

NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET

FAKULTET FOR INFORMASJONSTEKNOLOGI, MATEMATIKK OG
ELEKTROTEKNIKK

INSTITUTT FOR DATATEKNIKK OG INFORMASJONSVITENSKAP



HOVEDOPPGAVE

Kandidatens navn: Ole Morten Killi og Henrik Schwarz

Fag: Hovedoppgave, systemutvikling

Tittel (engelsk): An Empirical Study of Quality Attributes of the GSN System at Ericsson

Oppgavetekst (engelsk):

The main objective of this thesis is to define hypotheses regarding some dependability and maintainability properties of the GSN system at Ericsson. An extensive amount of data is recorded from three projects; this information shall be collected and organized using tools that have to be created. Each of the hypotheses shall then be statistically tested and the results evaluated.

Oppgaven gitt: 15. januar 2003

Besvarelsen levert: 04. juni 2003

Utført ved: Institutt for datateknikk, NTNU og Ericsson AS

Veileder (intern): Reidar Conradi, NTNU

Veileder (ekstern): Parastoo Mohagheghi, Ericsson AS

ABSTRACT

Software reuse and component-based development is the key technology to reduce costs, improve reliability and reduce time-to-market. In this thesis we first present a survey of state-of-the-art of reuse and component-based development. We also review the quality attributes dependability and maintainability, followed by a description of measurement and data analysis methods in software engineering. The main objective of this thesis is to perform an empirical study of quality attributes of the GSN telecommunication system at Ericsson. We have defined some hypotheses regarding reliability and maintainability. In order to verify these hypotheses we have used 13000 trouble reports and 280 change requests from three development projects at Ericsson. The variable quality of the dataset resulted in a large amount of time spent on data normalization and verification. In order to explore the available dataset and support the parsing and normalization, we developed some tools using C# and .NET. We created a SQL database for our research data for convenient data extraction, and this database constitutes the basis for the statistical analysis needed in order to verify the empirical hypotheses. We have found some interesting results regarding reuse and reliability; our analysis concludes that reused components are about twice as reliable as non-reused components. Our results also indicate a certain correlation between fault-proneness and component size. We found that the usage of different programming languages has no special impact on defect density for components. Our data also indicate that reused components are more change-prone; however we do not have a good significance to conclude this. We recommend Ericsson to investigate the results further, and we propose some improvements regarding data collection and development process.

PREFACE

This master thesis is a continuation of the work we performed in the course “SIF8094 Fordypningsprosjekt” during the autumn of 2002. The thesis concludes the fifth year of the MSc study in computer science at NTNU.

This report is a part of the Incremental and Component-based development (INCO) research project. The INCO project is a joint project between the Department of Computer and Information Science (IDI) at NTNU and the Department of Informatics (IFI) at the University in Oslo (UiO), supported by The Norwegian Research Council (NFR).

We wish to thank our project advisors that made it possible for us to achieve our goals in this project. First we want to thank our project supervisor, Professor Reidar Conradi at NTNU, for providing insightful input and valuable feedback throughout the project. Second we wish to thank our external contact and project advisor, Parastoo Mohagheghi at Ericsson AS, for guiding us in our work at Ericsson and providing valuable input and feedback on the report. Professor Tor Stålhane at NTNU provided valuable feedback on statistical analysis. We would also like to thank other employees at Ericsson for providing helpful information, and the INCO project for funding our work.

Trondheim, June 04, 2003

Ole Morten Killi

Henrik Schwarz

TABLE OF CONTENTS

<i>Preface</i>	5
<i>Table of Contents</i>	7
<i>List of Figures</i>	11
<i>List of Tables</i>	13

1 <i>Introduction</i>	15
------------------------------	----

2 <i>Survey of State-of-the-Art</i>	17
--	----

2.1 Component-Based Software Engineering (CBSE)	17
--	----

2.2 CBSE and Reuse	18
---------------------------	----

2.3 Development For Reuse	20
----------------------------------	----

2.4 Development With Reuse	21
-----------------------------------	----

2.4.1 COTS Products	21
---------------------	----

2.4.2 Application Frameworks	21
------------------------------	----

2.4.3 Product-Line Engineering	22
--------------------------------	----

2.5 Software Quality	23
-----------------------------	----

2.6 Quality Attributes	25
-------------------------------	----

2.7 Dependability	26
--------------------------	----

2.7.1 Attributes	26
------------------	----

2.7.2 Threats	27
---------------	----

2.7.3 Means	28
-------------	----

2.8 Maintainability	31
----------------------------	----

2.9 Measurement and Data Analysis Methods	32
--	----

2.9.1 Measurement in Software Engineering	32
---	----

2.9.2 GQM Method	36
------------------	----

2.9.3 Data Analysis Methods	36
-----------------------------	----

2.9.4 Statistical Analysis	37
----------------------------	----

2.9.5 Validity of Results	40
---------------------------	----

2.10 Overview of Empirical Strategies	40
--	----

2.11 Software Size	40
---------------------------	----

3 <i>State-of-the-practice at Ericsson</i>	42
---	----

3.1 Ericsson Context	42
-----------------------------	----

3.2 GSN Architecture	42
-----------------------------	----

3.3	Definition of Components	43
3.4	GSN RUP Development Process	44
3.5	Reuse and Product-Line Engineering in GSN RUP	46
3.6	Current Measurement Practice	47
3.7	Programming Languages	49
3.8	LOC Definitions	50
4	Research Method	51
4.1	Prestudy of the GSN System	52
4.2	Raw Dataset Exploration	52
4.2.1	Trouble Reports - Corrective Maintenance	53
4.2.2	Change Requests - Perfective Maintenance	55
4.3	Data Acquisition and Hypothesis Selection	57
4.3.1	Raw Data	57
4.3.2	Exploring the Available Data	57
4.3.3	Selecting Hypotheses	58
4.3.4	Selecting Data	58
4.3.5	Parsing Selected Data	58
4.3.6	Normalization and Verification	59
4.3.7	Normalized Database	59
4.4	Research Data Extraction	59
4.5	Statistical Analysis, Interpretation and Hypothesis Conclusion	59
5	Research Hypotheses and Results	61
5.1	Overview of Hypotheses	62
5.2	Validity of Results	62
5.3	Reliability	63
5.3.1	RH1 – Reuse and Reliability	64
5.3.2	RH2 – Component Size and Reliability	67
5.3.3	RH3 – Fault Categories	72
5.3.4	RH4 – Programming Languages and Reliability	74
5.3.5	RH5 – Correction Time	78
5.3.6	RH6 – Fault Categories & Development Phases	80
5.3.7	RH7 – Fault Distribution	84
5.3.8	RH8 – Fault Severity	85
5.3.9	RH9 – Test Phase Effectiveness	87
5.4	Maintainability	89
5.4.1	MH1 – Change Cause	89
5.4.2	MH2 – Reuse and Change-Proneness	91
5.4.3	MH3 – Requirement Change	93
5.5	Combination of Reliability and Maintainability	95
5.5.1	CH1 – Relationship Between Reliability and Maintainability	95
5.6	Summary	96

6	<i>Proposed Improvements at Ericsson</i>	99
6.1	Process	99
6.2	Data Collection	99
6.3	Summary	100
7	<i>Conclusion and Further Work</i>	101
	<i>References</i>	103
	<i>Abbreviations</i>	107
8	<i>Appendix (Confidential)</i>	109

LIST OF FIGURES

Figure 2-1: Traditional software reuse.....	19
Figure 2-2: ISO 9126 model for software product quality.....	24
Figure 2-3: Overview of quality attributes.....	25
Figure 2-4: The dependability tree.....	26
Figure 2-5: The fault - error - failure cycle.....	27
Figure 2-6: Failure modes.....	27
Figure 2-7: Fault classes.....	28
Figure 2-8: Illustration of independent and dependent variables.....	37
Figure 2-9: t-test calculations.....	38
Figure 2-10: The correlation coefficient r	38
Figure 3-1: The GSN architecture.....	43
Figure 3-2: Aggregation of the different component types.....	44
Figure 3-3: GSN RUP process map.....	44
Figure 3-4: Product Line Engineering in GSN RUP as viewed by [Schwarz02].....	46
Figure 4-1: Research Method.....	51
Figure 4-2: Overview of entry point for TRs and CRs(ERs).....	52
Figure 4-3: TR process with state transitions.....	54
Figure 4-4: CR in timeline.....	55
Figure 4-5: CR process with state transitions.....	56
Figure 4-6: Research data acquisition and hypothesis definition process.....	57
Figure 5-1: Scatter plot of MEKLOC (x-axis) and #TR (y-axis), subsystem level.....	67
Figure 5-2: Residual plot for MEKLOC and #TR, subsystem level.....	68
Figure 5-3: Scatter plot of EKLOC (x-axis) and #TR (y-axis), subsystem level.....	68
Figure 5-4: Residual plot for EKLOC and #TR, subsystem level.....	68
Figure 5-5: Scatter plot of MEKLOC (x-axis) and #TR (y-axis), block level.....	69
Figure 5-6: Scatter plot of MEKLOC (x-axis) and #TR/MEKLOC (y-axis), block level.....	69
Figure 5-7: Residual plot for MEKLOC and #TR, block level.....	69
Figure 5-8: Scatter plot of EKLOC (x-axis) and #TR (y-axis), block level.....	69
Figure 5-9: Residual plot for EKLOC and #TR, block level.....	69
Figure 5-10: Distribution over fault categories.....	73
Figure 5-11: Average TR effort.....	82
Figure 5-12: Distribution over severity A-C.....	85
Figure 5-13: Distribution over CR groups.....	90
Figure 5-14: Distribution of accepted/implemented and rejected/postponed CRs and ERs.....	93

LIST OF TABLES

Table 2-1: Summary of measurement scales and statistical methods	34
Table 3-1: Summary of Direct Metrics	47
Table 3-2: Derived Metrics defined in the measurement plan	48
Table 3-3: Equivalent factors for calculating ELOC at Ericsson	50
Table 4-1: Number of TRs used	54
Table 5-1: Overview of hypotheses	62
Table 5-2: Average fault density for reused vs. non-reused subsystems	65
Table 5-3: T-test for subsystem defect density	65
Table 5-4: Average fault density for reused vs. non-reused blocks	65
Table 5-5: T-test for block defect density	65
Table 5-6: Regression summary for MELOC and #TR, subsystem level	68
Table 5-7: Regression summary for EKLOC and #TR, subsystem level	68
Table 5-8: Regression summary for MEKLOC and #TR, block level	69
Table 5-9: Regression summary for EKLOC and #TR, block level	70
Table 5-10: Summary of average values for both Erlang and C units	74
Table 5-11: Summary of average values Erlang values divided by the C values	75
Table 5-12: T-test values for the difference between Erlang- and C-units using MKLOC	75
Table 5-13: Summary of average values for both Erlang and C units by using standard ELOC... ..	75
Table 5-14: Average values by using new ELOC (calculated by using EF=2,3)	75
Table 5-15: Different EF for Erlang	76
Table 5-16: Distribution of fault types for Erlang and C	76
Table 5-17: Percentile distribution	78
Table 5-18: Distribution of fault categories	80
Table 5-19: Initial fault categories	81
Table 5-20: Regrouping into implementation and non implementation	81
Table 5-21: Severity of implementation faults for the test- and maintenance phase	83
Table 5-22: Severity for A&D faults for the test- and maintenance phase	83
Table 5-23: Distribution over severity A-C after mapping	85
Table 5-24: Benefits of software inspection	87
Table 5-25: Distribution of reported faults	87
Table 5-26: Distribution of reported faults from test phase	88
Table 5-27: Distribution of reported documentation faults	88
Table 5-28: Average #CR/[KLOC, ELOC, MEKLOC] for reused vs. non-reused subsystems	92
Table 5-29: T-test results	92
Table 5-30: Distribution of mapped statuses	93
Table 5-31: Regression results	95
Table 5-32: Summary of results from the hypothesis testing	97

INTRODUCTION

The main research objective of this master thesis is to define and study some hypotheses regarding some dependability and maintainability properties of the GSN system at Ericsson. The project work is initiated by a survey of the state-of-the-art of component-based software engineering and reuse, followed by a description of measurement and data analysis methods in software engineering (see chapter 0). A survey of the state-of-the-practice at Ericsson is performed in chapter 0.

13 hypotheses have been defined regarding these two quality attributes in cooperation with Parastoo Mohagheghi at Ericsson (presented in chapter 0). We try to verify these hypotheses using data from three projects at Ericsson.

Component-based software development and reuse offers great advantages to development organizations in the form of lowered costs, higher software quality and shorter time-to-market. Component reuse is a fairly new field in the history of software engineering, and there is still much to explore. The goal is to draw some conclusions regarding the mentioned quality attributes in the context of component-based development.

The results may be used to achieve better quality. For example modules with high number of faults should be studied closer and actions should be taken such as inspections, restructuring, etc. When it comes to maintainability, high number of change requests indicates which parts of the system will probably change more often and therefore should be more maintainable.

Intended audience

The intended audience is employees at Ericsson and researchers in software engineering. Some of the hypotheses verify the validity of claims made in other studies and everyone with a general interest in the properties of large scale systems could find this thesis interesting. To understand the statistical basis on which we accept or reject our null hypotheses it is necessary to have basic knowledge of simple statistical analysis.

Context

This report is a part of the Incremental and Component-based development (INCO) research project. The INCO project is a joint project between the Department of Computer and Information Science (IDI) at NTNU and the Department of Informatics (IFI) at the University in Oslo (UiO), supported by The Norwegian Research Council (NFR). The background for INCO is the high cost and failure rate of software projects today. Recently it has been proposed that incremental and component-based development will reduce these problems. These techniques are immature in the form of missing industrial support. The INCO project explores and gathers experience of these challenges. Ericsson is a large supplier of telecommunication systems located in Grimstad. The GSN system is a complex telecommunication system incorporating GPRS nodes for GSM and WCDMA (UMTS).

Report Structure

Chapter 1 - Introduction: this chapter.

Chapter 2 - Survey of State-of-the-Art: presents an introduction to component-based software engineering and reuse, followed by a description of measurement and data analysis methods in software engineering.

Chapter 3 - State-of-the-practice at Ericsson: briefly describes the adapted RUP process at Ericsson as well as the current measurement practices and some programming languages used.

Chapter 4 - Research Method: describes our research method.

Chapter 5 - Research Hypotheses and Results: describes the results, conclusions, general discussions and a validity discussion for each of the hypotheses we have investigated.

Chapter 6 - Proposed Improvements at Ericsson: contains our suggestions for proposed improvements at Ericsson.

Chapter 7 - Conclusion and Further Work: contains our conclusions and ideas for further work.

References: the references used in this thesis.

Abbreviations: a list of the most common abbreviations used in this thesis.

Chapter 8 - Appendix: both confidential information for Ericsson and information that is too detailed to include in the other parts of this thesis. Due to confidentiality reasons the appendix has been removed from the public version of this thesis.

SURVEY OF STATE-OF-THE-ART

In this chapter we first present an overview of Component-Based Software Engineering, followed by a subchapter on CBSE and reuse. In subchapters 0 and 0 we further describe development for and with reuse. Subchapter 0 presents the ISO 9126 definition of software quality, followed by a general overview of quality attributes in subchapter 0. In subchapters 0 and 0 we describe the quality attributes dependability and maintainability in more detail, since this is the foundation for our research work at Ericsson. Subchapter 0 describes measurement and data analysis methods followed by an overview of empirical strategies. Finally we have a description of software size in subchapter 0.

Component-Based Software Engineering (CBSE)

The need for transition from monolithic to open and flexible systems has emerged due to problems in traditional software development, such as high development costs, inadequate support for long-term maintenance and system evolution, and often unsatisfactory quality of software [Atkinson02]. CBSE is an emerging development paradigm that enables this transition by allowing systems to be assembled from a pre-defined set of components explicitly developed for multiple usages. Developing systems out of existing components offers many advantages to developers and users as described in [Atkinson02]. In component-based systems:

- Development costs are significantly decreased because systems are built by simply plugging in existing components.
- Reliability of software is increased since it is assumed that components are previously tested in different contexts and have validated behavior at their interfaces. Hence, validation efforts in these systems have to primarily concentrate on validation of the architectural design.
- Time-to-market is shortened since systems do not have to be developed from scratch.
- Maintenance costs are reduced since components are designed to be carried through different applications and changes in a component are, therefore, beneficial to multiple systems.

As a result, efficiency in the development for the software vendor is improved and flexibility of delivered product is enhanced for the user. Component-based development also raises many challenging problems, such as:

- Building components that can be reused without much effort. This is not an easy task and a significant effort must be used to produce a component that can be used in different software systems.
- Composing a reliable system out of components. A system built out of components is in risk of being unreliable if inadequate components are used for the system assembly.
- The same problem arises when a new component needs to be integrated into an existing system.

- Verification of reusable components. Components are developed to be reused in many different systems, which make the component verification a significant challenge.
- For every component use, the developer of a new component-based system must be able to verify the component, i.e., determine if the particular component meets the needs of the system under construction.
- Dynamic and on-line configuration of components. Components can be upgraded and introduced at run-time; this affects the configuration of the complete system and it is important to keep track of changes introduced in the system.

CBSE and Reuse

Software reuse is one of the main motivations for component-based development. Decomposing large software systems into smaller, semi-autonomous parts opens up the possibility of reusing these parts again within other applications. Reusing a prefabricated component in another application not only saves the associated development costs but also the effort involved in ensuring the quality and integrity of the component. Component reuse therefore significantly enhanced returns on investment in development activities [Atkinson02a].

Component-based development differs from traditional approaches by splitting software development into two distinct activities (from [Atkinson02a]).

- *Development for reuse*: creating the primitive building blocks which hopefully will be of use in multiple applications.
- *Development with reuse*: creating new applications (or possibly larger components) by assembling prefabricated components.

The ultimate goal is to separate these activities to the extent that they may be performed by completely different organizations. Component vendors, *developing for reuse*, will focus on constructing and marketing high-quality, specialized components, which concentrate on doing a specific job well. Component assemblers, *developing with reuse*, will select, purchase and assemble a variety of such Commercial-Off-The-Shelf (COTS) components to create systems for their own customers (possible larger grained components).

While this model is fine in principle, traditional component-based development methods give very little consideration to the factors and properties that actually go into creating truly reusable artifacts. Moreover, when reuse factors are taken into account, it is typically in an ad hoc way [Atkinson02a].

Figure 0-1 (from [Atkinson02]) illustrates the traditional “reuse” life-cycle based on software artifacts (i.e. components) initially developed for use with one application in mind. Basically, the artifacts created in one application-engineering project are stored in an asset base so that they can be reused in other applications. Future application projects then attempt to develop parts of the application by reusing existing assets in the asset base (/reuse library). This kind of reuse is usually white-box reuse where source code is available.

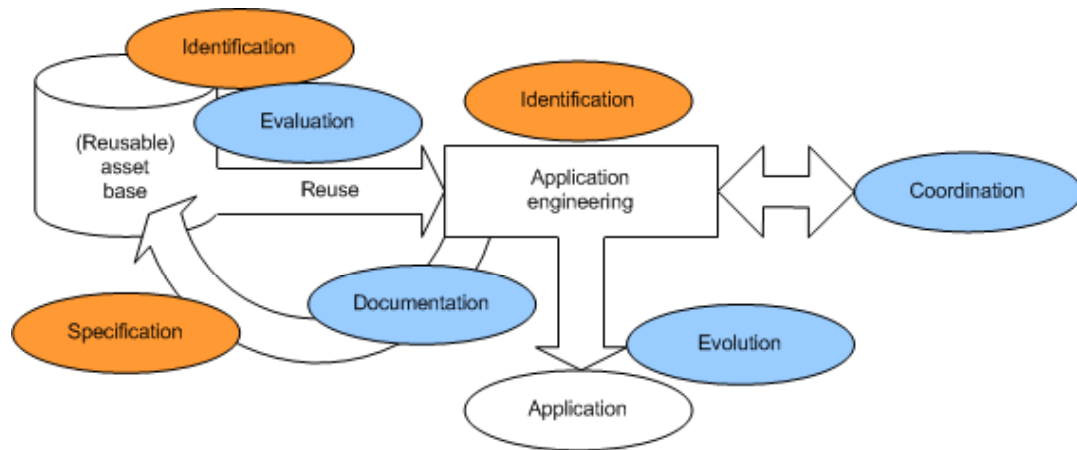


Figure 0-1: Traditional software reuse

The circles marked in blue and orange (in figure 0-1) are typical problem areas that have to be addressed for a successful reuse approach. Identification covers both deciding what components that should be developed from the requirements and the identification of reusable components. Specification covers how components are specified. In addition to these two aspects, the economic side of attempting to create reusable components for the future (making more generalized components usually takes more time) should be considered. The engineers should inform the customer what kind of impact the requirements (and the costs that follows it) have on his/her choices for the software. Involving the customer in this process might cause requirements to change, allowing a reusable component to be used. This is important to do as early as possible as the requirements are often more easily changeable early on in the project. Typically, a reusable component “earns” its value after 2-4 years, by being reused at least twice. In addition, the net economic savings are typically 20-80% return on initial investments according to [Sjøberg00]. Even with this perspective, developing for reuse might not be profitable because the reusable components created might not be reused at all. Ralph Johnson has emphasized that reusable components are “not (pre)planned, they are discovered (gradually later)” [Gamma95] and Jeffrey Poulin presents three phases of the corporate reuse libraries [Poulin95]:

1. very few parts,
2. many parts of low or poor quality,
3. many parts of little or no use.

Johnson’s and Poulin’s remarks should be kept in mind to prevent too many components to be developed for reuse that are not reused later on. It is also important not to overestimate the usefulness of an asset base and that an asset base requires extra documentation for each asset to offer effective searches in the base.

Thinking reuse when developing is the key for taking full advantage of CBSE possibilities. CBSE covers both reuse of traditional libraries and object-oriented frameworks, as well as program/product families and COTS (Commercial-Of-The-Shelf). COTS-based reuse has proved hard [Garland95] [Carney97] [Boehm99] [Morisio00] because there is no standardized development process for doing this. In addition we have no control over COTS risk factors, such as incompatible new versions, vendor delays and inadequate documentation. Problems with reuse is further elaborated in subchapter 0.

Development For Reuse

Development *for* reuse is developing components of a software system that can be reused in another context. When developing *for* reuse it is important to analyze and understand what requirements possible re-users may have as well as what cost benefits later reuses of the artifact may have. Proper research/analysis will have to be done in order to determine how general the component should be. The REBOOT Methodology Handbook [Karlsson95] distinguishes between three types of development *for* reuse:

1. *A priori* development for reuse.
 - Set up a specific group for designing and manufacturing reusable components for later use.
 - Reuse will have a clear separate role.
 - The reuse will not pay off until the component is used
2. *A post-priori* development for reuse.
 - Extract reusable components from already developed products
 - Limited to design decisions made during development
3. An *integrated* development for reuse.
 - Reusable components developed in parallel with development of a specific application.
 - A reuse project may include all the types above.

