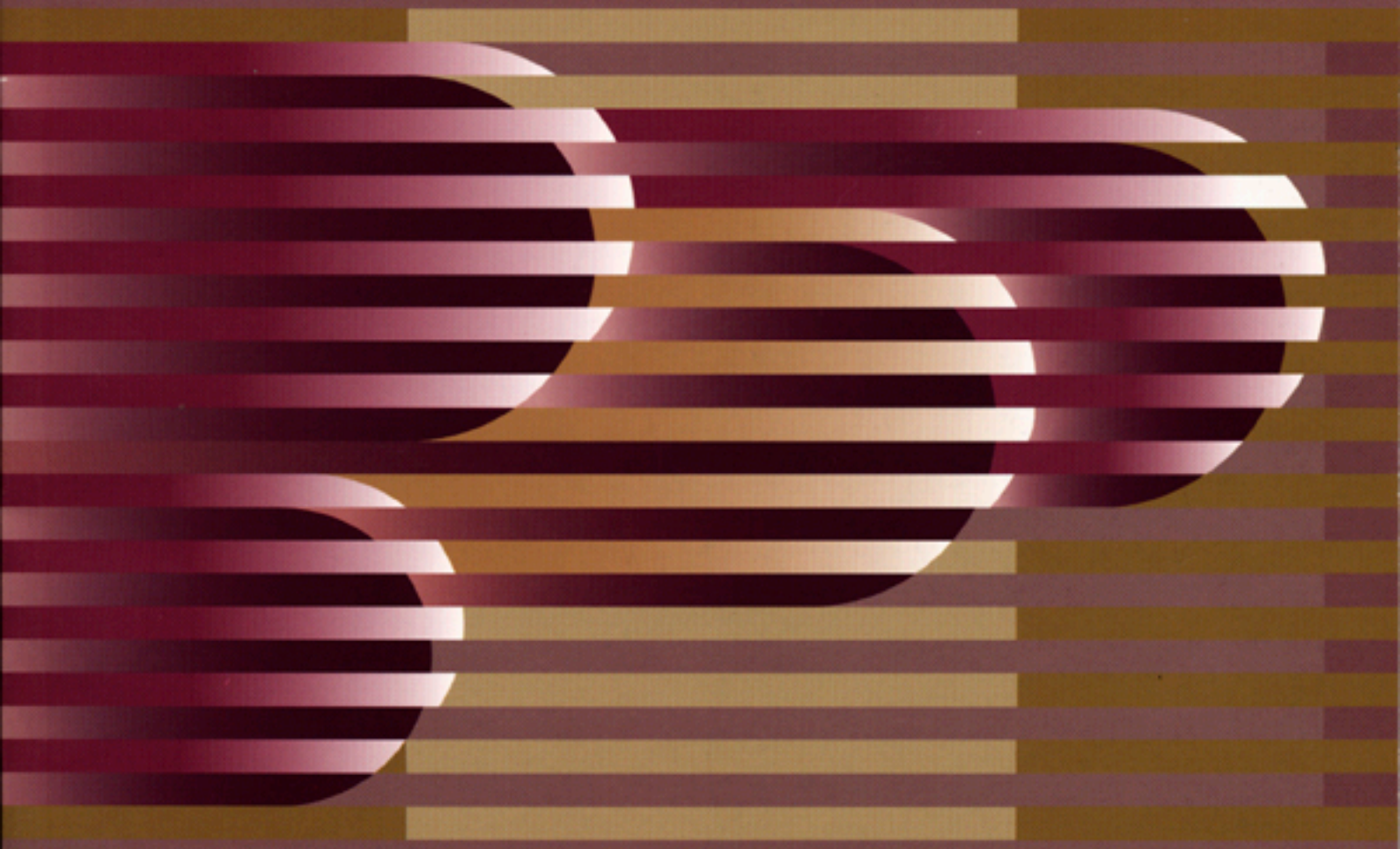
Telektronikk 1.99

95 YEARS ANNIVERSARY

Software Quality in telecommunications



Contents

Feature

Special

Status

Telektronikk

Volume 95 No. 1 – 1999
ISSN 0085-7130

Editor:
Ola Espvik
Tel: (+ 47) 63 84 88 83

Status section editor:
Per Hjalmar Lehne
Tel: (+ 47) 63 84 88 26

Editorial assistant:
Gunhild Luke
Tel: (+ 47) 63 84 86 52

Editorial office:
Telektronikk
Telenor AS, Telenor Research & Development
P.O. Box 83
N-2027 Kjeller, Norway
Tel: (+ 47) 63 84 84 00
Fax: (+ 47) 63 81 00 76
e-mail: telektronikk@fou.telenor.no

Editorial board:
Ole P Håkonsen, Senior Executive Vice President
Oddvar Hesjedal, Vice President, Research & Development
Bjørn Løken, Director

Graphic design:
Design Consult AS

Layout and illustrations:
Gunhild Luke, Britt Kjus, Åse Aardal
Telenor Research & Development

Guest editorial

ØYSTEIN SKOGSTAD

Computers are used in a wide variety of application areas. Their correct operation is critical for business success and even for human safety. Developing or selecting high quality software products is therefore of prime importance. When dealing with software quality, there are two basic problem areas: What specific actions should be made to ensure software quality, as seen from the user's point of view? How can we in an objective way measure what degree (level) of quality that has been achieved? Through the various papers in this issue of *Telektronikk* we look into some topics discussed in the software community today, in relation to the two problem areas. Comprehensive specification and evaluation of a software product is another key factor in ensuring adequate quality. This can be achieved by defining appropriate quality characteristics, taking into account the purpose of a software product.

A definition of software quality must bear upon the general definition of quality: *The totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs.* For the definition to be of any use, we must agree upon how quality is measured. To be able to measure software product quality we must have a framework for what to measure. Such a framework is often called a software quality model – also enabling us to measure software quality at early stages in the development process. The ISO/IEC quality model is presented in one of the papers to follow. Another paper argues that measurement theory establishes such a framework.

It is believable that the number of errors made in a software program increases with the length of the program. The number of errors for a given application can be kept low by minimising the program code developed for that application and reusing generic program code that is tested through extensive use of other applications. A theory of software quality by code reduction requires a measure of program length or size relative to the functionality provided by the program, a means to compare the lengths of two programs providing the same functionality and techniques to remove application specific code. One paper provides a possible approach to this problem.

The benefits of software reuse seem obvious. Building from pre-built components means less work to develop a particular application, leading to shorter time to market, fewer errors and greater cost reductions. Extensive reuse is still not common practice in the software industry, but over the last years we have seen it successfully demonstrated on several occasions.

The software industry can improve only by understanding their products and the processes that produce them. We need measurements in software development to improve the development process so that we can increase product quality and thus increase customer satisfaction. How can we in a systematic way learn from the errors that are made during development, and improve the situation in the next development? One popular way of starting a Software Process Improvement (SPI) program is to do an assessment.

Experimentation on a company wide scale is often necessary to understand the company's specific needs for Software Process Improvement activities. Results from such experiments are reported from the companies – Telenor and from NERA. One paper describes how Telenor developed and implemented processes, roles and tools to achieve reuse experience, i.e. organisational learning. Another study reports on the reuse of software development experience. The challenges were: How can software development experience be efficiently shared between different development teams? What types of experience are worth reusing? What is the role of reuse of 'local' (context-dependent) experience compared with more 'global' (best practice) experience?

There are few empirical studies on how CASE tools impact the software development and maintenance efficiency. One paper applies a method to evaluate four different CASE tools. Interesting findings were that CASE tools had a strong and systematic impact on the development and maintenance efficiency, i.e. the choice of CASE tool is important.

NERA decided at the end of 1996 to start an improvement project. They started making a generic GQM plan and measurement plan for software development. Data were collected, and some metrics were adjusted to make the definitions more appropriate. The findings for measurements on the effectiveness of verification and validation are presented in this issue. It should be noted that the system devised for recording data made the cost of this metrics collection insignificant.

Testing is essential for achieving quality software. However good the development process, there is still a need to adapt the system to overall needs. A pursuit of quality can improve the technical characteristics of a software product. But quality is not the only framework for making strategic market decision. One paper discusses various test strategies, to optimise on quality as well as on the other factors considered important at company strategic level.

A very important basic to providing good quality of service to the customers is the quality of the telecommunication products that a PNO (Public Network Operator) buys from its suppliers. In order to be able to assess the quality of such telecommunication products, a measurement system has to be in place. Such a system should provide a general overview of product quality

and reliability, together with supplier process quality. A paper presents the two quality measurement systems implemented in Belgacom for a particular telecommunication product. The first system contains in-process quality metrics that evaluate the processes of the supplier mainly during the development phase of the product. The second system contains measurements that analyse the quality and reliability during the operational phase. The paper also focuses on the role of EIRUS during the implementation of these quality measurement systems. The work done in EIRUS is a very interesting case where a special interest group of rather large companies have formed a common forum for parts of their systematic quality assurance work.

Quality measurement is one aspect of interaction between a supplier developing a software system and his customer. Other aspects of the general technical interfaces between supplier and customer are treated in a paper presenting results from a EURESCOM project.

Our knowledge of how to handle software quality is substantial, although far from complete. It is still a problem that important software is developed and released without the expected level of quality. A bigger problem is, however, that software production too often misses utilizing the knowledge we do have about quality issues. Even so, by reading the papers of this *Telektronikk* feature section, I hope you get a fair impression of the complexities of a technology that in our time has emerged to have an impact on the lives of every one of us.

Software quality

ØYSTEIN SKOGSTAD

1 An Introduction to Software Quality

Software quality is a topic of interest for both software developers and software users. Over the years, the focus of software quality has shifted. At least three stages can be identified:

1. "The first ages". During the first uses of software, the problems of software quality were felt by developers, and professional users (the batch age).
2. "How do we do it?" The problems of software quality very soon focused down to the quest for methods for developing quality software.
3. "How do we measure quality?" With the casual software users we have today, a need emerged to have a method for controlled quality in the dissemination of new products. This need made objective software quality measurements necessary.

1.1 The first ages

Over the years, several paradigms have blended to constitute what we today think of as software quality. During this process, the term 'software quality' has been given increasingly more precise definitions. In the beginning it was used more like a slogan, with no precise meaning. A good example of this is the well-known citation: *"You can't test quality into a piece of software, you have to build it in."* The origin of this slogan can be found in a conference sponsored by the NATO Science Committee in 1968 [1]. There it is stated:

> *"If you have your production group, it must produce something, but the thing to be produced has to be correct, ... I am convinced that the quality of the product can never be established afterwards. Whether the correctness of a piece of software can be guaranteed or not depends greatly on the structure of the thing made. This means that the ability to convince users, or yourself, that the product is good, is closely intertwined with the design process itself."*

This often-cited statement from Dijkstra emerged in a discussion about the need to distinguish between design and production of software. The discovery of the difference between design and production was one of the first achievements in the understanding of software quality. Other important topics were:

- **Programming concepts**. Especially structured programming [2] and the concept of software module, which was introduced in the early phases of building a framework for software quality [3]. These concepts are today 'common knowledge' for the skilled programmer.
- **Methods for achieving robustness in software**. This concept covers techniques as different as self-checking and fault-tolerant software. [4] presents early work in this area.
- **Testing**. The concept of testing includes system testing – as done by the developing party, as well as acceptance testing – as done on behalf of the product receiving party. In addition, testing also covers module testing (or debugging) as done by the (module) producer, as described in [5].

1.2 How do we do it?

Over the years, attention shifted into this issue: *How* do we build quality into a software system? The answers to the question on how to build good quality software are numerous. These answers have been topics for discussion in the software community for years. An outline of the origins of software quality would cover most of the research topics that have been dealt with in the software community over the last 30 years. Among the most prominent topics are:

- **Structured software design**. An early notion of this concept may be found in [6]. A wide variety of software design methods has emerged over the years. An overview of some of the classical methods may be found in [7]. Among the most known classical methods we can mention SASD (Structured Analysis and Structured Design (Yourdon)), SSA (Structured Systems Analysis (Gane&Sarson)) and SADT (Structured Analysis and Design Technique). Over the years, much work has also been performed to establish complete methodologies, covering all aspects of software development. Today, software engineers use various object-oriented methodologies. One commonly used technique is OMT – the Object Modelling Technique, see [8].
- **Data Structuring**. A key element of software design concerns structuring of the data the program is working upon. Several techniques have been devised, from the structuring of data records, see e.g. [9], via entity-relationship modelling [10] to today's object oriented techniques.
- **Software engineering**. Software engineering as a term seems to have come to light at the NATO conference in 1968 [1]. The term may be defined as: *"The application of science and mathematics by which the capabilities of a computer equipment are made useful to man via computer programs, procedures and associated documentation"* [11]. Software engineering as a topic today is an umbrella topic that covers all aspects of phased, or milestone based software development work. Included in the concept are of course many ideas from project management, and quality assurance. Included is also 'the art' of establishing work units for the various people engaged in a development project – often known as a Work Breakdown Structure.
- **Methods for specification of requirements**, see e.g. [12]. A more general idea of the concept of specifications can be found in [13]. Every skilled practitioner today deems a specification necessary. Without a requirements specification and some knowledge of what are the needs of the software user, there is little hope of achieving quality today. Elegance in programming may be achieved, but that does not imply quality.
- **Formal methods for software specification**. The representation of software in its early stages by some mathematical formalism allows more rigid verification of the software [14] as well as automated code generation. The hopes brought forward by the formal methods are that once the specification is made and agreed upon, then the programmer in his work will make fewer errors with the programming of the system.
- **Verification and validation**. V&V is a most popular topic that blends the topic of testing (of executable code) with the topic of reviewing or inspecting design specifications and other documents. The IEEE standard [15] has been heavily used in the software community.
- **Configuration Management**. CM is a recognised set of methods to assure that people are working towards the same goal, that changes are handled in a systematic way, and that the right

version of a program is delivered to the users. The IEEE standard [16] has set the standard in this area.

- **Human-machine relation** (or mmi). This area has a direct impact on how the user will perceive the system. With more widespread use of software in applications directed towards the casual user, as well as the use of software in more complex operations, the user (or operator) interface is important. Notions of methods for specification of this interface can be found in e.g. [17].
- **Human-human relation**. Working together on something as abstract as software, and often in large projects, may cause huge challenges to the ability of people to co-operate. In addition to good system architecture and an elaborate work breakdown structure people need to interact on the same piece of software. This interaction includes observing the software from different views; e.g. management view, quality assurance view, and programmer peer view. Commenting on the software unit (product) at hand has been shown to be a most efficient way to improve product quality. Common techniques for this include technical reviews, walkthroughs and inspections, see [18] and [19].
- **Cost-time estimation**. Cost has, at least intuitively, been considered as an element of software quality. Thus, techniques for cost estimation and cost control have been discussed among people concerned with software quality for some years, see [1]. Associated with the cost problem has always been the time problem: How long will the development of this software really last? One method in this area that has achieved wide use is COCOMO [11].

1.3 How do we measure quality?

After doing the very best in producing good quality software, the need emerged to find some objective way of measuring the achievement. This need led to experimentation in the area of quality measurements. As a consequence the following research topics emerged:

- **Measurements of software quality**, often called software metrics. The basic metrics try to identify the properties of software structure that in some way relate to quality. The best known origins for this concept are [20] and [21]. The basis for the more modern concepts can be found in [22].
- **Quality improvement**. The term quality improvement is used today as a collection of techniques for systematic improvement of products and processes. Often these techniques are based on the use of measurements as a means to understand the current situation, as well as a means to check whether improvement goals are reached. In the software community, this topic is often highlighted as Software Process Improvement [23].
- **The concept of quality in general**. Software quality is but a special case of quality. The concept of quality has been around for some decades. It started with industrialisation and mass production around 1900. Over the years, society has focused upon quality control. Popular topics today are quality assurance and quality improvement techniques. The precise meaning of quality management and quality assurance can be found in [24]. Once these concepts were invented and considered useful also for development work, it was natural to start talking about software quality assurance. The ideas of software quality assurance are presented in [25].

Once the trend towards standardisation of software quality aspects had started, it was only natural to start standardising the way software quality should be measured.

In this paper, we shall look at the status of this standardisation work, including the development of a software quality model as well as concrete metrics that make it possible to objectively measure the quality of a given piece of software.

1.4 Quality factors

Over the years, the notion of different *factors* as the constituents of software quality has grown. In the start, *reliability* was often considered the sole factor when people were discussing software quality. One of the reasons for this may be the fact that many of the people engaged in the discussions on software quality were in their everyday work concerned with the production of (or at least the use of) computer operating systems. Many of the early terms and notions of software quality clearly come from the operating system business. Problems and questions relating more directly to the human use of the software (as they can be found today) evolved later. One should remember that in these early days, even the administrative data processing systems were to a large extent batch systems, doing their work overnight, with (hopefully) no human intervention. The infamous two hours downtime in 40 years (from the production of #1 ESS as mentioned in [1]) is one of the first events where the quantitative factor reliability is attributed to software.

Over the years one might say that the software community has had a reasonable success in defining methods that may improve the quality of a software system. One question does, however, remain to be solved:

> How should software quality really be measured, in order to gain an objective evidence of to what degree a software system really is a quality product?

The remainder of this paper will give some answers to that problem, through an overview of the some of the relevant standardisation work.

2 Definition

In the following pages we shall concentrate upon the modern definition and metrication of the concept of *software product quality*.

2.1 Definition of software quality

A definition of software quality must bear upon the general definition of quality [26]:

> *Quality: The totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs.*

At this point, the following observations should be made:

- A requirements specification of some form is necessary, to have a description of the stated needs.
- The implied needs are often manifested by the software system user, through the use of the system.

Thus it can be observed that the needs of the user should be described in the requirements specification, to avoid bad surprises when a new system is taken into use.

For the definition to be of any use, we must agree upon how quality is to be measured. To be able to measure software product quality we must have a framework for what is to be measured. Such a framework is often called a software quality model. The model needs to take into account that we want to measure software quality as early as possible in the development process. The model must also help us perform standardised, or commonly acceptable, measurements of quality for software in use.

2.2 A software quality model

A quality model describes a set of quality characteristics that characterise the product and form the basis for the product evaluation. Over the years, several quality models have been launched, e.g. [21] and [22]. Today, a standardised model exists, ISO/IEC 9126 [27]. This standard is now under revision, in a joint effort by IEC and ISO. ISO/IEC 9126 [28] defines the quality *characteristics* as well as *subcharacteristics* for software products. Quality is categorised as a set of software quality *attributes.* The attributes are divided into six characteristics, which are further sub-divided into subcharacteristics (see the following sections for details). One important aspect of defining a software quality model, is to include a framework for the measurement of quality. Measurable subcharacteristics of software quality are referred to as software quality *metrics.*

The specification of a software quality model must meet many goals, due to the many situations where software quality is an issue: development, acquisition, quality assurance, independent evaluation, use, support, and maintenance. The characteristics defined are meant to be applicable to all types of roles and every kind of software, including computer programs and data contained in firmware. The quality model can be used to:

- Identify software requirements. In the requirement for a specific attribute, care should be taken to define how this particular attribute is to be measured.
- Validate the completeness of a requirements definition. It may be wise to have requirements covering all relevant external attributes of the software product.
- Identify software design objectives. The design effort should be directed at achieving the quality characteristics that are the most valuable for the product.
- Identify software testing objectives. Part of the testing should be directed at measuring the compliance to the defined quality requirements.
- Identify quality assurance criteria. The quality assurance effort should be directed towards the greatest risks, and the highest gains, when developing a software product.
- Identify user acceptance criteria for a completed software product. Without user acceptance any software, elegant though the code may be, cannot be judged to be a quality product.

2.3 Characteristics

The ISO/IEC 9126 standard applies several attributes in the characterisation of software product quality. The used attributes are categorised into six characteristics as shown in Figure 1.

The six characteristics cover the key aspects of how satisfied a software user is likely to be. For different software products different software characteristics may be of different importance. For instance, the system may be:

- A teller machine, used by hundreds of casual users each day. For such systems Usability is of paramount importance. These users do not care about Portability, nor about Maintainability. On the other hand, the system responsible personnel do care about Maintainability.
- An embedded telecom flight control system used for the vital communication between dispatchers and aeroplane cockpit personnel. We would surely care a lot about the Reliability of their system. But the personnel, as frequent users of the system, and well trained in the use, learn to live with systems with less than perfect Usability.

Often, a *customer* achieves a software system. The customer may act on behalf of, or as a representative of many *users.* In such cases it is important that the customer is aware of the users' needs. It is also important that the customer is aware of the various user roles. Often the user is thought of as the 'end user', the person using the software system for his own purposes. But it should be noted that there are other user roles. One other important user role is the software maintenance person.

2.3.1 Definition of characteristics

The six basic quality characteristics are defined in ISO/IEC 9126 [28]. The definitions are shown in Table 1. The idea at this level is to define each characteristic as a capability of the software. We can thus understand that the characteristics are external attributes of the software, that may be measured by observing the software in use.

The overall quality is not defined as some weighted function of these characteristics. The characteristics are to be understood in the sense of subdivisions, or factors. For a particular application, we can expect the characteristics to be made measurable and appear in the requirements specification, at least in form of some objectives for the characteristics.

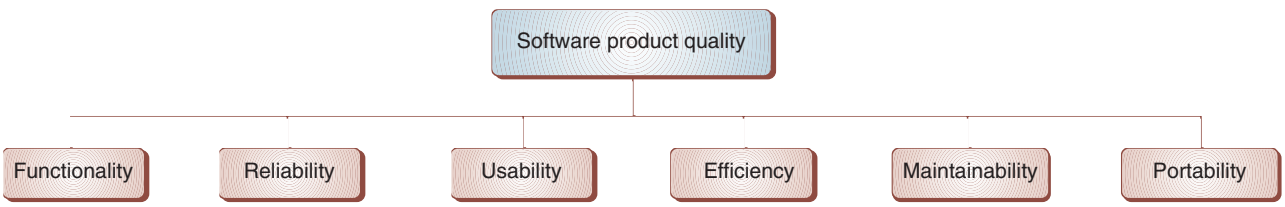


Figure 1 Software product quality characteristics

Table 1 Definition of characteristics

Characteristic	Definition
Functionality:	The capability of the software product to provide functions, which meet stated and implied needs when the software is used under specified conditions.
Reliability:	The capability of the software product to maintain a specified level of performance when used under specified conditions
Usability:	The capability of the software product to be understood, learned, used and liked by the user, when used under specified conditions.
Efficiency:	The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions.
Maintainability:	The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.
Portability:	The capability of the software product to be transferred from one environment to another.

The definitions of the characteristics are as shown in Table 1.

We see from the definitions that the requirements for these characteristics may differ for different applications. For instance, the requirements for Portability may wisely define to what environments the porting might be done. The requirements may also put restrictions on language constructs to be used, to make the system as general (i.e. portable) as possible. The characteristics above are measurable, if the measurement method is decided upon. ISO/IEC has chosen to describe the characteristics further, by applying subcharacteristics to each characteristic.

2.4 subcharacteristics

The division of the six product characteristics into subcharacteristics as described in [28] is shown in Figure 2.

Each of these subcharacteristics has their own definition. These definitions are given in the cited standard, and presented in Appendix 1 of this paper.

2.5 Metrics

2.5.1 The choice of metrics

One important purpose of characteristics and subcharacteristics, is to provide a means to measure quality attributes of a software system. One might agree that the subcharacteristics might be easier to refine to a measurable form, than the characteristics. The measurable form of a quality attribute is called a *metric*. Subcharacteristics can be measured by

- Internal metrics. Internal metrics are based on things that can be measured by examining the software itself. Examples of internal metrics are given in ISO/IEC 9126-3 [29].
- External metrics. These are based upon external system behaviour. Examples of external metrics are given in ISO/IEC 9126-2 [30].

It should be made clear that the metrics to use in a specific delivery cannot be directly extracted from these documents. For a specific delivery the choice of metrics will be given by:

- The particular needs of the end user (or customer specification);

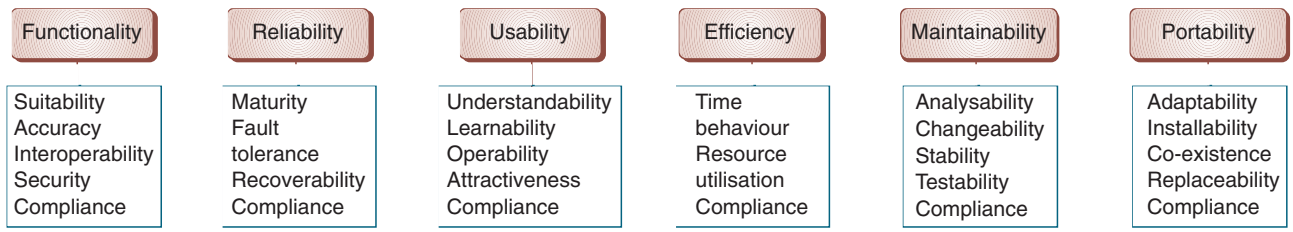


Figure 2 Subcharacteristics

- The specific software architecture that planning is done for and the way this architecture will help achieve the end user's quality needs;
- The specific design and the way the design will help to achieve the end user's quality needs.

2.5.2 Internal metrics

Measurements can be performed by inspection of the software without executing it. Internal metrics can be applied to non-executable software representations (such as a specification or source code) during design and production. When developing a software product the intermediate products should be evaluated using internal metrics. In addition to inspection, measurements may also be derived from simulated behaviour.

The internal metrics should indirectly indicate that the required external quality and quality in use might be achieved. Internal metrics should provide the ability to evaluate software product quality, and address quality issues before the software product becomes executable. The measurements of internal metrics use numbers or frequencies of software composition elements that are observed during the measurement task. Examples of such composition elements are the control graph, data flow and state transition representations. Documentation can also be evaluated using internal metrics.

In [29], some 70 metrics are defined. A typical example of an internal metric may be *Functional Implementation Coverage* defined as *"The ratio between Number of implemented functions confirmed in review and Number of requirements defined in requirements/ functional specifications"*. Fagan inspections [19] can be used for this purpose.

Metrics are defined for all the six characteristics of software quality. At this stage of the standardisation work of defining internal metrics, it is only natural that the selection of metrics that are agreed upon is still not complete. However, the work proceeds to define metrics for all subcharacteristics as well.

It should be noted that metrics based on structural properties of the code or the design, such as McCabes cyclomatic number, fan-in/fan-out and the like, are considered as structural properties of the software. These structural properties are *not* utilised in any of the quality characteristics. The idea that these structural properties may correlate with quality still exists. It is also still a valid idea, as it is evident that the possibility of making an error in coding is greater if the code is unnecessarily complex. Some of these structural metrics do have the merit of exposing bad code structures, especially high code complexity. On the other hand, code complexity relates strongly to problem complexity. For such reasons, and because any causal relation between the structural properties and quality characteristics have not been established, the standardisation community in general has abandoned the idea of relating quality to structural properties.

2.5.3 External metrics

Characteristics and subcharacteristics can be measured externally to the extent that a capability under study can be observed by the execution of the software. External metrics use measures of a software product derived from measures of the behaviour of the system of which it is a part, by testing or operating, and observing the executable software. Before acquiring or using a software product, the product should be evaluated using metrics based on business objectives. The business objectives should be related to the use, exploitation and management of the product in a specified organisational and technical environment. External metrics do provide the ability to evaluate software product quality during testing or operation.

In [30] some 90 metrics are defined. Metrics are defined for all the given characteristics of software quality. The definitions of the metrics are given in the form "that can be observed during testing or operation". An example may be Functional Implementation Completeness, defined as: "The number of implemented functions confirmed in software execution" relative to "The number of functions described in specifications".

At this stage of the work of defining external metrics, it is only natural that the selection of metrics agreed upon is still not complete. However, work is proceeding to define metrics for all subcharacteristics as well. The work will also proceed to divide these metrics more formally into 'External' metrics and 'Quality in use' metrics.

2.5.4 Relation between internal and external metrics

The levels of some internal metrics have been found to correspond to variations in the levels of some external metrics, so that there is both an external aspect and an internal aspect to most quality attributes for software. For example, reliability may be measured externally during a trial of the software, by observing the number of failures in a given period of execution time. Reliability may also be measured internally by inspecting the detailed specifications and source code to assess the level of fault tolerance.

The correlation between internal attributes and external measures is never perfect, and the effect that a given internal attribute has upon an associated external measure will be determined by experience, and will depend on the particular context in which the software is used. This context should be mirrored in the requirements specification for the software.

When the software quality requirements are defined, the software quality characteristics or subcharacteristics that contribute to the quality requirements should be listed. Then the appropriate external metrics and acceptable ranges should be specified to quantify the *quality criteria* that validate that the software will meet the user needs. The internal quality attributes of the software are defined and specified afterwards. The internal quality attributes are used as a quality control mechanism to achieve the required external quality and the quality in use. Appropriate internal metrics and acceptable ranges are specified to quantify the internal quality attributes so that they can be used for validating that the intermediate software representations meet the internal quality specifications during the development.

3 Quality in use

During the development of the standards for software quality, it was felt necessary to have some metrics to measure to what extent the software product really met the user needs and requirements. Such metrics are called *quality in use metrics*. They measure the extent to which a product meets the needs of specified users with effectiveness, productivity, safety and satisfaction. The measurements have to be taken in a situation with specified users, specified user goals and a specified

context of use. Thus, evaluating quality in use validates software quality in specific user-task scenarios. Quality in use should be the user's view of the quality of the software, and is measured in terms of the result of using the software, rather than properties of the software itself. Quality in use is the combined effect for the user of all the software quality characteristics.

The relationship of quality in use to the other software quality characteristics depends on the type of user:

- The end user for whom quality in use is mainly a result of functionality, reliability, usability and efficiency;
- The person maintaining the software for whom quality in use is a result of maintainability;
- The person porting the software for whom quality in use is a result of portability.

Quality in use has four characteristics as shown in Figure 3.

Formally, the Quality in use characteristic is defined as: *The capability of the software product to enable specified users to achieve specified goals with effectiveness, productivity, safety and satisfaction, in specified contexts of use*. The different characteristics are defined as shown in Table 2.

Also for Quality in Use Characteristics, there are plans to establish metrics. For the time being, this activity has still much work to do. Some indications of what may be the quality in use metrics can today be found in [30]. An example may be a metric called 'Task effectiveness'. This metric is defined as "The proportion of task goals represented in the output of task" multiplied with "The degree to which the task goals represented in the output have been achieved".

