Evaluating the effect of code duplications on software maintainability

Tibor Bakota

Department of Software Engineering
University of Szeged

Supervisor: Dr. Tibor Gyimóthy

October 2012
Szeged, Hungary

A dissertation submitted for the degree of doctor of philosophy of the University of Szeged

University of Szeged
Ph.D. School in Computer Science

"The greatest education in the world is watching the masters at work.'

Michael Jackson

Acknowledgements

I have been very lucky in my journey where I could watch the experts at work and learn from them not just what to think, but also how to think. I am really grateful to all those who helped me either with their comments, ideas or suggestions. They improved not just my work, but also opened my mind and broadened my knowledge.

First, I would like to thank my supervisor Dr. Tibor Gyimóthy who helped me by providing useful ideas, comments and interesting research directions. I would like to thank my article co-author and mentor, Dr. Rudolf Ferenc, for guiding my studies and teaching me a lot of indispensable things about research. He inspired me in times when I needed motivation and kept me on the right path. My thanks also go to my colleagues and article co-authors, namely Dr. Árpád Beszédes, Dr. István Siket, Dr. Lajos Fülöp, Dr. Judit Jász, Péter Siket, Péter Hegedűs, Dr. Lajos Schrettner, Dr. Tamás Gergely, Claudio Riva, Jianli Xu, Maarit Harsu, Kai Koskimies, Tarja Systa, Péter Körtvélyesi, László Illés, Gergely Ladányi, Milán Imre Gyalai and Dániel Füleki. I would also like to thank the anonymous reviewers of my papers for their useful comments and suggestions. And I would like to express my thanks to David P. Curley for reviewing and correcting my work from a linguistic point of view.

I wish to express my gratitude to my parents as well for providing a pleasant background conducive to my studies, and also for encouraging me to go on with my research. Last, but not least, my heartfelt thanks goes to my wife Mónika for providing a vital, affectionate and supportative background during the time spent writing this dissertation.

Tibor Bakota, 2012

Contents

List of Figures

List of Tables

To my wife, Mónika

and my daughter, Szandra.

"Sometimes it's easier to pretend that something is not happening rather than deal with it."

Cherie Currie

Chapter 1

Introduction

These days, software is a part of our everyday lives. Whether we know it or not, we rely on software systems to an ever increasing extent. From the watch on our wrists, through the smart phones and cars we use to the airplanes and nuclear plants; all these are being controlled and run by some software system. About 65 % of the U.S. population use embedded software in the form of smart phones, digital cameras and watches, car safety systems, entertainment devices, and so on [50]. Software does not just exist to make our lives easier; our lives may depend on them. We truly believe, for example, that the airbag will open in case of an accident, that the airplane will land safely and a nuclear reactor operate in complete safety.

An increase in the demand for new and more reliable functionalities has led to an enormous expansion in the software industry. In 2011, the 1,000 largest companies in the U.S. employed over one million software engineers, who are, as part of their daily jobs, continuously building new systems or improving existing ones [50]. Meanwhile, business demands keep a constant pressure on IT leaders to deliver the products as early and as cheaply as possible. This race generated by market sometimes forces IT leaders and software engineers to make compromises or take short cuts, i.e. to trade long term quality and safety for short term benefits.

One of the most controversial methods for increasing developer's productivity is the so-called copy&paste technique. When a software developer needs to implement something for which he knows that a similar code fragment already exists in the codebase, he might be tempted to simply make a copy of the existing part. Although this approach can reduce software development time, the price in the long term will usually be paid in terms of increased maintainability costs. One of the primary concerns is that if the original code segment needs to be corrected, all the copied parts need to be checked and modified accordingly as well. By inadvertently neglecting to change the related duplications, the programmers may leave bugs in the code and introduce logical inconsistencies. However, code duplication is not always a problem. Some researchers point out that there exist situations when duplicating code could even be beneficial, and clones should not always be considered harmful [55].

Quantifying source code maintainability is essential for measuring the effect of code duplications. Aggregating a measure for maintainability has always been a challenge in software engineering. The non-existence of formal definitions and the subjectiveness of the notion are the major reasons for it being difficult to express maintainability in numerical terms. Although the ISO/IEC 9126 standard [44] provides a definition for maintainability, it does not provide a standard way of quantifying it. Many researcher exploited this vague definition and it has led to a number of practical quality models proposed [43, 83, 13, 7].

For example, Heitlager et al. [43], who are members of the Software Improvement Group (SIG), proposed an extension of the ISO/IEC 9126 model that uses source code metrics at a low-level. Their metric values are split into five categories, from poor (– –) to excellent (++). Correia and Visser [28] presented a benchmark that collects measurement data from a wide variety of systems. This benchmark provides a systematic comparison of the technical quality of software products. Alves et al. [3] presented a technique for deriving metric thresholds from benchmark data, which is used to derive more reasonable thresholds for the SIG model as well. Correia and Visser [29] introduced a certification method that is also based on the SIG quality model. Their method allows one to certify the technical quality of software systems. Each system can get a rating from one to five stars (where "– –" corresponds to one star and "++" to five stars). Baggen et al. [8] refined this certification process by repeatedly performing a calibration of the thresholds based on this benchmark. The original SIG model uses binary relation among the system properties and high level quality characteristics. Correia et al. [27] carried out a survey to elicit weights for their model. Although the survey was completed by IT professionals, the authors eventually concluded that using weights did not improve their quality model because of the lack of consensus among developers.

In this study we propose a novel method for deriving maintainability models that in many senses differs from the state-of-the-art research achievements described above and overcomes some of the existing problems. Our method handles the ambiguity issue that arises from the different interpretations of key notions and it produces models that express maintainability objectively. While the ambiguity issue is dealt with by applying probabilistic techniques, the objectivity aspects of the method follow from using statistical benchmarks.

The importance of maintainability lies in its close connection with the cost of changing the behavior of the software. Modelling software development cost has been an intensive research area for a long time [1, 80, 56, 30, 19]. Effort estimation is important not just for software developers [85], but for system operators as well [20]. While summaries of research results achieved in the last thirty years are available [31, 52], the area still shows many promising future topics for research [94]. Relevant comparison studies of different techniques [51] in various domains [21, 22, 68, 86, 14] also exist. Currently, the ways of conducting development effort estimation [18] range from hands-on approaches [51] through benchmark approaches [95] to model-based approaches. Several methods using a combination of the above techniques also exist [90, 59]. Many studies also seek to compare the different approaches. For example, Radlinski and Hoffman [91] compared several machine learning algorithms using a benchmark for effort prediction. Mair et al. [76] examined a lot of papers concerning analogy and regression-based techniques for cost estimation.

In this dissertation, we present a formal mathematical model based on ordinary differential equations for modelling the relation between source code maintainability and development cost. In contrast to other studies, we begin by stating simple and reasonable assumptions, and after establishing their formal mathematical representations, solutions are derived and validated on real world systems. We employ the maintainability model described earlier to compute source code maintainability. It turns out that under some reasonable assumptions, the relation between cost and maintainability may even be exponential; i.e. source code maintainability has a great influence on the development cost.

As code duplications are generally considered to be one of the chief enemies of maintainability, clone metrics play an important role in our maintainability model as well. A plethora of clone detection algorithms exist nowadays that can help reduce the risks of making inconsistent changes. By the conservative approach, after the clones are detected they should be evaluated manually. Unfortunately, this approach cannot be applied in practice, since large software systems may have several thousands of duplications present in the software code. It has been discovered by researchers that the real threat does not lie in the existence of duplications but rather the worries are related to their evolution. Therefore, tracking the duplications across different versions is essential in order to evaluate their influence on maintainability and to have an efficient clone management technique. Constructing algorithms for tracking clone instances across different versions of a software system has only recently become an active research area.

Antoniol et al. [4], for instance, applied time-series techniques to model the changes in the average number of clones per function in a system. Later they studied the evolution of code clones in the Linux Kernel using their metric-based approach [5]. Merlo et al. [79] extended the concept of similarity of code fragments in order to quantify similarities at the release/system level. Kim et al. [57] defined the *cloning relationship* between two clone classes based on the lexical similarity of their representatives. Aversano et al. [6] analyzed the so-called co-changes, which are changes carried out by the same author, with the same notes, and within 200 seconds. Duala-Ekoko et al. [32] proposed the notion of an abstract Clone Region Descriptor (*CRD*), which describes the clone instances within methods in such a way that it is independent of the exact text or their location in the code. Geiger et al. [36] studied the relation between code clones and change couplings (files which are amended at the same time, by the same author, and with the same modification description). Krinke [62] used a version control system of open source systems to identify changes applied to code duplications. Their study revealed that clone classes were consistently changed in roughly half of the cases. Krinke also showed that classes that had been changed inconsistently earlier and were consistently changed later, were comparatively rare. Göde et al. [37] presented an incremental clone detection algorithm that detects clones based on the results of a previous revision analysis. Their algorithm creates a mapping between clones of one revision to the next, providing information about the addition and deletion of clones. They focus more on the gain in performance (they achieved a significant improvement) than on tracking the evolution of individual clone fragments.

To evaluate the effect of duplications on maintainability, we propose another method for tracking clones through the consecutive versions of an evolving piece of software. Our method differs in some ways from the above-mentioned approaches. The most significant difference is probably that we reduced back the issue of clone tracking to an optimization problem. Therefore, the mapping between the duplications is optimal in some sense. We also propose a highly efficient and practical code duplication management method that can help reduce maintenance efforts and risks of inconsistent changes being made. The key concept lies in the notion of *clone smells*, which represent different categories of suspicious clone evolution patterns. Clone smells can be used to identify those occurrences of duplications that could really cause problems in the future versions, i.e. the hazardous ones. The list of risky places is several orders of magnitudes smaller than the list of all duplications in a system, so a manual evaluation and elimination is more straightforward to perform.

1.1 Summary by results

The main contributions of this study are summarised as follows. First, we propose a metric-based probabilistic approach for modelling source code maintainability. Afterwards, we introduce a formal mathematical theory for modelling the relation between source code maintainability and development cost. Next, we present a novel method for tracking the lifetime of code duplications, and then we propose a classification of clone evolution patterns.

We state three thesis points in this dissertation, where the contributions of the author are clearly shown.

1.1.1 Development of a probabilistic source code maintainability model

Here, we present an experimental result concerning the expressiveness of low-level source code metrics in terms of high-level quality characteristics. We will show that, with the help of an appropriate modelling technique, the subjective notions of high level maintainability characteristics (and therefore maintainability too) can be modelled efficiently by using low-level source code metrics. We will also show that, unfortunately, these kinds of metric-based models may not be portable; i.e. the models trained on one system may not be used directly on other systems. To resolve this and several other difficulties, we present a novel approach for building metric-based models that handles some basic requirements one would expect from such a model. The thesis consists of the following research contributions.

An approach for constructing probabilistic maintainability models

We developed a novel approach for building metric-based maintainability models that fulfil most of the basic desired requirements. The model should be

- *Interpretable* – applying the model should provide information for high-level quality characteristics that is meaningful; i.e. conclusions can be drawn with the help of it.
- *Explicable* – there should be a way to efficiently evaluate the root causes; i.e. a simple way to break down information obtained for high-level characteristics to attributes or properties.
- *Consistent* – the information obtained for higher level characteristics should not contradict lower level information.
- *Scalable* – the model should provide valuable information even for large systems in a reasonable time.
- *Extendible* – it should be possible to extend the model with new characteristics and its attributes.
- *Reproducible* – applying the model on the same system twice should yield the same/similar information about the quality characteristics.
- *Comparable* – information obtained for quality characteristics of two different systems should be comparable and should correlate with the intuitive meaning of the characteristics.

The approach we adopt is based on a database of metrics obtained from a large set of software systems (i.e. benchmark) and it uses complex statistical techniques to aggregate the high-level quality attributes for a particular system. The resulting models integrate expert knowledge, handle ambiguity aspects, and manage *goodness* functions, which are continuous generalisations of threshold-based approaches.

A metric-based maintainability model for Java systems

Making use of the method described above, we constructed an instance of a metric-based probabilistic maintainability model for Java systems. The model consists of nine low-level source code metrics, five medium-level aggregate nodes and five high-level quality characteristics and sub-characteristics. A benchmark containing source code metrics of 100 Java systems was created and used for computing the goodness functions at a low level. The aggregation makes use of opinions of 28 experts who work in different areas of IT.

Validation of the model on real world systems

We validated the model on real world software systems, and found that changes in the results of the model reflect the development activities; i.e. during development the maintainability decreased, while during maintenance it increased. We also found that although the model rankings were different from the grades given by the developers, they still show relatively high correlations with each other.

The author's contribution to the thesis point was the development of the formal mathematical background of the approach and the implementation of the core statistical modules that were required to perform the aggregation. He also participated in the construction of the particular model used for the evaluation of the approach and in devising the methodology of the empirical validation.

1.1.2 Establishing a cost model based on source code maintainability

We developed a pioneer approach for modelling software development cost in terms of source code maintainability. The thesis consists of the following research contributions.

A formal mathematical model for relating cost and maintainability

We defined a system of ordinary differential equations for modelling the relationship between source code maintainability and development cost. The formal mathematical equations arise from two simple assumptions:

1. When making changes to a software system without explicitly seeking to improve it (e.g. adding new functionalities), its maintainability will decrease, or at least it will remain unchanged.

2. Performing changes in a software system with lower maintainability is more expensive.

We introduced the notion of an *erosion factor* – a vital parameter of the model – which measures the amount of "damage" caused by changing source code lines of a software. After the estimates for the parameters are available, predictions for the future can be obtained from the model.

Validation of the model on real world systems

We evaluated the model on five software systems implemented in the Java programming language. An analysis of the empirical data shed light on the following important points:

- The maintainability of an evolving software package decreases over time.
- Maintainability and development cost are related to each other in an exponential way, with a high correlation.
- The model is able to predict future development costs based on estimated rate of change of the code, to a good accuracy.

The author's contribution to the thesis point was the development of the formal mathematical background of the approach and laying down the methodology of the empirical validation.

1.1.3 Assessment of code duplications from a code evolution perspective

In this thesis we focus on the issue of code duplications and their effect on source code maintainability, followed by the introduction of an efficient clone management method. This point of the thesis consists of the following research contributions.

A novel approach for tracking the evolution of code clones

We developed an efficient algorithm for tracking duplications across subsequent versions of a software system. Tracking the evolution of individual clones is essential from the viewpoint of source code maintenance. We defined a heuristic function called the *evolution mapping* between two particular code fragments taken from different versions of the same system. The mapping between the clones is trivial in some special cases, but in general, a sophisticated approach is required. We propose a similarity distance function to measure the likelihood of two code fragments having an evolutionary relationship with each other. The optimal mapping is then obtained by solving the corresponding optimisation problem. We extended the mapping to the level of clone classes and showed that the extension resulted a well-defined mapping.

A classification of clone evolution patterns

We define the notion of *clone smells* which, similar to bad code smells, refers to particular code parts that should be further inspected manually. The smells are defined based on the possible categories of clone evolution patterns. We showed that clone smells may serve as a basis for an efficient clone management technique that reduces risks and maintenance efforts arising from the duplications. We validated the usefulness of clone smells on different open source software systems. We concluded that

- The approach allows the developers to find and manually evaluate the risky duplications.
- More than the half of the reported smells are caused by inconsistent code changes.
- Inconsistency is more likely to be introduced; consistency is less likely to be restored.
- Inconsistent changes may uncover coding problems unintentionally remaining in the code.

The relation between clones and coupling

We analysed the relationship between cloning and coupling in software systems. We found that they generally have an inverse relationship with each other; in other words improving one can worsen the other. It follows that, contrary to some ideas about using only coupling metrics for measuring maintainability, any model applied should take both aspects into consideration.

The author's contributions to the thesis point are the following:

- The development of the methodology of relating code duplications.
- An implementation of the required libraries and performing an optimisation of the weights.
- The formal definition of clone smells.
- The implementation of software tools required to perform the extraction of clone smells.
- The extraction of clone smells from a large number of consecutive revisions of two open source systems.
- The manual evaluation of a large number of reported clone smells.

Below, Table 1.1 summarises which publications cover which thesis points.

$\mathcal{N}o.$	[42]	[12]	[10]	[102]	[103]	[11]	[105]	[106]
1.	●	●	●				●	●
2.				●				
3.					●	●		

Table 1.1: The relation between the thesis topics and the corresponding publications

"Knowledge is never useless."
Anne McCaffrey

Chapter 2

Background

2.1 Maintainability, entropy and erosion

In order to make the right decisions about estimating the costs and risks of a software change, it is crucial for the developers and managers to be aware of the quality attributes of their software. The ISO/IEC 9126 standard [44] defines six high-level product quality characteristics that are widely accepted both by industrial experts and academic researchers. These characteristics are: functionality, reliability, usability, efficiency, maintainability and portability. Maintainability is probably the most attractive, noticeable and evaluated quality characteristic of all (for this reason, we will use the term *source code quality* as synonym of *source code maintainability* throughout the dissertation). It is generally defined as the effort (i.e. cost) required to perform specific modifications in a software package. The importance of maintainability lies in its very obvious and direct connection with the costs of altering the behaviour of the software package.

Aggregating a measure for maintainability has been a challenge in software engineering for a long time. Although the above standard offers a definition for maintainability, it does not provide a universal way of quantifying it. The difficulties arise from the absence of formal definitions and the subjectiveness of the term itself. If a number of different people familiar with the same software package were asked to rank the maintainability of their system on a ten-point scale, there would probably be no consensus among the votes. This suggests that there is probably no commonly acceptable way for expressing source code maintainability in terms of a single value. In this thesis we will present a method for deriving a measure for source code maintainability that in some ways differs from earlier approaches and overcomes many of the above difficulties.

Entropy is a notion that is becoming more commonly used outside physics. In thermodynamics, entropy is used to measure the disorder of a system. In the case of software systems, source code entropy can be defined as the measure of disorder of the source code; i.e. the level of organisation of information in it. Although maintainability and entropy are, by definition, quite different, there is nevertheless a strong connection between them. Assuming that the higher the disorder, the more effort is needed to perform the necessary modifications, entropy can be interpreted as an approximation of maintainability of the source code. According to the second law of thermodynamics, the entropy of a closed system cannot be reduced; it can only remain unchanged or increase. The only way to decrease entropy (disorder) of a system is to apply an external force; i.e. put energy in by making order.

Applying this law of thermodynamics analogy to software systems, we are led to make some interesting observations:

- The maintainability of the source code of an evolving software system does not improve; it can only remain unchanged or decrease.
- The only way to increase the maintainability of the source code is to apply external effort aiming code improvement (e.g. performing code refactorings).

Meir M. Lehman and László Bélády formulated a series of laws of software evolution based on observations concerning the evolution of IBM's systems [70]. The laws were not presented as laws of nature, but rather as general observations that are expected to hold for real-world software systems. The observations made by Lehman and Bélády seem to accord with the above general findings for thermodynamics. Among others, the laws state that

- *Increasing complexity* - as a real-world system evolves its complexity increases unless work is done to maintain or reduce it.
- *Declining quality* - the quality of real-world systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

The continuously decreasing maintainability in the case of software systems has been noticed by many academic and industrial experts. This phenomenon is known as *software erosion*.

2.2 The Columbus framework

Columbus [35] technology was developed in cooperation between the University of Szeged, FrontEndART Software Ltd. and the Software Technology Laboratory of Nokia Research Center. Columbus is able to statically analyse large C/C++, C# and Java projects and to extract facts from them. The main motivation for developing the Columbus system was to create a general framework for combining a number of reverse engineering tasks and to provide a common interface for them. Thus, Columbus is a framework tool which supports project handling, data extraction, data representation, data storage, filtering and visualisation. All these basic tasks of the reverse engineering process for the specific needs are accomplished by using the appropriate modules (plug-ins) of the system. Some of these plug-ins are provided as basic parts of Columbus, but the system can be extended to meet other reverse engineering requirements as well.

The incorporated plug-ins allow the extraction of over 50 source code metrics, bad smells, design patterns, coding rule violations and code duplications. We utilised this framework throughout this study to get facts from source code that we needed to realise our research objectives.

2.3 Source code metrics

Object-oriented metrics measure different properties of a program and express the result in numerical form, so that the values for different programs or parts of programs (e.g. classes) can be compared and conclusions can be drawn from them. Based on the aspects they express, these metrics can be classified into five groups; namely, size, inheritance, complexity, coupling and cohesion. Some of metrics are defined for different types of source code elements, while others are specific to classes, methods or scopes. In this study we will concentrate on the following source code metrics:

2.3.1 Size metrics

- LOC (Lines Of Code) - the gross amount of lines of code in a particular source code element (empty lines, comments included)
- LLOC (Logical Lines Of Code) - the net amount of lines of code in a particular source code element (empty lines, comments excluded)
- TLOC (Total number of Lines Of Code) - the overall number of lines of code in a package, namespace or system counted recursively
- TLLOC (Total Logical number of Lines Of Code) - the overall number of logical lines of code in a package, namespace or system counted recursively
- TNCL (Total Number of CLasses) - the overall number of classes in a package, namespace or system counted recursively
- TNM (Total Number of Methods) - the overall number of methods in a package, namespace or system counted recursively
- TNA (Total Number of Attributes) - the overall number of attributes in a package, namespace or system counted recursively

- NOS (Number Of Statements) - the number of statements in a method or function body
- NLMA (Number of Local Methods Accessed) - the cardinality of the set of method invocations of the method, where the invoked methods belong to the same class as the method itself
- NFMA (Number of Foreign Methods Accessed) - the cardinality of the set of method invocations of the method, where the invoked methods belong to other classes than the method itself
- NPAR (Number of PARameters) - the number of parameters of a method or function

2.3.2 Inheritance metrics

- DIT (Depth of Inheritance Tree) - the length of the longest path from the class to one of its root in the inheritance hierarchy
- NOC (Number Of Children) - the number of classes that are directly inherited from a given class

2.3.3 Complexity metrics

- McCC (McCabe Cyclomatic Complexity) - the number of decisions within the specified method or function plus 1
- WMC (Weighted Methods per Class) - the sum of McCabe cyclomatic complexities in a class
- RFC (Response For a Class) - the cardinality of the set of methods of a class and the set of methods directly invoked by these methods
- NL (Nesting Level) - the maximum depth of the control structure

2.3.4 Coupling metrics

- NOI (Number of Outgoing Invocations) - the cardinality of the set of all function and method invocations in the particular source code element
- NII (Number of Incoming Invocations) - the cardinality of the set of all functions and methods which invoke the particular method or function
- CBO (Coupling Between Object classes) - A class is coupled to another if the class uses any method or attribute of the other class or is directly inherited from it; CBO is the number of coupled classes

2.3.5 Cohesion metrics

- LCOM (Lack of Cohesion On Methods) - LCOM is the number of pairs of methods in the class that use no common attribute

2.4 Code duplications

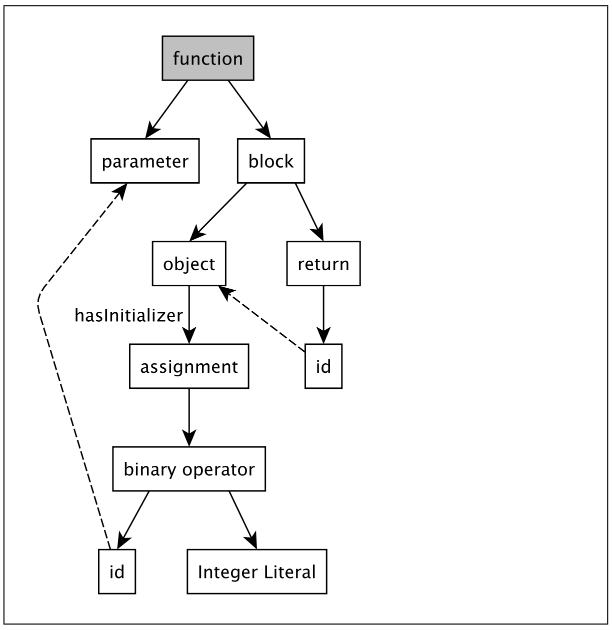
Code cloning (or copy-paste programming) means the copying of an existing piece of code, and after performing smaller modifications on it, pasting it somewhere else. Based on the level of similarity between the copied code fragments we can define the following duplication types:

- Type-1 - the copied code parts are identical code fragments except for variations in whitespace, layout and comments
- Type-2 - syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments
- Type-3 - copied fragments with further modifications such as altered, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments
- Type-4 - the copied code fragments performe the same function

While Type-1 and Type-2 clones are relatively easy to determine, there still do not exist efficient and well-grounded approaches for determining Type-3 and Type-4 clones. In the case of Type-1 and Type-2 clones, cloning defines an equivalence relation on the set of copied code segments (in the case of Type-3 and Type-4 clones, transitivity does not necessarily hold). Two code segments correspond to each other if they are copies of each other (according to the underlying clone detection approach). We will apply the notion of *clone classes* to the classes of the relation, and the members of the classes will be referred to as *clone instances*. Owing to the nature of the relation, each clone class must contain at least two clone instances.

2.4.1 Clone detection

There exist an abundance of clone detection algorithms ranging from lexical (token-based) [34, 48, 54], through AST-based [15, 61, 97] to metric-based [60, 65, 66, 72] approaches. These methods act on one particular version of the software and then a detailed list of copied code segments is provided that may eventually contain several thousand items in the case of a real-size software package. The Columbus framework uses a variant of one of the existing AST-based approaches presented by Koschke et al [61]. This method finds clone candidates that form syntactic units (e.g. classes, functions, blocks, iterations) with linear time and space complexity. The similarity of the clone instances for this approach is defined by the serialisation method of the AST. Two code parts are considered similar (and therefore fall into the same clone class) if they consist of the same AST node types (represented by the schema) in the same order. Figure 2.1 shows an example of a source code fragment and its AST counterpart. The AST contains every important detail about the source code so that a semantically equivalent source code can be generated from it. For simplicity reasons, just the AST node types are shown in the figure.



```
int foo (int i) {
    int j=i+1;
    return j;
}
```

Figure 2.1: An example of a source code fragment and its AST representation

The clone detection algorithm used by Columbus basically extracts all the identical AST structures from the source code, and returns a list of clone classes and clone instances with their precise location in the source code. It follows that every clone class can be uniquely represented by an AST fragment like the one in Figure 2.1. The root node of an AST fragment corresponding to a clone class is called the *head* of the clone class. In the example, the head of the code fragment is a function node (shown in grey).

2.4.2 Code duplication metrics

Clone metrics quantify some of the characteristics of the underlying system from the viewpoint of code duplication. The most widely used clone metrics, are the following:

- CC (Clone Coverage) - the real value between 0 and 1 expresses the degree to which the item is covered by code duplication
- CCL (Clone CLass) - number of clone classes having at least one element within the particular source code element
- CI (Clone Instances) - number of clone instances within the particular source code element

All three metrics can be defined at the system level and for namespaces, packages, classes, methods and functions.

"Measuring programming progress by lines of code is like measuring aircraft building progress by weight."
Bill Gates

Chapter 3

Modelling source code maintainability

According to the ISO/IEC 9126 standard, the high-level characteristics may be affected by low-level quality features that are

- *internal* - measured by looking inside the product, e.g. by analyzing the source code

or

- *external* - measured by execution of the product, e.g. by testing the product.

Figure 3.1 shows the different levels of characteristics, attributes and their relationships as defined by the above-mentioned standard.

Although the quality of source code without doubt affects maintainability, the standard does not provide a consensual set of source code measures as internal quality features. The standard also does not specify the way the aggregation of quality attributes should be performed. These are really not deficiencies of the standard; rather, it offers a certain degree of flexibility for adapting the model to specific needs.

Most of the existing studies share some common basic principles:

- With each given source code metric, its distribution over the source code elements is taken and a value (e.g. average) or a category (based on threshold values) is used as a representative element.
- The value or category is aggregated "upwards" in the model by using a simple weighting or linear combination of weights.
- Higher quality characteristics are also represented by one number or a category.