[Karlsson95] proposes following steps for developing *for* reuse:

- Capture and collect requirements for an initial solution.
- Define initial solution or identify previous solutions that satisfy the same requirements.
- Based on the above, identify possible generalizations.
- Identify possible reusers, and investigate their requirements.
- Identify additional cost and benefit related to the added functionality of the reusable component.
- Analyze added requirements related to variations and invariations (make req. hierarchy).
- Propose a generalized solution with specializations and cost estimates.
- Present the solution for the reusers and to other reuse experts for validation and approval.
- Develop and document the solution.

[Karlsson95] identifies the following problems:

- Lack of components suited for reuse:
- Failure to select and strengthen components for reuse
- Components written to rigid, giving few or no possibilities for reuse
- Immature adaptation of layered architecture
- Missing standardization of components written for reuse.
- Lacking documentation of software components (missing information of what requirements they fulfill, what functions they perform, how they have been tested, how they can be configured, modified and integrated).
- Lack of tools to perform reuse procedures.
- The fact that the development phase usually does not include a point where the developers search for potential material suited for reuse
- No existing role of reuse component engineer etc.
- Too much focus on one project at a time. No systematic practice of reuse within organizations.
- Conflict between the people involved in a single project and those involved in a broader domain (distrust of others).
- Lack of knowledge of how to organize for reuse.

- Developers lacking trust of reuse schemes (may have better trust in tools such as languages etc.).
- A developer wishing to build perfect systems and this is not possible reusing code.

The following list from [Karlsson95] contains things that may be of interest to measure in order to decide the feasibility of reuse:

- Improvements due to reuse (savings)
- The effort used in reuse (costs)
- Improvements in quality
- Improvements in productivity
- Improvements in lead time

Development With Reuse

Developing with reuse is constructing new applications (or possibly larger components) by assembling prefabricated components. After searching for appropriate components and assessing candidate components, some reengineering may be necessary for the component to fit into the system. Reusable components can be internally developed components, COTS products, standard libraries, application frameworks and product-lines. Refer to [Karlsson95] for more information about development *with* reuse.

COTS Products

The term COTS (Component-Off-The-Shelf) products can apply to any component that is offered by a third-party vendor [Sommerville01, p. 315]. Until recently there were only a few large systems, such as database systems, that were routinely reused. Assembling large systems using COTS components offer possibilities to reduce costs and delivery times by orders of magnitude compared to the development of new software [Sommerville01].

However, [Boehm99] discusses four problems with COTS integration:

- *Lack of control over functionality and performance.* Although the published interface of a product may appear to offer the required facilities, these may not be properly implemented or may perform poorly.
- *Problems with COTS system interoperability.* It is sometimes difficult to get COTS products to work together because each product embeds different assumptions about how it will be used.
- *No control over system evolution.* Vendors of COTS products make their own decisions on system changes in respond to market pressures.
- *Support from COTS vendors.* The level of support from COTS vendors varies widely. Because the source code is often not available the developers must rely on support from the vendor.

Application Frameworks

Frameworks (or application frameworks) are a subsystem design made up of a collection of abstract and concrete classes and the interfaces between them [Sommerville01, p.314]. Particular details of the application subsystem are implemented by adding components and by providing concrete implementations of abstract classes in the framework. Applications are normally constructed by integrating a number of frameworks.

Three classes of frameworks are identified [Sommerville01, p.314]:

- *System infrastructure frameworks*: Supports development of system infrastructures such as communications, user interfaces and compilers [Schmidt97].
- *Middleware integration frameworks*: Consists of a set of standards and associated object classes that support component communication and information exchange. Examples are CORBA, COM, DCOM and JavaBeans [Orfali98].
- *Enterprise application frameworks*: Concerned with specific application domains such as telecommunications or financial systems [Baumer97]. These embed application domain knowledge and support the development of end-user applications. These frameworks are related to product-lines described next.

Especially frameworks concerning graphical user interfaces (GUI) are widely used.

Frameworks are often an instantiation of a number of design patterns.

The fundamental problem with frameworks is their inherent complexity and the time it takes to learn to use them [Sommerville01, p.315]. As a consequence, some software engineers will be framework specialists, especially in large organizations. Frameworks are an effective approach to reuse, but the cost of introduction limits its widespread use.

Product-Line Engineering

According to [Sommerville01, p.318], one of the most effective approaches to reuse is based in the notion of application families. An application family or product line is a related set of applications that has a common domain-specific architecture. Each specific application is specialized in some way. The common core of the application is reused each time that a new application is required. The new development may involve writing some additional components and adapting some of the components in the application to meet new demands.

With some exceptions, product lines usually emerge from existing applications [Sommerville01, p.321]. An organization develops an application and then, when a new application is required, uses this as a basis for the new application. Further demands for new applications cause the process to continue. However, as change tends to corrupt application structure, at some stage a specific decision to design a generic product line is made. This design is made on reusing the knowledge that has been gained from developing the initial set of applications.

For many years, practical product line engineering was frustrated by the fact that traditional software implementation technologies, such as programming languages, did not really provide the mechanisms needed to support the rapid and cost-effective adaptation of implemented code [Atkinson02]. As a result, contemporary product line methods have been forced to concentrate on the early life-cycle activities, and thus are often seen as rather “high-level”. The advent of component-based development promises to change this situation by providing mechanisms that enable software elements to be rapidly and efficiently assembled into new applications. Hence components provide the perfect foundation for the practical application of product line development [Atkinson02].

Software Quality

Software quality is the degree to which software possesses a desired combination of attributes (e.g., reliability, interoperability) [IEEE-1061]. Software quality is one of three user-oriented product characteristics: quality, cost, and schedule. Cost and schedule can be predicted and controlled to some extent by mature organizational processes. However, process maturity does not translate automatically into product quality [Barbacci97]. Software quality requires mature technology to predict and control quality attributes. If the technology is lacking, even a mature organization will have difficulty producing products with predictable performance, dependability, or other attributes. Quality, cost, and schedule are not independent. Poor quality eventually affects cost and schedule because software requires tuning, recoding, or even redesign to meet original requirements. Cost and schedule overruns are common because serious problems are often not discovered until the system integration phase [Barbacci97].

The ISO 9126 standard is made up of six main attributes as described in [Wang02, p.3]:

- *Functionality*: The capability of the software to provide functions which meet stated and implied needs when the software is used under specified conditions.
- *Reliability*: The capability of the software to maintain the level of performance of the system when used under specified conditions.
- *Usability*: The capability of the software to be understood, learned, used and liked by the user, when used under specified conditions.
- *Efficiency*: The capability of the software to provide the required performance relative to the amount of resources used, under stated conditions.
- *Maintainability*: The capability of the software to be modified.
- *Portability*: The capability of software to be transferred from one environment to another.

The ISO 9126 model for software product quality is shown in figure 0-2.

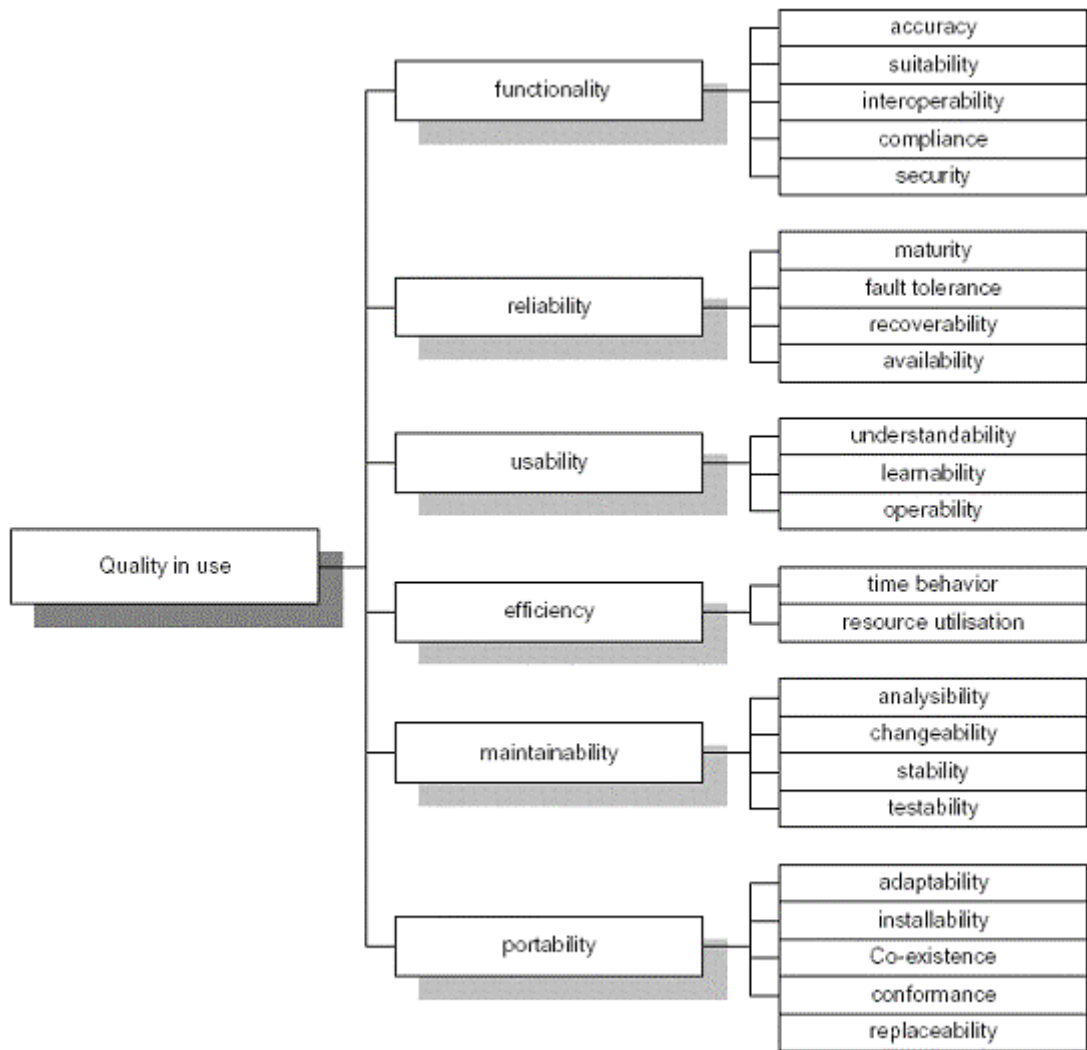


Figure 0-2: ISO 9126 model for software product quality

Quality Attributes

This subchapter is based on [Mohagheghi03, p.55]. While users of a system are concerned with functionality and performance, other stakeholders are concerned with other properties: The developing organization is concerned with schedule and cost, investors want the possibility to add features to the system that provide them competitive advantage, and operators want a system that is easy to maintain. While it is consensus on using the term *functional requirements* for requirements concerning business goals, other type of requirements are covered by different terms over time and classified differently in literature. They are sometimes called *non-functional requirements* (example is RUP), in some literature for *quality requirements* leading to *quality attribute* of a system [Bosch00], [Sommerville01] uses the term *emergent properties*, and [Bachman00, p.12] calls them for *extra-functional properties* or *quality attributes* or when associated with a service, *quality of service*.

It is difficult to give definitions of quality attributes that everyone agrees on. Attributes such as quality-of-service and dependability are broadly used while scope of them depends on the domain and the type of information being transformed. Besides dependability is a collective term and the attributes of dependability vary: It is consensus on reliability and availability as attributes of dependability, while maintainability, security and safety are in some literature included and in some other left out. Figure 0-3 from [Mohagheghi03] shows an overview of these quality attributes.

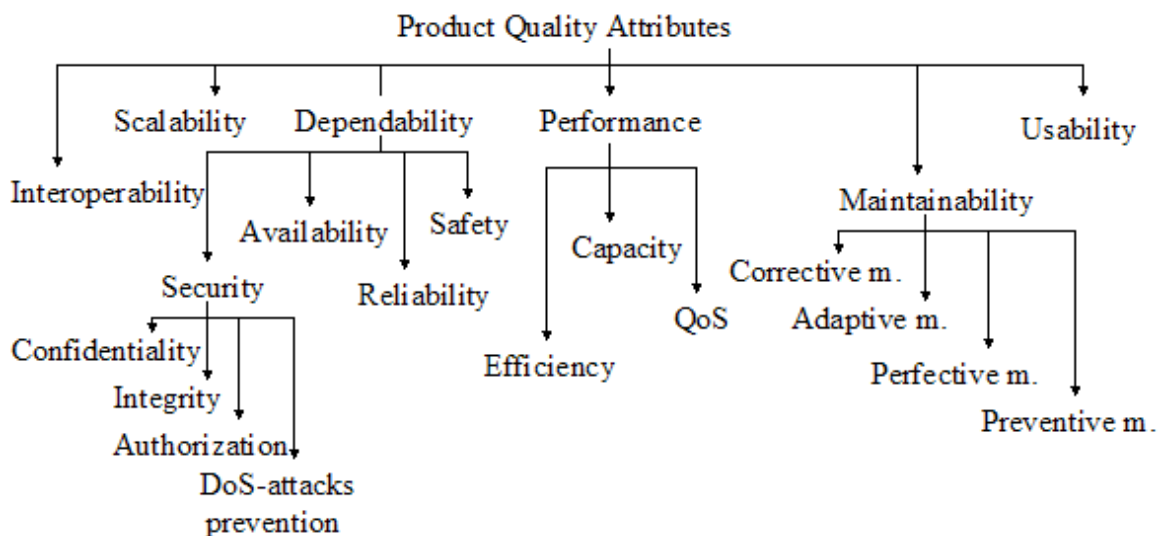


Figure 0-3: Overview of quality attributes

Dependability

[Sommerville01, p.354] defines four principal dimensions to dependability; availability, reliability, safety and security, and informally describe them. In short the system must be up and running, being able to deliver useful services and be able to correctly deliver those services. Safety and security are probabilities that are based on judgements that are made on the basis of evidence about the system. Safety is the judgement of how likely the system will cause damage to people or its environment. Security is similarly the judgement of how likely the system can resist accidental or deliberate intrusion.

[Laprie01, p.1] defines *dependability* as “the ability to deliver service that can justifiably be trusted”. The *service* delivered by a system is its behavior as it is perceived by its users; a *user* is another system (physical or human) that interacts with the service at the service interface. The *function* of a system is what the system is intended for, and is described by the system specification.

[Laprie01] divides the concept of dependability into three parts: the *threats* to, the *attributes* of, and the *means* by which dependability is attained, as shown in figure 0-4.

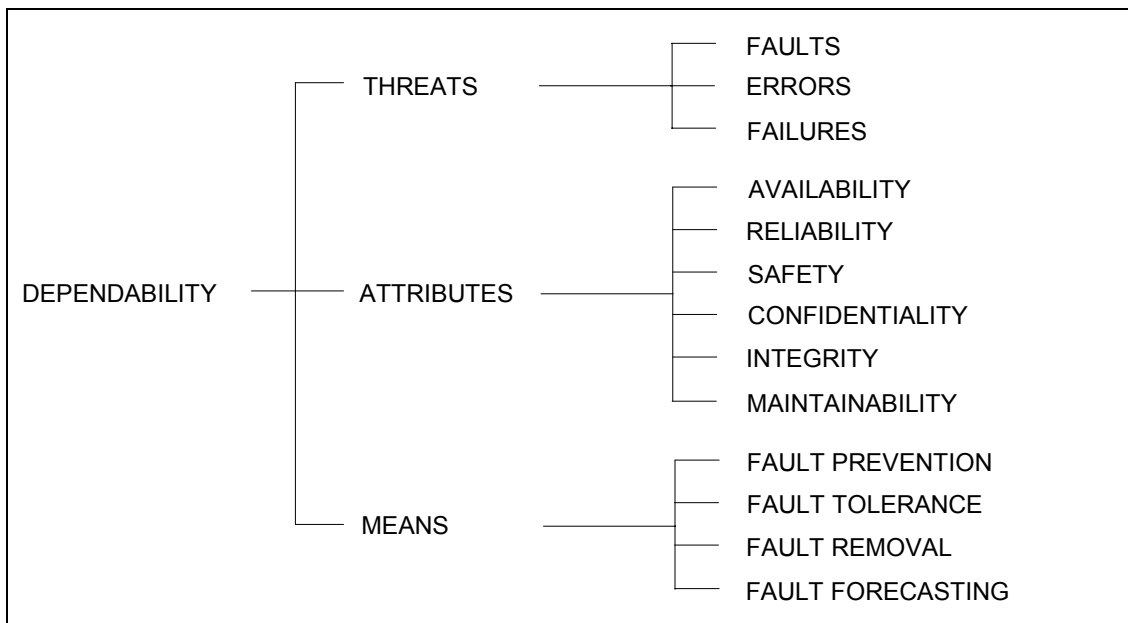


Figure 0-4: The dependability tree

Attributes

[Laprie01, p.2] lists the following attributes for dependability (as shown in figure 0-4):

- *Availability*: Readiness for correct service.
- *Reliability*: Continuity of correct service.
- *Maintainability*: Ability to undergo repairs and modifications.
- *Safety*: Absence of catastrophic consequences on the users and environment.
- *Confidentiality*: Absence of unauthorized disclosure of information.
- *Integrity*: Absence of improper system state alternations.

Security is the concurrent existence of a) availability for authorized users only, b) confidentiality and c) integrity.

In their definitions, availability and reliability emphasize the avoidance of failures, while safety and security emphasize the avoidance of a specific class of failures (catastrophic failures, unauthorized access or handling of information).

Threats

According to [Laprie01, p.1] *correct service* is delivered when the service implements the system function. A system *failure* is an event that occurs when the delivered service deviates from correct service. A system may fail either because it does not comply with the specification, or because the specification did not adequately describe its function. A failure is thus a transition from correct service to *incorrect service*, i.e., to not implementing the system function. A transition from incorrect service to correct service is *service restoration*. The time interval during which incorrect service is delivered is a *service outage*. An *error* is that part of a system state that may cause a subsequent failure: a failure occurs when an error reaches the service interface and alters the service. A *fault* is the adjudged or hypothesized cause of an error. A fault is *active* when it produces an error, otherwise it is *dormant/passive*. Figure 0-5 shows the fault-error-failure cycle.

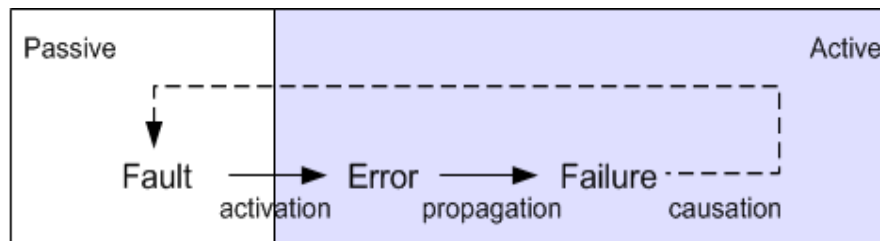


Figure 0-5: The fault - error - failure cycle

[Laprie01, p.2] defines *failure modes* as the ways a system can fail. As shown in figure 0-6, failures are divided into three viewpoints: the failure domain, the perception of a failure by system users and the consequences of failures on the environment. An error is *detected* if its presence in the system is indicated by an *error message* or *error signal* that originates within the system. Errors that are present but not detected are *latent* errors.

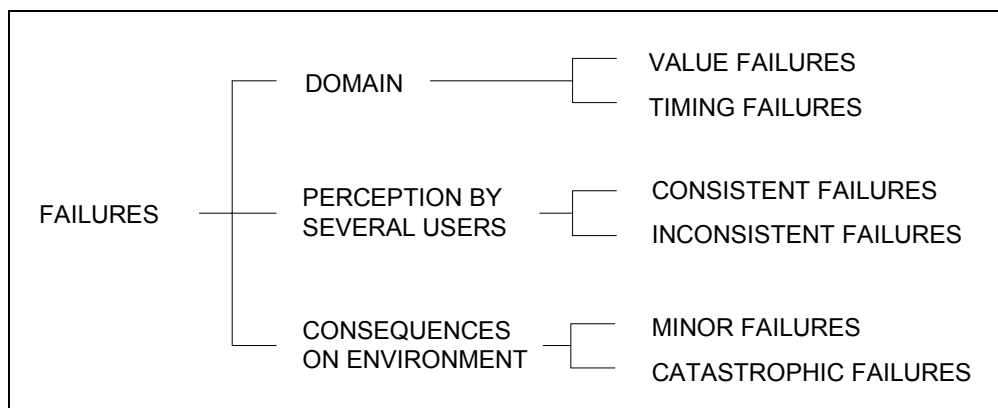


Figure 0-6: Failure modes

[Laprie01, p.3] gives a classification of faults based on six criteria as shown in figure 0-7.

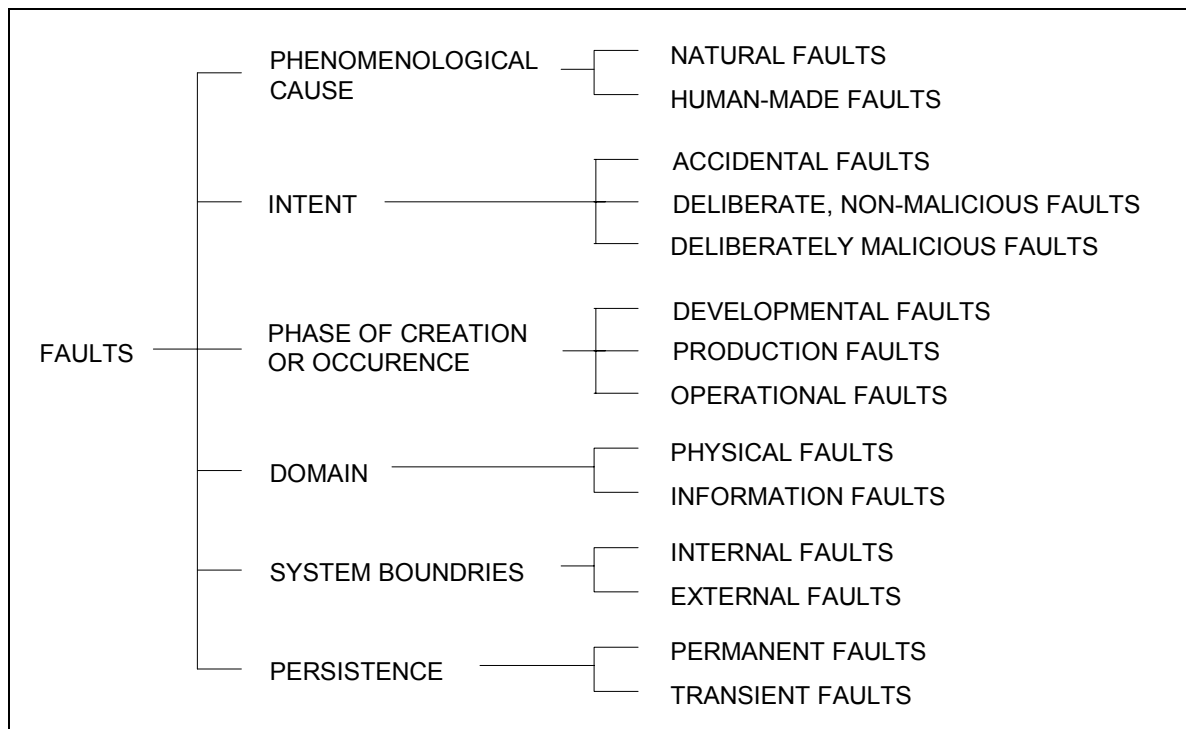


Figure 0-7: Fault classes

Means

Means to develop and maintain a dependable system are described in [Laprie01, p.3-5] (as shown in figure 0-4):

- *Fault prevention*: How to prevent the occurrence or introduction of faults.
- *Fault tolerance*: How to deliver correct service in the presence of faults.
- *Fault removal*: How to reduce the number or severity of faults.
- *Fault forecasting*: How to estimate the present number, the future incidence, and the likely consequences of faults.