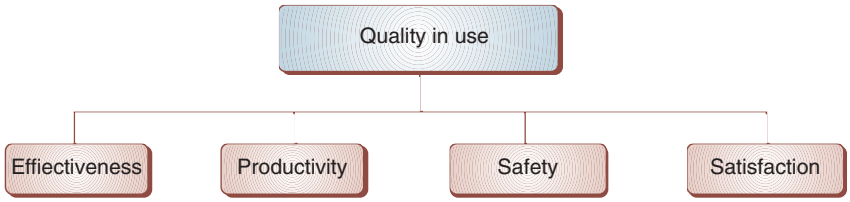


Figure 3 Characteristics of quality in use

Table 2 Quality in use characteristics

Quality in use characteristic	Definition and *Notes*
Effectiveness:	The capability of the software product to enable users to achieve specified goals with accuracy and completeness in a specified context of use.
Productivity:	The capability of the software product to enable users to expend appropriate amounts of resources in relation to the effectiveness achieved in a specified context of use. *Relevant resources can include time, effort, materials or financial cost.*
Safety:	The capability of the software product to achieve acceptable levels of risk of harm to people, software, equipment or the environment in a specified context of use. *Risks to safety are usually a result of deficiencies in the functionality, reliability, usability or maintainability.*
Satisfaction:	The capability of the software product to satisfy users in a specified context of use. *Psychometrically valid questionnaires can be used to obtain reliable measures of satisfaction.*

4 Product and Process

Through the use of metrics given in ISO/IEC 9126, one is able to define objective product evaluation criteria that relate to the software product in any representation. The quality model defined covers all phases of software development.

The metrics of ISO/IEC 9126 can be used in conjunction with ISO/IEC 15504 [31], which is concerned with software process assessment, to provide:

- A framework for software product quality definition in the customer-supplier process;
- Support for review, verification and a framework quantitative quality evaluation, in the support process;
- Support for setting organisational quality goals in the management process.

Measurement of quality may also be considered in conjunction with ISO/IEC 12207 [32], which is concerned with the software life cycle, to provide:

- A framework for software quality requirements definition in the primary life cycle process;
- A support for review, verification and validation in supporting life cycle processes.

Furthermore, measurement of quality may be considered in conjunction with ISO/IEC 14598 [33]. This standard is concerned with the process of evaluating a software product, for example in connection with acquiring it.

In addition, measurement must be considered in conjunction with ISO 9001 [24], which is concerned with the quality assurance process, to provide:

- Support for setting quality goals;
- Support for design review, verification and validation.

When a company masters all these aspects of software quality, it is today in line with the best companies world-wide.

5 References

1 Naur, P, Randell, B (ed.). *Software engineering. Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany,* 1968.

2 Dahl, O-J, Dijkstra, E W, Hoare, C A R. *Structured programming.* London, Academic Press, 1972.

3 Parnas, D L. A technique for software module specification with examples. *Comm. ACM,* 15 (5), 330–336, 1972.

4 Hecht, H. Fault-tolerant software : motivation and capabilities. In: *Proc. Symp. Computer Software Engineering.* New York, Polytechnic Press, 1976.

5 Llewelyn, A I, Wickens, R F. The testing of computer software. In: *Software engineering. Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany,* 1968.

6 Randell, B. Towards a methodology of computing systems design. In: *Software engineering report on a conference sponsored by the Nato Science Committee,* Garmisch, Germany, 1968, 204–208.

7 Peters, L J, Tripp, L L. Software design representation schemes. In: *Proc. Symp. Computer Software Engineering.* New York, Polytechnic Press, 1976.

8 Rumbaugh, J et al. *Object-oriented modelling and design.* Englewood Cliffs, N.J., Prentice-Hall, 1991.

9 Bachman, C W. Data structure diagrams. *Database : a quarterly newsletter of SIGBDP,* 1 (2), 4–10, 1969.

10 Chen, P P S. The entity-relationship model : toward a unified view of data. *ACM Trans, Database systems,* 1 (1), 9–36, 1976.

11 Boehm, B. *Software engineering economics.* Englewood Cliffs, N.J., Prentice-Hall, 1981.

12 Meseke, D W. The data-processing system performance requirements in retrospect. *Bell Syst. Tech. Journal, Special Supplement: Safeguard Data-Processing System.* 1975.

13 Scharer, L. Pinpointing requirements. *Datamation,* 27, 139–154, April 1981.

14 Hoare, C A R. An axiomatic basis for computer programming. *Comm. ACM*, 12 (10), 576–80, 1969.

15 IEEE. *Standard for software verification and validation plans.* New York, 1986. (IEEE 1012.)

16 IEEE. *Standard for configuration management plans.* New York, 1990. (IEEE 828.)

17 Hearn, D, Baker, P. *Computer graphics.* Englewood Cliffs, N.J., Prentice-Hall, 1986.

18 Freedman, D P, Weinberg, G M. *Handbook of walkthroughs, inspections and technical reviews.* Boston, Little, Brown, 1982.

19 Fagan, M E. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15 (3), 182–211, 1976.

20 Halstead, M H. Natural laws controlling algorithmic structure. *ACM SIGPLAN Notices* 7 (2), 1972. (See Elements of Software Science. Elsevier, 1978.)

21 Boehm, B et al. *Characteristics of software quality.* Amsterdam, North Holland, 1978.

22 McCall, J A, Cavano, J P. A framework for the measurement of software quality. In: *Proceedings of the Software Quality and Assurance Workshop.* New York, ACM, 1978, 133–139.

23 El Emam, K, Drouin, J-N, Melo, W. *SPICE : the theory and practice of software process improvement and capability determination.* Los Alamos, Calif., IEEE Computer Society Press, 1998.

24 ISO. *Quality systems. Model for quality assurance in design, development, production, installation and servicing.* Geneva, 1994. (ISO 9001.)

25 ISO. *Quality management and quality assurance standards. Part 3 : guidelines for the application of ISO 9001 to the development, supply and maintenance of software.* Geneva, 1992. (ISO 9000-3.)

26 ISO. *Quality management and quality assurance.* Geneva, 1994. (ISO 8402.)

27 ISO. *Software product evaluation : quality characteristics and guides for their use.* Geneva, 1991. (ISO/IEC 9126.)

28 ISO. *Information technology : software product quality. Part 1 : quality model 1998-06-26.* Geneva, 1998. (FCD 9126-1.2.)[1)][2)]

29 ISO. *Information technology : software quality characteristics and metrics. Part 3 : internal metrics 1997-05-07.* Geneva, 1997. (WD 9126-3.)[3)]

30 ISO. *Information technology : software quality characteristics and metrics. Part 2 : external metrics 1997-07-28.* Geneva, 1997. (PDTR 9126-2.)[4)]

31 ISO. *Information technology : software process assessment.* Geneva, 1996. (ISO/IEC PDTR 15504.)

32 ISO. *Information technology : software life cycle processes.* Geneva, 1995. (ISO/IEC 12207.)

33 ISO. *Information technology : software product evaluation. Part 1 : general overview.* Geneva, 1998. (ISO/IEC 14598-1.)

---

1) Voting on this document ended on 26.10.1998. Although a majority may vote in favour of the document, some editorial comments may have to be taken into account before 9126-1.2 is submitted as an approved document.

2) ISO/IEC 9126 is planned to consist of the following parts; 1: Quality models, 2: External Metrics, 3: Internal Metrics, 4: Quality in use metrics.

3) Voting on this document ended 01.09.1997. Although many nations approved the document, there were a substantial amount of comments to be processed before issuing a new version of the document.

4) Voting on this document ended on 11.11.1997. A restructuring of the document is anticipated to separate more formally between external metrics and metrics for quality in use.

Table A1 Functionality subcharacteristics

Functionality subcharacteristic	Definition
Suitability	The capability of the software product to provide an appropriate set of functions for specified tasks and user objectives.
Accuracy	The capability of the software product to provide the right or agreed results or effects.
Interoperability	The capability of the software product to interact with one or more specified systems.
Security	The capability of the software product to protect information and data so that unauthorised persons or systems cannot read or modify them and authorised persons or systems are not denied access to them.
Compliance	The capability of the software product to adhere to standards, conventions or regulations in laws and similar prescriptions.

Table A2 Reliability sucharacteristics

Reliability subcharacteristic	Definition
Maturity	The capability of the software product to avoid failure as a result of faults in the software.
Fault tolerance	The capability of the software product to maintain a specified level of performance in cases of software faults or of infringement of its specified interface.
Recoverability	The capability of the software product to re-establish its level of performance and recover the data directly affected in the case of a failure.
Compliance	The capability of the software product to adhere to standards, conventions or regulations relating to reliability.

Table A3 Usability subcharacteristics

Usability subcharacteristic	Definition
Understandability	The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use.
Learnability	The capability of the software product to enable the user to learn its application.
Operability	The capability of the software product to enable the user to operate and control it.
Attractiveness	The capability of the software product to be liked by the user.
Compliance	The capability of the software product to adhere to standards, conventions, style guides or regulations relating to usability.

Appendix 1 Subcharacteristics of software quality

A1 Functionality subcharacteristics

The subcharacteristics of functionality have the definitions shown in Table A1.

A2 Reliability subcharacteristics

The subcharacteristics of reliability have the definitions shown in Table A2.

A3 Usability

The subcharacteristics of usability have the definitions shown in Table A3.

A4 Efficiency

The subcharacteristics of efficiency have the definitions shown in Table A4.

A5 Maintainability

The sucharacteristics of Maintainability have the definitions shown in Table A5.

A6 Portability

The subcharacteristics of portability have the definitions shown in Table A6.

Table A4 Efficiency subcharacteristics

Efficiency subcharacteristic	Definition
Time behaviour	The capability of the software product to provide appropriate response and processing times and throughput rates when performing its function, under stated conditions.
Resource utilisation	The capability of the software product to use appropriate amounts and types of resources when the software performs its function under stated conditions.
Compliance	The capability of the software product to adhere to standards or conventions relating to efficiency.

Table A5 Maintainability subcharacteristics

Maintainability subcharacteristic	Definition
Analysability	The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.
Changeability	The capability of the software product to enable a specified modification to be implemented.
Stability	The capability of the software product to minimise unexpected effects from modifications of the software.
Testability	The capability of the software product to enable modified software to be validated.
Compliance	The capability of the software product to adhere to standards or conventions relating to maintainability.

Table A6 Portability subcharacteristics

Portability subcharacteristic	Definition
Adaptability	The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered.
Installability	The capability of the software product to be installed in a specified environment.
Co-existence	The capability of the software product to co-exist with other independent software in a common environment sharing common resources.
Replaceability	The capability of the software product to be used in place of another specified software product for the same purpose in the same environment.
Compliance	The capability of the software product to adhere to standards or conventions relating to portability.

Øystein Skogstad (55) is Senior Scientist at SINTEF Telecom and Informatics, Trondheim. He is working with safety issues relating to electronic railway signalling systems. During his career he has also been engaged in research activities in various other areas including reliability in telecommunication switching, telecom network reliability, software quality assurance and software engineering methods.

e-mail: Oystein.Skogstad@informatics.sintef.no

Software quality according to measurement theory

MAGNE JØRGENSEN

This paper analyses our ability to measure software quality. The analysis is based on the representational theory of measurement. We analyse three assumptions on software quality measurement and recommend an increased measurement focus on establishing empirical relational systems (models of what we measure) in software quality related work. Applying measurement theory on the measurement of software quality related properties means that we have a powerful tool to understand what we measure, what the meaningful operations on the measured values are, and how to interpret the results.

1 Introduction

Most of the software quality standards and frameworks, such as ISO 9001/9000-3, the Capability Maturity Model, [1], ANSI/IEEE Std. 730-1989 and ESA PSS-05-0 1991, require or recommend measurement of software quality. Unfortunately, there is a large gap between the requirement *that* quality measurement should be carried out and the guidelines on *how* to carry out the measurements. For example, the quality standard ISO 9000-3 Section 6.4.1 states that: *"There are currently no universally accepted measures of software quality. ... The supplier of software products should collect and act on quantitative measures of the quality of these software products."*

Here, ISO 9000-3 requires quality measurement and at the same time admits that there are no (universally) accepted quality measures. In order not to have contradicting requirements ISO 9000-3 (and other similar frameworks) may have made one or more of the following assumptions:

A1: Although no universal software quality measures exist, there are meaningful quality measures for particular environments.

A2: Widely accepted quality measures will occur, when the software quality measurement research becomes more mature.

A3: Measurement of software quality indicators (so-called quality factors) can be measured and used to predict or indirectly measure the software quality.

This paper analyses the validity of these assumptions from the perspective of the 'representational measurement theory'.

We give a brief introduction to measurement theory in Section 2. In Section 3 we apply the theory to formulate the necessary preconditions for meaningful measurement of software quality. Then, in Section 4, we discuss whether common quality definitions enable meaningful measurement of software quality. Finally, in Section 5, we draw conclusions about the validity of the assumptions A1-A3 and recommend actions based on the analysis results.

1.1 Previous work on measurement theory and software quality

Possibly, the first paper on measurement theory was written by Stevens [2] in 1946. In 1981, the first paper connecting software measurement and measurement theory was published, see DeMillo and Liption [3]. The first real application of measurement theory on software measurement was, according to Melton [4], not carried out before the late 1980s. In the 1990s, there have been several software measurement theory papers, mainly on the evaluation of software complexity measures, see for example Zuse [5] and Fenton [6]. As far as we know, no paper on measurement theory in depth analysing software quality measurement has been published.

2 A brief introduction to measurement theory

When we measure length in metres and weight in kilograms we do not reflect much on what we do when we measure. Measuring an attribute where we have a common understanding of what we measure and accepted measurement methods, this lack of reflection is not harmful. When measuring software quality, however, we do not have an accepted understanding of what quality is, nor do we have commonly accepted measures. For this reason, we need a framework for quality measurement reflections, such as reflections on

- The preconditions for quality measurement;
- How to analyse the meaningfulness of quality measures; and
- How to interpret and use the measured values.

We argue that measurement theory establishes such a framework.

2.1 Measurement theory definitions

The following definitions were adapted from Melton [4]:

Def. Empirical Relational system: $<E,\{R_1..R_n\}>$, where E is a set of entities and $R_1..R_n$ the set of empirical relations defined on E with respect to a given attribute (for example, the attribute quality).

The empirical relational system is a model of the 'world' and represents a perspective on our knowledge about the phenomenon to be measured. The model should ensure agreement about the empirical relations and enable measurement. For example, to state that program A has more modules than program B we need a model of programs that enable us to agree upon what a module is and how to identify it.

Def. Formal (numerical) Relational system: $<N,\{S_1..S_n\}>$, where N is a set of numerals or symbols, and $S_1..S_n$ the set of numerical relations defined on N.

Def. Measure: M is a measure for $<E,\{R_1..R_n\}>$ with respect to a given attribute iff:

1. M: $E \rightarrow N$
2. $R_i(e_1, e_2, ... e_k) \Leftrightarrow S_i(M(e_1), M(e_2), ... M(e_k))$, for all i.

Condition 1 says that a measure is a mapping from entities to numbers or symbols. Condition 2, the representation condition, requires equivalence between the empirical and the formal relations.

Def. Admissible transformation: Let M be a measure, E a set of entities, N a set of numerals and F a transformation (mapping) from $M(E)$ to N, i.e. F: $M(E) \rightarrow N$. F is an admissible transformation iff $F(M(E))$ is a measure.

In other words, an admissible transformation preserves the equivalence between the empirical relations and the formal relations.

The definition of admissible transformations enables a **classification of scales**.

A common classification of scales is the following:

Nominal scale: Admissible transformations are one-to-one mappings. The only empirical relation possible is related to 'equality'. *Separating programs into 'structured' and 'unstructured' leads to a nominal scale.*

Ordinal scale: Admissible transformations are strictly increasing functions. The empirical relations possible are related to 'equality' and 'order'. *Assigning the values 'high quality', 'medium quality' and 'low quality' to software leads to an ordinal scale.*

Interval scale: Admissible transformations are of the type $F(x) = ax + b$, $a > 0$. The empirical relations possible are related to 'equality', 'order' and 'difference'. *The temperature scale (in degrees Celsius) is an interval scale.*

Ratio scale: Admissible transformations are of type $F(x) = ax$, $a > 0$. The empirical relations possible are 'equality', 'order', 'difference' and 'relative difference'. *The length of programs measured in lines of code forms a ratio scale.*

Software quality should at least be measurable on an ordinal scale, an interval scale or a ratio scale. Otherwise, we would not be able to state that software A is 'better' than software B with respect to quality, i.e. our intuition of what software quality is would be strongly violated. In measurement theory terminology this means that we should require that the empirical relational system includes an accepted understanding of the relation 'higher quality than'.

The appropriateness of statistical and mathematical methods to the measured values is determined by the type of scale, i.e. by the admissible transformations on the scale values. For example, addition of values are meaningful for values on an interval or ratio scale, not on an ordinal or nominal scale. This means, for example, that 'units of software quality' must be meaningful to calculate the mean software quality.

There are diverging opinions on how rigidly the scale prescriptions should be interpreted, see Briand et al. [7] for an overview.

3 Preconditions for the measurement of software quality

Depending on how we define software quality, software quality may be *directly* measured, *indirectly* measured or *predicted.* The preconditions for the different measurement types differ. Below we have applied measurement preconditions on the measurement of software quality.

The preconditions for *direct measurement* of software quality are that:

1 The empirical relational system of software quality is established. In our opinion, this means that, at least, we should have a common understanding of the relations 'same quality as' and 'better quality than'.

2 A numerical or symbolic system with equivalent formal relations to the empirical quality relations is established, for example the formal relations "=" and ">".

3 A measure (mapping) from the software quality to numbers or symbols is defined.

4 A measurement tool or method implementing the measure must be implemented.

The preconditions for *indirect measurement* of software quality are that:

1 The preconditions for measurement of the directly measured software attributes are met.

2 A complete empirical connection between the directly measured attributes and the indirectly measured software quality is established. The connection is in our case complete when all aspects of the agreed understanding of software quality can be determined from the directly measured attributes.

3 The connection is accurately translated into the formal relational system (through a formula).

Predictions of software quality are similar to indirect measurement. The difference is that predictions do not require a complete empirical connection or an accurate translation into the formal relational system, i.e. all the empirical relations are not necessarily preserved into the formal relational system. Notice the difference between this definition and the common use of the term. Commonly, the meaning of predictions is limited to statements about future events, for example weather predictions. In measurement theory, prediction is used about all incomplete measurements, not necessarily about future events; for example, the prediction of the remaining number of errors in a piece of software based on the number of errors found in the system test.

4 Common quality definitions and measurement theory

The most common types of software quality definitions are, probably, the following three:

1 Quality is determined by a set of quality factors (see for example ISO 8402-1986 and IEEE 610.12-1990).

2 Quality is determined by the user satisfaction, see for example Deephouse [8].

3 Quality is determined by the errors or unexpected behaviour of the software, see for example Carey [9] and Lanubile [10].

These definitions seem to be based on a similar 'intuition' of what software quality is. For example, they seem to share the opinion that software quality is the degree of meeting the user needs. The difference may lay in whether they consider user needs in the form of 1) requirements and implied needs, 2) user satisfaction, or 3) the level of incorrect behaviour of the software. From the viewpoint of measurement theory these differences are significant, and lead to different empirical relational systems.

Note that there are numerous other definitions and perspectives on (software) quality, see for example Garvin [11] and Dahlbom & Mathiassen [12] for classifications and overviews.

4.1 Quality as a set of quality factors

The most common definitions of software quality in the quality standards are variants of the ISO quality definition: (ISO 8402-1986) *The totality of features and characteristics of a product or service that bear on its ability to meet stated or implied needs.*

Examples of such "features and characteristics" (quality factors) are efficiency, flexibility, integrity, interoperability, maintainability and portability.

The above type of definition has, among other things, the following implications for the empirical relational system of software quality:

- The relation 'better quality than' should be interpreted as a better ability to meet stated and implied needs.
- In order to measure and compare software quality we need to formulate an empirical connection between the quality factors and the software quality itself, i.e. software quality is *indirectly* measured.

We will argue that the preconditions for indirect measurement are very hard to meet for this type of empirical relational system for the following reasons:

- It is far from obvious that all the quality factors, for example maintainability, are measurable. In our opinion, some of the quality factors are just as difficult to measure as the software quality itself.
- As long as our understanding of software quality is at an 'intuitive' level and not explicitly formulated we will not be able to establish a complete empirical connection between the quality factors and the software quality. For example: *Assume that we believe (or agree on) that the only relevant quality factors are reliability and usability, and that these quality factors are measurable. We will probably not always be able to agree on the total impact on the quality of a decreased usability together with (or perhaps caused by) an increased reliability. In other words, we have no complete connection between quality and the quality factors reliability and usability.*

From a measurement theory viewpoint, this means that the quality factor based definitions do not enable measurement of software quality. One might argue that the quality factor definitions suggest a measurement of quality factors in order to *predict* the software quality. In this situation an accurate connection between the quality factors and the software quality is not needed. On the other hand, in order to have a meaningful prediction system we still need an empirical relational system including an accepted understanding of the relation 'higher quality than', i.e. an accepted understanding of 'a higher ability to meet user needs'. Currently, such an empirical relational system does not exist.

4.2 Quality as user satisfaction

A common approach, for example by the quality framework Total Quality Management (TQM), is to define or understand software quality as the level of user satisfaction. This understanding of software quality has, among other things, the following consequences for the empirical relational system of software quality:

- There must be a commonly accepted method of identifying the user satisfaction;
- There must be a commonly accepted meaning of the relations 'same quality as' and 'better quality than' based on user satisfaction

We may argue that this is possible because:

- A method of identifying the user satisfaction may be to ask them;
- 'Same/higher quality than' may be understood as the same/higher proportion of satisfied users.

In addition, a measure (mapping) from software quality to numbers or symbols can be the mapping from 'empirical proportions' to 'numerical proportions' of satisfied users. This measure preserves the relation 'higher quality than'.

The main problems with this type of software quality measurement are, in our opinion, that:

- The measured values do not always correspond with the common intuition of what software quality is. For example, the user satisfaction (quality) is not a characteristic of the software, but of the user, i.e. the software quality can change regardless of changes in the software.
- Why call it software quality, when we measure user satisfaction?
- How do we know that user A's 'very satisfied' means the same as user B's 'very satisfied'?

4.3 Software quality related to errors

In spite of the high number of sophisticated definitions of software quality, the software industry seems to operationalise software quality as the degree of errors in the software, for example those errors found by the end users. In order to compare the quality of software with different size the errors get divided by a measure of the software size, frequently the number of lines of code.

IEEE 1044-1993 defines an error (anomaly) as *any condition that departs from the expected*. The expectations may be described in a document or be based on other sources. To count (and categorise) errors we need to agree on how to decide whether a condition is expected or not, and whether a condition should be counted as one or more errors. To some extent, this seems to be possible. For example, we could agree that only the expectations specified in the requirement specification and the obvious implications of the requirement specifications should be counted as expectations.

Surprisingly, there is no commonly accepted definition of lines of code. There are, however, many company specific definitions which enable measurement of lines of code. Lines of code is a frequently criticised measure. The criticism, however, has mainly been on the use of lines of code in measurement of productivity. From a measurement theoretical viewpoint lines of code is a valid measure of the 'physical' length of the source code of software.

Assuming that we accept that software quality can be measured as errors divided by software size, the relation 'higher quality than' can, for example, be interpreted as a lower value of the ratio 'errors/lines of code'.

A naïve use of these measures, i.e. not distinguishing between different types of errors, may give counter intuitive results. For example, two cosmetic errors have a stronger impact on the quality than one serious error.

To avoid this counter intuitive result, a classification of the errors in for example severity of failure (see IEEE 1044-1993 for an overview of classifications) is needed. This means that we need a complete empirical connection between the error categories and the software quality

in order to measure quality, i.e. indirect measurement. In practice, this type of indirect measurement seems to be just as difficult as the indirect measurement through the quality factors.

5 Conclusions and recommendations

Based on the previous analyses, we now examine the assumptions from Section 1:

A1: *Although no universal software quality measures exist, there are meaningful quality measures for particular environments.*

In order to establish meaningful software quality measures for particular environments and situations the involved persons should have approximately the same empirical relational system of software quality. For example, the users may decide that the only software characteristic important for the quality is how much the software decreases the error density in registering customer orders. Assuming a baseline error density the software quality may be measured as the percentage decrease in error density. From a measurement theory viewpoint this is an example of valid measurement. It is, on the other hand, an example of a label on a measure that hides what is actually being measured.

A lot of empirical studies on software development and maintenance define quality measures for particular environments (or applications), see for example the quality measures of the Software Engineering Laboratory for the NASA Goddard environment described in Basili & Selby [13]. The variation of the empirical relational systems of software quality in these studies is very high. An example is Schneiderman [14] who uses software quality as a label on a measure of the ease of memorisation/recalling program information.

For this reason, the reader of software quality studies should always try to get an understanding of the empirical relational system of a software quality measure before interpreting the results.

When there is no agreed understanding of software quality, or only intuitive and non-articulated connections between what is measured and software quality, the measurement is not a meaningful quality measurement.

A2: *Widely accepted quality measures will occur when the software quality measurement research becomes more mature.*

If we require that the empirical relational system shall correspond with our intuitive understanding of software quality, it is not likely that we will see widely accepted measures of software quality. This lack of ability to agree on the understanding of software quality has, in our opinion, not much to do with the maturity of the software measurement discipline. Instead it has to do with the intrinsic difficulties of explicitly formulating what we understand by the very complex phenomenon 'software quality' and the many different context and culture dependent viewpoints on software quality.

It is, of course, possible to 'standardise' on one or more viewpoints on software quality – which has been done by many international standards and frameworks for software quality (see previous sections). Unfortunately, we may have to choose between a vague quality definition, which does not describe a complete empirical relational system enabling software quality measurement, and a well-described empirical relational system, which does not correspond to the intuitive understanding of software quality.

The wish expressed in Price Waterhouse [15] *"... research should be undertaken with a view to developing a workable definition of software quality, and measures of quality ... that can be easily implemented by software developers"* may therefore be hard to meet when aiming at widely accepted quality measures.

A3: *Measurement of software quality indicators (so-called quality factors) can be measured and used to predict or indirectly measure the software quality.*

The method of dividing a complex characteristic into less complex characteristics in order to indirectly measure the complex characteristic is typical for scientific work, and may have inspired for example IEEE in their software quality factor work, see Schneidewind [16]. This type of indirect measurement is only meaningful if an empirical connection between the directly and the indirectly measured characteristics is established. This type of connection is, for example, established for the indirect measurement of 'speed' through the direct measurement of 'length' per 'time unit'. However, in spite of a lot of effort we have not been able to agree on similar connections for software quality. Even worse, it is not obvious that the most common quality factors, such as maintainability and user friendliness, are less complex to measure than quality itself. As long as we are unable to establish empirical connections between the quality factors and the quality itself, the quality factor work may not be able to contribute much to the work on software quality.

On the basis of the previous analysis, we *recommend* that:

- Instead of trying to develop a widely accepted software quality measure (or set of quality factors) we should focus on establishing empirical relational systems and better measures for the more measurable quality related properties of software, such as user satisfaction or error density. Measurement theory should be applied in order to know what we measure and how to interpret the results.
- The measures of software quality related attributes should not be called software quality measures. For example, to call a measure of error density a quality measure is unnecessary and misleading.
- There should be an increased empirical research and industry focus on the connections between the different quality related attributes. Measurement theory should be consulted when describing these connections. This way we may be able to build better prediction systems for quality related attributes, such as user satisfaction or reliability.

References

1 Paulk, C et al. *The capability maturity model : guidelines for improving the software process.* Reading, Mass., Addison-Wesley, 1995.

2 Stevens, S. On the theory of scales of measurement. *Science,* 103, 677–680, 1946.

3 DeMillo, R A, Liption, R J. Software project forecasting. In: *Software metrics.* Perlis, A J et al. (eds.). Cambridge, MA, MIT Press, 1981, 77–89.

4 Melton, A (ed). *Software measurement.* London, International Thomson Computer Press, 1995.

5 Zuse, H. *Software complexity : measures and methods.* Amsterdam, de Gruiter, 1990.

6 Fenton, N. Software measurement : a necessary scientific basis. *IEEE Transactions on Software Engineering,* 20 (3), 199–206, 1994.

7 Briand, L et al. On the application of measurement theory in software engineering. *Empirical Software Engineering,* 1 (1), 1996.

8 Deephouse, C et al. The effects of software processes on meeting targets and quality. In: *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences,* Hawaii, 1995, 4, 710–719.

9 Carey, D. Is "Software quality" intrinsic, subjective or relational? *Software Engineering Notes,* 21 (1), 74–75, 1996.

10 Lanubile, F, Visaggio, G. *Evaluating predicting quality models derived from software measures : lessons learned.* University of Maryland, Dept. of Computer Science, 1996. (Technical report CS-TR-3606.)

11 Garvin, A. Competing on the eight dimensions of quality. *Harvard Business Review,* 65, Nov–Dec, 101–104, 1987.

12 Dahlbom, B, Mathiassen, L. *Computers in context.* Oxford, NCC Blackwells, 1995.

13 Basili, V R, Selby, R W. Calculation and use of an environment's characteristic software metric set. In: *Proceedings of the 8th Int. Conf. on Software Engineering,* London, 386–393, 1985.

14 Schneiderman, B. Measuring computer program quality and comprehension. *Int. J. Man-Machine Studies,* 9, 465–478, 1977.

15 Price Waterhouse. *Software quality standards : the costs and benefits.* London, Price Waterhouse, 1988.

16 Schneidewind, N. Methodology for validating software metrics. *IEEE Transactions on software engineering,* 18 (5), 410–422, 1992.

Magne Jørgensen (34) received the Dr.Scient. degree in informatics from the University of Oslo in 1994. From 1989 to 1998 he was a research scientist at Telenor Research, and since 1995 an associate professor at the University of Oslo. His research interests are in software engineering, empirical studies and process improvement. Since 1998 he has been leader of a software development process improvement group at Storebrand.

e-mail: magne.jorgensen@storebrand.no

Automation approach to software quality

ARVE MEISINGSET

This paper provides an approach to increasing software quality by reducing the amount of application specific code. A technique is provided to measure the amount of code of a given application, the amount of functionality provided by the application, and to measure differences between various implementations of the same application. This technique leads to a focus on parameterisation and management of code by repository editing tools. We have little experience on using the approach outlined in this paper. Rather, the paper summarises an approach used by the author for comparison of case tools [1] for large database applications back in 1991. The author does not know the applicability of the approach for other application areas; however, as even real time systems, e.g. for telecommunications, become more data centred, the approach may very well be applicable. The approach is proposed due to disbelief in the Mk II function point approach – even if this approach is developed for a related but somewhat different purpose. The use of the in-house case tool DATRAN [2] in Telenor has proved effective to remove software errors by reducing application specific code and moving generic code to the tool. This way, the generic code will be extensively tested in varied usage, and will be untouched by the application developers.

1 Overview and rationale

Suppose the number of errors made in a software program is proportional to the length of the symbolic code of the program. The number of remaining faults in the program is supposed to be exponentially decreasing with the time spent to identify and remove the errors. What is then a good strategy to keep the number of errors low for a given application? This paper provides one approach to this problem. The term *error* can here be understood as any kind of misbehaviour of the program, i.e. behaviour that is detrimental to the usage of the program.

Errors can be identified by a programmer inspecting, testing or by other means validating the program. The existence of errors can also be observed by end users testing or using the program. Validation need not be a focused activity, as error detection can be like solving criminal cases: The public is our best detective. A large number of users on a varied set of usages can be a secure – but not always appropriate – means of identifying errors. Each means of validation can have a different efficiency; however, the number of unidentified errors will hopefully decrease – maybe not monotonously – as the time spent on validation increases.

