An Empirical Investigation of Software Reuse Benefits in a Large Telecom Product

PARASTOO MOHAGHEGHI
and
REIDAR CONRADI

Background. This paper describes a case study on the benefits of software reuse in a large telecom product. The reused components were developed in-house and shared in a product-family approach. *Methods*. Quantitative data mined from company repositories are combined with other quantitative data and qualitative observations. *Results*. We observed significantly lower fault-density and less modified code between successive releases of reused components. Reuse and standardization of software architecture and processes allowed easier transfer of development when organizational changes happened. *Conclusions*. The study adds to the evidence of quality benefits of large-scale reuse programs and explores organizational motivations and outcomes.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification- *reliability*; D.2.8 [**Software Engineering**]: Metrics- *product metrics*; D.2.13 [**Software Engineering**]: Reusable Software-*reuse models*, K.6.3 [**Management of Computing and Information Systems**]: Software Management-*software maintenance*
General Terms: Reliability, Measurement
Additional Key Words and Phrases: Software reuse, fault-density, product family, risks, standardization

---

1. INTRODUCTION

This paper describes results of an empirical investigation of data collected from a large industrial telecom product to evaluate and explore software reuse benefits. A product family was initiated across two Ericsson organizations in Norway and Sweden based on extracting reusable components in a first product and developing a reusable, layered architecture to achieve benefits in productivity and lead time. Apart from the above incentives, two benefits are discussed in this paper: a) reuse led to benefits in quality, and b) reuse of software architecture, processes, infrastructure and domain knowledge showed to be beneficial for facing organizational changes.

This research was part of the first author's PhD study, which was performed at Ericsson in Grimstad, Norway and was supported by the INCO project (INcremental and COmponent-based Software Development, a Norwegian R&D project in 2001-2004), http://www.ifi.uio.no/~isu/INCO/.
First author's address: 1) SINTEF ICT, P.O.Box 124 Blindern, NO-0314 Oslo, Norway, 2) The Norwegian University of Science and Technology, Department of Computer and Information Science, NO-7491, Trondheim, Norway. parastoo.mohagheghi@sintef.no.
Second author's address: The Norwegian University of Science and Technology, Department of Computer and Information Science, NO-7491, Trondheim, Norway. conradi@idi.ntnu.no.
Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2001 ACM 1073-0516/01/0300-0034 $5.00

Quantitative data on fault-density and code modification rate were collected for several releases from the repositories of the organization in Norway. The results of the quantitative analysis showed that reused components had significantly lower fault-density than the non-reused ones and less modified code between successive releases. These benefits in quality should promote initiating future reuse programs. There is a lack of published results from industrial case studies, especially from large-scale products and over time, as stated in [Ramesh et al. 2004] and verified by our literature review. One contribution of this study is thus providing empirical evidence of reuse quality benefits in a large-scale industrial product.

Apart from verifying quality benefits, the case study reveals benefits of reuse as a risk mitigation strategy when initiating new products and facing challenges such as organizational restructuring. Reuse in-the-large was a difficult decision to take, considering the size of the products, the desire to invent rather than to reuse in the second organization, and the effort needed to coordinate development plans and teams across organizations and countries. Nevertheless, knowledge transfer and standardized infrastructure and development processes helped co-locating development when Ericsson decided to downsize its organization worldwide and gather all development of these products in Sweden.

The quantitative analysis revealed two problems in the data: 1) insufficient definition of metrics related to reuse and incremental development, and 2) ineffective data collection routines and difficulties in integrating data from several sources in spite of exploiting widely used commercial tools. These problems restricted the study and required development of special tools for data processing. They also affected validity of the results as discussed in Section 5.4. Neither of these problems is specific to this case study and a descriptive study might inspire others to reflect on how their data collection can be improved.

The context of the study is described in more details in [Mohagheghi 2004c]. Some of the quantitative results are published in [Mohagheghi et al. 2004a]. This version is expanded in terms of the literature review, new data, discussing the rationale, methods and contributions of the study in more details, and discussing the lessons learned regarding data collection.

The remainder of this paper is organized as follows: Section 2 discusses the motivation of the study and the research challenges and presents an overview of earlier studies. Section 3 presents the research method, the hypotheses of the study and related

work on the hypotheses. Section 4 provides a description of the context, software processes, data collection and processing. Section 5 is dedicated to quantitative analysis, results and discussion of them. The paper is concluded in Section 6 with a summary of contributions and directions for future work.

2. MOTIVATION, RESEARCH CHALLENGES AND RELATED WORK

2.1. Motivation

Software reuse (reuse in the remainder of the paper) is the systematic use of existing software assets to construct new assets or products. Software assets in this view may be source code or executables, design templates, software architectures or any other asset. Doug McIlroy first introduced the idea of systematic reuse as planned development and widespread use of software components in 1968 [McIlroy 1969]. Today, reuse encompasses a variety of approaches, from reusing components developed in-house, to reuse of Commercial-Off-the-Shelf (COTS) or Open Source Software (OSS) components. Product family engineering is reuse at the largest level of granularity, relevant for developing systems that share software architecture. Reuse and product family engineering are especially relevant to increase productivity and reduce lead time and thus develop sooner and cheaper.

With *incremental* (and *iterative*) *development* we mean delivering a subset of requirements in a working system, called a *release*. Incremental development reduces the risk of changing requirements and assists in developing systems with the correct and prioritized subset of functionality. Some define incremental development as a type of reuse since new releases of assets or products are based on the previous releases [Basili 1990] [Frakes and Terry 1996]. However, reuse in this paper means reuse of assets in more than one product, and not maintenance or evolution of the same product.

Ommering and Bosch summarized the driving forces of proactive, systematic, planned and organized approaches towards reuse for sets of products in Figure 1 [Crnkovic and Larsen 2002- Chapter 11]. Complexity and quality are best handled with an explicit software architecture, while quality, diversity of products and lead-time reduction are achieved by reuse of software components. We have added the dashed elements in Figure 1 to their model:

- The need for more standardized products promotes an explicit (reusable) architecture and reuse of components. Jacobson et al. [1997] wrote that several reuse programs are using reusable software to impose internal technical

standards. We add also standardization of non-technical assets such as processes and skills.

- By investing in a common core of software components and processes, organizations share the risks and costs of large-scale development, divide the work between them and increase their organizational flexibility. This is discussed as one of the outcomes of the study.

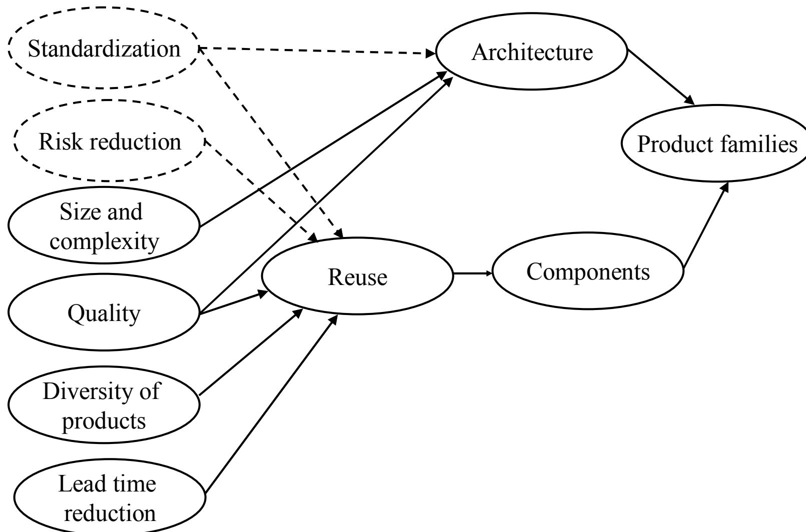

Figure 1. Basic arguments for software product families by Ommering and Bosch and extended by us

Product families face some of the challenges of other reuse-based approaches, where decision-making for initiating and maintaining any reuse program should be based on a cost-benefit analysis. While costs are often tangible, benefits may be intangible (not directly measurable) and are most often received over time. We search for benefits as the main research question of this study.

2.2. Research challenges

The *Research Question (RQ)* of the study is defined as:

***RQ**. Do we have evidence on the benefits of reuse in the context of this study?*

With benefits we mean reduced time-to-market, reduced cost of development and maintenance, improved quality or productivity, or any other observed benefits. Because of company affiliation and the permission to perform the study, we had access to the fault reporting system, the configuration management system for source code and some data collected by the company's quality management team. Based on the research question, literature review presented in Section 2.3 and a pre-study of available data, we selected fault-density, code stability or modification rate, and requirement modification or density

of it as *software quality attributes* for quantitative analysis. The research question is deployed in hypotheses for quantitative analysis in Section 3 and is answered in Section 5.6 based on the quantitative evidence. It is also answered by a qualitative discussion of risk reduction as a benefit of reuse in Section 4.1.

The context is described in Section 4 and may be characterized as a large telecom product that is developed incrementally, with reuse in a product family approach. To understand why certain data were collected or not, we discuss the two following *Context Questions* (*CQ*) on the motivations behind starting a reuse program and its evolution over time:

CQ1*: How the reuse program was initiated and implemented?*

CQ2*. What was the impact of incremental development on data collection?*

CQ1 and *CQ2* are answered in Sections 4.1 and 4.5 respectively. In addition to exploration and explanation, we look for a degree of prediction, formulated in the following *Validity Question* (*VQ*), which is answered in Section 5.5:

VQ*. Do results of assessments of one or several releases have prediction value for future releases?*

We believe that the research question is worth of study reflected by the extent of attention it has received in literature. The other questions are relevant for discussion of the results and generalization to future releases.