Fault Prevention

Fault prevention is attained by quality control techniques employed during the design and implementation phase. They include design rules and best practices for programming (structured programming, information hiding etc.), modularization, reviews and inspection techniques for early fault detection and testing before delivery. Operator interaction faults are prevented by training, procedures, documentation etc. Malicious faults are prevented by firewalls and similar defenses.

Fault Tolerance

Fault tolerance is mechanisms to isolate faults and preventing system failures in the presence of active faults. It is generally implemented by error detection and subsequent system recovery.

Error detection originates an error signal or error message within the system. There exist two classes of error detection techniques: a) *concurrent error detection*, which takes place

during service delivery; and b) *preemptive error detection*, which takes place while service delivery is suspended; it checks the system for latent and dormant faults.

Recovery transforms a system state that contains one or more errors and (possibly) faults into a state without detected errors and faults. Error handling eliminates errors from the system state and may take two forms: a) *rollback*, where the state transformation consists of returning the system back to saved state before that existed prior to error detection, and b) *rollforward*, where the state without detected errors is a new state.

Fault handling prevents located faults from being activated again and consists of four steps: a) *fault diagnosis* that identifies and records the cause of the error, b) *fault isolation* that performs physical or logical exclusion of the faulty components from further participation in service delivery, c) *system reconfiguration* that either switches in spare components or reassigns tasks among non-failed components, and d) *system reinitialization* that checks, updates and records the new configuration and updates system tables and records.

Examples of fault tolerance: starting separate threads for each user (if one thread should crash it would not affect the other users), persistent data storage and hardware redundancy combined with reconfiguration to use spare hardware (i.e. RAID system).

Fault Removal

Fault removal is performed during the development phase by using *verification* techniques. Verifying a system without actual execution is *static verification*. Verifying a system by using it is *dynamic verification*, normally called *testing*.

Fault removal during the operational life of a system is corrective or preventive maintenance. *Corrective maintenance* is aimed to remove faults that have produced one or more errors that have been reported, while *preventive maintenance* is aimed to uncover and remove faults before they might cause errors during normal operation. The latter faults include a) physical faults that have occurred since the last preventive maintenance actions, and b) design faults that have led to errors in other similar systems. Corrective maintenance for design faults is usually performed in stages: the fault may be first isolated (e.g., by a workaround or a patch) before the actual removal is completed. These forms of maintenance apply to non-fault-tolerant systems as well as fault-tolerant systems, as the latter can be maintainable on-line (without interrupting service delivery) or off-line (during service outage).

Fault Forecasting

Fault forecasting is conducted by performing an evaluation of the system behavior with respect to fault occurrence or activation. Evaluation has two aspects:

- *Qualitative* or *ordinal evaluation* aims to identify, classify and rank the failure modes, or the event combinations (component failures or environmental conditions) that would lead to system failures.
- *Quantitative* or *probabilistic evaluation* aims to evaluate in terms of probabilities the extent to which some of the attributes of dependability are satisfied; those attributes are then viewed as measures of dependability.

The methods for qualitative and quantitative evaluation are either specific (e.g., failure mode and effect analysis for qualitative evaluation, or Markov chains and stochastic Petri nets for quantitative evaluation), or they can be used to perform both forms of evaluation (e.g., reliability block diagrams, fault-trees).

The evolution of dependability over a system's life-cycle is characterized by the notions of stability, growth and decrease, that can be stated for the various attributes of dependability. These notions are illustrated by *failure intensity*, i.e., the number of failures per unit of time. It is a measure of the frequency of system failures, as noticed by its

users. Failure intensity typically first decreases (reliability growth), then stabilizes (stable reliability) after a certain period of operation, then increases (reliability decrease), and the cycle resumes.

The alternation of correct-incorrect service delivery is quantified to define reliability, availability and maintainability as measures of dependability:

- *Reliability*: a measure of the continuous delivery of correct service - or, equivalently, of the time to failure.
- *Availability*: a measure of the delivery of correct service with respect to the alternation of correct and incorrect service.
- *Maintainability*: a measure of the time to service restoration since the last failure occurrence, or equivalently, measure of the continuous delivery of incorrect service.
- *Safety* is an extension of reliability: when the state of correct service and the states of incorrect service due to non-catastrophic failure are grouped into a safe state (in the sense of being free from catastrophic damage, not from danger), safety is a measure of continuous safeness, or equivalently, of the time to catastrophic failure; safety is thus reliability with respect to catastrophic failures.

Maintainability

[IEEE93] defines maintainability as: “*The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changes environment*”.

Maintenance is defined as: “*The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment*” [IEEE93].

Usually, the maintenance process can be divided into four areas of focus [IEEE93]:

1. *Corrective maintenance*: Maintenance performed to correct faults in hardware or software.
2. *Adaptive maintenance*: Software maintenance performed to make a computer program usable in a changed environment. Adaptive maintenance does not lead to changes in the system’s functionality.
3. *Perfective maintenance*: Software maintenance performed to improve the performance, maintainability or other attributes of a computer program. Mainly deals with accommodating new or changed user requirements. It concerns functional enhancements to the system.
4. *Preventive maintenance*: Concerns activities aimed at increasing the system’s maintainability, such as updating documentation or adding comments.

With the maturation of software development practices, software maintainability has become one of the most important concerns of the software industry [Coleman94]. Parikh claims that the total cost of maintaining a software product is typically 45 to 60 percent of the cost of developing it [Parikh83]. Two recognized experts, Corbi and Yourdon, claimed that software maintainability is one of the major challenges for the 1990s [Coleman94]. These statements were validated by Dean Morton, executive vice president and chief operating officer of Hewlett-Packard. Morton stated that Hewlett-Packard (HP) currently has between 40 and 50 million lines of code under maintenance and that 60 to 80 percent of research and development personnel are involved in maintenance activities. He went on to say that 40 to 60 percent of the cost of production is now maintenance expense.

Mockus and Votta studied historical data from a large telecommunication system [Votta00]. They analyzed a large version control database by designing a tool that automatically classified changes according to maintenance activities based on textual description fields. To verify the validity of the classification they used developer surveys and also applied the classification to another product. They identified four primary reasons for change; adding new features (adaptive), fixing faults (corrective), restructuring code to accommodate future changes (perfective), and code inspection rework changes that represent a mixture of corrective and perfective changes. Each has a distinct size and interval profile. The interval for the adaptive changes is the longest, followed by inspection changes, with corrective changes being the smallest. They discovered a strong relationship between the difficulty of a change and its type: corrective changes tend to be the most difficult, while adaptive changes are difficult only if they are large. Inspection changes are perceived as the easiest. They also discovered a large amount of perfective changes in the examined system.

Measurement and Data Analysis Methods

This subchapter describes the state-of-the art of measurement in software engineering. Subchapter 0 gives an introduction to the GQM method followed by an overview of different data analysis methods. In subchapter 0 we present a brief description of some statistical methods used in this thesis. Finally in subchapter 0 we discuss validity threats.

Measurement in Software Engineering

This subchapter is based on [Mohagheghi03, p.33]. Metrics are defined in order to measure software quality attributes indirectly; i.e. we assume that there is a relationship between what we measure and some quality attribute. Measurement is central in any empirical study. Measurement and measure are defined in [Wohlin00, p.25] as: “*Measurement* is a mapping from the empirical world to the formal, relational world”. Consequently, a *measure* is the number or symbol assigned to an entity by this mapping in order to characterize an attribute [Fenton97]. The word *metrics* is both used to denote the field of measurement, the measured attribute of an entity and related data collection procedures.

Some attributes are directly measurable (e.g. size of a program in LOC), while others are derived from other measurements (e.g. productivity in LOC/effort). [Wohlin00, p.29] calls these two groups for *direct* and *indirect* measures. Measures can also be divided into objective and subjective measures: An *objective measure* is a measure where there is no judgment in the measurement value. Example is LOC. A *subjective measure* depends on both the object and the viewpoint. Example is personal skill.

Sommerville defines software metrics as “any type of measurement that relates to a software system, process or related documentation” [Sommerville01]. He calls metrics associated to software process for *Control metrics* and metrics associated to product for *Predictor metrics*.

In software engineering attributes we wish to measure are usually divided into three classes [Fenton97, p.74]:

- *Processes*: Software-related activities.
- *Products*: Artifacts that result from process activities.
- *Resources*: Entities needed by process activities.

Within each class, we distinguish between internal and external attributes [Fenton 97]:

- *Internal attributes*: Those attributes that can be measured in terms of the process, product or resource itself, separate from its behavior. Examples are effort used in design, size of code or other structural properties of the product, or price of software.
- *External attributes*: Those attributes that can be measured with respect to the environment. Here the behavior of the process, product or resource is important, not the entity itself. Examples are quality attributes of a product such as maintainability, stability of a process or experience of the personnel.

When we measure something, we either want to assess something or to predict some attribute that does not yet exist. The second goal is achieved by making a *prediction system*; i.e. a mathematical model for determining unknown parameters from known ones. The internal attributes are usually measured to predict external ones. For example complexity of modules can be used in prediction of their maintainability.

We also may try to establish a relationship between products and the processes used to develop them or resources. For example we may study whether some methods of development are better than other for reliability or maintainability.

Measurement scales are usually classified as one of five major types [Wohlin00, p.27], [Fenton97, p.47]:

- *Nominal scale*: The nominal scale maps the attribute of the entity into a name or symbol and is the least powerful of the scale types. Examples are classification of defect types or nationality of people. The measure of central tendency for nominal scales is *mode*; i.e. which category has the most members.
- *Ordinal scale*: The ordinal scale ranks the entities after an ordering criterion, and is therefore more powerful than the nominal scale. With ordering, one can answer to questions such as “better than” or “greater than”. Example is complexity of a task being easy, medium or hard. The appropriate measure of central tendency is the *median* and a percentile or quartile measure reveals the dispersion.
- *Interval scale*: The interval scale is used when the difference between two measures are meaningful, and not the value itself; i.e. the “relative distance” can be measured and hence the scale is more powerful than the ordinal scale type. Measures on this type are uncommon in software engineering. Example is temperature measured in Celsius degrees. The measure of central tendency is the arithmetic *mean*.
- *Ratio scale*: If there exists a meaningful zero value and the ratio between two measures is meaningful, a ratio scale can be used. Examples are length of a development phase or size of software in LOC. All measures of central tendency mentioned up are usable with ratio data.
- *Absolute scale*: The measurement of an absolute scale is made simply by counting the number of elements in the entity set. The attribute always takes the form “number of occurrences of x”. All arithmetic analysis of the resulting count is possible. Examples are numbers of failures during a testing activity or the number of people working in an organization. There is only one to measure them: to count them. Another example is that it is incorrect to say that LOC is an absolute scale measure of a program size; i.e. we can measure number of characters or number of bytes.

We mention that a measure of central tendency tells something about where the “middle” of the set is likely to be, while a measure of dispersion tells us how far data points are from the middle. A summary of measurement scales and the statistical method relevant to them is given in table 0-1 [Fenton97, p.59].

Scale type	Examples of appropriate statistics	Appropriate statistical tests
Nominal	Mode	Non-parametric

	Frequency	
Ordinal	Median Percentile Spearman r	Non-parametric
Interval	Mean Standard deviation Pearson product-moment correlation Multiple product-moment correlation	Non-parametric Parametric
Ratio	Geometric mean Coefficient of variation	Non-parametric and parametric

Table 0-1: Summary of measurement scales and statistical methods

Usually qualitative research is mostly concerned with measurement on the nominal and ordinal scales, while quantitative research mostly treats measurement on the interval and ratio scales. Methods of statistical significance that are used to test hypotheses with nominal and ordinal data are known as *non-parametric tests*, while *parametric tests* are used for data derived from interval and ratio measurements and are more powerful. Choose of test depends also on whether we have one sample or more than one sample of data. Examples of parametric tests are Z test and t-test for one sample case or two independent samples, and One-way ANOVA or n-way ANOVA (Analysis of Variance) for more than two independent samples. Non-parametric tests in a one-sample situation for nominal data are the binomial test or chi-square one-sample test. In these tests we test for significant differences between the observed distribution of data among categories and the expected distribution based on a null hypothesis. Chi-square test can also be used for ordinal data and with several samples.

Defining and collecting metrics in an organization need resources and is costly. Therefore determining what to measure is not a trivial task. Metrics should be tied to the goals of the organization's measurement process. The Goal-Question-Metric (GQM) approach [Wohlin00] is based upon the assumption that an organization must define goals for itself and its projects, and trace these goals to metrics by defining a set of questions.

[Poulin01] outlines some of the basic metrics we might consider for component-based software engineering projects:

- Schedule: actual vs. planned.
- Productivity: Total development hours for the project/total number of LOC.
- Quality: Total number of defects and severity.
- Product stability: Number of open and implemented change requests that affect the requirement baseline.
- Reuse%: Reused LOC/Total LOC.
- Cost per LOC.

For components, Poulin adds:

- LOC per component.
- Labor: Effort expended per component.
- Classification of the component: New code, changed code, built for reuse, reused code, etc.
- Change requests per component as indication of the volatility of component design.

- Defects per component as a measure of the reliability of the component.
- Cost per component.

For design and coding, Poulin suggests:

- Average number of methods per class or component as a measure of complexity.
- Average method size. Poulin mentions that more than 24 LOC for C++ and Java indicates design problem!
- Coupling: Number of subsystem-to-subsystem relationships. It should be less than the number of class-to-class relationships within a subsystem.
- Number of comment lines.
- Specific metrics proposed in the literature for object-oriented systems such as depth of inheritance tree.

GQM Method

The Goal-Question-Metric (GQM) method is a top-down approach for defining metrics [Basili84]: The organization should define goals derived from the industrial interest, define a set of questions for each goal, and finally define lean and relevant metrics for answering questions which are traceable to goals. The GQM method provides a template to define goals that include:

- Object of study: product, process, resources, ...
- Purpose: characterization, evaluation, prediction, improvement, ...
- Quality focus: cost, correctness, defect removal, changes, reliability, ...
- Viewpoint: user, customer, manager, developer, ...
- Environment: team, project, product, ...

[Briand02a] introduces the GQM/MEDEA method: GQM/Metric DEfinition Approach. The method adds the notion of empirical hypotheses to GQM and has its focus on the construction of *prediction systems*; i.e. models that establish a correspondence between software attributes. One of the metrics quantifies the dependent variable of a product or process and the other metrics quantify independent variables of the model. Based on the goals of the organization, a set of empirical hypotheses is defined that are subject to experimental verification. An *empirical hypothesis* is a statement believed to be true about the relationship between one or more attributes of the *object of study* and the *quality focus*. Examples of an empirical hypothesis is: “The larger the import coupling of a software part, the larger is fault-proneness”, where import coupling refers to the dependence of a software part on other software parts, and fault-proneness is measured in number of faults in a software part. The GQM goal becomes:

- Object of study: high-level design of a software system
- Purpose: prediction
- Quality focus: fault-proneness
- Viewpoint: development team and project leader
- Environment: low-reuse software projects (high level of reuse is considered as a confounding factor)

Data Analysis Methods

A good overview of some existing studies relating object-oriented design and system quality is given in [Briand02]. However, the classification of data analysis methods and some of the suggested metrics are relevant for our case. The studies are classified into two categories:

1. *Correlation studies*: These are studies which by means of univariate and multivariate analysis try to demonstrate a statistical relationship between one or more attributes of a system’s structural properties (as independent variables) and a quality attribute (as dependent variable).
2. *Controlled experiments*: these are studies that control some structural properties of the system (independent variables) and measure the performance of subjects (as dependent variable).

Our study belongs obviously to the first category. We will also use other usual techniques for analyzing measurement data described in most literature and statistical packages. Some examples are given in [Fenton97]:

- Box plots to have a summary of the range of the dataset. We can show size of the components, number of faults per component, etc.
- Scatter plots to show relationships between two variables. Example is size of component vs. number of faults.
- Correlation analysis to confirm whether there is a true relationship between two variables.
- Linear regression for expressing an association as a linear formula.

Statistical Analysis

This subchapter briefly describes the statistics used in this thesis. We present the concept of dependent and independent variables, hypothesis testing and p-values, followed by a description of t-tests and regression analysis.

Hypothesis testing and p-values

The basis for statistical analysis is hypothesis testing, described in [Wohlin00]. A hypothesis is stated formally and the data collected is used to, if possible, reject the null hypothesis.

Two hypotheses have to be formulated:

- *A null hypothesis - H_0* : This hypothesis states that there are no underlying trends or patterns in the collected data; the only reason for any differences is coincidental. This is the hypothesis that the analyzer wants to reject with as high significance as possible.
- *An alternative hypothesis - $H_1, H_2...$* : This is the hypothesis in favor of which the null hypothesis is rejected.

In most studies results are commonly summarized by a statistical test, and a decision about the significance of the result is based on a *p-value*. Assuming the study was designed according to good scientific practice, the strength of the evidence is contained in the p-value. The p-value provides a sense of the strength of the evidence against the null hypothesis. The lower the p-value, the stronger the evidence against the null hypothesis. The researcher decides what significance level to use. The most commonly used level of significance is 0,05. When the significance level is set at 0,05, any test resulting in a p-value under 0,05 would be significant.

Dependent and independent variables

Dependent variables are variables that change if we change the independent variables. In a way independent variables can be seen as inputs to a process causing changes to the dependent variable as illustrated in figure 0-8 (taken from [Wohlin00]).

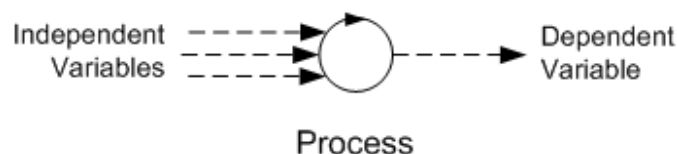


Figure 0-8: Illustration of independent and dependent variables

For instance if we want to study the effect a new development method has on the productivity, then the independent variables are the old and the new development method

and the dependent variable is a chosen measurement of productivity i.e. man-hours/MKLOC.

t-test

The t-test is a parametric test used to compare two independent samples. The result of the test is the probability of the two samples having equal average values.

$$t_0 = \frac{\bar{x} - \bar{y}}{S_p \sqrt{\frac{1}{n} + \frac{1}{m}}}, \text{ where } S_p = \sqrt{\frac{(n-1)S_x^2 + (m-1)S_y^2}{n+m-2}}, \text{ and } S_x^2 \text{ and } S_y^2 \text{ are the individual sample variances}$$

Figure 0-9: t-test calculations

Regression Analysis

Regression analysis is used to establish a correlation between two variables. Regression means fitting the data on to a curve. The linear regression line is the straight line that minimizes the sum of the quadratic distances to each data point.

The correlation coefficient *r* (also called *Pearson correlation coefficient*) is calculated as:

$$r = \frac{n \sum (xy) - \sum x \sum y}{\sqrt{[n \sum (x^2) - (\sum x)^2] [n \sum (y^2) - (\sum y)^2]}}$$

Figure 0-10: The correlation coefficient r

The *r-value* is between -1 and +1, and if there is no correlation *r* equals zero. The correlation coefficient measures only linear dependency and is only meaningful if the scales of *x* and *y* are interval or ratio, and works good for data that is normally distributed. *R-square* or *R²* measures the goodness-of-fit of the estimated regression line in terms of the proportion of the variation in the dependent variable explained by the fitted regression line. Thus, the value of *R²* = 0.894 simply means that 89.4% of the variation in the dependent variable is explained or accounted for by the estimated regression line. This information is quite useful in assessing the overall accuracy of the model.

Adjusted R-Square adjusts for the degrees of freedom lost in the process of estimating the regression parameters. If we have two parameters, *A* and *B*, the remaining degrees of freedom can be determined as *v* = *n* - 2. Hence, the adjusted *r-square* is a better measure of the goodness-of-fit of the estimated regression line than its nominal/unadjusted counterpart. It is always smaller in value than the unadjusted.

A *scatter plot* is good for assessing dependencies between variables. By examining the scatter plot we can see how outspread or concentrated the data points are, and if there is a tendency of a relation. Atypical values (outliers) may also be identified.

A *residual plot* shows how the different values are distributed above or below the regression line, and is used to check whether the values are evenly distributed above or below this line.

The *p-value* of the regression line coefficients indicates the probability (1-p) that the coefficients are correct and that we have a good correlation, hence we want as small p-values as possible.

Validity of Results

Threats to validity are classified and elaborated in [Wohlin00]. Adequate validity refers to that the results should be valid for the population of interest. First of all the results should be valid for the population from which the sample is drawn. Secondly it may be of interest to generalize the results to a broader population. Campbell and Stanley [Campbell63] define two types of validity: internal and external. Cook and Campbell extend this list to four types of threats to the validity of experimental results [Cook79]. These are:

- *Conclusion validity*: Concerned with the relationship between the treatment and the outcome. We want to make sure that there is a statistical relationship, i.e. with a given significance.
- *Internal validity*: If a relationship is observed between the treatment and the outcome, we must make sure that it is a causal relationship, and that it is not a result of a factor of which we have no control or have not measured.
- *Construct validity*: Concerned with the relation between theory and observation.
- *External validity*: Can the results of the study be generalized outside the scope of the study? Is there a relationship between the treatment and the outcome?

Overview of Empirical Strategies

There are three major types of investigations (strategies) that may be carried out [Wohlin00]:

- *Survey*: An investigation performed in retrospect. The primary means of gathering qualitative or quantitative data are interviews or questionnaires. These are done by taking a sample which is representative from the studied population. The results from the study is analyzed and generalized to the sample of which the sample was taken.
- *Case study*: Used for monitoring projects, activities or assignments. Data is collected for a specific purpose throughout the study. Based on the data collection, statistical analysis can be carried out. The case study is normally aimed at tracking a specific attribute or establishing relationships between different attributes. A case study is an observational study while an experiment is a controlled study. A case study may be aimed at building different kinds of models. Analysis methods include linear regression.
- *Experiment*: Normally performed in a laboratory environment which provides a high degree of control. The experimenting subjects are assigned to different treatments at random. The objective is to manipulate one or more variables and control all others.

Software Size

The most common metrics on the size of a software product is LOC. However, in practice, we must be careful when using this metrics. Giving the size of product in LOC or KLOC can hide notions such as reuse and complexity. Besides it must be clear whether we have counted total lines of the code, non-commented lines of the code, only lines with statements, etc. [Fenton97, p.245] suggests three attributes for software size:

- *Length*: Physical length of the product.
- *Functionality*: the functions offered to the user.
- *Complexity*: problem complexity, the structure of the software such as the hierarchical structure, or the effort needed to understand the software.

