5.2 shows the frequency of ϱ values in the Mozilla case study. The 93 file groups that did not have any change couplings caused by code clones, of course still have a ratio of 0%. Most file groups that were change coupled due to

[4]Remember: The number of files in a file group is determined by the CCFinderX, since we used the resulting file groups from the initially obtained clone data in Section 5.2.3.

Value	Absolute Frequency	Frequency in Percentage
0	93	68.38%
0.01 - 0.1	7	5.14%
0.11 - 0.2	16	11.75%
0.21 - 0.3	5	3.68%
0.31 - 0.4	1	0.74%
0.41 - 0.5	5	3.68%
0.51 - 0.6	1	0.74%
0.61 - 0.7	3	2.21%
0.71 - 0.8	1	0.74%
0.81 - 0.9	0	0%
0.91 - 1.0	0	0%
> 1.0	4	2.94%
	136 Groups	100%

Table 5.2: Frequency of ϱ Values (Mozilla)

code clones have a ϱ value between 1% - 30%. There are file groups with a ϱ value larger than 1 respectively 100%. This refers to the fact that a change coupling can be ”caused” by more than one code clone. Meaning that there are cases where more than one code clone changed in all files between a common check-in and the related previous revisions. Values of $\varrho > 1$ are set to 1 for the regression analysis.

We then plotted the number of code clones γ and the ratio between change couplings caused by code clones and total number of change couplings ϱ of all file groups against each other in a simple linear regression analysis. The analysis resulted in a correlation coefficient of 0.4585. Such a value does not express a significant correlation. The regression function has value for R^2 of 0.2102. Meaning that only one fifth of the scattering in the input sample is explained by the linear model. The resulting equation of the regression line is

$$\varrho = 0.0072 \times \gamma(F) + 0.0418$$

where F corresponds to the File-History and its associated file group for which ϱ was calculated. γ is the number of code clones detected in F.

Figure 5.1 shows the dot plot for the input sample of the Mozilla case study. The weak correlation between code clones and change couplings is stressed by the regression line that does not fit well in the dot plot. The assumption that a higher number of code clones has a more negative impact is not valid for our input sample.

We can identify several clusters of dots in the plot of Figure 5.1: There are some dots that lie on the regression line or closely to it (marked blue). These dots exhibit the property of a positive correlation. Contrary to the blue cluster, there are many file groups that have a small number of code clones, but their impact is strong, since they caused a high portion of all change couplings. Such dots are located on the plot away from the regression line near to the vertical axis on the upper half (marked green). An other cluster is formed by the four red marked file groups that have a ϱ value of 100%. In these file groups all change couplings were caused by structural changes in code clones. Such dots all lie on the horizontal line with the value 1.

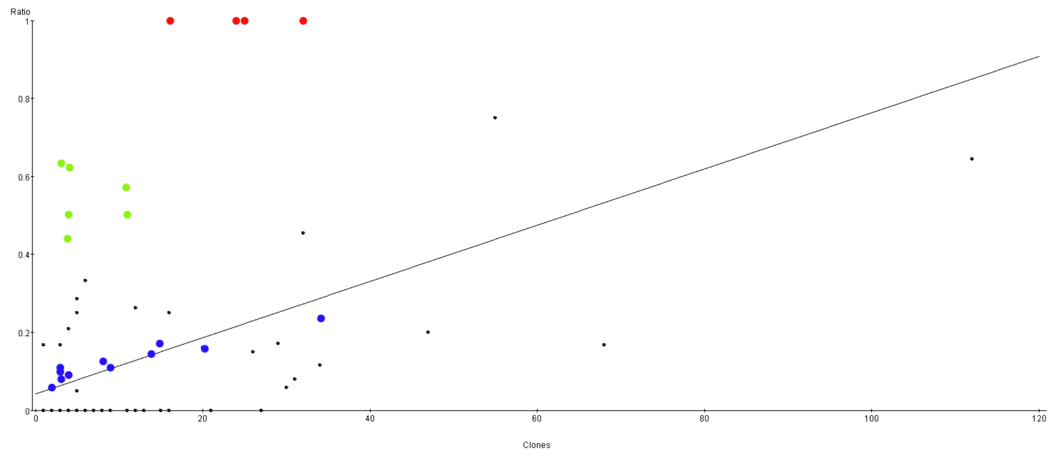


Figure 5.1: Sample Plot and Regression Function (Mozilla)

