

An Empirical Study of Software Reuse vs. Defect-Density and Stability

Parastoo Mohagheghi^{1,2,3}, Reidar Conradi^{2,3}, Ole M. Killi², Henrik Schwarz²

¹Ericsson Norway-Grimstad, Postuttak, NO-4898 Grimstad, Norway

²Department of Computer and Information Science, NTNU, NO-7491 Trondheim, Norway

³Simula Research Laboratory, P.O.Box 134, NO-1325 Lysaker, Norway

parastoo.mohagheghi@ericsson.com, conradi@idi.ntnu.no, henrik@schwarz.no

Abstract

The paper describes results of an empirical study, where some hypotheses about the impact of reuse on defect-density and stability, and about the impact of component size on defects and defect-density in the context of reuse are assessed, using historical data (“data mining”) on defects, modification rate, and software size of a large-scale telecom system developed by Ericsson. The analysis showed that reused components have lower defect-density than non-reused ones. Reused components have more defects with highest severity than the total distribution, but less defects after delivery, which shows that these are given higher priority to fix. There are an increasing number of defects with component size for non-reused components, but not for reused components. Reused components were less modified (more stable) than non-reused ones between successive releases, even if reused components must incorporate evolving requirements from several application products. The study furthermore revealed inconsistencies and weaknesses in the existing defect reporting system, by analyzing data that was hardly treated systematically before.

1. Introduction

There is a lack of published, empirical studies on large industrial systems. Many organizations gather a lot of data on their software processes and products, but either the data are not analyzed properly, or the results are kept inside the organization. This paper presents results of an empirical study in a large-scale telecom system, where particularly defect-density, and stability are investigated in a reuse context. Software reuse has been proposed e.g. to reduce time-to-market, and to achieve better software quality. However, we need empirical evidence in terms of e.g. increased productivity, higher reliability, or lower modification rate to accept the benefits of reuse.

Ericsson has developed two telecom systems that share software architecture, components in reusable layers, and

many other core assets. Characteristics of these systems are high availability, reliability, and scalability. During the lifetime of the projects, lots of data are gathered on defects, changes, duration time, effort, etc. Some of these data are analyzed, and results are used in the improvement activities, while some others remain unused. Either there is no time to spend on data analysis, or the results are not considered important or linked to any specific improvement goals. We analyzed the contents of the defect reporting system (containing all reported defects for 12 product releases), and the contents of the change management system. For three of these releases, we obtained detailed data on the size of components, and the size of modified code. We have assessed four hypotheses on reuse, and reused components using data from these releases. We present detailed results from one of these releases here. The quality focus is defect-density (as the number of defects divided by lines of code), and stability (as the degree of modification). The goal has been to evaluate parameters that are earlier studied in traditional reliability models (such as module size and size of modified code) in the context of reuse, and to assess the impact of reuse on software quality attributes.

Results of the analysis show that reused components have lower defect-density than non-reused ones, and these defects are given higher priority to solve. Thus reuse may be considered as a factor that improves software quality. We did not observe any relation between defect-density or the number of defects as dependent variables, and component size as the independent variable for all components. However, we observed that non-reused components are more defect-prone, and there is a significant correlation between the size of non-reused components, and their number of defects. This must be further investigated. The study also showed that reused components are less modified (more stable) than non-reused ones, although they should meet evolving requirements from several products.

Empirical evidence for the benefits of reuse in terms of lower defect-density, and higher stability is interesting for both the organization, and the research community. As the data was not collected for assessing concrete

hypotheses, the study revealed weaknesses in the defect reporting system, and identified improvements areas.

This paper is organized as follows. Section 2 presents some general concepts, and related work. Section 3 gives an overview of the studied product, and the defect reporting system. Section 4 the research method, and hypotheses. Hypotheses are assessed in Section 5. Section 6 contains a discussion, and summary of the results. The paper is concluded in Section 7.

2. Related work

Component-Based Software Engineering (CBSE) involves designing and implementing software components, assembling systems from pre-built components, and deploying systems into their target environment. The reusable components or assets can take several forms: subroutines in library, free-standing COTS (Commercial-Off-The-Shelf) or OSS (Open Source Software) components, modules in a domain-specific framework (e.g. Smalltalk MVC classes), or entire software architectures, and their components forming a product line or system family (the case here). CBSE, and reuse promise many advantages to system developers and users such as:

- Shortened development time, and reduced total cost, since systems are not developed from scratch.
- Facilitation of more standard, and reusable architectures, with a potential for learning.
- Separation of skills, since much complexity is packaged into specific frameworks.
- Fast access to new technology, since we can acquire components instead of developing them in-house.
- Improved reliability by shared components – etc.

These advantages are achieved in exchange for dependence on component providers, vague trust to new technology, and trade-offs for both functional requirements, and quality attributes.