2.3. State of earlier work

There is extensive literature on reuse, from books [e.g., Bosch 2000; Clements and Northrop 2001; Jacobson et al. 1997; Karlsson 1995], to the reports of the Software Engineering Institute (SEI) [SEI 2005] and the Fraunhofer Institute for Experimental Software Engineering (IESE) [IESE 2005], and to papers in various technical journals and conferences. This section presents results of earlier case studies and experiments published in journals and conferences relevant for this study.

Our sources have been the *ACM digital library* and the *IEEE Xplore* which also include many conference proceedings, *Empirical Software Engineering Journal*, *Journal of Systems and Software*, *Journal of Information Science*, *MIS Quarterly (MISQ)* from September 1994 (online), *IEEE Transactions of Software Engineering* (TSE), *IT Professional*, *ACM Computing Surveys (CSUR)*, and the *Journal of Research and Practice in Information Technology* (from 2003 online). We searched the above sources with keywords (e.g., “reuse”, “reuse benefits” and “reuse case study”). To assure better coverage, proceedings of the *International Conference on Software Reuse (ICSR)*, the

IEEE International Conference on Software Maintenance (ICSM) since 1995 online, the *International Software Product Line Conference (SPLC)* started in 2001, the *International Conference on Software Engineering (ICSE)* since 1995 online, the *International Conference on COTS-Based Software Systems (ICCBSS)* started in 2002, *MISQ* and the *IEEE Software* magazine were manually checked. We only discuss studies published between 1994 and 2005. [Frakes and Terry 1996] and [Hallsteinsen and Paci 1997] provide a survey of older studies. We excluded interviews, surveys and papers merely discussing how to implement reuse, holding on observed benefits in industrial studies or experiments. We also include results of [Hallsteinsen and Paci 1997] in Table 1, being published in a book.

The results of the search (11 papers) are shown in Table 1, with this study in the last row of the table. Note that the studies use the terms “defect”, “error” and “fault” when referring to problems and we kept the original terms in the results column. The size of software is often given in Kilo Lines of non-commented source Code (KLOC). Although reuse has been discussed for years, published results on quality benefits are few. In Table 1, only the study of Ramachandran and Fleischer [1996] is to some degree comparable to this case study in scale and reuse scenario (product family) but it has very little quantitative data. Long-term effects of reuse are also less studied. Lim [1994] reports productivity gains over several years, while Selby [2005] have collected data for several years of development, but does not report results as releases of the same product; only in the same environment. This scarcity of evidence increases the value of the current study.

Table 1. Overview of related case studies and experiments on software reuse

Authors & date	**Subject**	**Main results**
Lim [1994]	A case study of two internal reuse programs at HP. 1. HP product of 80 KLOC written in Pascal and SPL with 68% reuse. 2. HP product of 64 KLOC written in C with 31% reuse.	1. 0.9 defects/ KLOC for reused and 4.1 for non-reused code, 57% increase in productivity. 2. 0.4 defects/ KLOC for reused and 1.7 for non-reused code, 40% increase in productivity and 42% reduction in time to market.
Basili, Briand and Melo [1996]	A student experiment including eight small (less than 15 KLOC) projects with reuse rates from zero up to 64%. Reused components were C++ and GNU libraries. Data were also analyzed in [Devanbu et al. 1996].	Error density was 0.125 in verbatim reused code, 1.50 in slightly modified, 4.89 in extensively modified and 6.11 in newly developed code. Each 10% increase in reuse rate decreased rework per fault by 8.5%. A strong relation between productivity and reuse rate was observed.
Ramacha ndran and	A case study of four telecom products with reuse of large-scale	Stable system architecture is discussed as a benefit. No quantitative data on quality or

Authors & date	Subject	Main results
Fleischer [1996]	building blocks (verbatim or modified) in a product family approach. Verbatim reuse was 47-84% and software size was 189-310 KLOC. Programming language is not given.	productivity are given.
Thomas, Delis and Basili, [1997]	A case study of seven medium-scale Ada projects (12.8-27.1 thousand Ada statements or 61-184 KLOC) with 4-89% verbatim reuse of statements and 31-100% total reuse (verbatim and modifications).	90% reduction in error density for verbatim reused code, and 50% for slightly modified code and less relative rework in both cases. However, they observed increased correction effort per error associated with reuse, but the cost is outweighed by the benefit of the reduced number of errors.
Hallsteinsen and Paci (Eds.) [1997]	Twelve small-scale reuse studies in industry for internal reuse. Applications were in Small talk, C, C++, SDL, Ada and COBOL 2.	Three reuse scenarios are identified: 1) product family, 2) component-based application development, and 3) application framework. Scenarios 1 and 3 cover 6 experiments. Qualitative benefits (rapid introduction of new products, better quality because new applications are built from proven parts, and consistency of applications' content and their look-and-feel) are largely recognized, while quantitative results are few. Scenarios 1 and 3 led to higher reuse rates and savings, while they required higher up-front investments.
Frakes and Succi [2001]	A case study of four industrial data sets of C and C++ modules (maximum 16 modules). Reuse means reuse of functions.	Reuse resulted in higher quality (measured subjectively by developers and less error-density and module changes). The impact on productivity was ambiguous.
Succi, Benedicenti and Vernazza [2001]	The study compares two products in the same product family in an Italian medium sized software company. One product (99 KLOC) was developed using the company's standard development cycle (including also ad-hoc reuse) while the other product (101 KLOC) was developed after the development of a domain-specific library. Programming language is not given.	The product developed with domain-specific library and systematic reuse had 44% less customer complaints per LOC than the product developed with ad-hoc reuse.
Morisio, Romano and Stamelos [2002]	An exploratory case study where a single programmer developed five small applications based on an object-oriented framework and four based on traditional component development. Effort was 16-95 person-hours.	Productivity and quality in terms of effort needed to correct defects were substantially higher when using the object-oriented framework.
Baldassarre, Bianch, Caivano and	The study compares two projects in an Italian SME, one with a reuse-oriented development (ROD) process and one without it, otherwise similar in language	Software developed with ROD had lower Mean Cyclomatic Complexity-MCC (no discussion of why). Overall higher apparent productivity was observed with ROD (after populating the reuse repository).

Authors & date	Subject	Main results
Vissaggio [2005]	(COBOL), specification and design technique. Size is not given.	
Selby [2005]	Data are from 25 software systems from a NASA development environment, ranging from 3 to 112 KLOC of Fortran source code. In addition to analyzing data on project level, data from 2954 modules with complete data on their development are analyzed to evaluate reusability.	Modules reused verbatim had in average 98% less faults than modules newly developed. For modules slightly modified the result was 55%. Fault isolation effort per module was in average 99% less for verbatim reuse and 50% less for modules slightly modified compared to new modules. The number of changes per module was in average 94% less for verbatim reuse and 26% less for modules slightly modified compared to new modules.
Zhang and Jarzabek [2005]	A product line was initiated by identifying similarities and differences among four small games in an extractive way and these products were developed again based on the product line architecture (PLA). Further, a new game was developed twice, once based on the PLA and once without it. The J2ME platform (Java) is used for implementation.	Design quality as measured in OO metrics (complexity, coupling) was almost the same after using the PLA compared to before. LOC was reduced by 26.5%. Memory usage decreased from 0.6% to 19% in four games and the games run 2.7% to 10% faster than before. Effort was reduced by 68% for the game developed twice.
Mohagheghi and Conradi [2006]- this paper	Data are from three releases of a telecom product of 408 to 480 KLOC programmed in Erlang, C and Java. This product shares reusable components with a second product in a product family, and the reuse rate is around 60%.	Components in the study are on the subsystem and block level. Using mean values, the fault-density of reused blocks was 44% to 61% of the non-reused ones, less difference for modified code. Reused blocks were in average 43% modified between two successive releases, compared to 57% for the non-reused ones. The number of change requests per subsystems was lower for the reused components, but not when divided by the subsystem size. Reuse and standardization of software architecture and processes facilitated organizational change, and is considered as a risk mitigation strategy.

In Table 1, increased productivity, lower fault/defect-density, lower number of changes per module or per LOC, reduced development and maintenance effort, reduced complexity, and consistency of applications and the software architecture are reported as reuse benefits. When it comes to risks, the literature on reuse discusses the risks of reuse (disincentives), but not reuse for risk mitigation. In addition to reduced lead time or cost, reuse may reduce project risks by developing based on well-tested products, and sharing the risks and costs of large-scale development. A more detailed analysis of reuse effects in industrial cases is presented in [Mohagheghi and Conradi 2007].

3. RESEARCH METHOD, HYPOTHESES AND RELATED WORK

The research question in this paper is evaluating and exploring the benefits of software reuse in the context of the study. Various literature on case study research has given input to our analysis and discussion, especially [Yin 2003; Flyvbjerg 2004; Cooper and Schindler 2001; Kitchenham et al. 1995; Kitchenham and Pickard 1998; Lee and Baskerville 2003]. Generalization is transferring the results to other releases, other contexts, or to models and theories.

A case study may be *exploratory* (finding parameters relevant in a relation), *explanatory* (discussing why certain methods are chosen and how they are applied), *informative* (describing the practice), or *confirmative* (verifying theories). This case study is confirmatory in verifying reuse benefits already studied, and exploratory in searching for new variables.

