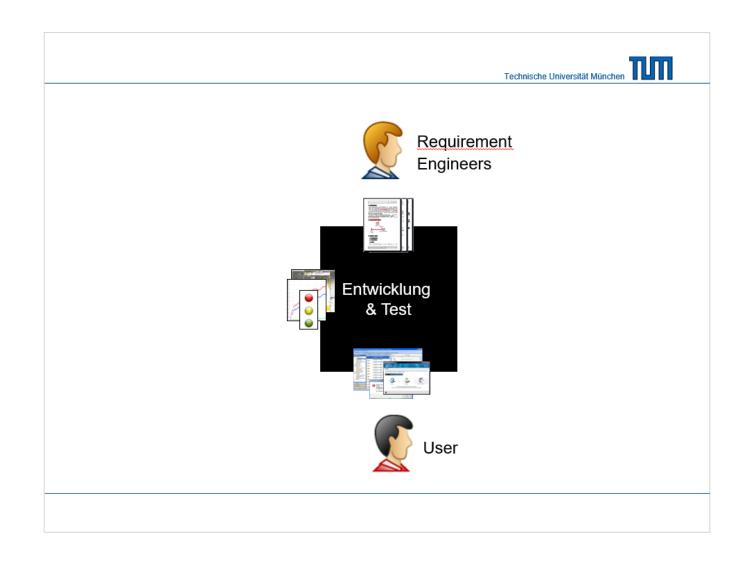
Kosten-Nutzen-Berechnung von Qualitätsanalysen Erfahrungen bei der Munich Re

Uwe Proft (Munich Re) Elmar Jürgens (CQSE)





Do Code Clones Matter?

Elmar Increens Elorian Deissenbosch Benjamin Hummel Stefan Wanne Institut für Informatik, Technische Universität München Roltzmannstr 3 85748 Garchine h München German (juergens deissenb.hummelb.wagnerst)@in.tum.de

Abstract

Code clowing is not only assumed to inflate maint nance costs but also considered defect-prone as inconsistent changes to code doplicates can lead to unexpected behavior. Consequently, the identification of duplicated code, clone detection, has been a very active area of research in recens years. Up to now, however, no substantial investigation of the consequences of code cloning on program correctives has been carried out. To remedy this shortcoming, this paper presents the results of a large-scale case study that was tection tool used in the case study implements a novel algorithm for the detection of inconsistent cloves. It is available is onen source to enable other researchers to use it as busis

Research in software maintenance has shown that many programs contain a significant amount of duplicated (cloned) code. Such cloned code is considered harmful for wo reasons: (1) multiple, possibly unnecessary, duplicates of code increase maintenance costs and. (2) inconsistent changes to cloned code can create faults and, hence, lead to incorrect program behavior 119, 281. While clone deter tion has been a very active area of research in recent years

tion has been a very active area of research in recent years, up to now, there is no thorough understanding of the degree of hammfulness of code cloning. In fact, some researchers even started to doubt the harmfulness of cloning at all [16]. To shed light on the situation, we investigated the effects of code cloning on program correctness. It is important to understand, that clones do not directly cause faults but inconsistent changes to clones can lead to unexpected program behavior. A particularly dangerous type of change

to cloned code is the inconsistent bug fix. If a fault was

found in cloned code but not fixed in all clone instances

found in cloned code but not fixed in all clone instances, the system is likely to still exhibit the incorrer behavior. To illustrate this, Fig. 1 shows an example, where a missing sull-check was retroffitted in only one clone instance. This paper presents the results of a large-scale case study that was understaten to find out (1) if clones are changed in-consistently, (2) if these inconsistencies are introduced intentionally and, (3) if unintentional inconsistencies can represent faults. In this case study we analyzed three commer cial systems written in C#, one written in Cobol and one open-source system written in Java. To conduct the study we developed a novel detection algorithm that enables us to detect inconsistent clones. We manually inspected about with the developers of the respective systems to determin if the inconsistencies are intentional and if they represent faults. Altogether, around 1800 individual clone group as sessments were manually performed in the course of the use study. The study lead to the identification of 107 faults that have been confirmed by the systems' developer

Research Problem Although most previous work agrees that code cloning poses a problem for software main sonce. "There is little information available concerning the meants of code clones on software amility" [28] As the impacts of code clones on software quanty [26]. As the consequences of code cloning on program correctness, in particular, are not fully understood today, it remains unclear how harmful code clones really are. We consider the absence of a thorough understanding of code cloning precari ous for software engineering research, education and prac-

First, we extend the existing empirical knowledge by a case study that demonstrates that clones get changed inconsis-tently and that such changes can represent faults. Second, we present a novel suffix-tree based algorithm for the detection of inconsistent clones. In contrast to other algorithms for the detection of inconsistent clones, our tool suite is



Figure 1, Missing null check on right side can cause exception (Sysiphu

2. Terms and definitions

The literature provides a wide variety of different defi ambiguity, we describe the terms as used in this paper.

Code is interpreted as a sequence of units, which for ex ample could be characters, normalized statements, or lines The reason to allow normalization of units at this stage, is that often nieces of code are considered equal even desni that often pieces of code are considered equal even despite differences in comments or naming, which can be leveled by the normalization. An exact clone is then a (consecutive) substring of the code that appears at least twice in the (nor-malized) code. Thus our definition of a clone is gruely syn-tactical, but catches exactly the idea of copydpaste, while allowing simple changes, such as renaming, due to normalization. An exact clone group is a set of at least two exact 3.1 Consequences of cloning clones that appear at different positions. To capture the notion of non-identical clones, we roughly

follow the definitions of a gapped or type 3 clone giver in [19, 28]. A substring s of the code is called an incomin [19, 28]. A sustering so the code is cause an incon-sistent clone, if there is another substring t of the code such that their edit distance is below a given threshold and that t has no significant overlap with s. The edit distance is a metric that counts the number of edit operations (insertion, removal, or change of a single unit) needed to transform one sequence into the other. Obviously, this definition is slightly vague, as it depends on the threshold chosen and the meaning of a "significant overlap". However, it captures our inestanding of an inconsistent clone as used in this paper. Examples are shown in Figs. 1 and 7. By clone we

paper. Exchingres are shown in ergs. 1 min 7. By come we denote both exact and inconsistent clones.

A clone group can be viewed as a connected graph, where each node is a substring, and edges are drawn between substrings that are clones of each other. If at least one pair of inconsistent clones is in the group, it is called an inconsistent clone group. We could also have required all clones in a clone group to be clones of each other, but often these slightly larger clone groups created by our definition reveal interesting relationships in the code.

For a thorough discussion of the consequences of incoristent clones, we define that a failure is an incorrect output f a software visible to the user and that a failur is the cause of a potential failure inside the code. Defects are the super

to code cloning in recent years. The detailed surveys by Koschke [19] or Roy and Cordy [28] provide a comprehensive overview of existing work. Since this paper targets consequences of cloning and detection of inconsistent clones we detail existing work in these areas.

Indication for harmfulness of cloning for maintainability or correctness is given by several researchers. Lague et al. [23], report inconsistent evolution of a substantial amount of clones in an industrial telecommunication system. Mon den et al. [27] report a higher revision number for files with clones than for files without in a 20 year old legacy system, clones than for files without in a 20 year old legacy system, possibly indicating lower maintainability. In [17], Kim et al. report that a substantial amount of changes to code clones occur in a coupled fashion, indicating additional mainte nance effort due to multiple change locations.

Li et al. 1251 present an approach to detect burs based on aconsistent renaming of identifiers between clones. Jiang Su and Chiu [12] analyze different contexts of clones, such so and condition in any activation of the condition of th by analysis of clone evolution are reported for open source

In contrast, doubt that consequences of cloning are unambiguously harmful is raised by several recent research results. Krinke [22] reports that only half the clones in sev eral open source systems evolved consistently and that only

again through later changes, potentially indicating a larger degree of independence of clones than hitherto believed. Geiger et al. 191 report that a relation between change cou plings and code clones could, contrary to expectations, not be statistically verified. Lozano and Wermelinger [26] renort that no systematic relationship between code cloping bility could be established

. The analyzed systems are too small to be represen-

The analyses specifically focus on faults introduced

during creation [12, 25] or evolution [2] of clones, in

hibiting quantification of inconsistencies in general.

Additional empirical research outside these limitation

is required to better understand consequences of cloning [19, 28], as presented in this paper: Developer rating of the actual inconsistent clones has been performed, the study ob-

jects are both open source and industrial systems and incon

3.2 Detection of inconsistent clones

sistencies have been analyzed independently of their mode

We classify existing approaches according to the pro-gram representation on which they operate. Text Normalized code fragments are compared textually in a pairwise fashion [29]. A similarity threshold governs

Token Ueda et al. [31] propose post-processing of the re

sults of a token-based detection of exact clones. Essen

tially, neighboring exact clones are composed into incon-

sistent clones. In 1251, Li et al. present the tool CP-Mine

which searches for similar basic blocks using frequent sub-sequence mining and then combines basic block clones into larger clones.

misclassification.

The effect of cloning on maintainability and correctness is thus not clear. Furthermore, the above listed publications suffer from one or more shortcomings that limit the trans-Program Dependence Graph Krinke [21] proposes a search algorithm for similar subgraph identification earch algorithm for similar subgraph identification. mondoor and Horwitz [18] propose slicing to identify morphic PDG subgraphs. Gabel, Jiang and Su [8] use a rability of the reported findings. odified slicing approach to reduce the graph isomorp · Instead of manual inspection of the actual inconsistent

- problem to tree similarity. The existing approaches provided valuable inspiration clones to evaluate consequences for maintenance and correctness, indirect measures¹ are used [1, 9, 22, 23, 26, 27]. Such approaches are inherently inaccurate and can easily lead to misleading results. For example, un-intentional differences and faults, while unknown to for the algorithm presented in this paper. However, none of them was applicable to our case study, for one or more of • Tree [3,7,11] and graph [8,18,21] based approaches re developers, exhibit the same evolution pattern as intentional independent evolution and are thus prone to
 - quire the availability of suitable context free grammars for AST or PDG construction. While feasible for modem languages such as lava, this noses a severe prob lem for legacy languages such as Cobol or PL/I, where suitable grammars are not available. Parsing such lan-guages still represents a significant challenge [5, 24].

Abstract Syntax Tree Baxter et al. [3] hash subtrees into

buckets and perform pairwise comparison of subtrees in the same bucket. Jiang et al. [11] propose the generation of

parison, they employ locality sensitive hashing for vector clustering, allowing for better scalability than [3]. In [7],

tree patterns that provide structural abstraction of subtree

- Due to the information loss incurred by the reducbers or vectors, the edit distance between inconsistent clones cannot be precisely controlled in feature vector [11] and hashing based [3] approaches
- · Idiosyncrasies of some approaches threaten recall. In inconsistent clones cannot be detected if their sistituent exact clones are not long enough. In [8], inconsistencies might not be detected if they add data or control dependencies, as noted by the authors.
- · Scalability to industrial-size software of some approaches has been shown to be infeasible [18, 21] or is at least still unclear [7,29]. · For most approaches, implementations are not publicly
- available In contrast, the approach presented in this paper sur

In contrast, the approach presented in this paper sup-ports both modern and legacy languages including Cobol and PL/I, allows for precise control of similarity in terms of edit distance on program statements, is sufficiently scalable to analyze industrial-size projects in reasonable time and is available for use by others as open source software. An approach similar to [31] for bug detection has been

outlined by the authors of this paper in [15]. In contrast to this work, it does not use a suffix tree based algorithm and

The task of the detection algorithm is to find clones in

4. Detecting inconsistent clones

This section explains the approach used for detecting inonsistent clones in large amounts of code. Our approach works on the token level, which usually is sufficient for finding copy-pasted code, while at the same time being effi ient. The algorithm works by constructing a suffix tree of the code and then for each possible suffix an approximat search based on the edit distance in this tree is performed. Our clone detector is organized as a nineline, which is

Filter

Figure 2. The clone detection pipeline used

ketched in Figure 2. The files under analysis are loaded and then fragmented by the scorner, yielding a stream of tokens, which is filtered to exclude comments and gener-ated code (recognized by user provided patterns). From the token stream, which consist of single keywords, identifiers, operators, and so on, the normalizer reassemble statements. This stage performs normalization, such that differences in identifier names or constant values are not vy those statements is then fed into our clone detection also by those statements is then tee almo one clone detection algo-rithms, which finds and reports clone groups in this stream Finally, clone groups are post-processed and uninteresting ones are filtered out. We outline the detection steps in more detail in the following subsections.

4.1. Preprocessing and normalization

As stated before, the code is read and split into tokens using a scanner. An important task during preprocessing is normalization, which creates statements from the scan-ner's tokens. This is used as it allows better tailoring of normalization and to avoid clones starting or ending within statements. The used normalization eliminates differences n naming of identifiers and values of constants or literals. but does not, for example, change operation order.

Further tasks of the preprocessing phase are the removal of comments or generated code, which is either already ex-cluded at the file level or on the token stream based on cerain patterns that recognize sections of generated code

ream of units provided by the normalizer. Stated difbut may have an edit distance bounded by some threshol This problem is related to the approximate string matching problem [13, 32], which is also investigated extensively in bioinformatics [30]. The main difference is that we are not interested in finding an approximation of only a single give word in the string, but rather are looking for all su

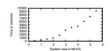
A sketch of our detection algorithm is shown in Figs. 3 and 4. The algorithm is an edit distance based traversal of a suffix tree of our input sequence. A suffix tree over a sequence s is a tree with edges labeled by words such that exactly all suffixes of s are found by traversing the tree from the root node to a leaf and concatenating the words on the edges encountered. Such a suffix tree can be construlinear time by the well-known online algorithm by Ukko

every possible index.

Searching for clones is performed by the procedure two parameters to this function are the sequence s we are working on and the position start where the search was started, which is required when reporting a clone. The pa rameter i (which is the same as start in the first call of warch) marks the current end of the substring under in search marks the current end of the substring under inspec-tion. To prolong this substring, the substring starting at j is compared to the word w being next in the suffix tree, which is the edge leading to the current node v (for the root node we just use the empty string). For this comparison an edit distance of at most e operations (fifth parameter) is allowed. For the first call of search, e is the edit distance maximally allowed for a clone. If the remaining edit operations are not enough to match the entire edge word w (else case), we eport the clone as far as we found it, otherwise the trave report the clone as far as we found it, otherwise the traver-al of the tree continues recursively, increasing the length (j-start) of the current substring and reducing the number e of edit operations available by the amount of operations already spent in this step. To actually make this algorithm work and its results us-

able, some details have to be fleshed out. For the com putation of the longest edit distance match we are using the simple dynamic programming algorithm found in algorithm textbooks. While this is easy to implement, it re juires quadratic time and space². To make this step work

²Actually the algorithm can be implemented using only linear space



on Eclipse source

4.3. Post-processing and filtering

Input: String $s = (s_1, ..., s_n)$, max odit distance s

Figure 3. Outline of approximate clone dete

proc search $(s, \operatorname{start}, j, v, e)$ Input: String $s = (s_0, \dots, s_n)$, start index of current search, current search index j, node v of suffix tree over s, max edit distance e

Let (w_1, \dots, w_m) be the weed along the edge leading to Calculate the maximal length $l \le m$.

Calculate the maximal length $l \le m$, such that there is a $k \ge j$ where the edit distance s' between

 (w_1, \dots, w_l) and (s_l, \dots, s_k) is at most e if l = m then

search (s, whet, k + m, u, e - e'

when if k — start \geq minimal clone length them report substring from atort to k of a so clone

Figure 4. Search routine of the approximate

poses. As each suffix we are running the search on will of course be part of the tree, we also have to make sure that no

When running the algorithm as it is, the results are often

not as expected because the search tries to match as many statements as possible. However, allowing for edit opera-tions right at the beginning or at the end of a clone is not

relpful, as then every exact clone can be prolonged into an aconsistent clone. Thus in the search we enforce the first

few statements (how many is parameterized) to match ex-

actly. (This also speeds up the search, as we can choose the

without looking at all children.) The last statements are also

specting random code. As the potential benefit for the de

elopers is low, the motivation would be low and hence the

As we set the developers for their expert opinion on

whether an inconsistency is intentional or unintentional and aulty or non-faulty, a threat is that the developers do not

judge this correctly. One case is that the developer assesses

something as non-faulty which actually is faulty. This case

only reduces the chances to positively answer the research

hing as faulty which is no fault. We mitigated this threat by

thing as faulty which is no fault. We mitigated this threat by only rating an inconsistency as faulty if the developer was completely sure. Otherwise it was postponed and the devel-oper consulted colleagues that know the corresponding part of the code better. Inconclusive candidates were ranked as intentional and non-faulty. Hence, again only the chance to

answer the research question positively is reduced.

The configuration of the clone detection tool has a strong

influence on the detection results. We calibrated the param

eters based on a pre-study and our experience with clone

eters based on a pre-study and our experience with close detection in general. The configuration also varies over the different programming languages encountered, due to their differences in features and language constructs. However, this should not strongly affect the detection of inconsistent clones because we spent great care to configure the tool in ways that the complete of doors or consider.

We also pre-processed the inconsistent clones that we

presented to the developers in order to eliminate false posi-

tives. This could mean that we excluded clones that are ac-

tually faulty. However, this again only reduces the chance

questions. The second case is that the developers rate some

orrect child node at the root of the suffix tree in one step

or allowed to differ, which is checked for and corrected just

self matches are reported.

these to be reported.

results would be unreliable.

7.2. Internal validity

for each child node u of v do

Construct suffix tree T from a

for each $i \in \{1, \dots, n\}$ de

ion algorithm

During and after detection, the clone groups that are reported are subject to filtering. Filtering is usually performe ported are subject to filtering. Filtering is usually performed as early as possible, so no memory is wasted with storing clone groups that are not considered relevant. Using these filters, we discard clone groups whose clones overlap with each other and groups whose clones are contained in other colone groups. Additionally, we enforce not only an absolute limit on the number of inconsistencies, but also a relative one, i. e., we filter clone groups where the number of inco sistencies in the clones relative to the clone's length exceed a certain amount. Moreover, we merge clone groups which efficiently we look at most at the first 1000 statements of share a common clone. While this leads to clone group efficiently we look at most at the first 1000 statements of the word w. As long as the word on the suffix tree edge is shorter, this is not a problem. In case there is a clone of more than 1000 statements, we would find it in chunks of 1000. We considered this to be tolerable for practical purwith non related clones (as our definition of an inconclone is not transitive), for practical purposes it is preferred to know of these indirect relationships, too.

To be able to experiment with the detection of inconsistent clones, our algorithms and filters have been imple mented as part of CloneDetective' [14] which is based or ConOAT 171. The result is a highly configurable and ex tensible platform for clone detection on the syntactic level found, and thus supports the rapid review of a large num her of clone groups

Due to the many implementation details, the worst case

clones are often not justified by different requirements bu

project. Because both Sysiphus and our detection tools ar

open source, the whole analysis can completely be repli-

cated independently. We provide a web site with the neces

tionally inconsistent changes of clones. The fewer clone

there are in the system, the less likely it is to introduce fault

by inconsistencies between them. In order to increase de-

reloner awareness of clones, we have integrated our clor

detection tool into the Visual Studio development environ-ment. At the Munich Re Group, as a reaction on the clone results, clone detection is now included in the nightly builds of all discussed projects. Furthermore, for existing clones, there should be tool support that ensures that all changes that are made to a clore-

that are made to a clone are made in the full knowledge of

its duplicates. Tools such as CloneTracker [4] or CReN [10

provide promising approaches. However, both approache

are not applicable to existing software that already contains inconsistent clones. Due to their high fault potential, we

consider the obility to detect incor-

can be explained by developer mistakes. We consider of special value the analysis of the Sysiphu

4.5. Scalability and performance

before reporting a clone. Including all of these optimizations, the algorithm can miss a clone either due to the thresholds (either too short or too many inconsistencies), or if it is covered by other clones. The later case is important, as each substring of a complexity is hard to analyze. Additionally, for practical "Available as Open Source at http://comqat.or.tum.edu. clone of course is a clone again and we usually do not want

purposes, the more complicated average complexity would be more adequate. Thus, and to assess the performance of the entire pipeline we executed the detector on the source code of Eclipse⁴, limiting detection to a certain amount of code. Our results on an Intel Core 2 Duo 2.4 GHz running Java in a single thread with 3.5 GB of RAM are shown in Figure 5. The settings are the same as for the main study min clone length of 10, max edit distance of 5). It is capuble to hundle the 5.6 MLOC of Eclinse in about 3 hours which is fast enough to be executed within a nightly build

In order to gain a solid insight into the effects of incon sistent clones, we use a study design with 5 objects and 3 research questions that guide the investigation.