5.2.7 Conclusion

Two main conclusions emerge from the results of the analysis. First, the relation of code clones and change couplings in the input sample of the Mozilla case study can not be described by a statistical correlation or a simple linear regression model. The results are scattered too randomly in Figure 5.1. The parameters – a correlation coefficient of 0.4585 and R^2 value of 0.2102 – from the regression analysis indicate a weak correlation between the number of code clones and the ratio of change couplings caused by code clones and the total number of change couplings. Second, we can use the results investigate, compare and identify critical file groups. The results are an instrument for developers to gain insight in the state of their software system.

5.3 Java Case Study - Eclipse

The second case study was done with Java source code from the Eclipse project. The first part of the Java case study is concerned with the identification of suitable file groups. To achieve a certain consistency between both case studies, the same proceeding was applied, and similar considerations were made as in the Mozilla case study. We will therefore omit the details of the selection process and discuss only additional Eclipse and Java specific aspects:

1. **Input Selection**: Java has only `*.java` as source file extension. The Java top level entity `interface` fulfills a similar purpose as header files in C / C++ source code. A Java interface can only contain abstract methods and `final` fields. It is thus senseless to detect code clones in such interfaces. Because it is not possible the to filter out Java interfaces by file extensions, this is a manual process.

   Due to its highly extensible character, the Eclipse source code contains a lot of such Java interfaces. Everyone that wants to customize and extend Eclipse for his very own needs can use and implement those interfaces. Besides those interfaces the Eclipse source code contains (default) implementations of those interfaces to provide the initial functionality of the Eclipse SDK when it is distributed. The interfaces lie in open / public packages while the implementations lie in internal / private packages.[5] For example the package

[5]The concept of *internal* and *open* packages has nothing to do with the visibility modifiers *public* or *private* in Java. They

`eclipse/core/resources`

contains mostly interfaces for the Eclipse resource API. The package

`org/eclipse/core/internal/resources`

contains the equivalent (default) implementations for the resource API. Internal and open packages usually differ in the prefix `internal` in the CVS source code structure. This prefix denotes packages with default implementations. For this case study all packages were filtered out that contain only interfaces. This lead to the input structure, in which most of the packages were internal.

2. **Initial Clone and Coupling Data**: The remaining packages were subject to the CCFinder to identify those files that are affected by code clones. The EVOLIZERBASE was used to calculate the change coupling groups for these packages.
3. **Identification of Possible Candidates**: As in the Mozilla case study we chose the file groups that resulted from the clone detection run rather than change coupling groups.

We analyzed 262 file groups from 5 (sub-)modules of the Eclipse project. These 262 file groups lead to a total sum of 975 investigated files. We tracked the change behavior of 1879 code clones over 4616 change couplings in these source code files. Equally to the Mozilla case study we calculated the following metrics for **every investigated file group** respectively its associated File-History:

1. Number of files in the file group
2. Number of code clones: γ
3. Number of change couplings: δ
4. Total number of change couplings that were caused by code clones: σ
5. Ratio between change couplings caused by code clones and total number of change coupling: ϱ

We showed in Section 5.2.5 of the Mozilla case study how these results can be used to compare, assess and identify critical file groups. The same methodology can be used for this Eclipse case study. We will therefore skip this part and go straight forward to the regression analysis of code clones and change couplings.

5.3.1 Correlation of Code Clones and Change Couplings

Table 5.3 shows the frequency of ϱ values in the Eclipse case study. One can see two analogies to the Mozilla case study (cf. Table 5.2): First, again approximately 70% of all file groups have a ϱ value of 0%. Meaning that 30% of all inevstigated file groups have change couplings caused by code clones. Second, most file groups have a ϱ value between 1% - 30%.