As the number of errors increases with the length of the program and decreases with the time spent on validation, the number of errors for a given application can be kept low by

1 Minimising the length of explicit program code developed for that application;

2 Reusing generic program code that is tested through extensive use of other applications.

The first item can be interpreted as a goal, which can be achieved by the second item as a means. Reuse of generic program code can be obtained by use of inheritance of object-oriented languages. However, there are many other means to reduce program length and to improve program structure. Inheritance can be a lazy approach to program length reduction without providing an appropriate structure. [3] explains that in addition, an appropriate software architecture is needed to harvest the benefits. A theory of software quality by code reduction requires

1 A measure of program length or size relative to the functionality provided by this program;

2 A means to compare the lengths of two programs providing the same functionality;

3 Techniques to remove application specific code.

For bullet 1 we need an understanding of what is a program unit and what is a functionality unit. These notions will be addressed in section 2. Sub-section 2.3 provides a short presentation of Mk II function points, and some of the criticism to this approach. For bullet 2 we need an approach for how to identify and compare differences. The point here is that a totality measure may provide too much uncertainty to provide useful results, while incremental comparisons of alternative programs for the same application can provide a better understanding. This is discussed in section 3. Section 4 provides an overview of alternative techniques to reduce program lengths, i.e. bullet 3. The above discussion on inheritance relates to this third topic.

2 Program and functionality notions

2.1 Notions

The role of a software entity, of whatever complexity, can be illustrated as in Figure 1a. The entity can be a statement, an object, a module, a process, a function, a system or other. This software entity we call a program – independent of the completeness of the code making up this entity.

The software entity prescribes the processing of the input data, their constraints and derivations of intermediate and final output data. This is depicted in Figure 1b.

Output data may be derived directly from input data or from intermediate data, which again may be derived from other intermediate data or input data. These transformations are explicitly specified by the program or implicitly prescribed by the functioning of the compiler or by references to other programs.

Input data instances can be inputted and executed into output data instances. Each data instance must comply with a permissible form generated from the pro-

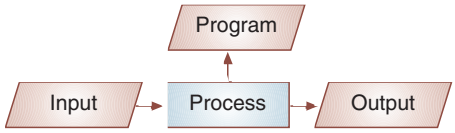


Figure 1a Software prescribes the transformation from input to output data

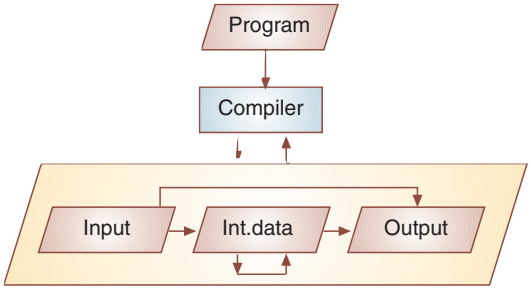


Figure 1b Software prescribes the mappings from input to output data

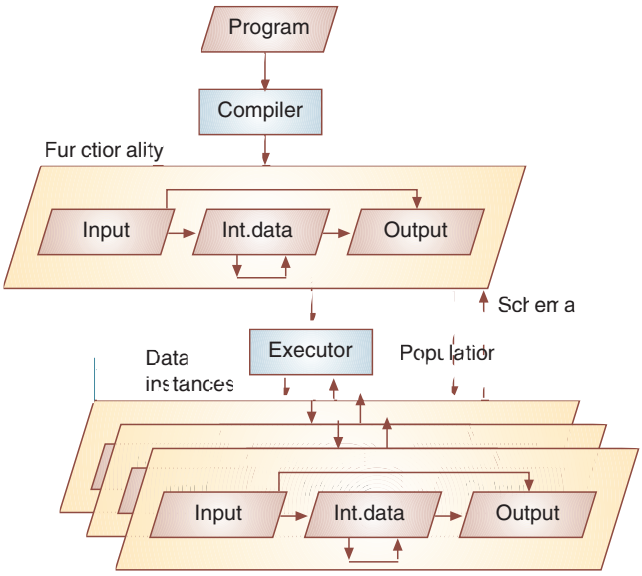


Figure 1c Transformation of data instances as prescribed by the functionality of a system

gram by the compiler. The executor process enforces this compliance and prescribed derivations. See Figure 1c.

The executor controls that each data instance has one of the permissible forms and produces all prescribed derived data instances from this data instance. We imagine that the collection of all permissible and prescribed forms of data is produced by the compiler. This collection is called the functionality of the application system.

The functionality defines schemata whose contents are homomorphic to the instance data. This means that for each data instance there is one form that is identical to that data instance, for each form there can be several instances; instances include references between instances. Each form is called a class relative to each of its data instances, and a data item is called an instance relative to its class.

A collection of instance data is called a population relative to its schema, and vice versa. A population together with its schema is called an application system. A schema separated from its population is called a software system. The functionality of the schema expresses the intention of the application system, while the data instances of the population express the extension of the application system.

Note that in a real software environment, the compiler may not precompile the complete homomorphic schema of the data instances. Frequently certain aspects of the executor and compiler are executed at run time. In an interpreter both the executor and the compiler are executed at run time.

The functionality of an application system states the permissible inputs and outputs and the functional, i.e. many-to-one, mappings from inputs to outputs. The functionality does not include instructions to the executor on how to perform its instructions. These execution aspects are considered to be included in and hidden by the executor.

For a given population we can imagine several alternative schemata that provide the permissible forms and derivations exhibited in that population. Hence, the functionality notion is so far not uniquely defined. Therefore, we seek something like a minimal schema for a given population; however, we are hesitant to define minimum in the strict sense, *as we are seeking the minimal schema for all perceivable populations, which are defined by the schema itself*. Therefore, we may compare alternative schema formulations, compare their permissible populations and derivations, and find the smallest schema if they are equivalent with respect to the studied population; but we may neither be able to find a theoretical minimum nor define what this means relative to something that cannot be defined. See section 2.3 on task dependent formulations.

Note that a schema prescribes the permissible data instances. A prescription of a set of permissible data instances we call a specification. Both data instances and schema data are inscribed, i.e. written down, while this paper does not discuss description mappings from data to phenomena denoted by the data. Many texts frequently misuse the term description and its synonym term 'model'. These terms should only be used in very rare cases, and then with great care. See for example [4].

2.2 Metrics

Automation degree (*AD*) is defined as the ratio between functionality length (*FL*) and program length (*PL*). These two notions will be defined in the subsequent text. The ratio can be any positive rational number, i.e. both smaller and larger than 1.

$$AD = FL / PL \qquad (1)$$

As the objective is to provide high automation, the automation degree should be made as large as possible, i.e. the program length should be reduced for a given functionality.

The length (*PL*) of a program we define to be the number of word inscriptions of that program. Constants, variables, function symbols, commands, operators, logical connectives and other reserved words are counted as word inscriptions. Note that if identical word inscriptions appear several times, for example several addition operators in one statement, they are counted as separate word inscriptions; duplicates are considered significant. Fields of tables and forms, icons and lines of graphic interfaces, and dictated words in sound are counted as word inscriptions, as well. Finally, editing commands, menu selections and clicks on icons are counted as words – double clicks are counted as (two letters of) one word. The need to provide a general technique which applies across various presentation formats and media is one reason for counting word inscriptions rather than tokens or pixels/bits. The count comprises only explicitly inputted words, and not words provided from the tools to the programmer, or otherwise automatically generated code.

Some editing tools allow various input sequences, resulting in different word counts for the same program. The count only applies for the sequence inputted in

the particular case. Thus, functionality over program length can be improved by finding a shorter editing sequence for creating the same program. This is particularly relevant when the program is put in a database repository. Then various overlapping views of the program can be defined by selections, projections and unions over the universal relation (table) making up the program. Editing may be done in any of these views. Careful design, choice and use of views in the best sequence can improve the ratio of functionality over program length. Hence, the word count is not only dependent on the program itself, but on the particular use of the editing tool provided to create the program. Note that the viewing mechanism allows presentation and reading of many overlapping views whose sum of word counts can be much larger than the word count of the explicitly inputted data.

The functionality of the application system is defined by a collection of classes and references between these classes that are homomorphic to the data instances. References [5], [6] and [7] define a language satisfying the homomorphism requirement. The functionality length (*FL*) is measured by word count of all classes of the functionality. The count comprises the entire syntax tree of the functionality, i.e. the number of object classes and attribute group classes and attribute classes and value classes and reference classes, etc., as well as all presentation forms, i.e. views, and their syntax trees on various media, including directives and commands on the presented data. Note that the count comprises all nodes of the data tree, and not only the leaf nodes. Each permissible individual full value is considered to be a leaf node.

The language for expressing the functionality – and most other languages – will allow for stating the same fact in various ways. Therefore, some kind of normal form is needed for comparison of word counts. This normal form is dictated by the homomorphism requirement, which implies that all functors, including inheritance, type specification, operators and connectives are executed into function values that are homomorphic to the data instances. Hence, the inheritance and type statements are not counted in the functionality length, but are counted in the program length. This way, ingenious use of various kinds of inheritance and type statements are important, but not the only means to improve the automation degree.

The basic automation degree (*AD*) expresses the ratio between the total word counts of the functionality (*FL*) and the actual program (*PL*) to provide this functionality. Other indicators may be more appropriate for practical usage. The attribute automation degree (*a-AD*) disregards the count of constant values of the functionality (*a-FL*). In this case individual attributes are considered to be leaf nodes.

$$a\text{-}AD = a\text{-}FL \,/\, PL \tag{2}$$

The kernel length (*KL*) measures the number of words of the (application) data schema [5], and does not count the (external or internal) presentation forms on various media or directives and commands on the presented data. The kernel automation degree (*k-AD*) measures the ratio between the kernel length (*KL*) and the total program length (*PL*).

$$k\text{-}AD = KL \,/\, PL \tag{3}$$

The kernel scope can be combined with the attribute level of detail, e.g. into the attribute kernel automation degree.

$$ak\text{-}AD = a\text{-}KL \,/\, PL \tag{4}$$

The object-reference length (*or-KL*) counts the number of object classes and references in the application schema, excluding count of attribute groups, attributes, values and behaviour specification. The object-reference automation degree (*or-AD*) measures the ratio between the object-reference length (*or-KL*) and the total program length (*PL*).

$$or\text{-}AD = or\text{-}KL \,/\, PL \tag{5}$$

The measures are becoming increasingly more high level from (1) to (5). This means a more inaccurate and easier count.

2.3 Alternative metrics

Unadjusted Mk II function points (*ftp*) [8] is defined as

$$ftp = k_4(k_1 i + k_2 d + k_3 o), \tag{6}$$

where i is the number of insert fields, i.e. insert, modify or delete, d is the number of object classes/entities, o the number of output/read fields, and k_i are proportionality constants. *ftp* is a number that is supposed to be proportional to the total amount of development work, and the k_i-s are based on a suit of test cases. The *ftp*-s are estimated per end user task and summarised across all tasks to be supported by the software developed by the project. The idea is that functionality should be measured from the end user perspective and not for a specific design.

There is a basic difference in the counting of *FL* versus *ftp*. *ftp* is based on counting of variable input fields, object classes and output fields only. In *FL*, headers – corresponding to object class labels, attribute group labels, etc. – are considered superior nodes of the data values found in the data fields. Hence, attribute classes, attribute group classes, object classes and recursively superior object classes are counted in the *FL* estimate, and not only the leaf data value fields of the generated screens as in *ftp*.

In *FL* estimation, each screen and each screen definition is considered to make up an entire syntax tree. The screen definition is called an external schema [5]. The external schemata are created by navigation through the syntax tree of the application schema data structure. All permissible presentation forms are defined in the application schema. The application schema can be split into sublayers. The permissible presentation forms to the end users are defined in the external terminology schema [7].

We observe that while *FL* counts the length of the entire syntax tree, *ftp* basically counts the number of variable fields in the generated application only. The *FL* syntax tree appears as variable fields (both input and output) within the software development environment, while *PL* appears as input fields only in the software development environment.

There are a number of other difficulties with the *ftp* approach:

1 The constants, k_i, are estimated for a set of applications, methods and tools used to develop these applications; the constants and their relative sizes may not be appropriate for other applications, methods and tools.

2 The tasks are assumed to exist independently of the design of the system, however, this disregards the fact that tasks are frequently created to manage data; hence, data and tasks are created interactively, and there may be many alternative tasks to provide the same management of data.

3 The focus on tasks invariably leads to tailoring the human-computer interface functions to each task; this may lead to a large set of simple functions rather than a small set of generally applicable functions.

4 Summation across tasks may lead to a large function point estimate compared to an estimate for a general query-by-example like interface, which provides more flexibility and power, but for which *ftp* does not apply.

5 The use of object class count only does not distinguish data structures having different complexity. This shortcoming may partly be accounted for by the *i* and *o* numbers; however, the basic *ftp* does not provide means to distinguish different behaviour complexities.

6 During analysis and early development, the designer should aim at harmonising and reducing the number of data classes, presentations and functions; however, this would lead to reducing the *ftp* and the estimated efficiency, i.e. *ftp* per working hour.

7 Even if the *ftp* count is very detailed, *ftp* per working hour gives no indication of what aspect of the total project work – analysts, managers, programmers, users, tools, applications or other aspects – caused the good or bad efficiency.

8 Comparison of *ftp* per working hour for different applications may not be a very meaningful exercise; however, it may be used as a crude indicator of the development efficiency if it is followed by more detailed analysis and interpretation.

9 *ftp* may be used for estimating the resources needed to undertake the development work; however, this implies that most of the analysis work is done outside the project, which means that the uncertainties and difficulties are moved outside the project.

3 Measurements and their usage

To count the number of words in a program can provide a reasonably unique result. However, the measurement can become considerably more ill-defined when measuring the length of a 'program' created from a form filling or graphic dialogue. Estimation of the length of the functionality provided by the program can be even harder to overview and, therefore, uncertain.

We assume that the mean estimation deviation is proportional to the estimated length.

$$\Delta PL = k_{PL} PL \tag{7}$$

$$\Delta FL = k_{FL} FL \tag{8}$$

$$\begin{aligned}\Delta AD &= (FL \pm \Delta FL) / (PL \pm \Delta PL) \\ &\quad - FL / PL = FL / (PL \pm \Delta PL) \\ &\quad \pm \Delta FL / (PL \pm \Delta PL) - FL / PL \\ &\quad \approx FL /(PL) \pm \Delta FL / (PL) \\ &\quad - FL / PL = \pm \Delta FL / PL \\ &\quad \text{if } \Delta PL << PL\end{aligned} \tag{9}$$

$$\begin{aligned}\Delta AD &\approx \pm \Delta FL / PL == \pm k_{FL} FL / PL \\ &\quad = \pm\, k_{FL} AD \text{ if } PL << PL\end{aligned} \tag{10}$$

$$\begin{aligned}\Delta AD &= k_{AD} AD \\ &\quad \approx \pm\, k_{FL} AD \text{ if } PL << PL\end{aligned} \tag{11}$$

If we compare lengths or automation degrees of two separate programs, then the estimation deviation becomes the sum of the individual deviations.

$$\Delta(PL_1 - PL_2) = \Delta PL_1 + PL_2 \tag{12}$$

$$\Delta(FL_1 - FL_2) = \Delta FL_1 + FL_2 \tag{13}$$

$$\begin{aligned}&\Delta(AD_1 - AD_2) = \Delta AD_1 + \Delta AD_2 \\ &\text{if } \Delta PL_1 << PL_1 \\ &\text{and if } \Delta PL_2 << PL_2\end{aligned} \tag{14}$$

$$\begin{aligned}&\Delta(AD_1 - AD_2) / (AD_1 - AD_2) \\ &= (\Delta AD_1 + \Delta AD_2) / (AD_1 - AD_2) \\ &\text{if } \Delta PL_1 << PL_1 \\ &\text{and if } \Delta PL_2 << PL_2\end{aligned} \tag{15}$$

For small differences $(AD_1 - AD_2)$ the estimation deviation $(\Delta AD_1 + \Delta AD_2)$ will become large relative to the difference and make the comparison, i.e. the difference estimate, invalid. Therefore, we recommend the following approach:

1 *Incremental comparisons.* Lengths should only be compared for different programs implementing the same or for overlapping applications.

2 *White box.* The major differences between programs implementing the same or overlapping applications should be identified and compared individually.

3 *Automation impact.* The total impact of the use of the programs should be estimated.

The incremental comparison in bullet 1 makes the comparison relevant, and ensures that only comparable programs are compared. On the other side, Mk II function point is typically used to compare development of unrelated applications.

The white box approach in bullet 2 ensures focus on major differences in the implementation of the same application, and provides estimates for these differences only. This way, the deviation estimates become proportional to the length of the differences and not to the total lengths. Bullet 2 is a white box approach, where the differences must be identified and understood before the estimates can be made. Mk II function point is a black box approach, which is used to compare development efficiency of unrelated applications.

The automation impact is studied in bullet 3 and ensures that the total effects of implementation differences are identified and estimated. This typically involves a cost-benefit analysis, which can be done in any traditional way, and is outside the scope of this paper. Mk II function point compares total development projects, and not only the automation aspects. The problem with the Mk II function point approach, as is the case with measures on any social system, is that the system tends to give you the answers you want, but you do not know what aspects of the system have been affected. Mk II typically compares function point efficiency and not cost-benefit.

Having listed several difficulties with Mk II function point analysis, some benefits should be listed, as well: Mk II seems to provide better estimates of project efforts than expert estimates. There can be several reasons for this result:

1 Mk II provides empirical and quantitative results, which can be reused when estimating new projects.

2 Mk II requires that the input-output functionality and other aspects of the project are carefully identified before estimates can be made, this way providing better data for estimation.

3 The required early data collection may lead to moving more analysis work before project start than with other approaches which incorporate the uncertainties into the project.

4 Project workers may tend to deliver as expected when Mk II function point analysis is made and the estimation results are known.

5 Mk II may lead to a certain task oriented design, and may avoid data ori-

ented or tool oriented designs which may involve more uncertainty.

6 Mk II is used for projects where it fits, but is not used for other projects which are more complex and uncertain, and this way provides a biased result when comparing total estimation accurateness.

4 Techniques to remove errors

Techniques to modify program and functionality lengths are application dependent, as some program architectures are appropriate for some applications and not for others. This section provides some application and implementation dependent techniques to manipulate the lengths, and hence the automation degree of a program.

1 *Unification of input-output.* If input and output data are provided at the same medium, e.g. the human-computer interface, they can be unified into one common definition. Hence, an output picture can be used for various inputs by accepting editing of the data fields. The unified picture will comprise one word inscription for the data field and one for the permissions, with the four value words *read*, *insert*, *modify* and *delete*, altogether six word inscriptions. If these functions were separated into four different pictures, three word inscriptions (two fields and one permission value) would be needed for each picture, giving twelve word inscriptions altogether. The unification provides the same functionality by a reduction of this program piece from twelve to six word inscriptions, and improves the automation degree for this program piece by a factor of two.

2 *Centralisation of data definitions.* Suppose an object class definition is needed in four screen pictures and two batch programs, altogether six presentations. If the definition is centralised into one application schema, the functionality remains unchanged, while the program length is reduced to one sixth for this piece of code used to provide the object class definition. Hence, the automation degree is increased by a factor of six.

3 *Data-driven approach.* A third generation language typically states instructions to the underlying machine, e.g. to transform or move data from input to the application, from the application to the database, etc. In a data driven approach, only the relationships between external application and internal forms of data are stated in the schemata in a repository. An application independent executor transforms and moves the data along these relations between all media without any application dependent instructions, as the instructions are built into the generic executor. This way, the system becomes parameter-driven by data in the repository rather than by instructions. The total effect on program length can become considerable.

4 *Default implementation.* Suppose the application schema (in bullet 2) comes in addition to the internal schema (of the database). For the calculation we assume that the application and internal schemata are identical, except that the class labels are different. Hence, for each word inscription in the internal schema, there is an extra word in the application schema, and an extra word is needed to refer one way between the two. The internal schema increases the program length by a factor of three. However, if the internal schema is generated automatically from the application schema, the total automation degree is unchanged. The very purpose of the internal schema is to allow various implementations without affecting the application. Also, internal schemata can be used to define various communication interfaces.

5 *Distribution of internal data.* There can be several reasons for creating a centralised application schema. Centralised definition (a) of user oriented class labels (as in bullet 3) can be one reason. If (b) a multi-valued attribute group has to be stored in a separate fixed-length record type, then a one-to-many mapping to the internal schema is needed. If (c) the application data are stored in several databases or are communicated to other systems, this could be another reason for having a centralised application schema – rather than mapping directly from internal to external schemata. Suppose (d) that one class is stored in two databases and is communicated to one other system; this object class is presented in four pictures. With the centralised application schema one extra word inscription is needed for the class label, two for references to the internal databases, and one for the reference to the communication interface; altogether four word inscriptions. In addition comes four references from the pictures to the application schema, which makes a total of eight word inscriptions. Without the application schema, references are needed from each screen picture to both the internal databases and the communication interface, i.e. four times three make

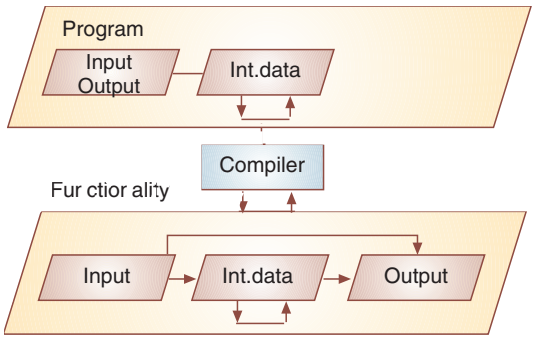


Figure 2a Unification of input-output

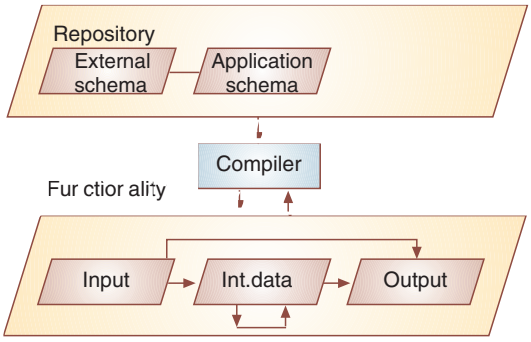


Figure 2b Centralisation of data definitions

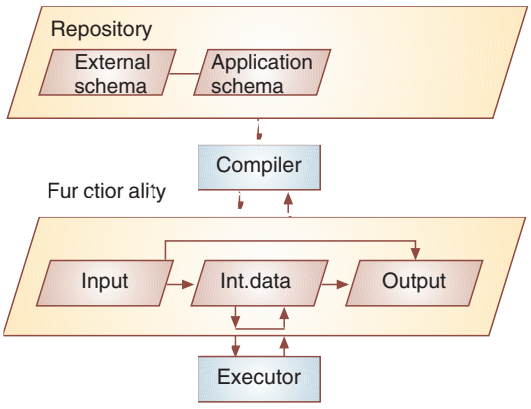


Figure 2c Data-driven approach

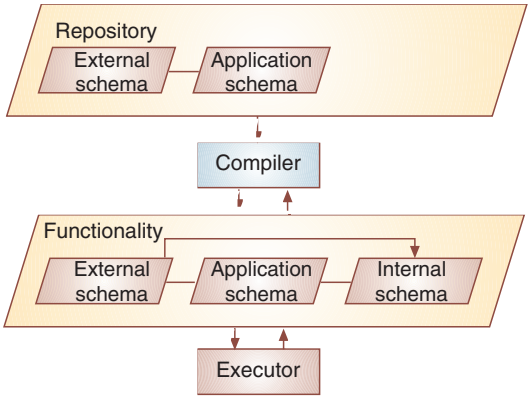


Figure 2d Default implementation

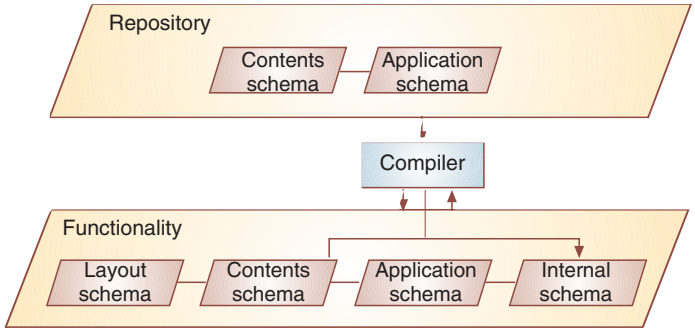


Figure 2e Default generation of layout

twelve word inscriptions. We observe that the centralisation of the application schema has reduced the program length from twelve to eight word inscriptions, while the functionality remains unchanged. For (e) an application having few overlapping pictures, the application schema can reduce the automation degree. For an application having a large fan-out to internal and external schemata, the introduction of an application schema can increase the automation degree.

6 *Default generation of layout.* Many Case tools include a screen painter, and the screen fields are mapped to the data definitions, typically in a database schema. If layout and contents of screens are separated, with a mapping between them, this will triple the program length for this aspect of the system. However, if the layout is generated automatically according to a default presentation style, the program length is reduced to the original length. Also, if the presentation style is default, all the manipulation means can be harmonised and common for all these presentations. This can provide much more functionality than provided by a tailored layout design. In addition, the designer can be freed from the detailed layout design, which would have required a greater program length without improvement of the functionality.

7 *Inheritance and type definitions.* The mapping from the application schema to a contents schema (in bullet 5) is already a sub-setting mechanism. Therefore, inheritance between object classes is not needed in most cases where the contents schema notion is provided. Rather, the inheritance is replaced by an ordinary relationship between the 'subclass' and the 'superclass', and they are both instantiated separately. The contents schema can then present information from both. The subclass notion of object classes can reduce the program length, but provides less overview and flexibility than the relationship notion combined with the contents schema view mechanism. Therefore, the minimum length should not always be strived for. At the attribute level, however, use of common data types is the only practical and most efficient means to define common value sets. Note also that encapsulation of attributes within object classes is not appropriate when wanting to make selection and projection via a contents schema. Behaviour definitions should be provided subordinate to data object classes, but can be split on object behaviour, attribute behaviour, value behaviour etc., and should not be centralised to object classes only. The dispersal of behaviour to where it belongs reduces the need for references between behaviour specifications and the affected data.

Arve Meisingset (50) is Senior Research Scientist at Telenor R&D. He is currently working on information systems planning, formal aspects of human-computer interfaces and middleware standardisation. He is ITU-T SG10 Vice Chairman and the Telenor ITU-T technical co-ordinator.

e-mail: arve.meisingset@fou.telenor.no

References

1 Meisingset A. *Økonomiske effekter ved bruk av systemutviklingsverktøy.* Kjeller, Telenor R&D, 1991. (R&D report R 18/91.)

2 Nordlund C. The DATRAN and DIMAN tools. *Telektronikk,* 89 (2/3), 104–109, 1993.

3 Lauesen S. Real-life object oriented systems. *IEEE Software,* 15 (2), 76–83, 1998.

4 Meisingset A. Three perspectives on information systems architecture. *Telektronikk,* 94 (1), 32–38, 1998.

5 ITU. *Data oriented human-machine interface specification technique : scope, approach and reference model.* Geneva, 1993. (ITU-T Recommendation Z.352, 03/93.)

6 ITU. *Draft recommendation Z.35x and appendices to draft recommendations.* Geneva, 1992. (CCITT COM X-R 12.)

7 Meisingset, A. *The HMI specification technique.* Kjeller, Telenor R&D, 1996. (R&D report N 54/96.)

8 Jørgensen, M, Bygdås, S S, Lunde, T. *Efficiency evaluation of CASE tools : method and results.* Kjeller, Telenor R&D, 1995. (R&D report R 38/95.)

The impact of software reuse on software quality

SVEIN HALLSTEINSEN

Building software from standard reusable parts is supposed to bring about significant improvements in terms of increased productivity, shorter time to market and better quality. However, to achieve the necessary degree of reuse to see such effects, new software engineering practices are required. In this article we discuss how increased reuse and the software engineering practices required to make it happen influence software quality. We conclude that positive effects on many aspects of software quality are to be expected and present some experience data backing this conclusion.

1 Introduction

Software reuse is an approach to the building of software based on extensive use of common prebuilt components. This principle has been adopted by many other industries, generally leading to cheaper and better products, and it is now slowly making its way into the software industry.

The benefits of reuse seem obvious. Firstly, building from pre-built components means there is less work to do to develop a particular application, leading to shorter time to market. Secondly, since the cost of developing the components can be shared by many applications, one must expect significant cost reductions. Thirdly, since major parts of a new system have already been tested and debugged in other systems, one should expect better quality.

At the time of writing, extensive reuse is still not common practice in the software industry, but over the last 10 to 15 years we have seen many examples of successful attempts to adopt this approach. Some companies have achieved more that 80 % reuse in typical new applications, and in parallel with this have more than doubled their productivity and halved time-to-market and error density[1)].

This article discusses the impact of software reuse on the quality of software products. First we give a brief introduction to what reuse oriented software development means, and introduce our definition of software quality. Then we present and substantiate common beliefs regarding the effect of software reuse on software quality. Finally we look for confirmation of those beliefs in the experience gathered by companies who have successfully adopted software reuse.

2 Software reuse

Software reuse is widely recognised as one of the major sources of cost savings in the software industry in the next few decades. Figure 1, presented by Barry Boehm at the STARS conference in 1991, predicts the relative importance of the three major sources of expected savings in software development:

- Working faster (due to better tools);
- Working smarter (due to better processes for software development and better control over the processes by estimation, planning, assessment and improvement);
- Work avoidance (due to increased reuse).

The baseline total is the expected expenditure without any improvement in software development technology.

This optimistic view about the benefits of reuse is reflected also by Capers Jones, author of many articles and books about software quality and productivity and software development process improvement. In [1] he states that:

> *"Full software reusability programs tend to have the highest return on investment of any technology since software began (about $30.00 returned for every $1.00 invested)".*

Such claims have led to high expectations towards reuse and the literature does report many success stories. However, there appears to be many failure stories as well, although these are seldom published. Therefore the adoption of reuse by the software industry is slower than one might expect considering the expected benefits.

The reason is that although the basic idea of software reuse is simple, making it work in a given organisation seems to be more complex than anticipated initially, and requires extensive changes in the entire software engineering process as well as mastering enabling technologies [2, 11]. Some of the issues involved are briefly discussed below.

Ad hoc reuse: This simplest form of reuse happens when developers copy arbitrary code parts from existing systems and insert them in a new system, possibly after some modification to make them fit. In addition to code, requirements specifications, design specifications, test specifications and user documentation, or generally any work product from the development process, can be reused in the same way.