Table 2 presents five hypotheses and related work is described below. Whenever we use the term *reused components* in this paper, we mean reuse in a product family approach, where reused components are developed in-house, reused as-is in a layered architecture (see Section 4), and shared with other products.

Table 2. Research hypotheses for quantitative analysis

HypId	Hypothesis text
H1	Reused components have a lower fault-density than the non-reused ones.
H2	Reused and non-reused components have equal code modification rate between releases.
H3	Reused and non-reused components are equally affected by requirement modification.
H4	**H4-a:** There is no relation between the number of faults and component size for all components. **H4-b:** There is no relation between the number of faults and component size for reused components. **H4-c:** There is no relation between the number of faults and component size for non-reused components.
H5	**H5-a:** There is no relation between fault-density and component size for all components. **H5-b:** There is no relation between fault-density and component size for reused components. **H5-c:** There is no relation between fault-density and component size for non-reused components.

H1. **H1** is based on previous work. Several studies have confirmed lower fault-density (or error/defect-density) of reused components [Lim 1994] [Basili et al. 1996]

[Thomas et al. 1997] [Frakes and Succi 2001] [Selby 2005], but not in a product family, and only Selby's study [2005] is large scale. Component size may be a confounding factor here which is the reason behind studying **H5**. Reused components may be designed more thoroughly and better tested, and the accumulated fault fixes in several releases also results in a higher product quality [Lim 1994].

H2 and H3. One of the reasons behind reuse is to reduce changes to software and reducing rework effort by packaging less change-prone software in separate modules or components that may be reused verbatim or with minor modifications across releases and products. Metrics of software change are therefore used as quality indicators of reuse programs such as in [Thomas et al. 1997] and [Selby 2005]. Graves et al. [2000] 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 and time elapsed since changes), while size and complexity metrics were not useful in such prediction. The studies of Ostrand et al. [Ostrand and Weyuker 2002, Ostrand et al. 2004] showed higher fault-density for new files than for older files and that fault-density tended to decrease as the system matured. **H2** and **H3** investigate code modification rate between releases and the number of requirement changes as metrics of software change. Reused components may be modified more to meet requirements of several products, or modified less since application specific functionally may be more volatile. Our expectation is that reused components become more stable as the system matures. In the study of Zhang and Jarzabek [2005], the total size of code was reduced after introducing a product line approach, but there is no explanation of why and the study covers only one release.

H4. Component size (in LOC here) may be a confounding factor when discussing the number of faults or its density. Some studies report a relation between the number of faults or fault-density and component size, while others do not. The diversity of results makes the question relevant to be evaluated for this context. Fenton and Ohlsson [2000] studied a large Ericsson telecom system and reported that size weakly correlated with the number of pre-release faults, but did not correlate with post-release faults. Ostrand et al. [2004] studied faults of 17 releases of an inventory tracking system at AT&T and found that size was the strongest individual factor for predicting the number of faults, followed by whether a file was new, changed or unchanged. Rosenberg [1997] proposes using LOC not as a predictor of quality by itself, but rather as a covariate adjusting for size in

using another metric. Selby [2005] reported that modules reused verbatim had the fewest number of faults and were the smallest in size. Selby's interpretation of the results is that smaller size may facilitate module reuse, as well as other characteristics such as simpler interfaces and less interaction with other modules.

H5. Fenton and Ohlsson [2000] and Ostrand et al. [2004] did not observe any relation between fault-density and module size. A number of studies have reported higher fault-density in smaller components than in larger components (see Thomas et al. [1997] for an overview). Rosenberg [1997] writes that there must be a negative correlation between the values of *X* and *1/X* and therefore caution is needed when studying size versus fault-density. The above studies are not related to reuse. Thomas et al. [1997] studied the question for reused vs. new or extensively modified code and concluded that lower fault-density of reused classes in their study did not result from their smaller sizes. Malaiya and Denton [2000] 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).

We perform an evaluation of the hypotheses using the raw data, inferential statistics and qualitative discussions. Gigerenzer [2004] and Dybå et al. [2006] discuss that most studies performing statistical hypothesis testing have only reported the observed level of significance; i.e., the *p-values*, or used fixed threshold values such as 0.05 or 0.10 for rejecting hypotheses without discussing the power of the tests, observed effect sizes or presenting descriptive statistics of their data. *Effect size* refers to the practical significance or meaningfulness of the observed difference and is not discussed in earlier studies. We report the p-values, present a descriptive statistics of the data, discuss the effect size whenever possible, and present our conclusions. For large effect sizes, it is not necessary to discuss the power of the test.

The results of statistical tests must be interpreted with care for generalization. One precondition of most statistical tests is the random assignment of subjects to treatments. There is no random assignment of components to treatments (reuse or not) in this case

study, and not in the other studies in Table 1 as well. Random assignment is generally difficult in industrial settings. Nevertheless, statistical tests are often applied for strengthening confidence on the results.

Microsoft Excel and Minitab are used for data visualization and statistical analysis. We used guidelines provided by the SPIQ project in our analysis [SPIQ 2000].

4. THE CONTEXT OF THE STUDY

In this section, the Ericsson context is described with an overview of the products, and the software development and maintenance process(es). We will also answer *CQ1* and *CQ2*, and partly the *RQ*.

4.1. How the reuse program was initiated and implemented? Answering *CQ1* and *RQ* regarding risk reduction

The products in the family are network nodes providing packet data services for mobile networks. The service allows a mobile station to be connected to an Internet Service Provider (ISP) or a corporate Local Area Network (LAN). Ericsson developed the first product to provide packet data services for the existing GSM network (Global System for Mobile Communication). When requirements for a new network (W-CDMA or Wideband Code Division Multiple Access; the so-called third generation- 3G mobile network) were later formulated, the similarities in requirements encouraged the company to study the developed solution for identifying reusable components that could also serve the upcoming network. It was decided to develop two variants of the product for the two networks as shown in Figure 2. These two variants share software architecture, software process, system platform and components in the two middle layers, and are considered as products in a product family. The system platform was developed by Ericsson, but is handled as an OTS (Off-the-Shelf) component (here) since its development was not part of the product family development. In 2004, 288 operators around the world had mobile packet services, where Ericsson was the supplier to over 110 of these.

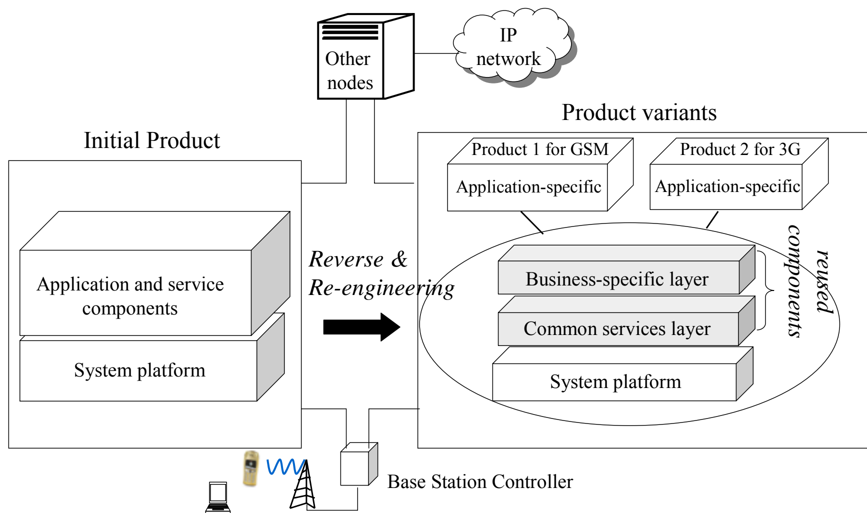

Figure 2. Transition from single product development to product family development

In Figure 2, the *common services* layer extends the system platform by providing additional middleware services. It also includes a component framework that facilitates building applications and is not specific to mobile networks, thus being generic for future reuse and represents *horizontal reuse*. The *business-specific* layer provides services specific for mobile packet data that are shared between the two products, representing *vertical reuse*. Components in the common services and business-specific layers are hereafter termed *reused components* (reused in two distinct products). All the code of the reused components was included in each product and was thus reused "as-is". Components in the *application-specific layer* are specific to each product and are termed *non-reused components*. Note that the products are nodes in a network and have no direct user interfaces, except for operation and maintenance such as for software upgrades.

Approaches to initiate a product family may be divided into *heavyweight* where commonalities are identified first, and *lightweight* with using mining efforts in an initial product to extract commonalities [McGregor et al. 2002]; alternatively as being *proactive*, *reactive* and *extractive* [Krueger 2002]. Krueger claims that being extractive (i.e., lightweight) can reduce the barrier to large-scale reuse, as it is a low-risk strategy with lower upfront cost. Earlier, Johnson and Foote [1998] wrote that useful abstractions are usually designed from the bottom up; i.e., they are discovered and not invented. Ericsson chose the extractive or lightweight approach since similarities in requirements between an emerging product and an existing one encouraged the company to evaluate reuse. When the reuse program is initiated, the company moves to a reactive or proactive

phase; i.e., planning for reuse. For comparison, from the papers in Table 1 who have reported the approach to initiating reuse, [Lim 1994] reported an incremental reactive approach and [Zhang and Jarzabek 2005] reported an extractive one.