We chose 2 companies and 1 open source project as sources of software systems. This resulted in 5 analyzed projects in total. We chose systems written in different lan-guages, by different learns in different companies and with different functionalities to increase the transferability of the study results. These objects included 3 systems written in C#, a Java system as well as a long-lived Cobol system. All these systems are already in production. For non-discloss reasons we gave the commercial syst D. An overview is shown in Table 1. ercial systems names from A to

Munich Re Group. The Munich Re Group is one of the Munich Re Group The Munich Re Group is one of the largest re-insurance companies in the world and employs more than 37,000 people in over 50 locations. For their in-surance business, they develop a variety of individual sup-porting software systems. In our study, we analyzed the systems A, B and C, all written in CP. They were each leveloped by different organizations and provide substan tially different functionality, ranging from damage predic tion, over pharmaceutical risk management to credit and nounv structure administration. The systems support be ween 10 and 150 expert users each.

LV 1871 The Lebensversicherung von 1871 a.G. (LV 1871) is a Munich-based life-insurance company. The LV 1871 develops and maintains several custom software systems for mainframes and PCs. In this study, we analyze a mainframe-based contract management system written in Cobol (System D) employed by about 150 users.

ributed software development projects. The inclusion of clone detection tool is also freely available, the results can be externally replicated. This is not possible with the de

	. Summary o	f the analy	rzed sys	tems
System	Organization	Language	Age	Sire
			(years)	(kLOC)
A	Munich Re	Ca	6	317
B	Munich Re	Cii	4	454
c l	Munich Re	Cil	2	495
D	LV 1871	Cobel	17	197
Sysiphus	TUM	Java	8	281

5.2. Research question

The first question we need to answer is whether inconsistent clones appear at all in real-world systems. This not only means whether we can find them at all but also whether they constitute a significant part of the total clones of a system. It does not make sense to analyze inconsistent clones if they

RQ 2 Are inconsistent clones created unintentionally

whether the inconsistency is intentional

RO3 Can inconsistent clones be indicators for faults in



Table 1	Table 1. Summary of the analyzed systems					
System	Organization	Language	Age	Sine		
			(years)	(kLOC)		
A	Munich Re	Ca	6	317		
В	Munich Re	C#	4	454		
C	Munich Re	Cil	2	495		
D	LV 1871	Cobel	17	197		

The underlying problem that we analyze are clones and specially their inconsistencies. In order to investigate this question, we answer the following 3 more detailed re

The first question we need to answer is whether inconsisten

Maxima actablished that there are inconsistent clones in real taving estatoushed that there are inconsistent coopes in real systems, we need to analyze whether these inconsistent lones have been created intentionally or not. It can ob-iously be sensible to change a clone so that it becomes nsistent to it counterparts because it has to conform to lifferent requirements. However, the important difference is whether the developer is aware of the other clones, i.e.



Figure 6, Clone Group Sets

After establishing these prerequisites, we can determine whether the inconsistent clones are actually indicators for faults in real systems. If there are inconsistent clones that have not been created because of different requirements this implies that at least one of these clones does not con-form to the requirements. Hence, it constitutes a fault.

We answer the research questions with the following study design. In the study we analyze sets of clone groups as shown in Fig. 6. The outermost est are all clone groups \mathcal{E} in a system, \mathcal{H} denotes the set of inconsistent clone groups. The subsect \mathcal{F} of $U\mathcal{H}$ C consists of those unintentionally inconsistent \mathcal{L} of $U\mathcal{H}$ C consists of those unintentionally inconsistent clone groups that indicate a fault in the program. Please note that we do not distinguish between created and evolved tent clones as for the question of faultiness it does

of matter when the inconsistences have been introduced.
We use these different clone group sets to design the
tudy that answers our research questions. The independent
variables in the study are development team, programming language, functional domain, age and size. The dependent variables for the research questions are explained below. RQ I investigates the existence of inconsistent clones in realistic systems. Hence, we need to analyze the size of set IC with respect to the size of set C. We apply our inconsistent clone analysis approach to all the systems, perform manual assessment of the detected clones to eliminate false

tives and calculate the inconsistent clone ratio |IC|/|C| For RO 2, whether clones are created For RQ 2, whether clones are created unintentionally, we then compare the size of the sets UIC and IC. The sets are established by showing each identified inconsistent clone to developers of the system and asking them to rate them as intentional or unintentional. This gives us the amintenionally inconsistent clone ratio |UIC|/|IC|. The most important question we aim to answer is whether inconsistent lones indicate faults (RQ 3). Hence, we are interested in the size of set F in relation to the size of IC. The set Fsined by asking developers of the respective is again determined by asking developers of the respective system. Their expert opinion classifies the clones in faulty and non-faulty. We only analyze unintentionally inconsis-tent clones for faults. Our faulty inconsistent clone ratio |F|/|R'| is thus a lower bound, as notential faults in inten-

onally inconsistent clones are not considered.

Using this, we are already able to roughly find the answer to RQ 3. As this is our main result from the study, the transform it into a hypothesis. We need to make sure that the fault density in the inconsistencies is higher than n randomly picked lines of source code. This leads to the

The fault density in the inconsistencies is higher than the

As we do not know the actual fault densities of the analyzed systems, we need to resort to average values. The span of available numbers is large because of the high vari-ation in software systems. Endres and Rombach [6] give 0.1-50 faults per kLOC as a typical range. For the fault density in the inconsistencies, we use the number of fault divided by the logical lines of code of the inconsistencies We refrain from testing the hypothesis statistically because of the low number of data points as well as the large rang of typical defect densities

The treatment we used on the objects was the approa to detect inconsistent clones as described in section 4. For all systems, the detection was executed by the researcher to identify consistent and inconsistent clone candidates. On an 1.7 GHz notebook, the detection took between one and two minutes for each system. The detection was configured to not cross method boundaries, since experiments showed that inconsistent clones that cross method houndaries in that inconsistent clones that cross method boundaries in many cases did not capture semantically meningful con-cepts. This is also noted for exact clones in [20] and is even more pronounced for inconsistent clones. Since in Cobol sections in the procedural division are the counterpart of Java or C# methods, clone detection for Cobol was limited

For the C# and Java systems, the algorithm was param eterized to use 10 statements as minimal clone length, a maximum edit distance of 5. a maximal inconsis tio (i.e., the ratio of edit distance and clone length) of 0.2 to (i.e., the ratio of call distance and ctone length) of 0.2 and the constraint that the first 2 statements of two clones need to be equal. Due to the verbosity of Cobol [5], mini-mal clone length and maximal edit distance were doubled to 20 and 10, respectively. Generated code that is not subject to manual editing was excluded from clone detection, since inconsistent manual updates obviously cannot occur. No malization of identifiers and constants was tailored as appropriate for the analyzed language, to allow for rena of identifiers while at the same time avoiding too large fals

nt clone mouns |IC| Faulty clone groups [F] Fault density in kLOC

random samples of the detected clones have been evaluated

The detected clone condidates were then manually rated by the researcher in order to remove false positives, i.e., code fragments that, although identified as clone candidates by the detection algorithm, have no semantic relationship. Inconsistent and exact clone group candidates were treated differently: all inconsistent clone group candidates were rated, producing the set of inconsistent clone groups. Since the exact clones were not required for further steps of the ase study, instead of rating all of them, a random same of 25% was rated, and false positive rates then extra

of 25% was rated, and raise positive rates then extrapolated to determine the number of exact clones.

The inconsistent clone groups were then presented to the developers of the respective systems in the tool CloneDelective mentioned in Section 4.4, which is able to display the commonalities and differences of the clone group in a clearly arranged way, as depicted in Figs. 1 and 7. The developers rated whether the clone groups were created intentionally or unintentionally. If a clone group was created sintentionally, the developers also classified it as faulty or on-faulty. For the Java and C# systems, all inco non-faulty. For the Java and C# systems, all inconsistent clone groups were rated by the developers. For the Cobol system, rating was limited to a random sample of 68 out of the 151 inconsistent clone groups, since the age of the sys-tem and the fact that the original developers were not available for rating increased rating effort. Thus, for the Cobol case, the results for RQ 2 and RQ 3 were computed based on this sample. In cases where intentionality or faultiness could not be determined, e.g., because none of the original developers could be accessed for rating, the inconsistencie were treated as intentional and non-faults

The quantitative results of our study are summarized in Table 2. Except for the Cobol system D, the precision values are smaller for inconsistent clone groups than for ex act come groups, as was expected, since inconsistent clon groups allow for more deviation. The high precision result of system D result from the rather conservative clone detec-tion parameters chosen due to the verbosity of Cobol. For system A, stereotype database access code of semantically unrelated objects gave rise to lower precision values.

About half of the clones (52%) contain inconsistencies Therefore, RQ I can be positively answered: Clones are herefore, RQ 1 can be positively answered: Clones are hanged inconsistently. All these would not be reported by xisting tools that search for exact matches. From these aconsistencies over a quarter (28%) has been introduced nintentionally. Hence, RQ 2 can also be answered positively: Inconsistent clones are created unintentionally in many cases. Only system D is far lower here, with only 10% of unintentionally inconsistent clones. With about three quarters of intentional changes, this shows that cloning and changing code seems to be a frequent pattern during devel

faults, we note that at least 3-23% of the inconsistencies ac-tually presented a fault. Again the by far lowest number comes from the Cobol system. Ignoring it, the total ratio of faulty inconsistent clones goes up to 18%. This consti tutes a significant share that needs consideration. To judge hypothesis H, we also calculated the fault densities. The bypothesis H, we also calculated the fault densities. They like in the range of 3-49 of faults by ed. DCC. Again, system D is an outlier. Compared to reported fault densities in the range of 0.1 to 3 foults and considering the fact that all systems are not only delibered but even how been productive for seceral years we consider our results to support hypothesis H. On average the inconsistencies contain more faults han average code. Hence, R3 3 can also be answered positively: Inconsistent clones can be indicators for faults in outle covered.

While the numbers are similar for the C# and Java



Figure 7. Different UI behavior since right side does not use operations (Sysiphus

faults are comparatively low for project D, which is a legacy system written in Cobol. To a certain degree, we attribute this to our conservative assessment strategy of treating in-consistencies whose intentionality and faultiness could not be unambiguously determined as intentional and non-faulty Furthermore, interviewing the current maintainers of the systems revealed that cloning is such a common pattern in Cobol systems, that searching for duplicates of a piece of Cobol systems, that searching for duplicates of a piece of code is actually an integral part of their maintenance pro-cess. Compared to the developers of the other projects, the Cobol developers where thus more aware of clones in the system. To account for this difference in "clone aware-ness" we added the row |F|/|UIC| to Table 2, which reeals that while the rates of unintentional changes are lowe for project D, the ratio of unintentional changes leading to a fault is in the same range for all projects. From our results it eems that about every second to third unintentional chang to a clone leads to a fault

Although not central to our research questions, the detection of faults almost automatically raises the question for their severity. As the fault effect costs are unknown for the analyzed systems, we cannot provide a full-fledged severity classification. However, we provide a partial answer by categorizing the found faults as (1) faults that lead to poential system crash or data loss, (2) unexpected behavior isible to the end user and (3) unexpected behavior not visible to the end user. One example for a category (1) fault is hown in Fig 1. Here, one clone of the affected clone group shown in Fig 1. Here, one clone of the affected clone group performs a ulti-heck to prevent a null-pointer dereference whereas the other does not. Other examples we encountered for category (1) faults are index-out-of-hounds exceptions, incorrect transaction handling and missing rollback. Fig. 7 shows an example of a category (2) fault. In one clone the performed operation is not encapsulated in an operation object and, hence, is handled differently by the undo mecha-

user forms and dialogs. Category (3) examples we iden tified include unnecessary object creation, minor memory leaks, performance issues like missing break statements in loops and redundant re-computations of cache-able values differences in exception handling, different exception and debug messages or different log levels for similar cases. Of the 107 inconsistent clones found, 17 were categorized as category (1) faults, 44 as category (2) faults and 46 as cat poory (7) faults. Since all analyzed systems are in pro-

We discuss how we mitigated threats to construct, internal and external validity of our study.

systems in order to determine if the inconsistencies really have been introduced by incomplete changes to the system and not by random similarities of unrelated code. This ha two reasons: (1) We want to analyze all inconsistent clones also the ones that have been introduced directly by copy and modification in a single commit. Those might not be visib in the repository (2) The industrial systems do not have

in the repository. (2) The industrial systems do not have complete development histories. We confronted this threat by manually analyzing each potential inconsistent clone. The comparison with average fault probability is not perfect to determine whether the inconsistencies are really more fault-prone than a random piece of code. A comparison with the actual fault densities of the systems or actual checks for faults in random code lines would better suit thi

that we can answer our research question positively 7.3. External validity The projects were obviously not sampled randomly from

a way that the resulting clones are sensible

all possible software systems but we relied on our connec tions with the developers of the systems. Hence, the set of systems is not completely representative. The majority of he systems is written in C# and analyzing 5 systems in to tal is not a high number. However, all 5 systems have been tal is not a high number. However, all 5 systems have been developed by different development organizations and the C#-systems are technically different (2 web, 1 rich client) and provide substantially different functionalities. We fur-ther mitigated this threat by also analyzing a legacy Cobol system as well as an open source Java system

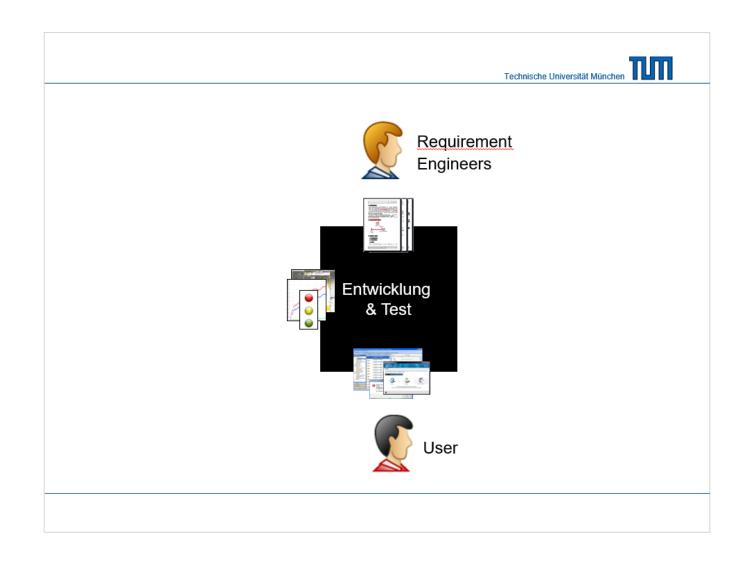
heir use more efficient for practical applications. More over, it will be interesting to compare different detection pa rameter values, algorithms and tools according to their per formance and accuracy when finding inconsistent clone

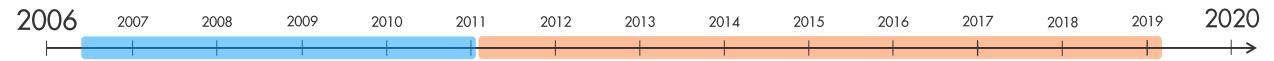
In this paper we provide strong evidence that inconsisin this paper we provide strong evidence that inconsis-tent clones constitute a major source of faults, which means that cloning can be a substantial problem during develop-ment and maintenance unless special care is taken to find and track existing clones and their evolution. Our results remove that the marks were conducted to the conduction of the consuggest that nearly every second unintentionally inconsis-tent change to a clone leads to a fault. Furthermore, we provide a scalable algorithm for finding such inconsisten clones as well as suitable tool support for future experi-

Future work on this tonic will evolve in multiple direct Future work on mrs topic will evolve in humapie ouections. One obvious development is the refinement of the algorithms and tools used. This includes refined heuristics to speed up the clone search and perform automatic assessment to discard obviously irrelevant clones. In addition, the usability of the tools could be advanced further to make

8. Discussion

pcr ans. rence. It handled differently by the units because the second one between the best and the se







Dashboard

Activity

▲ Findings

% Metrics

Issues

₹≡ Tasks

Architecture

∠ Delta

Projects

System

Admin

```
jenkins/test/src/main/java/org/jvnet/hudson/test/HudsonTestCase.java
```

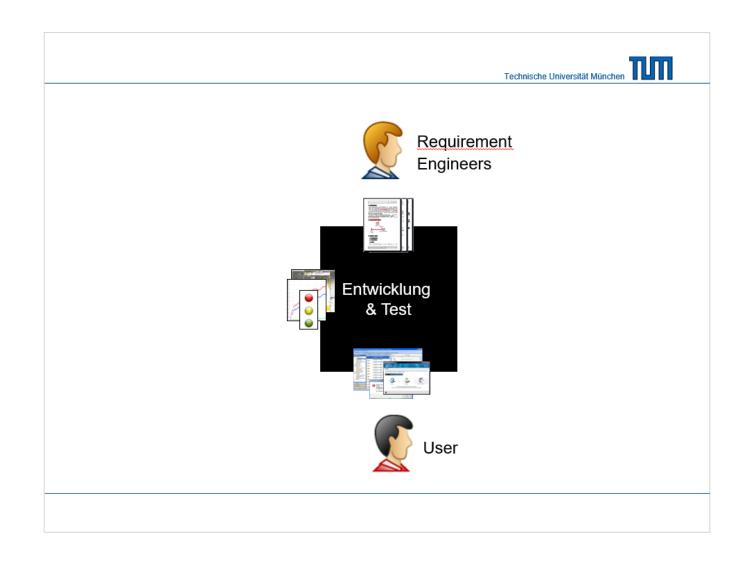
(revision 12d96a56...)

