

How Much is a Clone?

Elmar Juergens, Florian Deissenboeck
Institut für Informatik, Technische Universität München, Germany
{juergens,deissenb}@in.tum.de

Abstract

Real-world software systems contain substantial amounts of cloned code. While the negative impact of cloning on software maintenance has been shown in principle, we currently cannot quantify it in terms of increased maintenance costs. However, as long as its economic impact cannot be quantified, control of cloning is probable to be neglected in practice. This paper presents an analytical cost model to estimate the maintenance effort increase caused by code cloning. The cost model can be used to assess the economic impact of cloning in a system and to evaluate investments in clone management tool support. To show its applicability, we report on a case study that instantiates the cost model for 11 industrial software systems.

1. Introduction

Code cloning abounds in real world software. Numerous studies report substantial amounts of cloning in open source and industrial systems [26, 32]. To name just a few, significant cloning was detected in GCC [14], X Windows [2], Linux and JDK [30]. Although the extent varies between systems, cloning occurs across programming languages, development contexts and application domains. Furthermore, it is not limited to source code. Recent studies have discovered substantial amounts of cloning in models [10] and requirements specifications [12, 15]. Cloning thus has to be considered as a phenomenon that occurs across different software artifacts.

Substantial research effort on software clones has established the negative impact of cloning on software maintenance activities in general [17]. It, often unnecessarily, increases code size and thus effort required for size-related activities such as inspections. Since changes to a piece of code, such as a bug fix, often need to be performed to its duplicates as well, cloning increases modification effort. If duplicates are missed when cloned code is modified, inconsistencies can be introduced into the system that can lead

to faults, or existing faults can fail to be removed from the system. A study we published in [22] uncovered over 100 faults in productive software through analysis of unintentionally inconsistent changes to cloned code.

While the negative consequences of cloning have firmly been established qualitatively, their *quantitative* impact remains unclear. We are currently unaware of how large the economic impact of cloning is w.r.t. the total maintenance effort. We thus cannot determine how harmful cloning is for a project in economic terms. Industrial software engineering is done with scarce resources—factors that cannot be quantified are probable to be neglected in favor of competing factors that can be quantified, such as open bug issues or change requests. As long as we do not know the costs cloning causes, clone control is prone to be neglected—even though cloning could be the root cause, and open bugs and change requests the symptoms.

Understanding of the costs caused by cloning is a necessary foundation to evaluate alternative clone management strategies. Do expected maintenance cost reductions justify the effort required for clone removal? How large are the potential savings that clone management tools can provide? We need a clone cost model to answer these questions.

Problem While the negative consequences of cloning are established qualitatively, the economic impact of cloning on maintenance is poorly understood. Consequently, we lack the foundation to assess the economic harmfulness of cloning and to evaluate alternative clone management strategies.

Contribution We propose an analytical cost model to estimate the impact that code cloning has on software maintenance efforts. We present a case study that instantiates the cost model for 11 industrial software systems and estimates maintenance effort increase and potential benefits achievable through clone management tool support. The paper presents work in progress—we point out shortcomings and directions of future research—but provides a step towards a more economically substantiated discussion of cloning.

2. Terms & Definitions

Clones are regions of similar code. In the literature, different definitions of similarity are employed [26,32], mostly based on syntactical characteristics. In this paper, we require clones to be semantically similar in the sense that they implement one or more common concepts. This redundant concept implementations gives rise to change coupling: when the concept changes, all of its implementations—the clones—need to be changed. In addition, we require clones to be syntactically similar. While syntactic similarity is not required for change coupling, existing clone detection approaches rely on syntactic similarity to detect clones. Hence, we employ the term *clone* to denote syntactically similar code regions that contain redundant implementation of one or more concepts. A *clone group* is a set of clones. Clones in a single group are referred to as *siblings*.

We use the term *failure* to denote an incorrect output of a software visible to the user. A *fault* is the cause of a potential failure in the source code.

Lines of code (LOC) denote the sum of the lines of code of all source files, including comments and blank lines. *Source statements* (SS) are the number of all source code statements, *not* taking commented or blank lines and code formatting into account. *Redundancy free source statements* (RFSS) are the number of source statements, if cloned source statements are only counted once. It thus estimates the size of a system from which all cloning is perfectly removed. For example, if a file contains 100 statements (and no clones) in version 1, and 50 of them are duplicated in the file to create version 2. SS increases to 150, but RFSS remains at 100. *Overhead* denotes the ratio by which a system's size has increased due to cloning. It is computed as $\frac{SS}{RFSS} - 1$. For the above example, the resulting overhead is 0.5, denoting a cloning induced size increase by 50%.

3. Maintenance Process

This section introduces the software maintenance process on which the cost model is based. It *qualitatively* describes the impact of cloning for each process activity and discusses potential benefits of clone management tools. The process is loosely based on the IEEE 1219 standard [18] that describes the activities carried on single change requests (CRs) in a waterfall fashion. The successive execution of activities that, in practice, are typically carried out in an interleaved and iterated manner, serves the clarity of the model but does limit its application to waterfall-style processes.

