Evaluating the effect of code duplications on software maintainability

Summary of the Ph.D. Thesis

of

Tibor Bakota

Supervisor:
Dr. Tibor Gyimóthy

Ph.D. School of Computer Science
Institute of Informatics
University of Szeged

Szeged
2012

Introduction

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.

The copy&paste technique is one of the most controversial methods for increasing productivity. 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. Therefore, code duplications are generally considered to be one of the chief enemies of software maintainability. 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 propose a method for tracking clones through the consecutive versions of an evolving software system.

Quantifying source code maintainability is essential for making strategic decisions concerning the software system. But aggregating a measure for maintainability has always been a problem 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 [11] provides a definition for maintainability, it does not provide a standard way of quantifying it. Many researchers exploited this vague definition and it has led to a number of practical quality models being proposed [10, 17, 4, 1]. Here, we present 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.

The importance of maintainability is closely related to the cost of changing the behaviour of the software system. Here, we present a formal mathematical model based on ordinary differential equations for modelling the relation between source code maintainability and development cost. It turns out that with some reasonable assumptions, the relation between cost and maintainability may even be exponential; i.e. software maintainability has a great influence on development cost.

Next, to evaluate the effect of duplications on maintainability, we propose 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.

The three main results in the dissertation are the following:

1. **Development of a probabilistic source code quality model.**

2. **Establishing a cost model based on source code maintainability.**

3. The assessment of code duplications from a code evolution viewpoint.

In the following sections we shall briefly present these results and state the contributions of the author at the end of each section.

1 A probabilistic source code quality model

Aggregating a measure for source code maintainability has long been a challenge in software engineering. Although we showed that the high-level maintainability characteristics could be modelled fairly well by using low-level source code metrics [9], many difficulties arise from the subjectiveness of the term itself. In the dissertation, we also proved that the classical metric-based models trained on one system may not be readily usable on other systems [3]. Now, we will attempt to overcome many of the issues of the existing methods to model source code maintainability by applying a new approach.

An approach for constructing probabilistic maintainability models

In our approach, the relation between quality attributes and characteristics at different levels are represented by an acyclic directed graph, called the *attribute dependency graph (ADG)*. The nodes at the lowest level (i.e. without incoming edges) are called *sensor node*s, while the others are called *aggregate node*s. Figure 2 shows an instance of an ADG.

The sensor nodes in our approach represent source code metrics that can be readily obtained from the source code. In the case of a software system, each source code metric can be regarded as a random variable that can take real values with particular probability values. For two different software systems, let $h_1(t)$ and $h_2(t)$ be the probability density functions corresponding to the same metric. Now, the *relative goodness value* (from the perspective of the particular metric) of one system with respect to the other, is defined as

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

where $\omega(t)$ is the weight function that determines the notion of goodness, i.e. where on the horizontal axis the differences matter more. Figure 1 helps us understand the meaning of the formula: it computes the signed area between the two functions weighted by the function $\omega(t)$.

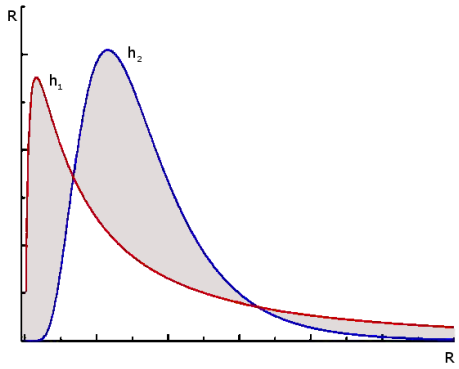

Figure 1: Comparison of probability density functions

For a fixed probability density function h, $\mathcal{D}(h, \_)$ is a random variable, which is independent of any other particular system. We will call it the *absolute goodness* of the system (from the perspective of the metric that corresponds to h). The empirical distribution of the absolute goodness can be approximated by substituting a number of samples for its second parameter, i.e. by making use of a repository of source code metrics of other software systems. The probability density function of the

absolute goodness is called the *goodness function*. The expected value of the absolute goodness will be called the *goodness value*. Following the path described above, the goodness functions for the sensor nodes can be easily computed.