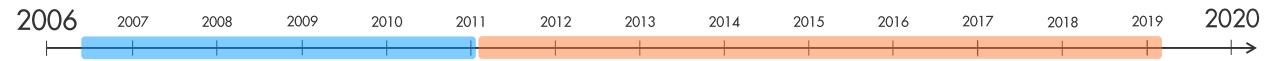
```
jenkins/test/src/main/java/org/jvnet/hudson/test/JenkinsRule.java
```

(revision 3909f5ac...)

```
fail("lhs is null while rhs="+rhs);
 if (lhs==null)
                     fail("rhs is null while lhs="+lhs):
 if (rhs==null)
 Constructor<?> lc = findDataBoundConstructor(lhs.getClass());
 Constructor<?> rc = findDataBoundConstructor(rhs.getClass());
  assertEquals("Data bound constructor mismatch. Different type?",lc,rc);
 List<String> primitiveProperties = new ArrayList<String>();
 String[] names = ClassDescriptor.loadParameterNames(lc);
 Class<?>[] types = lc.getParameterTypes();
  assertEquals(names.length,types.length);
 for (int i=0; i<types.length; i++) {</pre>
     Object lv = ReflectionUtils.getPublicProperty(lhs, names[i]);
     Object rv = ReflectionUtils.getPublicProperty(rhs, names[i]);
     if (Iterable.class.isAssignableFrom(types[i])) {
         Iterable lcol = (Iterable) lv;
         Iterable rcol = (Iterable) rv:
         Iterator ltr.rtr:
         for (ltr=lcol.iterator(), rtr=rcol.iterator(); ltr.hasNext() && rtr.hasNext();)
             Object litem = ltr.next();
             Object ritem = rtr.next();
             if (findDataBoundConstructor(litem.getClass())!=null) {
                 assertEqualDataBoundBeans(litem, ritem);
             } else {
                 assertEquals(litem, ritem);
         assertFalse("collection size mismatch between "+lhs+" and "+rhs, ltr.hasNext() ^
     } else
     if (findDataBoundConstructor(types[i])!=null || (lv!=null && findDataBoundConstructo
         // recurse into nested databound objects
         assertEqualDataBoundBeans(lv,rv);
     } else {
         primitiveProperties.add(names[i]);
 // compare shallow primitive properties
 if (!primitiveProperties.isEmpty())
     assertEqualBeans(lhs,rhs,Util.join(primitiveProperties,","));
Makes sure that two collections are identical via {@link #assertEqualDataBoundBeans(Objections)
blic void assertEqualDataBoundBeans(List<?> lhs, List<?> rhs) throws Exception {
 assertEquals(lhs.size(), rhs.size());
```

```
1T (ths==nutt ωω rns==nutt) return;
                      fail("lhs is null while rhs="+rhs);
  if (lhs==null)
                      fail("rhs is null while lhs="+lhs):
  if (rhs==null)
  Constructor<?> lc = findDataBoundConstructor(lhs.getClass());
  Constructor<?> rc = findDataBoundConstructor(rhs.getClass());
  assertThat("Data bound constructor mismatch. Different type?", (Constructor)rc, is((Cons
  List<String> primitiveProperties = new ArrayList<String>();
  String[] names = ClassDescriptor.loadParameterNames(lc);
  Class<?>[] types = lc.getParameterTypes();
  assertThat(types.length, is(names.length));
  for (int i=0; i<types.length; i++) {</pre>
      Object lv = ReflectionUtils.getPublicProperty(lhs, names[i]);
      Object rv = ReflectionUtils.getPublicProperty(rhs, names[i]);
      if (lv != null && rv != null && Iterable.class.isAssignableFrom(types[i])) {
          Iterable lcol = (Iterable) lv;
          Iterable rcol = (Iterable) rv;
          Iterator ltr.rtr:
          for (ltr=lcol.iterator(), rtr=rcol.iterator(); ltr.hasNext() && rtr.hasNext();)
              Object litem = ltr.next();
              Object ritem = rtr.next();
              if (findDataBoundConstructor(litem.getClass())!=null) {
                  assertEqualDataBoundBeans(litem, ritem);
              } else {
                  assertThat(ritem, is(litem));
          assertThat("collection size mismatch between " + lhs + " and " + rhs, ltr.hasNex
                  is(false)):
      } else
      if (findDataBoundConstructor(types[i])!=null || (lv!=null && findDataBoundConstructo
          // recurse into nested databound objects
          assertEqualDataBoundBeans(lv,rv);
     } else {
          primitiveProperties.add(names[i]);
  // compare shallow primitive properties
  if (!primitiveProperties.isEmpty())
      assertEqualBeans(lhs,rhs,Util.join(primitiveProperties,","));
 Makes sure that two collections are identical via {@link #assertEqualDataBoundBeans(Objec
blic void assertEqualDataBoundBeans(List<?> lhs, List<?> rhs) throws Exception {
```



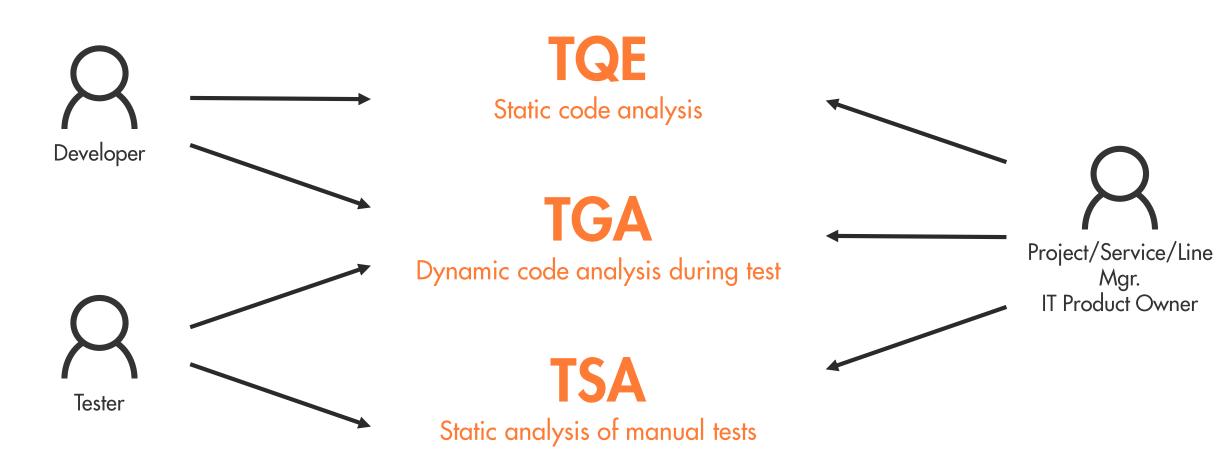


Wie sehen die Werkzeuge & Prozesse bei der Munich Re aus?

Uwe Proft

- Background im Software Engineering und Provider Management
- Seit 7 Jahren bei der MR in Rollen zum Qualitätsmanagement
 - Erläutern Nutzen und Aufwand intern
 - Ausrollen, auch international an unterschiedlichen Standorten
 - Change-Management
 - Vermittlung der Messergebnisse für Beurteilung der Qualität von Zulieferern und Projekten
 - Steuerung des Teams der Quality Engineers (CQSE) bei der Munich Re

Quality Tools



Munich Re Internal Services

Dashboards, IDE Plugin, Azure DevOps

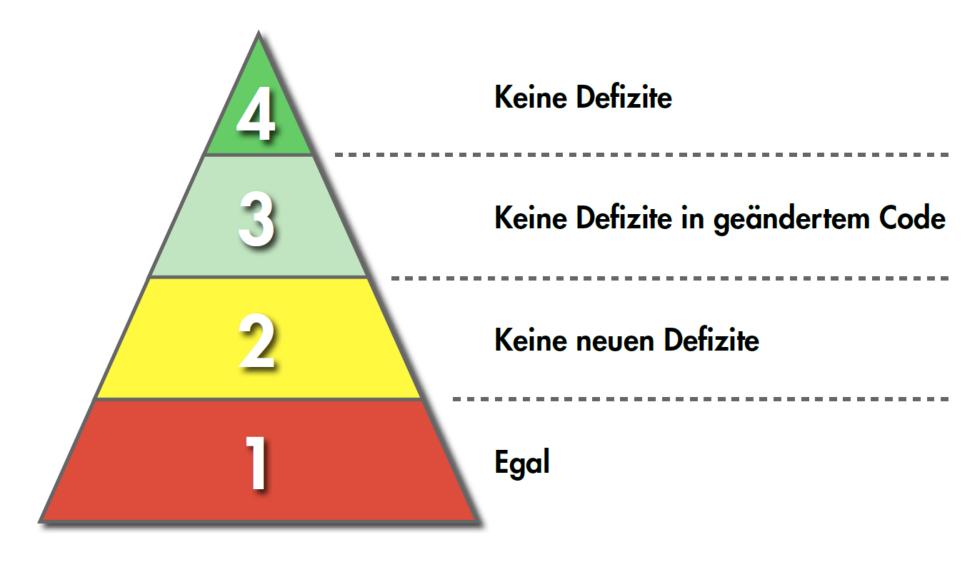


Monthly Assessments, Reports

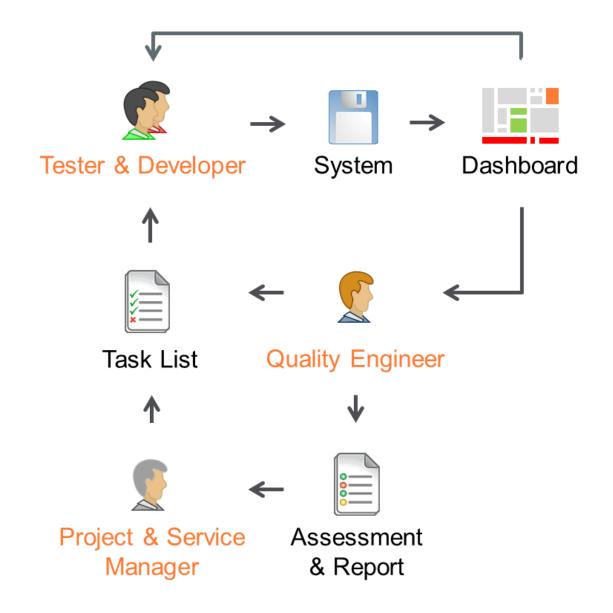


```
StyleCop - Microsoft Visual Studio (Administrator)
           VIEW PROJECT BUILD DEBUG TEAM
                                                 TOOLS
                                                          VISUALSVN
                                                                     TEST
                                                                           ANALYZE
                                                                                    ▶ Start ▼ Debug
Document Outline
                                                                                                                                                                            QueryClauseType.cs
     CodeParser.Expressions.cs + X CodeParser.Preprocessor.cs
                                                           CodeParser.Statements.cs
                                                                                    CsParser.cs
                                                                                                   CsToken.cs
                                                                                                                  ElementType.cs
                                                                                                                                     FileHeader.cs
                                                                                                                                                     ICodePartExtensions.cs
   StyleCop.CSharp.CodeParser
                                                                                                     else
      2736
      2737
      2738
                                   initializerValue = this.GetNextExpression(ExpressionPrecedence.None, initializerExpressionReference, unsafeCode);
      2739
      2740
                               // Create and add this initializer.
      2741
      2742
                               CsTokenList initializerTokens = new CsTokenList(this.tokens, identifier.Tokens.First, initializerValue.Tokens.Last);
                               AssignmentExpression initializerExpression = new AssignmentExpression(
      2743
                                   initializerTokens, AssignmentExpression.Operator.Equals, identifier, initializerValue);
      2744
      2745
                               initializerExpressionReference.Target = initializerExpression;
      2746
                               initializerExpressions.Add(initializerExpression);
      2747
      2748
                               // Check whether we're done.
      2749
              Clone with 2 instances of length 9 this.GetNextSymbol(expressionReference);
      2750
                                       bl.SymbolType == SymbolType.Comma)
      2751
              Redundancy
      2752
              Clones
      2753
                                        tokens.Add(this.GetToken(CsTokenType.Comma, SymbolType.Comma, expressionReference));
      2754
                                   // If the next symbol after this is the closing curly bracket, then we are done.
      2755
                                   symbol = this.GetNextSymbol(expressionReference);
      2756
                                   if (symbol.SymbolType == SymbolType.CloseCurlyBracket)
      2757
      2758
                                       break;
      2759
      2760
      2761
                               else
      2762
      2763
      2764
                                   break;
      2765
      2766
      2767
      2768
                           // Add and move past the closing curly bracket.
                           Bracket closingBracket = this.GetBracketToken(CsTokenType.CloseCurlyBracket, SymbolType.CloseCurlyBracket, expressionReference);
      2769
                           Node<CsToken> closingBracketNode = this.tokens.InsertLast(closingBracket);
      2770
```

Qualitätsziele



Prozess



Portfolio Overview - Links to Dashboards & Monthly Assessments

Application	Dev	Test \$	TQE	TGA ≑	TSA ≑
HyCl. Temple Cubbase	Mary HTL IT1.5 (MI)	MOM MO, 81 (48)	2019-12	X*	Q
10-000	HCD Stephen Information HCD Stephen HT1.2 (RAVI - Technology Design Review)	MPInC Regeri ET (1/E)/ET - Technology Design Sendons)	~	Q	2019-12
Mins	171.5 (MIN)	MRM TELES (MERCHAC) (C) 6-68)	2019-12	2019-12	2019-12
Next 3	IT1.5 (Mg)	NAME OF THE PERSON OF T	P	Q	Q
Uhileps	Miles IT1.8 (m) - On Little House,	140e0 - 371.6 (55 - 311.6 Shearing	2019-12	Q	///2019-12////

Portfolio Overview – Trends

a contraction	, p.	TQE assessment trend for		,		T0.6
Application	▲ Dev				TGA ∳	TSA
HyCu Temple Colsissor	660gG HGS, IT	Assessment Comment	QG relevant findings		X*	Q
15-000	MONE Applies	2019-09 Only one small finding in changed code	1	Show Details	Q	2019-12
	IT1.2 FAU Technolog	//2019-08// No code changes.	0		Ψ.	201012
-	ma 11.12 pm N	2019-07 Only minor new findings	6	Show Details	019-12	2019-12
No.	IT1.5	2019-06 Only a small change with no findings churn.	0		019-12	2019-12
4963	10M F15,000 K	2019-05 Only 2 small findings in modified code.	2		Q	Q
	IT1.5 💏	2019-04 Mostly minor violations.	70	Show Details		
Uhiteps	IT1.8	Notable findings: Naming convention violations in TmOneParam Method threshold violation in method DeleteProcessYear of class ProcessYear Method threshold violation in a lambda in class LossChartService Cloning between ReverseTriangleGrid and CommissionTriangleGrid (c.f. here			Q	//// <u>2</u> 919:12///,
		2019-03 Minor violations only.	3			
		2019-02 Only 5 new findings. Remaining findings are located in code that was changed during a migration to Angular 7 and thus can be ignored for tassessment.		Show Details		

Portfolio Overview – Trends

TQE assessment trend for

Application	▲ Dev
HyCL Temple Colsisione	660gG P60, IT
M-MA	PROD Septem TT1.2 (EAST Technology
Mos	IT1.5
R00-3	IT1.5 (MA)
Uhrlage	IT1.8

Assessment	Comment	relevant findings	Details
2019-09	Only one small finding in changed code	1	Show Details
/2019-08//	No code changes.	0	
2019-07	Only minor new findings	6	Show Details
2019-06	Only a small change with no findings churn.	0	
2019-05	Only 2 small findings in modified code.	2	
2019-04	Mostly minor violations.	70	Show Details
Notable finding	gs:		
Method tMethod t	convention violations in TmOneParam hreshold violation in method DeleteProcessYear Of Class ProcessYearsContr hreshold violation in a lambda in class LossChartService Detween ReverseTriangleGrid and CommissionTriangleGrid (c.f. here)	oller	
2019-03	Minor violations only.	3	
2019-02	Only 5 new findings. Remaining findings are located in code that was changed during a migration to Angular 7 and thus can be ignored for this assessment.	39	Show Details

QG

	Assessment	Comment	tindings	Details		
	2019-09	Only few findings compared to the amount of change	93	Show Details		
	2019-08	Mostly minor violations given large amount of changes.	87	Show Details		
	2019-07	Mostly minor violations given large amount of changes.	131	Show Details		
	2019-06	Mostly minor violations.	117			
	2019-05	Mostly minor violations.		Show Details		
	2019-04	Tolerable number of findings given amount of code changes	194	Show Details		
	2019-03	Tolerable but not significant amount of findings.	198	Show Details		
	2019-02	Tolerable but no significant amount of findings.	160	Show Details		
	2019-01	Tolerable but significant amount of findings.	173	Show Details		
	2018-12	Minor violations only.	77	Show Details		
×	2018-11	Some findings that could have been avoided and resolved.	123	Show Details		
	2018-10	Minor violations only. Amount of findings tolerable with	121	Show Details		
	7579.00	respect to code chum. Minor violations only. Amount of findings tolerable with	64		_	0.4
	2018-09	respect to code churn.	04		ı	SA
	2018-08	Minor violations only.	181	Show Details		
	2018-07	Duplicated code in lean and spc classes increased clone coverage significantly.	101	Show Details		Q,
	2018-06	Minor violations only:	77			
	2018-05	Tolerable amount of findings wrt to code change.	180			
ails	2018-04	Tolerable amount of findings wrt to code change	99	Show Details		
	2018-03	Findings correspond to minor structural violations only.	97			~
	2018-02	New findings correspond to minor structural violations.	77			
	2018-01	Goal reached. Amount of findings tolerable with respect to code churn.	122	Show Details		
tails	2017-12	Goal reached. Amount of findings tolerable with respect to code churs.	101			
	2017-11	Goal reached. Amount of findings tolerable with respect to code churs.	37			~
	2017-10	High code chum with positive trend in most quality indicators. Architecture specification not up-to-date anymore.	80	Show Details		
	2017-09	Code duplications, long methods, deeply nested code and coding guideline violations	92	Show Details		
1-11-	2017-08	Neetly duplicated code, new long methods, new coding guideline violations. However tolerable, increase of system size by 10,000 code lines.	148	Show Details		Q
ails	2017-07	new long methods, new deeply nested code,. OK will to file changes Build! Test: builds and leats are running.	8			
	2017-06	New findings tolerable, new compiler warnings may also be from test month	46		///.	~
	2017-05	SLOC +5000. Several clones between bios and normal submission header broke up (intended?). Also some new clones. Other findings tolerable.	99			
		New closed component (Treaty.Ktss.DetsAccess)	142			
		several new clones affecting not related business entitles, findings tolerable given growth of +10k SLOC	106			
	2017-02	Given large amount of new code (+10k LOC), new findings ok; also resolved several old ones	40			
		encludeds review related findings	53	Show Details		
		Peer review findings have been substracted	28	Show Details		
tails	2016-10	Given amount of development acceptable amount of new findings.	71			
	2016-09	Most findings are either architecture related or target revewl findings or lest code. The remaining ones or minor clones or minor method length violations	297			
	2016-08	Most findings concern the unfinished architecture spec. However several findings in new or modified code.	388			
				Close		

TQE assessment trend for

Monthly Assessment Results Portfolio Aggregation



Name	1001	Assessment	Comments
19840	TQE	GREEN	Only few violations
1540	TGA	YELLOW	Some relevant test gaps
PER	TSA	GREEN	The team added 47 new test case, changed 11 test case, moved 5 test case and removed 15 test cases, which introduced 0 new findings.

Was bringt's?

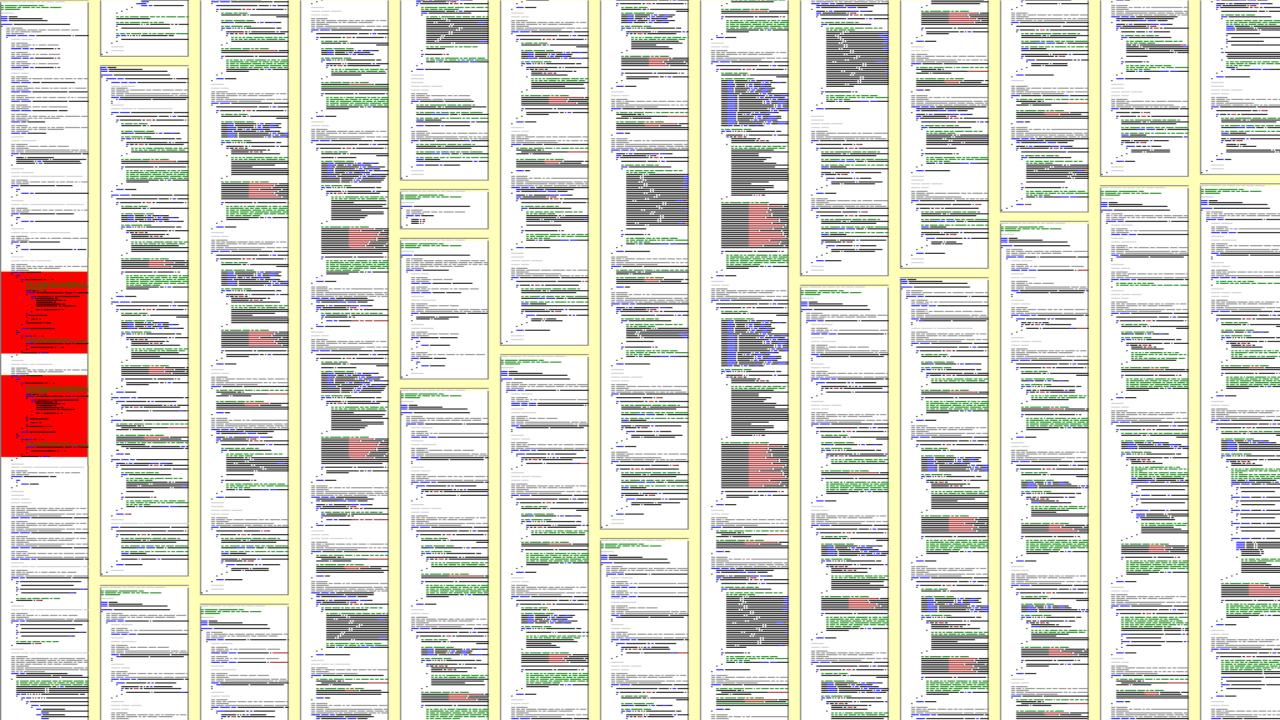


Wie können wir den Nutzen quantifizieren?