STATE-OF-THE-PRACTICE AT ERICSSON

This chapter first describes the Ericsson context. We then briefly describe the GSN system architecture and framework followed by the definition of components in subchapter 0. We give a short presentation of the adapted RUP process (GSN RUP) at Ericsson in subchapter 0, followed by the state-of-practice regarding reuse and product-line engineering in GSN RUP. Finally in subchapter 0 we then present the current measurement practice at Ericsson, directly influencing our research work, followed by a short presentation of the main programming languages used in the GSN system; Erlang and C.

Ericsson Context

This subchapter and subchapter 0 are based on [Mohagheghi03a, p.3]. Telecommunication and data communication are converging disciplines, and packet-switched services open for a new era of applications. The General Packet Radio Service (GPRS) system provides a solution for end-to-end Internet Protocol (IP) communication between a mobile entity and an Internet Service Provider (ISP). The GPRS Support Nodes (GSNs) constitute the parts of the Ericsson cellular system core network that switch packet data. The two main nodes are the Serving GPRS Support Node (SGSN) and the Gateway GPRS Support Node (GGSN).

GSN Architecture

The GSNs were first developed to provide packet data capability to the GSM (Global System for Mobile communication) cellular network. A later recognition of shared requirements with the forthcoming UMTS system (Universal Mobile Telecommunication System) lead to reverse engineering of the developed architecture to identify reusable parts across applications and to evolve the architecture to an architecture that can support both products. The enhanced, hierarchical reuse-based GSN architecture is shown in figure 0-1. Both systems are using the same platform (WPP), which is a high-performance packet switching platform developed by Ericsson. They also share components in the business specific layer and the middleware layer (called *Common parts* in figure 0-1). The business-specific components offer services for the packet switching networks. The middleware provides a customized component framework for building robust, real-time applications for processing transactions in a distributed multiprocessor environment that use CORBA and its Interface Definition Language (IDL). The organization has also been adapted to this view: an organization unit is assigned to develop common parts, while other units develop the applications. The reusable assets are evolved in parallel with the products, taking into account requirements from both products.

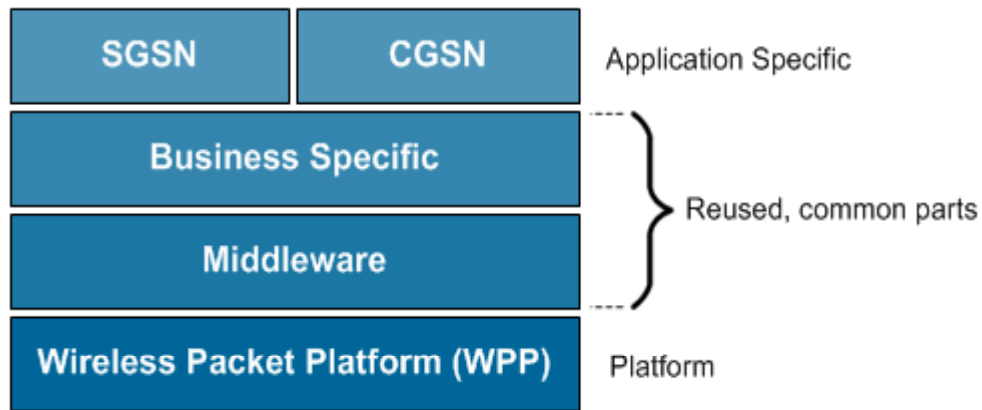


Figure 0-1: The GSN architecture

Figure 0-1 is one view of the system architecture, where the hierarchical structure is based on what is common and what is application specific. Other views of the architecture reveal that all components in the application and business-specific layers use the framework in the middleware layer, and all components in the three upper layers use the services offered by WPP.

The reused components in the common parts stand for 60% of the code in an application, where an application in this context is a product based on WPP and consisting of the three upper layers. Size of each application (not including WPP) is over 600 EKLOC (Non-Commented Lines Of Code measured in equivalent C code). Software components are mostly developed internally, but COTS components are also used. Software modules are written in C, Java and Erlang (a programming language for programming concurrent, real-time, distributed fault-tolerant systems). Several Ericsson organizations in Sweden, Norway and Germany have cooperated in developing the GSNs, but recently the development is moved to Sweden.

Definition of Components

RUP definition of Component: *A non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.*

[Kruchten00]

GSN RUP definition of Component: *"A component represents a piece of software code (source, binary or executable), or a file containing information (for example, a startup file or a Read Me file). A component can also be an aggregate of other components, for example, an application consisting of several executables."* [GSNRUP02]

Ericsson has defined the following design elements which also are referred to as *components*:

- **Subsystem:** The highest level of encapsulation used. A subsystem has formally defined interfaces and is a collection of function blocks.
- **Block:** Contains formally defined interfaces and is a collection of lower level (software) units. A block often implements the functionality represented by one or more analysis classes in the analysis model.
- **Unit:** A collection of (software) modules, i.e. classes/objects. Two units within the same block may communicate without going through an interface.

- *Module*: Corresponds to a source code file.

The aggregations of the different components are shown in figure 0-2. Refer to [Mohagheghi03] for more information on components.



Figure 0-2: Aggregation of the different component types

GSN RUP Development Process

RUP (Rational Unified Process) is a process for software engineering. RUP is built on the six best practices of software development:

- Develop software iteratively (as opposed to the traditional waterfall model)
- Manage requirements
- Use component-based architecture
- Visually model software
- Continuously verify software quality
- Control Changes in software

Ericsson AS uses a tailored or adapted version of the Rational Unified Process (RUP), called GSN RUP. Figure 0-3 shows the GSN RUP process map with phases and workflows. Refer to [Schwarz02 p.65] for more information on the GSN RUP process.

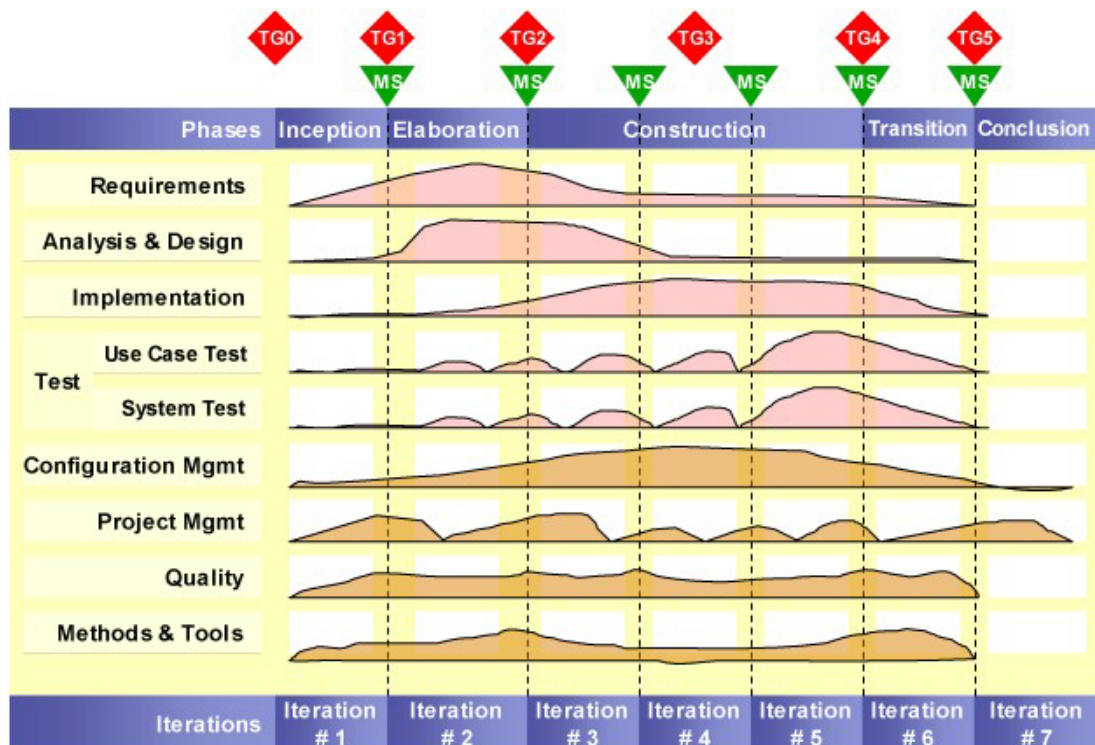


Figure 0-3: GSN RUP process map

Reuse and Product-Line Engineering in GSN RUP

After studying the GSN RUP process in our pre-diploma thesis [Schwarz02 p.83], we discovered that changes to the framework and identification of reusable components are performed in the application engineering activity only. There really does not exist any separate framework engineering activities; they are an indistinguishable part of the application engineering activity. Figure 0-4 illustrates this by drawing framework engineering activity as a separate process that *only* interacts with the application engineering activity. The framework engineering activity in GSN RUP is experienced based, and the Framework Responsible has few or no guidelines to rely on.

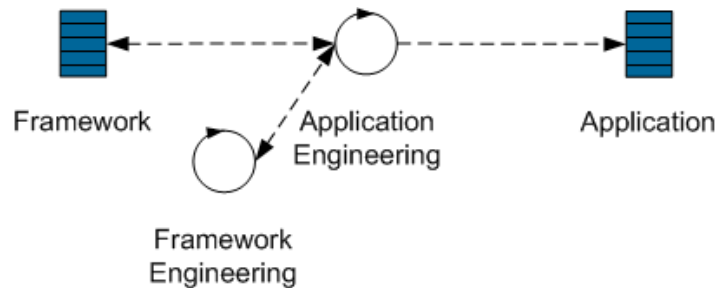


Figure 0-4: Product Line Engineering in GSN RUP as viewed by [Schwarz02]

Our study also showed that GSN RUP does not have activities that force the designers to identify already existing components, known as developing *with* reuse. Also GSN RUP lack a process of defining to what extent a component is developed *for* reuse.

Current Measurement Practice

As described in [Mohagheghi03], Ericsson defines two types of metrics:

- Direct metrics such as size of product or number of defects.
- Indirect metrics such as Defect Detection Percent.

Name	Description	Purpose	Quality Attribute
Requirements Stability (Percent)	Percent of high level requirements listed in the ARS not changed between TG2 and RFA	To check the stability of requirements	Stability, need for Extensibility
Requirements Review (Percent)	Percent of low level requirements (use case specifications and supplementary specifications) reviewed in a formal way	To check the quality assurance of requirements definition	Progress, has impact on Reliability
Appraisal to Failure Rate (A2F)	Person-hours used for Review & Inspections, rework included, divided with person-hours used for test and rework	To check the relation between early and late defect detection	Process compliance, has impact on Reliability
Defect Density (Defects/KLOC)	Quality on product: Defects identified/total code x 1000 Quality on work performed: Defects identified/new & modified code x 1000	To check the quality of product and/or work performed	Dependability/ Reliability
Defect Detection Percentage-Yield	Identified defects in a test phase in percent of current defects plus defects identified in later phases up to 6 months in operation	To check the efficiency of test phases	Dependability/ Reliability
Defect Removal Rate (Person-hours)	Total number of hours in a phase, rework included, divided by the number of defects identified and removed.	To check the effectiveness of verification and validation phases	Dependability/ Reliability
Productivity (Person-hours/LOC)	Total hours used in project, divided with total number of new and modified lines of code	To check project productivity	Process compliance
Planning Precision (Percent)	Absolute value of actual minus planned lead time (in number of weeks) divided with planned lead time multiplied with 100	To check project lead time	Scheduling capability

Table 0-1: Summary of Direct Metrics

Table 0-1 summarizes direct metrics. All the above metrics have ratio scale or absolute scale type. Defects are also classified based on:

- Priority: A, B and C, where A has the highest priority and should be solved in less time.
- Detection phase: A&D, Implementation, Test and maintenance.

Other types of direct metrics are:

- Classification of changes to requirements: new, removed and modified

- Classification of other modifications: modified solution, modified documentation and changed time schedule.

Table 0-2 summarizes indirect or derived metrics that are calculated by using direct ones in the measurement plan.

Name	Description	Purpose	Quality Attribute
Requirements Stability (Percent)	Percent of high level requirements listed in the ARS not changed between TG2 and RFA	To check the stability of requirements	Stability, need for Extensibility
Requirements Review (Percent)	Percent of low level requirements (use case specifications and supplementary specifications) reviewed in a formal way	To check the quality assurance of requirements definition	Progress, has impact on Reliability
Appraisal to Failure Rate (A2F)	Person-hours used for Review & Inspections, rework included, divided with person-hours used for test and rework	To check the relation between early and late defect detection	Process compliance, has impact on Reliability
Defect Density (Defects/KLOC)	Quality on product: Defects identified/total code x 1000 Quality on work performed: Defects identified/new & modified code x 1000	To check the quality of product and/or work performed	Dependability/Reliability
Defect Detection Percentage	Identified defects in a test phase in percent of current defects plus defects identified in later phases up to 6 months in operation	To check the efficiency of test phases	Dependability/Reliability
Defect Removal Rate (Person-hours)	Total number of hours in a phase, rework included, divided by the number of defects identified and removed.	To check the effectiveness of verification and validation phases	Dependability/Reliability
Productivity (Person-hours/LOC)	Total hours used in project, divided with total number of new and modified lines of code	To check project productivity	Process compliance
Planning Precision (Percent)	Absolute value of actual minus planned lead time (in number of weeks) divided with planned lead time multiplied with 100	To check project lead time	Scheduling capability

Table 0-2: Derived Metrics defined in the measurement plan

Programming Languages

The GSN system is programmed mostly in Erlang and C. Erlang is a concurrent, functional language and C is a procedural one. This subchapter briefly describes the two programming languages and their usage areas at Ericsson.

Erlang

Erlang is a functional programming language based on Prolog. It was originally developed by Ericsson more than 10 years ago. Erlang exists in both a free open source version and in a licensed version that includes support from Ericsson. Erlang was designed for handling telecommunication, but can also be used in other applications where reliability is important. Reliability is especially taken care of by the built in support for application distribution and allowing updating of a part of the program without restarting the whole application. This is made possible by the fact that two processes communicate by sending messages to each other and not by directly calling a function. Erlang (like Java) uses a compiler that compiles the application to run on a virtual machine (VM). This makes Erlang platform independent (at least the only thing to port to another platform is the VM and not the applications).

The Open Telecom Platform (OTP) was in 1996 developed by Ericsson to fulfill increasing demands for shorter development time and more openness in Erlang. It consists of an Erlang runtime system, a number of components and a set of design principles for Erlang programs. OTP is open in three ways: open for different platforms, open for different languages (it provides an interface for both C and Java) and open for protocols such as HTTP, SNMP, IIOP, FTP and TCP/IP.

Erlang/OTP is used in the GSN system as well as in several other of Ericssons products. Other companies such as One2One and Nortel Networks also use Erlang/OTP.

Example Erlang Code

```
-module(mathStuff).  
-export([factorial/1, list_length/1]).
```

```
factorial(0) -> 1;  
factorial(N) when N > 0 ->  
N * factorial(N-1).
```

```
list_length(List) ->  
list_length(List,0).
```

```
list_length([_ | Rest],N) ->  
list_length(Rest,N+1);  
list_length([],N) -> N.
```

C/C++

The C programming language was originally developed for and implemented on the UNIX operating system, on a DEC PDP-11 by Dennis Ritchie. C allows the manipulation of bits, bytes and addresses the basic elements with which the computer functions. C++ is an enhanced version of the C language. C++ includes everything that is part of C and adds support for object-oriented programming (OOP). In addition, C++ also contains many improvements and features over standard C, independent of object oriented programming. Both Erlang and C is used in the GSN product. Erlang is used in the control system. The control system is more complicated and considerable more stressed as it is located as a layer between the cellular phones and the rest of the system and thus providing an interface directly to the users. C is used in the transmission system and handles more low level tasks.

Equivalent Factors

To compare components written in different programming languages, Ericsson calculates LOC in equivalent number of lines in C by using the conversion shown in table 0-3. The multiplier is used to multiply the number of LOC to attain ELOC.

Language	Multiplier (Equivalent Factor - EF)
C	1
Erlang	3,2
Java	2,4
Perl	2
ASN	1
IDL	2,35

Table 0-3: Equivalent factors for calculating ELOC at Ericsson

LOC Definitions

In all references to lines of code (LOC) implicit means *uncommented* LOC. We use the following abbreviations for different types of LOC:

- *LOC*: Lines of Code (uncommented)
- *KLOC*: 1000 (Kilo) Lines of Code.
- *ELOC*: C-equivalent LOC having adjusted for the type of programming language. The adjustment is performed by multiplying LOC with an equivalent factor (EF, see subchapter 0 for more details) depending on programming language.
- *MKLOC/MEKLOC*: the number of modified and new KLOC/EKLOC.

In short:

- M - modified and new
- E – equivalent code, adjusted by equivalent factors
- K - thousands of

(M-E-K-LOC; modified/new - equivalent - thousand - lines of uncommented code).

RESEARCH METHOD

This chapter describes our research method outlined in figure 0-1. The remaining subchapters 0-0 describe the method in more detail. Our research goal is to statistically verify some hypotheses related to the quality attributes reliability and maintainability in the context of the GSN system at Ericsson. The hypotheses are described in chapter 0.

There are three types of empirical investigations that may be carried out as described in subchapter 0. Our empirical research method is based on *case studies* and *experiments*. Unlike case studies we do not have control over what data that should be collected; hence we must rely on the existing dataset at Ericsson. The GQM and GQM/MEDEA approaches described in subchapter 0 are useful when a measurement process is about to be started in order to determine which measures to collect, and are *top-down approaches* for defining metrics. However, in our case, we already have extensive data available from three projects. Our first step is to get an overview of the collected data at Ericsson. From *experiments* we use the concept of *hypothesis testing*. We define *dependent* and *independent variables* (described in subchapter 0), and then study the effect of changing one or more of the independent variables. We define empirical hypotheses that can be verified based on the available data. This is a *bottom-up approach* since our work is based on already existing data.

We start with a prestudy of the GSN system in subchapter 0 followed by an exploration of the raw dataset. In subchapter 0 we describe the data acquisition and hypothesis selection process in more detail. Finally we describe the research data extraction process, statistical analysis and hypothesis verification.

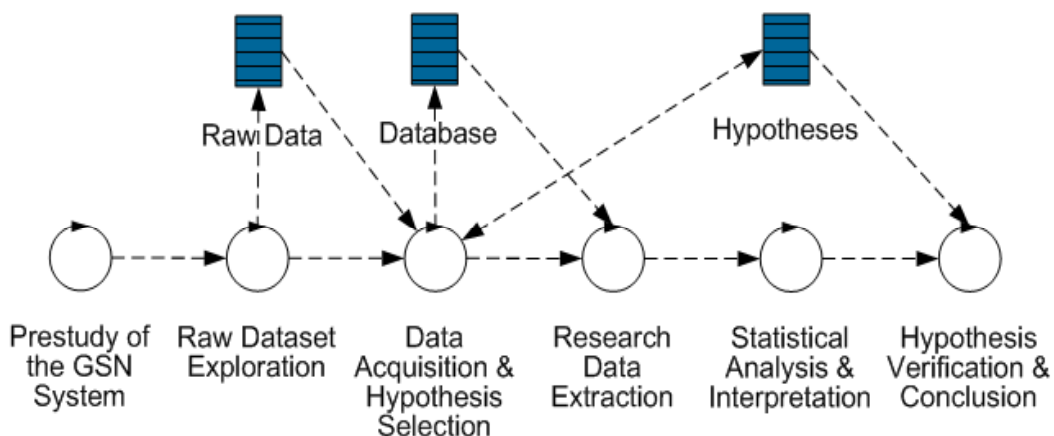


Figure 0-1: Research Method

Prestudy of the GSN System

Our method starts with a prestudy of the GSN system. We have used three GSN products. Refer to subchapter 0 for a description of the GSN system. A more detailed analysis of the GSN system can be found in our pre-diploma thesis [Schwarz02] and [Mohagheghi03].

Raw Dataset Exploration

We performed an exploration of the available data at Ericsson in order to get investigate the context of the collected data. We have the following data for exploratory analysis:

1. Trouble Reports (TRs) for three projects that are currently in the maintenance phase.
2. Change Requests (CRs) and Exemption Requests (ERs) for the three above projects.
3. Direct and derived metrics defined in table 0-1 and table 0-2.

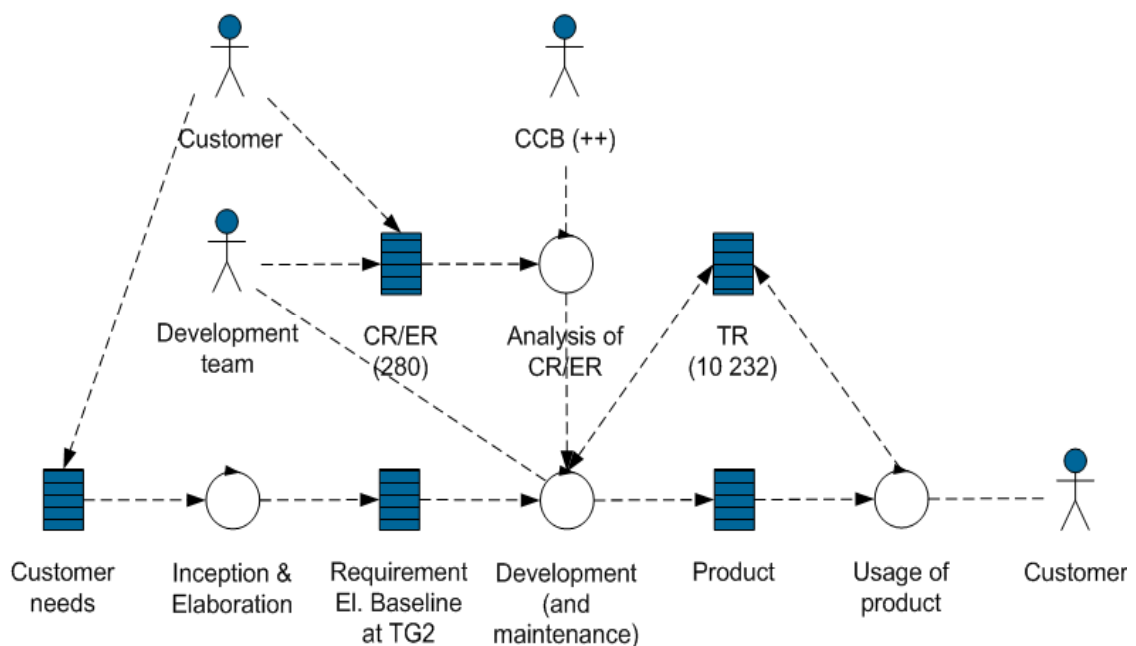


Figure 0-2: Overview of entry point for TRs and CRs(/ERs)

Throughout this document, when we use the term CR we implicitly also mean ER, except where specifically stated.