Testing is the key method for dynamic verification (and validation) of a system. A system undergoes testing in different stages (unit testing, integration testing, system testing etc), and of different kinds (reliability testing, efficiency testing etc). Any deviation from the system's expected function is usually called for a *failure*. Failures observed by test groups or users are communicated to the developers by means of *failure reports*. A *fault* is a potential "flaw" in a hardware/software system that causes a failure. The term *error* is used both for execution of a "passive" fault leading to erroneous (vs. requirements) behavior or system state [6], or for any fault or failure that is a consequence of human activity [2]. Sometimes, the term *defect* is used instead of faults,

errors or failures, not distinguishing between active or passive faults or human/machine origin of these. *Defect-density* or *fault-density* is then defined as the number of defects or faults divided by the size of a software module.

There are studies on the relation between fault-density and parameters such as software size, complexity, requirement volatility, software change history, or software development practices – see e.g. [1, 3, 5, 9, 13, 14]. Some studies report a relation between fault-density and component size, while others not. The possible relation can also be decreasing or increasing fault-density with growing size. Fenton et al. [3] have studied a large Ericsson telecom system, and did not observe any relation between fault-density and module size. When it comes to relation between the number of faults and module size, they report that size weakly correlates with the number of pre-release faults, but do not correlate with post-release faults. Ostrand et al. [14] have studied faults of 13 releases of an inventory tracking system at AT&T. In their study, fault-density slowly decreases with size, and files including high number of faults in one release, remain high-fault in later releases. They also observed higher fault-density for new files than for older files.

Malaiya and Denton [9] have analyzed several studies, and present interesting results. They assume that there are two mechanisms that give rise to faults. The first is how the project is partitioned into modules, and these faults decline as module size grows (because communication overhead, and interface faults are reduced). The other mechanism is related to how the modules are implemented, and here the number of faults increases with the module size. They combine these two models, and conclude that there is an "optimal" module size. For larger modules than the optimal size, fault-density increases with module size, while for smaller modules, fault-density decreases with module size (the economy of scale).

Graves et al. [5] have studied the history of change of 80 modules of a legacy system developed in C, and some application-specific languages to build a prediction model for future faults. The model that best fitted to their observations included the change history of modules (number of changes, length of changes, time elapsed since changes), while size and complexity metrics were not useful in such prediction. They also conclude that recent changes contributed the most to the fault potential.

There are few empirical studies on fault-density in the context of reuse. Melo et al. [10] describe a student experiment to assess the impact of reuse on software quality (and productivity) using eight medium-sized projects, and concluded that fault-density is reduced with reuse. In this experiment, reused artifacts are libraries such as C++ and GNU libraries; i.e. COTS and OSS artifacts. Another experiment that shows improvement in

reliability with reuse of a domain-specific library is presented in [15].

High fault-density before delivery may be a good indicator of extensive testing rather than poor quality [3]. Therefore, fault-density cannot be used as a de-facto measure of quality, but remaining faults after testing will impact reliability. Thus it is equally important to assess the effectiveness of the testing phases, and build prediction models. Probably such a model includes different variables for different types of systems. Case studies are useful to identify the variables for such models, and to some extent to generalize the results.

3. The Ericsson context

3.1. System Description

Our study covers components of a large-scale, distributed telecom system developed by Ericsson. We have assessed several hypotheses using historical data on defects, and changes of these systems that are either published by us, or will be published. This paper presents some of the results that are especially concerned with software reuse.

Figure 1 shows the high-level software architecture of the systems. This architecture is gradually developed to allow building systems in the same system family. This was a joint development effort across teams and organisations in Norway and Sweden for over a year, with much discussion and negotiation [12]. The systems are developed incrementally, and new features are added to each release of them. The two systems A and B in Figure 1 share the system platform, which is considered as a COTS component developed by another Ericsson organization. Components in the middleware, and business specific layers are shared between the systems, and are hereby called for *reused components* (reused in two distinct products and organizations, and not only across releases). Components in the application-specific layer are specific to applications, and are called for *non-reused components*.

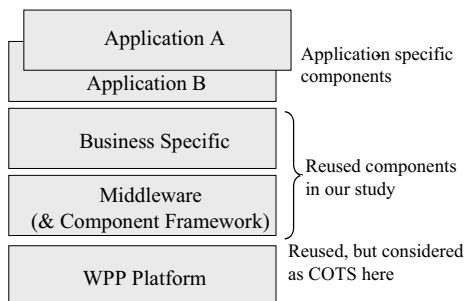


Figure 1. High-level architecture of systems

The architecture is component-based, and all components in our study are built in-house. Several Ericsson organizations in different countries have been involved in development, integration, and testing of the systems. But what a component is in this discussion?

Each system is decomposed hierarchically into subsystems, blocks, units, and modules (source files). A *subsystem* presents the highest level of encapsulation used, and has formally defined (provided) interfaces in IDL (Interface Definition Language). It is a collection of blocks. A *block* has also formally defined (provided) interfaces in IDL, and is a collection of lower level (software) units. Subsystems, and blocks are considered as components in this study; i.e. high-level (subsystems) and lower-level (blocks) components. Since communication inside blocks are more informal, and may happen without going through an external interface, blocks are considered as the lowest-level components.