Some programmers are very good at exploiting own work in this way, but

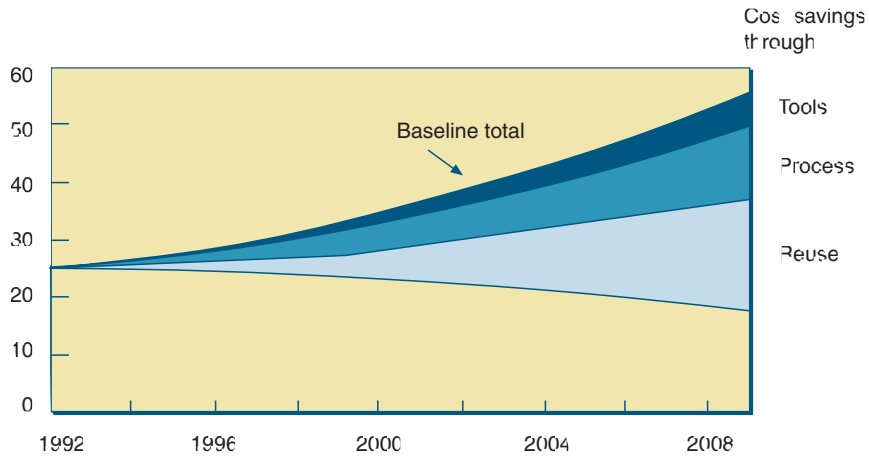


Figure 1 Projected software development cost savings

[1)] *We will use the term software industry in this article although many will claim that software building is more craft than industry.*

few are able to reuse other people's work. Another major shortcoming of ad hoc reuse is that there is no recording of the connection between the origin and the reused copy. Thus, there is no notion of common parts and therefore no possibility to share maintenance and evolution costs.

An extreme form of ad hoc reuse is when the entire system is copied, that is the new system is made by modifying an existing one. This has been quite successful in some cases but the weaknesses described above persist.

Component reuse: This is a more sophisticated approach, where the parts that are being reused are identifiable and well defined units with a common source that are shared by the systems in which they are used. Thus, common evolution and maintenance become possible.

This approach goes hand in hand with the idea of modular design, such that existing systems really contain components in the sense described above. Components that are believed to be reusable are extracted from their mother systems and collected in libraries with appropriate search facilities to ease retrieval of suitable components.

Unfortunately, these efforts did not always bring about the expected level of reuse. Much attention was devoted to sophisticated classification and search schemes, but the problem rather seemed to be that most components harvested as described above were not suited for reuse. They typically suffered from poor quality, poor documentation and lack of generality and reuse was therefore not found worth while.

Reusable components: This led to the idea of reusable components, that is components built explicitly for reuse, with sufficient generality and quality that they would be found useful and worth reusing in many applications and documentation made explicitly to support ease of reuse. However, to build a good set of such components turned out to be rather difficult.

Firstly there is the problem of finding out which components to build, that is to identify the recurring sub-problems that occur often enough to justify a reusable solution. There is of course a number of fairly fundamental things like commonly used data types, algorithms and user interface elements, that are not too hard to identify. Although such components make up a significant part of any application, the major part and the most difficult part to build, is usually that part that deals with the problem domain specific issues.

Secondly there is the problem of defining stable and flexible interfaces to the components, both in terms of provided and required resources, that will make the component easy to integrate into different future systems.

Domain analysis: Domain analysis has proved to be a valuable tool to identify and specify domain specific components. Domain analysis in this context means to analyse an entire problem or business domain, typically the domain for which a company wants to make applications, in order to improve domain knowledge and to make it explicit in the form of abstract models. Important issues are to capture commonality and variation of the needs of the users in that domain and how this will evolve over time. The idea is that equipped with this higher level of domain knowledge, it will be possible to identify and build truly reusable components for the domain [13].

Standardised architecture: Easy integration of components into different systems require a certain level of agreement on the interface to basic services such as persistence, distribution, communication, concurrency, transactions, exception handling, etc., which many components will depend on. Therefore a standardised and stable architecture seems to be necessary to ensure true reusability of components.

In-house standards may work for own components, but for use of third party components to become viable, more widely accepted standards are necessary. At present we see the emergence of several such standards, the most noteworthy being OMG's CORBA, Microsoft's ActiveX and Sun's Enterprise Java Beans, that are believed to pave the way for an extensive component industry.

Domain Specific Application Frameworks: The combination of a standard architecture and a set of domain specific components are usually termed a domain specific application framework. In well-understood and stable domains, this approach may yield very high degrees of reuse.

Standardised domain models: In some domains that represent major markets for the software industry, for instance accounting and finance, commerce, and health care, there are also attempts to standardise domain models and interfaces to domain components. Both OMG and Microsoft, among others, are working on such standards and this is expected to further stimulate the growth of a component industry.

Reuse oriented process and organisation: A different kind of obstacle to the success of reuse has been that traditional software development process and organisation models do not really encourage reuse. Reuse introduces new tasks and responsibilities into the software development business, for instance domain analysis, component development and component library management, and therefore requires new process and organisation models. These tasks tend to transcend individual projects and need to be managed and funded from a wider and longer-term perspective. What is required may be resembled with business process reengineering, applied to the business of software development [12].

The new way of business is more complex than before and therefore seems to require a higher level of process maturity than what is common practice in the software industry.

It appears that the companies that have obtained the best results with respect to reuse are those that have been willing to put up the necessary up-front investment to resolve all these issues and that have had patience to wait for the benefits to appear.

3 Quality

Software quality is normally defined in relation to a set of generally desirable properties that a software product may have, for instance

- absence of failure;
- satisfaction of user needs;
- ease of learning and using;
- ease of maintenance;
- easy of evolution;
- performance;
- scalability.

Traditionally, the common understanding of quality has been that the better the product scores with respect to such properties the better the quality. Our concern in this article is to investigate how such typical quality attributes will be affected by transition to more reuse oriented software engineering practices.

4 Expected effect on quality

Based on the presentation of reuse given above it is possible to reason about how this way to build software will influence typical quality attributes.

Absence of failure: As explained in Section 2 reusable components have to be built according to a higher quality standard than what is normal for one-shot software. Otherwise it is unlikely that they will be reused at all. In addition, successful components will have matured through several previous reuses. Thus, one must expect that the error density in reused parts is significantly lower than in custom-built parts. With reuse rates between 50 and 80 %, which is commonly achieved in successful reuse programs, one must therefore expect a significant effect on the error density in the total application.

Satisfies user needs: Domain analysis is supposed to lead to better understanding of the user needs of a domain, and this is likely to lead to systems that better match the needs of the users.

Easy to learn and to use: The above argument pertains to the ease of learning and using as well. Domain analysis will lead to consistent mental models that the user interface can build upon, making it easier for the user to understand and predict how the system will behave in different situations. Reuse of user interface elements will contribute in the same direction by enforcing a common look and feel between different systems.

Ease of maintenance and evolution: A system built to a large extent from reusable components is bound to have a clear component structure with well defined and well documented interfaces, which is generally accepted to be good for making changes. Another property of reusable components that plays an important role in this context is that they are often built to handle the variability of the domain, both between different users and over time. Therefore components will often already be prepared to support an evolutionary change. Finally there is the effect of shared maintenance. When an error is detected in a reusable component, its correction will potentially benefit a number of systems.

Performance: When it comes to performance there are two conflicting arguments: On the one hand, the need for generality in reusable components may lead to performance problems. This is often used as an argument against reuse in domains where performance is particularly important. On the other hand, more effort can be put into optimisation and tuning of reusable components than can normally be done in one-shot software development, since the cost can be shared between many uses of the component. Which will be the dominating effect will depend on the domain and the circumstances.

5 Experienced effect on quality

In this section we present some examples of software reuse programs that have collected data about the effects of reuse, and discuss their findings in light of the above arguments.

5.1 Experience at NASA

At the NASA Goddard Space Flight Center (GSFC) a reuse program was carried out and monitored over a period of almost 10 years in their Software Engineering Laboratory (SEL). This lab develops ground based flight support software for space flights. At the same time they co-operate with the University of Maryland on an advanced software process improvement program. The reuse program was one of several process improvement experiments carried out at SEL in this period, and metrics were collected systematically.

The observed trend in quality, cost and reuse in the NASA/GSFC reuse program as presented by Frank McGarry at a seminar at the European Software Institute in October 1994 is shown in Figure 1.

The figure compares data from a group of projects from the first half of the program with data from a comparable group of projects from the second half. In the first period, from 1985 to 1989, the typical percentage of reuse was around 20 %. In the same period the cost of developing a typical system in terms of effort was around 750 staff months, and the average number of errors per KLOC was around 9.

In the second period, from 1990 to 1993, a typical reuse percentage of 80 % was

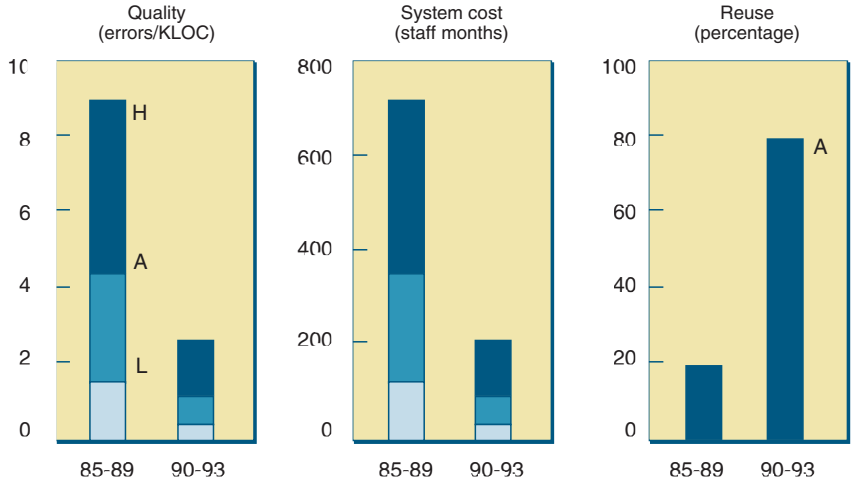


Figure 2 NASA/GSFC quality, cost and reuse data

achieved. In the same period the average cost of developing a system had dropped to 210 staff months and the error density had dropped to 2 errors per KLOC.

The data were collected from 7–8 similar systems in each time period. This does not provide any proof that reuse was the major contributor to these improvements, but other improvements, e.g. changes in tools (CASE) and process (Cleanroom), were introduced and measured in the same period, and neither gave more than limited improvements [2].

5.2 Experience at HP

Hewlett-Packard have staged several successful reuse programs. Experience from two of them is presented in [3].

The first reuse program took place within the Manufacturing Productivity department of HP's Software Technology Division, which produces large-application software for manufacturing resource planning. The program started in 1993 and is still going.

The second program was initiated within the San Diego Technical Graphics Division, which develops, enhances and maintains firmware for plotters and printers. It started in 1987 and continues to the present.

The data collected was analysed and both quantitative and qualitative aspects of the reuse programs were estimated. As a part of this assessment, data on the improvement of quality, productivity and time-to-market were analysed and documented as reproduced in Table 1.

The conclusion drawn by the author for these products is that because software is reused multiple times, the defect fixes from each reuse accumulate, resulting in higher quality. Furthermore, reuse improves productivity by reducing the amount of time and labour needed to develop and maintain a software product.

5.3 Experience at AT&T

AT&T have had several positive experiences with reuse programs. One of them produced a domain architecture and reusable components for telephone operation support systems named BaseWorX [4]. It started in 1987 when it was realised that many variants of this kind of system had been implemented over the years and that there seemed to be a big potential for reuse in the domain. The first two years were spent designing a reusable architecture and a set of large-scale components conforming to this architecture. Then this was matured through pilot use in two projects. In 1993 the reusable assets were used by over 70 projects and a separate support organisation had been set up to act as owners and maintainers of the reusable architecture and components and to provide training and a help desk for the reusers.

The reusing projects achieved reuse rates ranging from 40 % to 92 %, and typical development time dropped from 12–20 months to 6–12 months. Overall the division estimated a cost saving of at least 12 %, after allowing for the cost of developing and supporting the reusable components.

BaseWorX was productised for still wider use, but unfortunately we do not have cost benefit data for the remainder of the life cycle.

5.4 "Laboratory experiment" at the University of Maryland

In [5], Basili et al. report on an experiment with 8 development teams developing the same application in the same type of environment with the same technology, but with varying degree of reuse. Data was collected from these development efforts in order to investigate the effect of reuse.

The development environment was C++. The application was a system for managing home video rental, to be used by home video rental shops. This application was chosen in order to ensure that familiarity with the application domain was equally distributed among the teams. Several relevant libraries of reusable components were available to the teams, but the teams were free to make use of them or not. The teams consisted of students with reasonable experience in software development, and care was taken to ensure that overall capabilities of the teams were about the same.

The findings in this experiment are summarised below:

- Reuse rates ranged from close to zero % to close to 50 %;
- High reuse projects (reuse rates between 40 and 50 %) exhibited a productivity twice as high as that exhibited by the low reuse projects (reuse rates close to 0 %);
- The error density observed in the high reuse projects was about one third of that observed in low reuse projects;
- Rework was much lower in high reuse projects than in the low reuse projects. This appeared to be primarily due to fewer errors. There was no statistically significant indication that errors were easier to find and repair.

5.5 Modern PC software

Anyone who has used a recent Windows or Apple PC will probably have noticed the common look and feel of most applications. For instance, window frames look the same, corresponding information is in the same place, menu and toolbars look and behave similarly and can be configured in the same way. Also the same functionality is often offered by several applications. For instance browsing your file catalogue is the same whether you do it from within the text processor or the spreadsheet application, and if you want to make a drawing in your document, you can invoke a drawing tool inside the text processor that works exactly like the standalone drawing application.

Table 1 Effect of reuse on quality, productivity and time-to-market

Organisation	Manufacturing Productivity	Technical Graphics
Quality	51 % defect reduction	24 % defect reduction
Productivity	57 % increase	40 % increase
Time-to-market	Data not available	42 % reduction

This is all mainly due to reuse. Apple pioneered the idea of a standard user interface tool kit. Microsoft is following with VBX, OCX and ActiveX technologies (basically new names for improved versions of the same thing). ActiveX includes both user interface standards, reusable user interface components and component middleware that supports the integration of these components with own components to form complete applications. Both Microsoft and many third parties provide components, and this is probably the first example of a commercial market for reusable components.

Unfortunately there is little data available from the application of this technology, but the effect on ease of use and ease of learning seems obvious.

Provida, which is a Norwegian company delivering banking software, has developed an applications framework for client applications based on ActiveX technology. These applications typically run on standard PCs and support different tasks in the bank while providing access to the legacy mainframe based central banking system. The system is still in the pilot phase, but initial results indicate that a 50–80 % cost reduction compared to development from scratch is achievable.

5.6 Summing up

The data on the effect of increased reuse on productivity, time-to-market and quality are summarised in Table 2. Despite considerable variation, it appears that there is a clear correlation between reuse rate and both productivity and error rate. Increasing the reuse rate from close to zero to close to 50 %, a 50 % increase in productivity and halving of the error rate will not be unusual. For the time-to-market we only have data from one program, showing a decrease of about 40 %. These results support our hypothesis that increased reuse leads to fewer errors.

In addition to these quantitative results, there are also some less precise observations. HP Manufacturing Productivity claims that reuse eases the maintenance burden and supports product enhancements, but without quantifying the effect. The Bull Workflow project in SER reports better ability to cope with requirement changes due to the built-in generality of the reused components.

Table 2 Summary of data on effect of reuse on productivity, time-to-market and quality

	Reuse rate before (%)	Reuse rate after (%)	Productivity increase (%)	Time-to-market reduction (%)	Reduction in error rate
NASA	20	80	300[2)]		70
HP Man. Prod.	0	68	57		51
HP Technical Graphics	0	32	40	42	24
Other HP firmware div.	5	80	400		
Lab. exp. at Maryland Univ.	0–10	40–50	225		65
AT&T	0	40–92		50	

2) Derived from system cost, assuming that the system size has been stable in the period of observation.

These statements we take as backing for the expected effect on ease of maintenance and ease of evolution.

Finally, the experience with user interfaces on modern PCs confirms our hypothesis regarding the positive effect of reuse on the ease of learning and using.

There are two factors that may explain the variation in achievements between the different projects. Firstly these results have been obtained in different contexts: in different application domains, in different markets, using different development environments, and applying different reuse approaches. Secondly there is the inherent difficulty in measuring the achievements in software projects and the lack of standardised metrics. For instance, the numbers reproduced here have been produced in several different ways:

- In HP Manufacturing Productivity and HP Technical Graphics one compares the productivity and error rate in the entire product (including reused parts) with the productivity and error rate in the parts that were developed new.
- In NASA and the other HP firmware division one compares the productivity and reuse rate at the beginning and end of a period where a significant increase in reuse rate has been achieved, but where also other improvements are likely to have had an effect.
- In the laboratory experiment at Maryland University one compares the results achieved by different teams developing similar applications with different reuse rates.

It seems likely that the remarkably good results in NASA and the HP firmware division are due to the influence of other improvements in addition to reuse.

6 What does it cost

Normally there is cost associated with quality improvements. In one way, this is not the case in connection with reuse, since the quality benefits normally come hand in hand with productivity improvements and reductions in time-to-market that lead to significant cost reductions.

On the other hand, an up-front investment and serious commitment are required to bring about the level of reuse necessary to experience the benefits described above in an initially non reuse oriented development organisation.

Data concerning costs and achieved savings are summarised in Table 3. In addition to data from the reuse programs presented above, we have also included data

Table 3 Summary of Reuse cost and benefit data

Source	Period of observation (years)	Return on investment (%)	Break-even year	Cost to dev. RCs (%)	Cost to reuse (%)
HP Man. Prod.	83-92	410	2nd		
HP Technical Graphics	87-94	216	6th	111	19
Air Traffic Control				200	10 to 20
Ada Menu and Forms				120 to 480	10 to 63
Bull Workflow	91–94			130	5 to 10
Ericsson Radar	93–95	180	2nd	110–130	5 to 25
Ericsson Telecom			2nd	140	35

from the US Federal Aviation Administration's Advanced Automation Systems project reported in [6] (labelled Air Traffic Control in the table), from a menu and forms management system written in Ada reported in [7] (labelled Ada Menu and Forms in the table), and from a few projects reported in the SER experience report [8].[3)]

The most obvious observation to make from these data is probably that there is considerable variation:

- The return on investment for the observed reuse effort ranges from 180 % to more than 400 %. The break-even point ranges from 2 to 6 years. However, around 2–3 years seem to be more common than 6 years.
- The cost to develop reusable components ranges from 10 % extra to several hundred percent extra compared to the cost of developing the same component without considering reuse.
- The cost to integrate a reusable component compared to the cost of developing the needed functionality from scratch ranges from 5 % to 60 %.

Concerning return on investment, there is still a big gap to the 30 to 1 level predicted by Capers Jones. A possible explanation is that the data we have are all extracted rather early in the life cycle of the reuse programs, and therefore start-up costs are still dominating, and the full reuse potential of the developed reusable work products has not yet been exploited. It is worth while noting that the highest return on investment was observed in the program with the longest period of observation.

7 Conclusion

In this article we have argued that the growing adoption of reuse by the software industry that is currently taking place, will lead to better quality software. This is partly a direct effect of the added maturity that new systems inherit from the well-proven components they are built from, and partly an effect of the changes in software engineering practices necessary to succeed with reuse.

We have tried to back our arguments by experience data, but as usual when attempting to assess software engineering technology, there is little data available [10], and the investigations that have been carried out have generally focused on demonstrating cost reductions and reduced time-to-market rather than quality improvements.

The experience data we have been able to find seem to back the common belief that increased software reuse will lead to software products with fewer errors. The data indicate that as reuse becomes common practice, systems where more than 50 % of the code is made up of reused components will be common, and these systems are likely to have less than half as many errors as similar systems developed from scratch.

There is also some basis for believing that they will be easier to maintain and evolve and easier to learn and use.

To make the picture even brighter these benefits come together with substantial cost savings and reductions in time-to-market.

Considering the return on investment achieved by the companies that have succeeded with software reuse, one may ask why something as profitable as software reuse is not already common practice in the industry. The answer seems to be that a number of unforeseen obstacles have turned up, having to do both with technological and managerial/organisational aspects, that have hampered widespread adoption of reuse. It is only for the last 5 to 10 years that one has started to fully understand the full impact of converting a software development organisation to a reuse oriented mode of working.

References

1 Jones, C. Becoming "Best in Class" : the path to software excellence. From the Knowledge Base of the Software Productivity Research WWW Server, *http://www.spr.com.*

2 Karlsson, E A (ed.). *Software reuse : a holistic approach.* Chichester, Wiley, 1995. (Series in Software Based Systems.)

3 Lim, W C. Effects of reuse on quality, productivity and economics. *IEEE Software,* 11 (5), 23–30, 1994.

[3)] *SER (Software Evolution and Reuse) was an ESPRIT project that followed up the deployment of software evolution and reuse technology developed in earlier ESPRIT projects and collected practical experience with this technology.*

4 Beck, R P et al. Architectures for large scale reuse. *AT&T Technical Journal,* 71 (1), 34–45, 1992.

5 Basili, V R et al. How reuse influences productivity in object-oriented systems. *Communications of the ACM,* 39 (10), 104–116, 1996.

6 Margono, J, Lindsey, L. Software reuse in the air traffic control advances automation system. In: *Proceedings of the Software Reengineering and Reuse Conference.* Washington, DC, National Inst. for Software Quality and Productivity, 1991.

7 Favaro, J. What price reusability. In: *Proceedings of the First Symposium on Environments and Tools for Ada.* New York, ACM, 115–124, 1990.

8 Hallsteinsen, S, Paci, M (eds.). *Experience in Software Evolution and Reuse : Twelve Real World Projects.* Research Reports ESPRIT, SER vol. 1, Springer-Verlag, 1997.

9 Bennett, K. Legacy systems : copying with success. *IEEE Software,* 12 (1), 19–23, 1995.

10 Fenton, N et al. Science and substance : a challenge to software engineers. *IEEE Software,* 11 (4), 86–95, 1994.

11 Frakes,W B, Isoda, S. Success factors for systematic reuse. *IEEE Software,* 11 (5), 14–19, 1994.

12 Jacobson, I et. al. *Software reuse architecture, process and organization for business success.* New York, ACM Press, 1997.

13 Hewlett Packard and Matra Marconi Space and Cap Gemini Innovation. *Domain analysis method.* October 1993. (Proteus Deliverable D3.2A.)

Svein Hallsteinsen (51) is Research Scientist at SINTEF Telecom and Informatics, where he has been involved in research and technology transfer in the field of software engineering. His research interests include Software Reuse, Domain Analysis, Object Oriented Modelling, Software Architectures and Component Based Software Engineering. He is currently acting as technical coordinator of Magma, a project led by PROFF aiming to introduce software reuse and component based software engineering to Norwegian off-the-shelf software vendors.
e-mail: Svein.Hallsteinsen@informatics.sintef.no

The role of measurement in software and software development

TOR STÅLHANE

Why do we need measurement

The goal of the software industry – as any other industry – is to deliver products and services to the market. This must be done at a price that the customer accepts as reasonable and at a quality that the customer wants. One of the main characteristics of the market is that it changes, not by itself but through competition among those who provide the goods that are sold. The changes will not happen at the same time for all industries, but only when the players can no longer expand in any other way. The main results of this competition are lower prices and higher quality – not because the producers want it to be this way, but because it is the only way in which they can survive.

The software industry – like any other industry – can improve only in one way: by understanding the product and the process that produces it. Thus, we can define the first goal of software measurement.

We need measurement in software development to improve the development process so that we can increase product quality and thus increase customer satisfaction.

In addition to this long-term need, we also need measurement in order to agree on the quality of a delivered product. The time when the customer watched the flashing lights in awe is gone forever. The customers now want quality, and as a result of this we need to be able to establish a common ground for specifying, developing and accepting a software product with an agreed quality.

Even though we accept the ISO definition of quality as the product's degree of satisfying the customer's needs and expectations, we need something more concrete when we start to write a contract and develop a software system. The definition given in the standard ISO 9126 is a good starting point although it concentrates on the product and leaves out important factors such as price, time of delivery and quality of service. Thus, our second need is

We need measurement on software so that we can understand and agree on product quality.

The rest of this article will focus on the following ideas:

- What are the challenges when we want to improve the software process and the product quality through the use of measurement?
- How can we meet these challenges – what can we do and who will do it.
- What has been done up till now – national and international experiences.
- Where do we go from here – the future for Norwegian software industry.

The challenges

"You cannot understand what you cannot measure", said Lord Kelvin. Some unknown but probably frustrated software engineer has made the addendum "You cannot measure what you cannot understand". Both views contain some truth and all work in understanding the software process and aspects of software quality must relate to both.

The main problem with the idea of the need to measure to understand is simply "What shall we measure to understand what?" To take a practical example: Given that we want to understand the reason why there are so many errors in a software system, what should we measure? The suggestions have been – and still are – legion. The suggestions for software metrics are almost endless, from number of code lines via McCabe's cyclomatic number to Halstead's software science. Unfortunately, none of this has brought us any nearer a good understanding of anything. The reason for this lies in the approach taken. The research has often moved along the following lines:

1 Get some inspiration and define a set of software metrics.

2 Collect the software metrics and the product characteristic that it is supposed to predict, for instance number of errors.

3 Do some statistical analyses – mostly regression analyses although other, more sophisticated methods such as principal component analyses, have also been tried.

4 Look at the results and claim that any metric that correlates with the characteristic that we want to predict is – ipso facto – caused by whatever this software metric measures.

Statisticians call this particular brand of research "shotgun statistics" and consider it a waste of resources. In addition, it lacks one important component, namely control over or measurement of the influence of the environment – both the project's environment and the company's environment in general.

Some solutions

All sound science has started out with the practitioners. The approach has always been to start with the observations and knowledge of the practitioners, then to systematise their experience, deduce a hypothesis and then collect data in order to accept or reject the hypothesis. As a part of this, it has also been considered important to agree on important definitions in order to have a common vocabulary and a common frame of reference. There are several important lessons for software engineering here:

- By starting with the practitioners, we make sure that all or most of the available knowledge is included – or at least considered – when we start to build our models.
- By starting with formulation of hypothesis, we get a data collection process that is goal driven and thus easy to motivate.
- When we reject or accept a hypothesis, we always increase the available body of knowledge.

In this way we are able to accumulate knowledge and models, share them and discuss them with colleagues – in short: create a real software science, or at least a basis for one.

However, one last obstacle has to be surmounted – the fact that software engineering is not a natural science in line with chemistry or physics. Software is developed by people. In some sense, this is part of the environment problem since the strong dependence on the software engineer's skills, experience and knowledge makes it difficult to do repeatable experiments. This means that we must in some way include the vast body of knowledge already available in psychology and sociology. Without this, we are

missing an important component of software engineering and our understanding will forever be incomplete. Even if we cannot model the developers, we need to consider them in our model, for instance as a source of variance or uncertainty.

To sum up, we need to embark on the following program:

1 Collect the available expert knowledge from the software development community.

2 Define terms in order to improve communication.

3 Formulate hypotheses based on available knowledge and experiences.

4 Collect data, perform statistical tests and accept or reject the hypotheses.

5 Incorporate this knowledge into an agreed body of knowledge.

6 Use the body of knowledge to build models that can be used to predict the effect of changes to a development process concerning cost, lead time and product quality.

Do we really need all this? The answer is "Yes". Today, the software industry does not even have an agreed definition of productivity. Considering this, it cannot come as a surprise that we have problems when we discuss if a certain tool or method has increased productivity.

What has been done

Even though much of what has been done in the past can be criticised, the situation is not all bad. In the last ten years, several research communities have come up with important ideas – the ami project, the GQM method for data collection, more interaction with other industries, especially related to the TQM concepts, better knowledge of how to run experiments in an industrial setting, and so on. Unfortunately, the experience gained through the design of experiments (DoE) and the work of Tagushi has largely been ignored by the software industry up till now. The same goes for G. Box' work on analyses of non-repeatable experiments.

The EU initiative, ESSI, has been one of the main driving forces in introducing measurement into software development and improvement in Europe. The ESSI projects – PIEs – have enabled the commission to collect a large amount of data, measurement and experience that someday will hopefully serve as a basis for real research on software processes and software quality.

Below, we have summed up some of the results that we consider to be important from some improvement work we have done ourselves. We will describe three experiments, the reason why the experiment was performed, the strategy used for data collection and analyses and the most important results as seen from the point of view of the software industry. The three experiments can be summarised as follows:

1 A survey of 100 Norwegian companies where the persons responsible for software procurement were asked to rank a set of product and service quality characteristics on a five point scale;

2 Analyses of data from integration and system tests from a telecom product in order to see which factors influenced the number of faults in a software component at the time of delivery;

3 Analyses of data from an experiment with traditional development versus object-oriented development. The important questions were whether object-oriented development leads to fewer errors and a more efficient error correction process.

The PROFF project

The PROFF project was a joint undertaking by several Norwegian software producers. The goal of the project was to improve quality and productivity within this industrial segment. The project had a subproject catering to product quality estimation and control, to a large degree based on ISO 9126. One of the things that we needed to find out at an early stage was how customers ranked the quality characteristics of the ISO model. In order to find out, we performed a survey of 100 Norwegian companies. After having been contacted on the telephone, they received a questionnaire where they were required to give a score to each ISO 9126 product quality factor. In addition, we included a set of service quality factors, adapted form an early version of the ISO standard for service quality – ISO 4100. The analysis was performed by first ranking the results and then applying Friedman statistics to the ranks. The results were significant at the 5 % level.

Overall results

Each responder ranked a set of factors on a scale from 1 (not important) to 5 (highly important). First of all, we looked at the overall score, i.e. the ranking of the factors that we got when pooling together the responses from all price and product categories. The score for each factor was computed as the average of the scores for all criteria related to this factor. This gave the results shown in Table 1.

When looking at the results of this survey we found some rather surprising results – at least they surprised us. The most important finding was that the three most important determinants in a buy / no buy situation was service responsiveness, service capacity and product reliability – in that order.

If we split up the data set according to type of product, we got the results shown in Table 2.

In all cases, the producer service responsiveness and service capacity are considered to be the most important factors, while product maintainability and portability are considered to be the least important ones. The only product factor that consistently scores high is product reliability.

Table 1 Scores for all product categories pooled together

Factor	Score
Service responsiveness	2.81
Service capacity	2.74
Product reliability	2.71
Service efficiency	2.65
Product functionality	2.60
Service confidence	2.60
Product usability	2.57
Product efficiency	2.46
Product portability	2.05
Product maintainability	1.89

Table 2 Scores according to product category

Factor	COTS (36)	Standardized software packages (27)	Customized and tailored software (19)
Product functionality	2.55	2.65	2.62
Product reliability	2.65	2.77	2.74
Product usability	2.67	2.47	2.54
Product efficiency	2.47	2.35	2.58
Product maintainability	1.68	1.94	2.32
Product portability	2.01	2.04	2.09
Service confidence	2.55	2.59	2.67
Service efficiency	2.64	2.67	2.61
Service capacity	2.69	2.74	2.84
Service responsiveness	2.82	2.83	2.76

Statistical method used

We based the tests for the hypotheses on the Friedman statistics, described below. The method can be described by using Table 3.