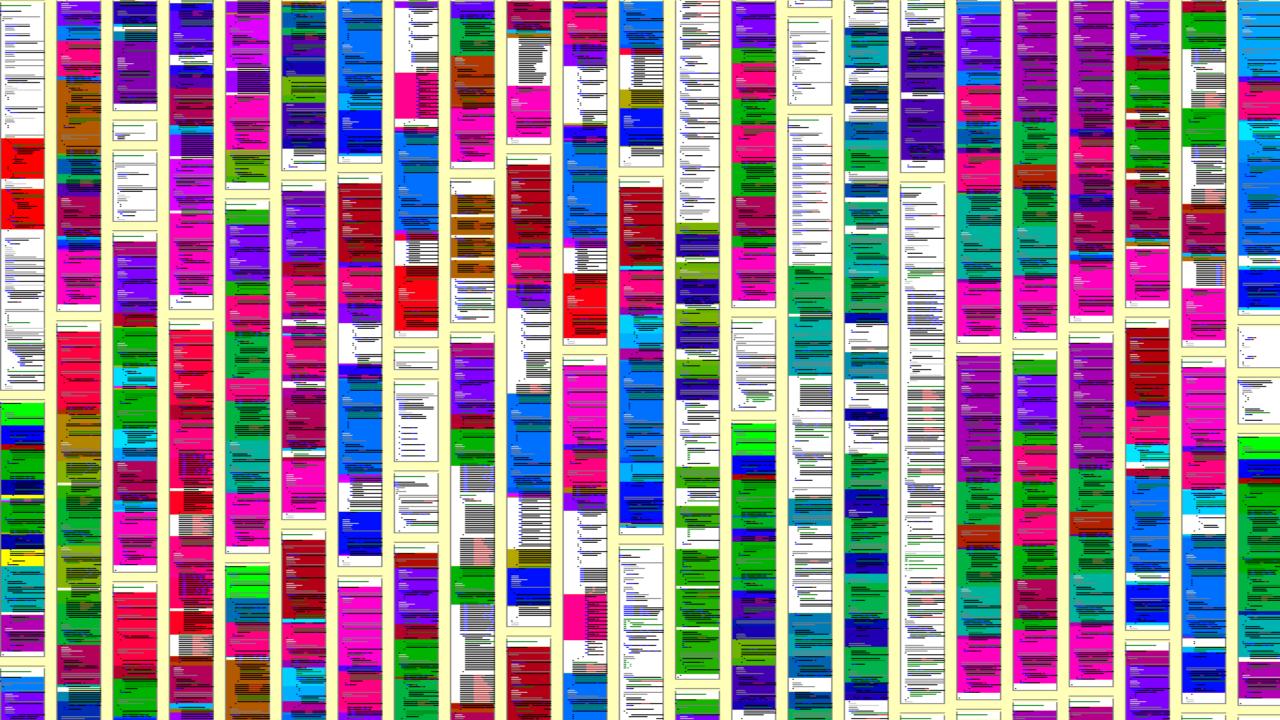
```
// Utilities for arrays of elements
public String showElements(ModelElement[] elements, String nomsg) {
    boolean found = false;
    StringBuffer res = new StringBuffer();
    if (elements != null) {
        Index.getInstance().setCurrentRenderer(
            FlatReferenceRenderer.getInstance());
        for (int i = 0; i < elements.length; i++) {
            ModelElement el = elements[i];
            res.append(showElementLink(el)).append(HTML.LINE_BREAK);
            found = true;
        }
        Index.getInstance().resetCurrentRenderer();
    }
    if (!found && nomsg != null && nomsg.length() > 0) {
        res.append(HTML.italics(nomsg));
    }
    return res.toString();
}
```

```
// Utilities for arrays of elements
public String showElements(ModelElement[] elements, String nomsg) {
    boolean found = false;
    StringBuffer res = new StringBuffer();
    if (elements != null) {
        Index.getInstance().setCurrentRenderer(
            FlatReferenceRenderer.getInstance());
    for (int i = 0; i < elements.length; i++) {
            ModelElement el = elements[i];
            res.append(showElementLink(el)).append(HTML.LINE_BREAK);
            found = true;
        }
        Index.getInstance().resetCurrentRenderer();
    }
    if (!found && nomsg != null && nomsg.length() > 0) {
        res.append(HTML.italics(nomsg));
    }
    return res.toString();
}
```





	<u>:::</u>		I == − I		<u> </u>				<u></u>	= ==		
					<u> </u>				 .			
· ====================================	<u></u>					·			· · · · · · · · · · · · · · · · · · ·		· 	
· ==	.:				-	***************************************						
	[.=											
			**************************************				<u> </u>					· · · · · · · · · · · · · · · · · · ·
		<u> </u>	<u> </u>				·		· 		12772	
==					*			<u> </u>		_		<u>. — — — </u>
===		- 		==-					. 	 	I <u>=-</u>	_
==			***************************************					1979/2417	=			==
===												======
		·	. —					_				·
					<u> </u>			==-		- 		
					===	. 					<u></u>	
==:												·
			·									
									1575 2277			<u> </u>
I	I BERNELSKE ANDERSON DER			1.=	. 						. —	i · —
==									=_	. === -		
									 			<u> </u>
·										<u>.:-</u>		
				T==					-			
								·				
<u> </u>		_ _			-					=	===-	===-
					19.75							
· 				==								
<u> </u>						=======================================			-			
===									—	<u>===</u>	<u>—</u>	_ _
-												
193 883 204-2270-					· 		 -			== <u>-</u>		:
				==		===		==				==-
**************************************				====					<u> </u>	_ _	=== <u></u>	
<u>:</u>					· ·	-	·					
<u> </u>			· · · · · · · · · · · · · · · · · · ·		: 							· ·
<u></u>								_===-				
	·		===		L.=			_		_		
	· 			: <u>-</u>		100000000000000000000000000000000000000				===		
· · ·				<u> </u>	==			-		=======================================		=======================================
===								 				
-		===				I =		=======================================			-	-
***************************************						==	*******			- -		
<u></u>				<u> </u>								
· · · · · · · · · · · · · · · · · · ·					==	· · · · · · · · · · · · · · · · · · ·		I =		::	 -	
<u>:</u>	-	. 						<u> </u>				
<u>:</u>				===-					= 			
<u></u>			:: _	_ <u> </u>	_			-		<u> </u>	· · · · · · · · · · · · · · · · · · ·	·
***********				 _	<u> </u>		<u> </u>		·	——		
<u> </u>	_ 					= =-		-		<u> </u>		
·												
	· -		· · · · · · · · · · · · · · · · · · ·			*				· · · · · · · · · · · · · · · · · · ·		· ===========
- 	<u> </u>	-							· · · · · · · · · · · · · · · · · · ·			
<u>:</u> -										-		
												
===		<u> </u>		==-				==-				
		 			22.777.72	<u>-</u>						
												7
===	· ·	***************************************	==	-	 				· · · · · · · · · · · · · · · · · · ·	· · · · · · · · · · · · · · · · · · ·		<u> </u>
						==						
							==					·
				***************************************		T==			=======================================			
			. 					· 			 	
					===	==						
:	==										= <u>-</u>	
												
===-		=	·	=-								
											The state of the s	
	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1					==						=_
: 	=	===										======================================
					: 	Transmission of the second		= <u>-</u>				
							·	= ==				
===		=	==========		-	==-						
	=======================================								-			
	· · · · · · · · · · · · · · · · · · ·											
		I == -	_		<u> </u>							.
· -	==-		 	-				-	===		===	==-
	===						===		-			
_												



```
// Utilities for arrays of elements
public String showElements(ModelElement[] elements, String nomsg) {
    boolean found = false;
    StringBuffer res = new StringBuffer();
    if (elements != null) {
        Index.getInstance().setCurrentRenderer(
            FlatReferenceRenderer.getInstance());
    for (int i = 0; i < elements.length; i++) {
            ModelElement el = elements[i];
            res.append(showElementLink(el)).append(HTML.LINE_BREAK);
            found = true;
        }
        Index.getInstance().resetCurrentRenderer();
    }
    if (!found &k nomsg != null &k nomsg.length() > 0) {
        res.append(HTML.italics(nomsg));
    }
    return res.toString();
}
```

Anzahl $\frac{Fehler}{Jahr}$ × Fehlerfolgekosten $\frac{PT}{Fehler}$

Anzahl $\frac{Fehler}{Jahr}$ × Fehlerfolgekosten $\frac{PT}{Fehler}$

Do Code Clones Matter?

Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, Stefan Waone Institut für Informatik. Technische Universität München Roltzmannstr 3 85748 Garchine h München German

changes to code displicates can lead to unexpected behavior. Consequently, the identification of displicated code, clone detection, has been a very active area of research in recens wars. Up to now, however, no substantial investigation of the consequences of code cloning on program correctives has been carried out. To remedy this shortcoming, this paper presents the results of a large-scale case study that was undertaken to find out if inconsistent charges to cloned code tection tool used in the case study implements a novel algorithm for the detection of inconsistent cloves. It is available s open source to enable other researchers to use it as busis

reasons: (1) multiple, possibly unnecessary, duplicates changes to cloned code can create faults and, hence, lead o incorrect program behavior 119, 281. While clone deter tion has been a very active area of research in recent years tion has been a very active area of research in recent years, up to now, there is no thorough understanding of the degree of hammfulness of code cloning. In fact, some researchers even started to doubt the harmfulness of cloning at all [16]. To shed light on the situation, we investigated the ef-

fects of code cloning on program correctness. It is important to understand, that clones do not directly cause faults but inconsistent changes to clones can lead to unexpected program behavior. A particularly dangerous type of change to cloned code is the inconsistent bug fix. If a fault was

tentionally and, (3) if unintentional inconsistencies can represent faults. In this case study we analyzed three commer cial systems written in C#, one written in Cobol and one open-source system written in Java. To conduct the study sessments were manually performed in the course of the use study. The study lead to the identification of 107 faults

acts of code clones on software quality" [28]. As th ous for software engineering research, education and prac-

found in cloned code but not theed in air clone instances, the system is likely to still eclobible the incorrect behavior. To illustrate this, Fig. 1 shows an example, where a missing sull-check was retrofited in only one clone instance. This paper presents the results of a large-scale case study that was understaten to find out (1) if clones are changed in-consistently, (2) if these inconsistencies are introduced in-

Research Problem Although most previous work agrees impacts or code contex on sutware quanty [20]. As the consequences of code cloning on program correctness, in particular, are not fully understood today, it remains unclear how harmful code clones really are. We consider the absence of a thorough understanding of code cloning precari

Contribution The contribution of this paper is twofold. First, we extend the existing empirical knowledge by a case study that demonstrates that clones get changed inconsiswe present a novel suffix-tree based algorithm for the detection of inconsistent clones. In contrast to other algorithms for the detection of inconsistent clones, our tool suite is

purposes, the more complicated average complexity would be more adequate. Thus, and to assess the performance of the entire pipeline we executed the detector on the source Figure 5. The settings are the same as for the main study min clone length of 10, max edit distance of 5). It is capuble to handle the 5.6 MLOC of Eclipse in about 3 hours

In order to gain a solid insight into the effects of incon sistent clones, we use a study design with 5 objects and 3 research questions that guide the investigation.

We chose 2 companies and 1 open source project as sources of software systems. This resulted in 5 analyzed projects in total. We chose systems written in different lam-grauges, by different teams in different companies and with different functionalities to increase the transferability of the study results. These objects included 3 systems written in C#, a Java system as well as a long-lived Cobol system. All these systems are already in production. For non-disclosur reasons we gave the commercial syst D. An overview is shown in Table 1.

Munich Re Group The Munich Re Group is one of the tially different functionality, ranging from damage predic tion, over pharmaceutical risk management to credit and uny structure administration. The systems support be

LV 1871 The Lebensversicherung von 1871 a.G. (LV 1871) is a Munich-based life-insurance company. The LV 1871 develops and maintains several custom software systems for mainframes and PCs. In this study, we analyze a mainframe-based contract management system written in Cobol (System D) employed by about 150 users.

Sysiphus The open source system Sysiphus⁵ is developed at the Technische Universität München (TUM) but none of the authors of this paper have been involved in the develributed software development projects. The inclusion of be externally replicated. This is not possible with the de-

Table 1. Summary of the analyzed systems								
System	Organization	Language	Age	Sire				
			(years)	(kLOC)				
A	Munich Re	Ca	6	317				
В	Munich Re	Cii	4	454				
C	Munich Re	Cil	2	495				
D	LV 1871	Cobel	17	197				
Sysiphus	TUM	Java	8	281				

5.2. Research question

The first question we need to answer is whether inconsistent clones appear at all in real-world systems. This not only means whether we can find them at all but also whether they constitute a significant part of the total clones of a system. It does not make sense to analyze inconsistent clones if they

RQ 2 Are inconsistent clones created unintentionally

is whether the developer is aware of the other clones, i.e.



2. Terms and definitions

ambiguity, we describe the terms as used in this paper. Code is interpreted as a sequence of units, which for ex

ample could be characters, normalized statements, or lines The reason to allow normalization of units at this stage, is that often pieces of code are considered equal even despi that often pieces of code are considered equal even despite differences in comments or naming, which can be leveled by the normalization. An exact close is then a (consecutive) substring of the code that appears at least twice in the (nor-malized) code. Thus our definition of a clone is purely syn-tactical, but catches exactly the idea of copy/dpaste, while of a potential failure inside the code. Defects are the super

to code cloning in recent years. The detailed surveys by Koschke [19] or Roy and Cordy [28] provide a comprehensive overview of existing work. Since this paper targets consequences of cloning and detection of inconsistent clones

again through later changes, potentially indicating a larger degree of independence of clones than hitherto believed. Geiger et al. 191 report that a relation between change cou plings and code clones could, contrary to expectations, not be statistically verified. Lozano and Wermelinger [26] rebility could be established

The effect of cloning on maintainability and correctness is thus not clear. Furthermore, the above listed publications suffer from one or more shortcomings that limit the trans-

- · Instead of manual inspection of the actual inconsistent ctones to evaluate consequences for maintenance and correctness, indirect measures¹ are used [1, 9, 22, 23, 26, 27]. Such approaches are inherently inaccurate and can easily lead to misleading results. For example, un-intentional differences and faults, while unknown to developers, exhibit the same evolution pattern as intentional independent evolution and are thus prone to misclassification.
- . The analyzed systems are too small to be represen-
- during creation [12, 25] or evolution [2] of clones, inhibiting quantification of inconsistencies in general.

buckets and perform pairwise comparison of subtrees in the same bucket. Jiang et al. [11] propose the generation of tree patterns that provide structural abstraction of subtree

are generated to identity cloned code.

Program Dependence Graph Krinke [21] proposes a search algorithm for similar subgraph identification. Komondoor and Horwitz [18] propose slicing to identify isomorphic PDG subgraphs. Gabel, Jiang and Su [8] use a odified slicing approach to reduce the graph isomor

The existing approaches provided valuable inspiration for the algorithm presented in this paper. However, none of them was applicable to our case study, for one or more of

- quire the availability of suitable context free grammars for AST or PDG construction. While feasible for modem languages such as Java, this poses a severe prob lem for legacy languages such as Cobol or PL/I, where suitable grammars are not available. Parsing such lan-guages still represents a significant challenge [5, 24].



Figure 2. The clone detection pipeline used

4. Detecting inconsistent clones

sistent clones in large amounts of code. Our approach works on the token level, which usually is sufficient for inding copy-pasted code, while at the same time being effi ient. The algorithm works by constructing a suffix tree of the code and then for each possible suffix an approximat search based on the edit distance in this tree is performed.

Our clone detector is organized as a nineline, which is ated code (recognized by user provided patterns). From the token stream, which consist of single keywords, identiflers, operators, and so on, the normalizer reassembles ements. This stage performs normalization, such that

The task of the detection algorithm is to find clones in This problem is related to the approximate string matching problem [13, 32], which is also investigated extensively in sioinformatics [30]. The main difference is that we are not

and 4. The algorithm is an edit distance based traversal of a suffix tree of our input sequence. A suffix tree over a equence s is a tree with edges labeled by words such that exactly all suffixes of s are found by traversing the tree from the root node to a leaf and concatenating the words on the sdaes encountered. Such a suffix tree can be constru

two parameters to this function are the sequence s we are working on and the position start where the search was

4.3. Post-processing and filtering

During and after detection, the clone groups that are re ported are subject to filtering. Filtering is usually performe ported are subject to filtering. Filtering is usually performed as early as possible, so no memory is wated with storing clone groups that are not considered relevant. Using these filters, we discard clone groups whose clones overlap with each other and groups whose clones are contained in other clone groups. Additionally, we enforce not only an absolute limit on the number of inconsistencies, but also a relative one, i. e., we filter clone groups where the number of inco certain amount. Moreover, we merge clone groups which efficiently we look at most at the first 1000 statements of share a common clone. While this leads to clone group efficiently we look at most at the first 1000 statements of the word w. As long as the word on the suffix tree edge is shorter, this is not a problem. In case there is a clone of more than 1000 statements, we would find it in chunks of 1000. We considered this to be tolerable for practical purwith non related clones (as our definition of an incon-

sistent clones, our algorithms and filters have been imple mented as part of CloneDetective' [14] which is based or ConOAT 171. The result is a highly configurable and ex found, and thus supports the rapid review of a large num

4.5. Scalability and performance

before reporting a clone. Including all of these optimizations, the algorithm can miss a clone either due to the thresholds (either too short or too many inconsistencies), or if it is covered by other clones. The later case is important, as each substring of a Due to the many implementation details, the worst car emplexity is hard to analyze. Additionally, for practical clone of course is a clone again and we usually do not want

clones are often not justified by different requirements but

project. Because both Sysiphus and our detection tools ar

open source, the whole analysis can completely be repl

cated independently. We provide a web site with the neces

vention by less cloning and (2) tools that prevent uninter

there are in the system, the less likely it is to introduce fault

veloper swareness of clones, we have integrated our clone detection tool into the Visual Studio development environ-ment⁸. At the Munich Re Group, as a reaction on the clone results, clone detection is now included in the nightly builds of all discussed projects. Furthermore, for existing clones, there should be tool support that ensures that all changes

that are made to a clone are made in the full knowledge of

its duplicates. Tools such as CloneTracker [4] or CReN [10

provide promising approaches. However, both approache

are not applicable to existing software that already contains inconsistent clones. Due to their high fault potential, we

ionally inconsistent changes of clones. The fewer clone

by inconsistencies between them. In order to increase de

can be explained by developer mistakes. We consider of special value the analysis of the Sysiphu

Studie

Munich Re #LV 1871

Über 100 Fehler in produktiver Software

Kritisch Nutzersichtbar Nicht nutzersichtbar

52% aller ungewollten Unterschiede fehlerhaft

We answer the research questions with the following study design. In the study we analyze sets of clone groups as shown in Fig. 6. The outermost set are all clone groups Cin a system, C denotes the set of inconsistent clone groups, and URC the unintentionally inconsistent clone groups. The

subset F of UIC consists of those unintentionally inconsis

ent clone groups that indicate a fault in the program. Please

note that we do not distinguish between created and evolved

listic systems. Hence, we need to analyze the size of set IC with respect to the size of set C. We apply our incon-

sistent clone analysis approach to all the systems, perform

nanual assessment of the detected clones to eliminate false

For RQ 2, whether clones are created unintentionary, we then compare the size of the sets UIC and IC. The sets are established by showing each identified inconsistent clone to developers of the system and asking them to rate them as intentional or unintentional. This gives us the aminten-

onally inconsistent clone ratio |UIC|/|IC|. The most im-

portant question we aim to answer is whether inconsistent

the size of set F in relation to the size of IC. The set F

lones indicate faults (RQ 3). Hence, we are interested in

sined by asking developers of the respective

tent clones as for the question of faultiness it does

The underlying problem that we analyze are clones and specially their inconsistencies. In order to investigate this question, we answer the following 3 more detailed re-

ivided by the logical lines of code of the inconsi We refrain from testing the hypothesis statistically because of the low number of data points as well as the large range

to detect inconsistent clones as described in section 4. For all systems, the detection was executed by the researcher to identify consistent and inconsistent clone candidates. On an 1.7 GHz notebook, the detection took between one and two minutes for each system. The detection was configured to not cross method boundaries, since experiments showed that inconsistent clones that cross method boundaries in

For the C# and Java systems, the algorithm was param eterized to use 10 statements as minimal clone length, a maximum edit distance of 5. a maximal inconsis to manual editing was excluded from clone detection, since inconsistent manual updates obviously cannot occur. No malization of identifiers and constants was tailored as appropriate for the analyzed language, to allow for ren of identifiers while at the same time avoiding too large fals

ndom samples of the detected clones have been evaluated The detected clone candidates were then manually rated

aconsistent and exact clone group candidates were treated differently: all inconsistent clone group candidates were rated, producing the set of inconsistent clone groups. Since the exact clones were not required for further steps of the ase study, instead of rating all of them, a random same

of 25% was rated, and raise positive rates then extrapolated to determine the number of exact clones.