Discussions and evolving the initial architecture to a layered one took almost a year to fulfill, but it had a beneficial impact on the architecture by leading to a clearer abstraction of the developed components, which is also observed in the Hallsteinsen and Paci's [1997] studies. Mintzberg [1993] identified *standardization of processes* (how work is done), *inputs* (skills of persons performing tasks) and *outputs* (final results) as three fundamental approaches in coordination of activities between organizations. To reuse, software development and maintenance processes were standardized in the two organizations and the infrastructure to support this was developed; including multi-site ClearCase and ClearDDTS, shared intranet for the software process (adapted Rational Unified Process or RUP [Rational 2005]) and shared templates. Knowledge sharing (standardization of inputs) across remote organizations required high quality documentation. There is a reuse cost not quantitatively assessed, e.g., for developing reusable software architecture and components, standardization, and coordination of plans. We discussed in [Mohagheghi 2004c] that in spite of sharing RUP, it was not adapted for reuse and proper metrics for reuse were not defined, which identifies need for future work.

At the time of the study, reused components were mostly developed by an Ericsson organization in Norway. Application-specific components of Product 1 were mostly developed in Norway and the other product (Product 2) in Sweden. All development was moved to Sweden shortly after this study. This transfer was facilitated by the fact that the two products shared software architecture and development processes, although some support from the developers in Norway was necessary. We therefore consider reuse as a risk mitigation strategy in the time of organizational changes, as part of our answer to *RQ*.

Data used in this study cover three years of development of the reused components in the two products and the application-specific components of Product 1. Reorganizations in Ericsson restricted access to data on the application-specific components of Product 2 and stopped further research.

4.2. Components in the study

The architecture is component-based (CORBA) and all the components in this study are built in-house. Each product is decomposed hierarchically into subsystems, blocks, units

and modules (source files). A *subsystem* is the highest level of encapsulation used. It has formally defined interfaces expressed in the Interface Definition Language (IDL) and is a collection of blocks. A *block* has also formally defined interfaces expressed 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 middle-level (blocks) components. Since communication inside blocks is more informal and may happen without going through an interface, blocks are considered the lowest-level components.

Components are mainly programmed in Erlang and C, and a few software modules (GUIs) in Java. Erlang is a functional language developed originally by Ericsson for programming concurrent, real-time, distributed fault-tolerant systems. Sometimes the term *equivalent code* is used for the size of products developed in multiple programming languages. To calculate the "equivalent" size in C, the software size in Erlang is multiplied with 3.2, Java with 2.4 and IDL with 2.35, according to the practice in the organization. Other studies have used other factors. However, the results of our quantitative analysis did not show any significant difference using pure LOC or the equivalent ones. We present the results for pure LOC in this paper.

4.3. Software development and maintenance process(es)

The two products are developed incrementally and new features are added to each release. Each release is maintained for a while, mostly doing corrections, before being replaced with a new one. RUP is an incremental software process with four phases: *inception*, *elaboration*, *construction* and *transition*. Each release is developed in 5-7 iterations of 2-3 months with major up-front specification as recommended by RUP. Figure 3 shows a simplified view of the software development, maintenance and change control processes:

- The thick continuous arrows show the development path and the circles show activities performed in each release.
- The thin continuous arrows show activities related to corrective maintenance; i.e., initiating and solving *Trouble Reports* (TRs).
- The thin dashed arrows show activities related to changing requirements or optimizing software in each release as reflected in *Change Requests* (CRs).

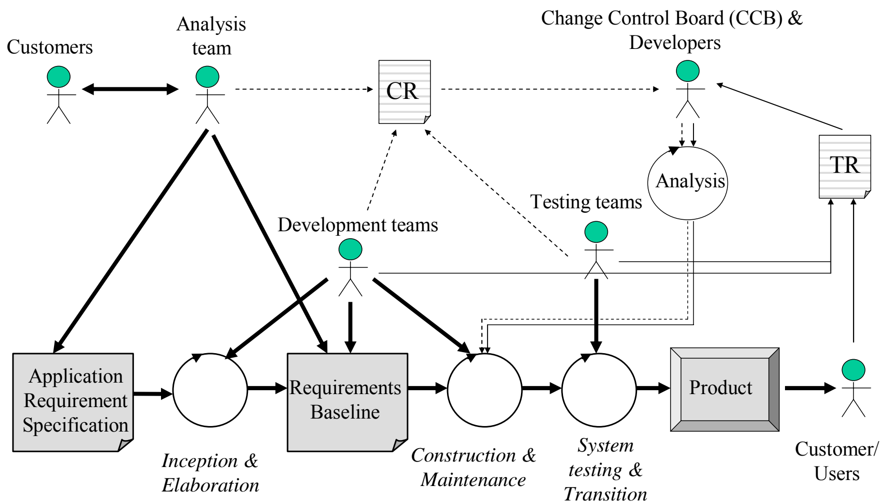

Figure 3. The software development and maintenance processes

In the remainder of this section, we describe the processes in more details. The analysis team specifies the initial requirements for each release in a textual document called the Application Requirement Specification (ARS). These requirements are refined iteratively during the inception and elaboration phases by the analysis and development teams, resulting in artefacts such as use case models and supplementary specification documents for non-functional requirements. At the end of the elaboration phase, the ARS and the detailed set of the requirements define the requirement baseline for a release. These requirements are implemented by development teams and tested, covering unit testing, integration testing and use case testing. System tests, acceptance tests including all nodes in a network and other tests (e.g., reliability or efficiency tests) are performed by separate testing teams. All developed and generated source code is stored in the Rational ClearCase configuration management tool.

Changes to requirements after the baseline are handled through formalized CRs. Examples of proposed changes are adding, modifying or removing a functional requirement or improving a quality attribute such as performance or availability. CRs reflect changes that can significantly impact the contents of a release, its cost or schedule. CRs are written in MS-word and contain a description of the current situation and the proposed change, documents and components affected (mostly subsystems), and an estimate of the effort needed to implement the CR. CRs are analyzed by the *Change Control Board* (CCB) and with support from developers. If accepted, a CR is forwarded to the development teams for implementation. If rejected, the CR is closed by the CCB.

Development teams also resolve *Trouble Reports (TRs)*, which are categorized under corrective maintenance. Failures (deviations from specifications or expected behaviour) observed by test teams or by users after delivery are reported as TRs. TRs are written for all sources of problems (software, hardware, toolbox and documentation) and there should be only one problem per TR.

TRs are analyzed by development teams and the cause of a failure (usually called a *fault* in literature) is stored in the TR. Sometimes a TR shows to be due to an already known fault, which results in labelling it as a *duplicate*. When duplicates are removed from the data, the remainder of TRs show the actual faults. In [Mohagheghi et al. 2004a] we used the term *defect* (instead of TR) as this term is widely used in other studies, although meaning different things. It is replaced by *fault* in this paper to avoid confusion between failures and faults. Terminology is inconsistent regarding reported problems as discussed in [Mohagheghi et al. 2006]. Therefore, it is important to notice that faults in this study refer to the causes of problems, which are reported during system testing and operation.

TRs are stored in the Rational ClearDDTS tool using a web interface. Each TR contains information about the product name, release number, when the problem was detected, severity (see below), type selected from a list (coding, documentation, wrong design rule applied etc.), the assumed faulty artefact (name of the document, software module, block or subsystem), estimated effort to correct it, and a description of the observed problem. Three different severities are defined: *A* (urgent TRs with highest priority that brings the system down or affect many users), *B* (major TRs that affect a group of users or restart some processes) and *C* (minor TRs that do not cause any system outage).

4.4. Data collection and processing

Because of affiliation of the first author in the company, we had good insight in the data collection routines. For CRs, their status and a short summary are given in html pages. A tool written in C#[1] parsed the html pages and inserted relevant fields in a SQL database. For other fields that were not given in the html pages, the associated data were inserted manually in the databaseT[2]. CRs for four releases of Product 1 (totally 169 CRs) were analyzed regarding origin (internal or external), functional or non-functional

[1] Tools in this study were developed by two master students (Schwarz and Killi) and the data were collected in spring 2003.

[2] Jointly by the first author and the two master students.

requirements, and acceptance rate [Mohagheghi and Conradi 2004b]. This paper only presents one hypothesis related to reuse (**H3**).

TRs are stored as plain text files and facilities for search were therefore restricted. A tool written in C# traversed the text files, extracted all the fields and created a summary text file. The summary was used to get an overview of the raw dataset and to decide which fields were relevant to the study. The exploration revealed a lot of inconsistencies in the TRs. For example, a subsystem is stored as 'ABC' or 'abc' or 'ABC_101-27'. Another major weakness showed to be the difficulty to track TRs to software modules without reading all the attached files or parsing the source code. Each TR has a field for the faulty software module, but this field may stay empty or not be updated when the actual faulty module is identified. These inconsistencies show that the data had hardly been systematically analyzed or used to a large extent before. After selecting the fields of interest, another tool written in C# read each TR text file, looked for the specified fields and created a SQL insert statement. The process was verified by randomly selecting SQL entries and cross checking them with TRs. Data from 13 007 TRs were inserted in the database for 12 releases of products. From these, 2775 TRs were either duplicated or deleted, leaving 10 232 TRs to analyze. We had access to data on the size of components for three releases of Product 1 developed in Norway[3]. Therefore, we focused on these three releases.

4.5. What was the impact of incremental development on data collection? Answering *CQ2*

We observed that the templates for reporting a TR or CR had changed several times across releases, introducing inconsistencies in data. More standardization is therefore necessary for comparison of results across releases. Data were not systematically collected in intervals such as after a release or 6 months later and proper measures for incremental development were missing such as the size of new and modified code (these data were collected by us) and requirement changes between releases. The data collection routines should be synchronized with the release dates and handle data for several releases while these are still in maintenance. We also observed that TRs and CRs often included only the estimated needed effort for implementation, and not the actual one. The actual effort spent on releases are stored in the effort reporting system per team, but not

[3] We had data from four releases originally, but one of these was just an internal restructuring release and is left out from the analysis.

detailed enough for evaluating effort spent on specific corrections, modifications or components.