Each category is given a rank score in the range 1 to 3 – 1 for the most important and 3 for the least important. If one of the categories consistently receive the best score, the rank sum for this column will be close to N, while a category that is consistently given bad scores will have a rank sum close to $3N$. If there are no differences, the score for all the columns will tend to be equal. The formula below is used to compute the Friedman rank statistics. Let N be the number of rows in the table. Let s be the number of categories (columns). We can then compute the Friedman statistics Q as follows:

$$Q = \frac{12N}{s(s+1)} \sum_{i=1}^{s} \left[R_{i.} - \frac{1}{2}(s-1) \right]^2$$

$$R_{i.} = \sum_{j=1}^{N} R_{i,j}$$

The $(s–1)/2$-term is the expected score in the case that all observations – rankings – were completely random. The Q-value thus measures the difference between the expected and the observed scores. As an approximation, we have that Q is Chi-square distributed with $s–1$ degrees of freedom. We will reject a hypothesis of no inter-column difference on the α-level if $Q > c$, where c is the α-percentile in the Chi-square distribution with $s–1$ degrees of freedom.

As an example on the use of Friedman's statistics, we can show the effect of product price on the ranking of the product and service quality factors, see Table 4. We have used three price categories – category 1, 2 and 3. The number in parenthesis below each category is the number of responses that fall in this category. For each price category, we have used the following lay-out: score / line rank column rank

For instance, for the factor 'Product functionality' we have a score of 0.60, line rank equal 3 since this is the lowest score on this line and a column rank of 5 since this is the fifth lowest score in this column. As score values, we are here using the portion of respondents that have given the factor the highest rank. Thus, a score of 0.6 means that 60 % of our respondents have given that factor a rank of 5.

If we compute the Q-statistics according to the formula above, we get $Q = 7.4$. With an α-value of 0.05, we find that we can reject the hypothesis of no difference. A quick glance at the table also shows that the requirements on product and service quality are much lower for the products in the lowest price category.

Table 3 Example of Friedman's rank statistics

Factor	Category 1	Category 2	Category 3
F_1	R_{11}	R_{21}	R_{31}
:	:	:	:
F_N	R_{1N}	R_{2N}	R_{3N}
Rank sum	$R_{1.}$	$R_{2.}$	$R_{3.}$

Summing up

The message from the marketplace was loud and clear – 'Service matters'. What was probably more important, the results mentioned above plus the rest of the results from the same survey, all followed nicely from simple economic arguments. For us, this was the most important result since it indicates that all categories of customers – large and small – behave in an economically rationalistic manner. Or in other words – once you understand your customer's economy, you can deduce his preferences.

The SISU project

The goal of the SISU project was to improve quality and productivity in software development, mainly for the telecom industry. It was co-financed by parts of the Norwegian telecom industry and the Norwegian research council. A problem that interested one of the participating companies was "What characterises a component that has many errors at integration and systems test?" In order to answer this question, we did as follows:

Table 4 Scores according to price category

Factor	Price category 1 (20)		Price category 2 (22)		Price category 3 (42)	
Product functionality	0.62 / 3	5	0.79 / 1	2	0.73 / 2	5.5
Product reliability	0.60 / 3	6	0.82 / 1	1	0.77 / 2	2
Product usability	0.70 / 1	2	0.60 / 3	7	0.64 / 2	8
Product efficiency	0.43 / 3	8	0.57 / 1	8	0.51 / 2	9
Product maintainability	0.33 / 2	9.5	0.38 / 1	9.5	0.07 / 3	10
Product portability	0.33 / 3	9.5	0.38 / 2	9.5	0.76 / 1	3.5
Service confidence	0.65 / 3	3	0.67 / 2	6	0.76 / 1	3.5
Service efficiency	0.59 / 3	7	0.72 / 2	4	0.73 / 1	5.5
Service capacity	0.75 / 3	1	0.77 / 2	3	0.81 / 1	1
Service responsiveness	0.64 / 3	4	0.71 / 1	5	0.69 / 2	7
Average line rank	2.7	–	1.6	–	1.7	–

1 We performed interviews with all developers in the division that was involved in the product under consideration.

2 Based on the interviews, we identified a set of possible influence parameters for the number of errors.

3 These possible influences were transformed into statistical hypotheses which could be tested in a standard way, based on the collected data.

The result was to be used to focus the test and inspection efforts. Thus, instead of distributing the verification and validation effort uniformly over all components, the company decided to use more effort on the components already flagged as potential troublemakers. We will in this paper only look at a few of these hypotheses. We have selected the following:

- Ha: SDL blocks which have been changed many times before will have more errors during integration and systems tests.
- Hb: SDL blocks which are given a high subjective complexity score will have more errors during integration and systems tests.
- Hc: The SDL blocks with many states in the SDL state machine will have more errors than those with few states will.

In order to test these hypotheses we used two statistical methods, namely ANOVA for Ha and Hb, and linear regression for Hc.

The ANOVA analysis is simply a question of the source of variance. We split the data set into two blocks according to some criterion and split the variance into two components: the within-block and between-block variances. If the between-block variance is significantly larger than the within-block variance, we will claim that there is an influence from the factor used as a basis for the data splitting. The statistical interference is based on the following relations between the sums of squares of, say X and Y and the number of elements in the sums. For X we have:

$$SS_X = \sum_{i=1}^{N} \left(x_i - \overline{X} \right)^2$$

For Y, we have correspondingly:

$$SS_Y = \sum_{j=1}^{M} \left(y_j - \overline{Y} \right)^2$$

In addition, we have the following statistical relationship:

$$\frac{SS_X / N}{SS_Y / M} \sim F_{N,M}$$

and we can use the F-statistics to check if there really is a difference between the two SS (Sum of Squares). In addition to the F-value, we get a number called the p-value, which tells us how likely it is that we could have got the observed F-value if the values were chosen at random. Thus, a small p-value is good – the question is "How small?" Statisticians have for convenience usually used p-values from 0.10 to 0.05.

For Ha – SDL blocks with many changes – we got $F_{N,M} = 10.25$ and $p = 0.000$ which is quite good. In addition, it is interesting to plot the intervals where the observations fall for the three categories – levels. Level 1 means low change traffic, level 2 means medium change traffic and level 3 means high change traffic. This is shown in the plot in Box 1.

The corresponding results for Hb – SDL blocks with high complexity – are as follows: $F_{M,N} = 1.20$ and $p = 0.315$. Again the plot is quite informative, see Box 2.

It is now straightforward to see that splitting the data set according to earlier change traffic will reduce the variance

Level	N	Mean	StDev	---------+---------+---------+------
1	21	4.190	4.412	(---*---)
2	6	12.667	11.587	(-------*------)
3	6	22.500	16.550	(-------*------)
				---------+---------+---------+------
Pooled StDev =	9.000			10 20 30

Box 1 Plot for Ha – SDL blocks

Level	N	Mean	StDev	---------+---------+---------+------
1	6	2.67	1.97	(----------*----------)
2	19	10.26	13.01	(-------*------)
3	8	11.00	10.16	(----------*----------)
				---------+---------+---------+------
Pooled StDev =	11.24			0.0 7.0 14.0

Box 2 Plot for Hb – SDL blocks

and that the probability that this change in variance due to random variations is less than 0.04 %. We will thus accept hypotheses Ha.

The probability of getting the observed variance reduction by splitting the data set according to the subjective complexity due to random variations is fairly large – more than 31 %. Thus, we will reject Hb.

Since we will use linear regression for Hc we first did a test for normal distribution of the number of code lines – LOC. We plotted the data in a normal-plot and used the Ryan-Joiner test for normality. As can be seen form the plot in Figure 1 we got an *R*-value of 0.97 and a *p*-value of 0.05, which is satisfactory for this use.

The linear regression analyses gave the following result:

sum-SDL
= - 3.37 + 0.00190 LOC – 0.0093 N-Op
+ 0.718 N-grs + 0.176 N-tilst

Again it is instructive to look at the *p*-values. A small *p*-value means that the sum-SDL value probably is influenced by this coefficient, while a large *p*-value means that any connection probably is a coincidence.

For the constant term, the *p*-value is 0.369, for the LOC term, the *p*-value is 0.092, for the N-Op term (number of operators) the *p*-value is 0.902, for the N-grs (number of interfaces) term, the *p*-value is 0.240 and for the N-tilst (number of states) term, the *p*-value is 0.050.

In order to see if the hypothesis holds, we will look at the *t*-statistics for the variables used in the regression analyses. Only LOC and number of states (N-tilst) have *p*-values that are below 10 %. The reduction in variance that is caused by the model is definitely significant – the *p*-value is again less than 0.0004. However, the adjusted R^2 is only 0.47. Thus, we have only explained 47 % of the data variation. The rest must stem from other factors than the ones included here.

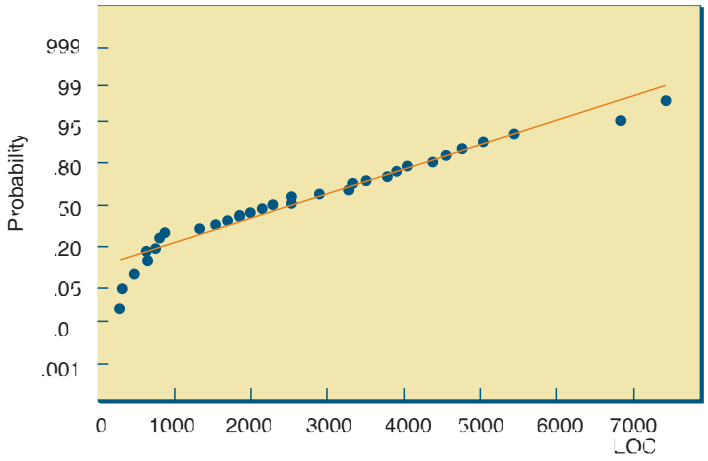


Figure 1 Normal Probability Plot

In order to check the model, we also computed the residual plots. A residual is the difference between the value observed and the value predicted by the estimated or proposed model. A residual plot will show the structure – or lack thereof – for the differences between the model and the observations.

From the plots in Figure 2 we see that the model is reasonable due to the following observations:

1 The residual chart indicates that there is no systematic difference in the residuals according to the observation number.

2 The residual histogram indicates that the residuals are spread around zero in a symmetric way.

However, there are also two worries:

1 The rather bad normality, as indicated by the far from linear normal plot – top left diagram;

2 The residual plot seems to indicate that the residuals increase with the number of SDL errors. Especially the rightmost four data points give some cause for worry – bottom right diagram.

Everything considered we should keep the model but look more into the reasons for the two above-mentioned points of concern. This implies that during inspections and testing we should be especially alerted when we consider modules with many states and/or many code lines. In addition, further experiments are needed in order to find the sources of the rest of data variation – 53 %. We have already ruled out some factors, like the number of operators and the number of interfaces in a module. Most likely, the source of the missing variation is variation in the development process.

The QARI project

The QARI project is an ESSI process improvement experiment (PIE) project. The company involved is planning to start using object-oriented development and had a list of benefits that they hoped to reap from this. One of these benefits is a reduction in the portion of serious errors inserted into the system. The strategy from a statistician's point of view is clear:

1 Identify a project that is developed by the use of traditional methods, but otherwise similar to the new project,

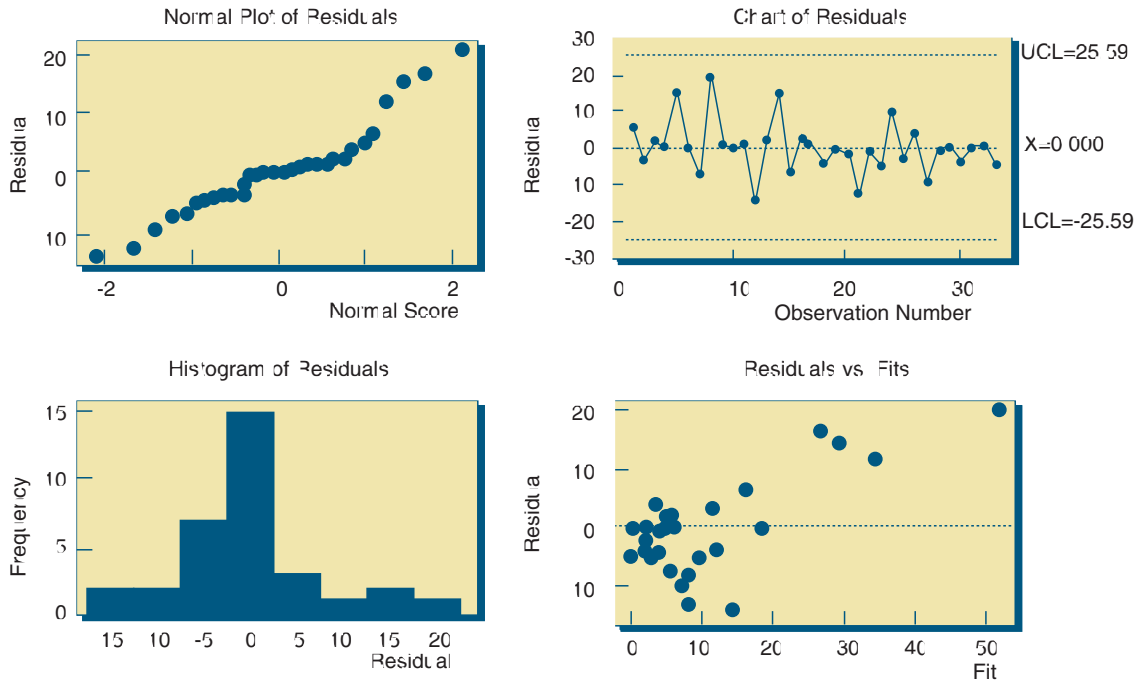


Figure 2 Residual Model Diagnostics

Table 5 Number of failures per category for O-O and traditional development

Failure category 1 is most serious	Traditional development	Object-Oriented development	Sum	Fractions (p_i)
1	11	1	12	0.08
2	3	36	39	0.25
3	30	31	61	0.40
4	22	19	41	0.27
Sum errors (N)	66	87	153	1.00

Table 6 Number of failures per category fo O-O and traditional development – categories 1 and 2 are combined

Failure category 1 is most serious	Traditional development	Object-Oriented development	Sum	Fractions (p_i)
1,2	14	37	51	0.33
3	30	31	61	0.40
4	22	19	41	0.27
Sum errors (N)	66	87	153	1.00

developed by using object-oriented development;

2 Take all reported errors in both projects – in this case products – and assign a seriousness score to each error;

3 Perform a Chi-square test on the data to see if there is any difference in the data.

Table 5 shows some of the data collected. The data shows how failures are distributed according to seriousness in two projects, where we have used traditional and object-oriented development methods respectively.

There is one statistical hitch with these data – the Chi-square test uses an approximation that does not hold if we have too few observations – less than five – in one or more of the table cells. Thus, we need to combine the data for the two most serious categories. This data combination gives us Table 6.

The Chi-square test computes the difference between what we should have seen if the distribution of errors over serious-

Table 7 Expected versus observed number of failures per failure category

Category	Traditional development	Object-oriented development	Sum
	14	37	
1,2	22.00	29.00	51
	30	31	
3	26.31	34.69	61
	22	19	
4	17.69	23.31	41
Total	66	87	153

ness scores was the same in both cases and what we really observe. The test value is computed as follows:

$$Q = \sum_i \frac{(N_i - p_i N)^2}{p_i N}$$

The terms N_i are found directly in the table. The Np_i terms are computed by multiplying the total number of errors for each project by the percentage of errors for each category. Thus, for category 1,2 we should expect 66 * 0.33 for traditional development and 87 * 0.33 for object-oriented development. The result of the Chi-square test as performed by Minitab is as shown in Table 7 – expected counts are printed below observed counts.

From the definition, we find that ChiSq = 2.909 + 2.207 + 0.516 + 0.392 + 1.052 + 0.798 = 7.874 and the p-value is 0.020. There is thus a 2 % probability that the observed difference is due to random variations in the observed data.

What will we make of this result? The statistics cannot help us here. The object-oriented development has more class 1 and 2 errors than expected while the inverse is the case for traditional methods. On the other hand, we had to combine the data from the most serious and the second most serious categories in order to perform the test and there is really much fewer serious errors reported in the project with object-oriented development. There are two lessons to be learnt from this:

1 Although the statistics can provide us with significance levels, we need to use common sense when we interpret the results.

2 Many statistical tests build on one or more assumptions. These assumptions must be checked and adhered to and this can lead to data sets that no longer can be used to test what we are really looking for.

Where do we go from here

First and foremost, we need to focus each experiment on a limited part of the development process. In order to get sound models, we can use two approaches:

- We can build on statistical analyses alone, which requires a large amount of experimental data. In order to obtain a large amount of data, we need to focus on a process that is performed several times and with a large variation in types of participants.
- We can combine experimental data and expert knowledge. There are several ways to do this – parametric Bayes and non-parametric Bayes are probably the two best known.

We can also use expert knowledge to get more information out of each experiment, for instance by using experience on which parameters influence which results. Such knowledge will for instance help us make more focused experiments.

Inspection and testing are activities that could provide interesting starting points. If we choose inspection, important questions could concern which parameters influence the effectiveness of inspections, for instance measured as the percentage of errors found during inspection. Among the parameters to be considered are:

- Preparation time;
- Use or no use of checklists;
- Amount of application knowledge among the participants;
- Time spent in meetings;
- Size and complexity of the document.

Similar sets of interesting parameters can be defined for other subprocesses. Based on sound scientific methods and engineering knowledge, we could then build a library of subprocess models that again could be combined into project specific development processes through the use of QFD or any similar method.

In addition, we must remember always to include the project's environment in our measurements. As a result of this, we need a common, documented way of describing a project environment. In this way, we can identify what is similar and what is different when we want to compare projects, both inside a company and between companies. ITUF's working group on project management has done an important first effort, but much is left to be done before we have a common starting point.

On the top level, we need to understand important relations like which parameters influence which project and product characteristics. This is for instance important for project planning and trade-offs as well as for developing a product that is delivered on time and satisfies the customer's needs and expectations.

Tor Stålhane (55) finalized his studies in electrical engineering at the Norwegian University of Science and Technology in 1969 and worked with compiler development and maintenance until 1985. After completing his Ph.D. in statistics in 1988 he returned to SINTEF, where he has been working on software reliability and safety as well as software process improvement. In addition to his job at SINTEF he is a professor in computer science at the Stavanger Polytechnic.

e-mail: tor.stalhane@informatics.sintef.no

Assessment-based software process improvement

TORE DYBÅ

The software engineering community is increasingly more focused on the potential benefits of software process assessment and software process improvement. In this paper, we present an overview of the general principles behind the use of process assessments and how to use such assessments within the context of software process improvement.

Introduction

One popular way of starting a software process improvement program is to do an assessment. In this paper we focus on the use of such assessments within the context of process improvement, addressing the following issues:

- The basic principles of software process assessment, reliability and validity issues, and key factors for success;
- General principles for assessment-based software process improvement;
- Presentation of the IDEAL model and ISO/IEC 15504 methodology for process improvement, including a brief comparison of these two models;
- Key factors for success in software process improvement, specifically as seen from the SPICE-project;
- Strengths and limitations of assessment-driven software process improvement.

What is process assessment?

The history of assessment-driven software process improvement dates back to the 1980s when two key initiatives were undertaken by the military in the US and in the UK. Both of these initiatives had as their objective to improve the selection criteria for potential software contractors, in order to reduce the risks associated with software projects and improve the quality of the delivered software. The US initiative resulted in the Capability Maturity Model (CMM) for software [1, 2, 3], and the UK initiative resulted in the emerging ISO/IEC 15504 standard for software process assessment [4]. ISO/IEC 15504 is also known as SPICE – the name of the project developing the standard.

Over the last years, several assessment models have emerged in the software industry, and there is a range of possible assessment models that one can choose from. In addition to the CMM for software and ISO/IEC 15504, further examples of such models are ISO 9001, 9000-3, TickIT, European Quality Award, Bootstrap, Trillium, and ISO/IEC 12207. It is outside the scope of this paper, however, to discuss all of these models in detail; hence, we limit ourselves to the issues underlying assessment-driven software process improvement in general. These general principles will be exemplified by the IDEAL model [5] and the informative framework for software process improvement in the emerging standard for software process assessment [6].

According to ISO, process assessment is "a disciplined evaluation of an organization's software processes against a model compatible with the reference model" [7]. In this definition, process is defined as "a set of interrelated activities, which transform inputs into outputs", where the term "activities" also covers the use of resources [7, 8].

Process assessment can be used in two principal contexts [9], as shown in Figure 1, and described below.

1 *Process improvement* – with the objective of understanding the state of the organization's own processes for process improvement;

2 *Process capability determination* – with the objective of determining the suitability of another organization's processes for a particular contract or class of contracts.

In this paper, we concentrate on the first of these two contexts – software process improvement. The role of assessment in this context is to provide a means of characterizing the current practices within the organization, and analyze the resulting strengths and weaknesses in terms of the organization's business needs. These analyses, then, should lead to clear priorities for improvement actions to enhance the capabilities of the software process.

Usually, an assessment method includes:

a) A *reference model* for processes, practices and capability levels;

b) A *measurement framework* for measuring an objective attribute or characteristic of a practice or work product that supports the judgement of the performance of, or capability of, an implemented process;

c) An *assessment instrument* (e.g. questionnaire) to assist the assessor in evaluating the performance or capability of processes and in handling

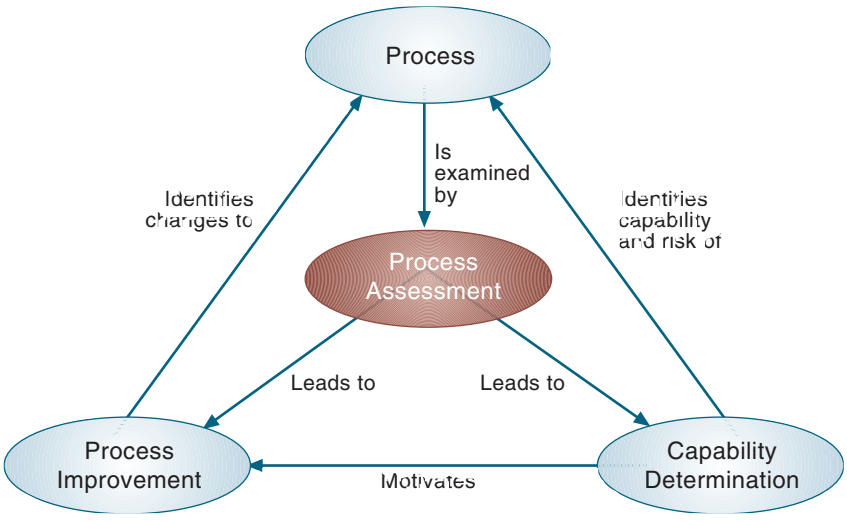


Figure 1 The use of software process assessment [7]

assessment data and recording the assessment results;

d) A clear mechanism to present the results.

All assessment models give a reference point of current strengths and opportunities. However, the reference point is someone else's ideal model – of software development and improvement. These models do not necessarily fit into *your* organization and how *you* think about software development and software process improvement. The choice of an assessment model, therefore, is most likely to influence your improvement philosophy. Consequently, you should consider if the underlying reference model matches your organization's idea of an ideal process for developing products for your customers.

Assessment principles

General principles

There are several references in the literature that suggest general principles for assessments, e.g. [1, 10, 11, 12, 13, 14, 15, 16].

Generally, an assessment model is based on the principle that the capability of a process can be assessed by demonstrating the achievement of certain process attributes. These attributes are measured against a defined set of criteria for "good software engineering practice", which is defined in the respective reference models.

The key objective for all types of assessments, however, is to uncover the problems of an organization, prioritizing their relative importance, and to agree on a way to fix them.

Overview of the assessment process

General guidelines for performing an assessment are given in e.g. [1, 11, 12, 16]. Specific principles and guidelines for performing CMM-based assessments are given in [17] and for ISO/IEC 15504 conformant assessments in [15].

Typically, however, an assessment process consists of the following four broad phases [18]:

1 *Preparing the assessment.* The key task in this phase is to prepare the input to the assessment. This includes defining the scope and goal of the assessment, the organizational unit and the set of processes to be assessed, and the process instances to be assessed.

2 *Gathering the data.* In this phase, the chosen process instances are investigated against the assessment model. Data gathering will often be performed through interviews and/or discussions with the people concerned with the process, and through the examination of relevant documents. Other forms of data gathering, however, may include the use of automated tools, or collecting data in a (semi-) continuous way.

The data gathering activities probe on both *what* is done in terms of activities and work products, and on *how well* it is done in terms of process effectiveness or capability.

3 *Analyzing the data*. In this phase, ratings are assigned to the process instances and the assessment output is prepared. The assessment model's rating scale defines what to rate and the scale of values. The collected data is analyzed against the definitions of the attributes in the assessment model, and the evidence and justifications for the ratings are recorded.

Reliability and validity issues in the rating process are of prime importance in major decisions regarding improvement actions based on assessment. This is dealt with in later sections of this paper.

The formal *output* of an assessment varies according to the chosen assessment model. CMM, for example, measures organizational maturity; consequently, the output of a CMM assessment is a maturity level. ISO/IEC 15504, on the other hand, measures process capability. Therefore, the output of an ISO/IEC 15504 assessment is a set of process profiles that can be presented in a number of ways.

For the purpose of process improvement, the assessment output is used to identify the current situation, highlighting its strengths, weaknesses, risks, and opportunities for improvement. Furthermore, the output feeds into the improvement cycle of planning and prioritizing improvement actions, implementing the improvement plans, monitoring the results, and taking further improvement actions.

4 *Feeding back the results.* Feedback is important to gain both management buy-in and commitment from software developers. The feedback from an assessment may vary, however, depending on the nature and purpose of the assessment and on any agreements reached about the dissemination and use of the results. Feedback and reporting may take the form of formal written reports, presentations to one or more groups, or simply informal, verbal communication.

Reliability and validity issues in performing assessments

Reliability

Both CMM and SPICE conformant assessments are measurement procedures, for which reliability is of prime concern. In this context, reliability refers to the consistency and stability of a score from a measurement scale.

Numerous methods are available for assessing the reliability of a measurement scale. However, the types of reliability

Table 1 Classification of reliability estimation methods

Assessment Sessions Required	Assessment Models Required	
	One	Two
One	Split-Half Internal Consistency	Alternate-Form (immediate)
Two	Test-Retest	Alternate-Form (delayed)

computed in actual practice are relatively few, and they are summarized in Table 1 in terms of the different techniques for measuring reliability in relation to the number of assessment models or procedures required and the number of assessment sessions required. A brief description of these methods are made below (for a more detailed treatment see [19]).

Since all types of reliability are concerned with the degree of consistency or agreement between two independently derived sets of scores, they can all be expressed in terms of a *correlation coefficient*, which expresses the degree of correspondence, or *relationship,* between two sets of scores. The absolute value of the correlation coefficient can range from 0 to 1.0, with 1.0 perfectly reliable and 0 perfectly unreliable. A perfectly reliable measure is seldom attainable, but one should strive for the best measures possible. There are no rules available for what constitutes a reliable measure. However, Nunnally [20] has suggested the following minimum standards:

- 0.7 is used for exploratory research;
- 0.8 is used for basic research;
- 0.9 or better is used in applied settings where important decisions will be made with respect to specific test scores.

According to these recommendations, an assessment instrument that is being used for decisions with respect to process improvement priorities should constitute coefficients of 0.9 or higher.

Test-retest reliability

In order to estimate reliability with this method, we must assess an organization's software processes at two different times using the exact same assessment procedure. The reliability coefficient for the test-retest method is simply the correlation between the scores obtained on the two assessments. Retest reliability shows the extent to which scores on an assessment can be generalized over different occasions, that is, the higher the reliability, the less susceptible the scores are to random daily changes in the condition of the organization or of the assessors. The error variance in this case corresponds to the random fluctuations of performance from one assessment to the other.

Alternate-form reliability

The important difference between the test-retest and the alternate-form methods is in the assessment procedure itself. In the former, the same instrument and procedure is used; in the latter, a different but equivalent procedure is used. This can be achieved, e.g. by using two different assessment instruments or by using two assessment teams, with or without a time interval. Most common is the use of a time interval of about two weeks, much like the test-retest method.

The correlation between the scores obtained on the two forms represents the reliability coefficient of the test. It will be noted that such a reliability coefficient is a measure of both temporal stability and consistency of response to different assessment procedures. This coefficient thus combines two types of reliability. If the two forms are administered in immediate succession, the resulting correlation shows reliability across forms only, not across occasions. The error variance in this case represents fluctuations in performance from one assessment to another, but not fluctuations over time.

Split-half reliability

Split-half reliability is accomplished by splitting a multi-item assessment instrument in half and then correlating the results of the scores in the first half with those in the second half. Split-half reliability thus provides a measure of consistency with regard to content sampling. Temporal stability of the assessment does not enter into such reliability, because only one assessment session is involved. It should be noted, however, that this correlation actually gives the reliability of only a half-assessment. In both test-retest and alternate-form reliability, on the other hand, each score is based on the full number of items in the assessment instrument.

The effect that lengthening or shortening an assessment instrument will have on its coefficient can be estimated by means of the Spearman-Brown formula [20]. Other things being equal, this formula shows that the longer the instrument, the more reliable it will be. Lengthening an instrument, however, will increase only its consistency in terms of content sampling, not its stability over time.

Internal consistency

In essence, the internal consistency technique computes the mean reliability coefficient estimates for all possible ways of splitting a set of items in half. This presumably results in a better estimate of reliability. This *interitem consistency* is influenced by two sources of error variance: 1) content sampling (as in alternate-form and split-half reliability); and 2) heterogeneity of the behavior domain sampled. The more homogeneous the domain, the higher the interitem consistency. By far the most commonly used internal consistency estimate is Cronbach's coefficient alpha [21].

Although there has been a general concern with the reliability of assessments, there is a very limited number of reports in the literature on the results of evaluating the reliability of software process assessments using the internal consistency method [22]. The most extensive program of research in this area has been conducted in the context of the SPICE trials.

Bollinger and McGowan [23] questioned the "statistical reliability" of the algorithm used to calculate the maturity level of an organization. Humphrey and Curtis [24] replied to this critique by quoting a Cronbach alpha figure of 0.9 for the level 2 and level 3 questions of the 1987 SEI Maturity Questionnaire [25]. However, they omit the details of the study.

El Emam and Madhavji [26] developed a maturity assessment instrument for the following dimensions of maturity: a) process and product standardization, b) project management, c) tool usage, and d) organization. For each of the four dimensions, the Cronbach alpha was computed in the range of 0.8–0.9, which is consistent with the reliability coefficient reported [24].

Fusaro, El Emam and Smith [27] present results of an empirical evaluation of the reliability of the 1987 SEI maturity questionnaire and the SPICE version 1 capability dimension. The results of the study show that both full-length instruments have reliability coefficients above 0.9. Furthermore, a recent study by [28] evaluates the internal consistency of ISO/IEC PDTR 15504. This study confirms the results of Fusaro, El Emam and Smith [27], indicating that the internal consistency of SPICE version 2 did not suffer any deterioration.