The systems' GUIs are programmed in Java, while business functionality is programmed in Erlang and C. Erlang is a functional language for programming concurrent, real-time, distributed fault-tolerant systems. We have data on defects and component size of 6 releases of one system (and several releases of other systems in the same system family releases). We present a detailed study of one of these releases in this paper. We obtained the same results with data from 2 other releases as well, but the data for this special release is more complete, and this release is the latest version of the system on the time of the study. The release in our study consisted of 470 KLOC (Kilo Lines of non-commented Code), where 64% is in Erlang, 26% in C, and the rest in other programming languages (Java, Perl, etc). Sometimes the term *equivalent code* is used for the size of systems developed in multiple programming languages. To calculate the "equivalent" size in C, we multiplied the software size in Erlang with 3.2, Java with 2.4, and IDL with 2.35, as the practice is in the organization. However, we found that other studies use other numbers. For example, Doug implemented 21 identical programs in C and Erlang, and reported an equivalent factor of 1.46 [16]. Based on the results of this study, we came to another factor (2.3) that must be further assessed. However, the results did not show any significant difference using pure LOC or equivalent ones.

All source code (including IDL files) is stored in a configuration management system (ClearCase). A product release contains a set of files with a specific label for the release in this system.

3.2. Trouble reports

When a defect is detected during integration testing, system testing or later in maintenance, a Trouble Report (TR) is written, and stored in a TR database using a web

interface. Besides, if requirement engineering, or analysis and design of iteration n find defects in software delivered in iteration $n-1$, a TR will also be written. If a defect is reported multiple times, it reports problems observed due to the same fault, and is considered as a *duplicate*.

A TR contains the following fields: header with a number as identifier, date, product (system name), release, when the defect is detected (analysis and design, system test etc), severity, a defect code (coding, documentation, wrong design rule applied etc), assumed origin of the defect, estimated number of person-hours needed to correct the defect, identifier of another TR that this one is a duplicate of (if known), and a description. Three different severities are defined: A (most serious defects with highest priority that brings the system down or affects many users), B (affects a group of users or restarts some processes), and C (all other defects that do not cause any system outage). TRs are written for all types of defects (software, hardware, toolbox, and documentation), and there should be only one problem per TR.

All registered TRs are available as plain text files. We created a tool in C# that traversed all the text files, extracted all the existing fields, and created a summary text file. The summary was used to get an overview of the raw data set, and to decide which fields are relevant for the study. The exploration revealed a lot of inconsistencies in the TR database, e.g. fields are renamed several times, apparently from one release to the other. For example, a subsystem is stored as 'ABC' or 'abc' or 'ABC_101-27'. Another major weakness of the current defect reporting system is the difficulty to track defects to software modules without reading all the attached files (failure reports, notes from the testers, etc) or parsing the source code. Each TR has a field for software module, but this is only filled if the faulty module is known when the TR is initiated, and is not updated later. These inconsistencies show that data had hardly been systematically analyzed or used to a large extent before.

After selecting the fields of interest, another tool in C# read each TR text file, looked for the specified fields, and created a SQL insert statement. We verified the process by randomly selecting data entries, and cross checking them with the source data.

We inserted data from 13,000 TRs in a SQL database for 12 releases of systems. Around 3,000 TRs were either duplicated or deleted. The release of system A in this study had 1953 TRs in the database, which are used for assessment of hypotheses in this paper. This release was in the maintenance phase on the date of this study (almost 8 months after delivery). TRs report both pre-delivery and post-delivery defects (from maintenance). 1539 TRs in

our study were initiated pre-delivery (79%), while 414 TRs (21%) were post-delivery defects.

4. Research method and hypotheses

The overall research question in our study is the impact of reuse on software quality. To address this research question, we have to choose some attributes of software quality. Based on the literature search, and a pre-study of the available data, we chose to focus on defect-density, and stability of software components in the case study. There are inherently two limitations in this design:

1. Are defect-density and stability good indicators of software quality?
2. Can we generalize the results?

To answer the first question, we must assess whether defect-prone components stay defect-prone after release, and in several releases, and build a prediction model. This is not yet done.

The second limitation has two aspects: definition of the population, and limitations of case study research. Our data consists of non-random samples of components, and defect reports of a single product. Formal generalization is impossible without random sampling of a well-defined population. However, there are arguments for generalization on the background of cases [4]. The results may at least be generalized to other releases of the product under study, and products developed by the same company when the case is a probable one. On the other hand, if we find evidence that there is no co-variation between reuse, and quality attributes, the results could be a good example of a falsification case, which could be of interest when considering reuse in similar cases.

We chose to refine the research question in a number of hypotheses. A hypothesis is a statement believed to be true about the relation between one or more attributes of the *object of study*, and the *quality focus*. Choosing hypotheses has been both a top-down, and a bottom-up process. Some goal-oriented hypotheses and related metrics were chosen from the literature (top-down), to the extent that we had relevant data. In other cases, we pre-analyzed the available data to find tentative relations between data and possible research questions (bottom-up).