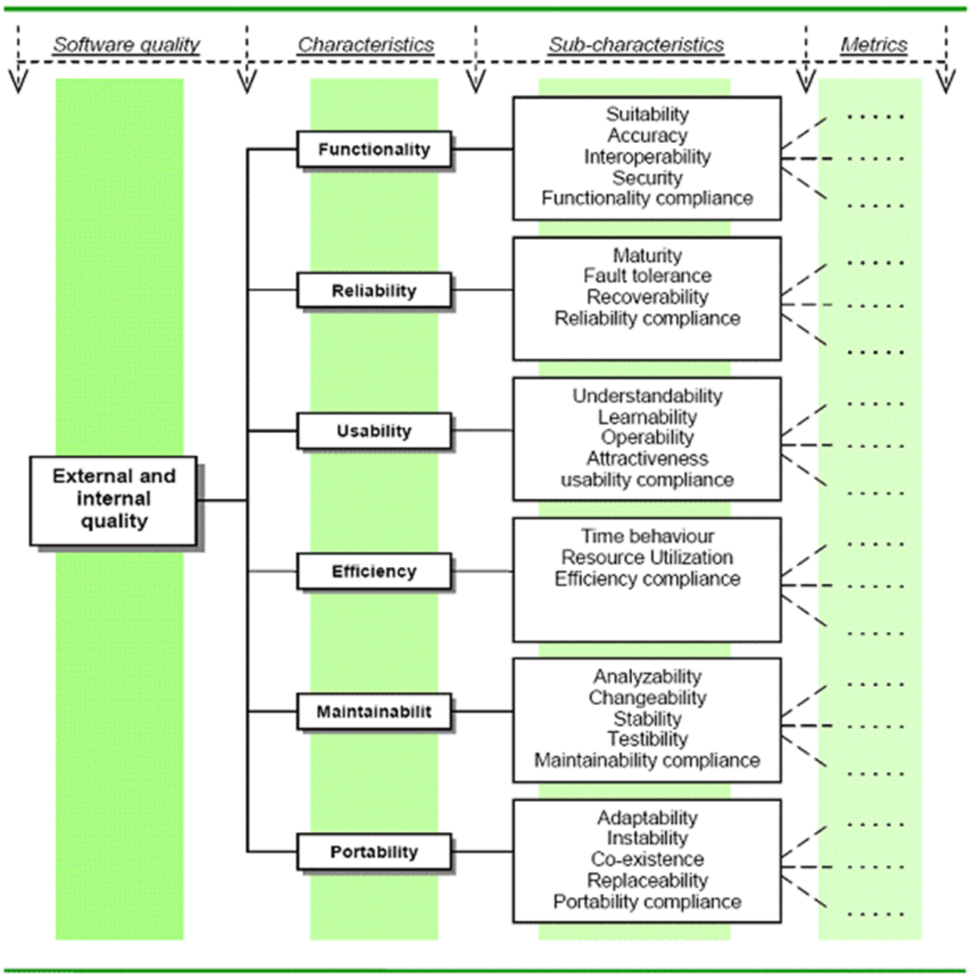


Figure 3.1: Characteristics and attributes defined by the ISO/IEC 9126 standard

We would also like to exploit the freedom of the standard and to propose a novel approach which is quite different from the existing ones. Next, we will show that, with the help of an appropriate modelling technique, the subjective notions of high-level maintainability characteristics (and hence maintainability as well) can be modelled efficiently by using low-level source code metrics. Later we will see that these kinds of metric-based models may not be portable; i.e. models trained on one system may not be readily used on other systems. We will overcome this difficulty by proposing a probabilistic approach for modelling source code maintainability at the end of this chapter.

3.1 Source code metrics and maintainability

The vast majority of existing source code quality models use source code metrics to measure low-level quality attributes. Most of the related studies tackle the correlation between source code metrics with objective measures like failure rates during operation or bug numbers reported in an issue tracking system. For example, Olague et al. [87] studied the ability to predict fault-proneness by using the CK [26], QMOOD [13] and MOOD [41] metric suites. Basili et al.[14] and Gyimóthy et al. [38] calculated code metrics and used regression and machine learning techniques to predict fault-proneness. Provided that it does not involve a great amount of manual effort, the reliability of the results strongly depends on the reliability of the data collected during the operation or recorded in the issue tracking system.

In our study we focused on the relationship between the low-level source code metrics and the high-level maintainability characteristics defined by the ISO/IEC 9126 standard. Unlike the above-mentioned approaches, we invested a large amount of manual effort in gathering reliable information on high-level quality attributes of the systems' source code. The study involved 35 IT professionals and the manual evaluation results of 570 class methods of an industrial and an open source Java system. Our aim was to show that, with an appropriate model, source code maintainability characteristics could be effectively expressed by using low-level source code metrics.

Several statistical models have been constructed to evaluate the relation between low-level source code metrics and high-level subjective opinions of IT experts.

3.1.1 Experiment setup

In order to analyse the relationship between the source code metrics and the high-level maintainability attributes, we performed a time-consuming manual evaluation task. The purpose of the evaluation was to collect subjective ranks for different quality attributes for a large set of methods.

One of the evaluated systems was JEdit, a well-known text editor designed for programmers. The system contains over 700 methods (over 20,000 lines of code), from which we selected 320 to evaluate. The chief criteria for the selection was the length of methods, e.g. we skipped the getter/setter methods and the generated ones. The other system we evaluated was an industrial software product that contained over 20,000 methods and over 200,000 lines of code. From this big set of methods, we selected 250 to evaluate.

The method level source code metrics that we considered were the following:

- Number of Outgoing Invocations (*NOI*)
- Lines Of Code (*LOC*)
- Logical Lines Of Code (*LLOC*)
- Number Of Statements (*NOS*)
- Number of Local Methods Accessed (*NLMA*)

- Nesting Level (*NL*)
- Number of Foreign Methods Accessed (*NFMA*)
- Number of Incoming Invocations (*NII*)
- Number of Parameters (*NPAR*)
- McCabe Cyclomatic Complexity (*McCC*)
- Clone Coverage (*CC*)
- Number of PMD warnings[1] in a method (*PMD*)

The survey was performed by 35 experts, who varied in age and programming experience. The experts were asked to assign a rank (bad, average, good) to each class method for each maintainability characteristic. To ease the evaluation process, a Web-based framework was developed to collect, store, and organise the evaluation results.

3.1.2 Experiment results

During the survey all of above-mentioned 570 methods were evaluated one by one and the results were stored in a database. Weka Experimenter [39] was used to analyse the relationship between source code metrics and high-level maintainability attributes. It is a set of machine learning algorithms for data mining tasks. It contains tools for data pre-processing, classification, regression, clustering, association rules, and visualisation.

Table 3.1 tells us that each source code metric separately has no statistically significant correlation with any of the maintainability features. We note, however, that almost all of the Pearson's correlation values are negative, so that higher metric values generally mean worse maintainability characteristics. This accords with our intuitive expectations.

	NOI	LOC	LLOC	NOS	NLMA	NL	NFMA	NII	NPAR	McCC	CC	PMD
Analysab.	-0.38	-0.41	-0.38	-0.34	-0.23	-0.16	-0.35	-0.03	-0.05	-0.27	0.12	-0.22
Changeab.	-0.35	-0.41	-0.38	-0.35	-0.20	-0.17	-0.33	-0.02	-0.10	-0.29	0.09	-0.21
Stability	-0.28	-0.35	-0.34	-0.31	-0.19	-0.13	-0.24	0.00	-0.06	-0.26	0.07	-0.22
Testab.	-0.25	-0.38	-0.37	-0.34	-0.16	-0.34	-0.22	0.01	-0.07	-0.29	-0.02	-0.24
Comprehen.	-0.34	-0.38	-0.36	-0.33	-0.22	-0.15	-0.30	0.02	-0.10	-0.26	0.09	-0.21

Table 3.1: Pearson's correlation between the code metrics and maintainability characteristics

To reduce the number of dimensions of the problem, we performed a Principal Component Analysis (PCA) [49] first, with parameters intented to cover 97% of the variance. Next, we tested the well-known base classifiers; namely, logistic regression, J48 decision tree and neural network. We used the *ZeroR* (default classifier) algorithm as a baseline to measure the effectiveness of our experimental results. This is the simplest classifier

	ZeroR	J48 decision tree	Log. regression	Neural network
Analysability	67.93%	73.68%	70.97%	70.25%
Changeability	66.79%	76.65%	73.00%	74.26%
Stability	70.20%	73.12%	70.55%	70.92%
Testability	66.55%	64.72%	69.45%	70.54%
Comprehension	70.92%	76.68%	70.93%	73.99%

Table 3.2: Rate of the correctly classified instances

that chooses the class which has the most elements in the training data set. Table 3.2 lists the rate of correctly classified methods for each maintainability feature.

We found that in four out of five cases the best classifier was the J48 (Weka implementation of C4.5) decision tree algorithm. However, it performed very poorly in the classification of *Testability*, which was attributed to the vague definition of the subcharacteristic. The IT experts involved in the survey varied in their range of testing skills and they sometimes interpreted the concept differently. In the case of *Changeability*, the precision value of the J48 classifier was by 10% higher than that of ZeroR. In this case the logistic regression and the neural network algorithms also performed well. Precision is a good way of measuring efficiency, but if we examine the precision and recall values separately for classes it appears that the J48 algorithm is much more useful than ZeroR.

Table 3.3 provides detailed statistics about the precision and recall values of the ZeroR and J48 algorithms in the case of *Changeability*. The precision value of the J48 algorithm is 17 % higher for the *Good* class than that for ZeroR. Moreover, it classified 64% of the *Average* and 23.8 % of the *Poor* instances correctly, while ZeroR missed them completely.

	J48 decision tree				ZeroR			
Class	TP Rate	FP Rate	Precision	Recall	TP Rate	FP Rate	Precision	Recall
Bad	0.238	0.011	0.455	0.238	0	0	0	0
Average	0.640	0.160	0.624	0.640	0	0	0	0
Good	0.852	0.330	0.839	0.852	1	1	0.668	1

Table 3.3: Statistics for classes in the case of Changeability

3.1.3 Conclusions

The empirical results presented in this section show that, with the help of a suitable modelling technique, the subjective aspects of high-level maintainability characteristics (and hence maintainability as well) can be modelled efficiently by using low-level source code metrics. Knowing this, we will present a negative result in the next section concerning the portability of maintainability models like these. Afterwards, in the last section of the chapter, we propose a novel approach for building metric-based maintainability models which, among other things, overcomes the portability issue as well.

[1]http://pmd.sourceforge.net

3.2 Model portability issues

Now we will address the question of which metrics are suitable for constructing portable models (which can be efficiently applied to unknown software systems). It seems quite obvious that a model can only be portable if the participating metrics behave in a similar way on the underlying software systems. From this starting point we analysed the metric values of four real-world software packages in order to see how the metrics behave on systems developed under different circumstances (e.g. open- and closed source) and for different domains (e.g. office and telecommunication applications). Here, we made use of the following well-known metrics (first presented by Chidamber and Kemerer [26]):

- **WMC** - Weighted Methods per Class
- **DIT** - Depth of Inheritance Tree
- **RFC** - Response For a Class
- **NOC** - Number Of Children
- **CBO** - Coupling Between Object classes
- **LCOM** - Lack of Cohesion On Methods
- **LCOMN** - Lack of Cohesion of Methods allowing Negative value
- **LOC** - Lines Of Code

In our statistical analysis, we employed the method of *linear discriminant analysis* (LDA) [93].

3.2.1 Experiment setup

Table 3.4 lists some of the size metrics for the underlying systems. Two of them are open source (Mozilla and OpenOffice), while the other two are closed source real-world software systems.

Software Systems	TNCL	TLOC	TNM	TNA
Mozilla 1.6	4,895	1,513,768	56,382	27,267
Nokia System	18,476	11,629,348	215,306	43,653
OpenOffice 2.0.1	27,900	5,913,418	282,167	131,457
Columbus 3.6	671	244,509	13,346	12,641

Table 3.4: System level metrics for the systems analysed

Our basic premise is that metric-based models should behave in a similar way on statistically similar systems. Nagappan et al. [84] arrived at much the same conclusion after they attempted to apply a bug-model trained on one system to another system. In order to compare systems in the statistical sense, we need to collect a large amount of samples. As we are examining object-oriented metrics, the samples in our case are the classes of the systems. The bigger the systems are, the more classes are available, so the statistical methods should give more accurate results.

We consider systems to be similar if their classes cannot be distinguished easily from each other using their metric values. The similarity of different systems can be measured with the precision values of classifiers (from a given set of statistical or machine learning algorithms) which predict which system a class belongs to based on its metrical values. If the accuracy value of any particular classifier is high, then the two systems are *easily distinguishable*. But, if there is no classifier from the given set of algorithms that efficiently distinguishes the classes of the systems, these systems will be considered *similar*.

Put simply, our hypothesis is that if a model performs well on a system, it can be readily applied to another similar system in the above-mentioned sense. Furthermore, if the systems are easily distinguishable (the metrics behave differently), then the differences prevent a model from being transferred between the systems. Here, our idea for preparing portable models is simple; namely, it should consist of only those metrics for which the systems appear similar to each other.

Next, we applied the *linear discriminant analysis* (LDA) method [93], which transforms the samples in such a way that the most significant differences between them could be seen. After the transformation was applied, the most and the least significant metrics in which the two systems differ from each other can be selected.

Having eight metric values, each class of a system can be regarded as a point in an $\Re^8$ vector space. When comparing two systems, all their classes are represented in the same space. By applying the LDA algorithm, the distance between the classes of the two systems increases and the deviation among the classes of each system decreases at the same time. This way, the classes of the systems are separated from each other. Being a linear transformation, the LDA algorithm returns a vector which represents a line in $\Re^8$ space along which the above-mentioned properties hold for the projections of the original source data; i.e. the distance between the mean values of the projected classes of the systems is maximised and the deviation among the projected classes of each system is minimised. If the two systems differ much there will be two separated sets of points in the image space of the transformation. But if the two systems are similar to each other, the transformation will not be able to separate them too well.

The vector returned by the LDA algorithm, which represents the line along which the separation is the best, contains the coefficients for each metric. A higher absolute value of a coefficient denotes a better separation property for the corresponding metric, while a low absolute value means that the metric it belongs to cannot readily distinguish the classes of the systems.

3.2.2 Experiment results

Figure 3.2 shows an example of the LDA algorithm applied to the classes of OpenOffice (black) and the Nokia System (grey). In the first case only a *principal component analysis* (PCA) [49] was applied to the sample in order to visualise it in 2 dimensions with a minimal loss of information. In the second case a random noise of $N(0,1)$ was added to the sample in the direction of the x axis, for visualisation purposes as well (the transformed points would have ended up along the vertical axis otherwise). It can be seen in the figure that the classes of the two systems could be separated quite well.

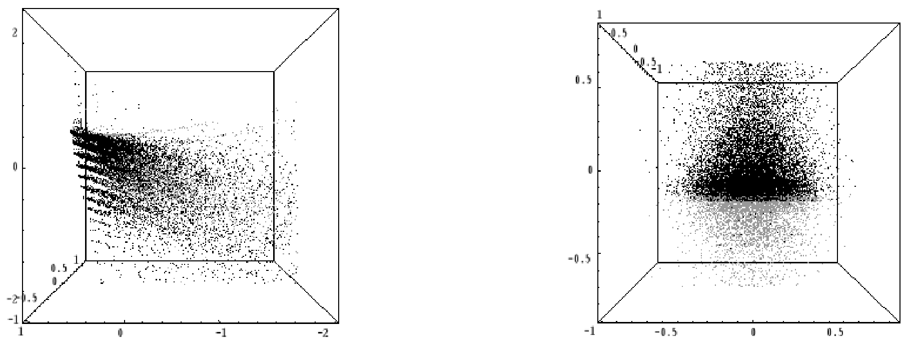


Figure 3.2: An example of LDA transformation

We used the LDA algorithm to measure the similarities between the systems in the way described in Section 3.2.1. We analysed two systems at a time and randomly divided the union of the classes of the two systems into two parts. We used the first part, which contained 90% of the classes, to calculate the LDA coefficients of the transformation. Next, we applied this transformation to these classes, and for the remaining 10% of the classes as well. With the transformed classes from the smaller part, we counted how many of them were close to its own transformed system. By doing this we got a precision value for this particular machine learning algorithm that was supposed to be a lower estimate for the similarity of the two examined systems. We repeated this procedure ten times by taking a different test suite each time. The average accuracy value (i.e. mean precision value) of this validation can be viewed as a measure for similarity between the systems.

By changing the underlying notion of the distance (between a system and a class of a system), we can get different variants of the above-mentioned classifier algorithm. The most common distance measure is the Euclidean distance measure, which in our case is the classical sum of squares of the coordinate-wise differences between the mean of classes of the system and the class to be classified. Another commonly used distance in machine learning is the Mahalanobis-distance [75], which takes into account the correlations between the metrical values of the classes of the systems as well. The greater the deviation in one direction, the smaller the distances will be in the same direction. Taking an example from Figure 3.2, if the point which denotes the transformed class falls into the upper part of the second diagram, the classifier should classify it as an OpenOffice class; otherwise if it falls into the lower region, it should be classified as a Nokia System class. The position and shape of the boundary between the two systems depends on the distance used for classification.

We applied the LDA algorithm for each pair of the systems examined. But before doing so, we standardised the input sample, which means that the average of each metric became zero and the deviations became one. We could have applied this transformation to all the systems at once in the same vector space, but in that case we would have had to expect the presence of the unnecessary influence of one system on every other. Besides this, when a model is intended to be transferred, only two systems are considered at once: the source (the system on which the model was trained on), and the destination (the system on which the model is to be applied).

Table 3.5 gives the coefficients which were calculated via the LDA algorithm. According to the results here and our basic hypothesis, a model on Mozilla which is based on the RFC and WMC metrics should not perform well on the other three systems because the coefficients of these two metrics are the highest. Furthermore, it seems to be a general tendency that these two metrics have the most significant differences between large-scale systems. In one case, for the Columbus-Nokia System pair, the biggest difference appears for the LCOM metric (-0.74). This is because there are several parser classes generated from grammar description files that increase the LCOM values of Columbus. Actually, the highest LDA coefficients for the LCOM metric appear in the pairs where one of the systems is Columbus. With all the other pairs this coefficient is negligible. Hence we concluded that Columbus is in some sense an exception among these large systems and that the LCOM values of the other systems do not distinguish them. The cohesion metric LCOMN is redundant in the model because of the high correlation with LCOM, which is obvious.

System 1	System 2	WMC	DIT	RFC	NOC	CBO	LCOM	LCOMN	LOC
Mozilla	Columbus	0.10	0.10	**-0.92**	0.01	0.01	-0.29	-0.15	-0.14
OpenOffice	Columbus	-0.06	-0.01	**-0.92**	0.00	0.17	-0.23	-0.21	-0.18
OpenOffice	Mozilla	**-0.69**	-0.05	0.26	-0.01	-0.33	0.05	-0.43	-0.40
OpenOffice	Nokia System	0.39	0.12	**-0.71**	0.01	0.49	0.15	-0.01	0.23
Nokia System	Columbus	-0.17	0.00	**-0.50**	-0.02	-0.15	**-0.74**	0.16	-0.35
Nokia System	Mozilla	**-0.55**	-0.35	**0.51**	-0.02	-0.44	-0.11	-0.18	-0.28

Table 3.5: Coefficients of metrics computed by the LDA algorithm

We also measured the similarities between any two systems using both Euclidean and Mahalanobis distance measures introduced earlier, the results being summarised in Table 3.6. From these results, the most similar systems are the Nokia System and Mozilla, and the most dissimilar are OpenOffice and Columbus. In actual fact, Columbus differs the most from every other system. This might have been expected.

Systems 1	System 2	Euclidean precision	Mahalanobis precision	Max. precision
Mozilla	Columbus	80.98%	51.46%	80.98%
OpenOffice	Columbus	86.77%	73.04%	86.77%
OpenOffice	Mozilla	68.87%	74.65%	74.65%
OpenOffice	Nokia Sys.	60.50%	78.41%	78.41%
Nokia Sys.	Columbus	84.03%	42.95%	84.03%
Nokia Sys.	Mozilla	72.90%	58.26%	72.90%

Table 3.6: Precision of the LDA approach

3.2.3 Conclusions

In this section we showed that the different software systems can be distinguished from each other fairly well based on their metric values. It means that classical metric-based models trained on one system may not be readily usable on other systems. To resolve this and other issues, below we present a novel approach for building metric-based models.

3.3 A probabilistic source code quality model

Existing quality models cannot handle ambiguity issues arising from the subjective interpretations of characteristics, which may depend on experience, knowledge, and even expert intuition. We seek to provide a probabilistic approach for computing high-level quality characteristics, which integrates expert knowledge, and handle ambiguity issues at the same time. We expect a source code quality model to be:

1. *Interpretable* – applying the model should provide information about high-level quality characteristics that is meaningful; i.e. conclusions can be drawn with the help of it.

2. *Explicable* – there should be a way to readily evaluate the root causes; i.e. a simple way to break down information get for high-level characteristics to attributes or even to properties.

3. *Consistent* – the information got for higher level characteristics should not contradict lower level information.

4. *Scalable* – the model should provide valuable information even for large systems in a reasonable time.

5. *Extendible* – there should be an easy way to extend the model with new characteristics and its attributes.

6. *Reproducible* – applying the model on the same system twice should yield similar results.

7. *Comparable* – information got for the quality characteristics of two different systems should be comparable and should correlate with an intuitive meaning of the characteristics.

By applying probabilistic methods, the approach we propose should meet all our requirements and eliminate many of the drawbacks of the current approaches that follow basic principles. First, we defined a probabilistic model for evaluating high-level quality characteristics for software systems by aggregating low-level properties to higher levels. We applied this model to two software systems and showed that the model correlates with expert quality opinions quite well.

3.3.1 An approach for constructing maintainability models

Our approach for computing high-level quality characteristics is based on a directed acyclic graph whose nodes correspond to quality properties that may be internal or external. Internal quality properties characterise the software product from an internal (developer) view and are usually estimated via source code metrics. External quality properties characterise the software product from an external (end user) view and are usually aggregated somehow using internal and other external quality properties. The nodes representing internal quality properties will be called *sensor nodes* as they measure internal quality directly. The other nodes will be called *aggregate nodes* as they get their measures through aggregation.

The edges of the graph represent simple dependencies between an internal and an external or two external properties. Internal properties do not dependent on any other attribute; they measure internal quality directly. Our aim is to evaluate all the chief external quality properties by performing an aggregation along the edges of the graph. Below we will refer to this graph as an *Attribute Dependency Graph (ADG)*.

Let $G = (S \cup A, E)$ stand for the *ADG*, where S, A and E denote the sensor nodes, aggregate nodes and edges, respectively. Next, we wish to measure how good or bad an attribute is. *Goodness* is the term that we use to express this measure of an attribute. For the sake of simplicity, we will write "goodness of a node" instead of "goodness of an attribute represented by a node". Goodness is measured on the $[0, 1]$ interval for each node, where 0 and 1 mean the worst and best, respectively. A straightforward solution would be to have a goodness value for each sensor node and then an approach for how to aggregate it "upwards" in the graph, as many other researchers do. We decided not to follow this path, however. Instead, we assume that the goodness of each sensor node u is not known precisely, hence it is represented by a random variable X_u with a probability density function $g_u : [0, 1] \to \Re$. We call g_u the *goodness function* of node u.

Constructing a goodness function

The currently presented way of constructing goodness functions is specific to source code metrics. For different sensor types, different approaches may be needed. Here, we make use of the metric histogram over the source code elements as it characterises the whole system from the aspect of a single metric. The aim is to provide a measure for the goodness of a histogram. As the notion of goodness is relative, we expect it to be measured by means of a comparison with other histograms. Let us suppose that H_1 and H_2 are the histograms of two systems for the same metric, and $h_1(t)$ and $h_2(t)$ are the corresponding normalised histograms (i.e. density functions). By using the formula

$$\mathcal{D}(h_1, h_2) = \int_{-\infty}^{\infty} (h_1(t) - h_2(t))\, \omega(t)\, dt,$$

we get a distance function (not in the mathematical sense) defined on the set of probability functions. Figure 3.3 helps clarify the meaning of the formula: it computes the signed area between the two functions weighted by the function $\omega(t)$.

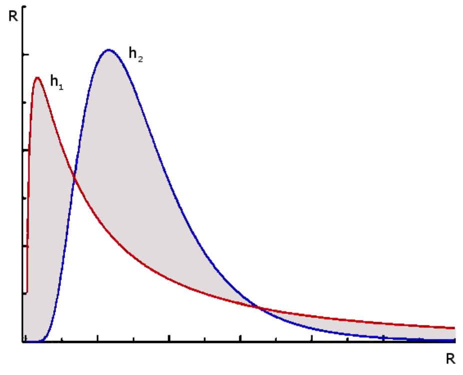


Figure 3.3: Comparison of probability density functions

The weight-function plays a crucial role. It determines the notion of goodness; i.e. where on the horizontal axis the differences matter more. If one wishes to express the fact that all the metric values matter to the same degree, we should set $\omega(t) = c$, where c is a constant; and in that case $\mathcal{D}(h_1, h_2)$ would be zero (as h_1 and h_2 integrate to 1). However, if one would like to say that higher metrical values are worse, one could set $\omega(t) = t$. Non-linear functions for $\omega(t)$ are also possible. As in the case of most source code metrics, higher values are considered to be worse (e.g. McCabe's complexity), we will use the $\omega(t) = t$ weight function for these metrics (linearity is implicity assumed here). This choice leads us to the very simple formula

$$\mathcal{D}(h_1, h_2) = \int_{-\infty}^{\infty} (h_1(t) - h_2(t))\, t dt = \int_{-\infty}^{\infty} h_1(t)\, t - \int_{-\infty}^{\infty} h_2(t)\, t = E\left(H_1'\right) - E\left(H_2'\right) \approx \tilde{H}_1 - \tilde{H}_2,$$

where H_1' and H_2' are the random variables corresponding to the h_1 and h_2 density functions, $E\left(H_1'\right)$ and $E\left(H_2'\right)$ are the expected values of these random variables (the equality is based on the definition of the expected value of a random variable). Then, $\tilde{H}_1$ and $\tilde{H}_2$ are the averages of the histograms H_1 and H_2, respectively. The last approximation is based on the Law of Large Numbers (the averages of a sample of a random variable tend to the expected value of the same variable). With this comparison we get one goodness value for the subject histogram, this value being relative to the other histogram.

In order to get a proper goodness function, we repeat this comparison with histograms of many different systems independently. In each case we get a goodness value which can basically be regarded as a sample of a random variable lying in the range $[-\infty, \infty]$. The linear transformation $x \ \rightarrow x/2 \cdot max\left(\mid max \mid, \mid min \mid\right) + 0.5$ changes the range to the $[0, 1]$ interval. The transformed sample is treated as a sample of the random variable X_u, which is what we wanted at the beginning. An interpolation of the empirical density function yields the goodness function of the sensor node.

We accomplish the section with a theoretical beauty of the approach. First, let us assume that one has histograms of N different systems for each particular metric. Each histogram can be considered to be sampled by different random variables $Y_i, (i = 1, \ldots, N)$. Furthermore, one would like to assess the goodness of another histogram corresponding to the random variable X. The goodness is by definition described by the following series of random variables:

$$Z_1 := E\left(Y_1\right) - E\left(X\right), \ldots, Z_N := E\left(Y_N\right) - E\left(X\right).$$

The random variable for goodness (before the transformation) is then described by the random variable Z:

$$Z := \frac{1}{N}\sum_{i=1}^{N} Z_i \rightarrow \Phi_{\nu,\sigma}, \text{ if } N \rightarrow \infty.$$

According to the Central Limit Theorem for independent (not necessarily identically distributed) random variables, Z tends to a normal distribution that is independent of the benchmark histograms. This is of course a theoretical result, and it means that having a large number of systems in our benchmark, the constructed goodness functions are (almost) independent of the particular systems in the benchmark. Actually, $\Phi_{\nu,\sigma}$ is a benchmark-independent goodness function (on $[-\infty, \infty]$) for X, which can be approximated by having a benchmark with sufficient number of systems.