Figure 0-2 shows a brief overview of when TRs (see 0), CRs (see 0) and ERs are involved in the development of a product. CRs are typically created after the Application Requirement Specification (ARS) and traditionally provide new requirements or modified requirements (see 0 for our investigation on what type of changes the CRs proposes) for the product. TRs are issued when a fault of some type has been detected (see 0 for the different fault categories in our dataset). This could happen during development, maintenance and even during analysis & design as faults from previous versions are

detected. In this figure CCB (Change Control Board) is a forum to decide what changes are allowed into the product release after baseline. More than just the CCB is involved in the analysis of the CR as depicted in figure 0-5.

Trouble Reports - Corrective Maintenance

In order to verify the hypotheses related to reliability we use Trouble Reports (TRs) from three GSN projects. As mentioned in subchapter 0, defect removal using trouble reports is called *corrective maintenance*.

When a problem is detected during integration testing, system testing or later in the maintenance phase by the customer, a TR is written and stored in a database (called “Clear DDTs”). Note that:

- TRs are written for all types of faults (software, hardware, toolbox and documentation).
- There should be only one problem per TR.
- TRs should be written on the lowest possible element in the product structure (see subchapter 0). Therefore, the TR author must try to track down the fault to a subsystem or block level. At least the subsystem should be stated.
- Each TR has to be given a priority that reflects the severity of the fault. There are three priority levels: A, B, and C, where Priority A represents the highest priority level. Examples of priority A failures are failure of the node or failure in a function that affects the majority of the users. Examples of priority B failures are non-frequent restarts or failure in a function that affects a subset of the users. Examples of priority C failures are failures that do not cause any downtime or service outage.

The Clear DDTs database stores a lot of information regarding each TR, see appendix **Error! Reference source not found.** for more detailed information. The following list describes the most relevant fields:

- Unique identifier
- Headline containing a short description
- Priority A, B, C (see appendix **Error! Reference source not found.**)
- Current status of the TR
- Affected product, development project, subsystem, block and unit
- Fault code, a code that describes the cause of the fault (see appendix **Error! Reference source not found.**)
- Answer code, a code that describes how a TR is handled (see appendix **Error! Reference source not found.**)
- Number of man-hours used to correct the fault
- Detection workflow (A&D, Implementation, Test or Maintenance)
- Minimum test level required (unit test, large basic test, use case test, etc)
- Duplicate of: Used if the TR is a duplicate of another TR
- Enhancement: If the TR is an enhancement request that generates a new CR

A TR can be in one of the following states (the abbreviations are used in figure 0-3):

- **N:** New
- **A:** Assigned
- **I:** Investigated
- **R:** Resolved
- **V:** Verified
- **Z:** Followed up
- **X:** Finished
- **T:** Sent to TR Forum
- **F:** Forwarded

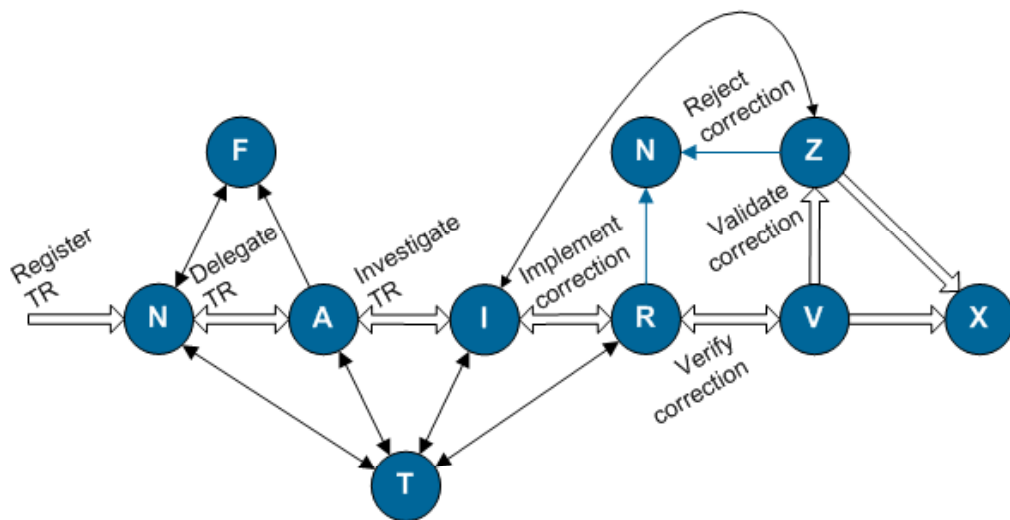


Figure 0-3: TR process with state transitions

Available Data

Table 0-1 shows the number of TRs used in our study. As shown in this table some TRs were deleted as a result of the normalization process described in subchapter 0. More details are shown in appendix **Error! Reference source not found.**

	#TR
Total:	13007
- Deleted or Duplicates:	2775
= TRs used:	10232

Table 0-1: Number of TRs used

Change Requests - Perfective Maintenance

In order to verify the hypotheses related to maintainability we use Change Requests (CRs) and Exemption Requests (ERs) from the same three GSN projects. As mentioned in subchapter 0, changes to functional and non-functional requirements are called *perfective maintenance*. At Ericsson, product changes and changes of the project plan is handled by Change Requests. ERs cover changes (within the total project time plan) that require deviation from processes, methods, routines, tools etc.

The GSN RUP is an incremental process that has the same phases as in the RUP in addition to an ending phase; the Conclusion phase. At the end of the Elaboration Phase, the Requirement Elaboration Baseline is defined (called Toll Gate 2 or TG2 internally). After this milestone, any changes should be handled by writing a CR (or ER) as shown in figure 0-4.

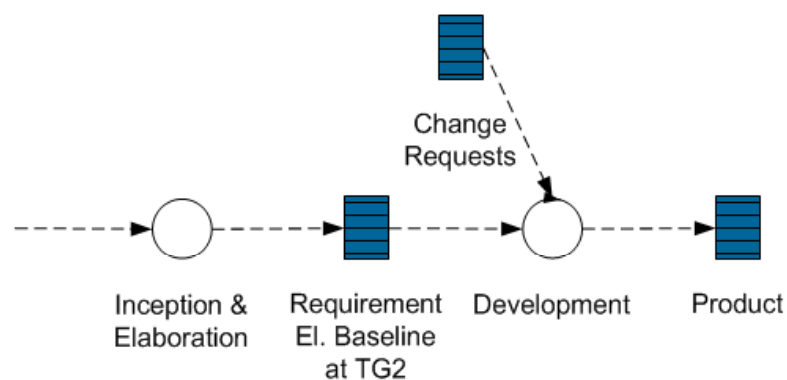


Figure 0-4: CR in timeline

CRs and ERs are written in MS word and should be based on a template with pre-defined fields. The CR/ER template includes the following information:

- Reason for writing this CR/ER
- Baseline affected (Project, Requirement, Environment and Product)
- Phases affected (TG1 scope or TG2 scope)
- Sub-projects affected
- Systems affected (3 levels of granularity)
- Description of proposed change
- Impact on cost and schedule

A CR/ER can be in one of the following states (the abbreviations are used in figure 0-5):

- **N: New** - the CR has been created and inspected by the CR administrator. The CR administrator should decide what impact analyses are needed and the person responsible for the investigation. The impact analysis may cover cost, added value of implementing the CR, time schedule impacts, products affected, and technical details.
- **R: Rejected** - the change is rejected.
- **I 1: Investigate** - the change is sent for investigation.
- **I 2: Investigated** - the change is investigated and ready for decision-making.
- **A: Approved** - a change is approved for implementation.

- **I 3: Implemented** - a change is implemented and ready for verification.
- **V: Verified** - changes are verified.
- **C: Closed** - a CR has been implemented in the baseline affected.

All this classification data has a nominal scale. We can find distribution over classes. Figure 0-5 shows the CR process.

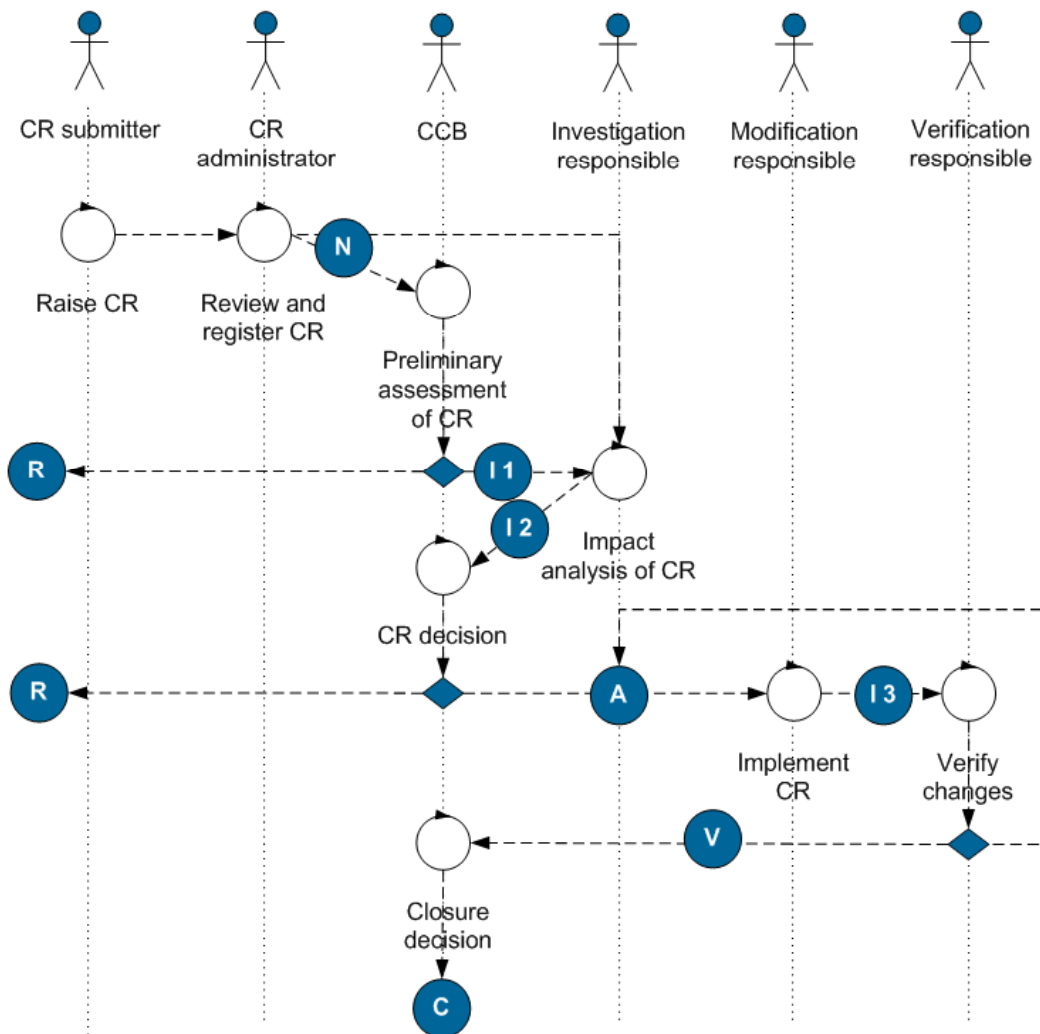


Figure 0-5: CR process with state transitions

Data Acquisition and Hypothesis Selection

This subchapter describes the data extraction process outlined in figure 0-6. We have created several tools in C# to assist this process, and a more detailed description of these tools can be found in appendix **Error! Reference source not found.** The process starts with an exploration phase, providing an overview of the available dataset. We then selected hypotheses and appropriate data based on the information from the exploration phase. We then parsed the selected data using a self-created parsing tool and transferred the data to a SQL database. Finally we performed a normalization and verification step to improve the quality of the dataset. The resulting database constitutes our research data. The following subchapters describe this process in more detail.

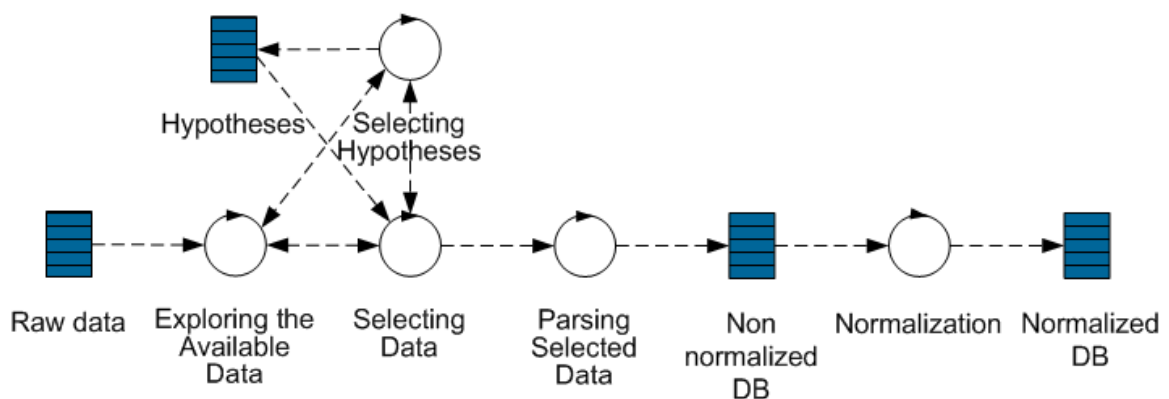


Figure 0-6: Research data acquisition and hypothesis definition process

Raw Data

Our dataset consists of TRs and CRs/ERs.

At Ericsson, TRs are managed by a system called “Clear DDTS” which consists of a database and a web interface. The database runs under UNIX and uses individual text files which are updated by perl scripts. At extraction time (29.01.2003) the database contained 13011 TRs. Each TR is stored as plain text in separate files, allowing convenient parsing of the information (see appendix **Error! Reference source not found.** for more information).

CRs/ERs are written in MS Word using a CR template and stored on a file server; hence the information is not searchable. Thus, we decided to transfer the data to our own database using a web interface we created using MS .NET and C#.

Exploring the Available Data

We created a tool in C# to assist us in exploring the available TR dataset. The tool traverses all the text files, extracts all the existing fields and creates a summary text file. The number of occurrences is counted for each field along with a count of the registered values and some sample values. We used this summary file in order to get an overview of the raw dataset and to decide which fields we wanted to transfer to our research database. The exploration revealed that there was a lot of inconsistency in the TR database. For example, the different fields had been renamed several times, apparently from one project to another. This made a search in the current database a hard task, as we were interested

in comparing results from many projects. In order to solve this problem we decided to create our own SQL database and transfer relevant data. By doing this we had better control over the data, and we could perform some normalization steps in order to improve the quality of the data base. The exploration phase also revealed that many values had been stored in different ways, i.e. the subsystem “CIS” had been entered in several ways, i.e. “CIS”, “cis” and “CIS_254_XXX”. The normalization step described in subchapter 0 addressed this problem.

Our exploration of the CR dataset revealed that Ericsson in many instances had not used the standard template, but instead written an ad-hoc document with the proposed changes. This made our data collection difficult as described in 0. The source code for all the tools is available on CD-ROM.

Selecting Hypotheses

The data exploration phase gave us detailed information about the available dataset, both what data that actually had been registered, the number of occurrences of each field, and also an indication of the data quality. We used this knowledge to define our research hypotheses. This work was done in cooperation with Parastoo Mohagheghi at Ericsson. The resulting hypotheses are presented in subchapter 0.

Selecting Data

Based on the defined hypotheses we selected the necessary data that needed parsing and transferal to our research database. As mentioned, several fields contained the same information; hence we were forced to merge several fields. We created a text file that specifies which fields that shall be extracted and also which fields that shall be merged into a common field. This file was later used by our parsing program.

Parsing Selected Data

We created a parsing tool in C# to carry out the parsing. The tool then reads each TR text file, looks for the specified fields, merges the fields if necessary, and finally creates a SQL insert statement that updates our database with a new record.

We received a text file containing path names of each module (source code files) for the entire system. This file was used in order to register the number of LOC for each unit. We created a tool that analyzed each line and summarized the LOC of each programming language. The file contained files on this format: LOC , MLOC
subsystem//block/unit/module.ext. The tool parses each line and updates the table “Unit” with the accumulated LOC for each programming language (identified by the file extension .c, .h, .erl, .hrl, .java, .idl, .asn) and the corresponding block ID. The “Block” table is also updated with the corresponding subsystem ID. In addition to the LOC information, these path names provide valuable information about unit-block and block-subsystem relationships needed by some hypotheses.

The CRs are accessible from a static HTML file on the Ericsson intranet. This file provides information about the title, status, baseline affected and more for each CR. Our program parsed this HTML file and inserted the relevant information into our database. Since the CRs only existed as Word documents, and because Ericsson had not used a fixed CR template, we manually had to enter all the information into our database for each CR. This was possible since the numbers of CRs was quite low. We created a web

interface in order to improve this manual task. This also allowed us to register some additional information as we looked through each CR, i.e. we registered if the CR was a new / modified requirement or a new / modified solution.

Normalization and Verification

After inserting the raw data into the two tables, we then used 25 additional tables to normalize the database and to register additional information. The normalization process was necessary both to bring under control the massive dataset and also for quality checking of the data, as done in [Votta00]. The normalization revealed numerous typing errors in many of the fields as well as multiple ways of denoting the same; e.g. "Unknown" and "?". All of these issues were handled by manually updating the field values and to let a few SQL-commands remove the duplicates and update the fields pointing to the removed records. To make this process easier and without any mistakes potentially destroying data, we created a tool for normalizing the database implemented in C#, see appendix **Error! Reference source not found.**

After normalizing the data we thoroughly verified the data looking for inconsistencies. This was done by randomly selecting different data and cross check them with the source data.

Normalized Database

The research database was implemented using Microsoft SQL Server 2000. Both the Change Request and Trouble Report dataset is stored in the same database, although in different tables. One CR accounts for one record in the table "ChangeRequest" and likewise one TR accounts for one record in "TroubleReport". The other tables were used in the normalization step to improve the quality of the data, see subchapter 0. Appendix **Error! Reference source not found.** provides more information about the SQL database, including an ER-diagram. The database is available on CD-ROM.

Research Data Extraction

The dataset for each hypothesis was extracted by using SQL queries. The numerous different SQL queries are found in the appendix **Error! Reference source not found.** The data was then entered into a spreadsheet for further analysis. We recommend using the ER diagram found in appendix **Error! Reference source not found.** as a starting point to both understanding the structure in which the data is organized and finally to write new SQL queries and extract data.

Statistical Analysis, Interpretation and Hypothesis Conclusion

We used Microsoft Excel for data visualization and statistical analysis. We used t-tests and regression analysis (described in subchapter 0). All spreadsheets are available on CD-ROM. With the statistical analysis we were able to either accept or reject each null hypothesis. The results are presented in chapter 5.

RESEARCH HYPOTHESES AND RESULTS

In this chapter we define our hypotheses regarding reliability using trouble reports in subchapter 0, hypotheses regarding maintainability using change requests in subchapter 0 and a hypothesis that combines reliability and maintainability in subchapter 0. Each hypothesis contains a description of the dataset used along with the results found, statistical analysis and a discussion (including validity) of the results. The hypotheses have been defined in cooperation with Parastoo Mohagheghi.

Table 0-1 shows an overview of the hypotheses. In the following we have assigned an identity to each hypotheses and prediction model. The first character is for the quality attribute or activity (R for Reliability, M for Maintainability and C for Combination of the R and M), the second character is “H” for Hypotheses and the third character is a number. H0 is an abbreviation for the null hypothesis, while H1, H2, etc. is an abbreviation for the alternative hypotheses.

Overview of Hypotheses

Hypothesis	Area	Metric	Granularity	References
Quality attribute: Reliability				
RH1: Reused components have the same reliability as other components	Reuse and Reliability	Defect Density	Subsystem and Block	
RH2: Larger components are not more fault-prone	Size and Reliability	Defect Density	Subsystem and Block	[Basili84a]
RH3: The fault categories are evenly distributed	Fault categories	Fault Type	GSN	[Wu00]
RH4: There is no difference in defect density between units programmed in Erlang and units programmed in C	Programming languages and Reliability	Defect Density	Unit	[Fenton97, p.499]
RH5: Most faults are corrected at once	Correction time	Defect Removal Rate	GSN	[Defamie99]
RH6: There are just as many TRs related to coding than to other faults	Fault categories & development phases		GSN	
RH7: 20% of all units account for 80% of all TRs	Fault distribution	Defect Density	Unit	
RH8: Most faults are marked as being of medium severity	Fault severity		GSN	
RH9: At least 70% of all remaining faults are reported in the test phase	Test phase effectiveness	Defect Removal Rate and Failure Rate	GSN	
Quality attribute: Maintainability				
MH1: Most CRs are due to modified requirements	Change cause		GSN	[Votta00], [Weber03]
MH2: Reused and non-reused components are equally change-prone	Reuse and Change-Proneness	Requirement Stability	Subsystem	
MH3: Most CRs and ERs are accepted and implemented	Response	Requirement Stability	GSN	
Combination of Reliability and Maintainability				
CH1: There is no linear correlation between #CRs and #TRs	Change Impact		Subsystem	

Table 0-1: Overview of hypotheses

Validity of Results

The external validity of all our hypotheses is threatened by the fact that the entire dataset is taken from only one organization. The external validity of the conclusions presented in this chapter are difficult to assess until similar results from other organizations is compared or included in our dataset. Internal-, conclusion- and construct validity are all discussed for each of the hypotheses in this chapter.

Reliability

This subchapter is written in cooperation with Parastoo Mohagheghi. This subchapter contains nine hypotheses (RH1-RH9). Number of defects is an indirect indication of reliability. Therefore the dependent variable in these hypotheses is usually *fault-proneness*; i.e. the number of TRs stored per product; i.e. a high-level package (Common vs. application specific), subsystem, block, unit or module. In some cases we use defect density defined as:

Defect density = Number of known defects / product size in KLOC.

[Fenton97, p.171] discusses the importance of right granularity: if the granularity is too low, we may end up with a large number of software modules and a few TRs for each of them. This would make it difficult to find trends. By combining modules and look at blocks and subsystems, it may be possible to see patterns not clear at lower levels. But again we must be careful as for some hypotheses, blocks and subsystems may hide details such as modules that are outliers (with extremely high number of defects).

As independent variable, several metrics are discussed in [Briand02]:

- Size in LOC. Usually size by itself is the best indication of the complexity; i.e. larger components or modules are more complex.
- Import Coupling through method invocation. Sometimes invoking library or non-library modules are discussed separately.
- Export Coupling or the degree a module or component is used by other components.

RH1 – Reuse and Reliability

Background

When a component is reused in several projects, detected defects are removed and we may assume that reliability grows, although the component is modified. A survey performed with some developers at Ericsson presented in [Mohagheghi03a] showed that the developers believe that reused components are more reliable. We have not found any other similar studies.

Hypotheses

H0 (null hypothesis): Reused components have the same reliability as other components.

H1: Reused components are more reliable.

Dataset

As *components* we used subsystems and blocks. We selected all non-duplicate TRs that affected the SGSN 3.0 system and that had registered a specific subsystem/block. We measured component size as number of *uncommented* lines of code (KLOC). Since it may be difficult to compare different programming languages, we chose to calculate the results using both LOC and ELOC. We have also used MLOC to measure the modified LOC since the previous product version.