Analysis (A) studies the feasibility and scope of the change request to devise a preliminary plan for design, implementation and quality assurance. Most of it takes place

on the problem domain. Analysis is not impacted by code cloning, since code does not play a central part in it.¹

Location (L) determines a set of change start points. It thus performs a mapping from problem domain concepts affected by the CR to the solution domain. Location does not contain impact analysis, that is, consequences of modifications of the change start points are not analyzed. Location involves inspection of source code to determine change start points. We assume that the location effort is proportional to the amount of code that gets inspected.

Cloning increases the size of the code that needs to be inspected during location and thus affects location effort. We are not aware of tool support to alleviate the consequences of code cloning on location.

Design (D) uses the results of analysis and location as well as the software system and its documentation to design the modification of the system. We assume that design is not impacted by cloning. This is a conservative assumption, since for a heavily cloned system, design could attempt to avoid modifications of heavily cloned areas.

Impact Analysis (IA) uses the change start points from location to determine where changes in the code need to be made to implement the design. The change start points are typically not the only places where modifications need to be performed—changes to them often require adaptations in use sites. We assume that the effort required for impact analysis is proportional to the number of source locations that need to be determined.

If the concept that needs to be changed is implemented redundantly in multiple locations, all of them need to be changed. Cloning thus affects impact analysis, since the number of change points is increased by cloned code. Tool support (clone indication) simplifies impact analysis of changes to cloned code. Ideal tool support could reduce cloning effect on impact analysis to zero.

Implementation (Impl) realizes the designed change in the source code. We differentiate between two classes of changes to source code. *Additions* add new source code to the system without changing existing code. *Modifications* alter existing source code and are performed to the source locations determined by impact analysis. We assume that effort required for implementation is proportional to the amount of code that gets added or modified.

We assume that adding new code is unaffected by cloning in existing code. Implementation is still affected by

¹Possible effects of cloning in requirements specifications, which could in principle affect analysis, are beyond the scope of this paper.

cloning, since modifications to cloned code need to be performed multiple times. Linked editing tools could, ideally, reduce effects of cloning on implementation to zero.

Quality Assurance (QA) comprises all testing and inspection activities carried out to validate that the modification satisfies the change request. We assume a smart quality assurance strategy—only code affected by the change is processed. We do not limit the maintenance process to a specific quality assurance technique. However, we assume that quality assurance steps are systematically applied, *e. g.*, all changes are inspected or testing is performed until a certain test coverage is achieved on the affected system parts. Consequently, we assume that quality assurance effort is proportional to the amount of code on which quality assurance is performed.

We differentiate two effects of cloning on quality assurance: cloning increases the change size and thus the amount of modified code that needs to be quality assured. Second, just as modified code, added code can contain cloning. This also increases the amount of code that needs to be quality assured and hence the required effort. We are not aware of tool support that can substantially alleviate the consequences of cloning on quality assurance.

Other (O) comprises further activities, such as, *e. g.*, delivery and deployment, user support or change control board meetings. Since code does not play a central part in these activities, they are not affected by cloning.

4. Detailed Cost Model

This section introduces a detailed cost model that quantifies the impact of cloning on software maintenance. The model assumes each activity of the maintenance process to be completed. It is thus not suitable to model partial change request implementations that are aborted at some point.

4.1. General Approach

The *total* maintenance effort E is the sum of the efforts of individual change requests:

$$E = \sum_{cr \in CR} e(cr)$$

The scope of the cost model is determined by the population of the set CR : to compute the maintenance effort for a time span t , it is populated with all change requests that are realized in that period. Alternatively, if the total lifetime maintenance costs are to be computed, CR is populated with all change requests ever performed on the system. The model can thus scale to different project scopes.

The effort of a single change request $cr \in CR$ is expressed by $e(cr)$. It is the sum of the efforts of the individual activities performed during the realization of the cr . The activity efforts are denoted as e_X , where X identifies the activity. For brevity, we omit (cr) in the following:

$$e = e_A + e_L + e_D + e_{IA} + e_{Impl} + e_{QA} + e_O$$

In order to model the impact of cloning on maintenance efforts, we split e into two components: *inherent effort* e^i and *cloning induced overhead* e^c . e^i is independent of cloning. It captures the effort required to perform an activity on a hypothetical version of the software that does not contain cloning. e^c , in contrast, captures the effort penalty caused by cloning. Total effort is expressed as the sum of the two:

$$e = e^i + e^c$$

The increase in efforts due to cloning, Δe , is captured by $\frac{e^i + e^c}{e^i} - 1$, or simply $\frac{e^c}{e^i}$. The cost model thus expresses cloning induced overhead *relative* to the inherent effort required to realize a change request. The increase in total maintenance efforts due to cloning, ΔE , is proportional to the average effort increase per change request and thus captured by the same expression.