To be able to construct goodness functions in practice, we decided to build a source code metric repository database, where we uploaded the source code metrics of over 100 open source and industrial software systems.

Aggregation

Now that we are able to construct goodness functions for sensor nodes, we need to find a way to aggregate them along the edges of the ADG. Recall that the edges only represent dependencies as we have not yet assigned any weights to them. Once again, assigning a simple weight would lead to the classic approach, which we chose not to follow. In models that use a single weight or threshold in aggregation, the particular values are usually supported with rational arguments that sometimes lead to debates among experts. We decided to equip our model with the ability to handle this ambiguity issue. We created an online survey where we asked many experts (both industrial and academic) for their opinions about the weights. For each aggregate node, they were asked to assign scalars to incoming edges such that the sum of these would be 1. The value assigned to an edge is viewed as the amount of contribution of source goodness to target goodness. This way, for each aggregate node v a multi-dimensional random variable $\vec{Y_v} = (Y_v^1, Y_v^2, \ldots, Y_v^n)$ exists, where n is the number of incoming edges. The components are dependent random variables, since

$$\sum_{i=1}^{n} Y_v^i = 1,$$

holds; that is, the range of $\vec{Y_v}$ is the standard $(n-1)$-simplex in $\Re^n$. It should be mentioned that one cannot simply decompose $\vec{Y_v}$ into its components because of the existing dependencies among them.

Having an aggregate node with a composed random variable $\vec{Y_v}$ for aggregation ($\vec{f}_{\vec{Y_v}}$ will denote its composed density function), and also having n source nodes along the edges, with goodness functions $g_1, g_2, \ldots g_n$, we define the aggregated goodness for the aggregated node in the following way:

$$g_v(t) = \int_{\substack{t=\vec{q}\vec{r} \\ \vec{q}=(q_1,\ldots,q_n)\,\in\,\Delta^{n-1} \\ \vec{r}=(r_1,\ldots,r_n)\,\in\,C^n}} \vec{f}_{\vec{Y_v}}(\vec{q})\, g_1(r_1)\ldots g_n(r_n)\, d\vec{r} d\vec{q},$$

where Δ^{n-1} is the $(n-1)$-standard simplex in $\Re^n$ and C^n is the standard unit n-cube in $\Re^n$. Although the formula may look horrendous at the first glance, it is just a generalisation of how aggregation is performed in classic approaches. Classically, a linear combination of goodness values and weights is taken, and it is assigned to the target node. When dealing with probabilities, one needs to take every possible combination of goodness values and weights – and also the probabilities of their outcome – into account. In the formula, the components of the vector $\vec{r}$ traverse the domains of source goodness functions independently, while vector $\vec{q}$ traverses the simplex, where each point represents a probable vote for the weights. For fixed $\vec{r}$ and $\vec{q}$ vectors their scalar product ($t = \vec{q}\vec{r} = \sum_{i=1}^{n} r_i q_i \in [0,1]$) is the goodness of the target node. To compute the probability for this particular goodness value, one has to multiply the probabilities of goodness values of source nodes (these are independent) and also the composed probability of the vote ($\vec{f}_{\vec{Y}_v}(\vec{q})$). This product is integrated over all the possible $\vec{r}$ and $\vec{q}$ vectors (please note that t is not uniquely decomposed to vectors $\vec{r}$ and $\vec{q}$). $g_v(t)$ is indeed a probability distribution function on the $[0,1]$ interval; i.e. its integral is equal to 1, because both $\vec{f}_{\vec{Y}_v}(\vec{q})$ and the goodness functions integrate to 1 on Δ^{n-1} and C^n, respectively.

With this method we are now able to compute goodness functions for each aggregate node. The way the aggregation is performed is mathematically correct, meaning that the goodness functions of aggregate nodes actually express the probability values of their goodness (by combining other goodness functions with weight probability values).

We should now make two important remarks concerning the aggregation process:

1. It may not be obvious, but the dependencies between the dependent (connected) nodes are implicitly assumed to be linear.
2. It might be imagined that there is a serious drawback in the method. Namely, the curse of dimensionality. According to the aggregation formula, the integration is performed on a closed, convex subset of the $\Re^{2n-1}$ vector space (n is the number of incoming edges). Even for small n values this computation can be expensive and imprecise. To avoid an exponential increase in the computational costs, we apply the *Monte Carlo method* with random sample points generated that are equally distributed both on Δ^{n-1} and C^n. This obviously does not help improve precision: the votes will simply be too scarce in higher dimensions. Empirically we, found that the approach works sufficiently well if the number of incoming edges is not higher than three for each aggregate node.

Although this approach provides goodness functions for each aggregate node, managers are usually only interested in having a single numeric value that represents an external quality attribute of the software system. Goodness functions carry much more information than this, but an average value for the function may satisfy even the managers.

The resulting goodness function at each node has a simple meaning: it is the probability distribution that describes how good a system is from the aspect represented by the node. Therefore, the approach leads to *interpretable* results. Provided that the goodness functions are computed for each node, and that the dependencies in the *ADG* are known, it is not hard to locate the root causes. Even formally, it would not be difficult to rank the low-level properties according to their impact on the high-level ones. It means that the results are *explicable*. *Consistency* trivially follows from the way of the aggregation (from the monotonity of integration). The approach scales well for large systems, as the memory and time consumption requirements do not depend on the size of the system to be evaluated (just the distributions are used). An *ADG* can be readily extended with new nodes at any level. In fact, adding a new sensor node with all its dependencies is a straightforward generalisation because setting the goodness function of the node to $g(u) = 1$ (on the $[0, 1]$ interval) will yield the original model irrespective of the votes on the edges. *Comparability* follows from the definition of goodness (i.e. the weight function $\omega(t)$) and from the consistency of aggregation. It also seems to follow from our earlier remark concerning the theoretical existence of benchmark independent goodness functions. Based on these observations, the approach bears with all the properties necessitated earlier.

3.3.2 An instance of a maintainability model for Java

The ISO/IEC 9126 standard defines six high-level product quality characteristics, which are functionality, reliability, usability, efficiency, maintainability and portability. Now we will just focus on maintainability; i.e. the capability of the software product to be modified, where modifications may include corrections, improvements, or adaptation of the software to changes in environment, and in requirements and functional specifications [44]. The standard defines the following attributes which affect maintainability:

- **Analysability:** the capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified.
- **Changeability:** the capability of the software product to enable a specified modification to be implemented, where implementation includes coding, designing and documenting changes.
- **Stability:** the capability of the software product to avoid unexpected effects from modifications of the software.
- **Testability:** the capability of the software product to enable modified software to be validated.

Our intention is to utilise low-level source code properties in order to compute the above attributes and thus the maintainability characteristic itself. For the low-level properties we used the most commonly applied types of code properties: source code metrics [26], coding rule violations and code duplications (clones) [15]. Our aim was to find a relatively small set of low-level properties that has representatives from every type and has as much an overall expressiveness as possible. With these points in mind, the following low-level properties were selected:

- **TLLOC** – total logical lines of code for the whole system (lines of code not counting comments and empty lines).
- **McCabe** – McCabe cyclomatic complexity [78] defined for the methods of the system.

- **CBO** – coupling between object classes, which is defined for the classes of the system.
- **NII** – number of incoming invocations (method calls), defined for the methods of the system.
- **Impact** – size of the change impact set (computed by the SEA/SEB algorithm [46]), defined for the methods of the system.
- **Error** – number of serious coding rule violations, computed for the methods of the system.
- **Warning** – number of suspicious coding rule violations, computed for the methods of the system.
- **Style** – number of coding style issues, computed for the methods of the system.
- **CC** – clone coverage, the percentage of copied and pasted source code parts, computed for the methods of the system.

The reason why the number of incoming invocations metric was included, but not the number of outgoing invocations, is that the Impact set size metric is really a generalisation of the NOI. The final set of low-level properties was selected through several iterations with the helpful advice of academic and industrial experts in the field. After this set of properties was formally fixed, in several other iterations the low-level properties were linked to higher level properties and to attributes. As the algorithm did not perform well when the number of incoming edges was high, we introduced new artificial (virtual) properties whose role was just to gather some nodes of the dependency graph. It was also presumed that these artificial nodes may also influence each other in some way.

The following artificial high-level properties were added:

- **Code complexity** – this represents the overall complexity (internal and external) of a source code element. This high-level property groups the *McCabe, CBO* and *NII* metrics.
- **Fault proneness** – this represents the possibility of having a faulty code segment. It is aggregated using the *Error* and *Warning* properties.
- **Low-level quality** – this represents the very low-level source code quality expressed in terms of coding errors, warnings and style.
- **Comprehension** – this tells us how easy it is to understand the source code. It is aggregated from the *TLLOC* and *Impact* low level metrics, and from the *Code complexity* high level properties.

- **Effectiveness** – this measures how readily the source code can be changed. The source can be readily changed if it is easy to change it and changes will most likely not have any unexpected side-effects. The high-level property is aggregated to *Changeability* and *Stability* attributes.

After, the edges of the model graph were weighted by developers, project managers, consultants, and testers independently (Table 3.7 shows the distribution of participants who voted).

Experience	Sw.eng.	Manager	Consultant	Tester
<1 year	2	0	0	1
1-3 years	10	2	1	1
3-7 years	3	1	1	2
>7 years	3	0	1	0

Table 3.7: Persons who were involved in the weighted voting survey

The contributors were asked to rank the importance of each edge with a value between 0 (unimportant) and 1 (most important). Provided that the random variables representing the weights on the incoming edges of a node are not independent (and the algorithm employs composed distributions), those surveyed were asked to assign weights that summed to one for the incoming edges of a node. In this way, relative importance is assigned for the incoming edges of a node (relative to the other incoming edges of the same node), and a composed distribution function was obtained that belonged to the node itself. Figure 3.4 shows the eventual model graph got at the end of the iterations.

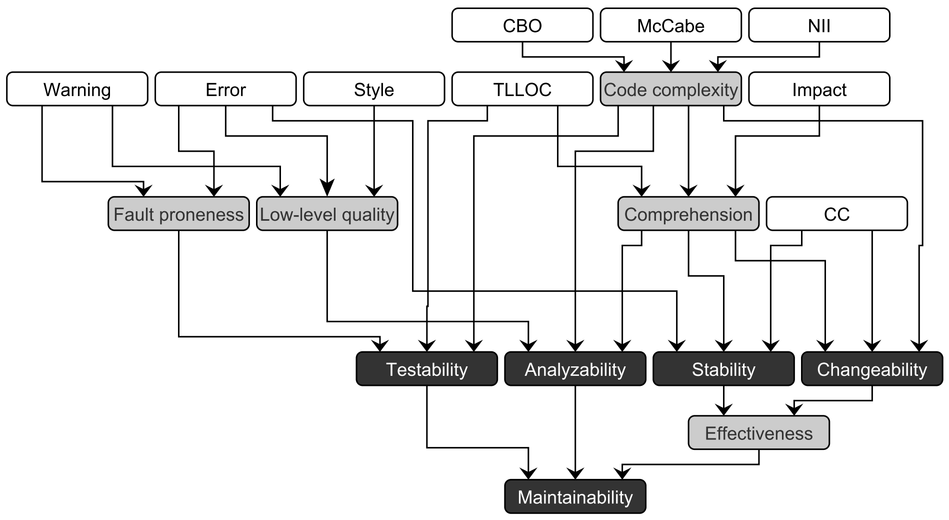


Figure 3.4: The *ADG* describing the relations among low-level properties (white), ISO/IEC 9126 attributes and maintainability characteristic (black) and high-level virtual properties (grey)

A benchmark consisting of 100 open source and industrial software systems implemented in the Java programming language was created. For each system and each sensor node, the distribution of the low-level quality properties over the source code elements of the system (classes and methods) is stored in the database. The size of the projects analysed varied from a couple of hundred to over a million lines of code. The largest system was the *glassfish v2.1* application server with 1,031,741 lines of code, while the smallest one was called *pdfsam* with only 346 lines of code. The average lines of code of the systems in the benchmark was about 122,500. The systems in the benchmark were chosen from 13 different domains, including games, database management systems, office tools, GUI and other frameworks and libraries. The distribution of the projects by their sizes and domains is depicted in Figure 3.5. Table 3.8 summarises some additional properties of the benchmark.

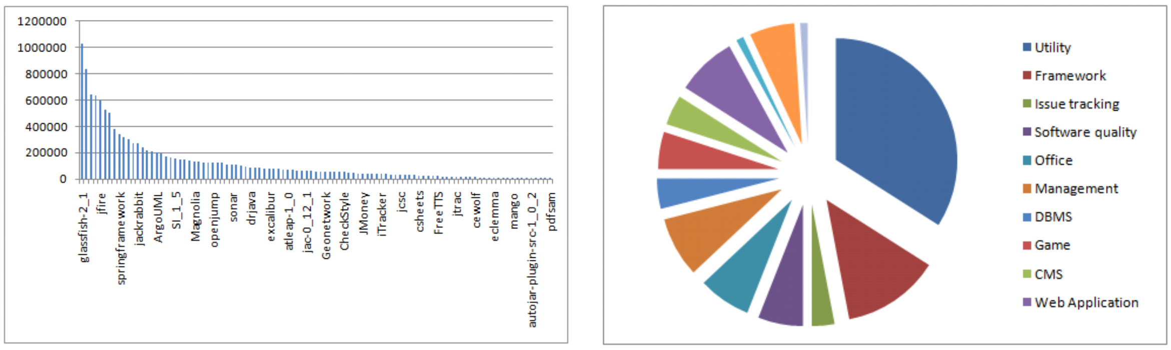


Figure 3.5: The distribution of the project sizes (left) and domains (right) in the benchmark

Metric	Max.	Min.	Avg.
Avg. McCabe	4.33	1.12	2.01
Avg. CBO	11.6	1.14	4.74
Avg. CC	0.8	0.014	0.157
TLLOC	1,031,741	346	122,503

Table 3.8: Statistics for the benchmark

The source code metrics, rule violations and code duplications for the benchmark were computed via the *Columbus* [35] static source code analyzer tools. An implementation of the algorithm was created which accepts the histograms of the system to be evaluated, uses the benchmark database, and returns goodness functions for each sensor and aggregate node.

3.3.3 Validation of the model

The quality model was evaluated on two software systems implemented in the Java programming language. The first one is an industrial system developed over several years by a Hungarian company. Due to a non-disclosure agreement, we have to refer to this system as *System-1*. In order for the results to be reproducible, we also evaluated the model on an open source software package developed at the University of Szeged. The *REM* framework is a persistence engine whose development began from scratch in 2010, and being a greenfield and well-documented project, the different development phases are easy to isolate from the beginning. Our intention was to compare the results of the quality model with the subjective opinions of those involved in its development. For this, we had to choose the kind of systems where the developers were accessible for interviews. In the case of *System-1*, three versions were studied; namely, version 1.3, 1.4 and 1.5 released in 2009, 2010 and 2011, respectively. In the case of *REM*, four versions were studied; namely, version 0.1, 1.0, 1.1 and 1.2. Table 3.9 summarizes some basic statistics of the evaluated systems.

System	Size (TLLOC)	Nr. of pkg.	Nr. of cl.
System-1 v1.3	35,723	24	336
System-1 v1.4	53,406	27	477
System-1 v1.5	48,128	27	454
REM v0.1	6,262	14	82
REM v1.0	7,188	22	83
REM v1.1	5,737	21	66
REM v1.2	8,335	21	94

Table 3.9: Basic properties of the evaluated systems

With the algorithm, first the goodness functions for the sensor nodes are computed. Figure 3.6 shows the computed goodness functions for *REM v1.2* for the sensor nodes. Provided that the goodness functions are probability density functions on the $[0, 1]$ interval, the area of the bars is always equal to one. A system is considered better when the higher bars lie closer to the right hand side. For example, in the case of *McCabe*, *REM* performs better than the other systems in the benchmark, while in the case of *CBO* it is vice versa. In the case of *TLLOC*, *REM* is "better" than the benchmark systems, meaning that its size is smaller than the average value. Although goodness functions express a lot, it is always useful to have a single value as a measure for goodness. The mean value of a goodness function is a good candidate for representing goodness by a single value. From now on, we will refer to the mean of a goodness function as the *goodness value*.

After the goodness functions for the low-level nodes are computed, the aggregation step follows. For each aggregate node a goodness function is computed based on the composed distribution function of votes and the goodness functions of the incoming nodes. Figure 3.7 shows how the goodness values of low-level properties changed with the versions of *System-1*. *System-1 v1.3* had been developed without using any particular quality assurance for the source code itself, and new features had been added without any

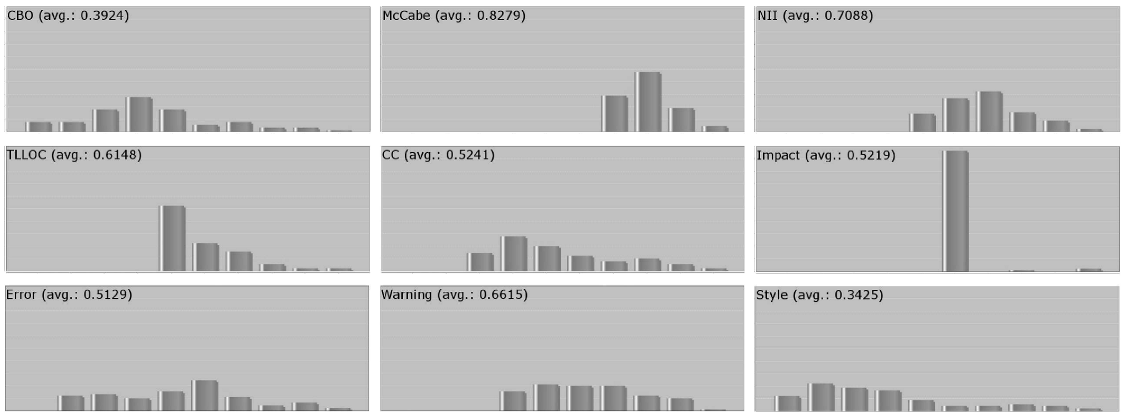


Figure 3.6: Goodness functions for low-level quality attributes (sensors) in the case of the *REM v1.2* system

quality control. As can be seen, the *Error, Warning,* and *Style* attributes are the worst for this version. After this version, static source code analysis was introduced into the development processes of the system, and source code metrics and coding rule violations came into focus. Although a lot of new code was added to the next version (as can be seen from the *TLLOC* attribute), from the aspect of *Error, Warning,* and *Style* attributes there was a significant improvement. The same holds for the *CBO, NII,* and *CC* metrics as well. Only the average function complexities became worse (*McCabe*), for which not much attention was paid. The last version (v1.5) was a result of a pure improvement project: no features were added, just refactorings and bugfixes were carried out. As a result, all the low-level properties improved or at least remained at the same level (even the size of the system became smaller). Figure 3.7 shows the aggregated attributes and maintainability characteristic of *System-1*, which conforms our expectations: version 1.4 is "better" than version 1.3 in the case of changeability, analysability and stability owing to the improvements in the low-level properties. However, in the case of testability, version 1.4 is "worse" because of the increasing *McCabe* complexity. Testability lumps maintainability with itself, which is therefore better for version 1.3. The last version outperforms version 1.4 in every attribute and maintainability characteristic as almost every one of its low-level properties had improved. Version 1.3 is still the best from the aspect of testability, because the complexities of its functions are the lowest on average. It should be added that for all three versions, every maintainability ISO/IEC 9126 attribute is less than 0.5, meaning that *System-1* is "worse" than an average system (based on our earlier remark about the theoretical beauty of the approach).

In the case of *REM*, the development was started as a green-field project, building everything from scratch. Figure 3.8 shows the low-level properties and the ISO/IEC 9126 attributes for *REM*. Being an entirely new development, it is not surprising that the maintainability characteristic and its ISO/IEC 9126 attributes were relatively high. The first version (*v0.1*) was developed by graduate students at the University of Szeged during their summer practice. Afterwards, the codebase was taken over by the depart-

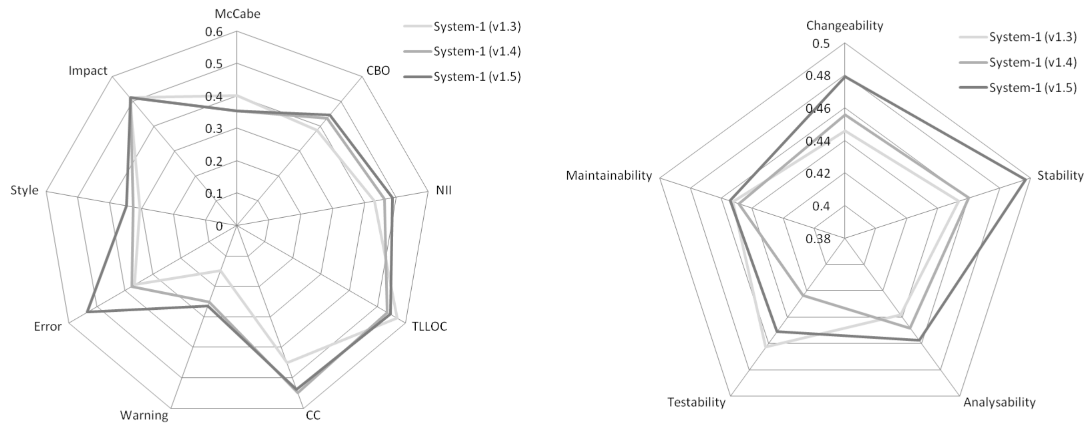


Figure 3.7: Goodness values for the low-level (left) and ISO/IEC 9126 Maintainability characteristic and its subcharacteristics (right) in the case of *System-1*

ment staff that had several years of development experience, and they performed major improvements and refactorings to the code. In Figure 3.8 it can be seen that every attribute except for *Testability* improved. *Testability* did not improve as the *McCabe, NII,* and *Warning* attributes became slightly worse. After this improvement phase, many new features were added, resulting in a significant decrease of most of the low- and high-level attributes, hence the maintainability characteristic fell slightly from 0.7539 to 0.7402. Finally, another improvement phase came that focused on coding rule violation (*Error, Warning*) properties, which resulted in an improvement of the maintainability characteristic (it increased to 0.7482) and its attributes.

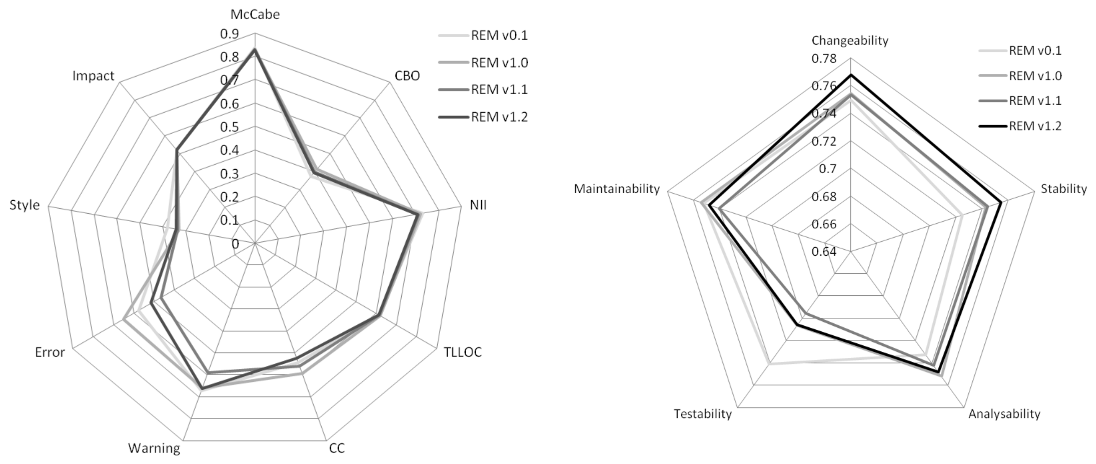


Figure 3.8: Goodness values for the low-level (left) and ISO/IEC 9126 Maintainability characteristic and its subcharacteristics (right) in the case of *REM*

To validate the results, the developers were asked to rank maintainability and its ISO/IEC 9126 attributes of their systems on a 0 to 10 scale, based on the definitions provided by the standard. For *System-1*, six developers answered the questions, four of them had experience of between one and three years, and two of them had experience of over seven years. With REM, five developers completed the survey, three of them had experience of less than one year and two of them had experience of over seven years. Table 3.10 lists the averages of the ranks (divided by ten) for every version of both software systems. The values in the brackets are the goodness values computed by the model.

Version	Changeab.	Stability	Analysab.	Testab.	Maintainab.
REM v0.1	0.625 (0.7494)	0.4 (0.7249)	0.675 (0.7323)	0.825 (0.7409)	0.625 (0.7520)
REM v1.0	0.6 (0.7542)	0.65 (0.7427)	0.75 (0.7517)	0.8 (0.7063)	0.75 (0.7539)
REM v1.1	0.6 (0.7533)	0.66 (0.7445)	0.7 (0.7419)	0.66 (0.6954)	0.633 (0.7402)
REM v1.2	0.65 (0.7677)	0.65 (0.7543)	0.8 (0.7480)	0.775 (0.7059)	0.7 (0.7482)
Correlation	**0.71**	**0.9**	**0.81**	**0.74**	**0.53**
System-1 v1.3	0.48 (0.4458)	0.33 (0.4535)	0.35 (0.4382)	0.43 (0.4627)	0.55 (0.4526)
System-1 v1.4	0.6 (0.4556)	0.55 (0.4602)	0.52 (0.4482)	0.4 (0.4235)	0.533 (0.4484)
System-1 v1.5	0.64 (0.4792)	0.64 (0.4966)	0.56 (0.4578)	0.46 (0.4511)	0.716 (0.4542)
Correlation	**0.87**	**0.81**	**0.94**	**0.61**	**0.77**

Table 3.10: Averaged grades for maintainability and its ISO/IEC 9126 attributes based on the developers' opinions

The results show that the experts' rankings differ significantly from the goodness values provided by the model in many cases. Actually, there are large differences between the opinions of experts, depending on their experience, knowledge and degree of involvement. There were cases when one of the developers who had little experience ranked testability at 9, while another who had over seven years of experience ranked it at 4. Despite the differences between the expert rankings and the goodness values, they surely indicate relatively high correlation with each other, meaning that they change in a similar way. The bold lines in Table 3.10 show the Pearson's correlation of the rankings and the goodness values. The positive (and relatively high) correlations indicate that the quality model partially expresses the same changes as the developers would expect.

3.3.4 Limitations

Both the proposed algorithm and the application have some properties which may affect the practical aspects and the usability of the approach.

- Although the method is sufficiently general, for simplicity reasons we applied it assuming linear $\omega(t) = t$ weight functions. With this choice, we implicitly assume, that there is a linear dependency between the source code metrics and goodness, which may not always be the case.

- As already mentioned, the algorithm suffers from curse of dimensionality problems, which necessitated restrictions on the structure of the *ADG* graph. This is a theoretical obstacle of the algorithm that cannot be overcome just by using heuristics, which inevitably leads to loosing some information during aggregation.
- In Section 3.3.2 the presented *ADG* was created as a result of brainstorming sessions over several iterations. We do not pretend that this graph is either complete or unique. For example, *DIT (Depth of Inheritance Tree)* is an important measure of maintainability, but it was not considered in the model, as the weight function $\omega(t) = t$ would have been unsuitable. Indeed, for small numbers, it should first decrease and then it should only start increasing for larger values. This constraint would have made the model more complex, harder to understand and to evaluate. Different contexts may require different *ADG*s, which always need to be created from scratch.
- The benchmark utilised here contains systems taken from different domains, which may be sufficient for evaluating the general quality of a system, but domain specific benchmarks would be required to get more comparable results.
- The systems used for the evaluation are small and the approach should be validated for larger systems as well.
- The ability of the developers to recall the earlier quality of systems may distort evaluations of older versions of a system.
- For the maintainability quality characteristic and its attributes, only the mean values of goodness functions were compared to averages of the experts' votes. It might have been better to compare the distribution of votes with the goodness functions, but in that particular case it seem unreasonable owing to the relatively small number of votes.