We defined the following two groups:

- *Reused*: Business specific and middleware components
- *Non-reused*: Application specific components

In total:

- Subsystem level: 12 subsystems (9 reused and 3 non-reused).
- Block level: 31 reused and 22 non-reused blocks.

Granularity

We first decided to perform the calculations on subsystem level. But since the number of subsystems are quite low (only three non-reused subsystems), we decided to perform the same analysis on block level in order to increase the level of significance (since there are many more blocks). Hence we present the results on both subsystem and block level.

Statistical Method

The dependent variables are defect density, and the independent variables are components (subsystems and blocks). We calculated defect density (#TR/KLOC, #TR/ELOC and #TR/MEKLOC) for each component and also the average defect density for reused versus non-reused components. We used a *t-test* to calculate a p-value used as basis for our decision whether to reject our null hypothesis or not.

Results – Subsystem Level

Table 0-2 shows the results for both the reused and non-reused subsystems. *%MOD* shows the degree of modification (MEKLOC/EKLOC). We notice that non-reused subsystems have the highest degree of modification. Table 0-2 shows that the reused subsystems have an average fault density of 1,87 (using #TR/KLOC) compared to 3,87 for the non-reused subsystems. We performed a t-test, and the value $P(T \leq t)$ *one-tail* is shown in table 0-3. A value of 15% means that there is a probability of 15% that the results are just coincidental. We see that the non-reused subsystems are **2,06** times more fault-prone than reused subsystems (using KLOC). If we use C-equivalent LOC (ELOC) the factor is **2,10**, and using #TR/MEKLOC the factor is **1.70**. **Error! Reference source not found.** shows the defect density for each subsystem. See appendix **Error! Reference**

source not found. for more detailed statistical information and detailed subsystem information.

Subsystems	#TR/ KLOC	#TR/ ELOC	#TR/ MEKLOC	%MOD
All	2,3742	2,5695	1,9806	50,6%
Reused (R)	1,8753	1,8753	1,6856	44,5%
Non-R (NR)	3,8706	3,8706	2,8656	60,2%
NR / R	2,06	2,10	1,70	

Table 0-2: Average fault density for reused vs. non-reused subsystems

t-Test: Two-Sample Assuming Unequal Variances	
P(T<=t) one-tail [#TR/KLOC]	0,1544
P(T<=t) one-tail [#TR/ELOC]	0,1390
P(T<=t) one-tail [#TR/MEKLOC]	0,2059

Table 0-3: T-test for subsystem defect density

Results – Block Level

Table 0-4 shows that the reused subsystems have an average fault density of 1,17 (using KLOC), compared to 2,75 for the non-reused subsystems. We performed a t-test, and the value $P(T \leq t)$ one-tail is shown in table 0-5. We see that non-reused blocks are **2,34** times more fault-prone than reused blocks (using KLOC). We also notice that the p-values have decreased to an acceptable level.

Blocks	#TR/ KLOC	#TR/ ELOC	#TR/ MEKLOC	%MOD
All	1,8289	0,8307	1,6639	48,0%
Reused (R)	1,1749	0,4411	1,1145	43,7%
Non-R (NR)	2,7505	1,3798	2,4379	57,5%
NR / R	2,34	3,13	2,19	

Table 0-4: Average fault density for reused vs. non-reused blocks

t-Test: Two-Sample Assuming Unequal Variances	
P(T<=t) one-tail [#TR/KLOC]:	0,0025
P(T<=t) one-tail [#TR/ELOC]:	0,0002
P(T<=t) one-tail [#TR/MEKLOC]:	0,0015

Table 0-5: T-test for block defect density

Evaluation and Discussion

The results indicate that reused components are more reliable, hence we *reject our null hypothesis*. Using results from the block level we see that non-reused blocks are **~2,3** times more fault-prone. We think that these are very interesting results regarding reuse and component based development. This conclusion also supports the opinions of the developers at Ericsson, as described in [Mohagheghi03a].

Another interesting result is the impact of using standard LOC versus ELOC. On the subsystem level the results show that non-reused subsystems are **2,06** times more fault-prone than reused subsystems using LOC and **2,10** using ELOC. This means that there is almost no difference in using LOC or ELOC. We think that this implies that the use of equivalent LOC is of no great importance at this level.

Table 0-2 and table 0-4 show the degree of modification for reused versus non-reused components. We see that the non-reused components are modified **~57%**, while the reused components are modified **~44%**. Table 0-5 shows a probability of **~0,2%** that the results are just coincidental. This means that the confidence increased dramatically by using block granularity instead of subsystem granularity.

Validity

If one group (i.e. reused components) is dominated by a specific type of components, this could be a confounding factor. If this is the case, then the fact that these components are developed for reuse may not be the main reason why they are more reliable, but instead because they belong to a group of components which are less fault-prone due to other circumstances (i.e. different type of logic/complexity etc). For example no reused components in the GSN system have user interfaces. This matter should be investigated further in order to verify this hypothesis. To improve *external validity* it can be useful to perform this analysis on other products and compare the results.

A threat to *internal validity* would be if for example more experienced employees are working with one type of components, i.e. reused components. Then the fact that these people are more skilled may be the reason why reused components are more reliable. The calculation of LOC values can affect internal validity. If there are errors in these values it may influence our results. However, this would only have a negative effect if the error affects one of the groups in particular. We have no evidence that this is the case. We have not identified any threats to *conclusion-* or *construct validity*.

Conclusion

We reject our null hypothesis. The results indicate that reused components are more reliable.

RH2 – Component Size and Reliability

Background

We want to study whether larger components are more fault-prone. Intuitively the hypothesis seems correct, but some studies show that larger modules are in fact less fault-prone and have better quality [Basili84a]. They may be developed more carefully or inspected better than smaller ones. See subchapter 0 for more information about regression analysis.

Hypotheses

H0 (null hypothesis): Larger components are not more fault-prone.

H1: Larger components are more fault-prone.

Dataset

As *components* we used subsystems and blocks. We selected all non-duplicate TRs that affected the SGSN 3.0 system and that had registered a specific subsystem/block. We measured component size using MEKLOC and EKLOC.

We defined the following two groups:

- *Reused*: Business specific and middleware components
- *Non-reused*: Application specific components

In total:

- Subsystem level: 12 subsystems (9 reused and 3 non-reused).
- 31 reused and 22 non-reused blocks.

Granularity

We first decided to perform the calculations on subsystem level. But since the number of subsystems are quite low (only three non-reused subsystems), we decided to perform the same analysis on block level in order to increase the level of significance (since there are many more blocks). Hence we present the results on both subsystem and block level.

Statistical method

The dependent variables are number of TRs, and the independent variables are components (subsystems and blocks). We counted the numbers of TRs for each component and used linear regression analysis to check the correlation between fault-proneness and size. We have created scatter plots to visualize the results. On the x-axis we have MEKLOC and EKLOC, and on the y-axis we have number of TRs. Two regression lines are drawn, one for reused and one for non-reused components, in order to visualize potential differences. On block level we also created a scatter plot showing MEKLOC on the x-axis and #TR/MEKLOC on the y-axis in order to see if there is any correlation between modification size and fault-proneness.

Results – Subsystem level

A residual plot is shown in figure 0-2 and a regression summary is shown in table 0-6. For EKLOC and #TR a scatter plot is shown in figure 0-8, a residual plot is shown in figure 0-9 and a regression summary is shown in table 0-7.

Figure 0-1: Scatter plot of MEKLOC (x-axis) and #TR (y-axis), subsystem level

A scatter plot with MEKLOC on the x-axis and #TR on the y-axis is shown in figure 0-1. The figure shows the linear regression line for reused and non-reused subsystems. The gradient of the regression line is higher for non-reused subsystems, indicating that non-reused subsystems are more fault-prone.

Figure 0-2: Residual plot for MEKLOC and #TR, subsystem level

Regression summary	
Adjusted R ² [Reused and Non-reused]:	0,6785
Adjusted R ² [Reused]:	0,8065
Adjusted R ² [Non-reused]:	0,3838
Regression line, P-value [Reused and Non-reused]:	9,0403E-06
Regression line, P-value [Reused]:	2,9656E-07
Regression line, P-value [Non-reused]:	0,0192

Table 0-6: Regression summary for MELOC and #TR, subsystem level

Figure 0-3: Scatter plot of EKLOC (x-axis) and #TR (y-axis), subsystem level

A scatter plot with EKLOC on the x-axis and #TR on the y-axis is shown in figure 0-3. The figure shows the linear regression line for reused and non-reused subsystems. The gradient of the regression line is higher for non-reused subsystems, indicating that non-reused subsystems are more fault-prone.

Figure 0-4: Residual plot for EKLOC and #TR, subsystem level

Regression summary	
Adjusted R ² [Reused and Non-reused]:	0,5581
Adjusted R ² [Reused]:	0,7669
Adjusted R ² [Non-reused]:	0,4634
Regression line, P-value [Reused and Non-reused]:	9,4977E-05
Regression line, P-value [Reused]:	9,0045E-06
Regression line, P-value [Non-reused]:	0,0077

Table 0-7: Regression summary for EKLOC and #TR, subsystem level

Results – Block level

For MEKLOC and #TR a scatter plot is shown in figure 0-5, a residual plot is shown in figure 0-7 and a regression summary is shown in table 0-8. A scatter plot of MEKLOC and #TR/MEKLOC is shown in figure 0-6. For EKLOC and #TR a scatter plot is shown in figure 0-8, a residual plot is shown in figure 0-9 and a regression summary is shown in table 0-9.

Figure 0-5: Scatter plot of MEKLOC (x-axis) and #TR (y-axis), block level

A scatter plot with MEKLOC on the x-axis and #TR on the y-axis is shown in figure 0-5. The figure shows the linear regression line for reused and non-reused blocks. The gradient of the regression line is higher for non-reused blocks, indicating that non-reused blocks are more fault-prone.

Figure 0-6: Scatter plot of MEKLOC (x-axis) and #TR/MEKLOC (y-axis), block level

A scatter plot with MEKLOC on the x-axis and #TR/MEKLOC on the y-axis is shown in figure 0-6. The figure shows no linear regression line for reused and non-reused blocks because there is no trend in the observed data.

Figure 0-7: Residual plot for MEKLOC and #TR, block level

Regression summary	
Adjusted R ² [Reused and Non-reused]:	0,6282
Adjusted R ² [Reused]:	0,5839
Adjusted R ² [Non-reused]:	0,8230
Regression line, P-value [Reused and Non-reused]:	9,17684E-17
Regression line, P-value [Reused]:	1,93E-10
Regression line, P-value [Non-reused]:	3,27E-13

Table 0-8: Regression summary for MEKLOC and #TR, block level

Figure 0-8: Scatter plot of EKLOC (x-axis) and #TR (y-axis), block level

Figure 0-9: Residual plot for EKLOC and #TR, block level

Regression summary	
Adjusted R ² [Reused and Non-reused]:	0,4716
Adjusted R ² [Reused]:	0,5391
Adjusted R ² [Non-reused]:	0,8558
Regression line, P-value [Reused and Non-reused]:	1,4004E-12
Regression line, P-value [Reused]:	1,1911E-08
Regression line, P-value [Non-reused]:	6,0608E-14

Table 0-9: Regression summary for EKLOC and #TR, block level

Evaluation and Discussion

The regression analysis visualized by the scatter plots indicates a certain correlation between fault-proneness and size, and this is obviously not a surprising result. The *adjusted R²* is 67% at subsystem level using MEKLOC, and 56% using EKLOC. At block level the *adjusted R²* is 63% using MEKLOC and 47% using ELOC. The residual plots show that the observations are quite evenly distributed above and below the regression line. The *P-values* for the regression lines are low, meaning that the probability for a random correlation is very low. However the adjusted *R²* values are also low, meaning that there are other factors than size that may explain why certain components are more fault-prone. One factor may be whether the component is reused or not, as shown in RH1. Since the adjusted *R²* values are as low as 40-60%, we *cannot reject our null hypothesis*.

Validity

We have not identified any threats to *internal validity*, *conclusion validity* or *construct validity*.

Conclusion

We cannot reject our null hypothesis. The correlation values are too low.

RH3 – Fault Categories

Background

[Wu00] introduces a fault model for component-based systems that classifies faults into three categories:

1. Inter-component faults: Faults introduced when integrating components.
2. Interoperability faults: Faults due to different operating systems, different programming languages, specification faults such as data misunderstanding or control misunderstanding.
3. Traditional faults or other faults.

We propose other categories as well such as:

- Intra-component faults: Faults within a component
- Inter-component faults: Faults introduced when integrating components
- Other faults: documentation, etc
- Requirement Management / Analysis & Design faults

It would be interesting to see the distribution for these the 4 categories. The distribution gives an overview of the properties of the corrective maintenance performed in SGSN.

Hypotheses

H0 (null hypothesis): The fault categories are evenly distributed.

H1: Most faults are inter-component.

Dataset

Each TR has registered a fault code that describes the category of fault the TR represents (see **Error! Reference source not found.** for a description of the fault codes used at Ericsson). We mapped the fault codes to our own classification groups using a conversion table (see appendix **Error! Reference source not found.**) and counted all TRs in each category. In this process we discarded groups that contained a low number of TRs.

Statistical Method

The dependent variables are fault-proneness in number of TRs, and the independent variables are percentage of TRs related to each fault category. We then found the distribution over the independent variables.

Results

Figure 0-10 shows the distribution over fault categories.

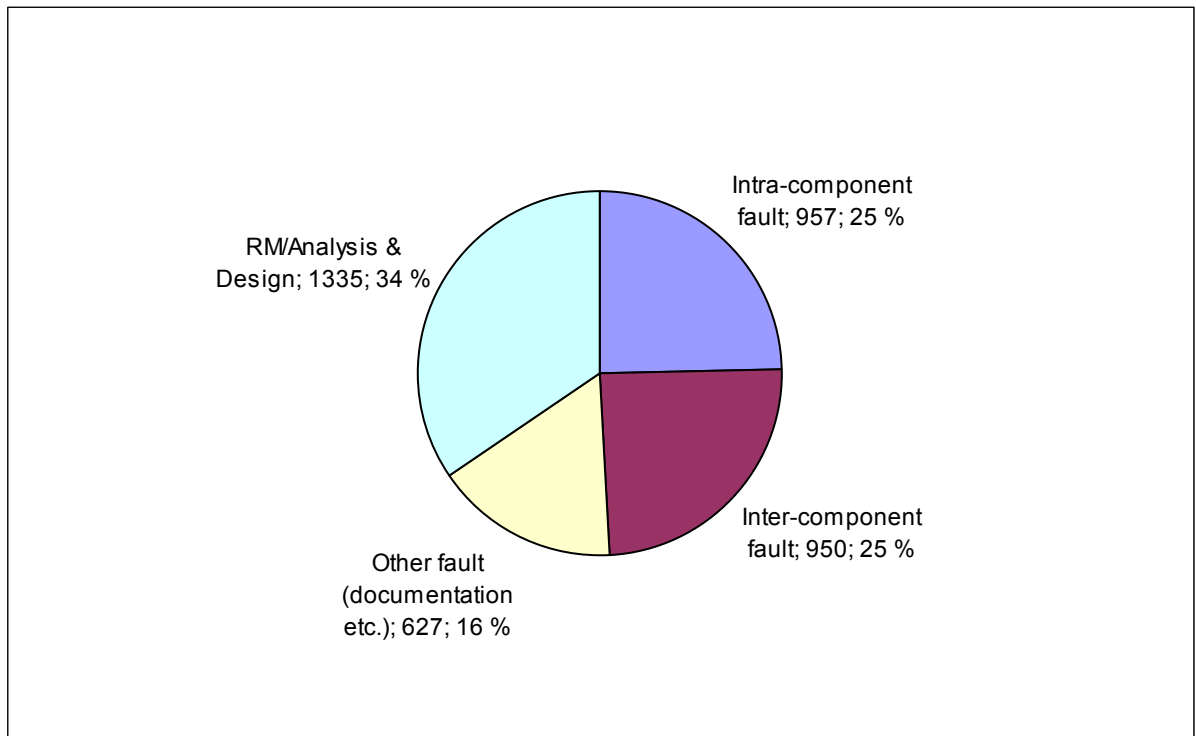


Figure 0-10: Distribution over fault categories

Evaluation and Discussion

Intra-component and inter-component faults are evenly distributed as shown in figure 0-10. The largest category is “RM / Analysis & Design” with 34%, while the “Other” category accounts for 16% of the TRs. We also notice that 50% of the faults are related to implementation (intra-component and inter-component faults). The intra-component and inter-component category is evenly distributed and hence we cannot reject our null-hypothesis.

Validity

We have not identified any threats to *internal validity*, *conclusion validity* or *construct validity*.

Conclusion

We cannot reject H0; the fault categories are evenly distributed.

RH4 – Programming Languages and Reliability

Background

There are studies on the effect the actual programming language has on defect density; an example is given in [Fenton97, p.499].

Hypothesis

H0 (null hypothesis): There is no difference in defect density between units programmed in Erlang (Erlang-units) and units programmed in C (C-units).

H1: Erlang-units are more fault-prone than C-units *or*

H2: C-units are more fault-prone than Erlang-units.

Dataset

We selected those units that are solely programmed in either C or Erlang, and used only these in our dataset. LOC for each unit and number of TRs are from GSN 3.0. The TRs are categorized in 4 categories:

- Intracomponent fault
- Intercomponent fault
- Fault regarding Requirement Management/Analysis & Design (RM/A&D)
- Other fault

and counted for each unit. To compare code size an equivalent factor (EF) is usually used. However we have not used ELOC in this hypothesis. How the results would be when using ELOC is summarized in the discussion section. Defect density was measured as number of TRs divided by KLOC and MKLOC (modified KLOC).

Statistical Method

The independent variable is units in Erlang or C, and the dependent variable is defect density. A t-test is used to test the null-hypothesis.

Results

The table below shows average LOC (both total and modified in GSN 3.0) for C-units and Erlang-units as well as the average number of TRs and number of TRs per LOC.

	% mod	#TR/KLOC	#TR/MKLOC
Erlang	41,84 %	2,16	5,15
C	65,54 %	1,51	2,30

Table 0-10: Summary of average values for both Erlang and C units

Erlang-units has 20% more faults per KLOC than C-units and 89,25% more faults per modified KLOC than C-units. By grouping the TRs into 4 fault types it is possible to identify if there are any particular fault types that accounts for the difference in the number of TRs.

	Intra-comp. #TR/MKLOC	Inter-comp. #TR/MKLOC	RM/A&D #TR/MKLOC	Other #TR/MKLOC	Total #TR/MKLOC
Erlang	0,70	0,96	0,92	2,58	5,16
C	0,46	0,23	0,27	1,34	2,30

Erlang/C	1,5	4,2	3,4	1,9	2,24
----------	-----	-----	-----	-----	------

Table 0-11: Summary of average values Erlang values divided by the C values

Table 0-11 shows a factor in difference from 1,5 to 4,2 for the different fault categories, in total the factor was 2,24.

A t-test was used on the number of TRs regarding intracomponent, intercomponent, RM/A&D and other failures per modified KLOC. These numbers are summarized in table 0-12 below.

	P(T<=t) one-tail
#TR/MKLOC	0,18 %
#TR[intracomponent]/MKLOC	18,31 %
#TR[intercomponent]/MKLOC	3,28 %
#TR[RM/A&D]/MKLOC	10,57 %
#TR[other]/MKLOC	1,46 %

Table 0-12: T-test values for the difference between Erlang- and C-units using MKLOC

Not surprisingly all the P-values are low, showing that there probably is a high difference in the number of TRs dependent on what programming language the unit is written in. Using the EF of 3,2 for Erlang units would have produced similar results only this time producing results in favor of Erlang.

Evaluation and Discussion

If we assume that all the units, regardless of what programming language it was written in, include approximately the same amount of logic we could calculate a new EF for Erlang in this project; $2805,54/1209,40 = 2,32^{\ddagger}$. This factor is remarkably near the factor found in table 0-11 for the relation between fault-proneness between Erlang and C, namely $2,24^{\ddagger}$.

	#TR	#TR/MEKLOC
Erlang	2,57	1,614
C	4,23	2,301

Table 0-13: Summary of average values for both Erlang and C units by using standard ELOC

When looking at table 0-13 we see that C-units are 1,43 (2,301/1,614) more fault-prone than Erlang. This could mean that the EF is incorrect or that the C-units actually are more fault-prone than Erlang-units.

	#TR	#TR/MEKLOC
Erlang	2,57	2,245
C	4,23	2,301

Table 0-14: Average values by using new ELOC (calculated by using EF=2,3)

By using a new EF of 2,3 (the middle value between 2,32 and 2,24 - see † above) we get a new result. Now the C-units are almost as fault-prone as Erlang-units (not surprisingly when the new EF is calculated both from fault-proneness and code size). Our assumption that Erlang and C units contain the same amount of logic could be incorrect and the nature of the logic the unit is set to implement in large decide what programming language it is implemented in. The difference in code size could of course also be ascribed the fact that C-units fulfill another role than Erlang-units (see subchapter 0). This difference in roles could also skew the results if for instance C-units implement more fault intensive code than Erlang or vice versa.

Source	Erlang EF	Method
Ericsson	3,2	Estimation used internally (unknown on what basis it has been calculated)
[Doug03]	1,46	Implemented 21 identical programs in C and Erlang and calculated a relation between LOC
This thesis	2,3	Comparing relation between LOC and fault-proneness for x units implemented in either Erlang or C in Ericsson GSN 3.0 (units with different implementations)

Table 0-15: Different EF for Erlang

Table 0-15 shows the different EF for Erlang from three sources and the way they where calculated. [Doug03] did however not calculate a EF, but we have used his data to do this (see appendix **Error! Reference source not found.** for the full dataset). This thesis proposes a value that lies almost exactly in the middle of the two first EFs. [Doug03] implemented the same 21 programs in several different languages (see column different implementations in table 0-15). Our thesis however uses units with different implementation. The language the units where programmed in was probably chosen because it was the most appropriate and not at random.

There are however differences in fault types regardless of EFs. Table 0-16 shows that there are twice as many intracomponent faults in than intercomponent faults in C units. This fact is almost the opposite for Erlang units where intercomponent faults outnumber intracomponent faults.

	Intra-comp.	Inter-comp.	RM/A&D	Other
Erlang	13,6%	18,6%	17,8%	50,0%
C	20,0%	10,0%	11,7%	58,3%

Table 0-16: Distribution of fault types for Erlang and C

We ascribe the some of the difference in RM/A&D faults to the difference in roles the C and Erlang units has in GSN (see 0 for more information).

By using plain LOC we find enough data to support a rejection of the null-hypothesis. When calculating ELOC we get a difference in fault-proneness dependent on whether or the equivalent factor is higher or lower than 2,3; if it is higher (i.e. 3,2) Erlang-units are less fault-prone than C-units and vice versa. The fact that this conclusion is so tightly connected with what kind of equivalent factor we use it is impossible to conclude that either programming language is more or less fault-prone than the other. Our guess is that they are just as fault-prone and that 2,3 seems like a reasonable EF for Erlang in GSN 3.0.