For the edges of the ADG, a survey was prepared, where those who filled it in were asked to assign weights to the edges, based on how they felt about the importance of the dependency. They were asked to assign scalars to incoming edges of each aggregate node, such that the sum is equal to one. Consequently, a multi-dimensional random variable $\vec{Y}_v = \left(Y_v^1, Y_v^2, \ldots, Y_v^n\right)$ will correspond to each aggregate node v. We define the aggregated goodness function for the node v in the following way:

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

where $f_{\vec{Y}_v}(\vec{q})$ is the probability density function of $\vec{Y}_v$, $g_1, g_2, \ldots g_n$ are the goodness functions corresponding to the incoming nodes, 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 frightening at first glance, it is just a generalisation of how aggregation is performed in the classic approaches. Classically, a linear combination of goodness values and weights is taken, and it is assigned to the aggregate 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. Now, we are able to compute goodness functions for each aggregate node; in particular the goodness function corresponding to the *Maintainability* node as well.

A metric-based maintainability model for Java systems

For validation purposes, we constructed a particular attribute dependency graph for the Java programming language. For sensor nodes, we used the most commonly applied types of code properties, namely, source code metrics [7], coding rule violations and code duplications [5]. The final set of low-level properties was got by sampling several iterations with the helpful advice of academic and industrial experts in the field. In several other iterations, the low-level properties were linked to higher level attributes and weights for the edges were also assigned. Figure 2 shows the ADG obtained at the end of the iteration process. At the same time, a benchmark consisting of 100 open source and industrial software systems implemented in the Java programming language was created. The model obtained was used to validate our approach for quantifying source code maintainability.

Validation of the model on real-world systems

The maintainability model described above 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 (*System-1*). The other system is the *REM* persistence framework which is being developed at the University of Szeged. Our intention was to compare the results of the model with the subjective opinions of those involved in its development. The developers were asked to rank maintainability and its ISO/IEC 9126 subcharacteristics of the systems on a 0 to 10 scale. Table 1 lists the average values of the normalised ranks for each version of both software systems (the values in the brackets are the goodness values computed by the model).

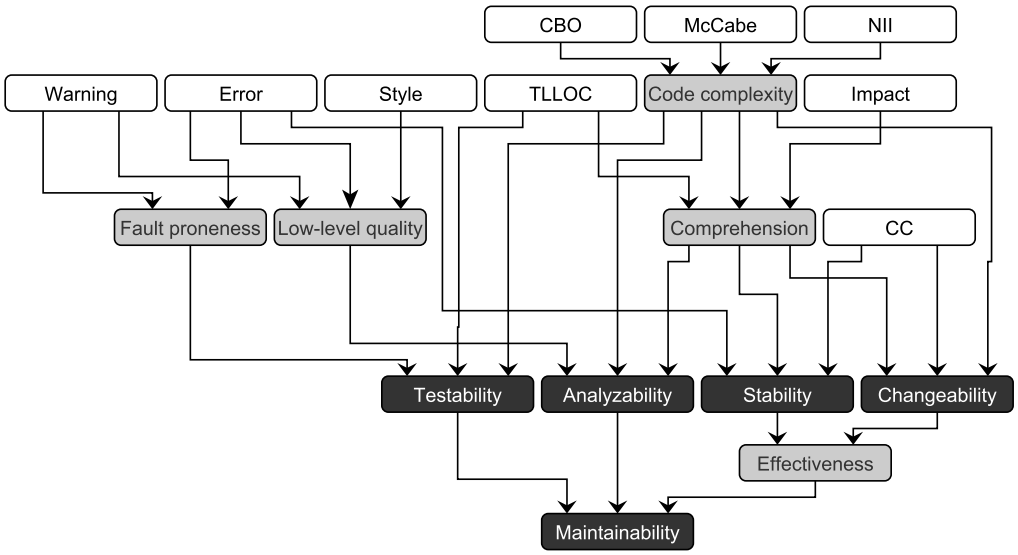

Figure 2: The *ADG* describing the relations among low-level properties (white), ISO/IEC 9126 subcharacteristics and maintainability characteristic (black) and high-level virtual properties (grey)

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 1: Averaged grades for maintainability and its ISO/IEC 9126 subcharacteristics based on the developers' opinions

The results show that the experts' rankings differ from the goodness values provided by the model in many cases. Actually, there are also large differences among the opinions of experts, depending on their experience, knowledge and degree of involvement. The bold lines in Table 1 show the Pearson's correlation of the rankings and the goodness values. The positive (and relatively high) correlations indicate that the quality model moderately expresses the same changes as the developers would expect.