Table 2 Three basic types of validity in process assessment

Types of Validity	Definitions
Content validity	The degree to which the items in the assessment instrument represent the domain or universe of the processes under study.
Criterion validity	The degree to which the assessment instrument is able to predict a variable that is designated a criterion.
Construct validity	The degree to which the assessment instrument represents and acts like the processes being measured.

These reliability studies indicate that both the 1987 SEI maturity questionnaire and the ISO/IEC PDTR 15504 both have sufficiently high internal consistency to be usable in practice.

Validity

For a scale to be valid, it must also be reliable. *Validity* is differentiated from reliability in that the former relates to accuracy while the latter relates to consistency. An assessment instrument is valid if it does what it is supposed to do and measures what it is supposed to measure. If such an instrument is not valid, it is of little use because it is not measuring or doing what it is supposed to be doing. In other word, process assessment validity could be defined as "*what* the assessment measures and *how well* it does so".

In the context of assessment-driven software process improvement, the validity concern could be expressed by the question: "Are [model X] conformant assessments actually measuring the effectiveness of [organization Y]'s software development processes".

Fundamentally, all procedures for determining the validity of an assessment are concerned with the relationships between the assessment results and other independently observable facts about the behavior characteristics under consideration. For process assessment, there are three basic types of validity of measurement that we must be concerned with. These validity concerns are outlined in Table 2 and are discussed next (for a more detailed treatment see [19]).

Content validity

Content validity has to do with the degree to which the scale items represent the domain of the concept under study. Essentially, procedures for content validation involve the systematic examination of the measurement instrument to determine whether it covers a representative sample of the behavior domain to be measured.

Software process assessment is based on the idea of "best practice". Therefore, expert judgement is important to ensure that the assessment procedures are at least perceived to measure software best practice [29]. Furthermore, it is necessary that the assessment instruments include items that adequately sample from the content domain.

Criterion validity

Criterion-related validity has to do with the degree to which the scale under study is able to predict a variable that is designated a criterion, i.e. the effectiveness of an assessment in predicting the organization's software process performance. The criterion measure against which the assessment is validated may be obtained at approximately the same time as the assessment or after a stated interval.

There are two general subtypes of criterion validity, which are differentiated on the basis of the time relations between assessment and criterion: predictive and concurrent validation. *Predictive validation* is the extent to which a future level of some criterion variable can be predicted by a current assessment. Here, the emphasis is primarily on the criterion (predicted) variable rather than the measured variable. Efforts at empirically investigating the predictive validity of software process assessment are reported in e.g. [30, 31], and the SPICE trial reports [32, 33].

Similar to predictive validity, *concurrent validity* is largely criterion-oriented, with the only major difference being the time dimension. With concurrent validity, the measure of the predictor and criterion variables is made at about the same point in time; i.e. concurrent validation is relevant to assessments employed for *diagnosis* of existing status, rather than prediction of future outcomes.

Construct validity

Construct validity is the degree to which the measurement scale represents and acts like the concept being measured, in other words, the extent to which the assessment may be said to measure a theoretical construct or trait. Construct validation has focused attention on the role of theory in test construction and on the need to formulate hypotheses that can be proved or disproved in the validation process.

Assessment success factors

In order for process assessment to be successful, reliability and validity are necessary but by no means sufficient conditions. Humphrey [11] identified a competent team, sound leadership, and a cooperative organization as the basic requirements for successful process assessments.

Furthermore, the SPICE project identified the following assessment success factors [15, 34]:

- *Commitment.* The commitment of the assessment team to the objectives they have established for the assessment is fundamentally important to ensure that these objectives are met. This commitment requires that the necessary resources, time, and personnel are made available to undertake the assessment.
- *Motivation.* Management attitudes and the data collection methods have a significant influence on the outcome of an assessment. Therefore, the assessment should be focused on the software process, rather than on the people implementing the process. Providing feed-

back and maintaining an atmosphere that encourages open discussion, without blaming individuals, about preliminary findings is essential to get meaningful assessment outputs.

- *Confidentiality.* Respect for the confidentiality of sources of information and documentation gathered during the assessment is essential to secure that information. Therefore, it is important that adequate controls are in place to handle, and protect, such information.
- *Relevance.* The members of the organizational unit being assessed should believe that the assessment will result in benefits that are relevant to their needs. It is the software developers that are the principal source of knowledge and experience about the process; thus, they are in the best position to identify potential weaknesses and relevance for performing the work.
- *Credibility.* The organizational members must believe that the assessment will deliver a result that is objective and representative of the assessment scope. Therefore, it is important that all parties are confident that the assessment team has adequate experience in performing assessments, an adequate understanding of the organizational unit and its business, and sufficient impartiality.

However, assessment alone does not create improvement. It just makes it possible and supports it. Therefore, to have effect beyond mere exploration, the assessment must be directed toward *action.* In this context, assessments can only be considered successful if they contribute to successful process improvements.

Using assessments to guide software process improvement

General principles

Process improvement is a complex undertaking that involves management issues, people issues, culture values, quality improvement techniques, and change management techniques. By choosing a ready-made assessment model as the instrument for prioritizing improvement actions, one is also choosing an underlying philosophy on these issues. As an example, choosing between CMM and ISO/IEC 15504 assessments means choosing between a focus on organizational maturity versus a focus on process capability, respectively.

The SPICE project built on the following general principles of improvement when they developed the guide for software process improvement [6]:

- Software process improvement is based on process assessment results and process effectiveness measures.
- Software process assessment produces a current process capability profile, which may be compared with a target profile based on the organization's needs and business goals.
- Process effectiveness measures help identify and prioritize improvement actions that support organizations in meeting their needs and business goals, and in achieving software process goals.
- Software process improvement is a continuous process. Improvement goals identified and agreed within the organization are realized through multiple cycles of planning, implementing and monitoring activities.
- Improvement actions identified within a process improvement program are implemented as process improvement projects.
- Metrics are used for monitoring the improvement process in order to indicate progress and to make necessary adjustments.
- Software process assessment may be repeated in order to confirm that the improvements have been achieved.
- Mitigation of risk is a component of process improvement and should be addressed from two viewpoints: 1) the risk inherent in the current situation, and 2) the risk of failure in the improvement initiative.

Hence, the aim of assessment-based software process improvement is to use both the organization's needs and business goals, and industry norms and benchmarks as the main stimuli for process improvement. For the majority of software companies, however, this is often easier said than done.

CMM-Based Appraisal for Internal Process Improvement (CBA IPI)

The CBA IPI method is a diagnostic tool, aimed at identifying the strengths and weaknesses of an organization's current software processes related to the CMM. Furthermore, to prioritize software improvement plans, focusing on the most beneficial software improvements, given the current level of maturity and the business goals of the organization [17, 35].

The method is an assessment of an organization's software process capability by a trained group of professionals who work as a team to generate findings and ratings relative to the CMM key process areas within the assessment scope. The findings are generated from data collected from questionnaires, document reviews, presentations, and in-depth interviews with middle managers, project leaders, and software practitioners.

The CBA IPI method supports the diagnostic phase in a cycle of ongoing improvement, and it has two primary goals:

- To support, enable, and encourage an organization's commitment to software process improvement;
- To provide an accurate picture of the strengths and weaknesses of the organization's current software process, using the CMM as a reference model, and to identify key process areas for improvement.

In the next section we shall take a closer look at this cycle, which the SEI has named IDEAL.

The IDEAL model

The SEI's recommended framework for software process improvement is the IDEAL model [5] shown in Figure 2. The IDEAL model was developed in order to present a consistent view of the activities of an improvement program based on transitioning the CMM into an organization's practice [36]. The IDEAL approach consists of the following five phases [37]:

I *Initiating*	Laying the groundwork for a successful improvement effort.
D *Diagnosing*	Determining where you are relative to where you want to be.

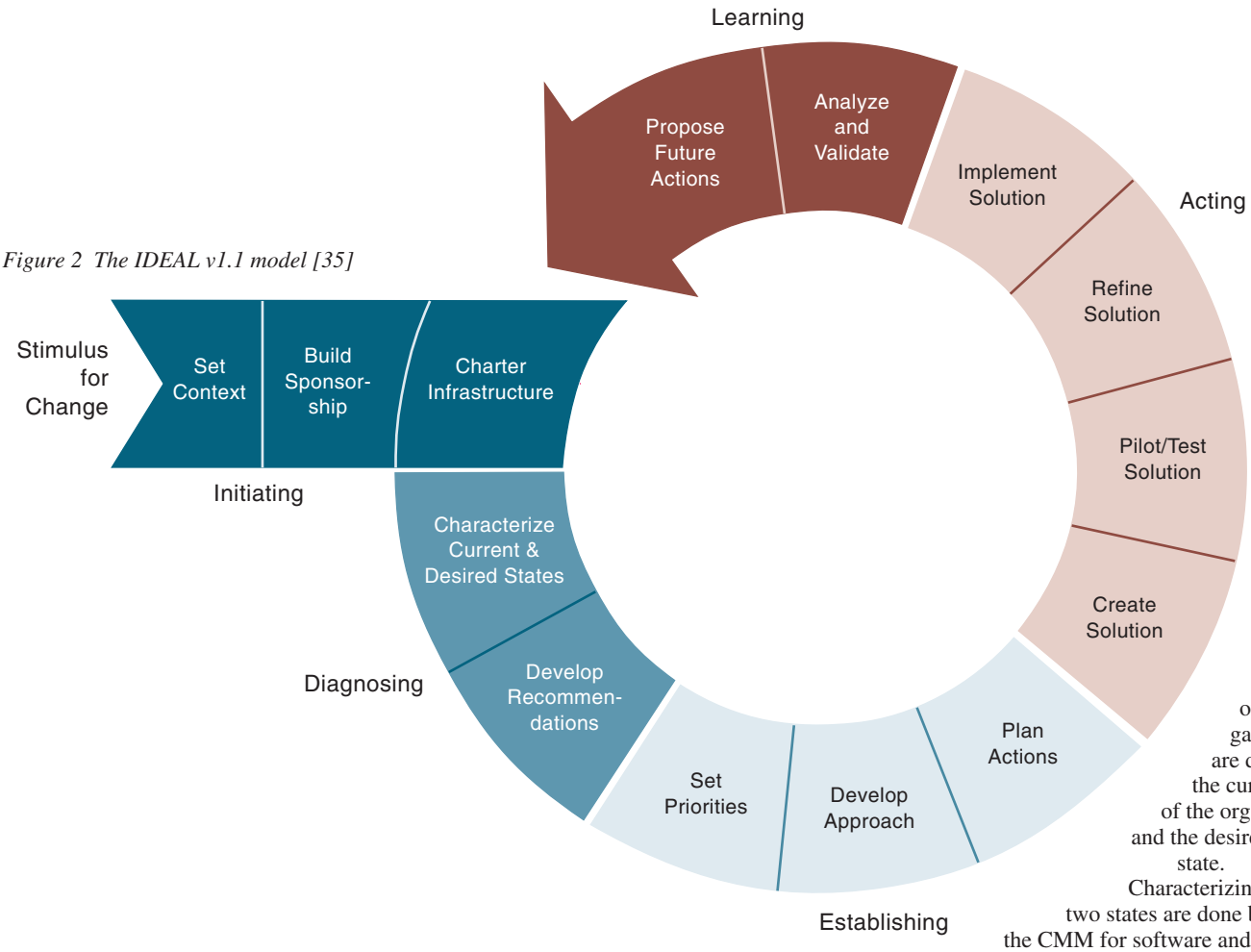


Figure 2 The IDEAL v1.1 model [35]

E *Establishing* Planning the specifics of how you will reach your destination.

A *Acting* Doing the work according to the plan.

L *Learning* Learning from the experience and improving your ability to adopt new technologies in the future.

A brief description of the five phases is made below.

The initiating phase

Critical groundwork is completed during the initiating phase. The business reasons for undertaking the effort are clearly articulated. The effort's contributions to business goals and objectives are identified, as are its relationships with the organization's other work. The support of critical managers is secured, and resources are allocated on an order-of-magnitude basis. Finally, an infrastructure for managing implementation details is put in place.

The activities of the initiating phase are critical. If they are done completely and well, subsequent activities can proceed with minimal disruption. If, however, they are done poorly, incompletely, or haphazardly, then time, effort, and resources will be wasted in subsequent phases.

The diagnosing phase

The diagnosing phase builds upon the initiating phase to develop a more complete understanding of the improvement work. During the diagnosing phase two characterizations of the organization are developed: the current state of the organization and the desired future state. Characterizing these two states are done by using the CMM for software and the CBA IPI. These organizational states are then used to develop an approach for improving business practices.

The establishing phase

The purpose of the establishing phase is to develop a detailed work plan, with priorities set that reflect the recommendations made during the diagnosing phase. An approach is then developed that honors and factors in the priorities. Finally, specific actions, milestones, deliverables, and responsibilities are incorporated into an action plan.

The acting phase

The activities of the acting phase help an organization implement the work that has been conceptualized and planned in the previous three phases. These activities will typically consume more calendar time and more resources than all of the other phases combined.

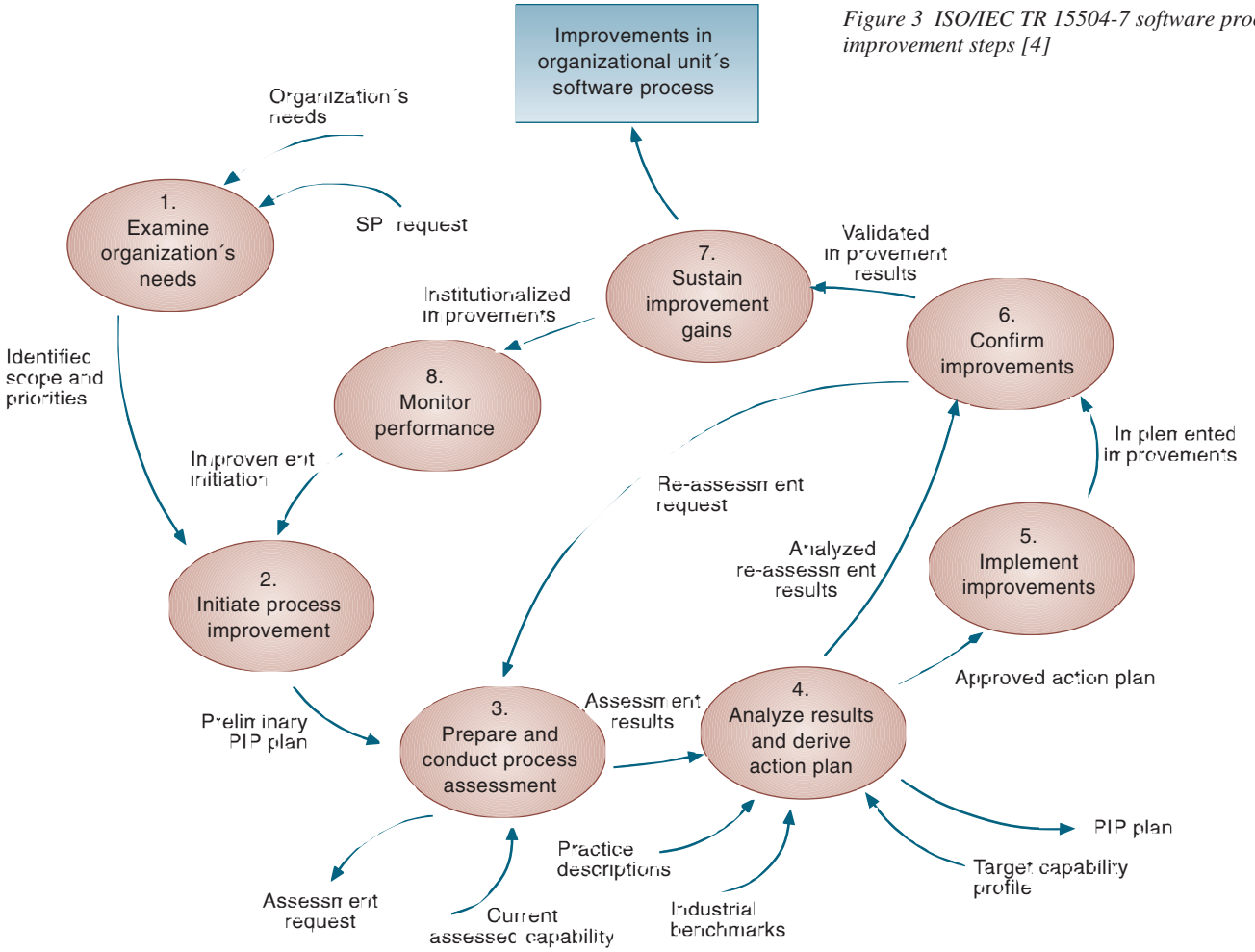


Figure 3 ISO/IEC TR 15504-7 software process improvement steps [4]

The learning phase

The learning phase completes the improvement cycle. One of the goals of the IDEAL model is to continuously improve the ability to implement change. In the learning phase, the entire IDEAL experience is reviewed to determine what was accomplished, whether the effort accomplished the intended goals, and how the organization can implement change more effectively and/or efficiently in the future. This requires that records must be kept throughout the IDEAL cycle with this phase in mind.

The ISO/IEC 15504 framework for software process improvement

Part 7 of the ISO/IEC 15504 document set provides guidance on how to prepare for and use the results of an assessment for the purpose of process improvement.

Figure 3 illustrates the cycle for continuous software process improvement using [6]. A description of the steps in this cycle is made below [6, 38].

Examine the organization's needs and business goals

The first step of the improvement cycle starts with a recognition of the organization's needs and business goals. The objectives of process improvement is then identified and described in terms of e.g. quality, time to market, cost and customer satisfaction. Finally, these objectives direct the choice and priorities of the processes to be assessed, the definition of improvement targets and ultimately the identification of the most effective improvement actions.

Initiate process improvement

This step emphasizes the need to consider a process improvement program (PIP) as a project in its own right that is planned and managed accordingly. Furthermore, a *PIP plan* should be produced, including the improvement goals, the improvement scope, an outline of all improvement steps, the identification of key roles, the allocation of adequate resources, and the establishment of appropriate milestones and review points.

Prepare for and conduct a process assessment

This step provides guidance on how to define the assessment inputs needed to carry out an ISO/IEC 15504-conformant assessment. In particular, the choice of a qualified assessor, the definition of a detailed purpose statement, and the identification of an appropriate assessment scope are all important issues to be considered. The assessment is initiated using the prepared assessment input, and it delivers the results as an assessment output consisting of the current process profile, the assessment record, and additional information for process improvement.

Analyze assessment output and derive action plan

In this step, information collected during the assessment is analyzed in the light of the organization's needs. Improvement areas are identified and prioritized, based on risks related either to not improving an area or to incurring in a failure of specific improvement actions. Targets for improvement should be quantified for each improvement area, including either target values for process effectiveness, target capability profiles, or a combination of the two. Specific actions should be developed to meet the quantified targets. The set of agreed actions should be documented as the process improvement action plan, which is a tactical plan that should be integrated with the *process improvement program plan* originally developed at a strategic level.

Implement improvements

In this step, specific process improvement projects should be initiated, each concerned with implementing one or more process improvement actions. A detailed *implementation plan* should be developed for each project. Finally, the process improvement projects should be monitored against the process improvement project plan.

Confirm improvements

When the process improvement project has been completed, management should be involved both to approve the results and to evaluate whether or not the organization's needs have been met. If, after improvement actions have been taken, measurements show that process goals and improvement targets have not been achieved, it may be desirable to redefine the improvement project by returning to an earlier step in the improvement cycle. Risks should be reevaluated to confirm that they remain acceptable.

Sustain improvement gains

After process improvements have been confirmed, all those for whom it is applicable should use the improved software process. Management is required to monitor the institutionalization of the improved process and to give encouragement when necessary. Responsibilities for monitoring should be defined, as well as how the monitoring will be done, e.g. by using appropriate effectiveness measurements.

Monitor performance

In the final step of the model, the performance of the organization's software process should be continuously monitored, and new process improvement projects should be selected and implemented as part of a continuing process improvement program, since additional improvements are always possible.

Contrasting the IDEAL and ISO/IEC 15504 improvement models

Clearly, there is a strong correlation between the IDEAL model and the ISO/IEC 15504 model for software process improvement, although some issues in IDEAL are not covered in ISO/IEC 15504 and vice versa. The levels of detail differ significantly: Chapter 5 in part 7 of ISO/IEC 15504 is about 12 pages long, and the IDEAL is a handbook over 200 pages long.

Recognizing that there is a limit to the amount of detail the emerging standard *can* cover, it was important to the ISO working group not to presume specific organizational structures and management philosophies [38].

Apart from the level of detail, the biggest difference between the IDEAL and ISO/IEC 15504 improvement models is in the architecture of the assessment model. One of the objectives of the ISO/IEC effort was to create a way of measuring process capability, while avoiding a specific approach to improvement such as the CMM's maturity levels. ISO/IEC selected an approach to measure the implementation and institutionalization of specific processes, that is, a pro-

Table 3 Ingredients for successful software process improvement (Zahran, 1998)

1.	Alignment with the business strategy and goals.
2.	Consensus and buy-in from all stakeholders.
3.	Senior and middle management support
4.	Dedicated resource to manage the implementation and coordinate the process improvement activities.
5.	Sensitivity to the organizational context.
6.	Management of change.
7.	Prioritization of actions.
8.	Creation of the support infrastructure.
9.	Monitoring the results of software process improvement.
10.	Learning from the feedback results.

cess measurement rather than an organization measurement. Using this approach, maturity levels can be viewed as sets of process profiles. Thus, whereas the CMM addresses organizations, ISO/IEC 15504 addresses processes.

The biggest similarity is that both models have an emphasis on continuous process improvement, thus representing two varieties of the Plan-Do-Check-Act cycle, developed by Dr. Walter Shewhart in the 1920s [39].

Key factors for success in software process improvement

At present, there are few empirical studies investigating the key factors for success in software process improvement. Despite this fact, most of the literature is full of advice regarding factors that should be taken into consideration to ensure the success of software process improvement efforts. Empirical studies related to the CMM and ISO/IEC 15504 can be found in e.g. [4, 40, 33, 41].

ISO/IEC 15504 highlight cultural and management issues as fundamental to succeed with software process improvement and organizational change. The following cultural issues were identified [6]:

- Management commitment, responsibility and leadership;
- Shared values, attitudes and behavior;
- Process improvement goals and motivation;
- Communication and teamwork;
- Recognition and award system;
- Education and training.

Furthermore, the following management issues were identified [6]:

- Organizational principles for process improvement;
- Planning for process improvement;
- Measurement of process improvement;
- Reviewing of process improvement activities.

It is, however, outside the scope of this article to elaborate on these issues; nevertheless, in Table 3 we provide a list of some critical factors which, according to Zahran [16], represent essential ingredients for successful implementation of software process improvement.

Strengths and limitations of assessment-driven improvement

As already described, an assessment provides ratings based on conformance with a reference model. However, software process improvements can hardly be claimed unless process effectiveness is closely related to the specific circumstances, needs, and business goals of the organization. Since organizations are unique and have unique circumstances, the choice of process effectiveness measures will differ between different organizations. This, then, raises the question of the applicability of models such as CMM and ISO/IEC 15504 that are based on the positivistic presumption of a universal 'best practice' and the 'one best way'. This can be contrasted with the use of models such as QIP [42], with the support of GQM [49] and EF [44], which are more concerned with the contingent characteristics of the individual organization [45].

Furthermore, both the CMM and ISO/IEC 15504 have their roots in total quality concepts developed by 'gurus' like Shewhart (1931), Deming (1982), Juran (1988) and Crosby (1979), and their principle of statistical control. Humphrey (1989) put the case for the need for statistical control of the software development process, arguing that a process under statistical control can achieve consistently better results only by improving the process. Moreover, he argued that "If the process is not under statistical control, sustained progress is not possible until it is". The concept of statistical control, however, is not of much help for most small and medium sized companies, developing only one project at a time, and consequently having too few data sets to base decisions on.

As for CMM, its authors make it clear that it was developed for use with large, government funded types of projects [1]. It can be tailored and the documentation to do so is available from the SEI [46]. However, the process of tailoring is itself so complex and resource demanding that it will most likely put off most small and medium sized organizations.

Despite these problems, assessment models can, nevertheless, be useful in that they provide a common language for organizational problems and a vision of what the organization could be like in the future. They can also help organizations evaluate themselves and assist in setting improvement priorities.

Conclusions

In this paper, we have presented an overview of software process assessment within the context of software process improvement. The crucial point is never to forget why process improvement is important. Assessment models such as ISO/IEC TR 15504 and CMM can be of valuable help to improve an organization's software processes. However, the achievement of a maturity level should be a measure of improvement, not the goal of improvement. A certificate on the wall does not necessarily mean that the processes actually being performed are efficient or conforming to the standard.

To remain competitive, software organizations must not only keep pace with the rapid technological evolution; it must also focus attention on the organization's ability to deal with change. What matters most, however, are the business needs and business goals of the improvement effort, the impact on cost, cycle time, productivity, quality, and – most importantly – customer satisfaction.

In short, we would like to conclude with a five hundred year old advice, which is equally important today:

"Whoever desires constant success must change his conduct with the times".

Niccolò Machiavelli,
The Discourses, III

References

1 Paulk, M C et al. *The capability maturity model : guidelines for improving the software process.* Reading, MA, Addison-Wesley, 1995.

2 Paulk, M C et al. *Capability maturity model for software, version 1.1.* Pittsburgh, PA, Software Engineering Institute, 1993. (Technical Report CMU/SEI-93-TR-24.)

3 Paulk, M C et al. *Key practices of the capability maturity model, version 1.1.* Pittsburgh, PA, Software Engineering Institute, 1993. (Technical Report CMU/SEI-93-TR-25.)

4 El Emam, K, Drouin, J-N, Melo, W (eds.). *SPICE : the theory and practice of software process improvement and capability determination.* Los Alamitos, CA, IEEE Computer Society Press, 1998.

5 McFeeley, B. *IDEAL : a user's guide for software process improvement.* Pittsburgh, PA, Software Engineering Institute, 1996. Handbook CMU/SEI-96-HB-01.

6 ISO. *Information technology : software process assessment. Part 7 : guide for use in process improvement.* Geneva, 1998. (ISO/IEC TR 15504-7.)

7 ISO. *Information technology : software process assessment. Part 9 : vocabulary.* Geneva, 1998. (ISO/IEC TR 15504-9.)

8 ISO. *Quality management and quality assurance : vocabulary.* Geneva, 1994. (ISO 8402.)

9 ISO. *Information technology : software process assessment. Part 1 : concepts and introductory guide.* Geneva, 1998. (ISO/IEC TR 15504-1.)

10 Pressman, R S. *Making software engineering happen : a guide for instituting the technology.* Englewood Cliffs, CA, Prentice Hall, 1988.

11 Humphrey, W S. *Managing the software process.* Reading, MA, Addison-Wesley, 1989.

12 Humphrey, W S. *Managing technical people : innovation, teamwork, and the software process.* Reading, MA, Addison-Wesley, 1997.

13 Olsen, T, Humphrey, W S, Kitson, D. *Conducting SEI-assisted software process assessments.* Pittsburgh, PA, Software Engineering Institute, 1989. (Technical Report CMU/SEI-89-TR-07.)

14 Kuvaja, P et al. *Software process assessment and improvement : the BOOTSTRAP approach.* Oxford, Blackwell, 1994.

15 ISO. *Information technology : software process assessment. Part 4 : guide to performing assessments.* Geneva, 1998. (ISO/IEC TR 15504-4.)

16 Zahran, S. *Software process improvement : practical guidelines for business success.* Harlow, Addison-Wesley, 1998.

17 Dunaway, D K, Masters, S. *CMM-based appraisal for internal process improvement (CBA IPI) : method description.* Pittsburgh, PA, Software Engineering Institute, 1996. (Technical Report CMU/SEI-96-TR-007.)

18 Rout, T P, Simms, P G. Introduction to the SPICE documents and architecture. In: *SPICE : The Theory and Practice of Software Process Improvement and Capability Determination.* El Emam, Drouin and Melo (eds.). Los Alamitos, CA, IEEE Computer Society Press, 1998.

19 Anastasi, A, Urbina, S. *Psychological testing.* Upper Saddle River, NJ, Prentice-Hall, 1997.

20 Nunnally, J C. *Psychometric theory.* New York, McGraw-Hill, 1978.

21 Cronbach, L. Coefficient alpha and the internal consistency of tests. *Psychometrica,* 16, 297–334, 1951.

22 Goldenson, D R et al. *Empirical studies of software process assessment methods.* Freiburg, Frauenhofer Institute for Experimental Software Engineering, 1997. (Technical Report ISERN-97-09.)

23 Bollinger, T, McGowan, C. A critical look at software capability evaluations. *IEEE Software,* 8 (4), 25–41, 1991.

24 Humphrey, W S, Curtis, B. Comments on 'A Critical Look'. *IEEE Software,* 8 (4), 42–46, 1991.

25 Humphrey, W S, Sweet, W. *A method for assessing the software engineering capability of contractors.* Pittsburgh, PA, Software Engineering Institute, 1987. (Technical Report CMU/SEI-87-TR-23.)

26 El Emam, K, Madhavji, N H. The reliability of measuring organizational maturity. *Software Process Improvement and Practice,* 1 (1), 3–25, 1995.

27 Fusaro, P K, El Emam, K, Smith, B. *The internal consistency of the 1987 SEI maturity questionnaire and the SPICE capability dimension.* Freiburg, Frauenhofer Institute for Experimental Software Engineering, 1997. (Technical Report ISERN-97-01.)

28 El Emam, K. *The internal consistency of the ISO/IEC 15504 software process capability scale.* Freiburg, Frauenhofer Institute for Experimental Software Engineering, 1998. (Technical Report ISERN-98-06.)

29 El Emam, K, Goldenson, D R. SPICE : an empiricist's perspective. In: *Proceedings of the Second IEEE International Software Engineering Standards Symposium,* August 1995. Los Alamitos, CA, IEEE Computer Science Press, 1995, 84–97.

30 Herbsleb, J et al. *Benefits of CMM-based software process improvement : initial results.* Pittsburgh, PA, Software Engineering Institute, 1994. (Technical Report CMU/SEI-94-TR-13.)

31 El Emam, K, Goldenson, D R. An empirical evaluation of the prospective international SPICE standard. *Software Process Improvement and Practice,* 2 (2), 123–148, 1996.

32 SPICE. *Phase 1 Trials Interim Report, Version 1.00.* 15 July, 1998.

33 SPICE. *Phase 2 Trials Interim Report, Version 1.00.* 17 June, 1998.

34 Coletta, A. Process assessment using SPICE : the assessment activities. In: *SPICE : The Theory and Practice of Software Process Improvement and Capability Determination.* El Emam, Drouin and Melo (eds.). Los Alamitos, CA, IEEE Computer Society Press, 1998.