4.2. Activity Models

The activities *Analysis*, *Design*, and *Other* are not impacted by cloning. Their cloning induced effort, e^c , is thus zero. Their total efforts hence equal their inherent efforts.

Location effort depends on code size. Cloning increases code size. We assume that, on average, increase of the amount of code that needs to be inspected during location is proportional to the cloning induced size increase of the entire code base. Size increase is captured by *overhead*:

$$e_L^c = e_L^i \cdot \text{overhead}$$

Impact analysis effort depends on the number of change points that need to be determined. Cloning increases the number of change points. We assume that e_{IA}^c is proportional to the cloning-induced increase in the number of source locations. This increase is captured by *overhead*:

$$e_{IA}^c = e_{IA}^i \cdot \text{overhead}$$

Implementation effort comprises both addition and modification effort: $e_{Impl} = e_{Impl_{Mod}} + e_{Impl_{Add}}$. We assume that effort required for additions is unaffected by cloning in existing source code. We assume that the effort required for modification is proportional to the amount of code that gets modified, *i. e.*, the number of source locations determined by impact analysis. Its cloning induced overhead is, consequently, affected by the same increase as impact analysis: $e_{Impl}^c = e_{Impl_{Mod}}^i \cdot overhead$.

The modification ratio *mod* captures the modification-related part of the inherent implementation effort: $e_{Impl_{Mod}} = e_{Impl} \cdot mod$. Consequently, e_{Impl}^c is:

$$e_{Impl}^c = e_{Impl}^i \cdot mod \cdot overhead$$

Quality Assurance effort depends on the amount of code on which quality assurance gets performed. Both modifications and additions need to be quality assured. Since the measure *overhead* captures size increase of both additions and modifications, we do not need to differentiate between them, if we assume that cloning is, on average, similar in modified and added code. The increase in quality assurance effort is hence captured by the *overhead* measure:

$$e_{QA}^c = e_{QA}^i \cdot overhead$$

4.3. Maintenance Effort Increase Model

Based on the models for the individual activities, we model cloning induced maintenance effort e^c for a single change request like this:

$$e^c = overhead \cdot (e_L^i + e_{IA}^i + e_{Impl}^i \cdot mod + e_{QA}^i)$$

The relative cloning induced overhead is computed as follows:

$$\Delta e = \frac{overhead \cdot (e_L^i + e_{IA}^i + e_{Impl}^i \cdot mod + e_{QA}^i)}{e_A^i + e_L^i + e_D^i + e_{IA}^i + e_{Impl}^i + e_{QA}^i + e_O^i}$$

This model allows to compute the relative effort increase in maintenance costs caused by cloning. It does not take consequences of cloning on program correctness into account. This is done in the next section.

4.4. Fault Increase

Quality assurance is not perfect. Even if performed thoroughly, faults may remain unnoticed and cause failures in production.

Quality assurance can be decomposed into two sub-activities: fault detection and fault removal. We assume that, independent of the quality assurance technique, the effort required to detect a single fault in a system depends primarily on its fault density. We furthermore assume, that average fault removal effort for a system is independent of its size and fault density. These assumptions allow us to reason about the number of remaining faults in similar systems of different size but equal fault densities. If a QA procedure is applied with the same amount of available effort per unit of size, we expect a similar reduction in defect density, since the similar defect densities imply equal costs for fault location per unit. For these systems, the same number of faults can thus be detected and fixed per unit. For two systems A and B, with B having twice the size and available QA effort, we expect a similar reduction of fault density. However, since B is twice as big, the same fault density means twice the absolute number of remaining faults.

A system that contains cloning and its hypothetical version without cloning are such a pair of similar systems. We assume that fault density is similar between cloned code and non-cloned code—cloning duplicates both correct and faulty statements. Besides system size, cloning thus also increases the absolute number of faults contained in a system. If the amount of effort available for quality assurance is increased by *overhead* w.r.t. the system without cloning, the same reduction in fault density can be achieved. However, the *absolute* number of faults is still larger by *overhead*.

This reasoning assumes that developers are completely ignorant of cloning. That is, if a fault is fixed in one clone, it not immediately fixed in any of its siblings. Instead, faults in siblings are expected to be detected independently. Empirical data confirms that inconsistent bug fixes do frequently occur in practice [22]. However, it also confirms that clones are often maintained consistently. Both assuming entirely consistent or entirely inconsistent evolution is thus not realistic.

In practice, a certain amount of the defects that are detected in cloned code are hence fixed in some of the sibling clones. This reduces the cloning induced overhead in remaining fault counts. However, unless all faults in clones are fixed in all siblings, resulting fault counts remain higher than in systems without cloning. Therefore, cloning can have negative consequences on program correctness beyond activity effort increase captured by the detailed model.

4.5. Tool Support

Clone management tools can alleviate the consequences of cloning on maintenance efforts. We adapt the detailed model to quantify the impact of clone management tools. We evaluate the upper bound of what two different types of clone management tools can achieve.