Own contribution

The author's contribution 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 for the empirical validation stage. Proving the portability issues concerning the classical metric-based approaches mainly reflects the author's work as well.

2 A cost model based on source code maintainability

Currently, the best ways of conducting development effort estimation [6] range from a hands-on approach [13] through a benchmark approach [18] to a model-based approach. In contrast to other studies, we begin by stating simple and reasonable assumptions, and after establishing a formal mathematical representation, solutions are derived and validated on real-world systems. We will employ the maintainability model described earlier to compute source code maintainability.

A formal mathematical model for relating cost and maintainability

The proposed model is based on two simple assumptions:

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 first assumption can be formalised as follows:

$$\frac{d\mathcal{M}(t)}{dt} = -q\mathcal{S}(t)\lambda(t) \qquad (q \geq 0), \tag{1}$$

meaning that the rate of decrease in maintainability ($\mathcal{M}(t)$) is linearly proportional to the number of lines changed ($\mathcal{S}(t)\lambda(t)$) 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.

Formalising the second assumption leads to the following equation:

$$\frac{d\mathcal{C}(t)}{dt} = k\frac{\mathcal{S}(t)\lambda(t)}{\mathcal{M}(t)}. \tag{2}$$

The nominator represents the amount of change introduced at time t. The formula says that the utilisation of the cost invested at time t ($\frac{d\mathcal{C}(t)}{dt}$) for changing the code is inversely proportional to maintainability ($\mathcal{M}(t)$).

After solving the system of equations above, we arrive at one of our main results:

$$\mathcal{M}(t_1) = \mathcal{M}(t_0)\, e^{-\frac{q}{k}(\mathcal{C}(t_1)-\mathcal{C}(t_0))}, \tag{3}$$

which suggests that the maintainability of a system decreases exponentially with the cost invested in changing the system. While k and $\mathcal{C}(t)$ can be readily computed, quantifying the erosion factor, which characterises the "harm" caused by changing one line, is non-trivial. Of course, if there were an absolute measure of maintainability, the erosion factor q could easily be computed using Equation 3. Fortunately, we can make use of the model presented earlier, to quantify the absolute maintainability of a software system.

Provided that the model parameters are known, the estimated development cost for the future can be easily computed:

$$\mathcal{C}(t) = -\frac{k}{q}\ln\left|1 - \frac{q}{\mathcal{M}(0)}\int_0^t \mathcal{S}(s)\lambda(s)\,ds\right|. \tag{4}$$

Validation of the model on real-world systems

We performed the validation on a large number of consecutive versions of five different Java projects, based on the following steps:

1. We calculated the maintainability of each source code revision by using our probabilistic source code quality model [2]. We used this value as an approximation for $\mathcal{M}(t)$.

2. 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 exactly equal to $\mathcal{S}(t)\,\lambda(t)$.

3. We computed estimates for k and q from Equation 2 and Equation 3, respectively, at some time $T_0 > 0$. The estimates for k and q are the following:

$$k = \mathcal{C}(T_0)\left(1/\int_0^{T_0} \frac{\mathcal{S}(t)\,\lambda(t)}{\mathcal{M}(t)}dt\right), \tag{5}$$

and

$$q = -\frac{k}{\mathcal{C}(T_0)} \ln \frac{\mathcal{M}(T_0)}{\mathcal{M}(0)}. \tag{6}$$

4. 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.

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

1. The maintainability of an evolving software package generally decreases over time. This result is independent of the cost model presented here; it was inferred based on the maintainability values computed for each revision of a system.

2. Maintainability and development cost have an exponential relationship with each other, with a high correlation. We computed the model parameters for each system and found that the predicted values for $\mathcal{C}(t)$ and $\mathcal{M}(t)$ are close to the measured values, meaning that the model describes the real world fairly well. Figure 3 shows the change of maintainability in terms of cost for System-1.

3. The presented model is able to predict the future development cost based on the rate of change of the code, to good accuracy. With Equation 4, the future development cost can be computed, without needing to know the change of maintainability in advance. The predictions made by our model outperform the classical linear model, which does not take the change of maintainability into account. The differences are especially noticeable for long-term predictions. Actually, it is a natural phenomenon because changes of maintainability are more significant over longer periods of time. Figure 4 shows the prediction power of the model. Here, the x axis represents the length of the time interval for which the future predictions were stated.