The inconsistent clone groups were then presented to the developers of the respective systems in the tool CloneDetective mentioned in Section 4.4, which is able to display he commonalities and differences of the clone group in a learly arranged way, as depicted in Figs. 1 and 7. The developers rated whether the clone groups were created intentionally or unintentionally. If a clone group was created con-faulty. For the Java and C# systems, all inconsistent clone groups were rated by the developers. For the Cobol system, rating was limited to a random sample of 68 out of able for rating increased rating effort. Thus, for the Cobol case, the results for RQ 2 and RQ 3 were computed based ould not be determined, e.g., because none of the original were treated as intentional and non-faulty

The quantitative results of our study are summarized in Table 2. Except for the Cobol system D, the precision values are smaller for inconsistent clone groups than for ex-nect clone groups, as was expected, since inconsistent clone groups allow for more deviation. The high precision results of system D result from the rather conservative clone detec-tion parameters chosen due to the verbosity of Cobol. For system A, stereotype database access code of semantically

unrelated objects gave rise to lower precision values. About half of the clones (52%) contain inconsistencie Therefore, RQ I can be positively answered: Clones are erefore, RQ 1 can be positively answered: Clones are anged inconsistently. All these would not be reported by isting tools that search for exact matches. From these consistencies over a quarter (28%) has been introduced intentionally. Hence, RQ 2 can also be answered positively: Inconsistent clones are created unintentionally in many cases. Only system D is far lower here, with only 10% of unintentionally inconsistent clones. With about three quarters of intentional changes, this shows that cloning and changing code seems to be a frequent pattern during devel

tually presented a fault. Again the by far lowest number comes from the Cobol system. Ignoring it, the total ratio of faulty inconsistent clones goes up to 18%. This consti tutes a significant share that needs consideration. To judge hypothesis H, we also calculated the fault densities. The bypothesis H, we also calculated the fault densities. They like in the range of 3.4–9.14 faults per LDC. Again, system D is an outlier. Compared to reported fault densities in the range of 0.1 to 30 faults and considering the fact that all systems are not only delivered but even have been productive for several years we consider our results to support hypothesis H. On average the inconsistencies contain more faults than average code. Hence, R32 cast also be answered positively: Inconsistent clones can be indicators for faults in outle content.

faults are comparatively low for project D, which is a legacy system written in Cobol. To a certain degree, we attribute this to our conservative assessment strategy of treating inensistencies whose intentionality and faultiness could not be unambiguously determined as intentional and non-faulty Furthermore, interviewing the current maintainers of the systems revealed that cloning is such a common pattern in Cobol systems, that searching for duplicates of a piece of Cobol systems, that searching for duplicates of a piece of code is actually an integral part of their maintenance pro-cess. Compared to the developers of the other projects, the Cobol developers where thus more aware of clones in the system. To account for this difference in "clone aware-ness" we added the row |F|/|UIC| to Table 2, which reeals that while the rates of unintentional changes are lowe for project D, the ratio of unintentional changes leading to a fault is in the same range for all projects. From our results it eems that about every second to third unintentional chang o a clone leads to a fault

ategorizing the found faults as (1) faults that lead to po ential system crash or data loss, (2) unexpected behavior isible to the end user and (3) unexpected behavior not visvisible to the end user and (3) unexpected behavior not visible to the end user. One example for a category (1) fault is shown in Fig 1. Here, one clone of the affected clone group performs a multi-check to percent a full-pointed clone group performs and the clone clone of the affected clone group for the contract of ect and, hence, is handled differently by the undo mechaper last, tence, is handled differently by the undo inculta-nian. Further causation we found or calegory (2) fails us purpose. However, the earth fail and earth results are at small-incorrect and seer alreages, inconsistent defit by values as able to be because of prompting defect defect degrees. To check the control of prompting defect defect degrees. To check the control of prompting defect defect degrees. To check the control of prompting defect degrees and control of the control of the

loops and redundant re-computations of cache-able values differences in exception handling, different exception and debug messages or different log levels for similar cases. Of the 107 inconsistent clones found, 17 were categorized a category (1) faults, 44 as category (2) faults and 46 as ca

We discuss how we mitigated threats to construct, internal and external validity of our study.

have been introduced by incomplete changes to the system two reasons: (1) We want to analyze all inconsistent clones also the ones that have been introduced directly by copy and in the repository (2) The industrial systems do not have

in the repository. (2) The industrial systems do not have complete development histories. We conformed this threat by manually analyzing each potential inconsistent clone. The comparison with average fault probability is not perfect to determine whether the inconsistencies are really more fault-prone than a random piece of code. A comparison with the actual fault densities of the systems or actu

thing as faulty which is no fault. We mitigated this threat by only rating an inconsistency as faulty if the developer was completely sure. Otherwise it was postponed and the devel-oper consulted colleagues that know the corresponding part of the code better. Inconclusive candidates were ranked as intentional and non-faulty. Hence, again only the chance to answer the research question positively is reduced.

The configuration of the clone detection tool has a strong influence on the detection results. We calibrated the param

immersee on the detection results. We canteraced the param-eters based on a pre-study and our experience with clone detection in general. The configuration also varies over the different programming languages encountered, due to their differences in features and language constructs. However, this should not strongly affect the detection of inconsistent closes because was set to see a configuration of a not ines because we spent great care to configure the tool in

specting random code. As the potential benefit for the de

elopers is low, the motivation would be low and hence the

something as non-faulty which actually is faulty. This case

only reduces the chances to positively answer the research

hing as faulty which is no fault. We mitigated this threat by

questions. The second case is that the developers rate som

proc detect (s, ϵ) Ispat: String $s = (s_1, \dots, s_n)$, max odit distance ϵ

Figure 3. Outline of approximate clone detec

Let (w₁, ..., w_m) be the word along the edge leading to 2. Calculate the maximal length l ≤ m, such that there is a k ≥ j where the odd distance o' between

for each child node u of v do

search (s, whet, k + m, u, e - e'

else if k - start > minimal clone length then

report substring from atort to k of a us clon

poses. As each suffix we are running the search on will of course be part of the tree, we also have to make sure that no

When running the algorithm as it is, the results are often

few statements (how many is parameterized) to match ex-

actly. (This also speeds up the search, as we can choose the

without looking at all children.) The last statements are also

these to be reported.

results would be unreliable.

orrect child node at the root of the suffix tree in one step

or allowed to differ, which is checked for and corrected just

for each $i \in \{1, ..., n\}$ do

We also pre-processed the inconsistent clones that we presented to the developers in order to eliminate false posiives. This could mean that we excluded clones that are actually faulty. However, this again only reduces the chance

all possible software systems but we relied on our connec tions with the developers of the systems. Hence, the set of systems is not completely representative. The majority of he systems is written in C# and analyzing 5 systems in to tal is not a high number. However, all 5 systems have been developed by different development organizations and the CB-systems are technically different (2 web, 1 rich client) and provide substantially different functionalities. We fur-ther mitigated this threat by also analyzing a legacy Cobol

system as well as an open source Java system

In this paper we provide strong evidence that inconsis-In this paper we provide strong evidence that inconsis-tent clones constitute a major source of faults, which means that cloning can be a substantial problem during develop-ment and maintenance unless special care is taken to find and track existing clones and their evolution. Our results suggest that nearly every second unintentionally inconsistent change to a clone leads to a fault. Furthermore, we clones as well as suitable tool support for future experi-

rounce work on an stope with convert manager direc-ions. One obvious development is the refinement of the algorithms and tools used. This includes refined heuristics o speed up the clone search and perform automatic assess-ment to discard obviously irrelevant clones. In addition, the heir use more efficient for practical applications. More over, it will be interesting to compare different detection pa

#Fehler durch inkonsistente Klone

Daten aus Studie

- 3 Systeme von Munich Re analysiert
- 79 Fehler gefunden (Impact auf Funktionalität, nicht nur Wartbarkeit o.ä.)
- Systeme waren produktiv, einzelne Fehler schon durch Anwender als Tickets reportet
- 1 Produktionsfehler durch inkonsistente Klone / 17k SLOC

Bedeutung heute

- Betrachtetes Portfolio der Munich Re umfasst ca. 8,25 Millionen SLOC
- Konservative Annahme: Clone Management spart 1 Produktionsfehler pro 50k SLOC pro Jahr
- 8,25 Millionen SLOC / 50k = 165

Anzahl $\frac{Fehler}{Jahr}$ × Fehlerfolgekosten $\frac{PT}{Fehler}$

$$\frac{Fehler}{Jahr} \times \text{Fehlerfolgekosten } \frac{PT}{Fehler}$$

$$165 \frac{Fehler}{Jahr} \times Fehlerfolgekosten \frac{PT}{Fehler}$$

ØFehlerfolgekosten von Fehlern in Produktion

Mögliche Auswirkungen fehlerhafter Software

- Nutzer bekommen falsche Ergebnisse
- Anwendung stürzt ab
- Daten gehen verloren
- Frustration bei Nutzern (Kunden und Mitarbeiter)

Aufwand für Reparatur

- Nutzer schreibt Ticket für Fehler
- Debugging (Nachstellen, Diagnose, ...)
- Fixing
- Test
- Ggf. Deployment

\$ PT

\$ bl

ØFehlerfolgekosten von Fehlern in Produktion

Mögliche Auswirkungen fehlerhafter Software

- Nutzer bekommen falsche Ergebnisse
- Anwendung stürzt ab
- Daten gehen verloren
- Frustration bei Nutzern (Kunden und Mitarbeiter)

O PT: bewusste Unterschätzung

Aufwand für Reparatur

- Nutzer schreibt Ticket für Fehler
- Debugging (Nachstellen, Diagnose, ...)
- Fixing
- Test
- Ggf. Deployment

3 PT

$$165 \frac{Fehler}{Jahr} \times Fehlerfolgekosten \frac{PT}{Fehler}$$

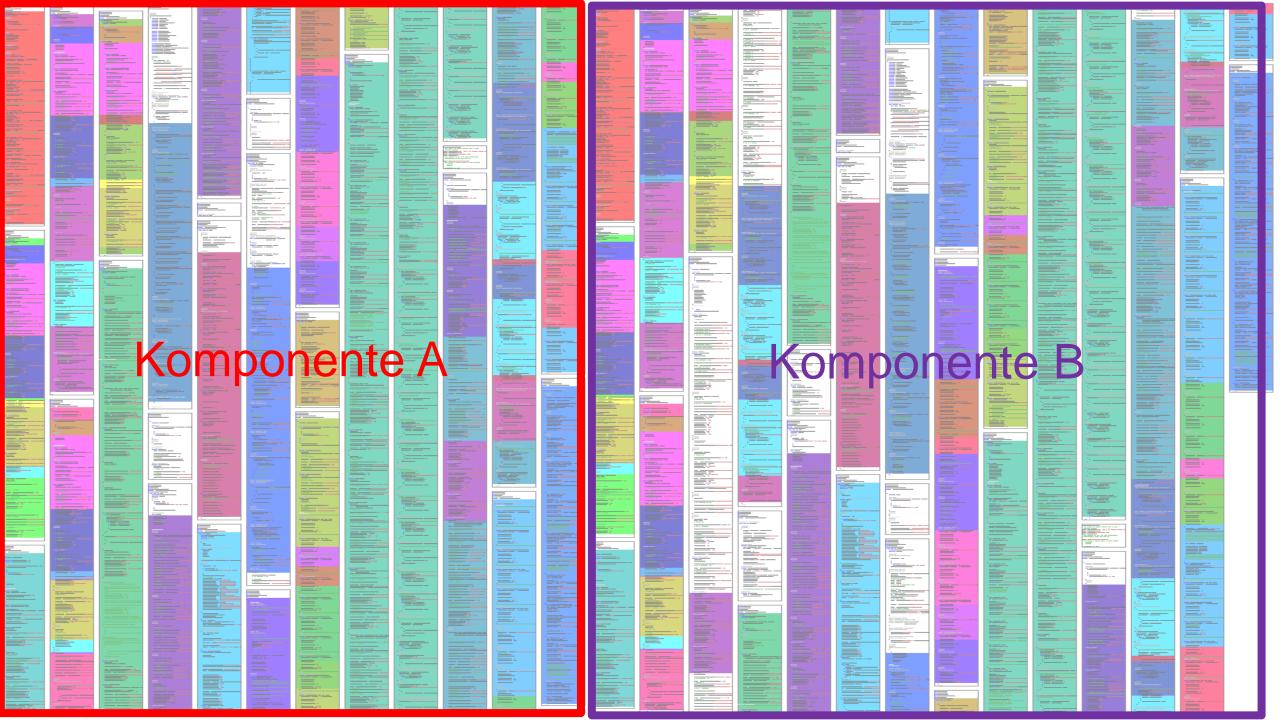
$$165 \frac{Fehler}{Jahr} \times 3 \frac{PT}{Fehler}$$

 $495 \frac{PT}{Jahr}$

$$500 \frac{PT}{Jahr}$$

Munich Re spart durch Einsatz von Clone Management jährlich ca. 500 PT Aufwand für Fehlerbehebung

								·						 _				<u></u>	===	=	=	=		<u>:</u>	<u>=</u>
<u> </u>				<u>=</u>		<u>=</u>							=	=	I I ≒ I			<u></u>							
	=			-				—					=	=					=						≣-
=	—				<u> </u>	_					=	<u>=</u>		<u>=</u>											
					<u> </u>	<u>=:</u>					= -		=				=						=		
									-																
	==			=		= -		÷		=	<u>=</u>									==					
		1 mm m	l ====	===	· <u>-</u> -	<u> </u>	= _				<u>=</u> _			1000											
<u></u>						===					= .								ļ <u>==</u>						=
===	= -		=		=								<u> </u>		<u> </u>			====	===-			=		<u> </u>	
<u>=</u>										<u>_</u>		=													
===						<u> </u>	·			- The second of	_		-									<u> </u>			
<u> </u>		3													<u> </u>	<u></u>		l <u>≔</u>							
=			<u> </u>			==						=					<u></u>						<u> </u>		
	<u> </u>					===			<u> </u>			<u> </u>	===				<u>=</u>	-			#-				
=									<u> </u>	==						=	==-		<u>=</u>						
						= -		- ÷	<u> </u>								=			<u>=-</u>					
																				=					
								=			= -									=					
<u>=</u>									=			<u>=</u> _		<u> </u>		<u> </u>			==_		7 -				
				<u> </u>							<u> </u>			l <u>≒</u>			==-					=			
									=		<u>=</u>					=				<u>=</u> -	- + -				
<u>=</u>		=-		=	==:									<u>-</u>			<u> </u>	===							
	-								■ ‡										<u>=</u>						
			===	-	=		=										-			<u> </u>	-÷-				
				-										==			. <u>—</u> .	: <u></u>							
= -		==		-								3	<u> </u>												
				· 			=		=			=	—		74										
===				-							==														
	<u> </u>			\equiv				<u></u>											= .						
			=																			y.=-			
			<u></u>	=	===			·		<u></u>								===	==-		3				
			l ==									<u>=</u> 3	<u> </u>		=====				= -	=					
	<u> </u>	=		<u> </u>												=	=						=-	= -	
						=						=						<u></u>							
		=						=	<u>=</u> +						=			==-				7/47/		<u> </u>	
= 7					=	<u> </u>									=										
																	=			=					
=7	=		=									=7				=									i
	<u></u>				==-														=			<u> </u>			
<u> </u>	-					<u>=-</u>		<u>_</u>				=-	-			=						<u> </u>			
<u> </u>						=						=							<u>=</u> -						
<u>=</u>												=				===					<u> </u>				
					==_		7 -																######################################		
					=-			=	#							====					— ——				
		·	=		===	<u></u>	<u> </u>		= '		=	=	=												
									===		**************************************			<u></u>		=					=		<u> </u>	<u> </u>	
			=			<u> </u>		=							=-				=-						
					=-	===										=				·		=:			
	=	=	=					=		=													=	=-	
	<u>_</u>												===						_						
							, 		===				===		===							_=			i .



		 -	<u>=</u> :							<u>==-</u>					<u> </u>						
<u>-</u>																					
- 11		<u> </u>	<u>=</u>	==						= -		===				1					
<u> </u>									<u>=</u> 7		—	====			=-	<u> </u>					
		<u> </u>											:	=		4					
		.==-	<u>=</u> :	=	-											4					
	-	·	<u>=</u> :	<u> </u>						== -											
				==		-			<u>=</u>	= -				=		1					
<u>=</u>			===		==-	*	=						<u> </u> =			4					
-	<u>'</u> ≡	7-	<u>=</u> :							=											
			≣-													1					
	: <u>=:</u>	 -	== -	-	====					<u> </u>					===	4					
≅ }						_=		Ī													
<u> </u>			==-		- V					=					<u> </u>						
						=							<u></u>		<u> </u>	4					
	. == ==	[[[]		<u>= </u>	<u> </u>											ı					
					<u> </u>				<u> </u>			<u>=</u>			==						
			7-			=				=					<u></u>						
		=	<u>:</u>		##-	=					X.	<u> </u>		<u></u>							
		=-			=		=	=-		<u>=</u> -			<u> </u>			A					
								===	<u> </u>			-	==			<mark>/</mark>					
			<u> </u>		=					<u> </u>						<mark>/</mark>					
						<u> </u>				<u>=</u> -						A					
									=						<u>=</u> -	<u> </u>					
				_								<u> </u>			_]					
				.=																	
<u> </u>			<u></u>	<u> </u>								<u>=</u>									
<u> </u>					·									<u> </u>							
		-						<u></u>								1					
					=											#					
					<u> </u>			;·					=			=					
=	-							<u> </u>		<u> </u>				=====		 ≢					
			·													4					
==			_ <u>=</u>						=-		=		=			<u>.</u>					
			<u> </u>						. 			= -				<u>:</u>					
		<u>=</u> -							 		2					Ē					
	<u> </u>							<u>======</u>								1					
·		= .								<u>`</u>				===		╛					
=					<u>= </u>	==			<i> </i> =	-	= <u>-</u>					-					
=														=-							
			=	<u> </u>										===							
-										=		<u>:==</u> -		<u>=</u>							
						=		<u></u>						=-							
<u></u>	<u> </u>						<u> </u>							<u>=</u>							
						=	<u>=</u>														
			<u>=</u>						<u> </u>	<u> </u>											
						=															
<u> </u>						<u> </u>								<u> </u>							
						WEET.	===			= -	2			<u> </u>							
		=-				<u>=</u>						7.22.		=-							
			- I		<u> </u>						7/-										
=		<u>=</u> -	<u> </u>				-		<u>=</u> -					≡							
	.====					===								= -							
					<u> </u>		<u> </u>					-									
		=-					-				+			=							
1						·	·=-				=	<u>;</u>	=								

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 unts of cloned code. While the negative impact of used to assess the economic impact of cloning in a system and to evaluate investments in clone management tool upport. To show its applicability, we report on a case study that instantiates the cost model for 11 industrial

1. Introduction

Code cloning abounds in real world software. Numerou tudies report substantial amounts of cloning in open source and industrial systems [26, 32]. To name just a few, sign ind moustrial systems [26, 32]. To name just a rew, sij icant cloning was detected in GCC [14], X Windows Linux and JDK [30]. Although the extent varies betw systems, cloning occurs across programming languages, de-velopment contexts and application domains. Furthermore, t 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 e considered as a phenomenon that occurs across differen software artifacts.