35 Basque, R. CBA IPI : How to build software process improvement success in the evaluation phase? *IEEE TCSE Software Process Newsletter,* (5), Winter, 1996.

36 Peterson, B. Software Engineering Institute. *Software Process Improvement and Practice, Pilot Issue,* 1995.

37 Gremba, J, Myers, C. The IDEAL model : a practical guide for improvement. *Bridge,* (3), 1997.

38 Jansen, P, Sanders, J. Guidelines for process improvement. In: *SPICE : The Theory and Practice of Software Process Improvement and Capability Determination.* El Emam, Drouin and Melo (eds.). Los Alamitos, CA, IEEE Computer Society Press, 1998.

39 Deming, W E. *Out of the crisis.* Cambridge, MA, Cambridge University Press, 1986.

40 El Emam, K et al. *Success or failure? Modeling the likelihood of software process improvement.* Freiburg, International Software Engineering Research Network, 1998. (Technical Report ISERN-98-15.)

41 Goldenson, D R, Herbsleb, J. *After the appraisal : a systematic survey of process improvement, its benefits, and factors that influence success.* Pittsburgh, PA, Software Engineering Institute, 1995. (Technical Report, CMU/SEI-95-TR-009.)

42 Basili, V R et al. The software engineering laboratory : an operational software experience factory. In: *Proceedings of the 14th International Conference on Software Engineering.* New York, ACM, 1992.

43 Basili, V R, Weiss, D. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering,* 10 (6), 728–38, 1984.

44 Basili, V R. Software development : a paradigm for the future. In: *Proceedings of the 13th Annual International Computer Software & Applications Conference (COMPSAC).* Los Alamitos, CA, IEEE Computer Society Press, 1989, 471–485.

45 Dybå, T, Skogstad, Ø. Measurement-based software process improvement. *Telektronikk,* 93 (1), 73–82, 1997.

46 Ginsberg, M, Quinn, L. *Process tailoring and the software capability maturity model.* Pittsburgh, PA, Software Engineering Institute, 1995. (Technical Report CMU/SEI-94-TR-024.)

Tore Dybå (37) is Research Scientist at SINTEF Telecom and Informatics. He holds a Master's Degree in Computer Science and Telematics from the Norwegian Institute of Technology in 1986. Tore Dybå is also Research Fellow in Computer Science at the Norwegian University of Science and Technology working on a Ph.D. thesis on Software Process Improvement and Organizational Learning.

e-mail: Tore.Dyba@informatics.sintef.no

Reuse of software development experience – a case study

MAGNE JØRGENSEN, DAG SJØBERG AND REIDAR CONRADI

In this paper we describe how Telenor Telecom Software (TTS) developed and implemented processes, roles and tools to achieve reuse of estimation and risk management experience, i.e. organizational learning. The results from the case study include:

- **The development and introduction of an experience database integrated with the software development process – offering relevant experience 'just in time';**
- **Examples of types of experience useful for software developers;**
- **Recommendations on how to collect, package and distribute experience;**
- **Experience on roles and process to support reuse of software development experience.**

1 Introduction

The reported case study on reuse of software development experience was carried out in 1997–1998, supported by the national research project SPIQ (Software Process Improvement for better Quality). Among other things, the case study was motivated by the following challenges:

1 How can software development experience be efficiently shared between different development teams?

2 What types of experience are worth reusing?

3 What is the role of reuse of 'local' (context-dependent) experience compared with more 'global' (best practice) experience?

Our approach and results to help meeting these challenges, we believe, can be useful for other organizations facing similar challenges.

The remainder of the paper is organized as follows. Section 1.1 describes the research project SPIQ. Section 1.2 describes the organization studied. Section 2 describes and argues for approach chosen. Section 3 describes the results. Section 4 describes related work. Section 5 concludes, summarizes and suggests further work.

1.1 Software Process Improvement for better Quality (SPIQ)

In April 1997, following a pre-project in 1996, the software process improvement project SPIQ was started. The program is sponsored by the Research Council of Norway (NFR) for at least three years. Its main goal is to

> *"... increase the competitiveness and profitability of Norwegian IT industry through systematic and continuous process improvement ..."*

The SPIQ project is based on the software process improvement principles of 'Total Quality Management', see for example [1], and the 'Quality Improvement Paradigm', see for example [2]. An important aspect of SPIQ is that it provides a means for the academia and the software industry to meet and discuss software improvement experiences and research results.

The work described in this paper has benefited from SPIQ in at least three ways:

1 The experience database design and results were discussed at the SPIQ meetings;

2 SPIQ has provided valuable research support;

3 SPIQ has financed parts of the Telenor Telecom Software's (TTS') internal work on 'reuse of experience'.

See http://www.fw.no/spiq/ for more information on SPIQ.

1.2 The organization

TTS is split into five geographical locations and has more than 400 employees, most of them software developers. In other words, reuse of software development experience is an important but not trivial task. In 1995–1996 the company went through a 'Business Process Reengineering', see [3], resulting in a well documented, standardized software development process. The process descriptions and documents are available to all employees through the Intranet using an Internet browser.

The software development process used by the developers is called 'solution delivery' and is based on incremental delivery of software functionality in so-called 'time-boxes'. Each 'time-box' lasts 3–6 month, which provides good conditions for experience reuse, at least compared with organizations with a waterfall development model leading to projects with cycles of 1–2 years.

The organization includes several support teams (development tool support team, measurement and estimation support team, test support team, quality team, etc.) for the development and maintenance processes. These teams turned out to be very important in the implementation of the process changes and collecting experience.

A recent, informal, in-house assessment (carried out by one of the authors of this paper) of the company, in accordance with the CMM framework, gave maturity levels on different key process areas between 2 and 4, i.e. TTS is a reasonably mature software development organization.

The company's software development process prescribes several steps motivated by the need for reuse of development experience: Each project should 1) be measured according to a measurement model, and 2) deliver an experience report when completed. The 'Measurement and Estimation Team' was allocated to carry out the measurements and the 'Quality Team' was the receiver of the experience reports.

We found that the project measurement and the experience reporting were to some extent carried out. However, there were not much systematic use of the information to improve the process. This observation was a major motive for our focus on reuse of experience in TTS.

2 The approach

Our approach can be characterized as action science [4], which is a typical research method when studying industrial software development. Action science has advantages as well as disadvantages. The advantages are, for example, that action science may be the most efficient way to get:

- In-depth knowledge of software development organizations. This belief is e.g. supported by the learning model of [5], which focuses on the role of collecting concrete and context-dependent experience to support the learning process. According to this learning model only the lower levels of knowledge is context-independent and rule-based. In order to achieve higher levels of knowledge (being an expert) lots of context-dependent experience (local experience) has to be collected. *Our observations support this learning*

model. For example, while inexperienced project leaders asked for rule based methods regarding risk management, more experienced project leaders were more interested in how other projects had carried out their risk management activities.

- Representative and realistic information on how terms and models important for meaningful reuse of experience are used. *For example, when we cooperated with the projected leaders on estimation of effort, we found a variety of interpretations of the term 'effort estimate'. This variety clearly reduced the potential for reuse of the effort estimation experience and data. Three major types of interpretations were found: Estimated effort means a) 'most likely effort'; b) 'the effort with the probability of 50 % not to exceed' (median); or c) 'the most likely effort + a (project dependent) risk buffer'.*

Disadvantages of action science are, on the other hand, that:

- Action science studies are not carried out as strict experiments with control of the variables. Thus, a formal cause-effect relationship between the actions and the results cannot be established. In particular, the mixing of the participation and observer role makes objective analyses difficult. In addition, it is unlikely that anyone will (be able to) repeat the study to validate our observations.
- There is no available observational language or theory to remove subjectivity and bias in the description of the observations. See for example the discussion of how the expectations impact the observational language in [6] – i.e. there is a danger of 'theory loaded observations'.

It is important to be aware of these disadvantages, but it should not stop anyone from carrying out studies like ours. Currently, action science (or similar methods) seems to be the only practical way of achieving in-depth 'real-world' results about software improvement. We believe, however, that more quantitative and experimental research on software processes should be the long-term goal of the software improvement research, leading to more general and objective knowledge. A more general discussion and comparison of research methods, particularly the role of case studies, can be found in [7].

Stimulated by the work at NASA-Software Engineering Laboratory on Experience Factory, see for example [8], and the opportunities we had at TTS, we started a search for 'pilots' where reuse of experience would improve the development process. Based on an informal analysis of the availability of information, availability of resources, time, probability of success, estimated cost and benefit, we decided to focus on the following two topics within the software development process:

- Estimation of software development effort;
- Risk management.

A brief analysis gave that in order to support reuse of estimation and risk management experience, there was a need for

- An experience reuse process, including new or modified role descriptions;
- A supporting tool (the experience database);
- Allocated experience reuse resources, both for implementing the experience reuse processes and for administrating the experience database.

3 The results

This section describes the work and some of the results achieved in the period Spring 1997 – Spring 1998. The organization continues to focus on experience reuse, i.e. the results and products are to some extent preliminary.

3.1 Manifestation of experience

During the requirement analysis we soon discovered that the manifestation of experience can and should take many forms to be useful to the developers, such as:

- Quantitative and qualitative information that can be stored in traditional databases;
- General tools implementing or based on 'best practice' within the organization;
- Rule based systems (expert systems) reflecting expert experience and knowledge;
- Pointers to people with useful experience (this may be the only way of 'representing' experience that cannot be articulated, i.e. tacit knowledge);
- Process descriptions on different levels and with different degrees of context dependence.

In addition, it was considered important that the experience database (the tool enabling the access to the stored experience) was available to all the developers at a low cost, integrated with the quality system, easy to use and easy to maintain.

3.2 Technical platform

The technical platform chosen to meet these requirements was based on

- The organization's own Intranet. This made the experience database available to all the developers and well integrated with the organization's quality system;
- A user interface based on a web-browser with links to experience of different types. This removed the need for local installation;
- An 'experience database' based on tables of data, spreadsheets, documents and rules implemented in executable programs, i.e. no traditional database.

Further, we decided to integrate the experience reuse support with the organization's process descriptions, i.e. from the relevant steps in the process descriptions we had links to useful information and tools in the experience database. The idea was to offer useful experience 'just in time'.

3.3 Reuse of effort estimation experience

The effort estimation experience we offered was of the following types (linked to the relevant process steps):

3.3.1 Determine the appropriate estimation model and process

An 'expert system' recommending one or more estimation models was developed based on the collection and analysis of the experience of the organization's estimation experts. Following an analysis of whether formalized effort estimation is recommended or not, the expert system asks the user to answer nine questions. A simplified description of the questions and some implications of different answers are indicated in Table 1. The estimation models are briefly described in Section 3.3.2. This expert system uses,

in addition to the answers from the users, empirical data from TTS on the accuracy of the different estimation models, see Table 2, and the quality of the relevant historical data, i.e. a high degree of organizational dependent experience.

3.3.2 Estimate effort

Depending on estimation model, different types of experience data are available. Among others, the following estimation models and planning tools were supported by the experience database:

a) MarkII Function Point Analysis (MkII FPA), see [9]. We improved and extended an existing spreadsheet implementing the MkII FPA estimation model. This estimation model takes as main input the estimated size of the functionality to be developed in function points.

Earlier we had analyzed data from more than 30 software development projects regarding how different variables, such as use of CASE tool, had had an impact on the development productivity, see [10]. This study indicated that the choice of development environment explained most of the productivity variance. Now we provided the estimator with historical data on previous projects similar to the current project. Table 3 shows some of the historical information that the estimator could make use of. The productivity is measured as UFP/w-h, unadjusted function points per work hour. Notice that the estimator has to predict a productivity category for his project, i.e. expert knowledge is still required.

b) A bottom up, task and risk based estimation model was developed. This estimation model was supported with experience in the form of lists of 'tasks to remember' and suggestions on the effort distribution between the phases. Currently, there is ongoing work on how to improve the collection and reuse of historical data to support this bottom-up, task and risk based estimation model, see [11]. We labeled this model *ROPD* (the Norwegian acronym for Risk Based Division into Sub-tasks).

c) A risk analysis tool integrated in the estimation tools (or to be used separately) was developed. The risk analysis tool contains risk models, textual advice and guidelines based on previous experience. The content varies from a simple (but useful) checklist of tasks and risk factors to more sophisticated probability (beta-distribution) based risk models. Typically, the content was based on general frameworks and models, then adapted to the organization's needs according to expert knowledge and experience. This tool resulted in a probability based effort estimate and predictions such as "there is an 80 % probability of not exceeding 3000 w-h of effort". It turned out that this type of probability based predictions were essential to introduce the distinction between planned and estimated effort in the organization. Similar to the results in [12] we found that probability based estimation had a positive impact on the realism in the effort estimates. Finally, pointers to the human estimation experts were provided.

Table 1

Question	Examples of implication
1) Will there be a high degree of infrastructure development and/or complex algorithms?	YES: FPA (function point analysis) based estimation is not recommended.
2) Is the project context significantly different from previous TTS-projects?	YES: Previous experience (collected productivity data) will not be of much use. Normally, this excludes the use of FPA.
3) Are most of the requirements described?	NO: The work intensive estimation models ROPD (risk based, bottom-up estimation) and FPA are not worth the effort.
4) Is a data model available?	NO: The simplified FPA version (useful when developing an early estimate) cannot be used.
5) Does the delivery consist of many small, not logically connected changes/modules?	YES: FPA may not be useful.
6) Will the effort to complete the project probably take more than six months?	NO: FPA may require too much effort.
7) Is the project willing to spend 1–2 man-days of effort for small projects (less than 12 man-months) and 2–4 man-days for larger projects?	NO: ROPD or simplified FPA may be too work intensive.
8) Will developers with experience from similar projects be available when estimating the effort?	NO: ROPD requires a division of tasks into sub-tasks, i.e. without experience from similar projects ROPD can hardly be used.
9) Will there be more than five deliveries similar to this one?	YES: If none of the standard estimation models are recommended, a tailored estimation model should be developed.

Table 2

Estimation model	TTS historical accuracy of model (average)
Full Function Point Analysis	+/- 15 % (mean magnitude of error)
Simplified Function Point Analysis	+/- 30 % (mean magnitude of error)
ROPD	+/- 20 % (mean magnitude of error)

Table 3

Batch-development	**Low prod.**	**Medium prod.**	**High prod.**	**Turbo prod.**
Cobol environment	0.05 UFP/w-h	0.10 UFP/w-h	0.20 UFP/w-h	0.30 FP/w-h
Powerbuilder environment	0.15 UFP/w-h	0.25 UFP/w-h	0.50 UFP/w-h	0.70 UFP/w-h
On-line development	**Low prod.**	**Medium prod.**	**High prod.**	**Turbo prod.**
Cobol environment	0.07 UFP/w-h	0.15 UFP/w-h	0.20 UFP/w-h	0.30 UFP/w-h
Powerbuilder environment	0.20 UFP/w-h	0.35 UFP/w-h	0.70 UFP/w-h	1.00 UFP/w-h

3.4 Reuse of risk management experience

Similar to the estimation support we linked our experience database to the risk management process. The experience database offers support through several tools to identify, analyze and manage software project risks. We interviewed several experienced project leaders in the organization to get the most relevant risk factors and the most relevant methods to reduce and control the risks. In addition, data from quality revisions were used to tailor the risk management support.

Based on the collected information we developed:

- A 'TTS best practice' risk management process (extensions to the existing development process);
- A tool to identify, assess and store risk factors, and suggestions on how to reduce or control the risks;
- A tool to visualize the risk exposure over time.

In many ways, what we did was to collect only a small fraction of the organization's knowledge about risk management. To become a learning organization the organization will need to continuously collect and distribute experience, i.e. new roles and a changed process are needed. Since systematic experience reuse in risk management has a short history in TTS, we found that we needed to start small in order to understand what sort of risk experience would be useful to collect.

3.5 Roles and process

The studies and results described earlier in this paper resulted in the identification of needs for new roles and an increased focus on the implementation of the development process.

Roles

- An *'experience database administrator'* (a 'gardener') responsible for the availability and usability of the experience to be reused. This role may be split into two roles dividing the responsibility into a technical administrator and a content administrator. We suggest that the 'gardener' should be a part of the software process improvement team of the organization.
- Several *'process analysts'* responsible for analysis of information from each sub-process, such as the estimating process, the project management process or the testing process. The 'process analysts' is responsible for collecting and analyzing relevant information from completed projects and to generalize, tailor and package the useful experience.
- A network of *'support teams'* teaching and guiding the project leaders and members how to properly reuse the experience within each sub-process/ topic.
- A *process owner* for the experience reuse process.

Notice the distinction between role and person. In a small organization a small team or (at least in theory) one single person may fill all these roles. Based on our experience at TTS, a critical minimum central effort to enable substantial reuse of estimation and risk management experience seems to be 2–3 man-years to fill the roles above.

Process

When we started our study, the organization did collect project data and it was mandatory to write experience reports, i.e. the process description had elements of experience reuse. However, the collected information was not systematically used to improve the processes. In other words, the process (or even more, the implementation of the process) had not had enough focus on the use of the collected information. Looking at other case studies of software process improvement, see for example [13], this seems to be a typical problem leading to graveyards of data and unused documents. In our opinion, this is a situation even worse than the situation where no data is collected and no reports written, and there is probably no more efficient way of destroying the respect for a measurement and experience report.

We believe that the current process description of TTS is sufficient to enable experience reuse, given sufficient resource to fill the experience reuse roles described earlier. For a more general experience reuse process and organization, see [8].

3.6 Benefits

An underlying initial hypothesis on experience reuse is, of course, that it has a long term benefit higher than the costs. Currently, we are not in the situation to decide whether this is true or not. We cannot validate the hypothesis, partly because it is too early, and partly because it is difficult to isolate the impact of our work from the impact of other parallel process improvement initiatives. However, even without a formal impact study we believe to see the following results of the experience reuse work:

- Improved estimation accuracy and more widespread use of the estimation models;
- An increased focus on experience based risk management in the projects;
- An acceptance in the organization for the need to collect and share experience.

In addition, we have made a number of interesting observations increasing the probability of successful reuse of experience in TTS, such as:

- Currently, the experience reports written by the projects were of little use to other projects. This may indicate that without a clear model on how the experience will be reused, there is a great danger of reporting and collecting useless information.
- The mere focus on reuse of experience had a positive impact on the 'improvement culture' in the organization. It would have been very interesting to carry out controlled experiments on how different actions impact the software improvement culture. An experimental design similar to the one described in 'Goals and performance in computer programming' [14] may be appropriate.

4 Related work

The Experience Factory or EF [15, 16] is a framework for reuse of software life cycle experiences and products. EF relies on the Quality Improvement Paradigm [17] for continuous and goal-oriented process improvement, resembling the Shewhart/Deming Plan-Do-Check-Act cycle [18].

The EF framework prescribes an improvement organization inside a company, a kind of 'extended quality department'. This implies the "logical separation of project development (performed by the Project Organization) from the systematic learning and packaging of reusable experiences (performed by the Experience Factory)" [16]. The PERFECT EF framework extends this model by adding a third organizational component: the Sponsoring Organization, which uses the EF for strategic purposes [19].

Within the EF framework, the NASA-Software Engineering Laboratory with its 275 developers has collected information about 150 projects in the period 1976 – 1996. The purpose is to record the effects of various software technologies (methods, tools, programming languages, QA techniques, etc.). However, NASA represents a special kind of stable and resourceful organization. It is a challenge to apply the EF ideas outside of NASA, i.e. to downscale it to companies with typically 10–30 developers, and where the EF roles are partly being played by the developers themselves. More applications of the EF framework in other contexts are therefore needed, see e.g. [19]. Our case study is a contribution in that respect.

5 Conclusions

> *"... (improvement) requires continual accumulation of evaluated experiences, in a form that can be effectively understood and modified, sorted in a repository of integrated experience models, that can be accessed/modified to meet the needs of the current project."* [15]

In the introduction (Section 1) we asked the following questions:

1 *How can software development experience be efficiently shared between different development teams?*

2 *What types of experience is worth reusing?*

3 *What is the role of reuse of 'local' (context-dependent) experience compared with more 'global' (best practice) experience?*

Through our study we have contributed to the answers, but cannot claim to have the answers. Our main contribution may have been to give an in-depth example of how the questions/challenges were approached by TTS.

TTS has introduced a standardized development process documented on the web and made the processes available for all the software developers through the organization's Intranet. In many ways, this opens new possibilities for software development organizations. We have found that software development experience efficiently can be linked to the process steps and made available to all the developers in a very flexible way. However, the main challenges regarding becoming a learning organization and reusing experience is not the technology. We found that a lot of 'trial and error' and pragmatism is needed to find the useful experience and ways to formulate and spread this experience.

We found it useful to be very pragmatic regarding the manifestation of experience. For example, a very useful information in our experience database was the links to the experts having the required experience. Regarding the role of local (organization dependent) experience vs. best practice experience we found that the local experience made the best practice processes significantly more useful. In other words, optimal use of best practice processes seems to require collection and reuse of more local experience.

Achieving a learning organization is a formidable task. Senge claims that the following five disciplines are essential to creating learning organizations: *personal mastery, mental models, shared visions, team learning* and *systems thinking* [20]. An experience database like the one we have designed and implemented in TTS can serve as a basis for activities involved in all five disciplines. An experience database is also a useful means to agree on a common understanding of the current situation. *"An accurate, insightful view of current reality is as important as a clear vision."* [20]

Future work will address the major issue of how projects (contexts) should be characterized so that experiences collected in one project (context) are applicable to another project (context). How can we judge whether a project is sufficiently similar to (a subset of) the projects for which we have experience? The approaches described in [16] will be taken as a starting point.

Acknowledgments

The authors wish to thank the TTS employees Pål Woje, Geir Ove Espås, Majeed Hosseiney, Oddmar Aasebø and Tor Larsen for their enthusiasm and contribution to the work described in this paper.

References

1 Deming, W E. *Out of the crisis.* MIT Center for Advanced Engineering Study, Cambridge, MA, MIT Press, 1986.

2 Basili, V R. Quantitative evaluation of software engineering methodology. In: *Proceedings of the First Pan Pacific Computer Conference, Melbourne, Australia,* 1985.

3 Hammer, M. *Beyond reengineering.* NY, Harper Collins, 1996.

4 Argyris, C et al. *Action science : concepts, methods and skills for research and intervention.* San Francisco, CA, Joosey-Bass, 1985.

5 Dreyfus, H, Dreyfus, S. *Mind over machine : the power of human intuition and expertise in the era of the computer.* NY, Free Press, 1986.

6 Goodman, N. *The structure of appearance.* Cambridge, Mass., Harvard University Press, 1951.

7 Flyvebjerg, B. *Rationalitet og magt : det konkretes videnskap (bind I).* Copenhagen, Akademisk Forlag, 1991.

8 Basili, V R et al. The software engineering laboratory : an operational software experience factory. In: *Proceeding of the 14th international conference in software engineering,* Melbourne, 1992, 370–381.

9 Symons, C R. *Software sizing and estimating, MkII FPA.* NY, Wiley, 1993.

10 Jørgensen, M. Empirical evaluation of CASE tool efficiency. In: *Proc. Sixth Int. Conf. on applications of Software Measurement, Orlando,* 1995, 207–230.

11 Schrader, T. *A bottom-up project cost estimation method using historic data and a standardized work breakdown structure.* Trondheim, The Norwegian University of Science and Technology, 1998. (Project Report.)

12 Conolly, T, Dean, D. Decomposed versus holistic estimates of effort required for software writing tasks. *Management Science,* 43 (7), 1029–1045, 1997.

13 Cusumano, M A, Selby, R W. *Microsoft secrets.* London, Harper Collins Business, 1996. ISBN 0006387780.

14 Weinberg, G, Shulman, E. Goals and performance in computer programming. *Human Factors,* 16, 1974.

15 Basili, V R. The experience factory and its relationship to other improvement paradigms. In: I Sommerville and M Paul (eds.). *Proc. From ESEC'93, 4th European Software Engineering Conference,* Garmisch-Partenkirchen, Germany, September 1993. Springer-Verlag, 1993, 68–83. (Lecture Notes in Computer Science 717.)

16 Basili, V, Briand, L, Thomas, W. Domain analysis for the reuse of software development experiences. In: *Proc. of the 19th Annual Software Engineering Workshop,* NASA/GSFC, Greenbelt, MD, 1994.

17 Basili, V R, Rombach, H D. The TAME project : towards improvement-oriented software environments. *IEEE Transactions on Software Engineering,* 14 (6), 758–773, 1988.

18 Deming, W E. *Quality, productivity, and competitive position.* Cambridge, Mass., Massachusetts Institute of Technology Center for Advanced Engineering Study, 1982.

19 PERFECT Consortium. *PIA Experience Factory, The PEF Model.* ESPRIT Project 9090, D-BL-PEF-2-PERFECT9090, 1996.

20 Senge, P M. *The fifth discipline : the art and practice of the learning organization.* Currency/Doubleday, 1995.

Magne Jørgensen (34) received his Dr.Scient. degree in informatics from the University of Oslo in 1994. From 1989 to 1998 he was a research scientist at Telenor Research, and since 1995 an associate professor at the University of Oslo. His research interests are in software engineering, empirical studies and process improvement. Since 1998 he has been leader of a software development process improvement group at Storebrand.

e-mail: magne.jorgensen@storebrand.no

Dag Sjøberg (38) received his Ph.D. in computing science from the Univ. of Glasgow in 1993. From 1993 to 1995 he held a post-doc scholarship at the Univ. of Oslo funded by the Research Council of Norway. Since 1995 he has been an associate professor at the Univ. of Oslo. From 1985 to 1989 he worked as a system developer and group leader at the Norwegian State Hospital and Statistics Norway. His research interests are software engineering, empirical studies, process improvement and object-oriented technology.

e-mail: Dag.Sjoberg@ifi.uio.no

Reidar Conradi received his Ph.D. in computer science from the Norwegian University of Science and Technology (NTNU) in 1976. From 1972 to 1975 he worked at SINTEF as a researcher. Since 1975 he has been assistant professor at NTNU and a full professor since 1985. He has participated in many national and EU projects and chaired several workshops. His research interests are in software engineering, object-oriented methods and software reuse, distributed systems, software evolution and configuration management, software quality and software process improvement.
e-mail: Reidar.Conradi@idi.ntnu.no

An empirical study of the correlation between development efficiency and software development tools

MAGNE JØRGENSEN AND SIGRID STEINHOLT BYGDÅS

There are many opinions and claims about how CASE tools impact the software development and maintenance efficiency, but few empirical studies. In this paper we describe a method for CASE tool efficiency evaluation, apply the method to evaluate four different CASE tools, and report the findings. Interesting findings were, for example, that:

- **The CASE tools had a strong and systematic impact on the development and maintenance efficiency, i.e. the choice of CASE tool is important.**
- **The impact of the CASE tools on the maintenance efficiency was larger than the impact of the CASE tools on the development efficiency.**
- **A high development efficiency was not always followed by a high maintenance efficiency.**

There have been surprisingly few empirical studies evaluating CASE tools[1)], see [1] for an overview. In addition, most of the studies carried out have focused on subjective measures of the developer's perceptions, such as the study reported in [2], rather than on objective measures of completed systems developed and maintained by professional software developers. For this reason, the software managers have not had much support from the software research when comparing CASE tool efficiency.

In this paper we describe the method and the results of a study comparing CASE tools, applying objective measures on completed, medium large, administrative software applications. The paper is organised as follows:

- Section 1 describes the evaluation method and relates it to the general evaluation process model described in the standard *IEEE Recommended Practice for the Evaluation and Selection of CASE Tools* [3];
- Section 2 reports the results of the evaluation;
- Section 3 discusses the quality and generality of the evaluation method; and
- Section 4 concludes and summarises the contributions.

1 The evaluation method

The CASE tool evaluation activity described in the evaluation process model of [3] comprises the following five steps:

1 *Prepare an evaluation task definition statement.* This statement should include: a) the purpose of the evaluation, b) the scope of the evaluation, c) explicitly stated assumptions and constraints of the evaluation, and d) a description of the evaluation activities.

2 *Select and define the evaluation criteria.* The criteria should be reviewed to ensure consistency with the task definition statement.

3 *Select the CASE tools to be evaluated.*

4 *Collect the data.*

5 *Evaluate the CASE tools relative to the criteria*, and *report the results.*

Section 1.1 – 1.4 describes how we carried out Step 1 – 4.

1.1 Task definition statement

The two main *purposes* of the CASE tool evaluation were:

- To develop and validate a method for the evaluation of how CASE tools impact software development and maintenance[2)] efficiency. The method should be based on simple, objective and easy to interpret measures;
- To study the impact of a few selected CASE tools on software development and maintenance efficiency.

The method we describe is based on the framework described in the general evaluation framework *'IEEE Recommended Practice for the Evaluation and Selection of CASE Tools'* [3]. The main value we add to this framework is the measures and techniques for evaluating the development and maintenance efficiency of CASE tools, and an empirical validation of these measures and techniques. In a *total evaluation* of a CASE tool, the IEEE framework can be applied for the issues not addressed in our method.

The *scope* of the evaluation can be described as:

- Only CASE tools currently present in the organisation subject to our study, Telenor, were evaluated.
- Only medium sized development projects were studied (on average approx. 3 man-years of effort).
- All the studied projects were developing administrative applications connected to a database.
- All the development projects had at least one developer with some previous experience with the use of the CASE tool. We did not try to estimate the degree of experience and skill of the developers. Instead, in accordance with the MkII FP method [4], we asked experienced developers to estimate the impact of any inexperience in use of the CASE tool on the efficiency. These estimates are part of the ENV factor in Table 2, Section 1.3.
- The evaluation is focused on the development and maintenance efficiency. Other criteria, such as vendor stability, application usability and hardware criteria, are important in a selection of a CASE tool, but are not part of the evaluation described in this paper.

The main *critical assumptions and limiting constraints* were:

- It is assumed that the similarity of the CASE tool supported development and maintenance projects enables a meaningful comparison of the CASE tools. This assumption is discussed in Section 3.
- It is assumed that the measures and techniques described in Section 1.2 are meaningful, for example that MkII Function Points is a meaningful measure of the user functionality produced. This assumption is discussed in Section 3.
- We had a limited number of available personnel time for evaluation activities. This constraint had the impact that an in-depth study collecting data from a high number of randomly selected development and maintenance projects

---

[1)] Definition: *A software tool that aids in software engineering activities, including but not limited to requirements analysis and tracing, software design, code production, testing, document generation, quality assurance, configuration management, and project management [3].*

[2)] *In this study we define software maintenance according to [5] as* the process of modifying existing operational software while leaving its primary functions intact, *i.e. major functionality changes on existing software are considered to be software development.*

was not possible to carry out. We had to choose between a superficial study of many projects or an in-depth study of a few (ten) projects. The latter option was chosen.

The *evaluation activities* were:

1 Identify simple, objective and easily interpretable measures of development and maintenance efficiency, see Section 1.2.

2 Decide on CASE tools and the CASE tool supported projects to study. Describe the main characteristics of the CASE tools and the projects, see Section 1.3.