Clone Indication makes cloning relationships in source code available to developers, for example through *clone bars* in the IDE that mark cloned code regions. Examples for clone indication tools include ConQAT and CloneTracker [13]. Optimal clone indication thus lowers the effort required for clone discovery to zero. It thus simplifies impact analysis, since no additional effort is required to locate affected clones. Assuming perfect clone indicators, e_{IA}^c is reduced to zero, yielding this cost model:

$$\Delta e = \frac{\text{overhead} \cdot (e_L^i + e_{Impl}^i \cdot \text{mod} + e_{QA}^i)}{e_A^i + e_L^i + e_D^i + e_{IA}^i + e_{Impl}^i + e_{QA}^i + e_O^i}$$

Linked Editing replicates edit operations performed on one clone to its siblings. Prototype linked editing tools include Codelink [35] and CReN [19]. Optimal linked editing tools thus lowers the overhead required for consistent modifications of cloned code to zero. Since linked editors typically also provide clone indication, they also simplify impact analysis. Their application yields the following model:

$$\Delta e = \frac{\text{overhead} \cdot (e_L^i + e_{QA}^i)}{e_A^i + e_L^i + e_D^i + e_{IA}^i + e_{Impl}^i + e_{QA}^i + e_O^i}$$

We do not think that clone management tools can substantially reduce the overhead cloning causes for quality assurance. If the amount of changed code is larger due to cloning, more code needs to be processed by quality assurance activities. We do not assume that inspections or test executions can be simplified substantially by the knowledge that some similarities reside in the code—faults might still lurk in the differences.

However, we are convinced that clone indication tools can substantially reduce the impact that cloning imposes on the number of faults that slip through quality assurance. If a single fault is found in cloned code, clone indicators can point to all the faults in the sibling clones, assisting in their prompt removal. We assume that perfect clone indication tools reduce the cloning induced overhead in faults after quality assurance to zero.

5. Simplified Cost Model

This section introduces a simplified cost model. While less generally applicable than the detailed model, it is easier to apply.

Due to its number of factors, the detailed model requires substantial effort to instantiate in practice—each of its nine factors needs to be determined. Except for *overhead*, all of them quantify maintenance effort distribution across individual activities. Since in practice the activities are typically interleaved, without clear transitions between them, is

difficult to get exact estimates on, *e. g.*, how much effort is spent on location and how much on impact analysis.

The individual factors of the detailed model are required to make trade-off decisions. We need to distinguish between, *e. g.*, impact analysis and location in order to evaluate the impact that clone indication tool support can provide, since impact analysis benefits from clone indication, whereas location does not. Before evaluating trade-offs between clone management alternatives however, a simpler decision needs to be taken: whether to do anything about cloning at all. Only then is it reasonable to invest the effort to determine accurate parameter values. If the cost model is not employed to assess clone management tool support, many of the distinctions between different factors are obsolete. We can thus aggregate them to reduce the number of factors and hence the effort involved in model instantiation.

Written slightly different, the detailed model is:

$$\Delta e = \text{overhead} \cdot \frac{e_L^i + e_{IA}^i + e_{Impl}^i \cdot \text{mod} + e_{QA}^i}{e}$$

The fraction is the ratio of effort required for code comprehension ($e_L^i + e_{IA}^i$), modification of existing code ($e_{Impl}^i \cdot \text{mod}$) and quality assurance (e_{QA}^i) w.r.t. the entire effort required for a change request. We introduce the new parameter *cloning-affected effort* (CAE) for it:

$$CAE = \frac{e_L^i + e_{IA}^i + e_{Impl}^i \cdot \text{mod} + e_{QA}^i}{e}$$

If *CAE* is determined as a whole (without its constituent parameters), this simplified model provides a simple way to evaluate the impact of cloning on maintenance efforts:

$$\Delta e = \text{overhead} \cdot CAE$$

6. Instantiation

This section describes how the cost model is instantiated and gives results from a large scale industrial case study.

We apply our quality analysis toolkit ConQAT² for clone detection and measure computation. ConQAT is described in general in [9, 11], its application as a clone detection workbench in [21].

6.1. Parameter Determination

This section describes how the parameter values can be determined to instantiate the cost model in practice.

²Available as open source at <http://www.conqat.org>

Overhead Computation *Overhead* is computed on the clones detected for a system. We have developed an algorithm that computes the *overhead* and implemented it in ConQAT. It is computed as explained in Section 2. *Overhead* captures cloning induced size increase independent of whether the clones can actually be removed with means of the programming language. This is intended—the negative impact of cloning on maintenance activities is independent of whether the clones can actually be removed.

The accuracy of the *overhead* value is determined by the accuracy of the clones on which it is computed. Unfortunately, many existing clone detection tools produce high false positive rates; Kapser and Godfrey [23] report between 27% and 65%, Tiarks et al. [34] up to 75% of false positives detected by state-of-the-art tools. False positives exhibit some level of syntactic similarity, but no common concept implementation and hence no coupling of their changes. They thus do not impede software maintenance and must be excluded from *overhead* computation.