Substantial research effort on software clones has established the negative impact of cloning on software main nance activities in general [17]. It, often unnecessarily, inses code size and thus effort required for size-relate creases 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, incon-sistencies 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 While the negative consequences of cloning have firmly been established qualitatively, their quantitative impact re-mains 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 compet ing factors that can be quantified, such as open bug issue or change requests. As long as we do not know the cos ing causes, clone control is prone to be neglected-though cloning could be the root cause, and open but

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

established qualitatively, the economic impact of cloning on maintenance is poorly understood. Consequently, we lack the foundation to assess the economic harmf of cloning and to evaluate alternative clone management

Contribution We propose an analytical cost model to es timate the impact that code cloning has on software mainte nance efforts. We present a case study that instantiates the cost model for 11 industrial software systems and estimat mintenance effort increase and potential benefits achiev

2. Terms & Definition

Clones are regions of similar code. In the literature, dit nt definitions of similarity are employed [26,32], mostly based on syntactical characteristics. In this paper, we re quire clones to be semantically similar in the sense that they implement one or more common concepts. This redundar entations gives rise to change coupling 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 similar-ity is not required for change coupling, existing clone detec-Hence, we employ the term clone to denote syntactically similar code regions that contain redundant implementation of one or more concents. A clone group is a set of clones Clones in a single group are referred to as sibling

We use the term fullure to denote an incorrect output of software visible to the user. A fault is the cause of a po

tential 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 rethe size of a system from which all cloning is perfectly re-moved. 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 IPSS remains at 100. Overhead denotes the ratio by which a sys-tem's size has increased due to cloning. It is computed as $\frac{33}{1000} - 1$. For the above example, the resulting overhead is \$\frac{\sqrt{61}}{RF35} - 1\$. For the above example, we consider the state of the s

3. Maintenance Process

This section introduces the software maintenance pro cess on which the cost model is based. It aualitatively de scribes the impact of cloning for each process activity and discusses potential benefits of clone management tools. The seess is loosely based on the IEEE 1219 standard [18] the describes the activities carried on single change requests (CRs) in a waterfall fashion. The succesive execution of ac-tivities that, in practice, are typically carried out in an intercaved 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 plementation 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.

Location (L) determines a set of change start points. It Document (L) determines a set of change start points. It has 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 modifi-cations 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 re not aware of tool support to alleviate the conse

well as the software system and its documentation to de-sign 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 Impact Analysis (1A.) 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 dantly in multiple locations, all of them need to b recurionantly in minuspic recursors, air of timen needs to see changed. Closting 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 changes to source code: Adminior and new source code
to the system without changing existing code. Modifications after 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

We assume that adding new code is unaffected by cloning in existing code. Implementation is still affected by cloning since modifications to cloned code need to be ner formed multiple times. Linked editing tools could, ideally reduce effects of cloning on implementation to zero.

Quality Assurance (QA) comprises all testing and in tion satisfies the change request. We assume a smart qua tion satisfies the change request. We assume a smart qual-ity 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 assur-

We differentiate two effects of cloning on quality assur-We differentiate two effects of cloning on quality assur-nace; 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

livery and deployment, user support or change con meetings. Since code does not play a central part in these activities, they are not affected by cloning

This section introduces a detailed cost model that qu tifies 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 Annmach

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

$$E = \sum_{r \in CR} e \; (cr)$$
 scope of the cost model is determined by the popu-

lation 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 life-time 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 or C CR is exsed by e(cr). It is the sum of the efforts of the ind

$e = e_A + e_L + e_D + e_B + e_{bigl} + e_{Ql} + e_D$

In order to model the impact of cloning on maintenance on outer to moter the impact of coming on immunerance efforts, we split e into two components: $inherent effort e^{i}$ and eloning $induced overhead e^{e}$, e^{i} is independent of cloning. It captures the effort required to perform an ac-tivity on a hypothetical version of the software that does not contain cloning. e., in contrast, captures the effort penalty caused by cloning. Total effort is expressed as the sum of

The increase in efforts due to cloning, Δe , is captured

by $\frac{e^2+e^2}{2\Gamma} = 1$, or simply $\frac{e^2}{2\Gamma}$. The cost model thus expresses cloning induced overhead relative to the inherent effort repuired to realize a change request. The increase in total naintenance efforts due to cloning, ΔE , is proportional to the average effort increase per change request and thus cap-tured by the same expression.

4.2. Activity Models

The activities Analysis, Design, and Other are not imed by cloning. Their cloning induced effort, e ", is thus ero. Their total efforts hence equal their inherent effort

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

 $e^{\frac{1}{2}} = e^{\frac{1}{2}} \cdot overbead$

Impact analysis effort depends on the number of change impact analysis error depension the number of change points that need to be determined. Cloning increases the number of change points. We assume that $e_{f,h}^{*}$ is pro-portional to the cloning-induced increase in the number of source locations. This increase is captured by overhead:

 $e_{IA}^{c} - e_{IA}^{i}$ overhead

ification effort: $e_{Impl} = e_{Impl_{dist}} + e_{Impl_{dist}}$. We assume that effort required for additions is unaffected by cloning in existing source code. We assume that the effort required for nodification is proportional to the amount of code that gets nodified, i.e., the number of source locations determined by impact analysis. Its cloning induced overhead is, conse sently, affected by the same increase as impact analysis

quernsy, attector by the same interacte as impact intripose. $e^{-}_{log} = e^{-}_{log} e^{-}_{log}$, overhead. The modification ratio mod captures the modification-related part of the inherent implementation effort: $e^{-}_{log} log = e^{-}_{log} log = e^{-}_{$

Quality Assurance effort depends on the amount of code on which quality assurance gets performed. Both modifirations and additions need to be quality assured. Since the querhood cantures size increase of both addition and modifications, we do not need to differentiale 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:

 $e_{ou}^{i} = e_{ou}^{i} \cdot overhead$

4.3. Maintenance Effort Increase Model

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

 e^{c} = overhead \cdot (e_{L}^{i} + e_{D}^{i} + e_{Dol}^{i} · mod + e_{O1}^{i})

The relative cloning induced overhead is computed as

$$\Delta e = \frac{overhead \cdot (e_L^i + e_R^i + e_{Bq\ell}^i \cdot mod + e_{QA}^i)}{e_A^i + e_L^i + e_D^i + e_{Br}^i + e_{Br\ell}^i + e_{QA}^i + e_D^i}$$

in maintenance costs caused by cloning. It does not take consequences of cloning on program correctness into ac count. 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

that, independent of the quality assurance technique, the effort required to detect a single fault in a system depends rimarily on its fault density. We furthe verage fault removal effort for a system is independent of its size and fault density. These assumptions allow us to reon about the number of remaining faults in similar system son 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 fault can thus be detected and fixed per unit. For two systems A and B, with B having twice the size and available OA effort, we expect a similar reduction of fault density. How ever, since B is twice as big, the same fault density means

activities: fault detection and fault removal. We assume

ever, since is a twee at any, are samining faults.

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

This reasoning assumes that developers are completely gnorant of cloning. That is, if a fault is fixed in one clone not immediately fixed in any of its siblings. Instead faults in siblings are expected to be detected independent Empirical data confirms that inconsistent bue fixes do fre ently occur in practice [22]. However, it als ing entirely consistent or entirely inconsistent evolution is

In practice, a certain amount of the defects that are de cted in cloned code are hence fixed in some of the sibling lones. 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 progr yond activity effort increase captured

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. Exameles for clone indication tools include ConOAT and Clone acker [13]. Optimal clone indication thus lowers the ef-

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

Linked Editing replicates edit operations performed on one clone to its siblings. Prototype linked editing tools in-clude Codelink [35] and CReN [19]. Optimal linked editing tools thus lowers the overhead required for consistent mod fications of cloned code to zero. Since linked editors ty ically also provide clone indication, they also simplify im pact analysis. Their application yields the following model

 $\Delta e = \frac{e^{\frac{i}{A}} + e^{\frac{i}{L}} + e^{\frac{i}{D}} + e^{\frac{i}{D}} + e^{\frac{i}{D}} + e^{\frac{i}{D}} + e^{\frac{i}{D}}}{e^{\frac{i}{A}} + e^{\frac{i}{D}} + e^{\frac{i}{D}}}$

We do not think that clone management tools can substantially reduce the overhead cloning causes for quality as surance. If the amount of changed code is larger due to surance. If the amount of changed code is larger due to cloning, more code needs to be processed by quality assur-ance 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 lunk 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 ca 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 nerally applicable than the detailed model, it is easie

to apply.

Due to its number of factors, the detailed model require 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 typi cally interleaved, without clear transitions between them, is

spent on location and how much on impact analysis.

The individual factors of the detailed model are required o make trade-off decisions. We need to distinguish be

tween clone management alternatives however, a simple decision needs to be taken: whether to do anything abo cloning at all. Only then is it reasonable to invest the effor o determine accurate parameter values. If the cost mode is not employed to assess clone management tool suppor ny of the distinctions between different factors are obs

difficult to get exact estimates on, e.g., how much effort is

lete. 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:

$\Delta e = overhead \cdot \frac{e \sum_{i} + e \sum_{k} + e \sum_{lipl} \cdot mod + e \sum_{Qk}^{i}}{e \sum_{i} + e \sum_{lipl} \cdot mod + e \sum_{lipl}^{i}}$

comprehension ($e^{i}_{L} + e^{i}_{Lb}$), modification of existing code ($e^{i}_{ligl} \cdot mod$) and quality assurance (e^{i}_{Qb}) w.r.t. the entire effort required for a change request. We introduce the new parameter cloning-affected effort (CAE) for it

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

6. Instantiation

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

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

6.1. Parameter Determination

This section describes how the nummeter values can be mined to instantiate the cost model in practic

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 onQAT. It is computed as explained in Section 2. Our ireal captures cloning induced size increase independent of whether the clones can actually be removed with means of whether the colors call actually be removed with means or 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. Unfortu nately, many existing clone detection tools produce high alse positive rates: Kanser and Godfrey [23] report b 7% and 65%. Tiarks et al. 1341 up to 75% of false positive 27% and 65%. Turks et al. [54] 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 exhibited the properties of the complete software maintenance and must be exhibited from the control of th

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 be tailored. Tailoring removes code that is not maintained manually, such as generated or unsured codes, inset it does not impede maintenance. Exclusion of generated codes is important, since generates typically produce similar-looking filts for which large amounts of clones are descread. Furthermore, tailoring adjusts detection to that false pos-tives due to overly aggressive momulations are roulded. This is necessary so that e.g., regions of Java getrier, that differ in their identifiers and have no conceptual relation ship, are not erroneously considered as clones by a detec tor that ignores identifier names. According to our expe ence [22], after tailoring, clones exhibited change

naintenance efforts depends on many factors, including the naintenance process employed, the maintenance environ ment, the personnel and the tools available 1331. To receive occurate results, the parameters for the relative efforts of the individual activities thus need to be determined for each

Coarse effort distributions can be taken from project nalysts w.r.t. all wages. As we cannot expect enginee oles to match the activities of our maintenance proce actly, we need to refine the distribution. This can be done y observing development efforts for change requests to determine, e. e. how much effort analysts spend on anal sis location and design respectively. To be feasible, such

observations need to be carried out on representative sar ples of the engineers and of the change requests. Stratified ampling can be employed to improve representativenes of results-sampled CRs can be selected according to the e type distribution, so that repre-ctive and other CRs are analyzed. perfective and other CNs 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 snent exclusively developing new code. This can, again, be done by observing developers working on CRs. The modification ratio can, in principle, also be deter-nined by observing developers and differentiating between

se estimated from change request type statistics

to instantiate the model. Unfortunately, the research com-munity 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 ation based on which a decision can be take

across maintenance activities. Rombach et al. [31] repor 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 20,000 notes of error. Both statistics were on tand 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 dis-tributed 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 analy sis and location. In 161, analysis subsumes location. The estimated average efforts are depicted in the fourth row of Table 1. Since the definitions of implementation, analise as arance and other are similar between the studies and our surunce 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 litera-ture 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 distribu tion effort during impermentation, we inspect the distribu-tion of change request types. We assume that adaptive, cor-rective and preventive change requests mainly involve mod-ifications, whereas perfective changes mainly involve addi-tions. Consequently, we estimate the ratio between addi-tion and modification by the ratio of perfective w.r.t. all other change types. Table 2 shows effort distribution acros change types from the above studies. The fourth row depicts the median of all three-37% of maintenance effort re spent on perfective CRs, the remaining 63% are disributed 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%

This section presents the application of the clone cost nodel to several large industrial software systems to quan-ify the impact of cloning, and the possible benefit of clone rement 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 tems, and the possible benefit of the application

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

eloners to contribute in clone detection tailoring, we had to rely on our contacts with industry. However, we chose systems from different domains (finance, content manage ment, convenience, power supply, insurance) from 7 dif-ferent companies written in 5 different programming lan-guages to capture a representative set of systems. For nonguages to capture a representative set of systems. For non-disclosure reasons, we termed the systems A-K. Table 3

Study Design and Procedure Clone detection tailoring was performed to achieve accurate results. System develop-ers 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 con-servative minimal clone length. Since the effort parameters are not available to us for the analyzed systems, we employed values from the litera-

ture. 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 mainte ance effort are affected by cloning. To estimate the impac

Results and Discussion. The results are denicted in Results and Discussion The results are depicted in Table 3. The columns show lines of code ($kLOC_s$), 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 _{lock}). 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 system The estimated overhead ranges from 75%, for system A, to 5.2% for system F. We could not find a significant correl tion between overhead and system size. On average, esti-mated maintenance effort increase is 20% for the analyze . The median is 15.9%. For a single quality 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

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

System	Language	kLOC	kss	KRFSS	overhead	ΔE	ΔE_{Tool}
A	XSLT	31	15	6	150.0%	75.0%	67.5%
В	ABAP	51	21	15	40.0%	20.0%	18.0%
C	CE	154	41	35	17.1%	8.6%	7.7%
D	Cn	326	108	95	13.7%	6.8%	6.2%
E	Cn	360	73	59	23.7%	11.9%	10.7%
F	CE	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%

Table 3. Add caption

7.1. Challenges of project diversity

Many factors influence maintenance productivity 15.7. 331: the type of system and domain, development process ble tools and experience of developers, to name just a available tools and experience of developers, to name just a few. Since these factors way substantially between projects, they need to be reflected by cost estimation approaches to achieve accument 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,

terms of costs, and since most factors influencing mainte deliberately chose a relative cost model to keen its number of parameters and involved instantiation effort at but

The cost model is based on a series of assumptions. It

roportional to the amount of code that gets inspected dur-ing location. In some situations, activity cost models might

be more complicated. For example, if an activity has a high meets seemp costs, me cost mouer snoulei incrusee a meet au-tor; diseconomy of scale could increase effort w.r.f. 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 experience from large scale application of clone detection in industrial contexts [11, 21, 22], if clone detection is tailored appropr ately. In case clones are uncounled. . e. e., because they are

group requires the same amount of effort. We ignore that quent 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 clon groups have size 2. The inaccuracy introduced by this sim plification should thus be moderate

8. Related Work

work on the consequences of code cloning and cost models

8.1. Consequences of Cloning

contexts of clones, such as missing if statements. Both papers report the successful discovery of bugs in released oftware. In [1] and [3], individual cases of bugs or inconsistent bug fixes discovered by analysis of clone evolution e reported for open source software. In earlier work [22], e inspected inconsistent clones together with the developrs of the analyzed systems. In four open source and four industrial systems, analysis revealed 107 faults due to unin-tentionally inconsistent changes to cloned code. The above studies give strong indication that that cloning impacts cor-rectness 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 trong indication that a substantial amount of the modifica arong materiors that a substantial amount of the mounter-tions to cloned code are coupled in many systems in prac-ice, and thus cause additional effort for location and con-istent modification.

The above work establishes qualitative relationships between cloning and maintenance efforts that provide the foundation of our cost model. It does, however, not allo to awartify impact of cloning on development efforts. In ntrast, the cost model presented here allows to estin mpact 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 mount of clones evolve independently [27] and that stability of cloned code was found to be, contrary to expe tions, similar or even higher than stability of non-closed tations, similar or even higher than stability of non-cloned code [28]. Gide shows that clone evolution patters vary between different systems [16]. While these studies em-phasize 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 sev eral reasons. First, data accuracy is unknown-clone detectors frequently produce large amounts of false positives that can dilute conclusions. Furthermore, evolution does not ecessarily represent intent: many changes are unintention ally inconsistent [22]; inspecti

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 consideration

A substantial number of approaches for software cost A substantan insmer or approaches for software cost estimation have been proposed. Besides badgeting and project planning & controls, they can in principle to a pilical for software improvement insententent analysis, such as a cg., tool selection of reneglineering. The survey by Bochm, Abes and Cultural gives a datalied overview of ex-isting approaches [7], including cost model based ones. Among the most undergreated cost models are COCCOMO and its successor COCOMO II [5]. Besides general purpose cost estimation models, some research has evaluated the consequences of individual quality characteristics on main tenance costs. Banker et al. investigate code complexity [4] Lanning and Khoshgoftaar [29] examine coupling and Bril

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

et al. evaluate architecture quality [8].

This paper proposes an analytical cost model to quan tify the economic effect of cloning on maintenance effort It can be used as a basis to evaluate clone management al-ternatives. Instead of computing absolute costs, the mode computes maintenance effort increase relative to a system without cloning. Since in a relative cost model many fac

sonable effort in practice. We have instantiated the cost model on 11 industrial sys tems. Although result accuracy could be improved by using project specific instead of literature values for effort param eters, the results indicate that cloning induced impact varie significantly between systems and is substantial for som

able savings by performing active clone management.

There is a definitive need for future work in this area and the conceptance of code classing. A survey is given by Herdigi et al. [17]?

Inspired of clossing on program conventions in the subject of clossing on program conventions in the subject of convent surface. It and [10] present an approach to de-tor-buy based on inconsistent resuming of desirations become classical survey. The convention of the survey of of t 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 rtantly, we need to validate the correctness of the r

[1] L. Aversano, L. Cerulo, and M. Di Penta. How clones are

(6) V Barit I British S Condon V M Vim W I Malo

of software maintenance release. In Proc. of the Interna-tional Conference on Software Engineering (ICSE), pages

st estimation approaches - a survey. Ann. Softs: Eng.

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 qual-

[15] M. F. B. H. B. S. S. W. C. D. J. S. Elmar Juergens, Flo-

gens. Flexible architecture confo

[12] C. Domann, E. Juergens, and J. Streit.
 [13] C. Domann, E. Juergens, and J. Streit.

naintained: An empirical study. In Proc. CSMR '07. IEEE,

References

 Blail. Standard 1219 be software marineaines. Standard, BEE, 1998.
 P. Jahnson and D. Hou. CREW, and for tracking copy-plants and D. Hou. CREW, and for tracking copying the IDE. In Proc. Englaw 'D. ACM, 2007.
 L. Jiang, Z. Su, and E. Chin. Context-based detection of close-related by the Theory of the Context-based detection of close-related byte. In Proc. ESCH.2-FSF '07' ACM, 2007.
 E. Juergens, F. Deissenboeck, and B. Hummel. Closerbective A workbench for close detection research. ofter clone consolidation with the predicted efforts

[18] IEEE, Standard 1219 for software maintenance, Standard

 B. S. Baker. On finding duplication and near-duplication in large software systems. In Proceedings of WCRE 1995.
 T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In Proc. ICSM '07. IEEE, 2007.
 R. D. Bunker, S. M. Datar, C. F. Kennerer, and D. Zweig. [24] C. Kapser and M. W. Godfrey. "Cloning considered ham ful" considered harmful. In Proc. WCRE '96. IEEE, 2006.