5. QUANTITATIVE ANALYSIS AND RESULTS

This section presents quantitative data from Product 1 and the results of the assessment of the hypotheses. We also discuss validity threats and answer *RQ* and *VQ* based on the quantitative evidence.

5.1. Overview of data

Table 3 compares three releases of Product 1, where:

- "#TRs Total" are the number of non-duplicated TRs, "#TRs Subsystems/Blocks" are the number of TRs with information on the faulty subsystem/block, and "#TRs Op6" are the number of TRs reported in the first 6 months of operation,
- "#CRs" shows the number of issued CRs,
- "KLOC" and "MKLOC" are size of code totally and modified,
- "PH test" is reported person-hours used in system test, including rework,
- Code modification rate or "MOD" is (MKLOC/KLOC)*100, and
- "Reuse rate" is the size of reused code (components in the common services and business-specific layers) divided by the total code size.

KLOC, MKLOC and Reuse rate are shown without considering equivalent factors.

Table 3. Summary of data for three releases of Product 1

Release	#TRs Total	#TRs Sub-systems	#TRs Blocks	#TRs Op6	#CRs	KLOC	MKLOC	PH test	MOD (%)	Reuse rate (%)
1	-[1]	9[1]	-	217[2]	10	408	206	11 000[2]	50	61
2	602	414	109	226[2]	37	452	230	30 000[2]	51	59
3	1953	1519	1063	414[3]	118	480	240	55 000[2]	50	61

Notes:
[1]Relaese 1 had very few reported TRs, indicating informal handling of problems in this release.
[2]These data are captured by the quality management team.
[3]This is data after 8 months of delivery.

We do not observe a reduction in the number of faults or fault-density over time as opposed to the study of Ostrand et al. [2004], but their study covered 17 releases of a system. The rate of growth in size is comparable to their case but we do not know the code modification rate in their study. They also found that after size, the next factor in predicting fault-density was whether a module was new or modified. In this case, the

code is modified by 50% between releases, which may introduce new faults. A high number of faults may also be an indication of better testing rather than poor quality [Fenton and Ohlsson 2000]. Effort spent on testing was almost doubled across releases. The increase in testing effort may be due to increased complexity, better testing routines, or that testing was initially underdone. The time a release has been in field usage and the number of users affect the number of reported faults during operation as well. In a study of a large software system, Mockus et al. [2005] write that the amount of usage is important in predicting the probability of observing a failure. Therefore, we included "#TRs Op6" to compare post-release faults after a certain period of time. Release 3 was still in the maintenance phase on the date of this study (almost 8 months after delivery). Although we lack quantitative data on the number of users, we know that Release 3 had several times larger a user base than the previous release. Therefore, the number of faults after 6 months of operation is not considered significantly higher for Release 3 compared to the previous releases.

In the coming sections, we evaluate our hypotheses with subsystems and blocks as components.

5.2 Results on the subsystem level

Table 4 compares reused and non-reused subsystems for two releases. Since #TRs is low for Release 1, it is not included in further analysis. The number of subsystems is too low to perform statistical tests. The fault-density of reused subsystems (#TRs/KLOC and #TRs/MKLOC) is 30% to 67% of the non-reused ones (less difference for modified code); thus supporting that **reused subsystems have significantly lower fault-density than the non-reused ones**. Reused subsystems are smaller in size in average; however size is not correlated with fault-density as discussed in **H5**. They are also **significantly less modified between releases**.

Table 4. Summary of data for subsystems of Release 2 and Release 3

	No. subsystems	#TRs	#TRs/ KLOC	#TRs/ MKLOC	#CRs	Average size (KLOC)	Code Modification rate (%)
Release 2							
Reused	8	170	0.44	1.43		33.41	44
Non-reused	3	244	1.44	2.19		61.60	60
Release 3							

Reused	9	664	2.31	5.17	120	31.92	45
Non-reused	3	855	4.62	7.67	56	61.70	60

We also wanted to assess whether reused or non-reused components are more vulnerable to changes in requirements as stated in **H3**. Release 3 had 118 CRs where only 81 had registered the affected subsystem name, and some CRs affected both reused and non-reused components. The mean number of CRs is 13 for reused subsystems and 18 for the non-reused ones; i.e., lower for the reused subsystems. When dividing #CRs by the size of components (density of requirement modification), the mean would be 0.0012 for reused and 0.0008 for non-reused components; i.e., higher for the reused subsystems. We may reject H3 in both cases, but the direction of results depends on the selected variable (the count of requirement changes or the density of them) and can be subject of future research. There are several reasons for why requirement changes may impact one type of components more than the other: a) if changes in requirements arose from external events such as unpredictable market conditions or customer demands, the application-specific components would be more affected. Our study on the origin of CRs in [Mohagheghi and Conradi 2004b] showed that the proportion of CRs due to external factors (customers and changing environments) is only 34% and therefore not as high as expected, b) other types of CRs such as optimization of non-functional attributes affect all types of components, and c) reused components should meet demands of several products.

5.3 Results on the block level

In this section we present data and statistical tests performed on TRs of Release 3 on the block level. Data for this release are more complete, but we will give results for Release 2 whenever possible.

5.3.1. H1- Reused components have lower fault-density than the non-reused ones. There were 29 reused (mean size of 10.19 KLOC) and 20 non-reused blocks (mean size of 9.13 KLOC) in Release 3. Table 5 compares fault-density of these components. Using mean values, the fault-density of reused components is 44% to 61% of the non-reused ones, less difference for modified code.

Table 5. Descriptive statistics for fault-density of blocks

Fault-density	Mean	Median	StDev
#TRs/KLOC, Reused	1.32	0.76	1.31
#TRs/KLOC, Non-reused	3.02	2.44	2.10
#TRs/MKLOC, Reused	3.50	1.78	4.61

#TRs/MKLOC, Non-reused	5.69	3.73	4.66

Data were tested for normality and the assumption of equal variances. The assumption of normality was violated for reused components and #TRs/KLOC, and the variances were not equal. For the modified code, distributions were not normal, but had almost equal variances. Table 6 shows the results of the statistical tests. The t-test was applied since it is robust to the violation of normality and the non-parametric Mann-Whitney test was also applied since it has no assumption on distribution. Both tests showed low p-values. Therefore, **H1 is not rejected.** Based on the data, we conclude that **reused blocks have significant lower fault-density than the non-reused ones**. We got the same results with the blocks of Release 2. Previous studies confirm our result, but not in a product family of large scale.

Table 6. Summary of the results of statistical tests regarding fault-density

p-values	t-test	Mann-Whitney test
P(T<=t) one-tailed [#TRs/KLOC]	0.002	0.000
P(T<=t) one-tailed [#TRs/MKLOC]	0.055	0.020

The pooled variance is calculated using group sizes and individual variances, and the pooled standard deviation is the square root of the pooled variance and equals 1.67. ES would then be the difference of mean values divided by the pooled standard deviation [Kline 2004], and equals to -1.01, indicating large detected effect using tables in [Dybå et al. 2006]. The negative sign indicates that reused components have performed better.

But are the differences in fault-density significant for the context, i.e., *practical significance*? Fault detection and removal costs roughly 40-50% of development [Glass 2001] and this cost increases as faults are detected later in the life cycle. The quality manager in Ericsson has estimated this cost to be around 150 person-hours per fault in system test, compared to 25 person-hours in integration testing and 3.5 person-hours in inspections. The lower fault-density of reused components has reduced the correction effort by approximately 40 000 person-hours and the total development effort by 20% (considering modified code), which is a significant reduction. Correction effort per fault may increase if reused components are external [Thomas et al. 1997] but this is not a threat here. On the other hand, the cost of creating reusable components at HP was estimated to be 111% of creating a non-reusable version [Lim 1994], and from 110% to several 100% extra in the [Hallsteinsen and Paci 1997] but mostly in the low end of this range. We do not have this data from Ericsson.

We also studied TRs regarding severity classes as shown in Table 7. All TRs included data on severity. For *Severity A* faults, reused blocks had a higher number of faults than expected (the expected value would be 0.31*435=134.85). A Chi-square test was performed and the returned p-value is 0.001; i.e., reused blocks had more faults with *Severity A* (urgent) than expected from the total distribution. Faults of reused components affect two products and they may more often lead to problems such as node unavailability or restarts, giving these faults higher priority to fix. However, as discussed in [Ostrand et al. 2004], severity rating is highly subjective and sometimes inaccurate and therefore may not reflect the importance of the modifications. This field is originally set by the person who reports a problem but may be changed later during analysis and correction. Such changes are not recorded or analyzed either.

Table 7. TRs and severity classes

Severity	Reused blocks	Non-reused blocks	% of all
A (urgent)	160	167	31%
B (major)	226	361	55%
C (minor)	49	98	14%
Sum	**435**	**626**	**100%**

We tested whether the distribution of TRs was different for pre- and post-delivery faults. The result showed that reused blocks had significantly fewer faults after delivery than expected, with p-value equal to 0.002.

5.3.2. H2- Reused and non-reused components have equal code modification rate between releases. Table 8 shows the means, medians and standard deviations for code modification rate (MOD) between Release 2 and 3. We observe that **reused components are less modified than the non-reused ones** and this trend was confirmed across releases.