Table 1 presents 4 groups of hypotheses regarding reuse vs. defect-density and modification rate, and the alternative hypotheses for two of them; i.e. **H1**, and **H4**. For the other two groups of hypotheses, the null hypotheses state that there is no relation between the number of defects or defect-density, and component size. The alternative hypotheses are that there is a relation between the number of defects or defect-density with component size. Table 1 also shows an overview of the

results. Section 5 presents the details of data analysis, and other observations.

Table 1. Research Hypotheses and results

HypId	Hypothesis Text	Result
H1	H01: Reused components have the same defect-density as non-reused ones.	Rejected
	HA1: Reused components have lower defect-density than non-reused ones.	Accepted
H2	H02-1: There is no relation between number of defects and component size for all components.	Not rejected
	H02-2: There is no relation between number of defects and component size for reused components.	Not rejected
	H02-3: There is no relation between number of defects and component size for non-reused components.	Rejected
H3	H03-1: There is no relation between defect-density and component size for all components.	Not rejected
	H03-2: There is no relation between defect-density and component size for reused components.	Not rejected
	H03-3: There is no relation between defect-density and component size for non-reused components.	Not rejected
H4	H04: Reused and non-reused components are equally modified.	Rejected
	HA4: Reused components are modified more than non-reused ones.	Rejected

5. Data analysis

We used Microsoft Excel and Minitab for data visualization, and statistical analysis. Statistical tests were selected based on the type of data (mostly on ratio scale). For more description of tests, see [11] and [17].

Most statistical tests return a *P-value* (the observed significance level), which gives the probability that the sample value is as large as the actually observed value if the null hypothesis (H_0) is true. Usually, H_0 is rejected if the *P-value* is less than a significance level (α) chosen by

the observer. Historically, significance levels of 0.01, 0.05 and 0.1 are used because the statistical values related to them are found in tables. We present the *P-values* of the tests to let the reader decide whether to reject the null hypotheses, and give our conclusions as well.

The *t-test* is used to test the difference between two population means with small samples (typically less than 30). It assumes normal frequency distributions, but is resistant to deviations from normality, especially if the samples are of equal size. Variances can be equal or not. If the data departs greatly from normality, non-parametric tests such as *Wilcoxon test*, and *Mann-Whitney test* should be applied. Mann-Whitney test is the non-parametric alternative to the two-sample *t-test*, and tests the equality of two populations' medians (assumes independent samples, and almost equal variances).

Regression analysis helps to determine the extent to which the dependent variable varies as a function of one or more independent variables. The regression tool in Excel offers many options such as residual plots, results of an ANOVA test (Analysis of Variance), R^2 , the adjusted R^2 (adjusted for the number of parameters in the model), and the significance of the observed regression line (*P-value*). R^2 and the adjusted R^2 show how much of the variation of the independent variable is explained with the variation of the dependent variable. Again it is up to the observer to interpret the results. We consider the correlation as low if the adjusted R^2 is less than 0.7.

Chi-square test is used to test whether the sample outcomes results from a given probability model. The inputs are the actual distribution of samples, and the expected distribution. Using Excel, the test returns a *P-value* that indicates the significance level of the difference between the actual, and expected distributions. The test is quite robust if the number of observations in each group is over 5.

5.1. H1: Reuse and defect-density

The quality focus is defect-density. We study the relation between component type (reused vs. non-reused), and defect-density.

H01: Reused components have the same defect-density as non-reused ones.

HA1: Reused components have lower defect-density than non-reused ones.

Results: Size of the release is almost 470 KLOC, where 240 KLOC is modified or new code (MKLOC= Modified KLOC). 61% of the code is from the reused components. Only 1519 TRs (from 1953 TRs) have registered a valid subsystem name, and 1063 TRs have registered a valid block name. We calculated defect-density using KLOC and MKLOC, and also using equivalent C-code. We do not present the results for equivalent C-code, but the conclusions were the same.

To compare the mean values of the two samples (reused, and non-reused components), we performed one-tail t-tests assuming zero difference in the means. However, the number of subsystems is low, which gave too few data points, and relatively high P-values. For example $P(T < t)$ one-tail=0.36 for #TRs/KLOC, which means that there is a probability of 36% that the observed difference is just coincidental. The same analysis on the block level gives results of statistical significance.

Table 2. No. of TRs for subsystems and blocks

Component	#TRs all	No. of Reused Comp.	%TRs Reused	No. of Non-reused Comp.	%TRs Non-reused
Subsystems	1519	9	44%	3	56%
Blocks	1063	29	41%	20	59%

Table 3 shows means, medians, and variances of defect-density. We tested the samples for normality, and the assumption of equal variances. The assumption of normality is violated for reused components and #TRs/KLOC, and the variances are unequal. For defect-density of modified code, distributions are not normal, but have almost equal variances. Table 4 shows results of the statistical tests. The t-test is applied since it is robust to the violation of normality, but a non-parametric test is also applied which has no assumption on distribution.