Because of the Central Limit Theorem, the benchmark is not viewed as a threat to validity provided there are enough systems in it. Then, the quality of the systems in the benchmark is not important.

3.3.5 Conclusions

In this section we defined a probabilistic model for evaluating high-level quality characteristics for software systems. Although our algorithm provides a mathematically consistent and clear way for assessing source code quality, it is not specific to source code metrics, and not even to software systems.

The presented model integrates expert knowledge, handles ambiguity issues and manages goodness functions that are continuous generalisations of threshold-based approaches. We found that the changes in the results of the model reflect the development activities, i.e. during development the quality decreases, but during maintenance the quality increases. We also found that although the goodness values computed by the model are different from the rankings provided by the developers, they still show relatively high correlations concerning the tendencies.

3.4 Summary

In this chapter we showed that, with the help of an appropriate modelling technique, the subjective notions of high-level maintainability characteristics (and hence maintainability as well) can be readily modelled by using low-level source code metrics. Unfortunately, the different software systems can be distinguished from each other quite well based on their metric values, which means that classical metric based models trained on one system may not be directly usable on other systems. To overcome these difficulties, we defined a probabilistic model for evaluating high-level quality characteristics for software systems. The model presented integrates expert knowledge, handles ambiguity issues and manages goodness functions that are continuous generalisations of threshold-based approaches. We found that the model rankings show relatively high correlations with the tendencies in expert opinions.

"The bitterness of poor quality remains long after the sweetness of low price is forgotten."

Benjamin Franklin

Chapter 4

Cost and maintainability

Because of the lack of formal definitions and the subjectiveness of the notion of maintainability, there is currently no clear consensus among researchers about its relation to development cost.

Radlinski and Hoffman [91], for example, compared 23 machine learning algorithms using local data as a benchmark for development effort prediction. The four datasets used in the experiments contained both process and product information, but lacked source code metrics (apart from KLOC, which measured the lines of code). The accuracy of these algorithms strongly depended on the set of predictors, hence no universal machine learning algorithm was found by the authors. Instead, they found that running several algorithms using arbitrary predictors could serve as a good starting point for project managers on estimating development costs.

Several comprehensive studies summarise the effectiveness of various cost prediction methodologies.

Riaz et al. [92] examined 710 studies and selected 15 for an in-depth analysis. Their aim was to evaluate the ability of existing methods to measure maintainability. They give a thorough overview of the interpretations of maintainability and summarise the most commonly used techniques for defining it. The authors analysed the measures used by these approaches and also the accuracy of the prediction results. A systematic review of research questions found in the papers was carried out. The answers to these questions were used to grade the effectiveness of the studies based on a fixed set of criteria.

Mair et al. [76] examined 171 papers regarding analogy- and regression-based techniques for cost estimation. They proposed a broad selection of criteria for defining the effectiveness of the different approaches. They found that regression models performed poorly, while analogy-based methods were far better. In many cases, the two different methods provided conflicting results on the same dataset. Case-based reasoning using a benchmark database gave good results in general, but exceptions were also found. Due to the lack of standardization in software quality assurance, no universally applicable method was found by the authors for conducting software cost estimation.

In the next section, we propose a unique formal mathematical model for relating development cost to the maintainability of the source code, which can also be used for predicting future costs of changes. The proposed model turns out to be more efficient than the classical regression-based approaches.

4.1 A cost model based on software maintainability

In our approach, we borrow the concept of entropy used in thermodynamics, which is utilised to measure the disorder of a system. In the case of software systems, maintainability seems to be an appropriate candidate for approximating the level of disorder; i.e. the entropy of the source code.

The proposed model is based on two simple assumptions:

1. When making changes to a software system without explicitly seeking to improve it (like adding new functionalities), its maintainability will decrease (i.e. its disorder will increase), or at the very least it will remain unchanged.
2. Performing changes in a software system with lower maintainability (i.e. higher disorder) is more expensive.

Only these two assumptions were used to derive a system of equations which serve as a model for relating maintainability to development cost. We introduce the notion of *erosion factor*, which is a vital parameter of the model that measures the amount of "harm" caused by changing source code lines of a software. As we will show in Section 4.1.1, the erosion factor may also serve as a measure for process quality. Model parameters can be computed from historical data, such as development costs in the past. After the estimates for the parameters are available, predictions for the future can be obtained from our model.

We evaluated the model on five software systems implemented in the Java programming language. Three of these are commercial closed-source systems. In order to facilitate the reproducibility of the experimental results, we performed an analysis on two open source systems as well.

4.1.1 A formal model for relating costs and maintainability

According to the second law of thermodynamics, the entropy of a closed system cannot be reduced; it can only remain unchanged or increase. The only way to decrease entropy (disorder) of a system is to apply external forces, i.e. to input energy to make order.

We will apply the notion of entropy in a very similar way for software systems. Maintainability of a source code is usually defined as a measure of the effort required to perform specific modifications in it. Assuming that the higher the disorder, the more effort is needed to perform the modifications, maintainability can be interpreted as a measure of the disorder; i.e. entropy of the source code.

Our approach rests on two basic assumptions:

1. Making changes in the source code does not decrease the disorder provided that one does not actively work against it. In other words, when making changes to a software system without explicitly seeking to improve it, its maintainability will decrease, or at the very least it will remain unchanged. In addition, we assume that the decrease rate of maintainability is linearly proportional to the amount of lines modified at time t.

2. The amount of changes applied to the source code is linearly proportional to the effort invested, and to the maintainability of the code. In other words, if one applies more effort, the code will change faster. In addition, a more maintainable code will change faster, even if the applied effort is the same. Another interpretation is that the effort on code change is inversely proportional to the maintainability at a particular time t.

Before formalising these assumptions, we will introduce the following mathematical notions:

- $\mathcal{S}(t)$ - the size of the source code at time *t*, measured in lines of code.
- $\lambda(t)$ - the change rate of the source code at time *t*; i.e. the probability of changing any line independently. $\mathcal{S}(t)\lambda(t)$ is the expected number of lines changed at time *t*.
- k - a constant for the conversion between different units of measure. Our approach handles two scalar measures; namely, maintainability and cost. We will not fix particular units of measure for each, but instead we will introduce the conversion constant k. In the following, we will assume without any loss of generality that cost can be expressed by any measure of effort like salary, person month or time, while maintainability may have some other scalar measure. In practice, after fixing the unit measures for each, k can be estimated from historical project data.
- $\mathcal{C}(t)$ - the cost invested into changing the system up to time *t*, measured from an initial time $t = 0$. Obviously, $\mathcal{C}(0) = 0$.
- $\mathcal{M}(t)$ - maintainability (i.e. disorder) of the system at time *t*.

In the following, we assume that modifications do not explicitly mean code improvement, only new functionality is being added to the system and no refactoring or other explicit improvements are performed. In this case, the first assumption can be formalised as follows:

$$\frac{d\mathcal{M}(t)}{dt} = -q\mathcal{S}(t)\lambda(t) \qquad (q \geq 0), \tag{4.1}$$

meaning that the rate of decrease in maintainability is linearly proportional to the number of lines changed at time *t*. The constant factor q is called the *erosion factor* which represents the amount of "harm" (decrease in maintainability) caused by changing one line of code. The erosion factor depends on many internal and external factors like the experience and knowledge of the developers, maturity of development processes, quality insurance processes used, tools and development environments, the programming language and the application domain. The $q \geq 0$ assumption makes it impossible for the code to improve by itself just by adding new functionality. This assumption is in accordance with Lehman's laws of software evolution, which state that the complexity of evolving software increases, while its quality decreases at the same time.

Formalising the second assumption leads to the following equation:

$$\frac{d\mathcal{C}(t)}{dt} = k\frac{\mathcal{S}(t)\lambda(t)}{\mathcal{M}(t)}. \tag{4.2}$$

The nominator represents the amount of change introduced at time t. The formula says that the utilisation of the cost invested at time t for changing the code is inversely proportional to maintainability.

Solving the above system of ordinary differential equations yields the following result:

$$\mathcal{C}(t_1) - \mathcal{C}(t_0) = \int_{t_0}^{t_1} k\frac{\mathcal{S}(t)\lambda(t)}{\mathcal{M}(t)}dt = -\frac{k}{q}\int_{t_0}^{t_1} \frac{\dot{\mathcal{M}}(t)}{\mathcal{M}(t)}dt =$$

$$= -\frac{k}{q}\left[\ln\mathcal{M}(t_1) - \ln\mathcal{M}(t_0)\right] = -\frac{k}{q}\ln\frac{\mathcal{M}(t_1)}{\mathcal{M}(t_0)}. \tag{4.3}$$

Expressing from the above equation in terms of $\mathcal{M}(t)$, we get one of our main results:

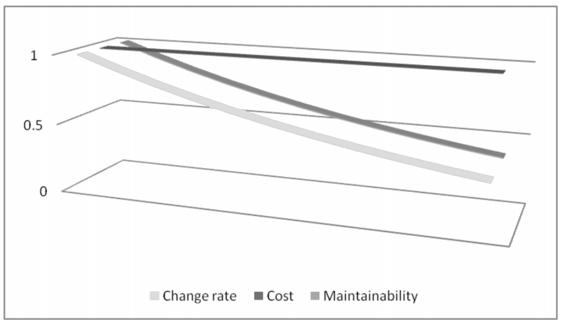
$$\mathcal{M}(t_1) = \mathcal{M}(t_0)\, e^{-\frac{q}{k}(\mathcal{C}(t_1)-\mathcal{C}(t_0))}, \tag{4.4}$$

which suggests that the maintainability of a system decreases exponentially with the invested cost to change the system. The erosion factor q affects the decrease rate of maintainability. It is obvious that for a higher erosion factor the decrease rate will be higher as well. It is crucial for software development companies to keep the erosion factor as low as possible – for instance by training the employees, improving processes and utilising sophisticated quality assurance technologies.

Although the formula does not provide a way of getting an absolute measure for maintainability, a *relative maintainability* for the system can easily be defined. Indeed, by letting $t_0 = 0$, and defining $\mathcal{M}(0) = 1$, we get the following function for maintainability:

$$\mathcal{M}(t) = e^{-\frac{q}{k}\mathcal{C}(t)} \tag{4.5}$$

To understand how it works, let us consider two artificial scenarios. The left hand diagram in Figure 4.1 shows the case where the invested effort is constant over time. In this case, both the maintainability $\mathcal{M}(t)$ and the change rate $\lambda(t)$ decrease exponentially. The right hand diagram shows the case when one intentionally wants to keep the change rate of the system constant. Now, the maintainability decreases linearly until it reaches zero, while the cost increases faster than an exponential rate. The cost will reach infinity in finite time precisely when maintainability reaches zero, meaning that any further change would require an infinite amount of effort. This is, of course, just a theoretical possibility, as no one disposes with infinite resources, which would be required to degrade the maintainability of a system to absolute zero.



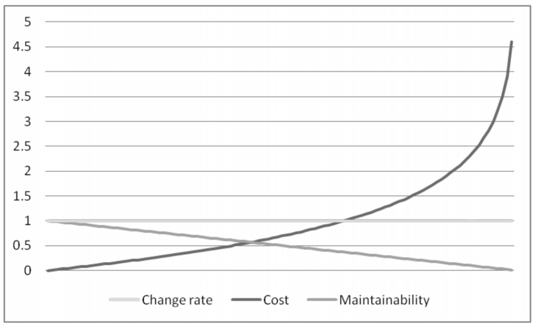


Figure 4.1: Changes of *Change rate* ($\lambda(t)$), *Maintainability* ($\mathcal{M}(t)$) and *Cost* ($\mathcal{C}(t)$) when the cost of development is constant (left) and when the change rate is constant (right) over time.

The challenge in applying the model to real-world software systems lies in the proper handling of the erosion factor q. While the other model parameters (k and $\mathcal{C}(t)$) can be readily computed, quantifying the erosion factor, which measures the "harm" caused by changing one line, is non-trivial. In contrast, if there was an absolute measure of maintainability, the constant, project specific erosion factor q could easily be computed using Equation 4.5. Furthermore, by having an absolute measure for q as well, the erosion factors of different projects, organisations could then be compared. The analysis of the causes of the differences would make it possible to lower the erosion factor, e.g. by improving the processes and training people better. In addition, the overall cost of development could also be expressed explicitly via the model:

$$\mathcal{C}(t) = -\frac{k}{q}\ln\left|1 - \frac{q}{\mathcal{M}(0)}\int_0^t \mathcal{S}(s)\,\lambda(s)\,ds\right|. \tag{4.6}$$

To compute the future development cost, all that would be required would be to have an estimate for the change rate $\lambda(t)$ over a specified time period.

In the previous chapter we presented an approach for getting an absolute measure of maintainability for software systems. We used source code metrics and benchmarks to probabilistically approximate a benchmark-independent measure of maintainability. Now, we will use this approach to compute absolute maintainability, to obtain an absolute erosion factor q, which can then be used to estimate further development costs and to compare the erosion factors of different projects and organisations.

4.1.2 Validation of the model

In order to evaluate the presented cost model, we analysed a large number of consecutive versions of five different Java projects. Three of these are commercial, closed source systems, which will be referred to as *System-1*, *System-2* and *System-3*. To facilitate the repeatability of the experiments, we performed an analysis of two open source systems as well. Some of the relevant details on the systems analysed are listed in Table 4.1.

System	Nr. of revisions	First date	Last date	System size[3] interval	Nr. of authors
System-1	149	06/03/2011	01/31/2012	14175-24861	7
System-2	357	05/09/2008	03/09/2010	53262-143017	21
System-3	641	11/05/2010	10/12/2010	128653-148903	12
jEdit[4]	1370	09/02/2001	07/25/2006	30986-96203	18
log4j[5]	1889	12/14/2000	08/15/2007	1464-25642	17

Table 4.1: Properties of the systems analyzed

The reader may have noticed that the model presented here contains parameters which must be approximated in order to be feasible. To compute the k and q parameters of the model, we need to know $\mathcal{C}(T_0)$ for some $T_0 > 0$; i.e. the cost of development up to time T_0. This can usually be estimated by using historical project records, but it can also be approximated in other ways. After k and q are computed for some time T_0 (i.e. the model is trained), the model can be used to make predictions for $\mathcal{C}(t), t > T_0$. Unfortunately, historical records regarding the development costs were not available in any of the cases. Therefore, in order to perform the evaluation, we were forced to make assumptions regarding the costs: we assumed that the costs of the development were proportional to the elapsed time. Provided that, in case of industrial systems, the teams work on a project with relatively little variation in their number, this assumption does

[3]Measured by the total of non-empty non-comment lines of code (TLLOC)
[4]https://jedit.svn.sourceforge.net/svnroot/jedit/jEdit/branches/4.5.x
[5]http://svn.apache.org/repos/asf/logging/log4j/branches/BRANCH_1_3

not seem too restrictive. Unfortunately, this might not be the case with open source systems as there is usually no stakeholder enforcing steady expectations regarding the invested effort. We will treat the case of open source systems as a threat to validity because of this reason.

We performed the evaluation based on the following steps:

1. First, we checked out every revision of the source code of each system from their configuration management systems.
2. We calculated the maintainability of each source code revision by using our probabilistic source code quality model [10]. We used this value as an approximation for $\mathcal{M}(t)$.
3. For each source code revision, we computed the number of altered source code lines (added, deleted and modified) compared to the previous revision. The value got in this way is precisely equal to $\mathcal{S}(t)\,\lambda(t)$, so computing $\mathcal{S}(t)$ explicitly is unnecessary here.
4. We computed estimates for k and q from Equation 4.2 and Equation 4.5, respectively, at some time $T_0 > 0$. The estimates for k and q are the following:

$$k = \mathcal{C}(T_0)\left(1/\int_0^{T_0} \frac{\mathcal{S}(t)\,\lambda(t)}{\mathcal{M}(t)}dt\right), \tag{4.7}$$

   and

$$q = -\frac{k}{\mathcal{C}(T_0)} \ln \frac{\mathcal{M}(T_0)}{\mathcal{M}(0)}. \tag{4.8}$$

5. These estimates, being constants according to our model, are valid for time $t > T_0$, and can be used to make predictions using Equation 4.6. The predicted costs will be denoted by $\tilde{\mathcal{C}}(t)$.

To compute the number of modified lines of code, in step three we applied a heuristic algorithm that combines *diff*s returned by the SVN client. We consider this as a threat to validity as well.

Different aspects of our results will be summarised in the later sections of this chapter.

The maintainability of evolving software decreases over time

The dark lines on the right hand diagrams in Figure 4.2 show how the maintainability $\mathcal{M}(t)$ changes as a function of time t, measured in number of revisions. All of the figures show a decreasing tendency in maintainability as more effort is put into the development of these systems. To test our intuition of decreasing functions, the linear regression lines and their equations are also shown on the diagrams. All the coefficients of x being negative, an average decrease in maintainability results in each case. This trend is in general agreement with Lehman's laws [70].

Maintainability and development cost are in exponential relationship with each other

Let $\tilde{\mathcal{M}}(t)$ denote the predicted maintainability computed using Equation 4.5 and $\tilde{\mathcal{C}}(t)$ (the cost function predicted by the model). Clearly, $\tilde{\mathcal{M}}(t)$ decreases exponentially as a function of $\tilde{\mathcal{C}}(t)$. It is sufficient to show that the real cost $\mathcal{C}(t)$ correlates well with the predicted cost $\tilde{\mathcal{C}}(t)$ and real maintainability $\mathcal{M}(t)$ with the predicted maintainability $\tilde{\mathcal{M}}(t)$ for some fixed k and q. It would mean that for some parameters the model describes the real world fairly well. Consequently, the measured maintainability should decrease exponentially as a function of real costs or something similar.

We can compute estimates for any time $T_0 > 0$, as suggested in step 4 above. Obviously, for larger T_0 values, the estimates are better, provided that more historical data is available for training the model. By taking the last revisions; i.e. the biggest possible T_0, we get the best estimates for k and q. These constants are then used to compute $\tilde{\mathcal{C}}(t)$ and $\tilde{\mathcal{M}}(t)$ for any $t \geq 0$.

The left-hand diagrams in Figure 4.2 show both $\mathcal{C}(t)$ (dark) and $\tilde{\mathcal{C}}(t)$ (light) functions. On the right-hand side, the dark lines show the changes of $\mathcal{M}(t)$, while the light ones show $\tilde{\mathcal{M}}(t)$. The diagrams also show the Pearson's correlations between the real and the predicted curves. The high correlations indicate that both cost and maintainability are quite well described by the model too. It means that maintainability and cost probably have an exponential relationship with each other like that described by the model. In the case of *System-3* and *log4j* the correlations are slightly worse than in the other cases. The reason of this might be that the time period of the analysis was relatively short, and according to the SVN logs, lots of refactoring work was done as well.

The presented model is able to predict future development costs based on change rate of the code, to good accuracy

Previously we showed that the model parameters k and q can be chosen such that the model describes real-world costs and maintainability quite realistically. Figure 4.3 shows the estimated k, q and q/k values for each system.

Based on the diagram, the biggest harm was caused in *System-1* when one line was changed, as the erosion factor q is the largest in this case. This might be due to the

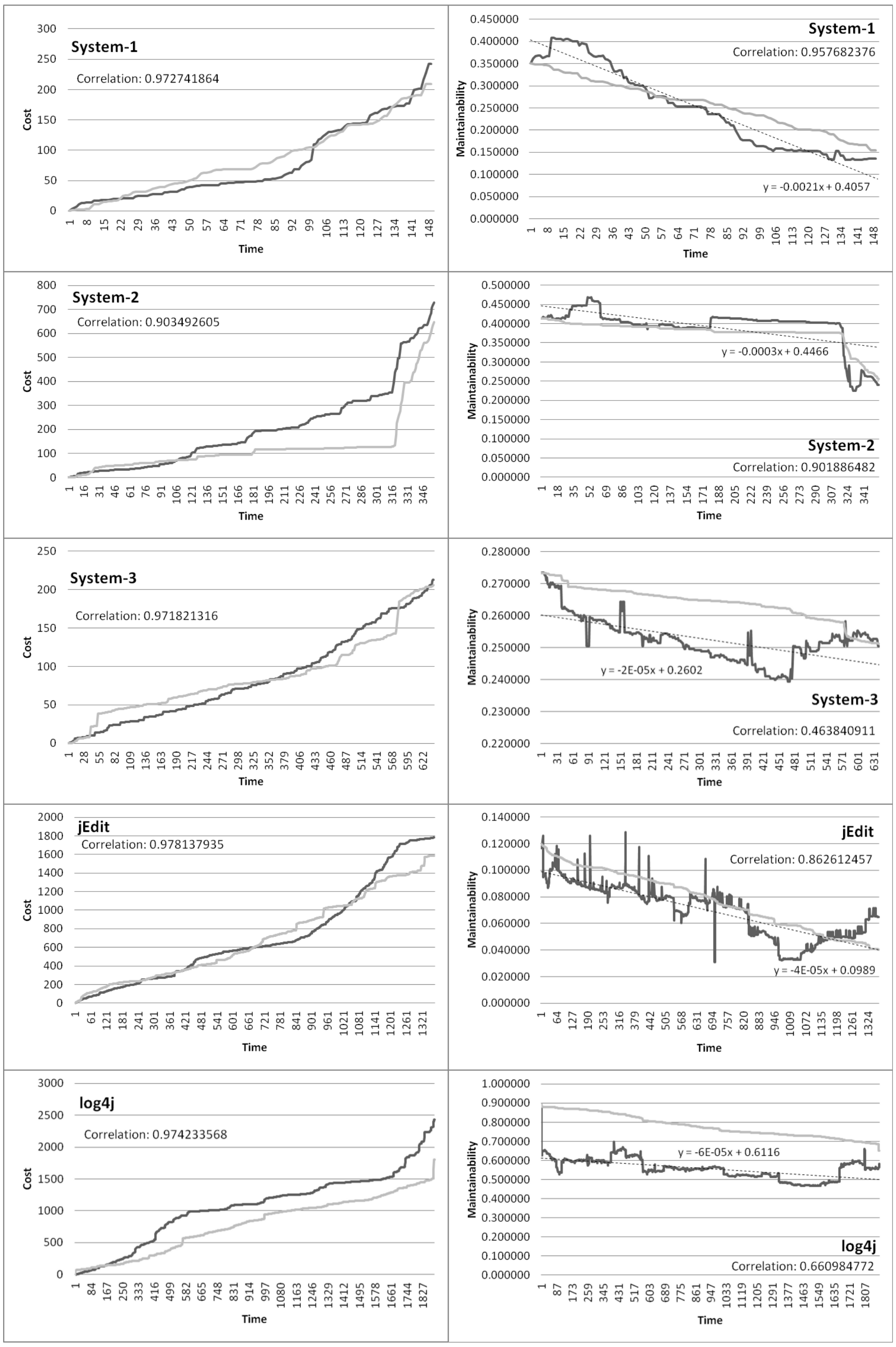


Figure 4.2: Estimated and real costs and maintainability as functions of time

rapid and intensive development of *System-1* during the period studied. This is also the system whose maintainability decreases the most, provided that the applied effort is the

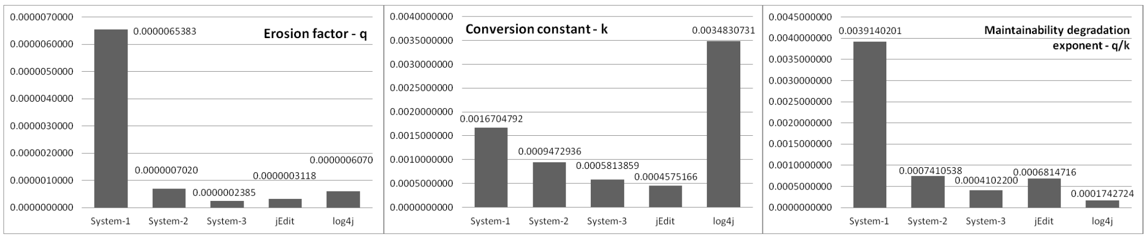


Figure 4.3: The calculated constant values for the different systems

same, because the ratio q/k is also the largest in this case. The conversion constant k is the largest for *log4j*, meaning that same amount of effort induces fewer changes compared to the other systems.

To estimate the cost of a new development, Equation 4.2 of the model requires that one has an estimate of the total amount of lines that will change, and the function describing maintainability change in the future. Although the total number of changes can be estimated in advance, based on requirements and impact analysis [67], maintainability is obviously unavailable before the changes have been committed, and maintainability has been measured. Luckily, the erosion factor introduced by Equation 4.1 makes it possible to approximate the future maintainability based on estimated change rates. Future development costs can be computed using Equation 4.6 without needing to know the change of maintainability in advance.

In order to validate the predictive power of our model, we performed future estimations with different window sizes measured in time. For a particular window size $n > 0$, we used the model to compute the estimated cost at time t, based on the already known cost at $t - n\,(\geq 0)$ and the planned amount of changes between $t - n$ and t. In other words, at time $t - n$ we attempt to estimate the overall cost at time t by knowing the overall cost up to time $t - n$ and the planned amount of future changes. In this way, for a particular window of size n, we get a sequence of predicted costs, for time $n + 1, n + 2$, etc. The window sizes vary from 1 to the largest possible ones, i.e. the number of revisions available. When the window size is 1, it means that the development cost of a revision is being approximated based on the previous revision, and the changes between them. In the case of the largest possible window, the overall development cost of the whole period is estimated based on the initial cost (which is zero), and the future changes. For each window size, we computed both the *mean squared error* [2] and *Pearson's correlation* between the real costs and the ones predicted by the model.

For comparability reasons, we also performed another, classical type of cost estimation. Namely, to estimate future costs, we computed the average cost of a change up to time $t - n$, then interpolated the future cost by multiplying the average change cost with the amount of overall change up to time t. In other words, we computed the average change cost based on historical data, and expected it to remain the same in the future. This classical model differs from ours as it does not take the change of maintainability over time into account, which makes the changes evermore expensive to do. We will refer to this classical type of cost estimation as a *linear prediction*.

The left-hand diagrams in Figure 4.4 tell us how the mean squared errors (*MSEs*) vary for various window sizes, while on the right-hand side the correlations of the predicted and real costs are shown. In both cases, the x-axis represents the size of the window, measured in number of revisions (i.e. time) and the y-axis prepresents the MSE and Pearson's correlation values, respectively.

Even though it appears that both models become more and more precise for larger window sizes, this occurs only because the prediction sequences are getting shorter. For example, with the largest possible window size, only one cost value is predicted; namely, the last one.

It can be seen that the predictions made by our model outperform the classical linear model, which does not take the changes of maintainability into account. The differences are especially noticeable for larger window sizes; i.e. long-term predictions. Actually, it is a natural phenomenon because changes of maintainability are more significant over longer periods of time.

4.1.3 Limitations

First of all, our cost model is based on two assumptions stated in Section 4.1.1. If these two assumptions do not hold, our cost model may be invalid. However, our practical experience and research feedback tell us that these assumptions are quite reasonable.

Another threat to the validity of the results is that we assume that the k conversion and q erosion factors in the model are constants. It may be that these factors actually change slightly over time. But even if this is the case, it only means that further improvements in the prediction model are achievable. Our model is a simplistic cost model that appears to be most promising in its current form based on the empirical results given in Section 4.1.2.

Due to the lack of real available data, we had to apply heuristics several times in the study. To calculate the total amount of altered lines between two revisions of the system, we used the SVN diff command that returns only added and removed lines. Modified lines are shown by consecutive inserted and deleted lines. Although our algorithm for calculating modified lines might not be totally precise, it does not affect the results obtained too much. Our experiments showed that we get similar results using just the number of inserted lines as a measure of the total changes.

Being unable to collect real development efforts from tracking systems, we just assumed that the amount of cost invested in the development was proportional to the elapsed time.