In Figure 5.2 we plotted the number of code clones γ and the ratio between change couplings caused by code clones and total number of change couplings ϱ of all file groups against each other in a simple linear regression analysis. The analysis resulted in correlation coefficient of 0.4571. This is almost the same as in the previous case study with a coefficient of 0.4585. The

are a concept of how the Eclipse source code is organized. The idea is to strictly separate the open API from the initial Eclipse SDK implementation.

Value	Absolute Frequency	Frequency in Percentage
0	183	69.85%
0.01 - 0.1	39	14.89%
0.11 - 0.2	16	6.11%
0.21 - 0.3	13	4.96%
0.31 - 0.4	2	0.76%
0.41 - 0.5	6	2.29%
0.51 - 0.6	0	0%
0.61 - 0.7	2	0.76%
0.71 - 0.8	1	0.38%
0.81 - 0.9	0	0%
0.91 - 1.0	0	0%
> 1.0	0	0%
	262 Groups	100%

Table 5.3: Frequency of ϱ Values (Eclipse)

Eclipse input sample does not express significant correlation either. The linear regression model has a R^2 value of 0.2089. Meaning that linear function explains only 20% of the scattering of the input sample. The equation of the linear regression line is:

$$\varrho = 0.0076 \times \gamma(F) + -0.0016$$

where F corresponds to the File-History respectively its associated file group for which ϱ was calculated. γ is the number of code clones detected in F. We can again identify some dots that lie closely or on the regression line (marked blue in Figure 5.2), and some file groups where a relative small number of clone results in a high ϱ value. But overall the dots are scattered too randomly and too ambiguously in the plot – even more randomly than in the Mozilla case study (cf. Figure 5.1).

5.4 Evaluation

We processed two case studies in this chapter. Both times the regression analysis could not establish a statistical correlation between code clones and change couplings. A higher number of code clones does not cause file groups to be change coupled more often. Most of the change couplings must have other reasons than code clones in our input samples: For instance, the files contribute to the same functionality of the system and are then change coupled together, when it needs to be adapted or updated, there is a strong code owner ship in the project, or updates in comments, or changes in the license header of source code file caused the change couplings.

Nevertheless the results from the CloneAnalyzer are not useless. The case studies also revealed certain specific file groups that have a significant number of change couplings caused by code clones. We can use the value of ϱ in combination with other measures and metrics to explore the source code to identify such critical file groups. The analysis based on these metrics and measures provide a mechanism to investigate the state of a software system regarding code clones and change couplings. However the decision whether a file group that was identified as ”critical” is really a refactoring candidate still needs the judgment of developers.

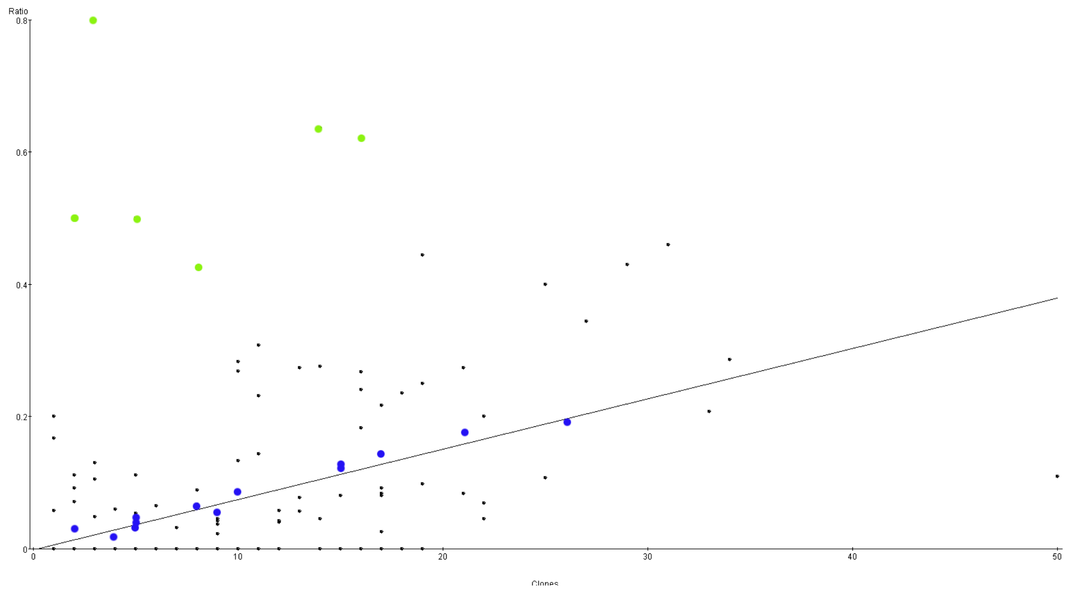


Figure 5.2: Sample Plot and Regression Function (Eclipse)