In order to achieve accurate clone detection results, and thus an accurate *overhead* value, clone detection needs to be tailored. Tailoring removes code that is not maintained manually, such as generated or unused code, since it does not impede maintenance. Exclusion of generated code is important, since generators typically produce similar-looking files for which large amounts of clones are detected. Furthermore, tailoring adjusts detection so that false positives due to overly aggressive normalization are avoided. This is necessary so that, *e. g.*, regions of Java *getters*, that differ in their identifiers and have no conceptual relationship, are not erroneously considered as clones by a detector that ignores identifier names. According to our experience [22], after tailoring, clones exhibited change coupling, indicating their semantic relationship through redundant implementation of a common concept.

Determining activity efforts The distribution of the maintenance efforts depends on many factors, including the maintenance process employed, the maintenance environment, the personnel and the tools available [33]. To receive accurate results, the parameters for the relative efforts of the individual activities thus need to be determined for each software system individually.

Coarse effort distributions can be taken from project calculation, by matching engineer wages against maintenance process activities. This way, the relative analysis effort, *e. g.*, is estimated as the share of the wages of the analysts w.r.t. all wages. As we cannot expect engineer roles to match the activities of our maintenance process exactly, we need to refine the distribution. This can be done by observing development efforts for change requests to determine, *e. g.*, how much effort analysts spend on analysis, location and design, respectively. To be feasible, such

observations need to be carried out on representative samples of the engineers and of the change requests. Stratified sampling can be employed to improve representativeness of results—sampled CRs can be selected according to the change type distribution, so that representative amounts of perfective and other CRs are analyzed.

The parameter *CAE* for the simplified model is still simpler to determine. Effort *e* is the overall person time spent on a set of change requests. It can often be obtained from billing systems. Furthermore, we need to determine person hours spent on quality assurance, working with code and spent exclusively developing new code. This can, again, be done by observing developers working on CRs.

The modification ratio can, in principle, also be determined by observing developers and differentiating between additions and modifications. If available, it can alternatively be estimated from change request type statistics.

Literature values for Activity efforts offer a simple way to instantiate the model. Unfortunately, the research community still lacks a thorough understanding of how the activity costs are distributed across maintenance activities [33]. Consequently, results based on literature values are less accurate. They can however serve for a coarse approximation based on which a decision can be taken, whether effort for more accurate determination of the parameters is justified.

Several researchers have measured effort distribution across maintenance activities. Rombach et al. [31] report measurement results for three large systems, carried out over the course of three years and covering around 10,000 hours of maintenance effort. Basili et al. [6] analyzed 25 releases each of 10 different projects, covering over 20,000 hours of effort. Both studies work on data that was recorded during maintenance. Yeh and Jeng [36] performed a questionnaire-based survey in Taiwan. Their data is based on 97 valid responses received for 1000 questionnaires distributed across Taiwan’s software engineering landscape. The values of the three studies are depicted in Table 1.

Table 1. Effort distribution

Activity	[31]	[6]	[36]	Estimate
Analysis			26%	5%
Location		13%		8%
Design	30%	16%	19%	16%
Impact Analysis				5%
Implementation	22%	29%	26%	26%
Quality Assurance	22%	24%	17%	22%
Other	26%	18%	12%	18%

Since each study used a slightly different maintenance

process, each being different from the one used in this paper, we cannot directly determine average values for activity distribution. For example, in [31], *design* subsumes *analysis* and *location*. In [6], *analysis* subsumes *location*. The estimated average efforts are depicted in the fourth row of Table 1. Since the definitions of *implementation*, *quality assurance* and *other* are similar between the studies and our process, we used the median as estimated value. For the remaining activities, the effort distributions from the literature are of little help, since the activities do not exist in their processes or are defined differently. We thus distributed the remaining 34% of effort according to our best knowledge, based on our own development experience and that of our industrial partners.—the distribution can thus be inaccurate.

To determine the ratio between modification and addition effort during implementation, we inspect the distribution of change request types. We assume that adaptive, corrective and preventive change requests mainly involve modifications, whereas perfective changes mainly involve additions. Consequently, we estimate the ratio between addition and modification by the ratio of perfective w.r.t. all other change types. Table 2 shows effort distribution across change types from the above studies. The fourth row depicts the median of all three—37% of maintenance efforts are spent on perfective CRs, the remaining 63% are distributed across the other CR types. Based on these values, we estimate the modification ratio to be 0.63.

Table 2. Change type distribution

Effort	[31]	[6]	[36]	Median
Adaptive	7%	5%	8%	7%
Corrective	27%	14%	23%	23%
Other	29%	20%	44%	29%
Perfective	37%	61%	25%	37%

6.2. Case Studies

This section presents the application of the clone cost model to several large industrial software systems to quantify the impact of cloning, and the possible benefit of clone management tool support, in practice.