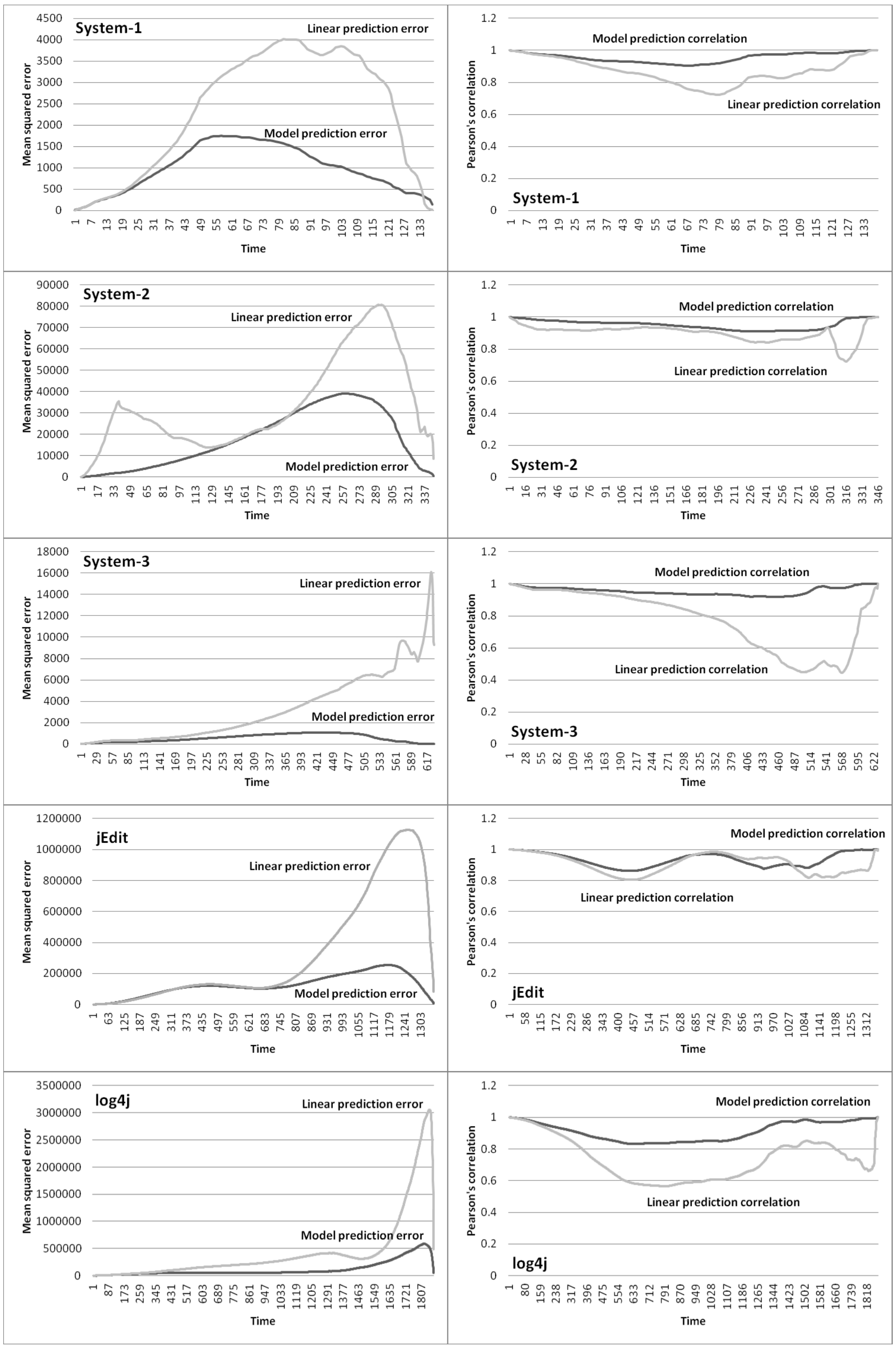


Figure 4.4: The mean squared errors and correlations between the linear and model predictions

The reason for this is that usually there is a fixes-person team that develops the software, each member putting a roughly constant amount of effort into the development. This was the case for the three proprietary systems that we analysed, but a possible threat to validity might be that this assumption is not valid for open source systems.

One might think that the model cannot deal with refactoring and other improvements, as it assumes that only pure feature developments are allowed. Fortunately, considering these activities as part of the development- or quality assurance process, which are meant to moderate quality degradation, they are implicitly encoded in the erosion factor q. In particular, q is smaller in cases where refactoring and other improvements are performed regularly or even occasionally. Therefore, we do not regard this as a threat to validity.

A major limitation of the approach, however, is that the predictions are made based on the amount of changes of lines in the system, which makes the model less useful in practice. This restriction arises from the simplicity of the model. Still the model can be readily amended to use function points instead of line changes, producing a more practical prediction model. We decided to use line changes because they can easily be extracted from a configuration management system, which is not the case with function points.

4.2 Summary

In this chapter we presented a cost model based on source code maintainability. The model describes the relationship between the change rate of the source code, the maintainability of the system and the cost of development. Measuring source code maintainability is one of the essential parts of our cost model. To measure the maintainability, we applied the probabilistic quality model presented in Chapter 3.

In order to validate our approach, we analysed five Java systems (three proprietary and two open source) and collected a vast amount of data. Altogether 4,396 different revisions of the systems and over 1 million lines of code changes were examined. We also checked all the change set logs of those commits that led to a greater improvement in the maintainability of the system manually. In most of the cases, some kind of refactoring (e.g. replacing components) was the reason for the improvement, which corroborates our hypothesis that pure development does necessarily not improve the maintainability of the software system. An analysis of the empirical data shed light on the following things:

- The maintainability of an evolving software system decreases over time.
- Maintainability and development cost are related to each other in an exponential way.
- The presented model is able to predict future development costs based on an estimated change rate of the code to reasonable accuracy.

Although our model and analysis procedure contains some threats to validity, we think that the results are valuable, and reflect the underlying theoretical connection between development cost and source code maintainability for a complex software system.

"The key to heaven's gate cannot be duplicated."

Doug Horton

Chapter 5

Code duplications from the perspective of code evolution

It is believed by many academic and industrial experts that source code duplications represent a significant threat to the maintainability of an evolving software system. During software development, when developers are under the constant pressure of deadlines, it is common practice to reuse source code by simply copying parts of it, and eventually performing smaller modifications on it (it has been estimated that both industrial and open source systems contain, on average, about 20% of duplicated code [77]). While this approach can reduce software development time, the price in the long-term will usually be paid in the form of increased maintainability costs. One of the primary concerns is that if the original code segment needs to be corrected, all the copied parts need to be checked and modified accordingly as well. By inadvertently neglecting to change the related duplications, the programmers may leave bugs in the code and introduce inconsistencies. Cory et al. [55] pointed out that there exist situations when duplicating code might even be beneficial (e.g. one way to evaluate possible new features for a system is to clone the affected subsystems and introduce the new features there, in a kind of sandbox testbed). Fowler [74], however, argues that code duplications are the most important among *bad code smells* (a particular part of the source code that reflects some kind of design or implementation-related flaw) and they should be eliminated aggressively by programmers. Kim et al. [57] found that the immediate elimination of volatile clones might not be cost-effective, as the rarely changing duplications may not be worth eliminating because there are usually lots of them and they might start causing problems only when they evolve (which happens rarely). As the question of harmfulness or usefulness of code duplications is still an open question, our intention is not to answer this question, but to provide new insights, theoretical and empirical results to the ongoing debate. The real threat regarding code duplications does not lie in their existence, but the fears arise in connection with their evolution. Therefore, tracking the evolution of individual clones is essential from a maintenance point of view.

In this chapter, we present an efficient algorithm for tracking duplications across subsequent versions of a software system. Using this concept, we will introduce the notion of *clone smells* which, similar to bad code smells, refer to particular code portions that should be further inspected manually. The smells are defined based on the possible categories of clone evolution patterns. We use clone smells to assess the impact of code duplications on maintainability. We will show that clone smells may serve as a basis of an efficient clone management technique that reduces risks and maintenance efforts arising from the duplications. Lastly, we will examine the relationship between code duplications to source code coupling.

5.1 Tracking the evolution of code clones

Now, we present an approach for mapping the clone instances of one particular version of the software to another, based on a similarity distance function. There are basically three kinds of approaches that map clones between different versions of a software. Two of the techniques utilise single-version clone detection. The first set of approaches detects clones in a reference version and calculates those of the following versions using change information from a version repository [6, 62]. The second set of approaches detects clones for all versions of the program. Clones are then retroactively mapped using heuristics [24, 32, 11]. The third category of approaches uses incremental clone detection for finding clones in subsequent versions of a software system. During the incremental phase, it also maps the clone instances between the versions by using change information got from a version control repository [37].

Here, we follow the second approach, by defining a heuristic we call *evolution mapping* between two particular code fragments taken from different versions. To be precise, a clone from a particular version is mapped to a clone in another version if the second code fragment has evolved from the first one. The mapping between the clones is trivial in some special cases, but in general, a sophisticated approach is required.

Our approach of tracking code duplications consists of the following steps:

1. Clone detection phase: identify code clones in each available version of the software independently.
2. Evolution mapping phase: map the identified clones onto each other across the versions of the system.

5.1.1 The Evolution Mapping

In this section, we shall describe the evolution mappings between clone instances across the versions of the software system.

In the following, C_i, C_j will denote arbitrary clone instances, not necessarily taken from the same class. We shall use the notion of C_i^v, C_j^v, etc. to emphasise version v of the system from which the instances originate. In a similar way, CC_i^v, CC_j^v refer to clone classes of version v, while CI^v stands for the set of instances extracted from version v of the underlying system. We will use CC^v to denote the set of all clone classes in version v of the system.

The *evolution mapping* is, in our context, a partial injective mapping of the clone instances of version v_1 to a version v_2 of the subject system:

$$e : G \subset CI^{v_1} \to CI^{v_2}$$

The intuitive meaning is that the images of a mapping have evolved from the instances of the domain. The mapping is considered to be *partial*, as there might be some clone instances that have vanished in the subsequent version. Partiality is expressed by the set G, which stands for the domain of mapping e, and it is a subset of all the instances from version v_1. *Injectivity* means that every clone instance from the newer version has evolved from at most one earlier instance. We would like to exclude the possibility of one clone instance having more than one predecessor. Keeping in mind that in larger systems there might be several thousand clone instances and that the asymptotic number of possible mappings grows exponentially with the number of elements of the sets (due to partiality), it turns out that finding the optimal evolution mapping is not easy. Not just the location of a clone instance may change, but also its syntactic structure, unique name, etc., which makes really hard to find the corresponding code fragments. Initially, every pair of instances from two subsequent versions should be considered as a potential pair of the mapping.

By trying to analyse how people would associate code parts that have evolved from each other (without any information coming from a configuration management system), we concluded that there are several features and constraints which should be taken into account. Starting with the constraints, we define one obvious rule in order to reduce the number of possible mappings:

R_1 : *The* head *of the clone classes (represented by the two instances) must be the same.*

The *head* of a clone class is the type of the root node of the syntax tree representing any/every clone instance of the class. This constraint prevents the mapping of different types of clone instances to each other. For example, it is intuitively clear that a *function node* should not be considered as the predecessor of a *class node*. By applying this "cutting" rule, a reasonable amount of computation can be saved, as its verification is simple and fast. For those clone instance pairs that satisfy the above rule, an evaluation of a similarity distance function is required, which is computationally more expensive and which is aggregated from the following features:

F_1 : Name of the file containing the clone instance

F_2 : Ordinal number of the clone instance in its class

F_3 : The unique name of the *head node* – if the unique name exists (just for named entities)

F_4 : Otherwise, if a unique name cannot be assigned to the head node (in general, for code fragments inside a function body) the unique name of the first *named* ancestor in the AST on the path going towards the root node of the AST is taken (e.g. the unique name of the function containing it)

F_5 : The relative position of the code segment inside its first named ancestor

F_6 : The lexical structure of the clone instance

It should be mentioned that the above features subsume clone instances that are whole syntactic units, which holds in our case. If other types of clone-detectors (lexical-based, metric-based, etc.) are used, some of the above features might not be suitable.

The intuitive meaning of the features is the following. If two pieces of code taken from different versions of the same system are examined from a code evolution perspective, the first thing one should check is how similar the two pieces of code are (F_6). If their similarity reaches a subjective level of acceptance, the second question is how close they are with respect to the entity that contains it (e.g. function or class) (F_5). Afterwards, one should check the names of the entities that contain it (e.g. functions) (F_4). If there are more clone instances in the same piece of code (function, class, etc.) close to each other, the mapping should preserve the order in which they appear (F_2). The name of the file should also be considered, as it could happen that there are two potential candidates for which all the attribute values are the same, but just the names of the files that contain it differ. (F_1). In the case of named entities the situation is much simpler: one should just compare the unique names of the two instances (F_3).

For example, if two clone instances from different versions have the same values for the above attributes, it may be that they have an evolutionary relationship with each other (they are lexically identical, they are in the same file, in the same function, at the same position, etc.). However, if the attribute values of the instances differ, that still does not mean that they are not related (the name of the file might have been changed, the position of code fragment might have been altered or the code fragment itself might have lexically been changed), but the likelihood that they are related is smaller.

It should be added that F_1-F_6 are only attributes of the clone instances which should be taken into account when trying to map them across the versions. These attributes can be used to compute a distance function between the clone instances arising from different versions of the software. The goal is to construct a mapping (by using a distance function), between clone instances arising from consecutive versions of a system, which is in some sense *optimal*; i.e. it relates as many pairs as possible, but makes mistakes as rarely as possible. We will give a more formal description of optimality later on.

Even though the notion of evolution mapping is not specific to duplicated code (it can be applied to any two fragments of code), it cannot be used on its own to track code evolution because the technique requires a list of candidates which is in our case provided by a clone detector.

In the following we will formalise the above-mentioned concepts. For each clone instance pair that satisfy the cutting rule R_1, the values of the listed features are computed and the results are represented by a similarity distance function that in some sense reflects all the features at once. Each feature F_i contributes to the distance function by a given predefined weight α_i. Let C_i and C_j be two particular clone instances, and let $D_k\left(C_i, C_j\right)$ be the distance value of the F_k features of these instances. Furthermore, let

$$D^*\left(C_i, C_j\right) = \sum_{k=1}^{6} \alpha_k D_k\left(C_i, C_j\right) \tag{5.1}$$

denote the overall composed similarity distance value of the features for the two given instances.

In our case the $D_k\left(C_i, C_j\right) = 0$ condition means an exact match of the F_k features. If the matching gets worse, the value of the function will increase. The functions D_k have different ranges for different values of k, based on their definitions (see Section 5.1.2). If the distance between two instances is zero then they are definitely maps of each other (based on our earlier discussion) and will not be considered when creating an optimal mapping. In this way, the number of possible mappings will be considerably reduced and a significant amount of CPU time will be saved.

After the similarity distances for each instance-pair have been computed, an optimal evolution mapping can be found. Considering just the pairs with non-zero distances (the rest can be excluded owing to our earlier remark), we arrive at a so-called *assignment problem* [98], which is a fundamental combinatorial optimisation problem. In its general form, the assignment problem is the following:

There are a number of agents and a number of tasks. Any agent can be assigned to perform any task, incurring some cost that may vary depending on the agent-task assignment. It is required that all the tasks be performed by assigning exactly one agent to each task in such a way that the total cost of the assignment is minimised. The number of agents and the number of tasks are equal.

The optimal solution of the above problem is an algorithm with polynomial time and space complexity. It is known as the *Hungarian method* [63], or *Munkres assignment algorithm* [82].

By considering the clone instances of the version v_s as agents, the instances of version v_t as tasks, and the similarity distances as costs, the original problem can be reduced to an assignment problem. If the number of instances is not the same (as is expected), virtual nodes should be added to the version which has fewer instances. These virtual nodes are unlike every instance in the other version (the distances are infinite). The solution of the assignment problem is a bijection between the agents and tasks (i.e. clone instances), but a partial injective mapping would be needed. Partiality needs to be enforced, otherwise it could happen that an instance would be mapped onto a very dissimilar (i.e. distant) one, just because "something needs to be assigned to everything". To resolve this issue, a threshold value β is introduced which serves to prevent mappings between instances being too dissimilar. In essence, the β value makes the mapping rather partial than bad. Thus, Equation 5.1 now has the following form:

$$D\left(C_i, C_j\right) = \begin{cases} D^*\left(C_i, C_j\right), & \text{if } D^*\left(C_i, C_j\right) \leq \beta \\ \infty, & \text{otherwise} \end{cases}$$

After the optimal mapping has been found, the edges having an infinite weight are deleted (it affects the pairs for which the similarity distance exceeds β and those in which one of the nodes is virtual). The remaining edges make up the optimal evolution mapping between the two versions.

When applying this procedure, two essential questions arise. They are:

- How should the similarity distance functions for the separate features be defined?
- How should the weights and the cutting threshold value be determined?

5.1.2 Similarity distance functions

As the features F_1, F_3, F_4 and F_6 operate on lexical (string) values, we employed a modified *Levenshtein distance* [71] (also known as the *edit distance*) to measure their distances. In our case the distance between two strings is their edit distance divided by the length of the longer one. In this way, we obtain a distance function for each of the above features lying between 0 and 1. After experimenting we found that for the F_6 feature it is not enough to compare the AST node types; it is more effective to use the original source code. The reason for this is that the node types of the AST tree represent just a high-level abstraction of the source code: variable names, function names and certain lexical types of information are missing. Hence if there are two or more instances of the same clone class, they cannot be distinguished, even though they might use entirely different variable names, or they might call different functions. To resolve the issue, we generated the source code fragments from the AST for the instances, and the resulting strings were compared by using textual similarity. In this way, the generated code parts were formatted in the same manner, which would generally not be the case if the original code fragments were taken.

With F_2 and F_5, a different approach is required, as they have numerical values. The F_2 feature here is responsible for the correct order of the clone instances. Now suppose that there are two textually similar instances of the same clone class inside the same function body near each other. They can only be distinguished from each other by the order of their appearance in the code.

Formally, if C_i and C_j are two instances, and $O(C_k)$ is the order of appearance of the C_k instance in its clone class, then let $D_2\left(C_i, C_j\right) = |O(C_i) - O(C_j)|$ be the distance function for feature F_2. If the order of C_i and C_j are the same, the feature F_2 does not contribute to the overall similarity distance.

The situation is slightly more complicated for F_5. This feature is responsible for measuring the displacement of the instances with respect to their first named ancestor. Figure 5.1 illustrates the notions we use for computing an estimate for the D_5 distance function. Let $L_{BI}(C_i)$ be the number of AST nodes from the beginning of the first

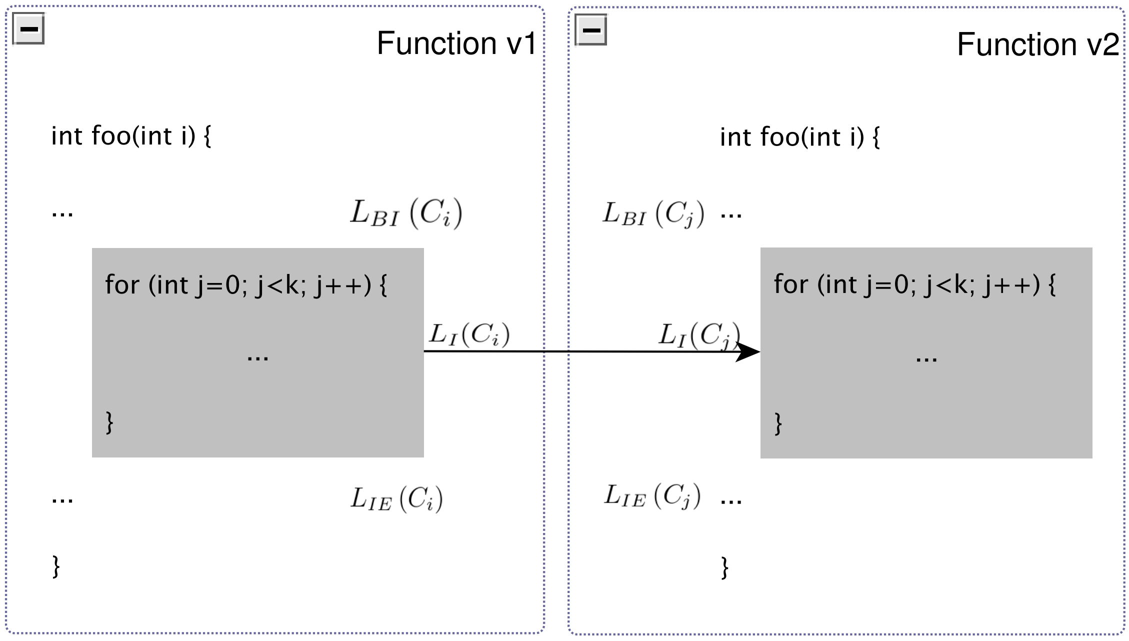


Figure 5.1: Diagram showing how F_5 is computed.

named ancestor node to the beginning of the instance, let $L_I(C_i)$ be the length of the instance (measured by the number of AST nodes), and let $L_{IE}(C_i)$ be the number of nodes from the end of the instance to the end of the first named ancestor node. Using these notions, let us define

$$D_5\left(C_i, C_j\right) = \sqrt{\left(\frac{L_{BI}\left(C_i\right)}{L_{BI}\left(C_j\right)} - L_{ij}\right)^2 + \left(\frac{L_{IE}\left(C_i\right)}{L_{IE}\left(C_j\right)} - L_{ij}\right)^2},$$

where

$$L_{ij} = \frac{L_{BI}(C_i) + L_I(C_i) + L_{IE}(C_i)}{L_{BI}(C_j) + L_I(C_j) + L_{IE}(C_j)}.$$

With this formula, two code segments are said to be close to each other if the differences in the amount of AST nodes before and after the segments do not differ much relative to the overall size of the AST starting from the first named ancestor nodes. For example, if the code segments are located inside a function node, the value of D_5 will remain small even if the body of the function is stretched linearly (approximately the same percentage of new AST nodes is added before and after the given segment). By doing this, we get a “stretch-invariant” displacement-measuring distance function.

5.1.3 Weights and threshold

Now we need to address the question of weights and cutting values. Fortunately, the weights are meaningful, so many of them can be set by relying on human experience. Even if the optimal values are not easy to determine, the relative priorities are easy to estimate. For example, the similarity between two code fragments (F_6) is obviously more important than their relative location inside their first named ancestor node (F_5). Provided the two segments differ a lot at the lexical level, we can choose the one which is more similar, instead of the one whose position is more convenient.

As we would like to have an optimal mapping, we have to define and measure somehow the goodness of a particular mapping. A candidate mapping is good if it relates as many pairs as possible, while the probability of making mistakes (i.e. amount of false positives) is as low as possible at the same time.

The number of pairs is measured by the number of edges of the mapping, while the overall similarity distance (each edge has a similarity distance value which contributes to the overall similarity distance of the mapping) must strictly correlate with the amount of false positives (if the overall weight is high – there are more pairs which are far away from each other, but they are still related – the probability of having false positives is higher).

Figure 5.2 shows an example of two possible candidate mappings. The values on the edges represent weights, i.e. the similarity distance between two clone instances. In the left diagram, the overall similarity distance of the mapping is 5, while there are just 3 edges. In the right diagram, the overall similarity is 9, while there are 4 edges. Which

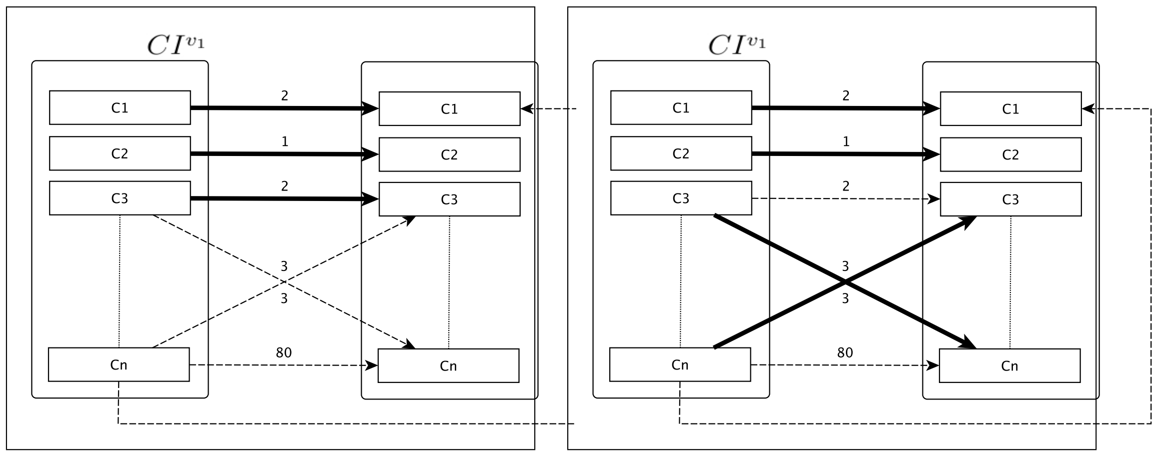


Figure 5.2: Example of candidate mappings

out of these two mappings is better? It depends on someone's subjective notion of goodness. We decided to take a weighted ratio of these two numbers. Formally,

$$\mathcal{C}\left(\vec{\alpha},\beta\right)=\frac{\left(\sum\limits_{C_i\in CI^{v_1},C_j=e(C_i)} D\left(C_i,C_j\right)\right)^n}{\|e\left(CI^{v_1}\right)\|}$$

where $n>0$, C_j is the map of C_i, $\|e\left(CI^{v_1}\right)\|$ is the number of edges of the mapping and the sum in the numerator expresses the overall weight of the edges involved in the evolution mapping. The smaller the $\mathcal{C}\left(\vec{\alpha},\beta\right)$ function value is, the better the mapping is – at least with our interpretation of goodness. Simply taking the sum of the distances would not be enough, as the trivial mapping with no edges would outperform any other (this is possible as the mapping might be partial). The parameter n is used to express the importance of a similarity distance compared to the number of edges. The higher values of n make small changes of the similarity distance less significant compared to the number of edges. This is important for smaller values of n as the optimisation algorithm may drop edges even for a small gain in distance. By increasing n, this should happen less frequently. Thus the proper value of this parameter needs to be determined empirically. However, for smaller values of n the number of false positives edges should be smaller and the amount of true negatives higher. When its value increases, the opposite should hold.

It is vital that the goodness of mapping defined above should depend only on the weights α_i and β. After the weights are fixed, the similarity distance between each pair of clone instances is computed, and after the Hungarian method described above selects the edges that will make up the optimal mapping in terms of the weights. We should note here that the number of degrees of freedom of the model is one, which means that multiplying all the α_i and β values by the same positive constant will not change the model itself (the resulting mapping would be the same). However, if the α_i values were to tend to zero then every instance would become increasingly similar to each other and the number of edges would also be high. As a consequence, $\mathcal{C}(\vec{\alpha}, \beta)$ would tend to zero, but not because the mapping estimate would be getting better, but because the features which should have been used to distinguish between the candidates would lose their importance. Luckily, we may assume that the $\vec{\alpha}$ vector lies on the surface of the six-dimensional sphere (i.e. $\sum \alpha_i^2 = 1$), otherwise when dividing the weights by this sum the assumption would hold (and would not change the mapping itself). With this idea, the $\vec{\alpha}$ vector cannot approach zero (actually its distance from the origin will always be equal to one).

Now, as the goodness of a mapping in terms of the weights can be measured, the need to apply an optimisation algorithm naturally arises. The cost function $\mathcal{C}(\vec{\alpha}, \beta)$ is a step-function (not differentiable, not even continuous at every point of its domain), hence to find its minimum value, probably the simplest solution is to apply a simulated annealing algorithm [58].

At each step of the iteration, the α_i and β values are independently varied by choosing random values of a normal distribution. The variance of the underlying distribution continuously decreases in such a way that it is proportional to the current temperature T. The probability of making a transition to a lower energy state is, in our case, always equal to one. Due to its high computational cost, we performed the above optimisation of the weights on only 36 equally distributed versions of the *Mozilla* system taken from the year 2006 (three different versions were taken from each month).