5.5 Limitation of the Results

During the manual inspection of the results in the evaluation phase we detected some side effects due to shortcomings and problems in the used technologies. These side effects biased and blurred unfortunately some of the results in our case studies. We will address these problems in this section and show their negative impact on the results.

- **CDT**: The CLONEANALYZER uses the CDT DOM-Parser[6] to construct the abstract syntax tree from C/C++ source code. In some circumstances the CDT is not able to parse the source code properly, and cannot construct the correct abstract syntax tree or certain nodes out of the source code. This leads to two problems: First, the API to extract the position of abstract syntax tree nodes within a source code file returns `NULL` instead of concrete values for offset and length. As the case may be, it is not possible to determine all necessary source code entities that correspond to a code clone, since we lack the information about the position of these falsely parsed nodes. Second the CDT parser creates an error node for every incorrectly parsed source code entity with a specific message *e.g., Syntax error encountered in file: nsMathMLmsqrtFrame.cpp at offset: 6'115*. Both problems make the CLONEANALYZER construct the regular labeled trees incorrectly. Structural changes then may be detected by the CHANGEDISTILLER although the source code did not change. This problem could neither be traced down to its cause nor solved up to now. It might disappear with upcoming releases of the CDT.[7]

- **CCFinder**: As showed in Section 4.2.2 the CLONEANALYZER uses clone ID, filename and the revision number appendix to track clone a code clone over different revisions. A code clone has a specific token length in the CCFinder. If a code clone experiences a structural change that changes its token length significantly, a new code clone with a new clone ID is reported by the CCFinder. In such cases the CLONEANALYZER cannot link between these

[6] org.eclipse.cdt.core.dom.ast

[7] The CLONEANALYZER uses CDT 3.0.2 (9th of February 2006) for constructing the regular labeled trees.

”two” code clones, because it finds two different clone IDs. Instead of detecting a structural change within a code clone, the CLONEANALYZER identifies a new clone in the relating revision of the affected file. Structural changes in code clones get ”lost”. To overcome this problem a similarity measure is required that checks, whether a newly detected code clone is indeed a new clone or just an old clone that has been changed significantly.

- **Offset/Length Calculation**: In certain circumstances the offset and length calculation of code clone tokens in the CCFinder differers from the offset and length calculation of abstract syntax tree nodes in Eclipse by a small number of characters. Due to this difference there is a certain inaccuracy when searching the source code entities that are covered by a code clone. The abstract syntax tree of the cloned source code fragment is build incompletely. If this problem occurs between two revisions, the CHANGEDISTILLER may falsely compare different abstract syntax trees and reports a code clones change. The problem could be solved by using *Begin-Line* and *End-Line* to denote the textual range of code clone instead of *offset* and *length*.

Chapter 6

Conclusion and Final Remarks

This chapter closes the thesis with an overall review and summary of its findings, contributions and lessons learned. It gives some ideas for future work which emerged while writing the thesis.

6.1 Contributions

The overall framing context of this thesis is the investigation of code clones and change couplings. Code clones compromise a major bad smell, and are said to have a negative impact on the maintenance and evolution of a software system. Change couplings refer to a group of files that have been changed together at the same time. A change coupling group occurring constantly during the evolution process might present a problem, and a refactoring should be considered. A positive correlation between code clones and change couplings has been assumed for some time, but could not be verified by recent research activities. The novelty of this thesis is the combination of different technologies to examine code clones and change couplings. We use:

- **Code Clone Data**: Such data is obtained by code clone detection mechanisms and draws a picture of the situation concerning duplicated source code fragments in a software system.
- **Release History Data**: This historical information describes important facts in the evolution of a software system. Its analysis gives useful information to reason about the future evolutionary behavior of a system.
- Abstract Syntax Tree (AST) Differencing: Tracking source code changes on textual basis lacks the information of the changed source code entity. Moreover this approach is line based and has therefore a limited level of granularity. In this thesis we use an abstract syntax tree differencing algorithm to meet the claims of ”*fine-grained source code change analysis*”. The advantage of our approach is, that were able to identify exactly the source code entity that changed. Because we are detached from textual aspects of source code, we achieve deeper level of granularity. Moreover we can distinct between structural and non structural changes.