[25] M. Kim, V. Sazawal, D. Notkin, and G. Murn cal study of code clone genealogies. In Proc. ESEC/FSE-1.

ACM, 3, 1993.

[5] Barry, E. Horowitz, R. Madachy, D. Reifer, B. K. Clark, B. Steece, W. A. Brown, S. Chulani, and C. Abts. Soft-

cal study of code charge genealogies. In Princ. InSIAT-19-10-2, ACM, 2005. [26] R. Koschkie. Survey of research on software clones. In Dis-plication, Redundancy, and Similarity in Software, Dagstabl Seminar Proceedings, 2007. [27] J. Kritike, A study of consistent and inconsistent changes to code clones. In Proc. WCRE '07, 1828, 2007. [28] J. Kritike, is closed code more table than non-cloned code?

SCAM'08, 2008.
[29] D. Lunning and T. Khoshgoftaur. Modeling the relationship 464-474 IEEE CS Press, 1996.

[7] B. Boehm, C. Abts, and S. Chulani. Software development

[30] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Fine

[8] R. Bril, L. Feiis, A. Glas, R. Krikhaur, and M. Winter. Main IEEE TSE, 2006.

H. D. Rombuch, B. T. Ulery, and J. D. Valett. Toward full taining a legacy: towards support at the architectural level. Journal of Software Maintenance, 2000.
[9] F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juerlife cycle control: Adding maintenance measurement to the SEL. J. Syar. Softw., 18(2):125–138, 1992.
2 C. K. Roy and J. R. Cordy. A survey on software clone de-tection research. Technical Report 541, Queen's University

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 at Kingston, 2007.
[33] H. Sneed. A cost model for software maintenance & evolu-

tion. In International Conference on Software Maintenan (ICSM). IEEE CS Press, 2004. 4] R. Tiarks, R. Koschke, and R. Falke. An assessment of typ

[35] M. Toomim, A. Berel, and S. L. Graham, Managing dupli cated code with linked editing. In VLHCC '04, 2004.

[36] D. Yeh and J.-H. Jeng. An empirical study of the influence of ESEM '09, 2009.
 E) Dulai-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In Proc. ICSE '07, IEEE, 2007.
 S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In Proceedings of the Computer Section (In Proceedings of the Computer Section 1). departmentalization and organizational position on software maintenance. J. Softse Maint. Evol. Res. Ps., 14(1):65–82 2002.

sessenti of repairments operficiation! In Accordad for the Congress of the Con

an came detection support quality as-ents specifications? In Accepted for

KLOC.	8.55	KRESS	overnesa	-AE	AMS Tool
31	15	6	150.0%	75.0%	67.5%
51	21	15	40.0%	20.0%	18.0%
154	41	35	17.1%	8.6%	7.7%
326	108	95	13.7%	6.8%	6.2%
360	73	59	23.7%	11.9%	10.7%
423	96	87	10.3%	5.2%	4.7%
461	208	155	34.2%	17.1%	15.4%
657	242	210	15.2%	7.6%	6.9%
1,005	400	224	78.6%	39.3%	35.4%

The assessment of the impact of cloning differs from the 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 nance productivity remain constant in both settings, they do not need to be contained in our cost model. In a nutshell, we

system	Language	KLOC	kss	KRFSS	overhead	ΔE	ΔE_{Tool}
A	XSLT	31	15	6	150.0%	75.0%	67.5%
В	ABAP	51	21	15	40.0%	20.0%	18.0%
C	CE	154	41	35	17.1%	8.6%	7.7%
D	Cn	326	108	95	13.7%	6.8%	6.2%
E	Cn	360	73	59	23.7%	11.9%	10.7%
F	CE	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%
1	Cobol	1,005	400	224	78.6%	39.3%	35.4%
J	Java	1,347	368	265	38.9%	19.4%	17.5%

such effort is unavoidable.

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

K Java 2,179 733 556 31.8% 15.9% 14.3%

This section relates the proposed cost model to existing

 Δ Aufwand = %BlowUp × %CloneAffectedEffort

 Δ Aufwand = %BlowUp × %CloneAffectedEffort

Blow-Up: 0%



20 Lines

20 Lines

Blow-Up: 50%



20 Lines

20 Lines

20 Lines

 Δ Aufwand = $\%12 \times \%$ CloneAffectedEffort

 Δ Aufwand = %12 × %CloneAffectedEffort

%CloneAffectedEffort

Aktivitäten

Analysis

Location

Design

Impact Analysis

Implementation

Quality Assurance

Other

Aufwändiger durch Cloning

_

Location

-

Impact Analysis

Implementation

Quality Assurance

_

Detaillierte Herleitung und Berechnung im Paper.

Wert für Berechnung: 51%.

 Δ Aufwand = %12 × %50

 Δ Aufwand = %12 × %50 = 6%

Die Munich Re setzt Clone Management seit ca. 10 Jahren ein.

Wie sähe es ohne aus?

Continuous Software Quality Control in Practice

Daniela Steidl*, Florian Deissenboeck*, Martin Poehlmann*, Robert Heinke[†], Bärbel Uhink-Mergenthaler[†] * CQSE GmbH, Garching b. München, Germany Munich RE, München, Germany

Abstract-Many companies struggle with unexpectedly high maintenance costs for their software development which are often caused by insufficient code quality. Although companies often use static analyses tools, they do not derive consequences from the metric results and, hence, the code quality does not actually improve. We provide an experience report of the quality consulting company CQSE, and show how code quality can be improved in practice; we revise our former expectations on quality control from [1] and propose an enhanced continuous quality control process which requires the combination of metrics, manual action, and a close cooperation between quality engineers, developers, and managers. We show the applicability of our approach with a case study on 41 systems of Munich RE and demonstrate its impact.

1. INTERDUCTION

Software systems evolve over time and are often maintained for decades. Without effective counter measures, the quality of software systems gradually decays [2], [3] and maintenance costs increase. To avoid quality decay, continuous quality control is necessary during development and later maintenance [1]: for us, quality control comprises all activities to monitor the system's current quality status and to ensure that the quality meets the quality goal (defined by the principal who outsourced the software development or the development team itself).

Research has proposed various metrics to assess software quality, including structural metrics1 or code duplication, and has led to a massive development of analysis tools [4]. Much of current research focuses on bester metrics and better tools [1], and mature tools such as ConQAT [5], Teamscale [6], or Sonar2 have been available for several years.

In [1], we briefly illustrated how tools should be combined with manual reviews to improve software quality continuously, see Figure 1: We perceived quality control as a simple, continuous feedback loop in which metric results and manual reviews are used to assess software quality. A quality engineer a representative of the quality control group - provides feedback to the developers based on the differences between the current and the desired quality. However, we underestimated the amount of required manual action to create an impact. Within five years of experience as software quality consultants in different domains (insurance companies, automotive manufacturers, or engineering companies), we frequently experienced that tool

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant EvoCon, 01IS12034A. The responsibility for this article lies with the authors.



Fig. 1. The former understanding of a quality control process

support alone is not sufficient for successful quality control in practice. We have seen that most companies cannot create an impact on their code quality although they employ tools for quality measurements because the pressure to implement new features does not allow time for quality assurance: often, newly introduced tools get attention only for a short period of time, and are then forgotten. Based on our experience, quality control requires actions beyond tool support.

In this paper, we revise our view on quality control from [1] and propose an enhanced quality control process. The enhanced process combines automatic static analyses with a significantly larger amount of manual action than previously assumed to be necessary: Metrics constitute the basis but quality engineers must manually interpret metric results within their context and turn them into actionable refactoring tasks for the developers. We demonstrate the success and practicability of our process with a running case study with Munich RE which contains 32 .NET and 9 SAP systems.

II. TERMS AND DEFINITIONS

- · A quality criterion comprises a metric and a threshold to evaluate the metric. A criterion can be, e. g., to have a clone coverage below 10% or to have at most 30% code in long methods (e.g., methods with more than 40 LoC).
- · (Quality) Findings result from a violation of a metric threshold (e.g., a long method) or from the result of a static code analysis (e.g., a code clone).
- · Quality goals describe the abstract goal of the process and provide a strategy how to deal with new and existing findings during further development: The highest goal is to have no findings at all, i.e., all findings must be removed immediately. Another goal is to avoid new findings, i.e., existing findings are tolerated but new findings must not be introduced. (III-B will provide more information).

III. THE ENHANCED QUALITY CONTROL PROCESS

Our quality control process is designed to be transparent (all stakeholders involved agree on the goal and consequences

but lifficult for history of site. my reliable opers ems in case quently, we ole systems rather than bone For these the before and

cient

either

eam-

OSE

eeds.

ulity

sting

ment

cours

WEYS

[7D.

ınder

iteria

code

ngth,

cture

teria

ancy.

and

nized

ng methods. bese quality contrast to, tomatically nd. Second both trends All trends

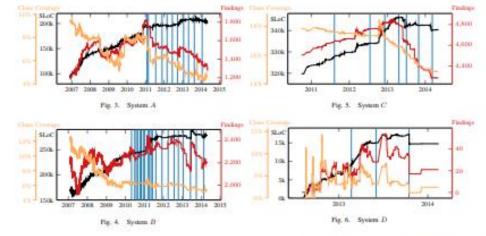
velopers

stems with long-term om size (to he behavior Il QG3, we igures 3-6 ndings trend e coverage The quality e for each ort date.

reat impact size grows control, the gs declines um of 5%. measurably

> However. ooperation e 2006, in verage had decreases

on the history ence, although Teamscale to



continuously in the available history (Figure 4). The number of findings, however, increases until mid 2012. In 2012, the project switched from QG2 to QG3. After this change, the number of findings decreases and the clone coverage settles around 6%, which is a success of the quality control. The major increase in the number of findings in 2013 is only due to an automated code refactoring introducing braces that led to threshold violations of few hundred methods. After this increase, the number of findings start decreasing again, showing the manual effort of the developers to remove findings.

For System C (Figure 5), the quality control process shows a significant impact after two years: Since the end of 2012, when the project also switched from QG2 to QG3, both the clone coverage and the overall number of findings decline. In the year before, the project transitioned between development teams and, hence, we only wrote two reports (July 2011 and

System D (Figure 6) almost fulfills QG4 as after 1 year of development, it has only 21 findings in total and a clone coverage of 2.5%. Technically, under QG4, the system should have zero findings. However, in practice, exactly zero findings is not feasible as there are always some findings (e.g., a long method to create UI objects or clones in test code) that are not a major threat to maintainability. Only a human can judge based on manual inspection of the findings whether a system still fulfills QG4, if it does not have exactly zero findings. In the case of System D, we consider 21 findings to be few and minor enough to fulfill QG4.

To summarize, our trends show that our process leads to actual measurable quality improvement. Those trends go beyond anecdotal evidence but are not sufficient to scientifically proof our method. However, Munich RE decided only recently to extend our quality control from the .NET area to all SAP

development. As Munich RE develops mainly in the .NET and SAP area, most application development is now supported by quality control. The decision to extend the scope of quality control confirms that Munich Re is convinced by the benefit of quality control. Since the process has been established, maintainability issues like code cloning are now an integral part of discussions among developers and management.

V. CONCLUSION

Quality analyses must not be solely based on automated measurements, but need to be combined with a significant amount of human evaluation and interaction. Based on our experience, we proposed a new quality control process for which we provided a running case study of 41 industry projects. With a qualitative impact analysis at Munich RE we showed measurable, long-term quality improvements. Our process has led to measurable quality improvement and an increased maintenance awareness up to management level at Munich Re.

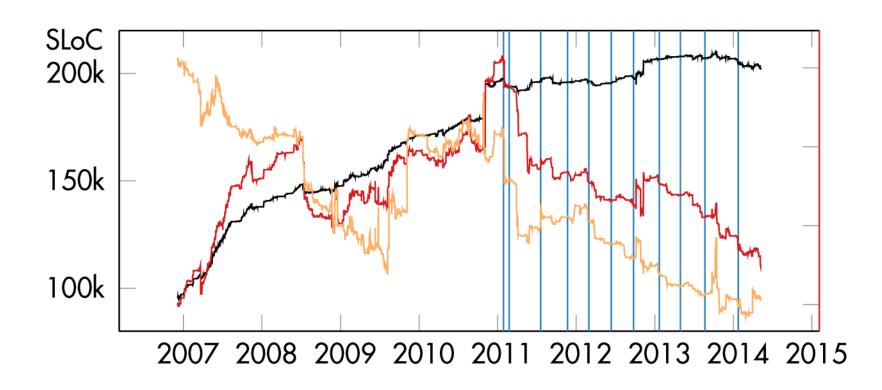
REFERENCES

- [1] F. Deissenbeeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pirka, "Tool support for continuous quality control," in IEEE Software, 2008.
- [2] D. L. Parsas, "Software aging," in ICSE '94.
- [3] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockau, "Does code decay? Assessing the evidence from change management data." IEEE
- [4] P. Johnson, "Requirement and design trade-offs in backystat: An in-process software engineering mea-statement and analysis system," in ESEM 07.
- [5] F. Deissenboeck, M. Pizka, and T. Seifert, "Tool support for continuous quality assessment," in STEP'05.
- [6] L. Heinemann, B. Hummel, and D. Steidl, "Teamscale: Software quality control in real-time," in ICSE 14.
- [7] R. C. Martin, Clean Code: A Handbook of Agile Software Craftonanship.

²e.g., file size, method length, or nesting depth

² http://www.nonarqube.com/

Einsparung durch Clone Detection



Menge an geklontem Code hat sich seit der Einführung von Clone Management halbiert. Ohne Clone Management wäre der Clone Blow-Up daher vorraussichtlich doppelt so groß. Ersparnis Aufwand = 6%

Munich Re spart durch Einsatz von Clone Detection jährlich 6% Aufwand durch vermiedene Redundanz ein.

Findings

✓ All	6793
✓ Architecture	1
Architecture Conformance	1
✓ Code Anomalies	1344
✓ Bad practice	971
✓ Correctness	2
Exception Handling	62
✓ General checks (built-in)	120
✓ Null pointer dereference	13
✔ Performance	36
✓ Unused code	93
Unused variable or parameter	47
Code Duplication	988
✓ Cloning	101
✓ Cloning✓ Redundant Literals	101 887
	101
▼ Redundant Literals	887
Redundant Literals Documentation	887 3378
 ✓ Redundant Literals ✓ Documentation ✓ Comment completeness 	887 3378 3236
 ✓ Redundant Literals ✓ Documentation ✓ Comment completeness ✓ Task tags 	887 3378 3236 142
 ✓ Redundant Literals ✓ Documentation ✓ Comment completeness ✓ Task tags ✓ Formatting 	887 3378 3236 142
 ✓ Redundant Literals ✓ Documentation ✓ Comment completeness ✓ Task tags ✓ Formatting ✓ Code formatting 	887 3378 3236 142 6
 ✓ Redundant Literals ✓ Documentation ✓ Comment completeness ✓ Task tags ✓ Formatting ✓ Code formatting ✓ Naming 	887 3378 3236 142 6 6
 ✓ Redundant Literals ✓ Documentation ✓ Comment completeness ✓ Task tags ✓ Formatting ✓ Code formatting ✓ Naming ✓ Java naming conventions 	887 3378 3236 142 6 110 110
 ✓ Redundant Literals ✓ Documentation ✓ Comment completeness ✓ Task tags ✓ Formatting ✓ Code formatting ✓ Naming ✓ Java naming conventions ✓ Structure 	887 3378 3236 142 6 110 110

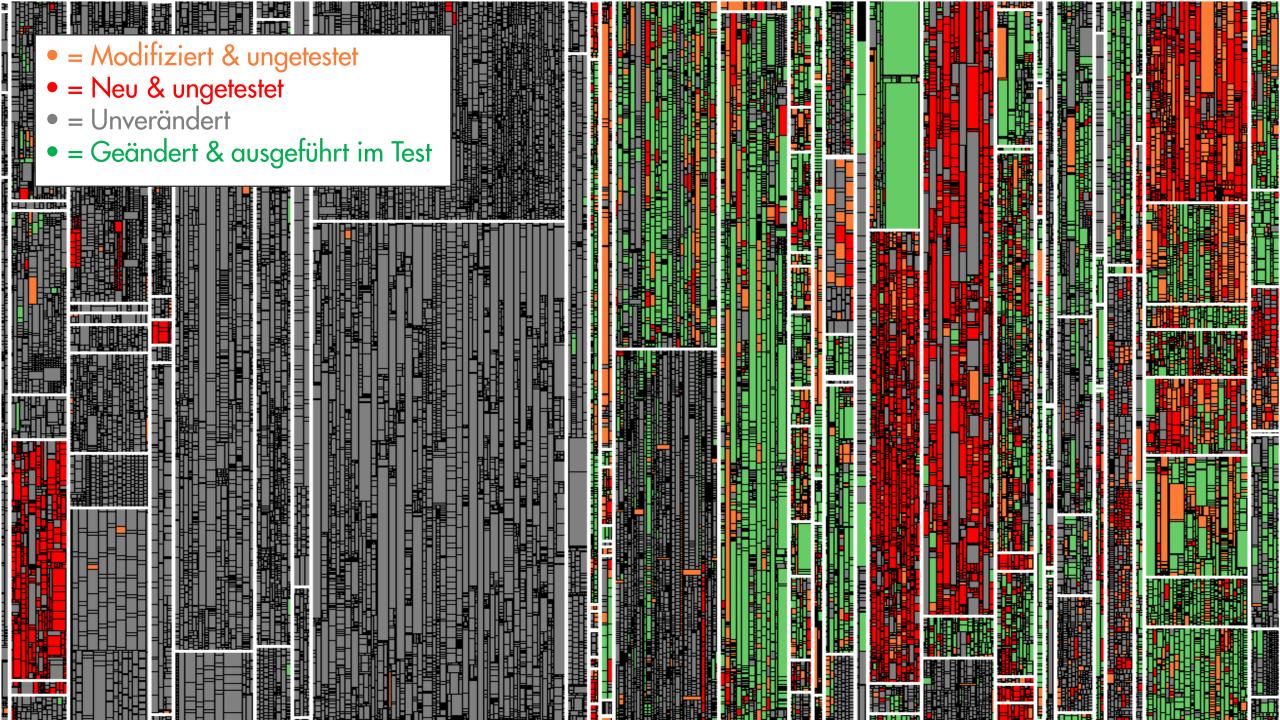
$$500 \frac{PT}{Jahr}$$

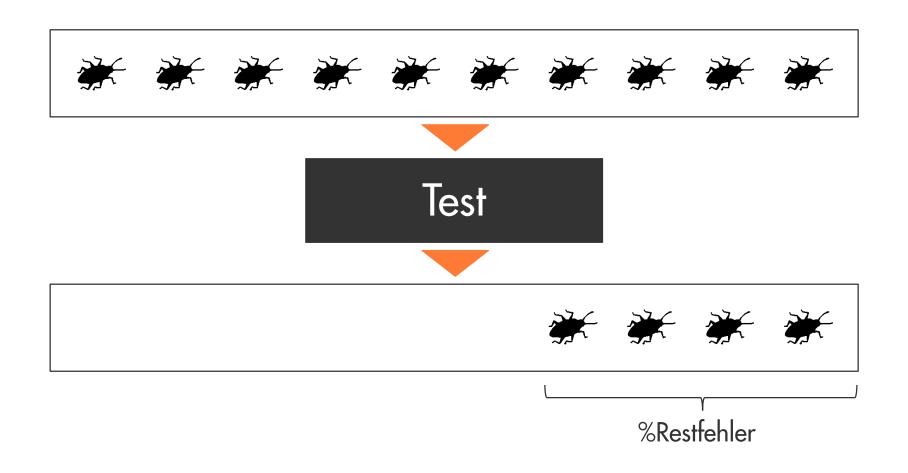
Munich Re spart durch Einsatz von Clone Detection jährlich ca. 500 PT Aufwand für Fehlerbehebung

Ersparnis Aufwand = **6**%

Munich Re spart durch Einsatz von Clone Detection jährlich 6% Aufwand durch vermiedene Redundanz ein.