Table 3. Descriptive statistics for defect-density of blocks

Defect-density	Mean	Median	Variance
#TRs/KLOC, Reused	1.32	0.76	1.70
#TRs/KLOC, Non-Reused	3.01	2.44	4.39
#TRs/MKLOC, Reused	3.50	1.78	21.26
#TRs/MKLOC, Non-Reused	5.69	3.73	21.76

Table 4. Summary of the results of t-tests

P-values	t-test	Mann-Whitney
$P(T \leq t)$ one-tail [#TR/KLOC]	0.002	0.000
$P(T \leq t)$ one-tail [#TR/MKLOC]	0.055	0.020

The P-values in Table 4 are low (lower than 0.1), which means that the reused blocks have in average lower defect-density than the non-reused ones. We add that 46% of TRs have not registered any block name. Therefore the values for defect-density are not absolute, and they would be higher if all TRs had a valid subsystem or block name.

Table 5 shows the distribution of TRs over severity for blocks (2 of the blocks did not register the severity). The expected values may be calculated by multiplying ‘% of

all’ with the actual number; e.g. we can expect that $0.31 * 435 = 134.85$ of TRs for reused blocks to be of severity A. As shown in Table 5, reused blocks have higher number of severity A defects than expected, while non-reused ones have lower number (167 compared with 190 expected). We performed a Chi-square test to evaluate whether the observed distribution is significantly different from the expected one. The returned P-value is 0.001; i.e. reused blocks have more defects with severity A (the highest priority defects) than expected from the total distribution. The same result is obtained if we perform the test with subsystems.

Table 5. TRs and severity classes for blocks

Severity	Reused	Non-Reused	% of all
A	160	167	31%
B	226	361	55%
C	49	98	14%
Sum	435	626	1

We also tested whether the distribution of TRs is different for pre- and post-delivery defects. The result is in the favor of reused blocks; i.e. they have significantly fewer defects after delivery than expected, with P-values equal to 0.003 and 0.002 for subsystems, and blocks.

5.2. H2 and H3: Reuse vs. component size and defects/defect-density

In this section, we study whether the number of defects or defect-density is correlated with the component size, and whether the result is different for reused, and non-reused components. We defined six null hypotheses in two groups in Table 1. The alternative hypotheses state that there is a relation between the variables of study.

Results: We first examine the relations graphically, and then perform regression analysis. A scatter plots with KLOC on the x-axis, and #TRs on the y-axis for blocks is shown in Figure 2, showing also regression lines, and polynomial functions of order 2 for reused and non-reused blocks. We have similar plots for MKLOC. The gradient of the regression line is higher for non-reused subsystems and blocks, indicating that non-reused blocks are more defect-prone. Table 6 shows a summary of the regression results.

Table 6. Regression results for #TRs and component size

	Subsystem	Block
Adjusted R-Square [KLOC]	0.631	0.491
Regression line, P-value [KLOC]	0.001	0.000
Adjusted R-Square [MKLOC]	0.643	0.590
Regression line, P-value [MKLOC]	0.001	0.000

A study of residual plots confirmed that points are evenly distributed on the both sides of the regression line, and thus the regression analysis is valuable. The P-values for the regression lines are low, meaning that the probability for a random correlation is very low. However the adjusted R^2 values are also low (between 49-64%), which indicate a weak correlation.

We performed the same analysis, but this time with reused and non-reused components separately. The adjusted R^2 was low for reused components (less than 0.70). However, the regression analysis for non-reused components had higher adjusted R^2 , as shown in Table 7. Results in Table 7 indicate that there is a relation between the size of the non-reused components, and #TRs. Plots also indicate that #TRs grow with the component size for this group.

Table 7. Regression results for #TRs and the component size, non-reused blocks

Adjusted R-Square [KLOC]	0.715
Adjusted R-Square [MKLOC]	0.633

The conclusion is that we don't reject **H02-1**, and **H02-2**. For non-reused components, we observe a relation between the number of TRs and the component size, and **H02-3** is therefore rejected. The same results are achieved using one-way ANOVA tests.

Referring to [9], we could explain the rejection of **H02-3** if non-reused blocks were larger than reused ones, but this is not true in our case. Reused blocks are in fact larger (as shown in Figure 2, and verified by statistical tests), and the result should be explained by other factors such as type of functionality or programming language. For reused blocks, Erlang is the dominant programming language, while C is dominant for non-reused blocks. We have studied the type of defects in Erlang and C units, and found that C units have more intra-component defects (defects within a component) than Erlang units, Therefore the number of defects can increase with component size. This needs further study.

Figure 3 shows a plot of defect-density, and component size for blocks. When we plot with defect-density instead of the number of defects, the points are scattered more, and there is no obvious relation between component size, and defect-density. Results for regression analysis between #TRs/KLOC, and size in KLOC for blocks and subsystems is shown in Table 8.

Table 8. Regression results for #TRs/KLOC and component size in KLOC

	Subsystem	Block
Adjusted R-Square [KLOC]	0.036	0.000
Regression line, P-value [KLOC]	0.553	0.455

The P-values for the regression lines are high, while the adjusted R^2 values are low, indicating no relation. Similar results were obtained when we performed the analysis for reused, and non-reused components separately. We conclude that there is no relation between defect-density and component size, and **H03-1**, **H03-2**, and **H03-3** are not rejected.