The thesis uses data from these technologies to provide a more detailed view about the evolutionary behavior of code clones and change couplings. The concrete results of this thesis are:

- A framework to combine the mentioned technologies and merge them into an analysis of the relation between code clones and change couplings.

- Part of the framework is a set of measures and metrics to assess file groups in the context of code clones and change couplings.

- A implementation of the framework using concrete tools and technologies.

- The core of this implementation is the CLONEANALYZER. An Eclipse Plug-In that allows an automated analysis of a software system.

- An evaluation of the framework as well as of the implementation by means of two real existing software project (Mozilla: C / C++; and Eclipse: Java).

The benevolence of the framework is, that it does not depend on concrete tools, technologies or programming languages. It is rather a recipe guiding everyone that wants to use this framework in other circumstances and different requirements than for our case studies. It is also possible to use only certain aspects of the framework *e.g.*, the metrics or the technology combination part.

The metrics further contain a classification system to evolve and describe the change behavior of code clones over time. We used these metrics and the classification system to investigate the relation of code clones and change couplings.

The CLONEANALYZER implements our abstract framework. It is able to analyze C / C++ and Java software systems. We used the automated mechanisms of the CLONEANALYZER to calculate the results of the metrics in our case studies.

6.2 Lessons Learned

The combination of code clone data, historical system release information and tree differencing algorithms emerged to be useful during the evaluation of the case studies. Using this data we can exactly determine when a code clone experienced structural changes in the release history of a software system. We are able to tell if a code clone remains stable or changed constantly; whether changes took places early or recently in the release history; and whether changes in code clones caused a change coupling in the affected files. This information is useful when reasoning about the state and the future evolutionary behavior of a system.

In both case studies we could not verify a statistical correlation of code clones and change couplings. The results were scattered too randomly and too ambiguously. For instance, we found file groups where a relatively small number of code clones caused a significant portion of all change couplings. On the other hand, there were file groups in our results in which a large number of code clones caused a negligible amount of change couplings. Most of all file groups – approximately 70% in both case studies – did not even possess change couplings caused by code clones. We can say that the impact of code clones is limited in our case studies. There are other reasons for change couplings.

Despite the fact that there is no a statistical correlation in our case studies, the results clearly highlighted several file groups were code clones caused change couplings. We could use the computed results during the analysis to identify those critical file groups, for which a refactoring should be considered.

6.3 Future Work

This thesis has proposed framework with methodology to merge code clone data, historical system release information and tree differencing algorithms into an analysis of the relation of code clones and change couplings. The second major result was the CloneAnalyzer; an implementation of our framework realized as an Eclipse Plug-In. Both outcomes give room for future work.

The evaluation of the results of the case studies revealed several limitations and problems in the technologies of our implementation. We addressed them and showed their negative impact on the results in Section 5.5. The CloneAnalyzer is only one way to implement the framework of this thesis. It is therefore necessary to compare and evaluate other implementations against each other in future work. This could lead to better and more suitable outcomes. Some work towards an improvement of the used technologies is already being done. The ChangeDistiller is analyzed and enhanced in an other diploma thesis.[1]

CDT 4.0 is planned to be released in summer 2007. This will be a major release that will feature better multi-language support, project creation templates, amongst other improvements.[2] This release may solve the problems of the CDT DOM parser too.

Another task is to implement the EvolizerBase for other version control system than CVS *e.g.*, SVN. This would enlarge the range of application for the CloneAnalyzer.

The framework is limited to the detection of source code changes in code clones. Up to now it does not consider the way a code clone changed. A major improvement is to include the taxonomy for source code changes and the information about their change significance level as described in [Fluri and Gall 2006].