Table 8. Descriptive statistics for code modification rate (MOD) of blocks

	Mean	Median	StDev
MOD, Reused	0.43	0.43	0.14
MOD, Non-reused	0.57	0.60	0.14

The distribution was not normal for reused blocks, but variances were not significantly different. A two-tailed t-test rejected the hypothesis of no differences in means with a p-value of 0.001; thus **H2 is rejected**. Since the standard deviations were almost equal, each of them can be used as the pooled standard deviation. ES equals to -1

which we consider as a large effect size, and verifies that reused components are significantly less modified.

5.3.3. H3- Reused and non-reused components are equally affected by requirement modification. We cannot evaluate this on the block level since most CRs did not report the affected blocks. The reason is twofold: CRs are not updated with detailed data on the modified components, and tools do not provide traceability between source code and requirements, or changes in them.

5.3.4. H4- There is no relation between the number of faults and component size for a) all components, b) reused components, and c) non-reused components. A scatter plot with KLOC on the x-axis and #TRs on the y-axis for is shown in Figure 4. Similar plots were obtained for MKLOC.

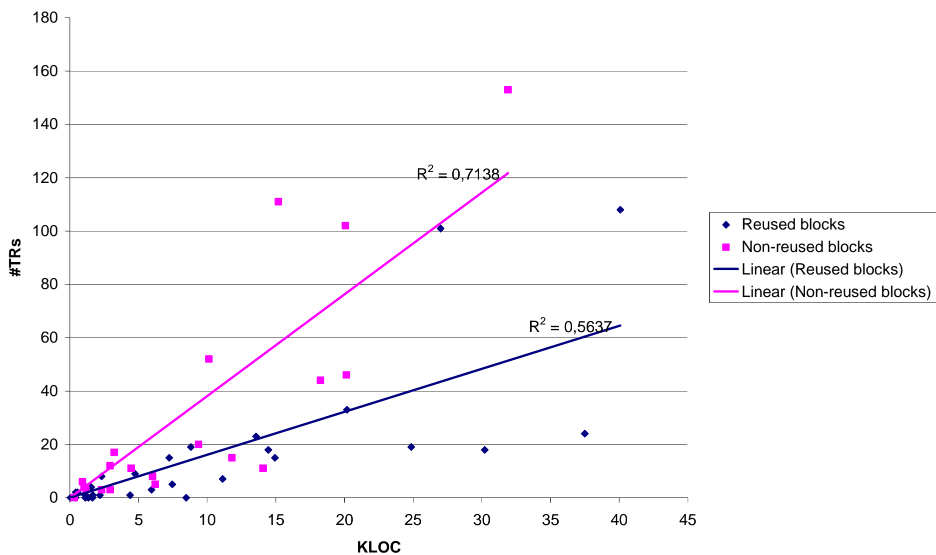

Figure 4. Relation between the number of TRs and the size of blocks of Release 3

For all the blocks, we got R^2 = 0.50, showing weak correlation (not shown). Therefore, **H4-a is not rejected**. When the reused and non-reused components were studied separately, we observed that the gradient of the regression line was higher for non-reused blocks and R^2 values (and adjusted R^2) were higher than 0.70. Therefore, **H4-b is not rejected while H4-c is rejected**. Only for the non-reused components, size is strongly correlated with #TRs. The p-values for the regression lines were lower than 0.01, meaning that the probability for a random correlation is low. To improve conclusion validity, we performed logarithmic transformation of size and observed the same pattern, indicating a relation between the size of non-reused blocks and their number of TRs. The

higher number of faults for larger non-reused components indicates that it is better to break these components down to smaller ones. These components have a greater share of modules programmed in C than the reused ones. Modules programmed in C had more intra-component faults [Schwarz and Killi 2003], which may increase with the module size.

5.3.5. H5- There is no relation between fault-density and component size for a) all components, b) reused components, and c) non-reused components. Figure 5 shows a plot of fault-density versus component size. When we plot with fault-density instead of the number of faults, points are scattered more. Results for regression analysis between fault-density and size in KLOC for blocks gave a p-value of 0.46, while the adjusted R^2 was 0.00, indicating no relation. Again to improve conclusion validity, we performed logarithmic transformation and square root transformation of the data and did not observe any significant relation either. We do not show those plots, since we mean that the original data better show the distances in the data.

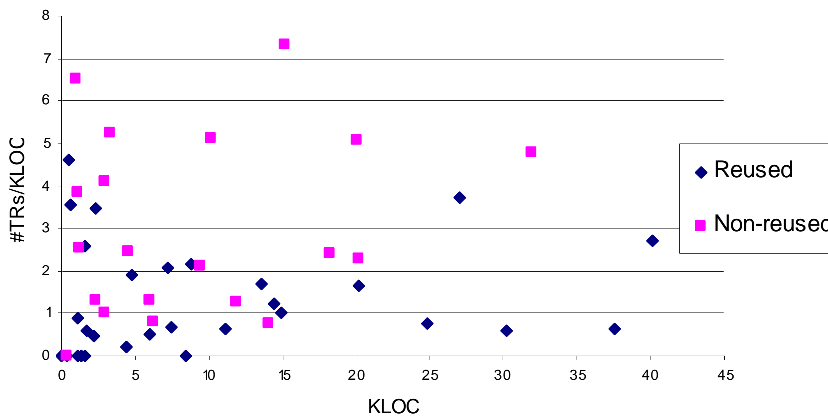

Figure 5. Relation between #TRs/KLOC and component size measured in KLOC for block

When the analysis was performed for reused and non-reused components separately, no significant relation was observed either. Therefore **H5-a, H5-b and H5-c are not rejected**. The adjusted R^2 are lower and the p-values are even higher for modified code, and the hypotheses are not rejected for modified code as well. Results from other studies are inconsistent (see Section 3), which makes the question relevant to study for each context.

5.4 Validity threats

Threats to the validity of the results are:

Construct validity is concerned with whether the selected metrics reflect the intervention and effects; i.e., “right metrics”. We used fault-density, code modification rate and requirement modification rate as software quality indicators in **H1**, **H2** and **H3**. Fault-density is extensively used in previous literature. It is used here to compare the quality of components within the same environment. For code modification rate, earlier studies have confirmed that modifications contribute significantly to the fault potential (e.g., [Graves et al. 2000; Ostrand et al. 2004]). Even when comparing reused components that are slightly modified to those that are extensively modified, significant differences in fault-density were observed in previous studies [Basili et al 1996; Selby 2005]. Therefore, increased code stability may reduce fault-density and maintenance effort. We have not found any studies evaluating code modification rate of reused components. The number of changes to a module is earlier studied in [Selby 2005]. Ideally quantitative results should be combined with qualitative judgments or a prediction model should be built (such as in [Ostrand et al. 2004]) where reuse could also be a variable.

Conclusion validity for statistical analysis is concerned with whether the relationship between the intervention and outcome are of statistical significance; i.e. “right analysis”. We tested the preconditions of statistical tests and performed additional analysis in **H4** and **H5** to improve conclusion validity. In case of **H3**, we could not perform any statistical analysis because of coarse-grained data.

Internal validity is concerned with whether the observed relation is a causal one or “right data” is collected. The following factors impact internal validity:

- Missing, inconsistent or wrong data is a threat to the internal validity of quantitative analysis - but mostly missing data in case of **H1**, **H4** and **H5** when TRs do not include the name of the faulty block. Missing data is caused by the trouble reporting tool that does not force reporters to fill all the fields. This is not considered to be related to the nature of the study or to introduce a systematic bias. The case is about an internal reuse program where all the components are developed in-house and developers know all code equally well, in contrast with external reuse where less knowledge about reused components may impact fault detection. Due to reorganizations, we do not have the possibility to verify that data are missing at random or other factors such as the type or size of

components or the complexity of problems may affect it. One solution would be to search the source code in the configuration management system for a sample of TRs with missing block name. A firm explanation of why data are missing needed such investigation. Other ways to handle missing data are, e.g., mean substitution, regression substitution, or just using the available data. We chose the last strategy and used the whole available dataset for 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 faults over severity classes for all data was almost the same as for subsystems or blocks (which have missing points). The second reason is that the size of our dataset is sufficient for comparing the means in t-tests or studying correlations. Some other statistical tests are more sensitive to missing data. We present a few lessons from this case study in Section 5.7 to improve validity of data.

- For **H1**, one confounding factor may be that Erlang is the dominant programming language for reused components (almost 70%) and C for the non-reused ones (almost 55%). However, Erlang units had an average of 20% more faults per KLOC than C units. Therefore, the impact of programming languages is rejected.
- A threat relevant for **H1** and **H2** is that reused and non-reused components have different functionality and constraints. Since the software is used in network nodes, there are no direct user interfaces. However, the reused components are in a layered architecture in a product family, and this reuse scenario is also discussed in [Hallsteinsen and Paci 1997; Jacobson et al. 1997]. Since we have both horizontally and vertically reused components, the threat is considered to be less. The application domain is also the same for all the components.
- Another threat relevant for **H1** and **H2** would be if more experienced developers or testers worked with one type of the components. This is not considered as a serious threat since the components are developed within the same organization in Norway. Testing is both done by developers and by special test teams.
- Reused components may be designed more thoroughly and be better tested, since faults in these components can affect two products and the prevention costs are amortized over two products. This is not a validity threat for **H1** but a positive consequence of reuse.