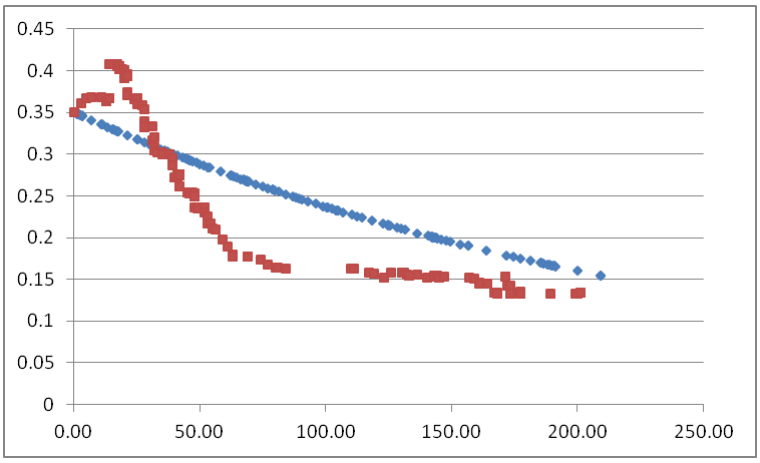

Figure 3: The change of maintainability ($\mathcal{M}(t)$) in terms of cost ($\mathcal{C}(t)$). The red line shows the results of empirical measurements, while the blue line shows the values predicted by the model.

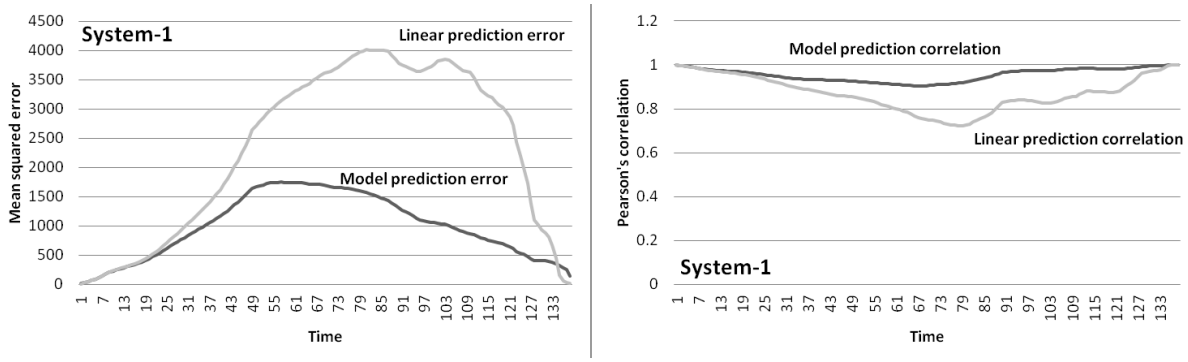

Figure 4: The prediction power of the model compared to the classical linear regression-based approach.

Own contribution

The author's contribution was the development of the formal mathematical background of the approach and also creating the methodology for the empirical validation stage.

3 The assessment of code duplications from a code evolution viewpoint

Being the chief enemies of software maintainability, code duplications play an important role in our maintainability model as well. Tracking clone instances across the successive versions of source code is essential in order to evaluate their influence on maintainability.

A novel approach for tracking the evolution of code clones

In our approach, the clones are extracted for all versions of the program and then they are retroactively mapped using a heuristic called the *evolution mapping*. Evolution mapping is a partial injective mapping of the clone instances of version v_1 to a version v_2 of the subject system:

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

For each possible clone instance pair of the mapping, a similarity distance value is computed which is aggregated from the following properties:

F_1 : The names of the files containing the clone instances.

F_2 : The order of the clone instances inside their classes.

F_3 : The unique names of the clone instances (where applicable).

F_4 : The unique names of their containing entities (e.g. function or class).

F_5 : Their relative positions inside their containing entities (e.g. inside a function or a class).

F_6 : The lexical structures of the clone instances.