The case studies could be extended in future work on more than just two projects. This might point out interesting peculiarities of different projects in the context of code clones and change couplings.

[1] Improving Abstract Syntax Tree based Source Code Change Detection: http://seal.ifi.unizh.ch/137/
[2] http://wiki.eclipse.org/index.php/CDT/planning/4.0

Appendix A

Glossary

API : Abbr. for ↦ Application Programming Interface

Application Programming Interface : Interface that a computer system, library or application is providing for other programs in order to be able to access its functionality and services. http://en.wikipedia.org/wiki/API.

Bug Tracking System : Software application for collecting and documenting bugs of a software system in a (logical) central repository. For instance, a bug tracking system allows to categorize bugs according to their severity, or to specify well defined workflows for handling bugs.

Bugzilla : ↦ Bug Tracking System developed and used by the ↦ Mozilla-Project.
Bugzilla Website: http://www.bugzilla.org/

C/C++ Development Tool : Plug-In providing additional support for the development of application written in C/C++ with ↦ Eclipse.
CDT-Project Website: http://www.eclipse.org/cdt

CDT : Abbr. for ↦ C/C++ Development Tool

CloneCoverage : According to [Geiger 2005] the *CloneCoverage* for two files *A* and *B* is defined as

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

and

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

where $CloneLines(X,Y)$ is the number of lines in file X that are clones of lines in file Y and $NCLOC(X)$ is the number of lines of source code in file X not counting comments and blank lines.

Concurrent Versions System : Implementation a version control system for files, mostly source code files. It is very popular among the open source community and released under the GNU General Public License
Official GNU-CVS Website: http://www.nongnu.org/cvs/

CVS : Abbr. for $\mapsto$ Concurrent Versions System

Eclipse : Eclipse is an freely available and powerful open source $\mapsto$ IDE-Framework developed by the Eclipse Foundation and IBM. It is primarily used as a Java-IDE. Several Plug-Ins provide support for other programming languages *e.g.,* $\mapsto$ CDT. Eclipse as well as its Plug-Ins are written in Java.
Eclipse-Project Website: http://www.eclipse.org

IDE : Abbr. for $\mapsto$ Integrated Development Environment

Integrated Development Environment : Computer software that assists computer programmers to develop software.
http://en.wikipedia.org/wiki/Integrated_development_environment

Java : Refers to the Java-Technology developed by Sun Microsystems. Its main components are the object-oriented programming language Java and the Java-Platform. Based on these two components other components are available *e.g.,* Java 2 Platform Enterprise Edition.
Java Technology Website: http://java.sun.com/

JDT : Abbr. for $\mapsto$ Java Development Tools

Java Development Tools : Is an Eclipse subproject. It provides very powerful and sophisticated support for developing applications in Java with Eclipse. Since Eclipse is primarily used $\mapsto$ as Java-IDE, the Java Development Tool is initially shipped with the Eclipse SDK distribution.
JDT-Project Website: http://www.eclipse.org/jdt

Mozilla : Mozilla is not a clearly defined term. It is often referred to other terms such as the Mozilla-Foundation, Mozilla-Project or a collection of freely available open source applications. Mozilla is concerned with the development of various open source software application. Well known products are the browser Firefox, the mail client Thunderbird or the $\mapsto$ bug tracking software Bugzilla. More information can be found on the official Mozilla webpage http://www.mozilla.org

Mozilla-Project : $\mapsto$ Mozilla

Precision : According to [Geiger 2005] the precision is defined as the ratio between the number of real clones among the reported candidates and the total number of reported candidates.

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

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

Recall : According to [Geiger 2005] the Recall is defined as the ratio between the number of real clones among the reported candidates and the number of clones that are in the system.