Goal The case study has two goals. First, evaluation of the clone cost model. Second, quantification of the impact of cloning on software maintenance costs across different software systems, and the possible benefit of the application of clone management tools.

Study Objects We chose 11 industrial software systems as study objects. Since we require the willingness of de-

velopers to contribute in clone detection tailoring, we had to rely on our contacts with industry. However, we chose systems from different domains (finance, content management, convenience, power supply, insurance) from 7 different companies written in 5 different programming languages to capture a representative set of systems. For non-disclosure reasons, we termed the systems A-K. Table 3 gives an overview ordered by system size.

Study Design and Procedure Clone detection tailoring was performed to achieve accurate results. System developers participated in tailoring to identify false positives. Clone detection and *overhead* computation was performed using ConQAT for all study objects. Minimal clone length was set to 10 statements for all systems. We consider this a conservative minimal clone length.

Since the effort parameters are not available to us for the analyzed systems, we employed values from the literature. We assume that 50% (8% location, 5% impact analysis, 26% · 0,63 implementation and 22% quality assurance; rounded from 51,38% to 50% since the available data does not contain the implied accuracy.) of the overall maintenance effort are affected by cloning. To estimate the impact of clone indication tool support, we assume that 10% of that effort are used for impact analysis (5% out of 50% in total). In case clone indication tools are employed, the impact of cloning on maintenance effort can thus be reduced by 10%.

Results and Discussion The results are depicted in Table 3. The columns show lines of code (kLOC), source statements (kSS), redundancy-free source statements (kRFSS), size overhead and cloning induced increase in maintenance effort without (ΔE) and with clone indication tool support (ΔE_{Tool}). Such tool support also reduces the increase in the number of faults due to cloning. As mentioned in Section 4.4, this is not reflected in the model.

The effort increase varies substantially between systems. The estimated overhead ranges from 75%, for system A, to 5.2% for system F. We could not find a significant correlation between overhead and system size. On average, estimated maintenance effort increase is 20% for the analyzed systems. The median is 15.9%. For a single quality characteristic, we consider this a substantial impact on maintenance effort. For systems A, B, E, G, I, J and K estimated effort increase is above 10%; for these systems, it appears warranted to determine project specific effort parameters to achieve accurate results and perform clone management to reduce effort increase.

7. Discussion

This section discusses challenges of cost modeling in general and of this cost model in particular.

Table 3. Case study results

System	Language	kLOC	kSS	kRFSS	overhead	ΔE	ΔE_{Tool}
A	XSLT	31	15	6	150.0%	75.0%	67.5%
B	ABAP	51	21	15	40.0%	20.0%	18.0%
C	C#	154	41	35	17.1%	8.6%	7.7%
D	C#	326	108	95	13.7%	6.8%	6.2%
E	C#	360	73	59	23.7%	11.9%	10.7%
F	C#	423	96	87	10.3%	5.2%	4.7%
G	ABAP	461	208	155	34.2%	17.1%	15.4%
H	C#	657	242	210	15.2%	7.6%	6.9%
I	Cobol	1,005	400	224	78.6%	39.3%	35.4%
J	Java	1,347	368	265	38.9%	19.4%	17.5%
K	Java	2,179	733	556	31.8%	15.9%	14.3%

7.1. Challenges of project diversity

Many factors influence maintenance productivity [5, 7, 33]: the type of system and domain, development process, available tools and experience of developers, to name just a few. Since these factors vary substantially between projects, they need to be reflected by cost estimation approaches to achieve accurate absolute results. The more factors a cost model comprises, the more effort is required for both its creation and its associated factor lookup tables, and for its instantiation in practice. If an absolute value is required, such effort is unavoidable.

The assessment of the impact of cloning differs from the general cost estimation problem in two important aspects. First, we compare efforts for two systems—the actual one and the hypothetical one without cloning—for which most factors are identical, since our maintenance environment does not change. Second, relative effort increase w.r.t. the cloning-free system is sufficient to evaluate the impact of cloning. Since we do not need an *absolute* result value in terms of costs, and since most factors influencing maintenance productivity remain constant in both settings, they do not need to be contained in our cost model. In a nutshell, we deliberately chose a *relative* cost model to keep its number of parameters and involved instantiation effort at bay.

7.2. Assumptions

The cost model is based on a series of assumptions. It can only sensibly be applied for projects that satisfy them. We list and discuss them here to simplify their evaluation.

We assume that the significant part of the cost models for the maintenance process activities are linear functions on the size of the code that gets processed. For example, we assume that location effort is primarily determined by and proportional to the amount of code that gets inspected during location. In some situations, activity cost models might

be more complicated. For example, if an activity has a high fixed setup cost, the cost model should include a fixed factor; diseconomy of scale could increase effort w.r.t. size in a super linear fashion. In such cases, the respective part of the cost model needs to be adapted appropriately. CO-COMO II, *e. g.*, uses an exponential function to adapt size to diseconomy of scale.