In the case of the textual properties (F_1, F_3, F_4 and F_6), the *Levenshtein distance* [15] is computed, while for F_2 and F_5 a special measure is derived. Next, the overall similarity distance is defined by the following formula:

$$D\left(C_i, C_j\right) = \begin{cases} \sum_{k=1}^{6} \alpha_k D_k\left(C_i, C_j\right), & \text{if } \sum_{k=1}^{6} \alpha_k D_k\left(C_i, C_j\right) \leq \beta \\ \infty, & \text{otherwise} \end{cases},$$

where $\alpha_1, \ldots, \alpha_6$ and β are parameters that will be determined by an optimisation algorithm. If the parameters were known in advance, the issue of evolution mapping could be reduced to an assignment problem, which could be efficiently solved by the *Hungarian method* [14].

To determine the parameters of the model, we performed simulated annealing. For each fixed parameter assignment, we computed an overall similarity of the assignment induced by the evolution mapping. Table 2 shows the optimal parameter assignment that was found. It should be mentioned that weights express the importance of the corresponding feature, namely the amount of their contribution to the overall similarity distance function. As it turned out, the most important feature was F_2 (the ordinal number of appearance of the clone instances). This feature affects the decision by 28.9%, while textual similarity has a contribution of 23%.

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 2: Initial and optimised weights of the model

A classification of clone evolution patterns

Now, we propose the notion of *clone smells*, which represent different categories of suspicious clone evolution patterns. They seem to be appropriate candidates for further manual evaluations. The following five smells cover all the basic cases when only two consecutive versions of the system are considered:

1. Disappearing Clone Class (DCC) – the clone class existed in the previous version of the software system, but it does not exist in the current version.

2. Appearing Clone Class (ACC) – the clone class did not exist in the previous version of the software, but it exists now.

3. Disappearing Clone Instance (DCI) – the clone instance existed earlier, but it does not exist any more, although its clone class is still present.

4. Appearing Clone Instance (ACI) – a clone instance that did not exist earlier, although its class did.

5. Moving Clone Instance (MCI) – the clone instance has moved to another clone class.

We evaluated clone smells on the *Mozilla Firefox* [16] and on the *jEdit* [12] software systems by taking 295 and 84 successive versions, respectively. The clone smells detected were manually checked in order to see if they could have been used to find hazarduous duplications.

	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 3: 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 4: Root causes of clone smells found in jEdit

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: Clone and coupling metrics for the five systems analysed

Table 3 lists the clone smells found in Mozilla, while Table 4 shows the smells for the jEdit system. Based on these results, it became clear that clone smells can be useful 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.
- 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 connection between clones and coupling

Coupling is a concept for measuring the level of interconnectedness and interrelatedness among software or source code components. High coupling between source code components is generally considered unfavourable from a maintenance aspect [8]. Somewhat surprisingly, our work suggests that there exists an inverse relationship between the amount of clones and class coupling.

In our case study, five systems were subjected to source code analysis. For each system, we computed the amount of code duplication coverage (CC) and two coupling metrics, namely CBO (Coupling Between Object classes) and NOI (Number of Outgoing Invocations). The CC metric is defined at the system level, but in the case of CBO and NOI an aggregation was needed to get system level variants for each. We call the aggregated metrics CBO-index and NOI-index, respectively.

As can be seen from the table, the coupling metrics display an inverse relationship with the cloning metrics, meaning that improving the system from one aspect may be to the detriment of the other.

The correlation between CBO and CC is -0.76, while in the case of NOI and CC it is -0.97. 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 2, which uses CBO and *Number of Incoming Invocations (NII)* as coupling metrics, and CC as a clone metric.

Own contribution

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

- The development of the methodology for relating code duplications.
- An implementation of the required libraries and performing the optimisation of the weights.
- A 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.
- Extraction of code duplications to facilitate their evaluation in terms of coupling.

Conclusions

We proposed a method for deriving a measure for maintainability that in many ways differs from earlier approaches. The presented model integrates expert knowledge, handles ambiguity issues and deals with goodness functions. 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 good correlations in the trends.

Next, we presented a formal mathematical model based on ordinary differential equations that seeks to establish a relationship between development cost and the maintainability of the source code. An analysis of the empirical data shed light on the following points:

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

Later, we proposed a method for tracking clones through the consecutive versions of an evolving software system. We defined the so-called clone smells that 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.

Table 6 summarises which publications are related to which thesis points.

$\mathcal{N}o.$	[9]	[3]	[2]	[22]	[23]	[24]	[25]	[26]
1.	•	•	•				•	•
2.				•				
3.					•	•		

Table 6: The relation between the thesis topics and the corresponding publications

Acknowledgements

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

References

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

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

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

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

[17] 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.

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

Listed publications

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.