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

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

Standard Widget Toolkit : Implementation of a library for building graphical user interfaces in Java. SWT is developed by IBM in conjunction with Eclipse and uses the native graphical elements of the underlying operating systems.
SWT Project-Site: http://www.eclipse.org/swt/

Subversion : Implementation of an open source version control system. It is sometimes unofficially named as the successor $\mapsto$ of CVS.
Subversion Website: http://subversion.tigris.org/

SVN : Abbr. for $\mapsto$ Subversion

SWT : Abbr. for $\mapsto$ Standard Widget Toolkit

Appendix B

File Formats

B.1 Format of Positional Transformed Clone Data Output File

The file is saved as a *.txt*-File and named *plaindata.txttransformed*

- CloneDataFile:= PreprocessedScriptOption FileSection ClonePairSection
- PreprocessedScriptOption:= ”preporcessed_script:” ScriptString ”\n”
- FileSection:= ”#begin{file description}” ”\n” Files ”\n” ”#end{file description}” ”\n”
- Files:= File*
- File := FileIDString ”\t” FileLineLengthString ”\t” FilePathString ”\n”
- CloneSection:= ”#begin{clone}” ”\n” Clones ”\n” ”#end{clone}” ”\n”
- Clones:= Clone*
- Clone := CloneSetIDString ”\t” File1Fragment ”\t” File2Fragment ”\n”
- File1Fragment:= FileFragment
- File2Fragment:= FileFragment
- FileFragment:= FileIDString ”\t” CodeCloneFragmentStart ”\t” CodeCloneFragmentEnd
- CodeCloneFragmentStart:= CodeClonePosition
- CodeCloneFragmentEnd:= CodeClonePosition
- CodeClonePosition:= BeginLine”,”BeginOffset ”\t” EndLine”,”EndOffset

B.2 Format of Statistic File

The stats file is saved as a .*txt*-File and named *logFile*.

- StatsDataFile:= FileHistorySection*
- FileHistorySection:= FileSection CloneChangeSection
- FileSection:= ”beginFiles{” ”\n” Files ”\n” ”}”
- Files:= File*
- File:= FilePathString
- CloneChangeSection = ”beginChangeAnalysis{” ”\n” ”#Files, #SharedClones, CloneSetId, #SharedTransactions, #AffectedSharedTransactions” ”\n” cloneSection ”\n” ”}”
- cloneSection:= CloneChangeAnalysis*
- CloneChangeAnalysis = NumberOfFiles ”,”NumberOfSharedClones”,” CloneSetIDString ”,” NumberOfSharedTransactions ”,” NumberOfAffectedTransactions

  where

- **NumberOfFiles** denotes the number of files of the underlying File-History
- **NumberOfSharedClones** denotes the number of code clones of the underlying File-History γ.
- **CloneSetIDString** denotes the unique ID assigned by the CCFinder
- **NumberOfSharedTransactions** denotes the number of change couplings of the files of the underlying File-History δ
- **NumberOfAffectedTransactions** denotes the number of change coupling caused by a specific code clone **CloneSetIDString** ε.

Parallel there is an output log file created documenting the progress of the statistics file calculation. This log files is saved as a **.txt* and named *outputlog*.

Appendix C

List of Used Tools

- **ChangeDistiller**
- **CCFinderX**: http://www.ccfinder.net
- **CDT**: http://www.eclipse.org
- **CloneAnalyzer**
- **Cygwin with c/c++ packages**: http://www.cygwin.com
- **Eclipse (& JDT)**: http://www.eclipse.org
- **EvolizerBase**
- **Hibernate**: http://www.hibernate.org
- **Java 5.0** *Tiger-Release 1.5*: http://java.sun.com

Appendix D

Content of the Delivered CD-ROM

Each copy of this thesis includes a CD-ROM. This CD-ROM contains:

- **Clone_Analyzer.zip**: The Eclipse project folder of the CloneAnalyzer
- **thesis.pdf**: Copy of the written elaboration of this thesis
- **abstract.pdf**: English version of the abstract of this thesis
- **Zusfsg.pdf**: German version of the abstract of this thesis

References

[Bartels and Jaehne 2000] Korbinian Bartels and Klaus Jaehne. Ganz kurze Einfuehrung in CVS. 16th of July 2000.
http://kj.uue.org/papers/cvs-handout/

[Beaton 2006] Wayne Beaton. Eclipse Platform Technical Overview. Revision 3.1, Updated for Eclipse 3.1, 19th of April 2006, ©2006 International Business Machines Corp. (IBM).
http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html

[Boehm 1981] Barry W. Boehm. Software Engineering Economics. Englewood Cliffs, N.J., Prentice Hall. 1981.