Kosten-Nutzen von Test-Gap-Analyse

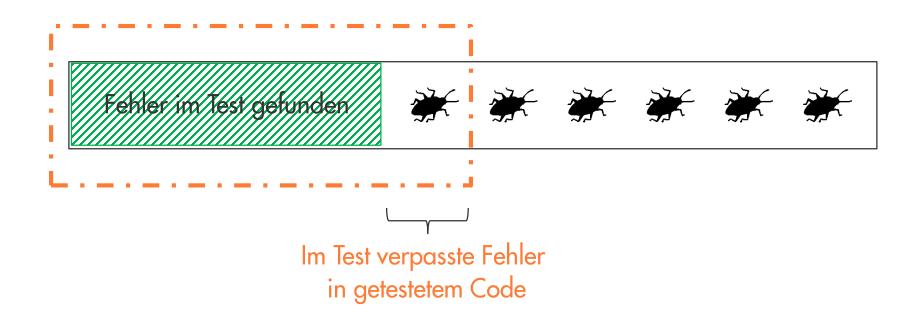




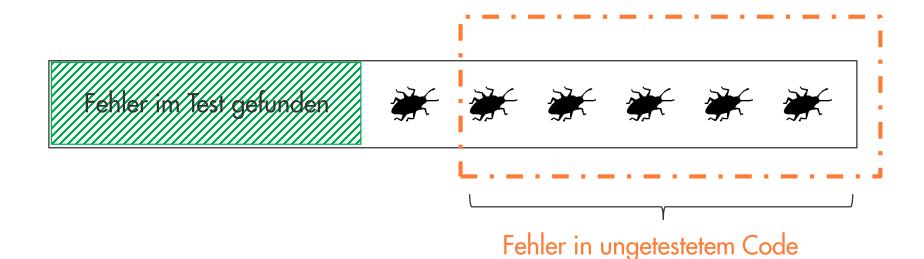
%Restfehler = %Getestet * Testineffektivität + %Testgap



%Restfehler = %Getestet * Testineffektivität + %Testgap



%Restfehler = %Getestet * Testineffektivität + %Testgap



Did We Test Our Changes? Assessing Alignment between Tests and Development in Practice

Sebastian Eder, Benedikt Hauptmann, Maximilian Junker Technische Universität München, Germany Elmar Juergens COSE GmbH Germany

Rudolf Vaas Karl-Heinz Prommer Munich Re Group Germany

zones. Since their distance complicates communication, close ment between development and testing becomes increasingly tens to decrease test effectiveness or increases costs.

In this paper, we propose a conceptually simple approach to assess test alignment by uncovering methods that were changed but never executed during testing. The paper's contribution is a large industrial case study that analyzes development changes, test service activity and field faults of an industrial business information system over 14 months. It demonstrates that the approach is suitable to produce meaningful data and supports

namic analysis, untested code

I INTRODUCTION

A substantial part of the total life cycle costs of longlived software systems is spent on testing. In the domain of business-information systems, it is not uncommon that successful software systems are maintained for two or even three decades. For such systems, a substantial part of their Contribution: This paper presents an industrial case study existing functionality has not been impaired.

crucial. Ideally, each test cycle should validate all implemented ferent organizations in Germany and India. The case study functionality. In practice, however, available resources limit analyzed all development changes, testing activity, and all field each test cycle to a subset of all available test cases. Since selection of test cases for a test cycle determines which bugs are are substantially more likely to occur in methods that were found, this selection process is central for test effectiveness.

A common strategy is to select test cases based on the changes that were made since the last test cycle. The underly ing assumption is that functionality that was added or changed recently is more likely to contain bugs than functionality that support this assumption [1], [2], [3], [4],

testing might focus on code areas that did not change, test suites, but does not give hints to improve them.

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "EvoCon, 01IS12034A". The responsibility for this article lies with the authors.

Abstract—Testing and development are increasingly performed or—more critically—substantial code changes might remain by different organizations, often in different countries and time untested. Test alignment depends on communication between testing and development. However, they are often performed by different teams, often located in different countries and time-zones. This distance complicates communication and thus challenges test alignment. But how can we assess test alignment and expose areas where it needs to be improved?

Problem: We lack approaches to determine alignment be tween development and testing in practice.

Proposed Solution: In this paper, we propose to assess test alignment by measuring the amount of code that was st alignment in practice.

Index Terms—Software testing, software maintenance, dy
changed but not tested. We propose to use method-level

and the propose to use method-level

changed but not tested by propose to use method-level

and the propose to use method-level

and the propose to use method level

a change coverage information to support testers in assessing test alignment and improving test case selection.

Our intuition is that changed, but untested methods are more likely to contain bugs than either unchanged methods or tested ones. However, our intuition might be dead wrong; method level churn could be a bad indicator for bugs, since methods can contain bugs although they have not changed in ages.

total lifecycle costs is spent on testing to make sure that new that explores the meaningfulness and helpfulness of methodfunctionality works as specified, and—equally important—that level change coverage information. The case study was performed on a business information system owned by Munich During maintenance of these systems, test case selection is Re. System development and testing were performed by difchanged but not tested.

II. RELATED WORK

The proposed approach is related to the fields of defect prediction, selective regression testing, test case prioritization has passed several test cycles unchanged. Empirical studies and test coverage metrics. The most important difference to the named topics is the simplicity of the proposed approach and If development and testing efforts are not aligned well, the fact that change coverage assesses the executed subsets of

> Defect prediction is related to our approach, because we identify code regions that were changed, but remained untested, with the expectation that there are more field bugs

therefore useful for maintainers and testers to identify relevant gaps in their test coverage

We perform the study on a business information system at Munich Re. The analyzed system was written in C# and its size are 340 kLOC. In total, we analyzed the system for 14 months. The system has been successfully in use for nine years. and is still actively used and maintained. Therefore, there is a well implemented bug tracking and testing strategy. This allows us to gain precise data about which parts of the system were changed and why they were changed.

We analyzed two consecutive releases of the system. Re- and a query interface that allows retrieving coverage, change. lease 1 was developed in five iterations in two months, and and change coverage information. The same tool support was release 2 was developed in ten iterations in four months. used in earlier studies [17], [19]. Both releases were deployed to the productive environment Validity Procedures: We focus on validity procedures and not due to hot fixes five times and were in productive use for on threats to validity due to space limitation six months. Note that one deployment may concern several bugs and changes in the system. The system contained 22123 (release 1) respectively 22712 (release 2) methods.

For both releases, test suites containing 65 system test cases covering the main functionality were executed three times.

C. Study Design and Execution

the categories shown in Figure 2: Tested or untested, changed or unchanged, and whether methods contain field bugs



then we answer RQ 1 and RQ 2 based on the collected data. RQ 2: We found 23 fixes in release 1 and 10 fixes in identify changes during the development phase and relate and coverage categories of methods is shown in Table I usage data to these genealogies. With this information, we The biggest part of bugs occurred in methods categorized a dentify method genealogies that are changed-untested.

defects for every category of methods by detecting changes less bugs in unchanged regions than in changed regions in the productive phase of the system in retrospective. This s valid for the analyzed system, since only severe bugs are in release 1 and 0.21% in release 2, the probability of bugs fixed directly in the productive environment, which is defined is higher in the group of methods that were changed-untested

in the productive phase, which means they were related to a bug. We then categorize methods by change and coverage E. Discussion during the development phase. Based on this, we calculate the RO 1: With 15% of all methods being changed and 34% of bug probability in the different groups of methods.

Study Execution: We used tool support, which consists of plays a considerable role in the analyzed system. The high three parts: An ephemeral [18] profiler that records which amount of changed methods results from newly developed methods were called within a certain time interval, a database features, which means that many methods were added during that stores information about the system under consideration, the development phase of both releases.

but untested code as validation of the approach.

Fig. 3. Probability of fixes in both release

information, [7], [8], [9],

We conducted manual inspections to ensure that every but that is identified by our tool support is indeed a bug.

based on locality and signatures, we conducted manual inspec tions of randomly chosen method genealogies. We found no false genealogies and have therefore a high confidence in the correctness of our technique. We also used the algorithm in For all research questions, we classify methods according to our former work [17], which provided suitable results as well.

RO 1: Untested methods account for 34% in both releases we analyzed, 15% of all methods were changed during the development phase of the system, also in both releases. The equality of the numbers for both releases is a coincidence

8% respectively 9% of all methods were changed-untested Considering only changed methods, only 44% were tested in 2. These numbers constitute that there are gaps in the test Study Design: First, we collect coverage and program data, coverage of changed code in the analyzed system.

For answering RQ 1, we build method genealogies and release 2. The distribution of the bugs over the different change changed-untested with 43% of all bugs in release 1 and 40% For answering RQ 2, we calculate the probability of field of all bugs in release 2. In both releases, there are considerably

The probabilities of bugs are shown in Figure 3. With 0.53% This confirms that tested code or code that was not changed in We gain our results by identifying methods that are changed the development phase is less likely to contain field defects.

all methods being not tested, untested code and changed code

There are several models for defect prediction [5]. In contrast to these models, we measure only changes in the system and the coverage by tests and do not predict bugs, but assess test suites and use the probability of bugs in changed,

The proposed approach is related to [6] which uses series of changes "change bursts" to predict bugs. The good results that were achieved by using change data for defect prediction encourage us to combine similar data with testing efforts. Selective regression testing techniques target the selection of test cases from changes in source code and coverage

In contrast to these approaches, the paper at hand focuses on the assessment of already executed test suites, because or new features are developed. Development usually occurs often experts decide which tests to execute to cover most of in iterations which are followed by test runs which are the the changes made to a software system [10]. However, their execution of a selection of tests aiming to test regressi estimations contain uncertainties and therefore possibly miss as well as the changed or added code. A development phase some changes. Our approach aims at identifying the resulting is completed by a release which transfers the system into uncovered code regions. Therefore, our approach can only be used if testing activities were already performed.

Compared to [11], we are validating our approach by measuring field defects, and do not take defects into account that were found during development.

Test coverage metrics give an overview of what is covered by tests. Much research has been performed in these topics [12] and there is a plethora of tools [13] and a number of metrics available, such as statement, branch, or path coverage [14]. In contrast to these metrics, we focus on the more coarse grained method coverage. Furthermore, we do not only consider static properties of the system under test, but changes.

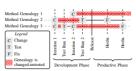
Empirical studies on related topics focus to the best of our knowledge mainly on the effectiveness of test case selection and prioritization techniques [9], [15]. In our study, we assess test suites by their ability to cover changes of a software system, but do not consider sub sets of test suites.

III. CONTEXT AND TERMS

In this work, we focus on system testing according to the definition of IEEE Std 610.12-1990 [16] to denote "testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements". System ests are often used to detect bugs in existing functionalit after the system has been changed. In our context, many tests are executed manually and denoted in natural language.

Our study uses methods as they are known from programning languages such as Java or C#. Methods form the entities of our study and can be regarded as units of functionality of a software system. They are defined by a signature and a body. To compare different releases of a software system over time, we create method genealogies which represent the evolution of changed, but untested code, to justify the problem statement of a single method over time. A genealogy connects all releases this work. Therefore, we quantify changed and untested code. of a method in chronological order [17].

In the context of our work, the life cycle of a software field bugs than unchanged or tested methods? The goal of ystem consists of two alternating phases (see Figure 1). In this research question is to decide whether change coverage the development phase, existing functionality is maintained can be used as a predictor for bugs in large code regions and is



the productive phase. In the productive phase, functionality is usually neither added nor changed. If critical malfunctions are detected, hot fixes are deployed in the productive phase.

We consider a method as tested if it has been executed during a test run. If a method has been changed or added and been tested afterwards before the system is released we consider it as changed-tested. If a method change or addition has not been tested before the system is transferred in the productive phase, we consider the method as changed-untested (see genealogy 1 and 3 in Figure 1).

To quantify the amount of changes covered by tests, we introduce the metric change coverage (CC). It is computed by the following formula and ranges between [0,1].

A change coverage of 1/(CC - 1) means that all methods which have been changed since the last test run have been tested after their last change. On the contrary, a coverage of 0 (CC = 0) indicates that none of the changed methods have been covered by a test.

A. Goal and Research Questions

The goal of the study is to show whether change coverage is a useful metric for assessing the alignment between tests and development. We formulate the following research questions RQ 1: How much code is changed, but untested? The goal of this research question is to investigate the existence of RQ 2: Are changed-untested methods more likely to contain

	Rele	ise 1	Release 2			
Category	Absolute	Relative	Absolute	Relative		
changed-tested	5	22%	3	30%		
changed-untested	10	43%	4	40%		
unchanged-tested	0	0%	0	0%		
unchanged-untested	8	35%	3	30%		

tested in the analyzed system. These high numbers also result consideration of code regions that are modified, but not very from features that are newly developed during the development well tested is important. This motivates future work on the phase. For these new features, there was only a very limited topic and the inference of improvement goals. number of test cases.

few bugs at the current stage of development and bugs are recovery to bridge the gap to test cases. brought into the system by changes.

Furthermore, the probability of bugs in untested code is, in both releases, less than half of the probability in changeduntested code. Hence, we conclude that only considering test coverage is not as efficient as considering change coverage.

The probability of bugs in changed code regions is also considerably higher than in untested regions. But the combination of both metrics, test coverage and changed methods points to code regions that are more likely to contain bugs than others. Is Change Coverage Helpful in Practice? We employed the proposed approach also in the context of Munich Re in currently running development phases. We showed the results to developers and testers by presenting code units, like types or assemblies ordered by change coverage. During the discussion of the results, we conducted open interviews with developers to gain knowledge about how helpful information about change

coverage is during maintenance and testing. Developers identified meaningful methods in changed but untested regions by using the static call graph to find methods [10] M. Harrold and A. Orso, "Retesting software during development and they know. With these methods, the developers were able to identify features that remained untested. For example the processing of excel sheets in a particular calculation was [15]. A Evizant-was 17-1 hagurian, "Effectively prioritizing tests in development environment," in ISSIA, 2002.

[18] A Evizant-was 18, 11, 2014. B, 20 changed, but remained untested afterwards. In this case, among some others, the (re-)execution of particular test cases and some others, the (re-)execution of particular test cases and
the creation of new test cases were issued. This increased
[14] Y. Malaiya, M. Li, J. Bieman, and R. Karcich, "Software reliability the change coverage considerably for the code regions where the features are located. This shows that change coverage is helpful for practitioners.

VI. CONCLUSION AND FUTURE WORK

We presented an automated approach to assess the alignment of test suites and changes in a simple and understandable way. Instead of using rather complex mechanisms to derive code units that may be subject to changes, we are focusing on changed but untested methods and calculate an expressive metric from these methods. The results show that the use of

change coverage is suitable for the assessment of the alignment of testing and development activities.

We also showed that change coverage is suitable for guiding testers during the testing process. With information about change coverage, testing efforts can be assessed and redirected if necessary, because the probability of bugs is increased in changed-untested methods. Furthermore, we presented our tool support that allows us to utilize our technique in practice

However, the number of bugs we found is too small to derive generalizable results. Therefore, we plan to extend our studies to other systems to increase external validity. But the 43% respectively 40% of the changed methods were not first results that we presented in this work point out that the

One challenge is the identification of suitable test cases from RO 2: With a probability of bugs in untested-changed methods code regions to give hints to testers and developers which test of 0.53% respectively 0.21%, this group of methods contains case to execute to cover more changed, but untested methods. most of the bugs. This means that the system itself contains Therefore, we plan to evaluate techniques related to trace link

[1] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in ICSE, 2005. V. Nagappan, B. Murphy, and V. Basili, "The influence of organizational

structure on software quality," in ICSE, 2008.

[3] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," IEEE Trans. Softw. Eng., vol. 26, no. 7.

[4] T. J. Ostrand, E. J. Wevuker, and R. M. Bell, "Where the bugs are," in

[5] T Hall S Reecham D Rowes D Gray and S Counsell "A systematic

T. Hall, S. Beccham, D. Bowes, D. Gray, and S. Counsed, "A systematic literature review on fair prediction performance in software engineering." IEEE Trans. Softw. Eng., vol. 38, no. 6, 2012.
 [6] E. Berna, Softw. Eng., vol. 38, no. 6, 2012.
 [6] Y. Chandachawa, V. K. Shambhay, A. Panigrahi, R. Sisodi, and R. Murphy, "Change burnes, "Scie unbest-ergosion test selection for managed S. Labelmann," "Scie unbest-ergosion test selection for managed S. Labelmann, "Scie unbest-ergosion test selection for managed Section of the Computer of

and adequacy," ACM Comput. Surv., vol. 29, no. 4, 1997.
[13] Q. Yang, J. J. Li, and D. Weiss, "A survey of coverage based testing

Malanya, M. Li, J. Bieman, and K. Karcken, "Software Felancing growth with set coverage," *IEEE Trans. Rel.*, vol. 51, no. 4, 2002.
 G. Rothermel, R. Unich, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *IEEE Trans. Softs. Eng.*, vol. 27, no. 10, 2001.
 EEE, "IEEE Standard Glossary of Software Engineering Terminology,"

New York, USA, 1990.

[17] S. Eder, M. Junker, E. Jurgens, B. Hauptmann, R. Vaas, and K. Promme

S. Eder, M. Junker, E. Jurgens, B. Hauptmann, K. Vaas, and R. Prommer.
 "Hos much does unused code matter for maintenance?" in CSE, 2012.
 Grand G. G. Garden, "Language of the Company of the Com

Wieviele Änderungen sind ungetestet?

Studie: C# System @ Munich Re

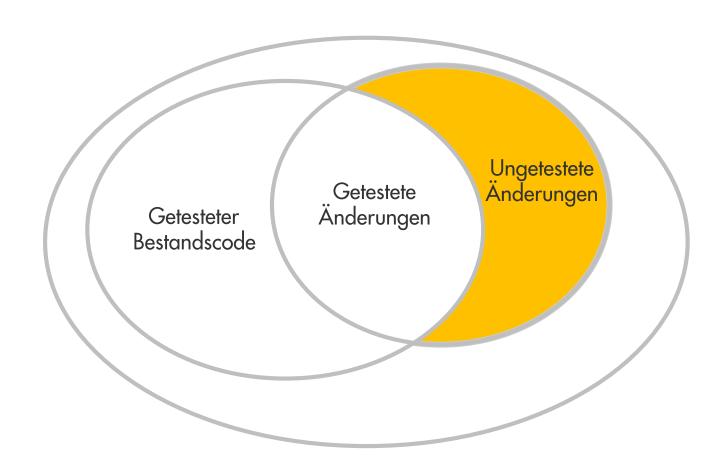
Release A:

15% Code neu/geändert, >50% ungetestet

Release B:

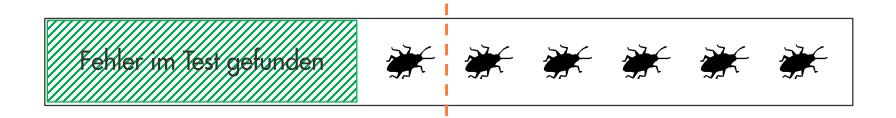
15% Code neu/geändert,

>60% ungetestet



Feldfehlerwahrscheinlichkeit 5x höher für ungetestete Änderungen!

%Restfehler = 60%



$$%$$
Restfehler = 28%



Reduzierte Feldfehler = 50%

Reduzierte Feldfehler = 50%

Test-Gap-Analyse reduziert Feldfehler in den Applikationen der Munich Re um 1/3

Fazit

- Conformance Costs << Costs of Non-Conformance
- Mit der Nutzenargumentation im Rücken konzentrieren uns auf umfassende Nutzung der Tools und Prozesse.
- Tools und Prozesse wichtig, etabliert und fest verankert.
- Internes Change Management ("1/3") notwendig.
- Sichtbarmachen von Qualität ist essentiell.

Kontakt - Wir freuen uns auf Diskussionen ©

Uwe Proft Dr. Elmar Jürgens uproft@munichre.com juergens@cqse.eu