5.3. H4: Reuse and Stability

Each release of the system adds some features to the previous release, and some bugs are fixed, and therefore the code is modified between releases. As reused components must fulfill requirements for two products, we may assume that they are modified more than non-reused components, and therefore are more fragile.

H04: Reused and non-reused components are equally modified.

HA4: Reused components are modified more than non-reused ones.

Results: We define $MOD = \text{Size of new or modified code} / \text{Total Size of the component}$. We calculated MOD, visualized the results in a scatter plot, and performed t-tests, and ANOVA to evaluate whether there is a significant difference between means of reused and non-reused components. Table 9 shows means, medians, and variances.

Table 9. Descriptive statistics for MOD of blocks

Defect-density	Mean	Median	Variance
MOD, Reused	0.43	0.43	1.87E-2
MOD, Non-Reused	0.57	0.60	2.04E-2

Our study showed that blocks are 49% modified totally; 43% for reused, and 57% for non-reused ones (with KLOC). The distribution is not normal for reused blocks, but variances are not significantly different. A scatter plot with KLOC on the x-axis, and MKLOC on the y-axis for blocks is shown in Figure 4. The gradient of the regression line is larger for non-reused blocks, indicating that they are modified more than reused ones. A two-tail t-test confirms that the difference in means is not zero, with P-value=0.001. A one-tail t-test for blocks assuming equal variances (we test for the hypothesis that reused blocks are modified more than non-reused ones), gives a P-value equal to 0.999. Results show that we can reject both **H04** and **HA4**, and conclude that non-reused components are more modified than reused ones, despite these being specific to one system. One explanation could be that non-reused components have more external interfaces than the reused ones. This must be further studied. We have data for MOD in earlier releases of the

product, and it seems to be relative stable between releases.