Validity

The source files could be analyzed to provide another way to measure the complexity of the units. This could improve the *internal validity* of our results. We have compared our results with data from two other sources to increase the *external validity*. However both the *external-* and *internal validity* of these results are probably dependent on more than just what type of language that is used. Experience of the developers and whether or not the language selected is the best to fulfill the tasks it is set to fulfill is only two factors that probably has great impact on the results (see the discussion section). We have not identified any threats to *construct-* or *conclusion validity*.

Conclusion

We cannot reject H_0 ; there is no difference in defect density between Erlang-units and C-units.

RH5 – Correction Time

Background

[Defamie99] discusses assumptions in a software reliability model called Non-Homogeneous Poisson Process model (NHPP). One of these assumptions is that “detected faults are immediately corrected”. By examining the severity classifications for the TRs, [Defamie99] concludes that in general this is not the case. Only about 55% of all TRs needed to be corrected within a few days.

Hypothesis

Corrected at once is defined as a corrected within 5 days.

H0 (null hypothesis): Most faults are corrected at once.

H1: Most faults are not corrected at once.

The hypotheses are checked against the average number of days until the TRs are marked as finished. The definition on the severity levels from A to C lacks a definition as to how fast the fault must be corrected as [Defamie99] had in their dataset. The distribution of severity levels is shown as part of 0.

Dataset

The dataset consists of non-duplicated TRs from all projects which had registered dates for both creation and completion (date verified). In addition only those TRs that were marked as finished (TRs marked with status X; see figure 0-3) were included in the dataset.

In total:

- 5 255 TRs

Statistical Method

Average value and histogram.

Results

An average of number of days was calculated and a histogram is used to display the distribution. The results are shown in the appendix in **Error! Reference source not found.** and **Error! Reference source not found.**

Only 28,81% of all TRs in this dataset were corrected within 5 days and less than half of the TRs were corrected within 14 days (see table 0-17; 50% of all TRs were corrected within 15 days).

% corrected	#days
25 %	5
50 %	15
75 %	44
100 %	458

Table 0-17: Percentile distribution

Evaluation and Discussion

Based on these results *we reject our null-hypothesis*. We have selected 5 days as a limit to what we define as "corrected at once". This number could of course be subject to debate; on whether or not it should be higher or lower, but even if we had selected a limit of 14 days we would still reject our null hypothesis.

Validity

We have not identified any threats to *internal-* or *construct validity*. However the fact that we might have selected a too low number of days (see Discussion above) is a threat to *conclusion validity*.

Conclusion

We reject H0 in favor of H1: most faults are not corrected at once.

RH6 – Fault Categories & Development Phases

Background

In this hypothesis we find the distribution of types of TRs.

Hypothesis

H0 (null hypothesis): There are just as many TRs related to coding than to other faults.

H1: Most faults are related to implementation

Dataset

As our dataset we used all non duplicated TRs. The TRs were categorized into the following 5 categories and counted:

1. Analysis & Design
2. Implementation
3. Documentation
4. Production
5. Other (including Hardware)

In total:

- 10 232 TRs

Statistical Method

Independent variables are the 6 categories defined above, and dependent variables are number of TRs.

Results

The distribution of fault categories is shown in table 0-18.

Fault Category	#TR	%
Analysis & Design	1254	12,3 %
Implementation	3366	32,9 %
Documentation	630	6,2 %
Production	86	0,8 %
Other (incl. hw)	2313	22,6 %
Unknown	2583	25,2 %
SUM	10232	100 %

Table 0-18: Distribution of fault categories

Initially there was an additional category called "Hardware", but the number of TRs was only 27 and ended up as being included in "Other" to avoid having a category with so few members. Considering that Ericsson Grimstad does not (normally) address hardware issues also talks in favor of not having a hardware category. The two categories "Other" and "Unknown" does not indicate whether or not it is a coding fault (the TRs related to hardware are unrelated to implementation, but they are too few to make a difference). If we assume that the distribution of implementation and non implementation faults are the same in these two groups as in the remaining 4 categories we get the results shown in table 0-20.

Fault Category	#TR	~%
Analysis & Design	1254	23%
Implementation	3366	63%
Documentation	630	12%
Production	86	2%
SUM	5336	100%

Table 0-19: Initial fault categories

Fault Category	#TR	~%
Non implementation (all groups other than implementation)	1970	37%
Implementation	3366	63%
SUM	5336	100%

Table 0-20: Regrouping into implementation and non implementation

As shown in table 0-20 the implementation category is almost twice the size as the non implementation category.

Evaluation and Discussion

The results here coincide with other similar researches showing that implementation faults are the dominating fault category. The assumption that the distribution of implementation and non implementation faults are the same in “Other” and “Unknown” may be incorrect and the dataset does not include any data to evaluate this.

Validity

The mapping of the fault categories could be incorrect causing a threat to the *internal validity*. The same is true for our assumption that the distribution of implementation and non implementation faults are the same in “Other” and “Unknown”. We have not identified any threats to *construct-* or *conclusion validity*.

Conclusion

We reject H0 in favor of H1: most faults are related to implementation.

Linking with RH8 and including fix hours for each TR

These results could also be linked with the hypothesis in RH8 and the distribution of A, B and C faults for each of the 5 categories (leaving out the Unknown group). When combining data on the time used for each TR and in which phase the fault was discovered we can estimate the cost of the different fault types in each phase. The full dataset is found in the appendix and the results are only summarized in figure 0-11.

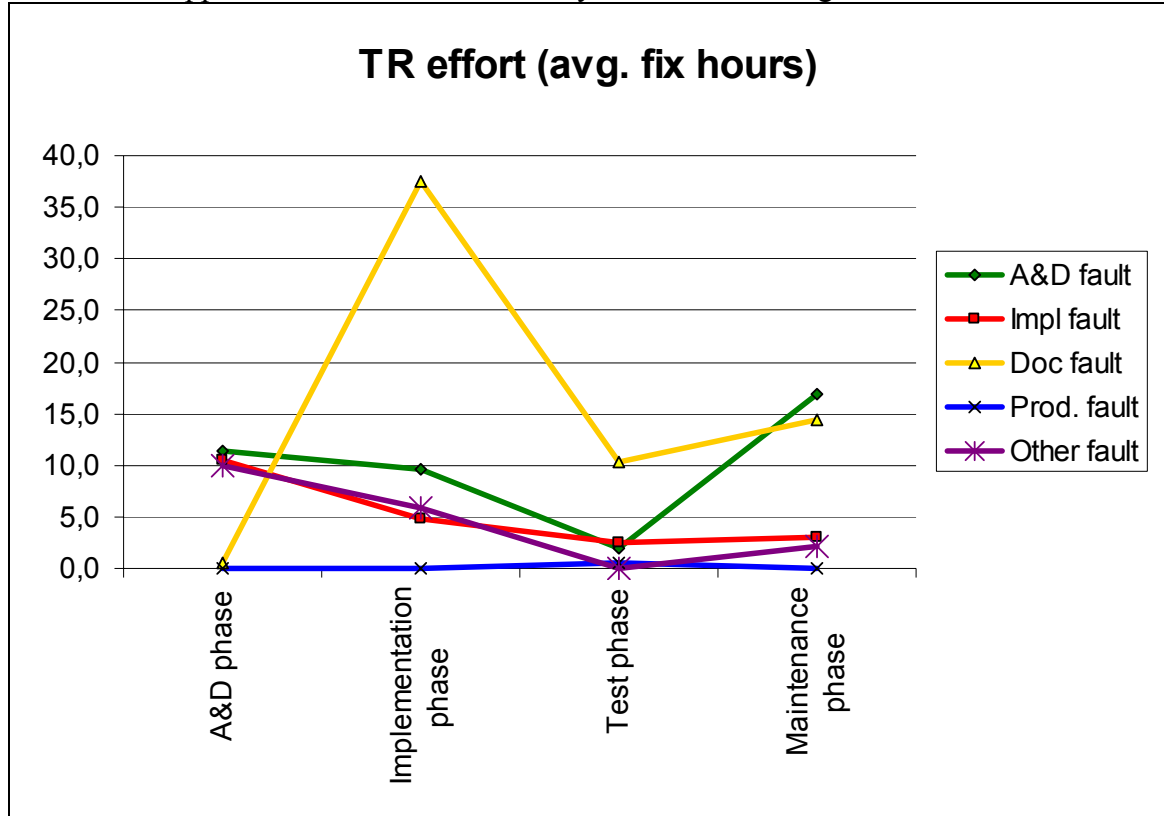


Figure 0-11: Average TR effort

We used data from the SGSN 3.0 only and the results are biased as a result of the lack of ability to differ between faults “created” in a previous version and those created in this version. This also explains why there are implementation faults in the analysis & design phase; those TRs are in fact the result of faults made in previous versions. The general results shown in figure 0-11 are not what one could expect; in fact the time it takes to correct faults made in one phase of the process decline the longer the development process has moved forward. The remainder of this section tries to describe the different ways of explaining this.

The cost of implementation faults only rises about 0,5 hours when moving the project from the test phase to the maintenance phase. The time it takes to correct an analysis & design fault however rise from 1,9 to 16,8 hours when moving from the test phase to the maintenance phase. The small increase for in fix hours for implementation faults might be because the most serious faults might be discovered early. We made an attempt to verify this by checking if the distribution of the severity of the faults is different in the test- and maintenance phase as shown in table 0-21.

(Total 892 TRs)	A	B	C
Test	36%	54%	10%
Maintenance	25%	61%	14%

Table 0-21: Severity of implementation faults for the test- and maintenance phase

The results are indicating that it might be the case that the most serious faults are in fact discovered earlier in the process. When knowing this it is clear that the 0,5 hours rise for implementation faults when moving from the test- to the maintenance phase is not due to an increase in severity but due to other causes. We find similar results for analysis & design as shown in table 0-22.

(Total 217 TRs)	A	B	C
Test	42%	48%	10%
Maintenance	29%	61%	10%

Table 0-22: Severity for A&D faults for the test- and maintenance phase

When moving from implementation- to test phase the hour costs for analysis & design faults drop from 9,6 to 1,9. We cannot apply the same scheme as done above for analysis & design faults when moving from the implementation- to the test phase because there are only registered 12 TRs regarding analysis & design faults in the implementation phase and this does not provide basis to make a statistical comparison as done above.

RH7 – Fault Distribution

Background

Most studies indicate that some components are more fault-prone than others. We try to verify the validity of the claim that 20% of the units produce 80% of all the faults in this hypothesis. The number of TRs is used as an indication on the number of faults.

Hypothesis

H0 (null hypothesis): 20% of all units account for 80% of all TRs

Dataset

In total we have 289 different units registered, but only 128 of them are linked to at least one TR. Of course by using the different numbers we get two different results and we present them both. The number of non-duplicated TRs from SGSN 3.0 is counted for each unit.

In total:

- 128 units (289 units in total)
- 1 953 TRs (several of them marked as being involved with multiple units)

Statistical Method

None used.

Results

Of the 128 units that are linked to at least one TR the top 20% of the units with the most TRs (26 units) accounted for 70,19% of all TRs. (By instead using all 289 units we get that the top 20% / 56 units accounts for 88,50%).

Evaluation and Discussion

Since we might have units that either is obsolete or not developed at Ericsson Grimstad then some of the 289 units should not be counted and included in this analysis. The two results 70,19% and 88,50% defines the interval in which the correct percentage is. We feel that the sample of 128 units provides the most correct value and that 20% of the units accounts for about 70% of all TRs.

Validity

Some of the units might also not be developed at Ericsson Grimstad and therefore not have the correct number of TRs registered. This poses a threat to *internal validity*. We have not identified any threats to *construct-* or *conclusion validity*.

Conclusion

We reject H0: 20% of all units account for about 70% of all TRs.

RH8 – Fault Severity

Background

Each TR is marked with a severity level and this hypothesis concerns the distribution.

Hypothesis

H0 (null hypothesis): Fault severities are evenly distributed

H1: Most faults are marked as being of medium severity

Dataset

As our dataset we used non duplicated TRs from all projects. The TRs had either a severity classification going from 1 to 5 or A to C.

Those TRs with a severity classification going from 1 to 5 were mapped to A to C.

Severity 1 was mapped to severity A, 3 was mapped to B and 5 was mapped to C. There are only two TRs with severity 2 or 4 and these were mapped to severity B.

In total:

- 10 232 TRs

Statistical Method

None used.

Results

The distribution over severity A-C is presented in table 0-23 and figure 0-12.

Severity	#TR	%
A	2557	24,99%
B	5600	54,73%
C	2075	20,28%
SUM	10232	100,00%

Table 0-23: Distribution over severity A-C after mapping

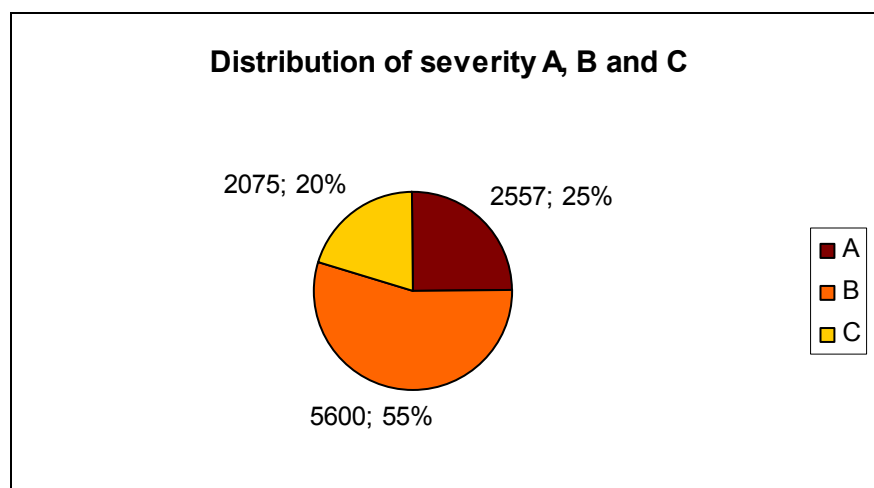


Figure 0-12: Distribution over severity A-C

Evaluation and Discussion

More than half of the TRs are classified as medium severe (severity B) fault not requiring immediate corrections. What could be more interesting would be to combine this

distribution with the information on what type of fault (coding, A&D etc) and in which part of the development process the TRs were discovered. This is done as a part of RH4.

Validity

We have not identified any threats to *internal-*, *construct-* or *conclusion validity*.

Conclusion

We reject H0 in favor of H1: most faults are marked as being of medium severity.

RH9 – Test Phase Effectiveness

Background

Software defects are one of the most important factors affecting the success of a software project. Software inspection has become an essential tool for managing software defects. Inspections reduce cost and increase productivity and the quality of software developed [Sapsomboon00]. The benefits of software inspection are confirmed to be valid across many types of projects and table 0-24 (from [Sapsomboon00]) summarizes some benefits of software inspection reported by the industry.

Reported by	Benefits
IBM [Fagan86]	Reducing rework, increasing net saving 80% of the total defects were uncovered 23% productivity gain (person-hour per non commented KLOC)
Jet Propulsion Lab [Bush90]	Net saving of \$1,700 per defect
Bell-Northern Research [Russel91]	33 hours saving in maintenance effort
Shell Company [Doolan92]	30 hours saving in maintenance effort
Jet Propulsion [Kelly92]	Net saving of 5 – 17 hours per defect
Bull HN Information [Weller93]	70% of defects were uncovered
Hewlett-Packard [Grandy94]	Estimated \$21 million dollars saving per year

Table 0-24: Benefits of software inspection

The amount of faults detected in the test phase is therefore not only crucial for delivering software within user acceptance, but is also important in a cost perspective.

Hypothesis

H0 (null hypothesis): At least 70% of all remaining faults are reported in the test phase

H1: The test phase detects less than 70% of all remaining faults

Dataset

The dataset from RH2 is used.

Statistical Method

None used.

Results

The percentage distribution of when the TRs are reported is shown in table 0-25 (the table on the right is the distribution not counting the faults reported in the earlier phases).

When found	%
<i>Analysis & Design</i>	3,54 %
<i>Implementation</i>	3,49 %
Test	68,48 %
Maintenance	24,48 %
SUM	100 %

Table 0-25: Distribution of reported faults

When found	%
Test	73,67 %

Maintenance	26,33 %
SUM	100 %

Table 0-26: Distribution of reported faults from test phase

24,48% of the TRs are reported so far in the maintenance phase. The number of TRs found in the test phase stands for 68,48% of the TRs in our dataset and more than 75% (3,54%+3,49%+68,48%) of all TRs are reported before entering the maintenance phase.

Evaluation and Discussion

The distribution shown in table 0-25 is bound to change as the maintenance phase is continuing to report faults. However the TRs from the maintenance phase is taken over a 9 ½ month period (as appose to 6 months used in other studies).

To reveal potential improvement areas the same distribution was made for 5 fault types;

- Analysis & Design
- Implementation
- Document
- Deployment
- Other

The distributions for all 5 fault types are listed in appendix **Error! Reference source not found..** There is only one fault type that stands out as being different from the others and that is documentation faults. Less than half of the documentation faults are reported before entering the maintenance phase as shown in table 0-27.

When found	% Doc
Analysis & Design	0,70 %
Implementation	2,10 %
Test	44,76 %
Maintenance	52,45 %
SUM	100 %

Table 0-27: Distribution of reported documentation faults

Validity

We have not identified any threats to *construct-* or *conclusion validity*. The *internal validity* is threatened by our lack of excluding faults made in previous versions.

Conclusion

We cannot reject H0; more than 70% of all faults are reported before entering the maintenance phase.

Maintainability

This subchapter contains three hypotheses regarding maintainability (MH1-MH3).

MH1 – Change Cause

Background

[Votta00] showed that in the system they studied, adaptive changes accounted by 45% of all changes, followed by corrective changes that account for 34%, while perfective changes accounted for 4% of changes. [Weber03] says that although change is part of the daily project life, change proposals occur more often than actual project changes. CRs have a field that indicates the CR type; i.e. new requirement, modified requirement, modified solution, etc.

Hypothesis

H0 (null hypothesis): Most CRs are due to modified requirements.

H1: Most CRs are not due to modified requirements.

Dataset

Each CR was by manual inspection grouped into 6 classifications regarding what kind of change it proposes. The groups used are:

- New requirement
- Modified requirement
- Removed requirement
- Modified documentation
- Modified solution
- Other

Only those CRs that were included in the manual inspection (all but those CRs marked as being an exemption requests) was included.

Statistical Method

None used.

Results

The distribution over CR groups is shown in figure 0-13.

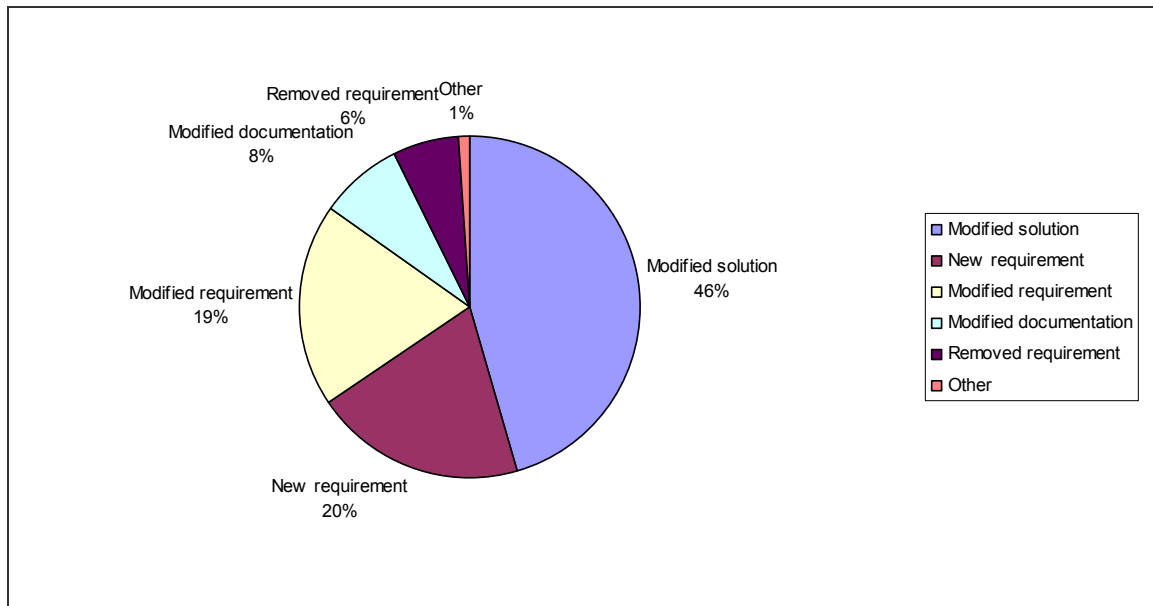


Figure 0-13: Distribution over CR groups

The largest group with 46% is not modified requirement as suggested in our null-hypothesis, but in fact the category modified solution. Modified requirement only accounts for 19% of all CRs. We therefore reject our null-hypothesis.

Validity

We have not identified any threats to *construct-*, *conclusion-* or *internal validity*. These results probably vary a lot from organization to organization as it is dependent on customers, market situation and development process etc posing as an added threat to *external validity*.

Conclusion

We reject H0 in favor of H1; most CRs are not due to modified requirements.

MH2 – Reuse and Change-Proneness

Background

We define *change-proneness* as $\#CR/KLOC$. In this hypothesis we want to check whether means are statistically different for reused components versus non-reused components regarding change-proneness.

Hypotheses

H0 (null hypothesis): Reused and non-reused components are equally change-prone.

H1: Reused components are more change-prone than non-reused components.

H2: Non-reused components are more change-prone than reused components.

Dataset

As *components* we used subsystems. We selected all CRs that affected the SGSN 3.0 system that had registered a specific subsystem. We measured component size as number of *uncommented* lines of code (KLOC). Since it may be difficult to compare different programming languages, we chose to calculate the results using both LOC and ELOC. We have also used MLOC to measure the modified LOC since the previous product version.

We define the following two groups:

- *Reused*: Business specific and middleware components
- *Non-reused*: Application specific components

In total:

- Subsystem level: 12 subsystems (9 reused and 3 non-reused)

Granularity

Subsystem level. The CRs are not registered on block level.

Statistical Method

The dependent variables are number of CRs, and the independent variables are components (subsystems). We then calculated $\#CR/KLOC$, $\#CR/EKLOC$ and $\#CR/MEKLOC$ for each component. We also calculated the average values for reused versus non-reused components. We used a *t-test* to calculate a p-value used as basis for our decision whether to reject our null hypothesis or not.

Results

Table 0-28 shows the results for both the reused and non-reused subsystems. Summary of the t-test results are shown in table 0-29. The value of 22% means that there is a probability of 22% that the results are just coincidental. See appendix **Error! Reference source not found.** for more detailed statistical information and detailed subsystem information.