[Fahrmeir *et al.* 2001] Ludwig Fahrmeir, Rita Kuenstler, Iris Pigeot, and Gerhard Tutz. Statistik - Der Weg zur Datenanalyse, 3rd Edition. Springer Verlag Berlin Heidelberg, Germany, 2001.

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

[Fluri and Gall 2006] Beat Fluri and Harald C. Gall. Classifying Change Types for Qualifying Change Couplings. In *Proceedings of the 14th International Conference of Software Comprehension (ICPC)*, to appear, Athens, Greece, June 2006. IEEE Computer Society Press.

[Fluri *et al.* 2005] Beat Fluri, Harald C. Gall, and Martin Pinzger. Fine Grained Analysis of Change Couplings. In *Proceedings of the 5th International Workshop on Source Code Analysis and Manipulation*, pages 66-74, Budapest, Hungary, September 2005. IEEE Computer Society Press.

[Fowler *et al.* 1999] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. Refactoring: Improving the Design of Eisting Code. Addison-Wesley, 1999.

[Geiger 2005] Reto Geiger. Evolution Impact of Code Clones - Identification of Structural and Change Smells based on Code Clones. Diploma Thesis, University of Zurich, Switzerland, 2005.

[Giger 2006] Diploma Thesis Specification for Emanuel Giger. Evolving Code Clones, University of Zurich, Switzerland, 25th of January 2006.
Thesis Description: http://seal.ifi.unizh.ch/113/

[Glinz 2005] Martin Glinz. Qualitaetsplanung und -Messung, Kapitel 8 aus Folienskript zur *KV Software Engineering* SS05, University of Zurich, Switzerland, ©2004 Martin Glinz.
http://www.ifi.unizh.ch/groups/req/ftp/kvse/kapitel_08.pdf

[Glinz 05/06] Martin Glinz. Warum ist Requirements Engineering wirtschaftlich?, Kapitel 5 aus Folienskript zu *Spezifikation und Entwurf von Software* WS 05/06, University of Zurich, Switzerland 2005/2006, ©2001, 2004 Martin Glinz.
http://www.ifi.unizh.ch/groups/req/ftp/ses/kapitel_05.pdf

[Grubb and Takang 2003] Penny Grubb and Armstrong A. Takang. Software Maintenance - Concepts and Practice. World Scientific, 2nd Edition, 2003.

[Jetter and Wuersch 2005] Andreas Jetter and Michael Wuersch. Evolizer Base - CVS Importer Documentation. Universityof Zurich, Switzerland, 26th of October 2005.

[Kamiya 2005] Toshihiro Kamiya. File format and command line of CCFinderX. National Institute of Advanced Industrial Science and Technology, Japan, 15th of December 2005.
http://www.ccfinder.net/doc/FileFormats-en.pdf

[Kamiya 2006] Toshihiro Kamiya. CCFinderX Quick Guide. National Institute of Advanced Industrial Science and Technology, Japan, 22nd of February 2006.
http://www.ccfinder.net/doc/QuickGuide-en.pdf

[Krahn and Rumpe 2006] Holger Krahn and Bernhard Rumpe. Grundlagen der Evolution von Software Architekturen. In Ralf Reussner and Wilhelm Hasselbring (editors) *Handbuch der Software-Architektur*, Chapter 6. dpunkt-Verlag, Germany, 2006.

[Lehman and Belady 1985] Meir M. Lehman and Laszlo Belady. Program Evolution Process of Software Change. Academic Press, 1895.

[Marjanovic 2006] Dane Marjanovic. Developing a Meta Model for Release History System. Diploma Thesis, University of Zurich, Switzerland, 2006.

[Springgay *et al.* 2002] Dave Springgay, Jin Li, Julian Jones, and Grag Adams. Eclipse User Interface Guidelines. Last Updated 27th of February 2002, ©2001-2002 Object Technology International, Inc. and others.
http://www.eclipse.org/articles/Article-UI-Guidelines/v200202/Contents.html

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