We assume that changes to clones are coupled to a substantial degree. This is in accordance with our experiences from large scale application of clone detection in industrial contexts [11, 21, 22], if clone detection is tailored appropriately. In case clones are uncoupled, *e. g.*, because they are false positives or because parts of the system are no longer maintained, the model is not applicable.

We assume that each modification to a clone in a clone group requires the same amount of effort. We ignore that subsequent implementations of a single change to multiple clone instances could get cheaper, since the developer gets used to that particular clone group. In practice, most clone groups have size 2. The inaccuracy introduced by this simplification should thus be moderate.

8. Related Work

This section relates the proposed cost model to existing work on the consequences of code cloning and cost models.

8.1. Consequences of Cloning

Substantial research has been carried out to better understand the consequences of code cloning. A survey is given by Hordijk et al. in [17]:

Impact of cloning on program correctness is the subject of several studies. Li et al. [30] present an approach to detect bugs based on inconsistent renaming of identifiers between clones. Jiang, Su and Chiu [20] analyze different

contexts of clones, such as missing *if* statements. Both papers report the successful discovery of bugs in released software. In [1] and [3], individual cases of bugs or inconsistent bug fixes discovered by analysis of clone evolution are reported for open source software. In earlier work [22], we inspected inconsistent clones together with the developers of the analyzed systems. In four open source and four industrial systems, analysis revealed 107 faults due to unintentionally inconsistent changes to cloned code. The above studies give strong indication that that cloning impacts correctness in practice and that developers cannot be assumed to have complete knowledge of cloning when fixing bugs.

Several researchers have investigated the impact of cloning on modification effort. In [25], Kim et al. report that a substantial amount of changes to code clones occur in a coupled fashion, indicating additional maintenance effort due to multiple change locations. Aversano et al. [1] confirm a high ratio of co-evolution in a separate study. Detected late propagations indicate that a substantial number of inconsistent changes to clones were later detected and corrected during quality assurance. These studies give strong indication that a substantial amount of the modifications to cloned code are coupled in many systems in practice, and thus cause additional effort for location and consistent modification.

The above work establishes qualitative relationships between cloning and maintenance efforts that provide the foundation of our cost model. It does, however, not allow to *quantify* impact of cloning on development efforts. In contrast, the cost model presented here allows to estimate impact of cloning on development costs.

Several empirical studies on the evolution of clones are not conclusive w.r.t. the negative impact of cloning on maintenance activities. Krinke reports that a substantial amount of clones evolve independently [27] and that stability of cloned code was found to be, contrary to expectations, similar or even higher than stability of non-cloned code [28]. Göde shows that clone evolution patterns vary between different systems [16]. While these studies emphasize the importance of further studies on the evolution of clones to better understand their impact on modification activities, we are convinced that they do not contradict the negative impact of cloning on maintenance efforts for several reasons. First, data accuracy is unknown—clone detectors frequently produce large amounts of false positives that can dilute conclusions. Furthermore, evolution does not necessarily represent intent: many changes are unintentionally inconsistent [22]; inspection on evolution history alone cannot always reveal them. Second, cloning still impedes maintenance, if not all changes to clones are coupled—it is sufficient, if a substantial fraction of them are.

Kapsner and Godfrey [24] list situations in which cloning can be considered as a sensible development strategy. How-

ever, their work does not challenge negative consequences of cloning for software maintenance. Instead, they argue that cloning can be a sensible tool, if either the software is not maintained (and thus the negative consequences do not take effect) or the available alternatives are still more costly. The proposed cost model can provide a first step to base such decisions on sound economic considerations.

8.2. Cost Estimation

A substantial number of approaches for software cost estimation have been proposed. Besides budgeting and project planning & control, they can in principle be applied for software improvement investment analysis, such as, *e.g.*, tool selection of reengineering. The survey by Boehm, Abts and Chulani gives a detailed overview of existing approaches [7], including cost model based ones. Among the most widespread cost models are COCOMO and its successor COCOMO II [5]. Besides general purpose cost estimation models, some research has evaluated the consequences of individual quality characteristics on maintenance costs. Banker et al. investigate code complexity [4], Lanning and Khoshgoftaar [29] examine coupling and Brill et al. evaluate architecture quality [8].

To the best of our knowledge, the cost model presented here is the first to focus on the impact of code cloning on development costs.

9. Conclusion

This paper proposes an analytical cost model to quantify the economic effect of cloning on maintenance efforts. It can be used as a basis to evaluate clone management alternatives. Instead of computing absolute costs, the model computes maintenance effort increase relative to a system without cloning. Since in a relative cost model many factors that are independent of cloning remain constant, they do not need to be reflected in it. This way, it only requires a small number of factors and can be instantiated with reasonable effort in practice.