- A threat to internal validity of **H2** is that more stable code is often packaged in reusable components. This may be considered as a benefit of reuse to separate stable and change-prone code.
- Size is not correlated with fault-density as discussed in **H5**. Therefore, size is not considered as a threat to the internal validity of **H1**.

External validity of the results should be discussed to evaluate whether the results are generalizable to future releases (Section 5.5), similar products, other contexts ("right context") or to theory. For the quality benefits of reuse, the evidence collected from this study and previous research promotes initiating reuse programs. There are arguments for generalization on the background of cases, e.g., to similar products in the same company. However, what characterizes similar products needs evaluation. On the other hand, if we found evidence of no co-variation between reuse and quality attributes, the results could be an example of a falsification case. Another type of generalization is to theories or models [Lee and Baskerville 2003] as shown in Figure 1. A case study may be valuable in enhancing the evidential force to encourage a technology or approach.

Pfleeger presented some characteristics in evaluating credibility of single studies such as sensitivity to errors, quality of observations, the expertise of those conducting and reporting studies, and their interest in the results [Pfleeger 2005]. This study was performed by researchers and master students not involved in the project at the time of study. One limitation of performing case studies in industry is that industrial data cannot be available to public for further analysis. Finally, as Pfleeger wrote, the evidence must be seen in the context of reader's experience and how it contributes to the overall argument on benefits of large-scale software reuse in a scenario similar to this study.

5.5 Do results of assessments of one or several releases have prediction value for future releases? Answering *VQ*

Based on the results of quantitative analysis and the validity discussion in Section 5.4, we may now answer *VQ*. Some results such as the lower fault-density and code modification rate of reused components were verified across releases. Other data such as the total code modification rate (almost 50%) and reuse rate (almost 60%) were almost the same for releases. However, as discussed by Glass [2002] on whether past maintenance data can be used to predict future, the amount of modification and faults depend on the extent of enhancements. The extent of field usage is also important. As long as future releases are similar to the past ones, e.g., in the effort spent or the extent of enhancements, past data can serve as baseline for future releases.

5.6 Do we have evidence on the benefits of reuse in the context of this study? Answering *RQ*

This section answers *RQ* based on the quantitative evidence. We observed significant benefits in quality in terms of lower fault-density (**H1**) and code modification rate of reused components (**H2**). Reuse benefits are reported in previous studies (see Table 1 and Table 2), but here on a much larger scale. Fault-density during testing cannot be used as a de-facto measure of quality, but remaining faults after testing will impact reliability. Reused components had significantly fewer post-delivery faults than the non-reused ones. More important, reuse provides incentives to prevent and remove defects [Lim 1994] and keep reused code more stable. The lower code modification rate will also reduce fault-density of the reused components.

We have discussed several validity threats and confounding factors. Some of them such as different skills of developers or differences in fault-detection were rejected since all components are developed in-house and by the same organization. We also rejected the impact of size (**H5**) on fault-density. Two factors that affect internal validity are differences in functionality between components (an observation also relevant for other reuse cases) and the amount of missing data in TRs regarding faulty components.

We estimated that the total development effort is reduced by 20% because of savings in correction effort. However, we could not collect data on reuse costs and actual effort spent on corrections to build a cost-benefit model. In spite of extensive literature on reuse cost-benefit models [see Lim 1996 for 17 models], only [Lim 1994] have reported data on return-on-investment.

Since changes in requirements affect cost, schedule and quality, and changes in reused components affect several products, we defined a hypothesis on whether reused and non-reused components are equally affected by change requests. The analysis showed a lower number of CRs for reused subsystems, but not when divided by size. However, we did not have data on the block level. Selby [2005] reported fewer changes to reused modules, but these modules also showed to be smaller in size and less complex. It could be subject of future research to compare the type and extent of changes to reused components versus the non-reused ones.

5.7 Lessons learned regarding data collection

Based on answers to *CQ1* (Section 4.1) and *CQ2* (Section 4.5) and the validity discussion in Section 5.4, we may summarize lessons learned from the quantitative analysis. These are related to gaols, processes for data collection and tools.

Goals: As discussed by Kitchenham et al. [2006], databases should support a hierarchy of goals for corporate, project and software process. Reuse in this case was a corporate decision but RUP and the project metrics were not updated to collect data to evaluate reuse costs or benefits. Incremental development is also a corporate decision but metrics for incremental development on the project level were missing, such as the size of new and modified code between releases and requirement changes between releases.

Processes for data collection: The granularity of data was too coarse with subsystems, and in cases also with blocks. Many TRs and CRs had missing fields that reduced the internal validity of the results since processes (and tools) do not enforce developers to enter sufficient data before closing a TR or CR. We have observed such problems with industrial data in other studies as well (see [Mohagheghi et al. 2006]) and Kitchenham et al. [2006] reported several problems when analyzing large corporate databases developed by IBM Global Services in Australia, to the extent that defect data were not useful for analysis. However, the impact of missing data or ill-defined goals and processes on the studies is not discussed in the papers in Table 1.

Tools: Tools such as ClearDDTS do not provide facilities for search as SQL databases do. Such facilities are important if any detailed analysis should be performed. Sometimes, additional tools may be developed cheaply, like the ones developed in this study to insert data from TRs and CRs in a SQL database. It was difficult to integrate data from several sources such as the textual requirements, the configuration management system and the trouble reporting tool. This is a known problem, also discussed by Kitchenham et al. [2006]. However, the problem is not solved for many commercial tools. Since we do not have enough detailed data, it is not possible to e.g., answer why non-reused components are more modified or how much effort is spent on corrections.

The above discussion shows that data are hardly analyzed by organizations, while they might believe that collecting data allows them to perform analysis when needed. However, data mining is difficult and may lead to invalid conclusions without valid data.

For Ericsson, the results of such empirical studies may be useful to achieve better quality by identifying fault-prone components and taking actions such as inspections or restructuring the components (e.g., splitting or merging), and improving their data collecting routines. All these insights represent explicit knowledge based on own data.

6. CONCLUSION AND FUTURE WORK

This study has “data-mined” trouble reports, change requests, data from the source code configuration management system and those collected by the quality management team, and combined the quantitative results with qualitative observations. We consider this study worthwhile to report due to the contributions, the scale of the study and the industrial context. The goal of the study was to evaluate and explore reuse benefits in the context of the study, which may be summarized as:

- Significant benefits in quality in terms of lower fault-density and code modification rate of reused components were observed, which impact the correction effort. Lower fault-density was “expected” based on previous studies, although rarely studied in large industrial projects, so this study is a contribution in that context.
- Evolving the initial software architecture to a layered one led to a clearer abstraction of the components.
- Reuse reduced risks and lead time of the second product since it was developed based on a tested platform.
- One possible lesson is that for large companies, reuse and standardization of software architecture, processes and skills can help reduce organizational restructuring risks. This is a plausible conclusion based on long-term observation of a case, which is added to the model presented in Figure 1 as one contribution of the study. In the keynote address of ISESE’05 (4th International Symposium on Empirical Software Engineering), Philip Yetton presented a new model in developing large software systems. The idea is to develop a common IT infrastructure and executing business projects based on that as the market unfolds, basically for risk reduction. We think that large-scale reuse should also be valued from this point of view, and further lack of reuse in large companies be considered as a warning.

Frakes and Succi [2001] wrote that reuse benefits are not granted and that organizations should allow own data guide their reuse projects. Internal evaluations can therefore justify future investments on reuse in Ericsson. We also discussed generalization of the results to future releases and similar products. Verifying the benefits of reuse in a company with an extractive approach to reuse (not pre-planned from the beginning), across multiple organizations and countries, and with high risks of large-scale development can promote reuse in other cases. Other contributions of the study are:

- A review of earlier work and their findings.

- Hypotheses from earlier work were assessed in a new context (**H1**, **H4** and **H5**). Reuse benefits are earlier discussed, but few studies are of large-scale or long-term. There are inconsistent results on the relation between component size and the number of faults or fault-density which makes the question relevant to study in a specific context.
- New hypotheses (**H2** and **H3**) were identified for future research.
- The study showed that there is a need to adapt software processes such as RUP for reuse and define metrics that might be periodically and automatically collected to evaluate corporate goals, as well as project and software process goals.

One challenge of studies on reuse is to evaluate economic benefits of reuse. We estimated the reduction in maintenance effort and discussed the practical significance, while we did not have enough data for a complete cost-benefit analysis. For data collection, one main improvement area is identifying tools that provide better traceability and validity. We have already identified one open source tool for trouble reporting that has benefits for data collection. The SEVO project (Software Evolution in Component-based Software Engineering [SEVO 2004]) continues research on reuse in industrial contexts.

8. ACKNOWLEDGEMENTS

The first author was an employee of Ericsson in Grimstad-Norway during this study. Many thanks to the company for this opportunity and to the former colleagues who supported us in collecting the data and discussing the results. Thanks to master students Henrik Schwarz and Ole M. Killi who participated in the quantitative analysis, and to Magne Jørgensen for his comments on the statistical results. We also thank the anonymous reviewers for their comments and hints.

REFERENCES

BALDASSARRE, M.T., BIANCHI, A., CAIVANO, D. AND VISAGGIO, G. 2005. An industrial case study on reuse oriented development. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, 283-292.

BASILI, V.R, 1990. Viewing maintenance as reuse-oriented software development. *IEEE Software* 7(1), 19-25.

BASILI, V.R., BRIAND, L.C. AND MELO, W.L. 1996. How software reuse influences productivity in object-oriented systems. *Communications of the ACM* 39(10), 104-116.