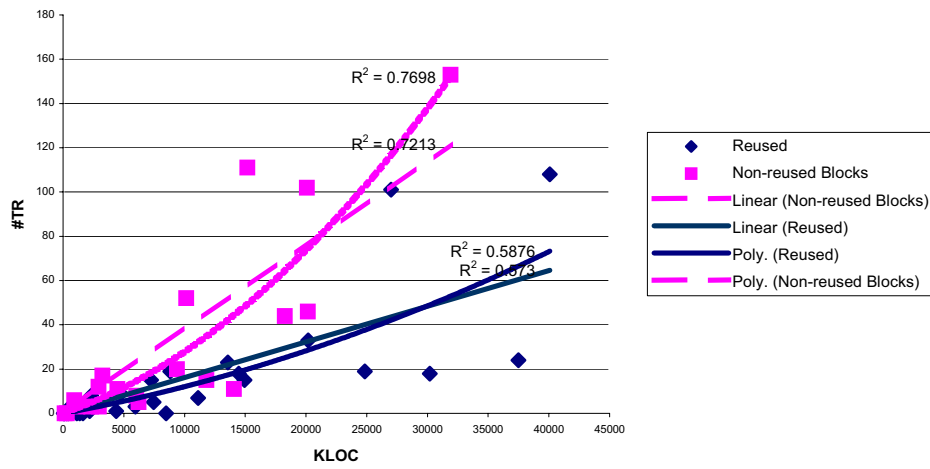


Figure 2. Relation between #TR and KLOC for blocks

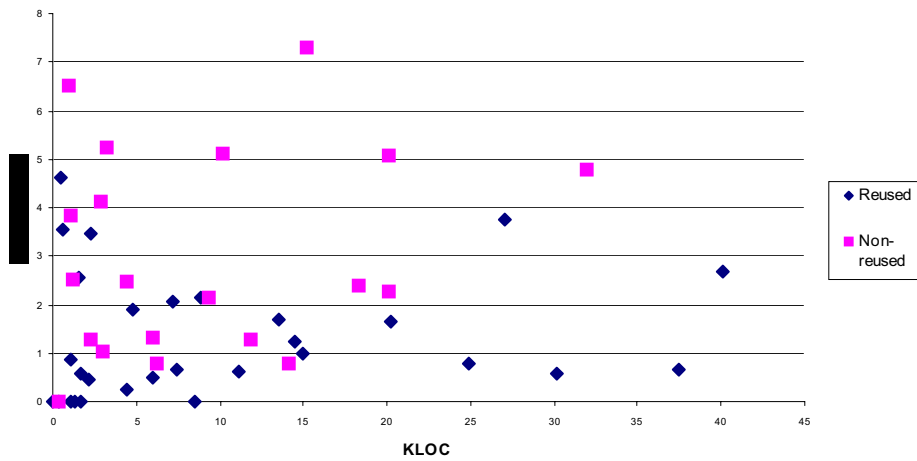


Figure 3. Relation between #TRs/KLOC and KLOC for blocks

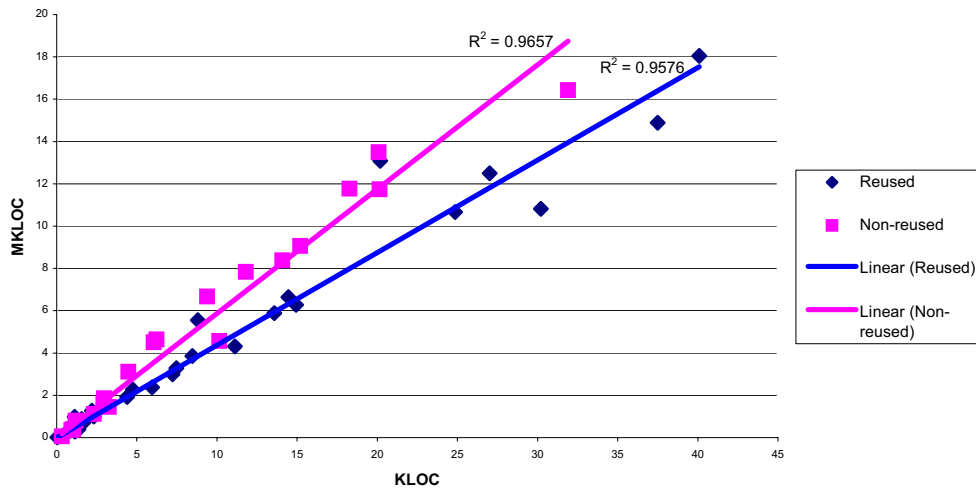


Figure 4. Relation between size of components and size of modified code for blocks

6. Summary and discussion of results

Wohlin et al. [17] describe four types of validity threats in empirical studies. In our study, these threats are:

Conclusion validity: A threat, or more a confounding factor, would be if reused and non-reused components had very different functionality and constraints. For example, non-reused components have user interfaces while some reused components handle other interfaces with complex protocols. Another threat would be if more experienced developers worked with one type of the components. This is not considered as a threat, since the components under study are developed within the same development unit, and by almost homogenous teams. A third threat is that TRs report defects mainly discovered during integration testing, system testing, and maintenance. We don't include data from inspections, and unit testing because this data is not in the same database.

Internal validity: Missing, inconsistent, or wrong data is a threat to internal validity- but mostly missing data. Sometimes gaps in data are systematically related to the behavior to be modeled, or to the nature of the problem. In our case, missing data is because of the process of reporting defects, which does not ask the tester to fill in the missing fields in the reports. This is not considered to be related to the nature of the problem or to introduce a systematic bias. Ways to handle missing data are e.g. mean substitution, regression substitution, or just trying with the existing data, and be aware of the lost efficiency of tests. We chose the last strategy because of two reasons. In some cases we could verify that the distribution of data is not significantly different if all or fractions of data are used. For example, the distribution of defects over severity classes for all data was almost the same as data for subsystems or blocks (which have missing points). The second reason is that our dataset is not too small for comparing the means or correlations. Some other statistical tests are more sensitive to missing data.

Construct validity: We use defect-density and stability as software quality indicators. The weaknesses of this assumption are discussed in Sections 2, and 4. Nevertheless these measures are used broadly in studies.

External validity: The external validity of all hypotheses is threatened by the fact that the entire data set is taken from one company. Our dataset consists of a non-random sample of defect reports (1953 from almost 10,000), and all components of a single release of one product. Two other releases are assessed with similar results. In Section 4, we discussed the possibility to

generalize the results to other releases of the same product, to other products in the same company, and possibly to other companies in the same domain, where the case can be considered as a probable one. There are few published results from such large-scale products to compare with the results.

The remainder of this section discusses the results. Rejection of a null hypothesis does not mean that the inverse is accepted automatically, but we discuss when evidence for such conclusion exists.

H1: Reuse and defect-density. Our results showed that reused components have lower defect-density than non-reused ones (almost 50% less). The difference was less for modified code. Melo et al. [10] report fault-densities from 0.06 for components that are reused verbatim, 1.50 for slightly modified components, and 6.11 for completely new components in their experiment. They concluded also that reused components have lower fault-density. We observed that reused components had more severity A defects than expected from the total distribution, but fewer post-delivery defects. This could mean that defects of these components are given higher priority to fix.

H2: Number of defects and component size. We did not observe any significant relation between the number of defects, and component size for all the components as a group. We conclude there are other factors than size that may explain why certain components are more defect-prone. One factor may be whether the component is reused or not. When reused and non-reused components were analyzed separately, we did not observe any relation between size and the number of defects for reused components, while larger non-reused components had significantly higher number of defects, which may indicate that it is better to break these components down to smaller ones. Factors such as type of functionality or programming language may explain the result.

H3: Defect-density and component size. The plots, and regression analysis showed no relation between these two factors. The result is the same for all components, and for reused, and non-reused ones.

H4: Reuse and stability. Our results showed that reused components are in fact modified less than non-reused ones; i.e. when components are reused across several products, they don't get more fragile, although they should meet requirements from several systems. Stability is important in systems that are developed incrementally, and over several releases.

Based on **H1** and **H4**, we observe that packaging shared functionality into reusable components reduces defect-proneness and improves stability (thus decreasing

the need for modifications). An internal survey of 9 developers in the same organization by us in spring 2002 indicated that developers consider reused components to be more reliable, and stable, in line with the quantitative results. These attributes may be interdependent as other studies show that modified code has more defects than old code [5, 10, 14]. The results of hypothesis testing cannot be used to build causal models, but rather be combined with other types of studies to discuss causes.