Initially, we assumed that each feature was equally important; i.e. the weights were the same. Because of the assumption that $\vec{\alpha}$ lies on the surface of the unit sphere, the values of α_i should be equal to $\frac{1}{\sqrt{6}} = 0.4082$. At the start, β was also set to this value. For the initial state, the overall distance was equal to 141.31 and there were $34,747$ edges between the consecutive versions. After the optimisation, $34,635$ edges remained, while the overall distance dropped to 4.87 (the sum of squares of the weights was necessarily equal to one throughout the whole process). As the overall dissimilarity was minimised in this way, the probability of having false edges (edges between instances which are not evolutionary related) was also smaller.

Table 5.1 shows the initial values of the weights and the values we got by applying this optimisation procedure. It turns out that the most important feature is F_2 (the ordinal number of appearance of the clone instances). This feature affects the decision by 28.9%. The textual similarity is also important – 23%. The least important feature is F_5 (the position of the code segment), which was what we expected. The small value of β allows the mapping to have edges only with very low $D(C_i, C_j)$ values.

The above choice of optimal weights results in an evolution mapping e which will be referred to as the *optimal evolution mapping* in the following sections. It should be

Weights	Initial	Optimised	Contribution
α_1	0.4082	0.3122	14.2 %
α_2	0.4082	0.6365	28.9 %
α_3	0.4082	0.2066	9.4 %
α_4	0.4082	0.4293	19.5 %
α_5	0.4082	0.1101	5.0 %
α_6	0.4082	0.5080	23.0 %
β	0.4082	0.0284	

Table 5.1: Initial and optimised weights of the model

mentioned that the portability of a such an optimised mapping should be further investigated. We shall treat this as an eventual threat to validity as we will also use the mapping for a different system (*jEdit*) other than it was trained on.

5.1.4 Evolution of clone classes

So far we have only focused on the evolution mapping of individual clone instances, which was defined as a partial injective mapping of the clone instances of one particular version onto the instances of another one. This concept can naturally be extended to clone classes in the following way. Let us suppose that $CC_i^{v_1}$ and $CC_j^{v_2}$ are two different clone classes taken from version v_1 and v_2 respectively. We say that $CC_j^{v_2}$ has evolved from $CC_i^{v_1}$ (i.e. $e\left(CC_i^{v_1}\right) = CC_j^{v_2}$) if and only if one of the following two conditions hold:

- There exists an instance $C_u^{v_1} \in CC_i^{v_1}$ for which $e\left(C_u^{v_1}\right) = C_v^{v_2} \in CC_j^{v_2}$ and $D\left(C_u^{v_1}, C_v^{v_2}\right) = 0$
- $\|\{e\left(C_k^{v_1}\right) : C_k^{v_1} \in CC_i^{v_1}\}\| > \frac{\|CC_j^{v_2}\|}{2}$ and $\|\{e^{-1}\left(C_k^{v_2}\right) : C_k^{v_2} \in CC_j^{v_2}\}\| > \frac{\|CC_i^{v_1}\|}{2}$

In this case, the evolution mapping between any two clone classes will be well defined.

Indeed, in the first case if the similarity distance between the instances is zero, it means that they are identical at the AST level, and the instances falling into the same clone class must also be identical at the AST level as well (due to the clone-detection approach used). Therefore there should not exist any instance from $CC_i^{v_1}$ that would be identical to an instance outside $CC_j^{v_2}$, as otherwise it would be identical to all the instances falling in $CC_j^{v_2}$ and therefore it would also need to be in the same class (which is a contradiction). In the other case, there is a one-to-one correspondence between over half of the instances of both classes. Of course, there must exist at most one such class in version v_2.

Figure 5.3 shows an example where two clone classes taken from one version cannot be mapped onto clone classes of the next version. Although the instances are mapped

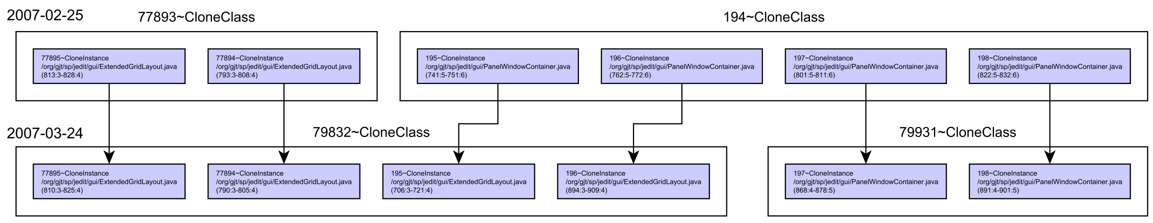

2007-02-25
77893~CloneClass
77895~CloneInstance /org/gjt/sp/jedit/gui/ExtendedGridLayout.java (813:3-828:4)
77894~CloneInstance /org/gjt/sp/jedit/gui/ExtendedGridLayout.java (793:3-808:4)
194~CloneClass
195~CloneInstance /org/gjt/sp/jedit/gui/PanelWindowContainer.java (741:5-751:6)
196~CloneInstance /org/gjt/sp/jedit/gui/PanelWindowContainer.java (762:5-772:6)
197~CloneInstance /org/gjt/sp/jedit/gui/PanelWindowContainer.java (801:5-811:6)
198~CloneInstance /org/gjt/sp/jedit/gui/PanelWindowContainer.java (822:5-832:6)
2007-03-24
79832~CloneClass
77895~CloneInstance /org/gjt/sp/jedit/gui/ExtendedGridLayout.java (810:3-825:4)
77894~CloneInstance /org/gjt/sp/jedit/gui/ExtendedGridLayout.java (790:3-805:4)
195~CloneInstance /org/gjt/sp/jedit/gui/ExtendedGridLayout.java (706:3-721:4)
196~CloneInstance /org/gjt/sp/jedit/gui/ExtendedGridLayout.java (894:3-909:4)
79931~CloneClass
197~CloneInstance /org/gjt/sp/jedit/gui/PanelWindowContainer.java (868:4-878:5)
198~CloneInstance /org/gjt/sp/jedit/gui/PanelWindowContainer.java (891:4-901:5)

Figure 5.3: Example of clone class evolution

appropriately, it is not well-defined, which clone class should be mapped onto which (both clone class 77893 and 194 could be mapped onto 79832).

Lastly, the extended evolution mapping will also be an injective partial mapping defined on the clone classes of the system:

$$e : G \subset CC^{v_1} \to CC^{v_2}$$

5.2 A classification of clone evolution patterns

In the following, we will systematically summarise all of the possible evolution patterns and give a brief description of their meaning and consequences. First, let us suppose that $v_1, v_2, \ldots, v_n$ are consecutive versions of the same software system.

Disappearing clone class (DCC)
Definition: if $e\left(CC_j^{v_{i-1}}\right) = \emptyset$ for some $CC_j^{v_{i-1}} \in CC^{v_{i-1}}$ clone class (i.e. the evolution mapping was unable to find a correspondence with any clone class from version v_i), then $CC_j^{v_{i-1}}$ is said to be a *disappearing clone class*. The most likely reason is that the clone instances of the class had changed inconsistently, making the clone class disappear.

Appearing clone class (ACC)
Definition: if $e^{-1}\left(CC_j^{v_i}\right) = \emptyset$ for some $CC_j^{v_i} \in CC^{v_i}$ clone class (i.e. the clone class is not a map of any other clone class from version v_{i-1}), then $CC_j^{v_i}$ is said to be an *appearing clone class*. It is very probable that the developers created a new type of duplication.

The following smells apply to just those clone instances which are not members of clone classes that had already been reported as DCC or ACC smells. This means that from now on it may be assumed that both the $e^{-1}\left(CC_j^{v_i}\right) \neq \emptyset$ and $e\left(CC_j^{v_i}\right) \neq \emptyset$ relations hold for the classes of the particular instances.

Disappearing clone instance (DCI)
Definition: $C_k \in CI^{v_{i-1}}$ is a *disappearing clone instance* if $e\left(C_k\right) = \emptyset$, that is, C_k has not

been mapped to any instance of the subsequent version. It means that a clone instance had been modified, but at least two other copies still remained the same, perhaps because the developer forgot to change the other instances.

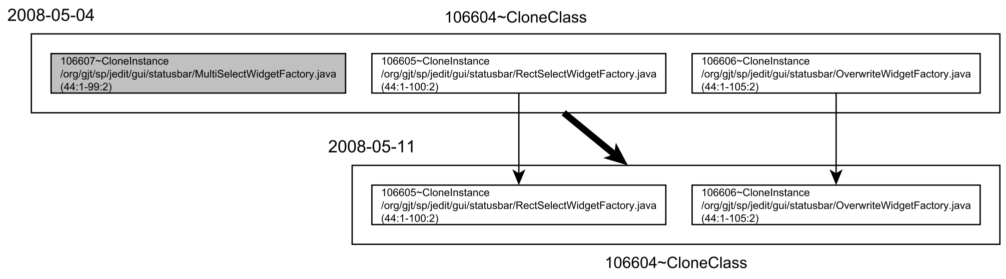

2008-05-04
106604~CloneClass
106607~CloneInstance /org/gjt/sp/jedit/gui/statusbar/MultiSelectWidgetFactory.java (44:1-99:2)
106605~CloneInstance /org/gjt/sp/jedit/gui/statusbar/RectSelectWidgetFactory.java (44:1-100:2)
106606~CloneInstance /org/gjt/sp/jedit/gui/statusbar/OverwriteWidgetFactory.java (44:1-105:2)
2008-05-11
106605~CloneInstance /org/gjt/sp/jedit/gui/statusbar/RectSelectWidgetFactory.java (44:1-100:2)
106606~CloneInstance /org/gjt/sp/jedit/gui/statusbar/OverwriteWidgetFactory.java (44:1-105:2)
106604~CloneClass

Figure 5.4: Example of a DCI smell in the *jEdit* system

Figure 5.4 gives an example of a DCI smell that was found in the version of 4^{th} May, 2008 in the *jEdit* system. Please note that the clone classes in the diagram are mapped onto each other. In this case, the clone instance shown in grey has disappeared as an execution branch, possibly due to a NullPointerException having been fixed. The same bugfix was applied for the other two instances just two weeks later.

Appearing clone instance (ACI)
Definition: $C_k \in CI^{v_i}$ is an *appearing clone instance* if $e^{-1}(C_k) = \emptyset$, that is, C_k is not a map of any instance from the previous version. It means that a new duplicate of an already copied code was created.

From this point on we will just consider those clone instances for which none of the above smells have yet been reported. Hence, we shall assume that the relations $e^{-1}(C_k^{v_i}) \neq \emptyset$ and $e(C_k^{v_i}) \neq \emptyset$ both hold.

Moving clone instance (MCI)
Definition: Let $C_k \in CI^{v_i} \cap CC_j^{v_i}$ be a clone instance in version v_i. C_k is said to be a *moving clone instance* if $e^{-1}(C_k) \notin e^{-1}\left(CC_j^{v_i}\right)$, i.e. C_k comes from a different clone class as the other instances. It usually means that the developers had modified the clone instance in such a way that it became a clone instance of another class, while the other instances remained in the same clone class.

These five smells cover all the basic cases when just two consecutive versions of the system are considered.

5.2.1 Empirical validation

We evaluated the usefulness of clone smells on two different open source systems; namely, the *Mozilla Firefox* Internet browser [81] written in C/C++ and *jEdit* text editor [47] written in the Java programming language. In the first case, the analysis and clone detection was performed on a daily basis with the help of the *Columbus framework* by taking 365 consecutive versions of the head revision from the year 2006. In the case of *jEdit* the clone detection was performed starting with the year 2008, but in this case weekly steps were taken between the versions. For both systems, the evolution mapping between the consecutive versions was computed using the weights obtained in the optimisation step (see Section 5.1.3).

The clone detection toolset used needs the code to be compilable, at least in the case of C/C++ (it is needed so as to resolve macro and include definitions provided in the command line during compilation). As in the case of the C/C++ system, it was quite common that the head revision would not compile properly (because of dependency or configuration-related issues), these versions were skipped as otherwise a large amount of false positive clone smells would have been reported. This filtering was performed just by neglecting the revisions of *Mozilla* which could not fully be compiled (some modules failed to compile). The only consequence is that the construction of the evolution mapping needs to be done over a longer period of time (not just one day), which may result in lower precision values. We treated this as an external threat to validity. In the case of *jEdit* this did not cause a problem, as the AST is built without compiling the code.

After applying this filtering process, 295 versions of the system remained (there was a case when the code could not be built properly for a whole month). Table 5.2 lists the ranges in which the basic metric values of the underlying software systems varied.

Metrics	Mozilla	jEdit
TLLOC(Total Logical Lines Of Code)	1,245,450 - 1,356,623	99,994 - 106,196
TNCL (Total Number Of Classes)	3,770 - 4,079	905 - 979
CI (Clone Instances)	4,407 - 6,840	349 - 376
CCL (Clone Classes)	1,784 - 2,641	125 - 135
CC (Clone Coverage)	6% - 6%	4.9% - 10%
Number of versions	295	84
Bijective mappings	97	51

Table 5.2: Basic metric values of Mozilla Firefox and jEdit

With *Mozilla*, in 97 out of the 295 revisions the evaluation mapping got between consecutive versions was identical; i.e. a bijective mapping between the clone instances could be created. In the case of *jEdit*, this number was 51 out of 84 versions.

The detected clone smells were manually evaluated in order to see if they were useful for finding suspicious evolution patterns. We also attempted to investigate the root causes of the reported smells. Analysing clone smells is much harder than checking the results of a simple clone detector tool as it requires considering the code, the change sets, the version control history, the versions in question, and some other aspects. To complicate matters, there are often hidden dependencies that may affect the code for which a particular smell is reported. For example, a change in the list of compiler macro definitions may cause new instances to appear or old ones to disappear. In other cases, modifying include definitions may also produce the same effect. Even after the cause is identified, one needs to be familiar with the system itself, its architecture and the semantics of changes, in order to reliably offer an opinion about the changes. Not being *Mozilla* or *jEdit* experts we performed the evaluation based on our experience in using C++ and Java, without knowing the exact purpose of the applied changes. We tried to answer the following questions:

- Are the pieces of code really duplications? If they are not (they are false positives of the clone detector), then it is not necessary to evaluate the particular smell.
- Are the reported code segments really smells of a particular type? If they are not, then there is probably an error somewhere in the evaluation mapping.
- What are the root causes of the particular smell? Here, we seek to learn what kind of change in the code led to the reporting of the smell. We do not wish to speculate whether the changes were really necessary, or whether they could have been performed differently.

Table 5.3 summarises the main results of our evaluation. As can be seen, 366 smells were found in 295 versions of *Mozilla* and 50 other smells in 84 versions of *jEdit*. The most frequent smells are the DCC and the ACC smells. The table also shows that around 80%-90% of the hits were real smells (relative to the precision of the clone detector). The 10%-20% of false positives is caused by the evolution mapping which either could not find corresponding instances (because there were too many changes in the code) or the instances were incorrectly mapped. The very low value of 38% for the clone detection precision, in the case of the ACI smell, was due to large array initialisations in the code that were erroneously identified as code duplications.

	Hit		Clones?		Smells?	
	Mozilla	*jEdit*	*Mozilla*	*jEdit*	*Mozilla*	*jEdit*
DCC	151	21	130 (86%)	19 (90%)	104 (80%)	17 (89%)
ACC	126	29	103 (82%)	24 (83%)	83 (81%)	22 (92%)
DCI	32	1	25 (78%)	1 (100%)	20 (80%)	1 (100%)
ACI	39	1	15 (38%)	0	12 (80%)	0
MCI	18	0	17 (94%)	0	13 (76%)	0
Overall	366	50	290 (79%)	44 (88%)	232 (80%)	40 (91%)

Table 5.3: Number of clone smells

Table 5.4 lists the clone smells found in *Mozilla*. The categories are defined based on the types of the causes that resulted in the reporting of the particular smell. The columns

	Cause	DCC	ACC	DCI	ACI	MCI	Σ
Consistent changes	**C1:** All instances deleted	26					**26**
	C2: All instances became too short	19					**19**
	C3: File deleted	5					**5**
	C4: Intentional refactoring	3					**3**
	C5: All instances have been newly created		51				**51**
	C6: Instances became sufficiently long		3				**3**
	Σ	**53**	**54**				**107**
Inconsistent changes	**C7:** Some instances of a class deleted	11		6			**17**
	C8: Inconsistent changes applied	38	21	14	7	13	**93**
	C9: Some instances added to a class	2	8		5		**15**
	Σ	**51**	**29**	**20**	**12**	**13**	**125**
Σ		**104**	**83**	**20**	**12**	**13**	**232**

Table 5.4: Root causes of clone smells found in *Mozilla*

	Cause	DCC	ACC	DCI	ACI	MCI	Σ
Consistent changes	**C1:** All instances deleted						**0**
	C2: All instances became too short						**0**
	C3: File deleted	1					**1**
	C4: Intentional refactoring	3					**3**
	C5: All instances have been newly created		9				**9**
	C6: Instances became sufficiently long		1				**1**
	Σ	**4**	**10**				**14**
Inconsistent changes	**C7:** Some instances of a class deleted	1					**1**
	C8: Inconsistent changes applied	12	8	1			**21**
	C9: Some instances added to a class		4				**4**
	Σ	**13**	**12**	**1**			**26**
Σ		**17**	**22**	**1**			**40**

Table 5.5: Root causes of clone smells found in *jEdit*

of the table refer to the clone smells, while the rows cite the reasons for their being reported.

The first main category is the category of *Consistent changes*, which includes 6 subcategories (C1-C6). For *Mozilla*, 107 out of 232 evaluated smells fell into the *Consistent changes* group. Even though the changes were consistent in this case (the changes affected all the instances of a class in the same way), it does not mean that the code had improved. In fact, only C1, C3 and C4 refer to changes which improved the maintainability of the system because they mean that all the duplications were eliminated. In the case of C2 the instances still exist, but they became too short, i.e. invisible to the clone detector (the minimal clone length was considered to be 10 lines of code). This might also be a problem, as future inconsistent changes can no longer be detected for these code segments. In the period surveyed, 46 new clone classes were created (C5) which might have a negative impact on maintainability. In three other cases the duplications already existed (C6), but they were too short and had just become visible to the clone detector. There were five cases reported where all the instances of a newly created clone class were located in the same newly created file (C6). Only 34 out of these 107 smells might imply an improvement in maintainability (C1, C3, C4); the other 73 cases suggest the possibility of a decreased code quality.

The other main category is the category of *Inconsistent changes*, where not all the instances of a clone class are affected in the same way by the changes. In this category all the enumerated cases are suspicious, as they might have occurred because the developers were unaware of the existence of duplications. Most of the smells are caused by inconsistent changes (C8), which suggests that the developers were unaware of them.

In the case of *jEdit*, (Table 5.5) the situation is similar in the sense that the majority of the smells are caused by inconsistent changes. It is very hard to compare the result obtained for the two systems because of the differences in programming languages, code size, domain, maturity, difference in period surveyed, and some other aspects. In spite of this, a few conspicuous things should be noted:

- In both cases over half (54% and 65%, respectively) of the reported smells refer to inconsistent code changes, which are at least suspicious.
- the root cause labelled C8 (Inconsistent changes applied) as the major cause of the smells in both cases.
- Among the inconsistent changes, DCC smell is more frequent than ACC. In this case ACC is reported when the consistency between the earlier existing duplications is restored. This suggests that there is a gap between the degree of restored consistency and the degree of newly introduced inconsistency in the code.

Although the main goal of smells is not that of detecting coding issues, inconsistently changing duplications may uncover unintentionally remaining serious coding problems as well. Two such examples are given below that were discovered during the manual evaluation of smells.

In the case of the *Mozilla* system, an ACI smell was reported on 3^{rd} March in 2006; a new clone instance appeared in the class which already had two instances:

- mozilla/content/svg/content/src/nsSVGLengthList.cpp (166:1-192:1) - rev: 1.10
- *mozilla/content/svg/content/src/nsSVGNumberList.cpp* (164:1-190:1) - rev: 1.7
- mozilla/content/svg/content/src/nsSVGPathSegList.cpp (164:1-190:1) - rev: 1.12

In this case, the second instance printed in italics was reported as a new one. The source codes of the first and second instances are shown in Figure 5.5. As can be seen, the two pieces of code are indeed duplications.

As the change log suggests, the changes made correspond to a bugfix: “Fixing bug 328439. Use EqualsLiteral and AppendLiteral where appropriate in svg/content...”. This change was applied to revision 1.6. By using the version control system of *Mozilla*, it transpired that exactly the same modifications had been applied to the other two instances on 16^{th} June in 2004, where the commit log was the following: "Bug 226439. Convert code base to use AppendLiteral/AssignLiteral/LowerCaseEqualsLiteral...". If code smells had been computed at that time, this change would have resulted in a DCI

```
NS_IMETHODIMP
nsSVGLengthList::GetValueString(nsAString& aValue)
{
  aValue.Truncate();

  PRInt32 count = mLengths.Count();

  if (count<=0) return NS_OK;

  PRInt32 i = 0;

  while (1) {
    nsISVGLength* length = ElementAt(i);
    nsCOMPtr<nsISVGValue> val = do_QueryInterface(length);
    NS_ASSERTION(val, "length doesn't
                           implement required interface");
    if (!val) continue;
    nsAutoString str;
    val->GetValueString(str);
    aValue.Append(str);

    if (++i >= count) break;

    aValue.AppendLiteral(" ");
   }

  return NS_OK;
}
```

mozilla/content/svg/content/src/nsSVGLengthList.cpp (rev.: 1.10)

```
NS_IMETHODIMP
nsSVGNumberList::GetValueString(nsAString& aValue)
{
  aValue.Truncate();

  PRInt32 count = mNumbers.Count();

  if (count<=0) return NS_OK;

  PRInt32 i = 0;

  while (1) {
    nsIDOMSVGNumber* number = ElementAt(i);
    nsCOMPtr<nsISVGValue> val = do_QueryInterface(number);
    NS_ASSERTION(val, "number doesn't
                           implement required interface");
    if (!val) continue;
    nsAutoString str;
    val->GetValueString(str);
    aValue.Append(str);

    if (++i >= count) break;

    aValue.AppendLiteral(" ");
  }

  return NS_OK;
}
```

mozilla/content/svg/content/src/nsSVGNumberList.cpp (rev.: 1.7)

Figure 5.5: Two pieces of code which were found to be clones on the 3^{rd} March in 2006.

smell (one of the three existing instances was neglected) and the bug would not have remained undetected for almost two years.

Another example was discovered in the *jEdit* system. Figure 5.6 illustrates the evolution of code duplications in this particular case. First, a bugfix eliminating the possibility of a NullPointerException was applied to the instance in the upper right corner. As there had been two other copies of it, which remained untouched, a DCI smell was reported. Two weeks later the same bugfix was applied to the other two instances, but in a syntactically different way which caused the class to disappear (DCC) and a new class appeared at the same time (ACC). Clone smells would have revealed the inconsistency immediately.

5.2.2 Conclusions

We introduced the notion of clone smells to describe the possible inconsistent clone instance evolution situations, which should be addressed by the developers to maintain or even improve the technical quality of the software system.

Our evaluation suggests that clone smells can be useful during software development because of the following observations:

- The approach presented here results in a comparatively short list of critical code segments which may comprise issues arising from inconsistent code changes. The list is significantly shorter than the list of existing duplications in the code, which allows the developers to manually evaluate them and investigate the causes.

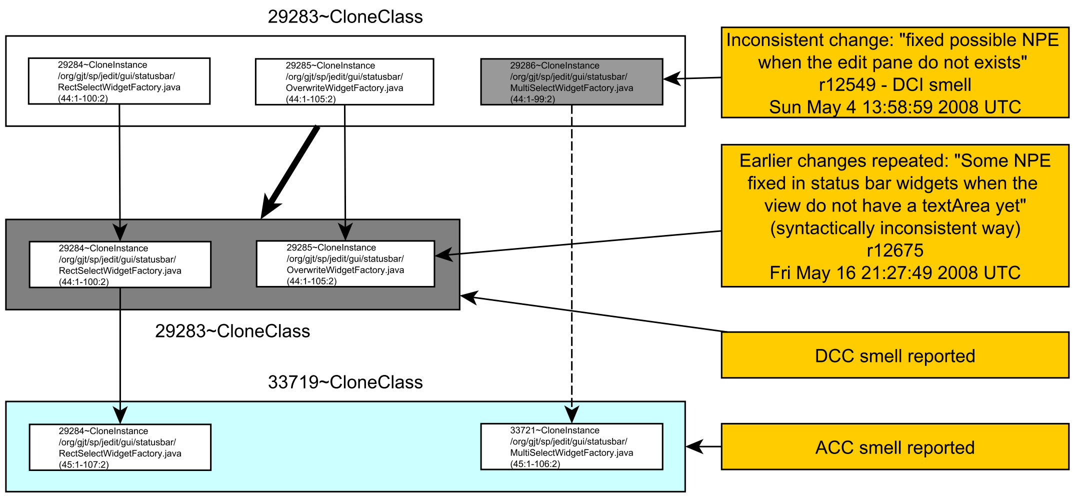

29283~CloneClass
29284~CloneInstance /org/gjt/sp/jedit/gui/statusbar/ RectSelectWidgetFactory.java (44:1-100:2)
29285~CloneInstance /org/gjt/sp/jedit/gui/statusbar/ OverwriteWidgetFactory.java (44:1-105:2)
29286~CloneInstance /org/gjt/sp/jedit/gui/statusbar/ MultiSelectWidgetFactory.java (44:1-99:2)
Inconsistent change: "fixed possible NPE when the edit pane do not exists" r12549 - DCI smell Sun May 4 13:58:59 2008 UTC
Earlier changes repeated: "Some NPE fixed in status bar widgets when the view do not have a textArea yet" (syntactically inconsistent way) r12675 Fri May 16 21:27:49 2008 UTC
29284~CloneInstance /org/gjt/sp/jedit/gui/statusbar/ RectSelectWidgetFactory.java (44:1-100:2)
29285~CloneInstance /org/gjt/sp/jedit/gui/statusbar/ OverwriteWidgetFactory.java (44:1-105:2)
29283~CloneClass
DCC smell reported
33719~CloneClass
29284~CloneInstance /org/gjt/sp/jedit/gui/statusbar/ RectSelectWidgetFactory.java (45:1-107:2)
33721~CloneInstance /org/gjt/sp/jedit/gui/statusbar/ MultiSelectWidgetFactory.java (45:1-106:2)
ACC smell reported

Figure 5.6: Inconsistent changes uncover the possibility of a remaining NullPointerException

- Over half of the reported smells were caused by inconsistent code changes; i.e. they were probably worth an additional manual inspection.
- Inconsistency is frequently introduced; consistency is rarely restored.
- Inconsistent changes can uncover unintentionally remaining coding problems in the code.

In the approach presented here, we did not use any information concerning the underlying version control system (code repository).

5.3 The connection between clones and coupling

Now we shall examine the effect of cloning on the coupling between software units; that is, on the degree of mutual dependencies of the units. Since we assume here object-oriented systems, the units are classes. We will analyse the different cloning situations in terms of changes in class dependencies and discuss the empirical data got from the source code analysis of several industrial systems, especially that concerning class coupling and clones. Somewhat surprisingly, this data suggests that there exists an inverse relationship between the amount of clones and class coupling. Since low coupling is conventionally regarded as favorable for maintenance, this finding appears slightly surprising at first sight.

5.3.1 Coupling

Coupling is a concept for measuring the level of interconnectedness and interrelatedness of software or source code components. High coupling between source code components is generally considered unfavorable from a maintenance aspect. The primary concerns for this view are the following:

- Tightly coupled components are harder to analyse and understand as their environment has to be considered as well.
- These components are harder to modify as their context should also be taken into account.
- Testing the changes in tightly coupled parts is harder as their entire environment needs to be simulated artificially during the testing stage.
- Changing tightly coupled components is more risky as the possibility of an inappropriate adjustment of their environment is higher.

There are lots of coupling metrics which are defined at different levels of abstraction, measure different types of dependencies, and so on. Probably the most widely used and accepted coupling metric for source code elements is the so-called *Coupling Between Object classes (CBO)*. Here, a class is said to be coupled to another one if it uses its member functions and/or instance variables. This measure tells us the number of classes to which a given class is coupled.