BOSCH, J. 2000. *Design and Use of Software Architecture: Adpoting and Evolving a Product-Line Approach*. Addison-Wesley.

BŐCKLE, G., CLEMENTS, P., MCGREGOR, J.D., MUTHIG, D. AND SCHMID, K. 2004. Calculating ROI for software product lines. *IEEE Software* 21(3), 23-31.

CLEMENTS, P. AND NORTHROP, L.M. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley.

COOPER, D.R. AND SCHINDLER, P.S. 2001. *Business Research Methods*. McGraw-Hill International, 7th edition.
CRNKOVIC, I. AND LARSEN, M. 2002. *Building Reliable Component-Based Software Systems*. Artech House Publishers.
DEVANBU, P., KARSTU, S., MELO, W. AND THOMAS, W. 1996. Analytical and empirical evaluation of software reuse metrics. In *Proceedings of the 18th International Conference on Software Engineering (ICSE'96),* 189-199.
DYBÅ, T., KAMPENES, V. AND SJØBERG, D.I.K. 2006. A systematic review of statistical power in software engineering experiments. *Journal of Information & Software Technology* 48(8), 745-755.
FENTON, N.E. AND OHLSSON, N. 2000. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Software Engineering* 26(8), 797-814.
FLYVBJERG, B. 2004. Five misunderstandings about case-study research. In SEALE, C., GOBO, G.,
GUBRIUM, J.F. AND SILVERMAN, D. 2004. *Qualitative Research Practice*. London and Thousand Oaks, CA, Sage. 420-434.
FRAKES, W.B. AND TERRY, C. 1996. Software reuse: metrics and models. *ACM Computing Surveys* 28(2), 415-435.
FRAKES, W.B. AND SUCCI, G. 2001. An industrial study of reuse, quality and productivity. *The Journal of Systems and Software* 57(2001), 99-106.
GIGERENZER, G. Mindless statistics. *The Journal of Socio-Economics* 33(2004), 587–606.
GLASS, R.L. 2001. Frequently forgotten fundamental facts about software engineering. *IEEE Software* 18(3), 112, 110-111.
GLASS, R.L. 2002. Predicting future maintenance cost, and how we're doing it wrong. *IEEE Software* 19(6), 112, 111.
GRAVES, T.L., KARR, A.F., MARRONM, J.S. AND SIY, H. 2000. Predicting fault incidence using software change history. *IEEE Trans. Software Engineering* 26(7), 653-661.
HALLSTEINSEN, S. AND PACI, M. (Eds.) 1997. *Experiences in Software Evolution and Reuse*. Springer.
JACOBSON, I. GRISS, M. AND JONSSON, P. 1997. *Software Reuse: Architecture, Process and Organization for Business Success*. ACM Press.
JOHNSON, R.E. AND FOOTE, B. 1998. Designing reusable classes. *Journal of Object-Oriented Programming* 1(3), 26-49.
KARLSSON, E.-A. (Ed.) 1995. *Software Reuse, a Holistic Approach*. John Wiley & Sons.
KITCHENHAM, B.A., PICKARD, L. AND PFLEEGER, S.L. 1995. Case studies for method and tool evaluation. *IEEE Software* 12(4), 52-62.
KITCHENHAM, B.A. AND PICKARD L.M. 1998. Evaluating software eng. methods and tools, part 10: designing and running a quantitative case study. *ACM SIGSOFT Software Engineering Notes* 23(3), 20-22.
KITCHENHAM, B.A., KUTAY, C., JEFFERY, R. AND CONNAUGHTON, C. 2006. Lessons learned from the analysis of large-scale corporate databases. In *Proceedings of the 28th International Conference on Software Engineering and Co-located Workshops (ICSE'06)*, Experience Papers, 439-444.
KLINE, R.B. 2004. *Beyond Significance Testing. Reforming Data Analysis Methods in Behavioural Research*. American Psychological Association.
KRUEGER, C. 2002. Eliminating the adoption barrier. *IEEE Software* 19(4), 29-31.
LIM, W. C. 1994. Effect of reuse on quality, productivity and economics. *IEEE Software* 11(5), 23-30. Best paper award.
LIM, W.C. 1996. Reuse economics: a comparison of seventeen models and directions for future research. In *Proceedings of the 4th Intl Conference on Software Reuse (ICSR'96)*, 41-50.
LEE, A.S. AND BASKERVILLE, R.L. 2003. Generalizing generalizability in information systems research. *Information Systems Research* 14(3), 221-243.
MALAIYA, K.Y. AND DENTON, J. 2000. Module size distribution and defect density. In *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)*, 62-71.
MCGREGOR, J.D., NORTHROP, L.M., JARRED, S. AND POHL, K. 2002. Initiating software product lines. *IEEE Software* 19(4), 24-27.
MCILROY, D. 1969. Mass-produced software components. In *Proceedings of Software Engineering Concepts and Techniques*, 1968 NATO Conference on Software Engineering, Buxton, J.M., Naur, P., Randell, B. (eds.), 138-155, available through Petroceli/Charter, New York, 1969.
MILI, A., FOWLER CHMIEL, S., GOTTUMKKALA, R. AND ZHANG, L. 2000. An integrated cost model for software reuse. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, 157-166.
MINTZBERG, H. 1993. *Structure in Fves: Designing Effective Organizations*. Prentice Hall. Cited from
MOCKUS, A., ZHANG, P. AND LI, P.L. 2005. Predictors of customer perceived software quality. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, 225-233.
MOHAGHEGHI, P., CONRADI, R., KILLI, O.M. AND SCHWARZ, H. 2004a. An empirical study of software reuse vs. defect-density and stability. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, 282-292.

MOHAGHEGHI, P. AND CONRADI, R. 2004b. An empirical study of software change: origin, acceptance rate, and functionality vs. quality attributes. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering (ISESE'04)*, 7-16.

MOHAGHEGHI, P. 2004c. *The Impact of Software Reuse and Incremental Development on the Quality of Large Systems*. PhD thesis, NTNU, http://www.idi.ntnu.no/grupper/su/publ/phd/mohagheghi-thesis-10jul04-final.pdf. ISBN 82-471-6408-6.

MOHAGHEGHI, P., CONRADI, R. AND BØRRETZEN, J.A. 2006. Revisiting the problem of using problem reports for quality assessment. In *Proceedings of the 28th International Conference on Software Engineering and Co-located Workshops (ICSE'06), 4th Workshop on Software Quality (WoSQ)*, 45-50.

MOHAGHEGHI, P. AND CONRADI, R. 2007. Quality, productivity and economic benefits of software reuse: a review of industrial studies. To appear in the *Empirical Software Engineering Journal*.

MORISIO, M., ROMANO, D. AND IOANNIS, S. 2002. Quality, productivity, and learning in framework-based development: an exploratory case study. *IEEE Trans. Software Engineering* 28(9), 876-888.

OSTRAND, T.J., AND WEYUKER, E.J. 2002. The Distribution of faults in a large industrial software system. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02)*, TACM SIGSOFT Software Engineering Notes, 27(4): 55 – 64, 2002.

OSTRAND, T.J., WEYUKER, E.J. AND BELL, R.M. 2004. Where the bugs are. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'04)*, ACM SIGSOFT Software Engineering Notes 29(4), 86 – 96.

PFLEEGER, S.L. 2005. Soup or art? The role of evidential force in empirical software engineering. *IEEE Software* 22(1), 66-73.

RAMACHANDRAN, M. AND FLEISCHER, W. 1996. Design for large scale software reuse: an industrial case study. In *Proceedings of the 4th International Conference on Software Reuse (ICSR'96)*, 104-111.

RAMESH, V., GLASS, R.L. AND VESSEY, I. 2004. Research in computer science: an empirical study. *Journal of Systems and Software* 70 (2004), 165-176.

RATIONAL 2005. IBM Rational Software http://www-306.ibm.com/software/rational/

ROSENBERG, J. 1997. Some misconceptions about lines of code. In *Proceedings of the 4th International Symposium on Software Metrics (Metrics'97),* 137-142.

SCHWARZ, H. AND KILLI, O.M. 2003. *An Empirical Study of Quality Attributes of the GSN System at Ericsson*. NTNU master's thesis, 109 pages, NTNU. http://www.idi.ntnu.no/grupper/su/su-diploma-2003/index.html

SEI 2005. http://www.sei.cmu.edu/.

SELBY, W. 2005. Enabling reuse-based software development of large-scale systems. *IEEE Trans. Software Engineering,* 31(6), 495-510.

SEVO 2004. http://www.idi.ntnu.no/grupper/su/sevo/.

SPIQ- Software Process Improvement for better Quality 2000. http://www.geomatikk.no/spiq/.

SUCCI, G., BENEDICENTI, L. AND VERNAZZA, T. 2001. Analysis of the effects of software reuse on customer satisfaction in an RPG environment. *IEEE Trans. Software Engineering* 27(5), 473-479.

THOMAS, W.M., DELIS, A. AND BASILI, V.R. 1997. An analysis of errors in a reuse-oriented development environment. *Journal of Systems and Software* 38(3), 211-224.

YIN, R.K. 2003. *Case Study Research, Design and Methods*. Sage Publications.

ZHANG, W. AND JARZABEK, S. 2005. Reuse without compromising performance: industrial experience from RPG software product line for mobile devices. In *Proceedings of the 9th int'l Software Product Line Conference (SPLC'05)*, 57-69.