Subsystems	#CR/ KLOC	#CR/ ELOC	#CR/ MEKLOC	MOD
All	0,3095	0,1138	0,2337	50,6%
Reused (R)	0,2250	0,0856	0,2034	44,5%
Non-reused (NR)	0,5630	0,1986	0,3246	60,3%
NR / R	2,50	2,32	1,60	

Table 0-28: Average #CR/[KLOC, ELOC, MEKLOC] for reused vs. non-reused subsystems

t-Test: Two-Sample Assuming Unequal Variances	
P(T<=t) one-tail [#CR/KLOC]	0,2207
P(T<=t) one-tail [#CR/ELOC]	0,1867
P(T<=t) one-tail [#CR/MEKLOC]	0,2611

Table 0-29: T-test results

Evaluation and Discussion

The results indicate that non-reused subsystems change more often compared to reused subsystems by a factor of 1,6 – 2,5; still we *cannot reject our null hypothesis* based on the calculated p-values using a 95% confidence interval. Nevertheless the results are interesting because the p-values are quite good. We also notice that the difference is less distinct using MEKLOC.

Validity

If one group (i.e. reused components) is dominated by a specific type of components, this could be a confounding factor. If this is the case, then the fact that these components are non-reused may not be the main reason why they change more often, but this can be a result of other circumstances (i.e. different type of logic/complexity etc). For example no reused components in the GSN system have user interfaces. This matter should be investigated further in order to verify this hypothesis. To improve *external validity* it can be useful to perform this analysis on other products and compare the results.

The calculation of LOC values can affect *internal validity*. If there are errors in these values it may influence our results. However, this would only have a negative effect if the error affects one of the groups in particular. We have no evidence that this is the case. We have not identified any threats to *construct-* or *conclusion validity*.

Conclusion

We cannot reject H0; reused components change just as much as other components.

MH3 – Requirement Change

Background

A CR can be in different states indicated in figure 0-5. The CRs are however not updated for every change in its status and only 4 different statuses are used in our dataset. One could assume that most CRs and ERs are accepted and implemented. This hypothesis tests whether or not this is true for this dataset.

Hypothesis

H0: Most CRs and ERs are accepted and implemented.

H1: Most CRs and ERs are not accepted and implemented.

Dataset

All CRs and ERs are used as the dataset. The CRs and ERs are divided into 4 categories according to their reported status. The groups used are:

- Approved
- Cancelled
- Closed
- Rejected

Statistical Method

None used.

Results

Status	%
Accepted/implemented	68,2%
Rejected/postponed	31,8%
SUM	100,00%

Table 0-30: Distribution of mapped statuses

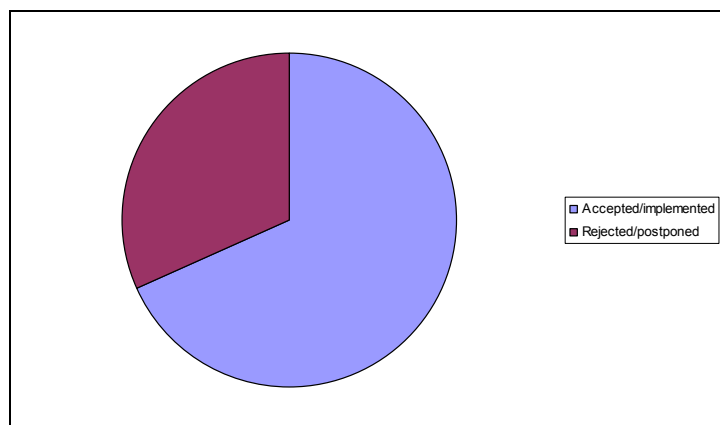


Figure 0-14: Distribution of accepted/implemented and rejected/postponed CRs and ERs

About 68,2% of all CRs and ERs are accepted and implemented. We therefore cannot reject H0; most CRs and ERs are accepted and implemented.

Validity

We have not identified any threats to *construct-*, *conclusion-* or *internal validity*.

Conclusion

We cannot reject H_0 ; most CRs and ERs are accepted and implemented.

Combination of Reliability and Maintainability

CH1 – Relationship Between Reliability and Maintainability

Background

We wanted to examine the relationship between reliability and maintainability by checking the correlation between TRs and CRs.

Hypotheses

H0 (null hypothesis): There is no linear correlation between #CRs and #TRs

H1: There is a linear correlation between #CRs and #TRs

Dataset

We selected all non-duplicate TRs and CRs that affected the SGSN 3.0 system and that had registered a specific subsystem.

In total:

- Subsystem level: 12 subsystems (9 reused and 3 non-reused)

Granularity

Subsystem level.

Statistical method

The dependent variables are number of TRs, and the independent variables are components (subsystems). We counted the numbers of TRs and CRs for each component and used linear regression analysis to check the correlation between TRs and CRs. We have created scatter plots to visualize the results. On the x-axis we have number of TR and on the y-axis we have number of CRs.

Results

A regression summary is shown in table 0-31.

A scatter plot with #CR on the x-axis and #TR on the y-axis is shown in **Error!**

Reference source not found. The figure shows the linear regression line.

Regression summary	
Adjusted R ² :	0,6095
Regression line, P-value:	3,90347E-05

Table 0-31: Regression results

Evaluation and Discussion

Since the adjusted R² is as low as 60%, we *cannot reject our null hypothesis*.

Validity

We have not identified any threats to *internal validity*, *conclusion validity* or *construct validity*.

Conclusion

We cannot reject our null hypothesis. The correlation values are too low.

Summary

All the results from the hypothesis testing are summarized in table 0-32. The following summary is a short description of the results we have reached.

When a component is reused in several projects, detected defects are removed and we may assume that reliability grows, although the component is modified. A survey performed with some developers at Ericsson presented in [Mohagheghi03a] showed that the developers believe that reused components are more reliable. We performed an analysis at both subsystem and block level (see 0). Our results indicate that reused components in fact are more reliable. We also wanted to study whether larger components are more fault-prone than smaller ones (see 0). Intuitively this seems like a valid claim, but some studies show that larger modules are in fact less fault-prone and have better quality [Basili84a]. However we could not find enough statistical evidence to support these other studies. To possibly identify if our results were different we investigated if reused components change more often or vice versa (see 0), but we could not find any difference in how change-prone reused components are versus other components.

[Defamie99] concludes that only about 55% of all TRs are corrected within a few days.

We perform an identical analysis (see 0) and conclude that only about 25% of all TRs are corrected within a few days (5 days). We have also investigated how many TRs that are related to coding as oppose to other fault categories for instance analysis & design (see 0). Each TR is also marked with a severity level and we investigated the distribution of these severity levels. These two analyses give an overview of the nature of the TRs reported. The conclusion was that about 63% of all TRs are related to coding and that TRs of medium severity accounted for almost 55% of the TRs (see 0). We also calculated an estimation of the effort needed to correct a specific type of fault in a particular phase in the development process (as a subsection of 0). The results were however strongly biased by the fact that we could not distinguish new faults from the faults made in previous version and thus got for instance implementation faults in the analysis & design phase.

By grouping the TRs into different categories regarding the type of fault (fault within a component, faults introduced when integrating components, RM/analysis & design faults and others) we could get an overview of the properties of the corrective maintenance performed in SGSN (see 0). The biggest category is actually those concerning RM/analysis & design.

[Sapsomboon00] present several studies showing the costs saved by identifying and correcting faults before entering the maintenance phase. The amount of faults detected in the test phase is therefore not only crucial for delivering software within user acceptance, but is also important in a cost perspective. We have found that in our dataset more than 70% of all faults are reported before entering the maintenance phase (see 0).

Pareto's Law is the economic theory that 20% of the population earns 80% of the income. Pareto's Law has proven to be valid in many other areas and we wish to check if it could be applied for: 20% of the units produce 80% of the faults (see 0). We conclude that it is nearly true, but the 20% most fault reporting units produces only about 70% of all faults.

There are studies on the effect the actual programming language has on defect density; an example is given in [Fenton97, p.499]. We checked how fault-prone the units programmed in Erlang were and compared this with the equivalent result for C (see 0).

By using plain number of LOC we get that Erlang is more fault-prone. However this result could change when using ELOC to identify any differences in fault-proneness. The

results are so tightly connected with identifying the correct equivalent factors (EF; see 0) that we make any decision about the fault-proneness regarding the different programming languages Erlang and C.

[Votta00] discusses and present different categories of CRs and the distribution of these categories. We have performed a similar research (see 0) and conclude that most CRs are not due to modified requirements as in [Votta00], but due to a request for a modified solution. There are about just as many CRs related to new requirements as to modified requirements.

Finally we have studied how many of the CRs and ERs that actually are accepted (see 0) and conclude that most of them are accepted.

Hypothesis	Results
Quality attribute: Reliability	
RH1: Reused components have the same reliability as other components	We reject H0. The results indicate that reused components are more reliable
RH2: Larger components are not more fault-prone	We cannot reject H0. The correlation values are too low; larger components are not more fault-prone
RH3: The fault categories are evenly distributed	We cannot reject H0; the fault types are evenly distributed
RH4: There is no difference in defect density between units programmed in Erlang and units programmed in C	We cannot reject H0; there is no difference in defect density between Erlang-units and C-units
RH5: Most faults are corrected at once	We reject H0 in favor of H1: most faults are not corrected at once
RH6: There are just as many TRs related to coding than to other faults	We reject H0 in favor of H1: most faults are related to implementation
RH7: 20% of all units account for 80% of all TRs	We cannot reject H0: 20% of all units account for only 70% of all TRs
RH8: Most faults are marked as being of medium severity	We reject H0 in favor of H1: most faults are marked as being of medium severity
RH9: At least 70% of all remaining faults are reported in the test phase	We cannot reject H0; more than 70% of all faults are reported before entering the maintenance phase
Quality attribute: Maintainability	
MH1: Most CRs are due to modified requirements	We reject H0 in favor of H1; most CRs are not due to modified requirements
MH2: Reused components change just as much as other components	We cannot reject H0; reused components change just as much as other components
MH3: Most CRs are accepted and implemented	We cannot reject H0: most CRs are accepted and implemented
Combination of Reliability and Maintainability	
CH1: There is no linear correlation between #CR and #TR.	We cannot reject H0. The correlation values are too low and we have not found enough evidence to support a linear correlation

Table 0-32: Summary of results from the hypothesis testing

PROPOSED IMPROVEMENTS AT ERICSSON

Most of our hypotheses are just verification/testing of statements found in other publications and hypotheses with interest to Ericsson. Our thesis does not have as a goal to identify improvement areas for Ericsson, but during the process of analyzing data and statistical analysis we did identify some potential improvement areas.

Process

In our dataset, less than half of all documentation faults were detected before entering the maintenance phase. Herein lays a great potential for improvement. By identifying problems in their current process for checking the documentation in the test phase, Ericsson might be able to detect almost twice as many faults as they do today (from ~40% to the ~70% for the other fault categories. See 0 for more details).

Data Collection

Trouble Reports

In our work with the data available at Ericsson we discovered both unnecessary data and data that should have been collected. Our lack of possibility to differ between faults created in earlier versions and reported in later ones biased the results in most of our hypotheses, but especially for hypothesis RH6 (0) would this information yield more reliable data.

Change Requests

The CRs is now written in a template in Microsoft Word, some of the CRs in our dataset was even written in Framemaker or Microsoft Word documents not following the template standard. This made it impossible to do a data analysis without manual inspection (what was we did as described in subchapter 0). Ericsson should implement a similar system for CRs as the one for TRs. The CRs could then be registered via a web interface so that the data could be analyzed directly without a manual data extraction/analysis. The .NET solution we used in our process could be used as a basis.

General

The quality of the entered data was also varying greatly thus undermining the quality of statistical analysis. By removing the possibility to select “Unknown” where possible, one could force the TR/CR/ER submitter to enter more usable data. Of course sometimes the option to choose “unknown” must be available and a better solution would be to encourage the TR/CR/ER submitters to put extra effort into submitting the most correct information possible. One other problem we encountered during the Data Acquisition and Hypothesis Selection process (see figure 0-6) was that some data had been entered in multiple ways. This is the result of allowing the TR submitter to use free text input for data that should be selected from a list box or similar.

Summary

- Analyze the improvement areas for revealing more documentation faults in the test phase
- Add a field for TRs: from which version does this fault originate?
- Implement a web interface for the submitting of CRs
- Where possible remove free text input
- Where possible remove options like “unknown”

CONCLUSION AND FURTHER WORK

In this thesis we have analyzed 13 hypotheses regarding reliability and maintainability for the GSN system at Ericsson. We have found some interesting results regarding reuse and reliability; our analysis concludes that reused components are about twice as reliable as non-reused components. Our results also indicate a certain correlation between fault-proneness and size of components. We found that the usage of different programming languages has no special impact on defect density for components. Our data also indicate that reused components are more change-prone; however we do not have a good significance to conclude this. We recommend Ericsson to investigate the results further, and we propose some improvements regarding measurement and data collection.

We need advanced theories in order to further generalize our empirical observations. The theories must take into account information from the development process, problem complexity, design complexity etc. This is needed to fully understand cause and effect relations. It is important to include the causal effects of programmers and designers. After all it is they who introduce the faults; hence any attribution for faulty implementation must finally rest with individuals.

We have not analyzed all the data in our database. For further work we propose that additional hypotheses should be defined and verified using the existing data material. This can be done without much effort other than constructing the correct SQL queries and performing statistical analysis. The hypotheses can also be combined to perform multivariate analysis to identify more complex correlations.

By including additional data from Ericsson or other organizations one could retest our hypotheses or any new ones. We also encourage Ericsson to further investigate why some parts of the system are more fault-prone than others.

We propose the collection of the following new data as basis for further research:

- Use source code files as a basis to calculate complexity and retest the hypotheses regarding LOC. For instance one could use the exported IDL interfaces and calculate the number of functions it contains. ([Fenton99, p.3] discusses the use of function points instead of LOC as it is a better one dimensional size metric than LOC).
- Use the information in Application Requirement Specification (ARS) and test hypotheses on estimation precision.
- Include data from other organizations to provide (better) external validity.

REFERENCES

- [Atkinson02] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst and J. Zettel: "Component-Based Product Line Engineering with UML", Fraunhofer IESE, 2002.
- [Atkinson02a] C. Atkinson and D. Muthig: "Enhancing Component Reusability through Product Line Technology", Fraunhofer IESE, Springer-Verlag Berlin Heidelberg, 2002.
- [Bachmann00] F. Bachmann et.al: "Volume II: Technical concepts of Component-based Software Engineering". SEI technical report number CMU/SEI-2000-TR-008, 2000.
- [Barbacci97] M.R. Barbacci, M.H. Klein and Charles B. Weinstock: "Principles for Evaluating the Quality Attributes of a Software Architecture", Technical Report, CMU/SEI-96-TR-036, ESC-TR-96-136, 1997.
- [Basili84] V.R. Basili and D. Weiss: "A Methodology for Collecting Valid Software Engineering Data", IEEE Trans. Software Eng., 10(11), pp. 758-773, November 1984.
- [Basili84a] V.R. Basili and B.T. Perricone: "Software Errors and Complexity: An Empirical Investigation". Communications of the ACM, 27(1), pp. 42-52, 1984.
- [Baumer97] D. Baumer, G. Gryczan, et al.: "Framework development for large systems", Comm. ACM 40(10), 52-9, 1997.
- [Boehm99] B. Boehm and C. Abts: "COTS Integration: Plug and Pray?", IEEE Computer, pp. 135-138, January 1999.
- [Bosch00] J. Bosch: "Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach", Addison-Wesley, 2000.
- [Briand02] L. Briand and J. Wuest: "Empirical Studies of Quality Models in Object-Oriented Systems", Advances in Computers, Academic Press, 2002.
- [Briand02a] L. Briand, S. Morasca and V. Basili: "An Operational Process for Goal-Driven Definition of Measures". Forthcoming in IEEE Trans. on Software Engineering, 2002.
- [Bush90] M. Bush: "The use of formal inspections at the Jet Propulsion Laboratory". Proceeding of the 12th International Conference on Software Engineering (ICSE'90). Los Alamitos, CA: IEEE CS Press, 196-9, 1990.

-
- [Carney97] D.J. Carney and P.A. Oberndorf: "The Commandments of COTS: Still in Search of the Promised Land", Crosstalk, Vol. 10, No. 5, 25-30, May 1997.
- [Campbell63] D.T. Campbell and J.C. Stanley: "Experimental and Quasi-Experimental Designs for Research", Houghton Mifflin Company, Boston, 1963.
- [Coleman94] D. Coleman, D.Ash, B. Lowther and P. Oman: "Using Metrics to Evaluate Software System Maintainability", IEEE, 1994.
- [Cook79] T.D. Cook and D.T. Campbell: "Quasi-Experimentation – Design and Analysis for Field Settings", Houghton Mifflin Company, 1979.
- [Defamie99] M. Defamie, P. Jacobs and J. Thollembeck: "Software Reliability: Assumptions, Realities and Data". Proceedings of the International Conference on Software maintenance (ICSM99), 1999.
- [Doolan92] E. Doolan: "Experience with Fagan's Inspection Method. Software Practice and Experience" 173-82 22(2), 1992.
- [Doug03] The Great Computer Language Shootout, May 10, 2003.
<http://www.bagley.org/~doug/shootout>
- [Fagan86] M. Fagan: "Advances in software inspections". IEEE Transactions on Software Engineering 12(7), pp. 744-751, 1986.
- [Fenton97] N. Fenton and S.L. Pfleeger: "Software metrics: A Rigorous & Practical Approach". 2nd edition, International Thomson Computer Press, 1997.
- [Fenton99] N. Fenton and M. Neil: "A Critique of Software Defect Prediction Models". IEEE Transactions on Software Engineering, Vol. 25, No. 3, May/June 1999.
- [Gamma95] E. Gamma, R. Johnson, R. Helm and J. Vlissides: "Design Patterns – Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.
- [Grandy94] R. Grandy and T. van Slack: "Key lessons in achieving widespread inspection use", pp. 46-57, IEEE Software 11(4), 1994.
- [GSNRUP02] GSN RUP Adaptation, release R4A, Ericsson Intranet (not public), 2002.
- [IEEE93] IEEE Std. 610.12-1990, 'Glossary of Software Engineering Terminology,' in Software Engineering Standards Collection, IEEE CS Press, Los Alamitos, Calif., Order No. 1048-06T, 1993.
- [Karlsson95] Even-André Karlsson (Ed.): "Software Reuse: A Holistic Approach" (The REBOOT Methodology Handbook), Wiley Series in Software Based Systems. John Wiley. 510 p. REBOOT report no. 8218, 1995.
- [Kelly92] J. Kelly, J. Sherif and J. Hops: "An analysis of defect densities found during software inspections". Journal of Systems and Software 17(2), 111-17, 1992.
-

-
- [Kruchten00] P. Kruchten: "The Rational Unified Process, An Introduction", Addison Wesley, 2000.
- [Laprie01] A. Avizienis, J. C. Laprie and B. Randell: "Fundamental Concepts of Dependability", 2001.
- [Mohagheghi03] P. Mohagheghi: "A Study of Quality Attributes for a Component-based Telecommunication System", PhD thesis, preliminary version, 2003.
- [Mohagheghi03a] P. Mohagheghi, R. Conradi, E. Naalsund and O. A. Walseth: "Reuse in Theory and Practice: A Survey of Developer Attitudes at Ericsson", 2003.
- [Morisio00] M. Morisio, M. Ezran and C. Tully: "Investigating and Improving a COTS-Based Software Development Process", Proc. 22nd International Conference on Software Engineering (ICSE'2000), ACM Order No. 592000, pp. 31-40, 2000.
- [Orfali98] R. Orfali and D. Harkey: "Client/server programming with Java and CORBA", John Wiley and Sons, 1998.
- [Parikh83] G. Parikh and N. Zvegintzov, Tutorial on Software Maintenance, IEEE CS Press, Los Alamitos, Calif., Order No. 453, 1983.
- [Poulin01] J.S. Poulin: "Measurement and Metrics for Software Components", 2001.
- [Poulin95] J. S. Poulin: "Populating Software Repositories: Incentives and Domain-Specific Software", Journal of System and Software 1995, pp. 187-199.
- [Russel91] G. W. Russel: "Experience with inspection in ultralarge-scale developments". IEEE Software 8(1), 25-31, 1991.
- [Sapsomboon00] B. Sapsomboon: "Shared Defect Detection : The Effects of Annotations in Asynchronous Software Inspection", 2000.
- [Schwarz02] H. Schwarz, O.M. Killi and S.R. Skånhaug: "A Study of Industrial, Component Based-Development, Ericsson", pre-diploma thesis, NTNU, 105 p., 2002.
- [Schmidt97] D.C. Schmidt: "Applying design patterns and frameworks to develop object-oriented communications software", Macmillan Computer Publishing, 1997.
- [Sjøberg00] D. Sjøberg and R. Conradi: "Incremental and component-based software development (INCO)", 2000.
- [Sommerville01] I. Sommerville: "Software Engineering", Addison-Wesley, 2001.
- [Votta00] A. Mockus and L.G. Votta: "Identifying Reasons for Software Changes Using Historic Databases", Proceedings of the International Conference on Software Maintenance (ICSM00), 2000.
- [Weber03] M. Weber and J. Weisbrod: "Requirements Engineering in Automotive Development: Experiences and Challenges", IEEE Software, pp. 16-24, Jan-Feb 2003.
- [Weller93] E. Weller: "Lessons from three years of inspection data". IEEE Software 10(5) 1993, pp. 38-45.
-

- [Wohlin00] C. Wohlin, P. Runeson, M. Host, M.C. Ohlsson, B. Regnell and A. Wesslen: “Experimentation in Software Engineering”, Kluwer Academic Publications, 2000.
- [Wang02] M. Torchiano, L. Jaccheri, C.-F. Sørensen and A.I. Wang: “COTS Products Characterization”, *SEKE '02*, July 15-19, 2002, Ischia, Italy, 4p.
- [Wu00] Y. Wu and M.H. Chen: “Techniques for Maintaining Evolving Component-Based Software”, Proceedings of the International Conference on Software Maintenance (ICSM00), 2000.

ABBREVIATIONS

ARS	Application Requirement Specification
CBD	Component-based Development
CBSE	Component Based Software Engineering
CR	Change Request
ELOC	Equivalent LOC
EF	Equivalent Factor (used to calculate C Equivalent LOC)
GSN	GPRS Support Node
GPRS	General packet Radio Service
GSM	Global System for Mobile Communications
GUI	Graphical User Interface
ISO	International Standard Organization
KLOC	1000 (Kilo) Lines Of Code
LOC	Lines Of Code
MDD	Measurement data Definition
ME(K)LOC	Modified Equivalent (K)LOC
M(K)LOC	Modified (K)LOC (includes added and modified (K)LOC)
MRD	Measurement Result Definition
OO	Object-Oriented
OTP	Open Telecom Platform
OS	Operating System
QoS	Quality of Service
RFA	Ready For Acceptance

RUP	Rational Unified Process
SGSN	Serving GPRS Support Node
TG2	Toll Gate 2
TR	Trouble Report
UMTS	Universal Mobile Telephone System
VM	Virtual Machine
WCDMA	Wide Band Code Division Multiple Access
WPP	Wireless Packet Platform is the platform used for GPRS applications.

APPENDIX (CONFIDENTIAL)

The appendix is excluded from the public version.