The CBO metric was found to be the best predictor of software bugs among the Chidamber&Kemerer metric suite [38].

5.3.2 The Bonus-Malus model

In Section 3.3, we proposed a model for quantifying high-level quality characteristics based on a benchmark and using probabilistic aggregation. The development of the approach was motivated by an earlier (in many ways primitive) model for obtaining comparative system level measures. The *Bonus-Malus model* operates with baselines got from the analysis of various independent software systems of different types and sizes. The baseline values for some of the most important metrics are based on past experience and on the analysis of the actual values in existing systems. A few of these baseline values are listed in Table 5.6.

Then, for the subject system, each metric value is compared to the corresponding baseline value and placed into one of the 8 predefined categories, which are computed as equal intervals with a step of 25% of the baseline. As with most metrics a bigger value means worse, we will use the following, so-called *Bonus-Malus classification* of a metric value M relative to the baseline value B:

Metric	*Baseline*
WMC	22.38
McCC	2.74
CBO	5.63
NOI	9.36

Table 5.6: Baseline values

B3:	$-\infty$ / 0	$\leq$	M	$\leq$	$0.25B$
B2:	$0.25B$	$<$	M	$\leq$	$0.5B$
B1:	$0.5B$	$<$	M	$\leq$	$0.75B$
B0:	$0.75B$	$<$	M	$\leq$	B
M0:	B	$<$	M	$\leq$	$1.25B$
M1:	$1.25B$	$<$	M	$\leq$	$1.5B$
M2:	$1.5B$	$<$	M	$\leq$	$1.75B$
M3:	$1.75B$	$<$	M	$\leq$	$+\infty$

After, the number of metric values that fall in each of the categories is counted, which will be used to assess whether the value of the specific metric type is typical, below or above the average. Furthermore, this rating is expressed as a single normalised value as well, which will be called the *baseline index*. It will provide an overall rating of the system in terms of the particular metric. This value falls in the interval $[-10\ldots10]$, and, in order to emphasise the more extreme values, it is a weighted sum of the number of elements of each category with the following weight values: B3(-10), B2(-7), B1(-3), B0(-1), M0(1), M1(3), M2(7), M3(10). Thus, a negative rating indicates a better-than-average system, while a positive value shows that a system is worse than the baseline.

By using a fixed set of baseline values, the approach described above allows us to assign a comparative *baseline index* for different systems, with any source code metric.

5.3.3 Hidden vs. visible dependencies

Visible dependencies are a straightforward way to recognise interconnections among software components. The previously described Coupling Between Object Classes is one of these visible dependencies because the related elements can be readily obtained, just by looking at a particular class. In software systems, there also exist dependencies which are not obvious and they are hard or impossible to compute. For example, if there are two different methods in the same system, which are supposed to compute the same function without invoking each other, then they are considered to be dependent on each other. Because of this requirement, neither of these functions should be modified without adjusting the other one. Yet it is computationally impossible to decide in general whether two methods always give the same result or not. These kinds of interrelations among different parts are called *hidden dependencies*. Another example of hidden dependencies are code duplications. Indeed, if any of the duplications needs to be changed, every other instance must be adjusted accordingly. Despite the fact that Type-1 and Type-2 duplications are relatively easy to compute, there is no efficient solution available for identifying Type-3 and Type-4 duplications.

While the visible dependencies are relatively easy to handle, the hidden ones can cause serious headaches and they represent serious stability risks. The hidden dependencies should be considered during the design phase; prudent design can significantly reduce the risk of these dependencies arising.

Not surprisingly, the visible and hidden dependencies are not independent of each other. In particular, the hidden dependencies can be eliminated by introducing visible ones. In the case of code duplications, for instance, duplications of one type can be extracted to one class only, and all the occurrences replaced by a reference, which is a visible dependency.

5.3.4 An empirical study

Within the context of software architecture evaluations in three Finnish machine industry companies, five subsystems were subjected to source code analysis, with the aim of pointing out potentially problematic parts of the system. The target systems were selected by the company representatives and we will refer to these systems as *A, B, C, D* and *E*. The sizes of these systems range from 22,000 to 341,000 LLOC (Logical Lines Of Code) written in C++ and C#. The tool applied to extract the source code metrics was the Columbus framework.

After computing source code metrics, the baseline indices were calculated in the way outlined in Section 5.3.2. Besides the CBO metric, we computed the baseline index of another coupling metric; namely, the *Number of Outgoing Invocations (NOI)*. Baselines for CBO and NOI were established and set to 4.79 and 7.94, respectively.

Table 5.7 lists the baseline indices for CBO and NOI metrics, and the system level *Clone Coverage (CC)* metric as well. The NOI-index for system A is missing as its value is not comparable with other NOI, owing to the different baseline index used in evaluating this system. We should add that in the case of baseline indices, lower values mean better values. But CC is a measure for the amount of cloning observed across the system. As can be seen from the table, the coupling metrics have an inverse relationship with the cloning metrics, meaning that improving the system from one aspect can result in decay in terms of the other. The correlation between CBO and CC is -0.76, while in the case of NOI and CC it is -0.97.

System	A	B	C	D	E
CBO-index	-8.85	-7.60	-6.15	-3.74	1.17
NOI-index		-7.97	-4.67	-2.56	1.39
CC	32.7	16.9	11.44	9.94	7.47

Table 5.7: Clone and coupling metrics for the systems analyzed

5.3.5 Conclusions

Here, we examined the relationship between cloning and coupling in software systems. We found that they are, in general, inversely proportional to each other, meaning that improving one can worsen the other. It follows that, contrary to some ideas about using just coupling metrics for measuring maintainability, every model used should take into consideration both aspects. At this point we refer to the quality model shown in Figure 3.4, which uses CBO and *Number of Incoming Invocations (NII)* as coupling metrics, and CC as well.

5.4 Summary

In this chapter we examined the connection between code duplications and maintainability from different perspectives. At present, the question of whether code duplications are actually harmful is still not completely clear. Our intention was not answer this question, but rather to provide new ideas, and theoretical and empirical inputs to the ongoing debate.

We looked at the connection between cloning and coupling in software systems and found that maintainability models should take both aspects into consideration. We said that the real threat regarding code duplications does not lie in their existence, but rather in connection with their evolution. Hence, we proposed an efficient algorithm for tracking duplications across subsequent versions of a software system. Based on the proposed algorithm for relating clone occurrences, we defined the concept of clone smells, which describe the different evolution patterns of duplications. We empirically evaluated the usefulness of clone smells, and we showed that they may serve as a basis for an efficient clone management technique that reduces risks and maintenance efforts arising from the duplications.

"Men will die upon dogma,
but will not fall victim to a conclusion."
John Henry Newman

Chapter 6

Conclusions

In this dissertation we proposed a method for deriving a measure for maintainability that in many ways differs from earlier approaches. Although our algorithm provides a mathematically consistent and clear way for assessing source code quality, it is not specific to source code metrics or any software system. The presented model integrates expert knowledge, handles ambiguity issues, manages "goodness" functions, which are continuous generalisations of threshold-based approaches. We found that the changes in the results of the model reflect the development activities; i.e. during development the quality decreases, while during maintenance the quality increases. We also found that although the goodness values computed by the model are different from the rankings provided by the developers, they still show relatively good correlations in the trends.

We also presented a formal mathematical model based on ordinary differential equations, which showed that with some reasonable assumptions, this relation may even be exponential. In other words, software maintainability has a great influence on development cost. The analysis of the empirical data shed light to the following points:

- The maintainability of an evolving software system decreases over time.
- Maintainability and development cost have an exponential relationship with each other.
- The new model is able to predict future development cost based on the estimated change rate of the code to good accuracy.

To evaluate their effect on maintainability, we proposed a method for tracking clones through the consecutive versions of an evolving software system. We defined a heuristic function called the evolution mapping between two particular code fragments taken from different versions of the same system. We proposed a similarity distance function to measure the likelihood of two code fragments having an evolutionary relationship with each other. The optimal mapping is then obtained as a solution to an optimisation problem. We extended the mapping to the level of clone classes and showed that the extension resulted in a well-defined mapping.

Based on the proposed algorithm for relating clone occurrences, we proposed a highly efficient and practical code duplication management method which can help reduce maintenance efforts and risk of inconsistent changes being made. The key concept lies in the notion of "clone smells", which represent different categories of suspicious clone evolution patterns. Clone smells can be used to identify those occurrences of duplications that could really cause problems in the future versions; i.e. the hazardous ones. The evaluation of clone smells suggests that they can be useful during software development cycles because of the following:

- The approach presented here results in a comparatively short list of critical code segments that may comprise issues arising from inconsistent code changes. The list is significantly shorter than the list of existing duplications in the code, which allows the developers to manually evaluate them and investigate the causes.
- More than half of the reported smells are caused by inconsistent code changes; i.e. they are probably worth further manual inspection.
- Inconsistency is frequently introduced, consistency is rarely restored; i.e. clones really represent a risk from a maintainability perspective.
- Inconsistent changes can reveal overlooked coding problems remaining in the code.

In the future, we would like to extend the maintainability model with process metrics (e.g. cost, time, effort) at low level, and with other ISO/IEC 9126 characteristics (e.g. reliability, usability) at a high level. We also plan to compare goodness functions at the highest level with the distribution of developers' rankings. The overall goal would be to build the kind of models where these two functions have a good fit meaning that the model expresses everything that is possible; i.e. the opinions of the experts with their usual ambiguity.

To facilitate clone tracking, we also intend to make use of the configuration management system. In this way, the other available pieces of process-related information about the source code should help improve the evolution mapping procedure. Furthermore, by utilising the bug tracking system, and by mapping the already fixed bugs to the precise locations in the source code, we should be able better to identify critical duplications where a bug-fix was applied just to some of the instances of a clone class. These would be precisely the code segments that have probably been overlooked and which might never have been noticed without monitoring clone instances throughout the versions of the system.

Appendix A

Related Work

A.1 Software maintainability models

The appearance of the widely accepted ISO/IEC 9126 and related standards [44] has stimulated the research in the area of quality models. Numerous papers, ranging from highly theoretical to more practical ones, examine this important area.

Some of the studies have focused on developing a methodology for adapting the ISO/IEC 9126 model in practice[17, 96, 25]. They provide guidelines or a framework for constructing effective quality models. In contrast, we focus more on presenting an algorithmic approach and a particular application of it for evaluating source code quality based on source code metrics.

Other papers tackle software quality from the end user's point of view. For example, Ozkaya et al.[89] stress the importance of using quality models like ISO/IEC 9126 in practice right from the outset of the design phase. Although their approach is sufficiently general for evaluating design or end user quality, here we just focus on source code (i.e., product) quality evaluation.

Jung et al.[53] conducted a survey and found the correlation among characteristics of the ISO/IEC 9126 standard. They showed that the strongly correlated characteristics relative to the users' opinions are different from the ones defined by the standard. We also conducted a survey to validate our approach, the difference being that here we used this information to show the usefulness of the model and not to evaluate any correlations among the characteristics of the standard.

Studies by Bansiya and Davis[13] and Muthanna et al.[83] focused on the software design phase. They adapted the ISO/IEC 9126 model to support the quality assessment of system design. In contrast, our model approach is based on low-level source code metrics not on design-level metrics and seeks to assess the product quality of an already existing system. Moreover, they use a simple linear combination to evaluate high-level characteristics, while we apply a probabilistic approach and use a large benchmark of systems as a reference instead of calculating absolute values directly. The direct use of metrics in their approach violates our Interpretable requirement, while using negative weights breaks the Explicable requirement of our model.

Kuipers and Visser introduced a maintainability model [64] as a replacement for the Maintainability Index by Oman and Hagemeister [88]. Based on this study, Heitlager et al.[43], who are members of the Software Improvement Group (SIG), proposed an extension of the ISO/IEC 9126 model that uses source code metrics at a low level. Similar to our study, their paper focuses on the *Maintainability* characteristic of the standard. Metric values are split into five categories, from poor (--) to excellent (++). The evaluation in their model involves summing the values for each attribute (having the values between -2 and +2) and then aggregating the values to get an index.

Correia and Visser[28] presented a benchmark that collects measurements for a wide variety of systems. This benchmark allows one to make a systematic comparison of the technical quality of (groups of) software products.

Alves et al.[3] presented a technique for deriving metric thresholds from benchmark data. This method is used to derive more reasonable thresholds for the SIG model as well. Since the threshold values need to be derived via a complex method for each new metric, this approach fails to satisfy the Extendible requirement of our model.

Correia and Visser[29] introduced a certification method that is based on the SIG quality model. The method allows one to certify the technical quality of software systems. Each system can get a rating from one to five stars (-- corresponds to one star, ++ to five stars). Baggen et al.[9] refined this certification process by performing a regular re-calibration of the thresholds based on the benchmark.

The original SIG model uses a binary relation between system properties and characteristics. Correia et al.[27] created a survey to elicit weights for their model. The survey was filled out by IT professionals, but the authors eventually concluded that using weights would not improve the quality model because of the lack of consensus among developers.

Our model differs from the SIG model in many ways. Firstly, our model is probabilistic, which naturally combines all the ambiguity issues arising from the lack of consensus, and the eventual result obtained is not a single value, but a probability distribution. Secondly, we use different source code-level metrics chosen by a set of experts working in the area of software quality assurance (see Section 3.3.2). Although we also created a benchmark of systems for assessing the quality, it is used differently. Instead of using the benchmark for obtaining thresholds, we implicitly compare the metric distributions of the subject system with the distributions of each system in the benchmark. The comparison results in a goodness function (see Section 3.3.1), which is used as a low-level input for the model. Thirdly, while the SIG model employs system properties expressed on a five-level scale, our model does not have any classification scale, so there is no loss of information. Another significant difference is that our model uses weights during the aggregation process. We also conducted a survey for eliciting the weights but we do not use the average (which may not improve the model[27]). Instead, we work with the whole distribution of the votes. Since source code quality is a subjective concept (and our model takes this subjectiveness into account), we believe that our approach is more expressive than methods which attempt to characterise the system quality in terms of a single value.

Luijten and Visser [73] showed that the metrics of the SIG quality model strongly correlate with the time needed to resolve an error in a software system. We also examined the relation between the metrics of our quality model and the different software development phases. Moreover, we applied a more direct approach for validation by comparing the developers' opinions with the results got from of the quality model applied on their software code.

A.2 Modelling the cost of software development

Modelling software development cost has been an intensive research area for quite a long time [1, 80, 56, 30, 19]. Effort estimation is important not just for software developers [85], but for system operators as well [20]. While summaries of research results achieved in the last thirty years are available [31, 52], many things remain to be discovered and clarified. [94]. Relevant comparison studies of different techniques [51] in various domains [21, 22, 68, 86, 14] also exist.

In analogy to the notion of entropy in thermodynamics, *software entropy* was first introduced by I. Jacobson et al. [45] to measure the disorder of a software system. The second law of thermodynamics in principle states that disorder in a closed system cannot be reduced; it can only remain unchanged or increase. This law also seems reasonable for software systems, since when as a system is modified, its disorder or entropy always increases. Based on this, Lehman said that software which is being used needs to be modified, and the changes result in an increase in complexity and decrease in quality [69].

Canfora et al. [23] applied the software entropy concept when examining changes in ArgoUML[1] and Eclipse.[2] First, the authors computed source code metrics, various design patterns and different process metrics. They approximated the cost by counting the number of contributors to file changes. They found that different types of changes may

[1]http://argouml.tigris.org/
[2]http://www.eclipse.org/

contribute either negatively or positively to the entropy. Namely, refactoring decreases entropy, while feature development usually increases it. They also showed that entropy tends to increase with the number of contributors to the file changes.

Bianchi et al. [16] investigated software entropy using bug information. They showed that constantly changing software systems are affected by degradation. The authors collected a dataset from individual groups of university students responsible for developing systems with a predefined functionality. The dataset consisted of bug reports from various development stages, which contained the number of found and slipped bugs as well as the amount of time spent on fixing them. They found that the more time was spent on bug correction, the harder it was to correct newly appeared defects. This study correlates with ours; although we used product metrics instead of process metrics. Moreover, they found that there was close correlation between the decrease in source code maintainability and the number of faults created in the analysis and design phases.

Hanssen et al. [40] examined software entropy in agile product development. An industrial study was conducted at a company with 60 developers. The code being developed was continuously monitored by a third party consultant using an automated toolchain. They found that code entropy strongly affected agile processes, and development tasks took longer when code complexity increased because of a higher entropy value. Moreover, in the short term iterations of the agile model, the resources needed to detect and resolve coding issues were insufficient. Instead of lengthening the iterations, the authors gave a viable solution to overcome code entropy. The authors stated that refactoring helps to overcome entropy problems in an agile environment. Our findings are similar, though we stated them generally for all kinds of development methodologies as the processes were not taken into account during our experiments. They suggested using continuous and automated code smell detection and refactoring to preserve maintainability through the iterations.

The ways of performing development effort estimation [18] range from experience [51] through benchmark approaches [95] to model-based approaches. Several methods that use a mixture of the above-mentioned techniques also exist [90, 59]. Many studies sought to compare the various approaches and draw pertinent conclusions.

In a recent study, Dubey at al. [33] defined a model based on object-oriented metrics for maintainability analysis. The authors emplyed the maintainability definition derived from the ISO/IEC 9126 standard [44]. They proposed high-level properties (e.g. fault proneness, defect density, etc.) of a software system that can affect maintainability. An extensive review of existing methodologies concerning high-level characteristics was carried out by the authors. They found that source code metrics are very useful for quantifying software maintainability based on the ISO/IEC 9126 standard. They created a non-benchmark based approach by using the Chidamber & Kemerer object-oriented metrics suite [26]. However, experimental validation of the model was performed. In our case, we used our previously published model [10] based on source code metrics to quantify maintainability. Our model utilises a benchmark database of source code measurement data of many systems taken from various domains.

Radlinski and Hoffman [91] compared 23 machine learning algorithms using local data as a benchmark for development effort prediction. The four datasets used for the experiments contained both process and product information, but lacked source code metrics (except for KLOC, which measured the lines of code). As the accuracy of the algorithms strongly depended on the set of predictors, therefore no universal machine learning algorithm was found by the authors. Instead, they found that running several algorithms using arbitrary predictors could serve as a good starting point for project managers on estimating development costs.

Several comprehensive studies summarise the effectiveness of various cost prediction methodologies.

Riaz et al. [92] examined 710 studies and selected 15 for an in-depth analysis. Their aim was to evaluate the ability of existing methods to measure maintainability. They provide a thorough overview of the interpretations of maintainability and summarise the most commonly used techniques for defining it. The authors analysed the measures used by these approaches and also the accuracy of the prediction results. A systematic review of research questions found in the papers was then carried out. The answers to these questions were used to grade the effectiveness of the studies based on a system of criteria. They found that there was little evidence on the effectiveness of maintainability prediction models.

Mair et al. [76] examined 171 papers concerning analogy-based and regression-based techniques for cost estimation. They proposed a broad selection of criteria for defining the effectiveness of the different approaches. They found that regression models performed poorly, while analogy-based methods were far better. In many cases, the two different methods provided conflicting results on the same dataset. Case-based reasoning using a benchmark database gave nice results in general, but exceptions were also found. Due to the lack of standardisation in software quality assurance, no universally applicable method was found by the authors for conducting software cost estimation.

A.3 Evolution of code duplications

There are a number of papers that deal with various clone detection approaches. Starting from the algorithms which are based on a lexical comparison of the source lines or tokens [34, 48, 54], through the metric-based approaches [60, 65, 66, 72] which use metric values of the code parts in order to identify similar fragments, to the more sophisticated AST-based approaches [15, 61, 97] that require a full syntactic analysis of the source code before clone detection can be performed. Here we used our implementation of one of the existing AST-based approaches presented by Koschke et al. [61].

Constructing algorithms to track clone instances across different versions of a software system has only recently become a hot area of research. Antoniol et al. [4], for example, applied time-series techniques to model the changes in the average number of clones per function in the system. When building a model like this, it is not necessary to map the clones from one version to the other; just the number of clones in each function needs to be computed. Metric values of functions were used to identify code duplications across the versions. The model was able to predict the amount of cloning in the system with an average error rate of 3.81%. Unlike their approach, we offer a lower level of correspondence between the cloned segments by way of comparison. Our approach does not use source code metrics to find the related code segments, but utilises the notion of a similarity distance function. Antoniol et al. [5] also studied the evolution of code clones in the Linux Kernel using their metric based approach. They found that the percentage of cloned code did not change during software evolution cycles and that, while new clones were added, some were factored out.

Merlo et al. [79] extended the concept of similarity of code fragments in order to quantify similarities at the release/system level. Similarities were captured by four software metrics representing the commonalities and differences within and among software artifacts. Doing this, they were able to model the changes in the similarity of the code between any two versions of the system without really performing clone detection.

Kim et al. [57] proposed a similar approach to ours. They defined the *cloning relationship* between two clone classes based on the lexical similarity of their representatives. In this way, a directed acyclic graph was obtained (the nodes are the clone classes and the edges are represented by the evolution pattern relationship). *Clone Genealogy* is a connected component of the above graph where every clone group is connected by at least one evolution pattern. Clone genealogy was used to perform a study on two small Java systems from the viewpoint of the cloning habits of the developers. According to the empirical study they performed, refactoring does not necessarily constitute an improvement and, in many cases, it is not worth doing as many clones tend to quickly disappear either because they evolve independently or because they are removed. There is a conceptual difference between their method and ours: their approach tackles ambiguity issues; i.e., it allows a clone instance to have more than one succeeding or preceding instance. The ambiguity arises from handling textual similarity and location overlapping attributes of the duplications differently. In our study we combine all the attributes into one global similarity distance function and provide an optimal mapping between the instances. Therefore, our approach eliminates this particular ambiguity; each clone instance is allowed to have at most one successor and one predecessor. Consequently, the identified evolution patterns (clone smells) significantly differ from the ones presented by Kim et al. Furthermore, they used evolutionary patterns to analyse the long term behaviour of duplications, while our intention was to detect suspicious evolution patterns in two subsequent versions only.

Similar to Kim et al., Aversano et al. [6] undertook a similar empirical study with a slightly refined framework, where they analysed the so-called co-changes, which are changes made by the same author, with the same notes, and within 200 seconds. They used a clone detector for the Java language that compared subtrees in the abstract syntax tree. The systems analysed were DNSJava and ArgoUML. They reported that the majority of clone classes had always been maintained consistently.

Duala-Ekoko et al. [32] proposed a technique for tracking clones in an evolving software system. They advanced the notion of an abstract Clone Region Descriptor (*CRD*), which describes the clone instances within methods in such a way that it is independent of the exact text or their location in the code. A CRD is a lightweight and abstract description of the location of a clone region (i.e. clone instance) in the source code. Given a CRD and a code base, they identify the corresponding clone region through a series of automatic searches. In this way, their *CloneTracker* system is able to keep track of clones even if the code evolves. The attributes used for constructing the CRD are similar to those applied by us for defining our similarity distance function. For example, a particular CRD contained information such as the file name, class name, signature of the method, anchor identifier of a function block, and also some additional metrical information about the clone region that was used just for conflict resolution purposes. The subsequent versions of the system were searched thoroughly for code pieces having the same CRD. If multiple candidates were found, a conflict resolution algorithm was applied based on the metrical values (e.g. code complexity) of the candidate regions. Compared to ours, their approach differs in a number of ways:

1. Although the set of features considered by us partially overlaps with the one proposed by Duala-Ekoko et al., we do not use these attributes to characterise one particular clone region, but rather to measure a similarity distance of clone region pairs. It follows that our approach might be able to relate code fragment pairs to each other where the proposed CRD differs, and therefore, cannot be related by *CloneTracker*.

2. In contrast to their technique, our approach is a probabilistic one, in the sense that a weight is assigned to each candidate pair. When computing the final mapping between the clone regions, all the pairs are considered at once, and the mapping with the lowest overall weight is taken. As a consequence, this approach is more flexible in the sense that it relates more clone fragment pairs, even with high structural and lexical differences. Still, our mapping produces more false positives, and the algorithm is harder to configure.

3. Our approach is resistant to changes in the nesting level of particular clone regions, which does not hold in the case of theirs.

4. They measured precision and accuracy of CRD by performing both the CRD extraction and lookup on the same particular version of a system. Owing to the nature of our algorithm, the same measurement technique would yield 100 % for the precision value (based on the definitions of the features and similarity used).

Geiger et al. [36] studied the relation between code clones and change couplings (files which are committed at the same time, by the same author, and with the same modification description). They proposed a framework for examining code clones and relating them to change couplings taken from a release history analysis. The results showed that, although the relationship is not statistically significant, the systems analysed have a fair amount of cases where the relationship holds.

Krinke [62] used a version control system of open source systems to identify changes applied to code duplications. They checked to see whether changes applied to cloned segments were generally consistent (repeated for all instances) or not. Their study revealed that clone classes were consistently changed in roughly half of the cases. Krinke also showed that classes that had been changed inconsistently earlier and were consistently changed later, were comparatively rare. This conclusion reinforces our view that identifying inconsistent changes, which may lead to unexpected behaviour in the future, is an important maintenance-related challenge Identifying inconsistent changes in software code was the chief goal in our study.

Göde et al. [37] presented an incremental clone detection algorithm that detects clones based on the results of the previous revision analysis. Their algorithm creates a mapping between clones of one revision to the next, supplying information about the addition and deletion of clones. The incremental approach considerably speeds up clone detection itself, and makes it possible to track the changes on-the-fly (even while the developer is typing). They utilised general suffix trees in their approach built on token tables representing tokens of individual files. When a file is added, deleted or modified, the corresponding token table is either added, subtracted or modified accordingly. During the update procedure, the clones from consecutive versions are also related. Their mapping procedure differs in several aspects from ours:

- They use information that comes from the version control system. We do not make use of this information.
- Clones moving between files cannot be tracked as each file handled has a different token table.
- Clones can effectively be tracked only if at most one of the tokens before the copied segment is modified. When the structure of a clone is altered or the clone has moved, the approach does not provide an effective solution for keeping track of them.

Göde et al. focus more on the gain in performance (they achieved a significant improvement) than on tracking the evolution of individual clone fragments.

Appendix B

Summary

B.1 Summary in English

Nowadays, whether we know it or not, software is part of our everyday lives. It doesn't just exist to make life easier, but our lives may sometimes even depend on it. The software industry has faced an enormous expansion recently, which in turn places a constant pressure on IT leaders to deliver the products as early as and as cheaply as possible. This "race" forces IT leaders and software engineers to sometimes make compromises and trade long-term quality for short-term benefits.

Quantifying source code maintainability is essential for making strategic decisions concerning the software system. Aggregating a measure for source code maintainability has long been a challenge in software engineering. We showed that the high-level maintainability characteristics could be modelled fairly well by using low-level source code metrics [42]. Unfortunately, it also turned out that the classical metric-based models trained on one system may not be readily usable on other systems [12]. We presented a novel method for deriving maintainability models that in many senses differs from the state-of-the-art research achievements and overcomes some of the existing problems. Our method handles the ambiguity issue that arises from the different interpretations of key notions and it produces models that express maintainability objectively. We compared the results of the model with the subjective opinions of those involved in its development. Although the experts' rankings differed from the values provided by the model in many cases, the high correlations indicated that the quality model expressed the same changes as the developers would expect.

The importance of maintainability is closely related to the cost of changing the behaviour of the software system. Grounded on reasonable assumptions, we presented a formal mathematical model based on ordinary differential equations for modelling the relation between source code maintainability and development cost. These assumptions were the following:

1. When making changes to a software system without explicitly seeking to improve it, its maintainability will decrease, or at the very least it will remain unchanged.
2. Performing changes in a software system with poorer maintainability is more expensive.

The evaluation of the empirical results shed light on the following findings:

1. The maintainability of an evolving software package generally decreases over time.
2. Maintainability and development cost have an exponential relationship with each other, with a high correlation.
3. The presented model is able to predict the future development cost based on the rate of change of the code, to good accuracy.