We did not observe a significant relation between defect-density, and component size. That is, defect-density cannot be used to predict the number of defects in a component, so other parameters should be studied.

The study also showed the weaknesses of the defect reporting system that has lead to inconsistencies and difficulties in analyzing, and presenting data. A better solution for quality managers (and researchers) would be if the defect reports had automatically been stored in a SQL database from the existing web interface. We may wonder if it is possible to develop a “standard”, minimal metrics (TR schema), which all projects at Ericsson could use. Many interesting hypotheses on reuse, and various software properties were impossible to answer due to the lack of sufficient, and/or relevant data. On the other hand, amassing empirical data without any specific goals or with no post-processing is almost worse than not collecting any data!

7. Conclusions and future work

This study has “data-mined” defect reports, and associated data that had hardly been analyzed or used to a large extent before. We don’t claim that the results are surprising. However, there are few published results on the impact of *reuse* on quality attributes in large industrial projects, so this study is a contribution in that context.

The hypotheses could be used to make a prediction model for future systems in the same environment or for maintaining the current system. For Ericsson, the results of this empirical study may be used to achieve better quality by identifying more defect-prone components (we have not presented the detailed results here), and by taking actions such as inspections or restructuring the components (e.g. split or merge to exploit economy of scale). Higher stability, and lower defect-density of reused components clearly show the industrial advantage of reuse. All these insights represent explicit knowledge based on own data, and thus important for deciding future approaches around reuse. Results can also be used as a baseline for comparison in future studies on software reuse.

8. Acknowledgements

The work is done in the context of the INCO project (INcremental and COmponent-based Software Development [7], a Norwegian R&D project in 2001-2004), as part of the first author’s PhD study, and the MSc thesis of Killi and Schwarz [8] in spring 2003. We thank Ericsson for the opportunity to perform the study, and help to collect the data.

9. References

- [1] Banker, R.D., Kemerer, C.F., “Scale Economics in New Software Development”, *IEEE Trans. Software Engineering*, 15(10), 1989, pp. 1199-1205.
- [2] Endres, A., Rombach, D., *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*, Pearson Addison-Wesley, 2004.
- [3] Fenton, N.E., Ohlsson, N., “Quantitative Analysis of Faults and Failures in a Complex Software System”, *IEEE Trans. Software Engineering*, 26(8), 2000, pp. 797-814.
- [4] Flyvbjerg, B., *Rationalitet og Magt I- det konkrates videnskab*, Akademisk Forlag, Odense, Denmark, 1991.
- [5] Graves, T.L., Karr, A.F., Marron, J.S., Siy, H., Predicting Fault Incidence using Software Change History. *IEEE Trans. Software Engineering*, 26(7): 653-661, July 2000.
- [6] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12, 1990.
- [7] INCO project (INcremental and COmponent-based Software Development), <http://www.ifi.uio.no/~isu/INCO/>
- [8] Killi, O.M., Schwarz, H., “An Empirical Study of Quality Attributes of the GSN System at Ericsson”, MSc thesis, 109 pages, NTNU, June 2003, available on: <http://www.idi.ntnu.no/grupper/su/su-diploma-2003/index.html>
- [9] Malaiya, K.Y., Denton, J., “Module Size Distribution and Defect Density”, *Proc. 11th International Symposium on Software Reliability Engineering- ISSRE '00*, 2000, pp. 62-71.
- [10] Melo, W.L., Briand, L.C., Basili, V.R., “Measuring the Impact of Reuse on Quality and Productivity on Object-Oriented Systems”, Technical Report CS-TR-3395, University of Maryland, 1995, 16 pages.
- [11] Mendenhall, W., Sincich, T., *Statistics for Engineering and the Sciences*, Prentice Hall, 1995.
- [12] Mohagheghi, P., Conradi, R., “Experiences and Challenges in Evolution to a Product line”, *Proc. 5th International Workshop on Product Line Development- PFE 5*, 2003, Springer LNCS 3014, pp. 459-464.
- [13] Neufelder, A.M., “How to Measure the Impact of Specific Development Practices on Fielded Defect Density”, *Proc. 11th International Symposium on Software Reliability Engineering (ISSRE '00)*, 2000, pp. 148-160.
- [14] Ostrand, T.J., Weyuker, E.J., “The Distribution of Faults in a Large Industrial Software System”, *Proc. The International Symposium on Software Testing and Analysis (ISSTA '02)*, ACM SIGSOFT Software Engineering Notes, 27(4): 55 – 64, 2002.
- [15] Succi, G., Benedicenti, L., Valerio, A., Vernazza, T., “Can Reuse Improve Reliability?”, Fast abstract in *International Symposium on Software Reliability Engineering (ISSRE '98)*, 1998, available on <http://www.chillarege.com/fastabstracts/issre98/98420.html>

[16] The Great Computer Language Shootout, 2003,
<http://www.bagley.org/~doug/shootout>

[17] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers, 2000.