We have instantiated the cost model on 11 industrial systems. Although result accuracy could be improved by using project specific instead of literature values for effort parameters, the results indicate that cloning induced impact varies significantly between systems and is substantial for some. Based on the results, some projects can achieve considerable savings by performing active clone management.

There is a definitive need for future work in this area. The assumptions the cost model is based on need to be validated. The consequences of cloning on the number of field failures needs to be modeled quantitatively, instead of qualitatively as currently done in the model. We plan to perform sensitivity analysis to determine the relative impor-

tance of the individual parameters to guide instantiation in practice. Furthermore, we intend to instantiate the model using project specific effort parameters. Lastly but most importantly, we need to validate the correctness of the results, *e. g.*, through comparing efforts on projects before and after clone consolidation with the predicted efforts.

References

- [1] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *Proc. CSMR '07*. IEEE, 2007.
- [2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of WCRE 1995*.
- [3] T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In *Proc. ICSM '07*. IEEE, 2007.
- [4] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig. Software complexity and maintenance costs. *Commun. ACM*, 3, 1993.
- [5] Barry, E. Horowitz, R. Madachy, D. Reifer, B. K. Clark, B. Steece, W. A. Brown, S. Chulani, and C. Abts. *Software Cost Estimation with Cocomo II*. Prentice Hall, January 2000.
- [6] V. Basili, L. Briand, S. Condon, Y.-M. Kim, W. L. Melo, and J. D. Valett. Understanding and predicting the process of software maintenance release. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 464–474. IEEE CS Press, 1996.
- [7] B. Boehm, C. Abts, and S. Chulani. Software development cost estimation approaches – a survey. *Ann. Softw. Eng.*, 2000.
- [8] R. Bril, L. Feijs, A. Glas, R. Krikhaar, and M. Winter. Maintaining a legacy: towards support at the architectural level. *Journal of Software Maintenance*, 2000.
- [9] F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juergens. Flexible architecture conformance assessment with ConQAT. In *ICSE'10; submitted for publication*, 2010.
- [10] F. Deissenboeck, B. Hummel, E. Juergens, B. Schaetz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *ICSE'08*, 2008.
- [11] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka. Tool support for continuous quality control. *IEEE Software*, 25(5):60–67, 2008.
- [12] C. Domann, E. Juergens, and J. Streit. The curse of copy&paste – redundancy in requirements specifications. In *ESEM'09*, 2009.
- [13] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proc. ICSE '07*. IEEE, 2007.
- [14] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of ICSM 1999*.
- [15] M. F. B. H. B. S. S. W. C. D. J. S. Elmar Juergens, Florian Deissenboeck. Can clone detection support quality assessments of requirements specifications? In *Accepted for publication at ICSE'10*, 2010.
- [16] N. Göde. Evolution of type-1 clones. In *SCAM'09*, 2009.
- [17] W. T. B. Hordijk, M. L. Ponisio, and R. J. Wieringa. Harmfulness of code duplication - a structured review of the evidence. In *EASE'09*, 2009.
- [18] IEEE. Standard 1219 for software maintenance. Standard, IEEE, 1998.
- [19] P. Jablonski and D. Hou. CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Proc. Eclipse '07*. ACM, 2007.
- [20] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proc. ESEC-FSE '07*. ACM, 2007.
- [21] E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective – A workbench for clone detection research. In *ICSE'09*, 2009.
- [22] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE '09*, 2009.
- [23] C. Kapsner and M. W. Godfrey. Aiding comprehension of cloning through categorization. In *IWPSE'04*, 2004.
- [24] C. Kapsner and M. W. Godfrey. “Cloning considered harmful” considered harmful. In *Proc. WCRE '06*. IEEE, 2006.
- [25] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proc. ESEC/FSE-13*. ACM, 2005.
- [26] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*. Dagstuhl Seminar Proceedings, 2007.
- [27] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proc. WCRE '07*. IEEE, 2007.
- [28] J. Krinke. Is cloned code more stable than non-cloned code? *SCAM'08*, 2008.
- [29] D. Lanning and T. Khoshgoftaar. Modeling the relationship between source code complexity and maintenance difficulty. *Computer*, 1994.
- [30] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE TSE*, 2006.
- [31] H. D. Rombach, B. T. Ulery, and J. D. Valett. Toward full life cycle control: Adding maintenance measurement to the SEL. *J. Syst. Softw.*, 18(2):125–138, 1992.
- [32] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 541, Queen’s University at Kingston, 2007.
- [33] H. Sneed. A cost model for software maintenance & evolution. In *International Conference on Software Maintenance (ICSM)*. IEEE CS Press, 2004.
- [34] R. Tiarks, R. Koschke, and R. Falke. An assessment of type-3 clones as detected by state-of-the-art tools. In *SCAM'09*, 2009.
- [35] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *VLHCC '04*, 2004.
- [36] D. Yeh and J.-H. Jeng. An empirical study of the influence of departmentalization and organizational position on software maintenance. *J. Softw. Maint. Evol. Res. Pr.*, 14(1):65–82, 2002.