Code duplications are generally considered to be one of the chief enemies of software maintainability. Although the copy&paste approach can reduce software development time, the price in the long term will usually be paid in terms of increased maintainability costs. One of the primary concerns is that if the original code segment needs to be corrected, all the copied parts need to be checked and modified accordingly as well. By inadvertently neglecting to change the related duplications, the programmers may leave bugs in the code and introduce logical inconsistencies. The real threat does not lie in the existence of duplications, but rather the worries are related to their evolution. To facilitate the evaluation of code duplications in terms of maintainability, we proposed a method for tracking clones through the consecutive versions of an evolving software system. In our approach, the clones are extracted for all versions of the program and then they were retroactively mapped using a heuristic called evolution mapping. A similarity distance function is defined, for measuring the likelihood of two code fragments being in an evolutionary relationship with each other. We reduced the issue of obtaining an evolution mapping to an assignment problem, which could be efficiently solved by the Hungarian method.

The close connection between code duplications and development cost motivated us to seek for an efficient clone management method. A plethora of clone detection algorithms exist nowadays that can help reduce the risks of making inconsistent changes. Applying a conservative approach, after the clones are detected they can be evaluated and eliminated if necessary. Unfortunately, this approach cannot be applied in practice, since large software systems may have several thousands of duplications present in the software code. Moreover, most of these duplications are harmless, as they may not ever be modified in the future. We also propose a highly efficient and practical code duplication management method that can help reduce maintenance efforts and risks of inconsistent changes being made. The key concept lies in the notion of clone smells, which represent different categories of suspicious clone evolution patterns. Clone smells can be used to identify those occurrences of duplications that could really cause problems in the future versions, i.e. the hazardous ones. The list of risky places is several orders of magnitudes smaller than the list of all duplications in a system, so a manual evaluation and elimination is more straightforward to perform. Based on the empirical results, it became clear that clone smells can be useful because of the following:

- The approach results in a comparatively short list of critical code segments which may comprise issues arising from inconsistent code changes.
- Over half of the reported smells were caused by inconsistent code changes; i.e. they were probably worth an additional manual inspection.
- Inconsistency is frequently introduced; consistency is rarely restored.
- Inconsistent changes can uncover unintentionally remaining coding problems in the code.

The main contributions of the dissertation can be summarised as follows. First, we proposed a metric based probabilistic approach for modelling source code maintainability. Afterwards, we established a formal mathematical model for relating source code maintainability and development cost. Next, we presented a novel method for tracking code duplications through an evolving piece of software. Finally, we proposed a classification method of suspicious clone evolution patterns, which may serve as a basis of an efficient clone management technique that helps reducing risks and maintenance efforts arising from the duplications.

Whether clones are harmful or not is still an open question. Our intention was not to give the final answer to that question, but to add new aspects, theoretical and empirical results to the ongoing debate.

B.2. Summary in Hungarian

A szoftverek mára életünk szerves részévé váltak. Nem csupán kényelmesebbé teszik mindennapjainkat, hanem sokszor még az életünket is rájuk bízzuk. Az iparág robbanásszerű növekedése komoly kihívás elé állítja a szoftverfejlesztéssel foglalkozó cégeket és szakembereket egyaránt. A piaci nyomás arra kényszeríti az informatikai vezetőket, hogy egyre gyorsabban és olcsóbban állítsák elő a különböző szoftver termékeket. A rövid távú célok elérése érdekében kötött kompromisszumok óhatatlanul is a minőség rovására mennek hosszú távon.

Megalapozott stratégiai döntések meghozatala szempontjából elengedhetetlen a forráskód karbantarthatóságának számszerűsítése. A karbantarthatóság mérése a mai napig komoly kihívásnak számít a szoftverfejlesztés ipari és akadémiai berkeiben egyaránt. Megmutattuk, hogy az alacsony szintű jellemzők (forráskód-metrikák, kódolási szabálysértések, kódmásolatok) mérhetően befolyásolják egy szoftver forráskódjának karbantarthatóságát [42]. Kihívást jelent azonban a karbantarthatóság szubjektivitásának és az egyes modellek hordozhatóságának kezelése [12]. Az általunk kidolgozott forráskód-karbantarthatóság modell a jelenleg létező megközelítésekkel kapcsolatos problémák többségére megoldást nyújt. A kidolgozott eljárás megfelelően kezeli a fogalmak szubjektív értelmezéséből adódó problémákat, ugyanakkor az előálló modell objektív módon fejezi ki egy rendszer karbantarthatóságának mértékét. A validáció során összevetettük a modell által szolgáltatott mérőszámokat a fejlesztésben résztvevő szakértők véleményével. A modell által számított értékek ugyan nem esnek egybe a fejlesztők által becsülttel, azonban a kettő közötti korreláció mégis viszonylag magasnak mondható.

A karbantarthatóság jelentősége, a szoftver módosításának költségeivel összefüggésben mutatkozik meg. Munkánk során, kidolgoztunk egy közönséges differenciálegyenleteken alapuló, formális matematikai modellt, amely a fejlesztési költségek és a forráskód-karbantarthatóság között fennálló összefüggések leírására szolgál. Költségbecslő modellünk két egyszerű feltételezésen alapul:

1. A forráskódon végrehajtott bármilyen módosítás, amely nem kifejezetten annak javítását célozza (pl. funkcionalitás hozzáadása) nem növeli annak karbantarthatóságát.
2. Kevésbé karbantartható szoftverek esetén a módosítások végrehajtása költségesebb.

Az empirikus adatok elemzése során az alábbi következtetéseket vontuk le:

1. Általánosságban véve elmondható, hogy egy fejlesztés alatt álló szoftver karbantarthatósága az idő során csökken.
2. Egy rendszer karbantarthatósági mértéke és a fejlesztési költségek alakulása nagy korrelációt mutat a modell által becsült értékekkel, azaz egymással közel exponenciális kapcsolatban állnak.

3. A bemutatott modell nagy pontossággal képes előre jelezni a fejlesztések jövőbeli költségeit a kódváltozás mértéke alapján.

A kódmásolatokat általánosságban véve a forráskód karbantarthatóság fő ellenségének szokás tekinteni. Rövid távon a klónozás csökkenti ugyan a fejlesztés idejét, hosszú távon azonban a karbantartási költségek drasztikus növekedéséhez vezethet. Az egyik legsúlyosabb érv a klónozással szemben, hogy amennyiben valamely kódrészlet módosításra szorul, úgy valamennyi másolt szakasz ellenőrzésre és kiigazításra szorul. Amennyiben a másolt szakaszok módosítását elmulasztják, a működés során hibák és logikai inkonzisztenciák léphetnek fel. A valódi aggályok nem is a másolatok jelenlétéből adódnak, hanem a forráskód evolúciójával összefüggésben merülnek fel. Annak elősegítése érdekében, hogy a kódmásolatoknak a forráskód karbantarthatóságára gyakorolt hatásait elemezni tudjuk, kidolgozásra került egy eljárás, amely segítségével a kódmásolatok időben követhetővé válnak az evolúció során. Megközelítésünkben a kódmásolatok, a szoftver egymást követő verzióiban, egymástól függetlenül kerülnek azonosításra, majd az így talált klónok egy heurisztikus eljárás segítségével kerülnek megfeleltetésre. Meghatároztunk egy *hasonlósági távolság* értéket, amely annak a valószínűségét fejezi ki, hogy két kódrészlet közül az egyik a másik továbbfejlesztéseként jött létre. Ezen evolúciós megfeleltetés meghatározását visszavezettük egy hozzárendelési feladatra, amely a *Magyar módszer* segítségével hatékonyan oldható meg.

A fejlesztési költségek és a kódmásolatok között fennálló kézenfekvő kapcsolat arra késztetett bennünket, hogy egy hatékony kódmásolat-menedzsment eljárás után kutassunk. Manapság már számos megoldás létezik kódmásolatok azonosítására és a belőlük adódó kockázatok csökkentésére. A konzervatív megközelítés szerint, a kódmásolatok azonosítását kézi kiértékelés követi. Sajnálatos módon, a gyakorlatban ez a megközelítés nehezen alkalmazható, mivel egy nagyméretű szoftverben akár több ezer kódmásolat is létezhet. Továbbá a legtöbb klón ártalmatlan, mivel várhatóan a jövőben már nem fognak változni többet. A kódmásolatok követésére bemutatott eljárás alapján kidolgoztunk egy hatékony, gyakorlati kódmásolat-kezelést elősegítő módszert, amely a duplikációk inkonzisztens módosításaiból fakadó karbantartási költségeket és üzemeltetési kockázatokat csökkenti. Ennek érdekében bevezettük az ún. „clone smell"-ek fogalmát, amelyek a gyanús klón-evolúciós minták leírására szolgálnak. A „clone smell"-ek felhasználásával lehetőség nyílik a valóban veszélyesnek tekinthető másolatok azonosítására, és a klónok hatékony kezelésének megvalósítására. Az aggályos kódrészek listája nagyságrendekkel kevesebb elemet tartalmaz, mint amekkora egy rendszerben lévő másolatok száma, ezáltal azok kézi kiértékelése is elvégezhető. Az eredmények alapján világossá vált, hogy a „clone smell"-ek hasznosak a karbantarthatóság javítása szempontjából, mivel:

- A módszer egy viszonylag rövid, manuálisan ellenőrizhető listát eredményez azokról a kritikus forráskód részekről, amelyek inkonzisztens módosulásokkal kapcsolatos veszélyeket rejthetnek.
- A „clone smell"-ek több mint fele inkonzisztens módosulásból adódik, ezért érde-

mesek lehetnek további kézi elemzésre.

- Inkonzisztens módosulások gyakrabban előfordulnak, mint konzisztens változtatások.
- A módszer segítségével inkonzisztens módosulásokból adódó kódolási problémákra is fény derülhet.

A jelen tudományos értekezés az alábbiakkal járul hozzá a karbantarthatóság és kódmásolatok kutatási területéhez. Kifejlesztésre és bemutatásra került egy valószínűségszámítási módszereken alapuló, a forráskód karbantarthatóságának modellezését lehetővé tevő módszer. Felállítottunk egy formális matematikai elméletet a forráskód-karbantarthatóság és a fejlesztési költségek viszonyának modellezésére. Kidolgoztuk egy kódmásolat példányok időbeli követését megvalósító eljárást, amely a másolatok és a karbantarthatóság kapcsolatának vizsgálata szempontjából elengedhetetlen. Továbbá megalkottunk egy kódmásolat klasszifikációs eljárást, amely a másolatok evolúciójának gyanús mintáin alapul, és amely alapját képezheti egy hatékony kódmásolat-menedzsment módszernek.

A mai napig nyitott kérdés, hogy vajon a kódmásolatok valóban kártékonyak-e a karbantarthatóságra nézve. Jelen disszertációval a szándékunk nem ezen kérdés megválaszolása volt, hanem, hogy új és értékes nézőpontokkal, elméleti és gyakorlati eredményekkel járuljunk hozzá a kutatók között jelenleg is zajló vitához.

Bibliography

[1] A.J. Albrecht and Jr. Gaffney, J.E. Software function, source lines of code, and development effort prediction: A software science validation. *Software Engineering, IEEE Transactions on*, SE-9(6):639 – 648, Nov. 1983.

[2] David M Allen. Mean square error of prediction as a criterion for selecting variables. *Technometrics*, 13(3):469–475, 1971.

[3] Tiago L. Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[4] Giuliano Antoniol, Gerardo Casazza, M. Di Penta, and Ettore Merlo. Modeling clones evolution through time series. In *Proceedings of the 17th International Conference on Software Maintenance (ICSM 2001)*, pages 273–280. IEEE Computer Society, 2001.

[5] Giuliano Antoniol, Ettore Merlo, U. Villano, and M. Di Penta. Analyzing cloning evolution in the linux kernel. In *Information and Software Technology*, Volume 44, pages 755–765, 2002.

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

[7] Motoei Azuma. Software products evaluation system: quality models, metrics and processes - international standards and japanese practice. *Information and Software Technology*, 38(3):145 – 154, 1996.

[8] Robert Baggen, José Correia, Katrin Schill, and Joost Visser. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, 20:287–307, 2012. 10.1007/s11219-011-9144-9.

[9] Robert Baggen, Katrin Schill, and Joost Visser. Standardized Code Quality Benchmarking for Improving Software Maintainability. In *Proceedings of the Fourth International Workshop on Software Quality and Maintainability (SQM2010)*, 2010.

[10] T. Bakota, P. Hegedus, P. Kortvelyesi, R. Ferenc, and T. Gyimothy. A probabilistic software quality model. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 243 –252, Sept. 2011.

[11] Tibor Bakota, Rudolf Ferenc, and Tibor Gyimothy. Clone smells in software evolution. *Proceedings of the 23rd International Conference on Software Maintenance (ICSM 2007)*, pages 24–33, 2-5 Oct. 2007.

[12] Tibor Bakota, Rudolf Ferenc, Tibor Gyimothy, Claudio Riva, and Jianli Xu. Towards portable metrics-based models for software maintenance problems. *Software Maintenance, IEEE International Conference on*, 0:483–486, 2006.

[13] J. Bansiya and C.G. Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*, 28:4–17, 2002.

[14] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. In *IEEE Transactions on Software Engineering*, Volume 22, pages 751–761, October 1996.

[15] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–377, Washington, DC, USA, 1998. IEEE Computer Society.

[16] A. Bianchi, D. Caivano, F. Lanubile, and G. Visaggio. Evaluating software degradation through entropy. In *Software Metrics Symposium, 2001. METRICS 2001. Proceedings. Seventh International*, pages 210 –219, 2001.

[17] Jorgen Boegh, Stefano Depanfilis, Barbara Kitchenham, and Alberto Pasquini. A Method for Software Quality Planning, Control, and Evaluation. *IEEE Software*, 16:69–77, 1999.

[18] Barry Boehm, Chris Abts, and Sunita Chulani. Software development cost estimation approaches - a survey. *Annals of Software Engineering*, 10:177–205, 2000. 10.1023/A:1018991717352.

[19] Barry Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. Cost models for future software life cycle processes: Cocomo 2.0. *Annals of Software Engineering*, 1:57–94, 1995. 10.1007/BF02249046.

[20] Barry W. Boehm. Software engineering economics. *Software Engineering, IEEE Transactions on*, SE-10(1):4 –21, jan. 1984.

[21] Lionel C. Briand, Khaled El Emam, Dagmar Surmann, Isabella Wieczorek, and Katrina D. Maxwell. An assessment and comparison of common software cost estimation modeling techniques. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 313–322, New York, NY, USA, 1999. ACM.

[22] Lionel C. Briand, Tristen Langley, and Isabella Wieczorek. A replicated assessment and comparison of common software cost modeling techniques. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 377–386, New York, NY, USA, 2000. ACM.

[23] G. Canfora, L. Cerulo, M. Di Penta, and F. Pacilio. An exploratory study of factors influencing change entropy. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 134 –143, 30 2010-july 2 2010.

[24] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Identifying changed source code lines from version repositories. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 14, Washington, DC, USA, 2007. IEEE Computer Society.

[25] Juan Pablo Carvallo and Xavier Franch. Extending the ISO/IEC 9126-1 Quality Model with Non-technical Factors for COTS Components Selection. In *Proceedings of the 2006 international workshop on Software quality*, WoSQ '06, pages 9–14, New York, NY, USA, 2006. ACM.

[26] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.*, pages 476–493, June 1994.

[27] Jose Pedro Correia, Yiannis Kanellopoulos, and Joost Visser. A survey-based study of the mapping of system properties to iso/iec 9126 maintainability characteristics. *Software Maintenance, IEEE International Conference on*, 0:61–70, 2009.

[28] José Pedro Correia and Joost Visser. Benchmarking technical quality of software products. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, WCRE '08, pages 297–300, Washington, DC, USA, 2008. IEEE Computer Society.

[29] Jose Pedro Correia and Joost Visser. Certification of technical quality of software products. In *Proc. of the Int'l Workshop on Foundations and Techniques for Open Source Software Certification*, pages 35–51, 2008.

[30] AME Cuelenaere, MJIM van Genuchten, and FJ Heemstra. Calibrating a software cost estimation model: why and how. *Information and Software Technology*, 29(10):558 – 567, 1987.

[31] F. Deissenboeck, E. Juergens, K. Lochmann, and S. Wagner. Software quality models: Purposes, usage scenarios and requirements. In *Software Quality, 2009. WOSQ '09. ICSE Workshop on*, pages 9 –14, may 2009.

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

[33] Sanjay Kumar Dubey and Ajay Rana. Assessment of maintainability metrics for object-oriented software system. *SIGSOFT Softw. Eng. Notes*, 36(5):1–7, September 2011.

[34] Stephane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the 15th International Conference on Software Maintenance (ICSM'99)*, pages 109–118. IEEE Computer Society, 1999.

[35] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, pages 172–181. IEEE Computer Society, October 2002.

[36] Reto Geiger, Beat Fluri, Harald C. Gall, and Martin Pinzger. Relation of code clones and change couplings. In *Proceedings of the 9th International Conference of Funtamental Approaches to Software Engineering (FASE), Number 3922 in LNCS*, pages 411–425. Springer, 2006.

[37] Nils Göde and Rainer Koschke. Incremental clone detection. *Software Maintenance and Reengineering, European Conference on*, 0:219–228, 2009.

[38] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. *IEEE Transactions on Software Engineering*, pages 897–910, 2005.

[39] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, 2009.

[40] G.K. Hanssen, A.F. Yamashita, R. Conradi, and L. Moonen. Software entropy in agile product evolution. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pages 1 –10, Jan. 2010.

[41] R. Harrison, S.J. Counsell, and R.V. Nithi. An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, 24:491–496, 1998.

[42] Péter Hegedus, Tibor Bakota, László Illés, Gergely Ladányi, Rudolf Ferenc, and Tibor Gyimóthy. Source code metrics and maintainability: A case study. In Tai-hoon Kim, Hojjat Adeli, Haeng-kon Kim, Heau-jo Kang, KyungJung Kim, Akingbehin Kiumi, and Byeong-Ho Kang, editors, *Software Engineering, Business Continuity, and Education*, Volume 257 of *Communications in Computer and Information Science*, pages 272–284. Springer Berlin Heidelberg, 2011.

[43] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, QUATIC '07, pages 30–39, Washington, DC, USA, 2007. IEEE Computer Society.

[44] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[45] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[46] Judit Jász, Árpád Beszédes, Tibor Gyimóthy, and Václav Rajlich. Static Execute After/Before as a replacement of traditional software dependencies. In *ICSM*, pages 137–146, 2008.

[47] jEdit Homepage. `http://www.jedit.org`.

[48] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research (CASCON'93)*, pages 171–183. IBM Press, 1993.

[49] I.T. Jolliffe. *Principal Component Analysis*. Springer Verlag, 1986.

[50] C. Jones, O. Bonsignour, and J. Subramanyam. *The Economics of Software Quality*. Addison-Wesley, 2011.

[51] M. Jorgensen, B. Boehm, and S. Rifkin. Software development effort estimation: Formal models or expert judgment? *Software, IEEE*, 26(2):14 –19, March-April 2009.

[52] M. Jorgensen and M. Shepperd. A systematic review of software development cost estimation studies. *Software Engineering, IEEE Transactions on*, 33(1):33 –53, Jan. 2007.

[53] Ho-Won Jung, Seung-Gweon Kim, and Chang-Shin Chung. Measuring Software Product Quality: A Survey of ISO/IEC 9126. *IEEE Software*, pages 88–92, 2004.

[54] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[55] Cory Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society.

[56] Chris F Kemerer. An empirical validation of software cost estimation models. *Commun. ACM*, 30:416–429, May 1987.

[57] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. *SIGSOFT Software Engineering Notes*, 30(5):187–196, 2005.

[58] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[59] M. Klas, A. Trendowicz, Y. Ishigai, and H. Nakao. Handling estimation uncertainty with bootstrapping: Empirical evaluation in the context of hybrid prediction methods. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 245 –254, sept. 2011.

[60] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. In *Reverse engineering*, pages 77–108. Kluwer Academic Publishers, 1996.

[61] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06)*, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society.

[62] Jens Krinke. A study of consistent and inconsistent changes to code clones. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 170–178, Washington, DC, USA, 2007. IEEE Computer Society.

[63] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistic Quarterly*, 2:83–97, 1955.

[64] Tobias Kuipers and Joost Visser. Maintainability Index Revisited - position paper. In *System Quality and Maintainability, satellite of CSMR 2007*. IEEE Computer Society Press, 2007.

[65] Bruno Lague, Daniel Proulx, Jean Mayrand, Ettore M. Merlo, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of the 13th International Conference on Software Maintenance (ICSM'97)*, page 314, Washington, DC, USA, 1997. IEEE Computer Society.

[66] Filippo Lanubile and Teresa Mallardo. Finding function clones in web applications. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 379–388. IEEE Computer Society, 2003.

[67] Michelle Lee, A. Jeerson Outt, and Roger T. Alexander. Algorithmic analysis of the impacts of changes to object-oriented software. In *Proceedings of the International Conference on Software Maintenance*, pages 171–184. IEEE, 2000.

[68] Taeho Lee, Donoh Choi, and Jongmoon Baik. Empirical study on enhancing the accuracy of software cost estimation model for defense software development project applications. In *Advanced Communication Technology (ICACT), 2010. The 12th International Conference on*, Volume 2, pages 1117 –1122, feb. 2010.

[69] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[70] M M. Lehman, J F. Ramil, P D. Wernick, D E. Perry, and W M. Turski. Metrics and laws of software evolution - the nineties view. In *Proceedings of the 4th International Symposium on Software Metrics*, METRICS '97, pages 20–, Washington, DC, USA, 1997. IEEE Computer Society.

[71] Levenshtein distance.
`http://en.wikipedia.org/wiki/Levenshtein_distance`.

[72] Giuseppe A. Di Lucca, Massimiliano Di Penta, and Anna Rita Fasolino. An approach to identify duplicated web pages. In *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment (COMPSAC'02)*, pages 481–486. IEEE Computer Society, 2002.

[73] Bart Luijten and Joost Visser. Faster Defect Resolution with Higher Technical Quality Software. In *Proc. of the Fourth Int'l Workshop on System Quality and Maintainability*, pages 11–20. IEEE Computer Society Press, 2010.

[74] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[75] P. Mahalanobis. On tests and measures of group divergence i. theoretical formulae. *J. and Proc. Asiat. Soc. of Bengal*, 26:541–588, 1930.

[76] C. Mair and M. Shepperd. The consistency of empirical comparisons of regression and analogy-based software project cost prediction. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, page 10 pp., Nov. 2005.

[77] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 244, Washington, DC, USA, 1996. IEEE Computer Society.

[78] T. J. McCabe. A Complexity Measure. *IEEE Trans. Softw. Eng.*, 2:308–320, July 1976.

[79] Ettore Merlo, Michel Dagenais, P. Bachand, J. S. Sormani, S. Gradara, and Giuliano Antoniol. Investigating large software system evolution: The linux kernel. In *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment (COMPSAC'02)*, pages 421–426. IEEE Computer Society, 2002.

[80] Yukio Miyazaki and Kuniaki Mori. Cocomo evaluation and tailoring. In *Proceedings of the 8th international conference on Software engineering*, ICSE '85, pages 292–299, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[81] The Mozilla Firefox Homepage.
`http://www.firefox.com`.

[82] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, Vol. 5, No. 1:32–38, 1957.

[83] S. Muthanna, K. Ponnambalam, K. Kontogiannis, and B. Stacey. A maintainability model for industrial software systems using design level metrics. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, WCRE '00, pages 248–, Washington, DC, USA, 2000. IEEE Computer Society.

[84] N. Nagappan, T. Ball, and A. Zeller. Mining Metrics to Predict Component Failures. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, page to appear. IEEE Computer Society, May 2006.

[85] Ning Nan and D.E. Harter. Impact of budget and schedule pressure on software development cycle time and effort. *Software Engineering, IEEE Transactions on*, 35(5):624 –637, Sept.-Oct. 2009.

[86] P Nesi and T Querci. Effort estimation and prediction of object-oriented systems. *Journal of Systems and Software*, 42(1):89 – 102, 1998.

[87] Hector M. Olague, Letha H. Etzkorn, Sampson Gholston, and Stephen Quattlebaum. Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes. *IEEE Transactions on Software Engineering*, pages 402–419, 2007.

[88] P Oman and J Hagemeister. *Metrics for Assessing a Software System's Maintainability*, Volume 19, pages 337–344. IEEE Computer Society Press, 1992.

[89] Ipek Ozkaya, Len Bass, Raghvinder S. Sangwan, and Robert L. Nord. Making Practical Use of Quality Attribute Information. *IEEE Software*, 25:25–33, 2008.

[90] P.C. Pendharkar, G.H. Subramanian, and J.A. Rodger. A probabilistic model for predicting software development effort. *Software Engineering, IEEE Transactions on*, 31(7):615 – 624, July 2005.

[91] L. Radlinski and W. Hoffmann. On predicting software development effort using machine learning techniques and local data. In *International Jounral of Software Engineering and Computing*, Vol.2, No.2. International Science Press, 2010.

[92] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 367–377, Washington, DC, USA, 2009. IEEE Computer Society.

[93] D.H. Stork R.O. Duda, P.E. Hart. *Pattern Classification, 2nd Ed.* Wiley Interscience, 2000.

[94] M. Shepperd. Software project economics: a roadmap. In *Future of Software Engineering, 2007. FOSE '07*, pages 304 –315, May 2007.

[95] K. Srinivasan and D. Fisher. Machine learning approaches to estimating software development effort. *Software Engineering, IEEE Transactions on*, 21(2):126 –137, Feb 1995.

[96] Witold Suryn, Pierre Bourque, Alain Abran, and Claude Laporte. Software Product Quality Practices Quality Measurement and Evaluation Using TL9000 and ISO/IEC 9126. *Software Technology and Engineering Practice, International Workshop on*, pages 156–162, 2002.

[97] Wuu Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21(7):739–755, 1991.

[98] D Yudin and E G Gol'shtein. *Linear Programing Problems of Transportation (in Russian)*. NAUKA, 1969.

Corresponding publications of theses

[99] Péter Hegedus, Tibor Bakota, László Illés, Gergely Ladányi, Rudolf Ferenc, and Tibor Gyimóthy. Source code metrics and maintainability: A case study. In Tai-hoon Kim, Hojjat Adeli, Haeng-kon Kim, Heau-jo Kang, KyungJung Kim, Akingbehin Kiumi, and Byeong-Ho Kang, editors, *Software Engineering, Business Continuity, and Education*, Volume 257 of *Communications in Computer and Information Science*, pages 272–284. Springer Berlin Heidelberg, 2011.

[100] Tibor Bakota, Rudolf Ferenc, Tibor Gyimothy, Claudio Riva, and Jianli Xu. Towards portable metrics-based models for software maintenance problems. *Software Maintenance, IEEE International Conference on*, 0:483–486, 2006.

[101] T. Bakota, P. Hegedus, P. Kortvelyesi, R. Ferenc, and T. Gyimothy. A probabilistic software quality model. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 243 –252, Sept. 2011.

[102] T. Bakota, P. Hegedus, G. Ladányi, P. Kortvelyesi, R. Ferenc, and T. Gyimothy. A cost model based on software maintainability. In *28th IEEE International Conference on Software Maintenance (ICSM), 2012*, page to appear, Sept. 2012.

[103] Tibor Bakota. Tracking the evolution of code clones. In *Proceedings of the 37th international conference on Current trends in theory and practice of computer science*, SOFSEM'11, pages 86–98, Berlin, Heidelberg, 2011. Springer-Verlag.

[104] Tibor Bakota, Rudolf Ferenc, and Tibor Gyimothy. Clone smells in software evolution. *Proceedings of the 23rd International Conference on Software Maintenance (ICSM 2007)*, pages 24–33, 2-5 Oct. 2007.

[105] Marit Harsu, Tibor Bakota, Siket István, Kai Koskimies, and Systä Tarja. Code clones: Good, bad, or ugly? In *Proceedings of 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering*, 2009.

[106] Marit Harsu, Tibor Bakota, Siket István, Kai Koskimies, and Systä Tarja. Code clones: Good, bad, or ugly? In *Nordic Journal of Computing special issue dedicated to SPLST'09 and NW-MODE'09*, 2010.