3 Identify a data collection approach leading to high quality data. Document the approach in order to make the data collection process repeatable and the findings interpretable. The data collection approach is described in Section 1.4.

4 Collect the data.

5 Based on the collected data, analyse how the CASE tool impacts the efficiency.

6 Discuss the analysis results with the tool and project experts.

7 Analyse the quality of the evaluation, see Section 3.

8 Report the results, see Section 2 and Section 3.

1.2 The evaluation criteria

This section describes the measures and techniques applied in the measurement of the development (Section 1.2.1) and the maintenance (Section 1.2.2) efficiency.

1.2.1 Development efficiency measures

The development efficiency measures were based on the MkII Function Point (MkII FP) Analysis, as described in [4, 6 and 8], together with the measures Effort (E), Time (T) and Lines Of Code (LOC). The variables are described below:

- *MkII Function Points* measure the size of the applications in terms of its user required functionality. In spite of the similarities between Albrecht FP and Mk II FP, there is no simple relationship between these two functionality measures. This is, among others, caused by the different views on what Function Points are supposed to measure. While Symons [4] relates his MkII FP to the *effort* necessary to develop the functionality, Albrecth [8] relates his FP to the *value* of the functionality. The necessary steps to calculate the Unadjusted MkII Function Points (UFP) are (very briefly):

  1 *Define the system boundary, i.e. the border between the measured system and its users.*

  2 *Identify and categorise the entities into primary (business information) and non-primary entities.*

  3 *Identify the logical business transactions, i.e. the lowest level business processes supported by the system.*

  4 *Count the input types (I), output types(O) and the entity references (E) for each logical transaction.*

  5 *UFP = 0.58 * I + 1.66 * E + 0.26 * O. (The constants in this formula are based on an empirical study, see [4], and indicate the development effort relationship when developing "input-functionality", "data processing" and "output-functionality". For example, an underlying assumption is that one unit "input-functionality" requires more than twice as much effort compared to one unit "output-functionality".)*

- *Effort* (development) is a measure of the total effort in man-hours spent on a development or maintenance project, from the requirement phase to the installation phase, including user participation in the requirement and test phases.

- *Effort* (maintenance) is a measure of the estimated effort (average case) in man-hours to solve a specified maintenance task (a benchmark task). The effort estimate should include requirement analysis, design, implementation, inspections and tests. Administration, installation and documentation work were, due to the small size of the benchmark task, not included in our study.

- *Time* is a measure of the total calendar time on the development project, from the requirement phase to the installation phase.

- *Lines of Code* is a measure of the number of physical lines of source code written by a developer/maintainer, i.e. generated lines of code or comments were not counted. Reused code was only counted if there were major changes on the reused code. All source code was counted, inclusive the physical lines of code written in CASE tool forms (tool screens). Menu selections, navigation etc., were not counted.

Based on these variables we defined the following measures:

- **UDE:** UFP / E
  (Unadjusted Development Efficiency = Unadjusted Function Points / Effort)
- **ADE:** UDE * AF
  (Adjusted Development Efficiency = Unadjusted Development Efficiency * Adjustment Factor, AF is defined below)
- **WE:** LOC / UFP
  (Work Expansion = Lines of Code / Unadjusted Function Points)
- **WA:** UFP / LOC
  (Work Avoidance = Unadjusted Function Points / Lines of Code)
- **WS:** LOC / E
  (Work Speed = Lines of Code / Effort)
- **TP:** E / T
  (Time Pressure = Effort / Time)

Note that UDE = WA * WS, i.e. the Unadjusted Development Efficiency, equals Work Avoidance * Work Speed. This relation enables us to analyse whether a relatively high development efficiency is caused by a high Work Speed or a high Work Avoidance, or a combination of both.

In order to enable a fair comparison of the development efficiency between the different development projects, the Unadjusted Development Efficiency (UDE) may get adjusted (i.e. multiplied) with a number of standardised factors to get the Adjusted Development Efficiency (ADE). The inputs to the Adjustment Factor (AF) are briefly described below. A more detailed description can be found in [6].

AF = TCA * B * CF * ENV, where:

- **TCA** is the Technical Complexity Adjustments. TCA intends to measure the impact of the non-functionality software requirements on the efficiency. TCA is a value between 0.65 and 1.35 and the value is determined through the answers on 19 questions.
- **B** is the Batch-factor. B is based on the assumption that batch functionality on average requires 50 % more effort to develop, i.e. *B = (UFP on-line + 1.5*

** UFP batch) / (UFP on-line + UFP batch)*. B is therefore a value between 1.0 (pure on-line) and 1.5 (pure batch).

- **CF** is the Change factor. CF is based on the assumption that change of existing functionality on average requires twice as much effort as development of new functionality, i.e. *CF = (UFP new + 2 * UFP changed) / (UFP new + UFP changed)*. CF is therefore a value between 1.0 and 2.0.
- **ENV** is the Environmental factor. ENV intends to measure the impact of the environment factors on the productivity. ENV includes the developer's own estimates on the impacts of the factors 1) system size (system to be developed is significantly larger than any other previously developed by the organisation or project manager or team); 2) problem structure (for example, 're-implementation of an existing well-documented system' or 'geographically distributed project members'; and 3) technology (for example, 'first use of new method and tools'). ENV is a value between 0.66 and approx. 3.2. More about ENV in [6].

Consequently, the AF is between 0.43 and 13.0. However, the projects we studied had a much narrower interval; between 0.8 and 2.1.

The measure Work Expansion (WE) measures how many lines of code the developers on average must write to create an MkII Function Point. WE is defined the same way as Language Level, see for example [9]. A low WE means, in this context, that the combination of tool and language enables a high work avoidance (WA), i.e. less lines of code has to be written in order to produce the functionality.

Work Speed (WS) measures how fast in LOC per work-hour the developers work. WS is impacted by how easily lines of code are developed and/or changed, how appropriate the development method is, the efficiency of the CASE tool debugger, etc.

Time Pressure (TP) measures the effort in person-year per elapsed calendar-year. It is supposed to indicate the time pressure of the project. Time pressure is an important information when comparing the efficiency of different projects. It is clear, for example, that a project with one developer working one year on average will have a higher development efficiency compared to ten developers each working one month, given the same conditions.

1.2.2 Maintenance efficiency measures

The Function Point based development efficiency measures may also be applicable for measurement of the maintenance efficiency, and a set of rules for counting the size of changes is included in the MkII FP standard [7]. However, we believe that the measures based on Function Points are more meaningful for the measurement of created functionality of a (medium/large) software application than for the measurement of small change oriented tasks. One reason for our belief is exemplified in Section 3.1, where a maintenance task with four times as many MkII Function Points did not require more than 50 % additional effort. In other words, MkII Function Points may be a good measure of the *effort-related user required functionality*, but not of the *effort-related functionality of small change oriented tasks*.

For this reason, we developed a simple technique for determining the maintenance efficiency: 'The Software Maintenance Benchmark Task Technique'. The technique is described below. The technique assumes that the CASE tools to be evaluated, and a number of comparable applications developed applying these tools, have been selected.

The Software Maintenance Benchmark Task Technique

1 *Determine the common characteristics of the applications maintained applying the CASE tools.* In our study, all of the applications were connected to a database and had screens for user input and output.

2 *Develop typical maintenance tasks (the benchmark maintenance tasks) which are meaningful to carry out on all the applications, i.e. maintenance tasks based on the common application characteristics from step 1.* Appendix 1 outlines the specification of the seven benchmark tasks used in our study.

3 *For each of the selected applications, ask experienced maintainers to estimate the average effort needed and the average LOC to be written to solve the task.* It would, of course, be even better to ask the maintainers to carry out the maintenance tasks and then measure the effort and LOC. However, this may not be possible in a professional maintenance environment.

4 *Compare and analyse the estimates for each benchmark task.* The analysis will benefit from a comparison between the maintenance efficiency results and the development efficiency results.

Although this approach is simple we have not seen similar approaches to evaluate the maintenance efficiency. As opposed to traditional 'maintainability metrics', see [10] for an overview, our technique is based on the assumption that maintainability is not a characteristic of the application *alone*, but of the application, the tools and the maintenance tasks to be carried out.

Our main indicators of maintenance efficiency are the sum of estimated effort and the sum of the estimated size for the maintenance tasks on an application. In addition, we measure the work speed for maintenance tasks. Note that the work speed for the maintenance tasks uses another definition of effort (including less activities), and is not directly comparable with the development work speed.

- **MBTE:** Maintenance Benchmark Task Effort = Σ Effort on the benchmark tasks;
- **MBTS:** Maintenance Benchmark Task Size = Σ Lines of Code spent to solve the benchmark tasks;
- **WS:** Work Speed (maintenance) = MBTS / MBTE.

Comparing the maintenance efficiency of two CASE tools, the CASE tool with the lowest estimated effort on the benchmark tasks will have the highest maintenance efficiency.

1.3 CASE tools evaluated and applications studied

We evaluated four different CASE tools. A description of some of the main characteristics of the CASE tools, *the way they were used by the studied organisation,* is given in Table 1.

Ten software development projects were selected. We believe that the development projects are relatively similar, but there were, of course, some differences complicating the comparison. Table 2

contains a selection of development project characteristics.

All the projects developed administrative applications connected to a database. The distribution of the frequency of projects using different types of tools was a result of practical considerations.

1.4 The data collection

In order to get high quality data, the data were collected using interviews with the developers and studies of the applications, not questionnaires. We collected data from ten development projects and performed the maintenance benchmark test on seven applications, and spent in total about 25 man-days on the data collection.

If the quality of the project log system is not very high, we believe this amount of effort is necessary in order to collect high quality data. Data collected through questionnaires, we believe, may not have the required quality, see [11] for quality problems of questionnaire based software studies. We experienced, for example, that a meaningful and consistent collection of the effort spent on the development (including the user participation) required a lot of phone-calls, discussions and expert judgements.

Table 1 Overview of the CASE tools

CASE Tool	Tool A	Tool B	Tool C	Tool D
Informal description	A fourth generation language tool	A data oriented tool	A modern object oriented tool	A modern data oriented tool
Platform	MVS application servers	MVS application servers	Windows-clients + Unix application servers	Unix or Windows-clients + Unix application servers
Database type	Network	Network	Relational	Relational
Generated by the tool	Application code and screens	Application code, screens, db-schema and system doc.	Application code, screens, db-schema and system doc.	Screens, db-schema, and system doc.
Phases supported	Mainly implementation	Design and implementation	Mainly implementation	Design and implementation
Architecture of developed application	Mainframe-terminal	3-schema[3], mainframe-terminal	Client-server with presentation and parts of application logic on client	3-schema, client-server with presentation and parts of application logic on client
User interface of developed application	Character-based	Character-based	Windows-based	Windows or character-based
Main languages[4]	Tool specific 4GL + Cobol + Easytrive	Tool specific data oriented language + Cobol	Tool specific object oriented language + SAS + SQL	Tool specific data oriented language + C + SQL

2 The results

This section contains an overview of the development and maintenance efficiency results, see Table 3, and a more detailed description of the results on the development project level, see Table 4. A description of the measures can be found in Section 1.2.

The number of measured projects is not very high and not significant from a statistical point of view. Nevertheless, we believe that the results in Tables 2 and 3 indicate that:

- The Tool A supported projects had a low work avoidance, and a medium/ high work speed. This led to the lowest development and maintenance efficiency (i.e. the highest MBTE). The time pressure is higher than for most of the other projects, which may have decreased the efficiency.
- The Tool B supported project had a high work avoidance, but the lowest work speed. This led to a medium development efficiency, and the second highest maintenance efficiency. Time pressure is medium.
- The Tool C supported projects had the highest work avoidance and a high work speed for both development and maintenance. This led, with distance, to the best development and maintenance efficiency. Time pressure is lower than for the other projects, which may have increased the efficiency.
- The Tool D supported projects had a low work avoidance but a high work speed. This led to a relatively high development efficiency, but to a low maintenance efficiency. Time pressure is medium.
- The relative difference between the highest and the lowest maintenance efficiency (1:13) was higher than the relative difference between the highest and lowest adjusted development efficiency (1:4). This indicates that it may be even more important to focus on the maintenance efficiency than the development efficiency when evaluating CASE tools.

Table 4 contains the development and maintenance efficiency data for each of the ten projects. The value *n.m. (not measured)* means that, for practical reasons, we were unable to collect the data necessary to calculate the value.

In addition to the above data, we measured the UDE and the ADE on a Cobol development project. The Cobol project, which may be considered as a *baseline*

[3] *See [12].*

[4] *Frequently, the tool specific language does not meet all development needs, and external languages are needed. This row lists the tool specific and external programming languages typically included in a development project in the organisation applying the CASE tool.*

Table 2 Overview of the development projects

Development project	1	2	3	4	5	6	7	8	9	10
CASE Tool	Tool A		Tool B	Tool C			Tool D			
Size[5] [UFP]	1050	931	390	698	1631	1377	627	501	1180	150
Tech. Comp. Adj. (TCA)	0.88	0.99	0.90	0.87	0.88	0.87	0.85	0.87	0.87	0.92
Batch Factor (B)	1.01	1.15	1.02	1.05	1.00	1.03	1.08	1.01	1.00	1.06
Change Factor (CF)	1.00	1.00	1.66	1.00	1.00	1.00	1.49	1.00	1.00	1.75
Env. Factors (ENV)	1.52	1.37	0.80	0.92	0.92	1.16	1.57	1.10	0.99	1.13
AF=TCA*B*CF*ENV	1.35	1.57	1.22	0.84	0.81	1.04	2.14	0.97	0.86	1.93
Time Pressure (TP)	2.9	4.3	2.1	0.7	2.0	1.6	3.6	1.0	3.6	1.7
Size [LOC]	40000	33000	4800	10000	6050	11000	42000	10040	32000	4000
Effort [work-hour]	8900	6446	1800	887	1850	1520	3775	1219	1886	1850

[5] *Size of development project in MkII Unadjusted Function Points. Note that this size is not necessarily the size of the application subject to the development project. Some of the development projects were either developing a new version of an existing software application (Development projects 3, 7 and 10) or a subsystem of a large software application (Development project 2).*

project, had a lower UDE (0.08) than all the other projects, but was just as good as the Tool A projects on the ADE (0.20). The so-called *industry standard* of Mk II UDE is, according to [6], for systems of more than 1150 FP applying 4GL 0.087 UFP/work-hour. This means that Tool C enabled a development efficiency on average 10 times higher than 'industry average' for 4GL.

Table 3 Overview of results

CASE Tool	Tool A	Tool B	Tool C	Tool D
Mean Unadjusted Development Efficiency (UDE) [UFP / work-hours]	0.13	0.22	0.86	0.32
Mean Adjusted Development Efficiency (ADE) [UFP * Adjustment Factor / work-hours]	0.20	0.26	0.77	0.37
Mean Maintenance Benchmark Task Effort[6] (MBTE) Task 1-7 [work-hours]	71.5	21	5.5	59.5
Mean development Work Expansion (WE) [LOC / UFP]	36.8	12.3	8.7	35.2
Mean development Work Avoidance (WA) [UFP / LOC]	0.03	0.08	0.12	0.03
Mean development Work Speed (WS) [LOC / work-hours]	4.8	2.7	7.3	9.8
Mean Maintenance Benchmark Task Size (MBTS) Task 1-7, [LOC]	1100	82	69	405
Mean maintenance Work Speed (WS) [LOC / work-hours]	15.3	3.9	11.9	9.7
Mean development Time Pressure (TP) [effort work-months / time calendar months]	3.6	2.1	1.4	2.5

[6] *A low MBTE corresponds to a high* ***maintenance efficiency, and*** *vice versa.*

Interesting findings analysing the data in Table 4 were:

- The projects supported by Tool D had a high variance in the development efficiency. This variance may be caused by the high degree of change of functionality of the two projects with the lowest development efficiency (Project 7 and 10). Tool D supported projects had a relatively low maintainability, and a high degree of change makes the development projects more similar to maintenance projects.
- In some cases, the development (UDE, ADE) and maintenance efficiency (MBTE) varied a lot for projects applying the *same* CASE tool. This variance is however much lower than the variance between the projects applying *different* tools.
- The project supported by Tool B had a much better maintenance efficiency than we would expect from the development efficiency. The situation is reversed for Tool D, i.e. it is not always possible to derive the maintenance efficiency of a CASE tool from the development efficiency.
- It seems that a high work avoidance (WA) is a fairly good indication of a high maintenance efficiency (low MBTE). It was on the other hand difficult to conclude on the maintenance efficiency from observations of the development Work Speed (WS). For

Table 4 Development and maintenance efficiency data for each of the projects

CASE tool	Tool A		Tool B	Tool C			Tool D			
Dev. project	1	2	3	4	5	6	7	8	9	10
UDE	0.12	0.14	0.22	0.79	0.88	0.91	0.17	0.41	0.63	0.08
ADE	0.16	0.23	0.26	0.66	0.71	0.94	0.36	0.40	0.54	0.16
MBTE	71	72	21	n.m.	2.0	9.0	n.m	n.m.	35	84
Dev. WE	38.1	35.4	12.3	14.3	3.7	8.0	67.0	20.0	27.1	26.7
Dev. WA	0.03	0.03	0.08	0.07	0.27	0.13	0.02	0.05	0.04	0.04
Dev. WS	4.5	5.1	2.7	11.3	3.3	7.2	11.1	8.9	17.0	2.2
MBTS	n.m.	1100	82	n.m.	22	115	n.m.	n.m.	580	230
Maint. WS	n.m.	15.3	3.9	n.m.	11.0	12.8	n.m.	n.m.	16.6	2.7
TP	2.9	4.3	2.1	0.7	2.0	1.6	3.6	1.0	3.6	1.7

(Dev. = Development, UDE = Unadjusted Development Efficiency, ADE = Adjusted Development Efficiency, MBTE = Maintenance Benchmark Task Effort (Tasks 1-7), WE = Work Expansion, WA = Work Avoidance, WS = Work Speed, MBTS = Maintenance Benchmark Task Size (Tasks 1-7), Maint. = Maintenance, TP = Time Pressure)

example, the high Work Speed of Tool D supported projects may have been caused by the development of a lot of redundant code ('copy and paste'-philosophy stimulated by Tool D), which in turn leads to a low maintainability. The low Work Speed of Tool B supported projects may be caused by a result of a low redundancy degree, which in turn may lead to a high maintenance efficiency.

- There is a lower (linear) correlation between the adjustment factor (AF) and the work avoidance (WA) (Corr = 0.6, p = 0.03), than between AF and the work speed (WS) (Corr = 0.7, p = 0.005). This (together with our knowledge of the projects) indicate that the work avoidance is less impacted by difficulties in the project environment, or complex non-functionality requirements than the work speed.

3 Discussion of the method

In this section we discuss:

- The use of MkII function points (Section 3.1);
- The use of software maintenance benchmark tasks (Section 3.2).

3.1 The development efficiency measures

The MkII Function Point method can, among other things, be criticised for:

1 Oversimplification of the concept of functionality to a formula based on the input, output and entities part of the 'logical transactions'[7)].

2 Subjectively measurement of software functionality. (Different persons may, for example, find different 'logical transactions'.)

3 Adjustment factors based on unvalidated assumptions.

Comments to 1: There were indications that the Function Points, in some cases, did not correlate well with our intuition of functionality nor the effort required to develop the functionality. For example, both Maintenance Benchmark Tasks 1 and 2, see Appendix 1, developed a new user screen containing data from only one object class. The only important difference was that while Task 1 specified a read-only user screen, Task 2 specified a read, insert, update and delete user screen. MkII FP analysis of the specifications gave that the size of Task 1 is 3.8 UFP (1 log. transaction) and Task 2 is 15.9 UFP (4 log. transactions). According to the MkII FP estimation method, the effort to solve Task 2 should be approx. four times greater than for Task 1. The estimates we collected, however, showed that none of the projects believed that the effort was more than twice as great. In fact, all the projects supported by Tool B and Tool C believed that the extra effort concerned with Task 2 was less than 50 %!

The proportion of read-only user screen compared to read, insert, update and delete user screens were rather similar in the development projects we studied. This shortcoming may, therefore, not be serious for our comparison. In addition, even with all the shortcomings the Function Point Method may have, it seems to be one of the better technology independent methods of measuring software size.

Comments to 2: Several studies, such as [13] and [14], indicate a rather small difference in the size measured by different persons within the same organisation. In our study, one of the authors participated in all the Function Point measurements to assure consistency in counting.

Comments to 3: The adjustment factors may be the weakest point of the Function Point method. Many of the underlying assumptions, such as the assumption that changes take twice as much effort as development of new functionality, are

7) *Logical transactions are (simplified) the lowest level business processes supported by the system.*

poorly validated. In addition, it is highly disputable if the project members objectively are able to isolate and accurately estimate the impact of particular environmental factors on the development efficiency. On the other hand, who could develop better estimates of the impact of the project environments?

3.2 The maintenance efficiency measures

Techniques, based on maintenance benchmark tasks, measuring the maintenance efficiency have, as far as we know, not been published before. We had a positive experience with the technique and consider it to have several advantages compared to other ways of assessing maintenance efficiency/maintainability:

- It is a very direct way of assessing the maintenance efficiency, i.e. the interpretation is relatively simple.
- Effort on real-world tasks is measured, not, the much more abstract maintainability.
- Maintainability becomes a characteristic of the software relative to typical maintenance tasks (and to the maintainers carrying them out).

There are pros and cons connected with the use of estimated effort compared to actual effort. Estimated effort can provide the average case for the maintenance task, while actual effort may be influenced by untypical circumstances. On the other hand, actual effort is more objective than estimated effort.

The generality of the results applying the technique depends a lot on how representative the tasks are. In our study, the types of tasks we found meaningful for all the studied applications were concerned with user screens and database changes. This means that our maintenance efficiency results are mainly valid for those types of tasks, not for more domain specific tasks.

4 Conclusions and contributions

We believe that the output of our evaluation method provides useful and important information when selecting and/or evaluating CASE tools.

Our *methodological contributions* to the current state of CASE tool evaluation methods are:

- Validation of the usefulness of Function Point based efficiency measures on real development data; and
- A new approach of assessing the maintenance efficiency, based on software maintenance benchmark tasks.

In the evaluation of four different CASE tools currently in use by Telenor our results suggest that the CASE tools have a strong impact on both the development and (even more) on the maintenance efficiency.

An efficiency ranking of the evaluated CASE tools based on both the development and maintenance efficiency and the degree of work avoidance may, based on our results, look like this:

1 Tool C;
2 Tool B and Tool D;
3 Tool A.

This ranking is directly derived from the efficiency results in Table 3 with one exception: The shared second rank of Tool B and Tool D. The main reason for this is that while the Tool B projects had the highest maintenance efficiency, the Tool D projects had the highest development efficiency.

The differences we have measured in development efficiency and in maintenance efficiency are, in some cases, surprisingly large. For example, the difference in the adjusted development efficiency between the Tool C and the Tool A projects was 1:4 and the difference in maintenance efficiency 1:13. This strongly indicates that the choice of CASE tool really matters!

Several of our results contribute to an increased knowledge of CASE tools. For example, that:

- Some CASE tools (for example Tool D) may lead to a relatively high development efficiency, but to a relatively low maintenance efficiency, i.e. an evaluation of CASE tools should analyse the development *and* the maintenance efficiency. Most evaluation reports and research papers only look at the development efficiency.
- A high work avoidance seems to be a good indicator of high maintenance efficiency.
- The work avoidance seems to be more independent of extraordinary development conditions than the work speed, i.e. a more robust measure of the CASE tool efficiency.

Acknowledgements

The authors wish to thank Dag Sjøberg (University of Oslo), Arne Maus (University of Oslo), Paul Fagerli (Telenor) and the Telenor employees assisting in the collection of the data.

References:

1 Coupe, R T. A critique of the methods for measuring the impact of CASE software. *European journal of information systems,* 3 (1), 28–36, 1994.

2 Norman, R J, Nunamaker, J F Jr. CASE productivity perceptions of software engineering professionals. *Communications of the ACM,* 32 (9), 1102–1108, 1989.

3 IEEE. *IEEE recommended practice for the evaluation and selection of CASE tools.* 1992. (IEEE Std 1209.)

4 Symons, C R. Function point analysis : difficulties and improvements. *IEEE transactions on software engineering,* 14 (1), 2–11, 1988.

5 Boehm, B W. *Software engineering economics.* NJ, Prentice-Hall, 1981. ISBN 0-13-874160-00-7.

6 Drummond, I. *Estimating with MkII function point analysis.* London, CCTA, 1992. ISBN 0 11 330578 8.

7 UFPUG (United Kingdom Function Point User Group). *Mark II function point analysis counting practices manual.* 1994. Version 1.1 (draft).

8 Albrecht, A J, Gaffney, J E Jr. Software function, source lines of code, and development effort prediction : a software science validation. *IEEE transactions on software engineering,* 9 (6), 639–648, 1983.

9 Jones, C. *Applied software measurement, assuring productivity and quality.* New York, McGraw Hill, 1991. ISBN 0 07 032813 7.

10 Zuse, H. *Software complexity, measures and methods.* Berlin, Walter de Gruyter, 1991. ISBN 3 11 12226-X.

11 Jørgensen, M. The quality of questionnaire based software maintenance studies. *ACM SIGSOFT, Software Engineering Notes,* 20 (1), 71–73, 1995.

12 Tsichritzis, D and Klug, A (eds.). The ANSI/X3/SPARC DBMS framework. *Information systems,* 3, 173–191, 1978.

13 Kemerer, C. Reliability of function points measurement : a field experiment. *Communications of the ACM,* 36, 85–97, 1993.

14 Low, G, Jeffery, R D. Function points in the estimation and evaluation of the software process. *IEEE transactions on software engineering,* 16, 64–71, 1990.

15 Nosek, J T, Palvia, P. Software maintenance management : changes in the last decade. *Journal of software maintenance,* 2, 157–174, 1990.

Appendix 1 Outline of the Maintenance Benchmark Tasks

General task requirements:

- The maintenance work should not change the usability, the quality or the maintainability of the application. This means, for example, that an appropriate error handling and error message due to incorrect user input must be provided for each task.
- Changes should follow the common practice with respect to how user screens and reports are designed.
- The effort estimates should include requirement analysis, design, implementation, inspections, documentation and tests (but not installation). Administration/management work should not be included.
- The effort required to insert data in created tables should not be counted.

Data collected:

- Estimated effort (average case).
- Estimated physical Lines Of Code written by the maintainer to solve the task (average case). Lines of Code should be interpreted as the not weighted sum of lines inserted, updated and deleted, comments not included.

The tasks:

Type I: New user screens

Task 1: Development of a new, read-only, user screen with data from an existing database table (or object class) with 5 attribute types.

Task 2: Extension of Task 1. Instead of read-only, the user screen should enable read, insert, update and delete operations.

Task 3: Development of a new read-only screen with data from Table X (5 attribute types, X.1 to X.5) and Table Y. Table X has to be created, Table Y exists and has 3 attribute types (Y.1 to Y.3). There is a foreign key from X.4 to Y.1. The new user screen should present the attributes in the following sequence: X.1, X.2, X.3, Y.2, Y.3, X.5.

Type II: Change of user screens

Task 4: Given a typical read, insert, update and delete user screen, swap the position of two columns.

Task 5: Given a typical read, insert update and delete user screen containing 5 attribute types (columns) from Table X. Extend Table X and the user screen with one attribute type/column. The new attribute type/column should be possible to read, insert, update and delete.

Type III: Change of control logic

Task 6: Given a typical user input field on a screen. Change the correct value interval from [a, d] to [a, b] U [c, d]. The now incorrect instances present in the database (incorrect according to the new interval specification) should be marked.

Type IV: Change of domains

Task 7: Three user screens contain a column with data from attribute type Tx. Change the domain of Tx from numeric to characters. The attribute type is not included in any numerical calculations.

Magne Jørgensen (34) received the Dr.Scient. degree in informatics from the University of Oslo in 1994. From 1989 to 1998 he was a research scientist at Telenor Research, and since 1995 an associate professor at the University of Oslo. His research interests are in software engineering, empirical studies and process improvement. Since 1998 he has been leader of a software development process improvement group at Storebrand.

e-mail: magne.jorgensen@storebrand.no

Sigrid Steinholt Bygdås (34) has been employed by Telenor R&D as Research Scientist since 1992. She has been working with models and methods for systems planning, system development and CASE tool evaluation. Her current interests are Java and middleware technologies for the Internet.

e-mail: sigrid.bygdas@fou.telenor.no

Software process improvement through software metrics – an ESSI project

HANS ERIK STOKKE AND REIDAR PALMSTRØM

The goal of the project was to identify good and effective metrics to measure the software development and maintenance process. We started by making a generic GQM plan and measurement plan for software development at TeleScada, with help from SINTEF and TeleScada's developers. From the baseline project we collected data, and adjusted some metrics so the definitions of those metrics were more accurate. In this article we present the findings from our measurement on the effectiveness of verification and validation.

Introduction

In the early eighties Nera (then EB Nera) decided to start development of a Telecommunication Management Network (TMN) system. This system concentrated on surveillance of Nera's radio link equipment, but marked analysis showed that to stay in business, the system had to support other manufacturers' equipment also. The business unit TeleScada was then created.

Today TeleScada is Nera TMN AS and is one of the leaders in providing non-proprietary Telecommunication Management Network systems.

As this market and product grew, the complexity of the product grew as well. This was handled in the beginning. However, at the end of 1996, the development manager and the manager team at Nera TMN were concerned with quality issues regarding the product, and the effectiveness of the development process. They therefore decided to start an improvement project with the goal to improve this effectiveness.

All presented statistics in this article are collected during or right after the review process. By the review process, we mean the walk-through of the documents or the code by two or more persons (usually there are 3 or 4 persons). During this walk-through we use checklists and a form where we collect the necessary data. The metrics are collected as one page of the review report. This review report is archived after a copy of the metrics form is sent to the 'Telmet mailbox'. Up until today these metrics have been punched into a database by the QA-manager, but this will change so that each project manager will do this job for his or her project. The analysis will still be carried out by the QA-manager together with the management team of the company.

The database for these metrics is developed as an in-house application. To enter one form takes approximately one minute. To produce a preliminary analysis report from this system on one project takes approximately 15 minutes. So the cost of this metrics collection is insignificant.

The ESSI project

In 1996 we applied for a research grant from the EU Commission's ESSI program and got it. During 1996 we worked closely with a team from SINTEF to identify the various metrics that we were going to use, and how best to measure these. The team from SINTEF had good experience with the use of the GQM system, and it turned out to be a good solution for us as well.

The GQM system is not part of this article so for a deeper understanding of this system, see [1]. In short, the GQM system is a system where the goal definition has a set of questions that identify a set of metrics. The metrics are the answers to the questions. We defined four goals, but settled on just two for our experimentation. This article will concentrate on just one, effectiveness of verification and validation.

Experience from first baseline project

During 1996 Nera TMN undertook what was probably one of our biggest development and engineering projects yet. The project was a total Telecommunication Management Network system for a Swedish company, which is one of Sweden's largest telecommunication companies. We decided to make this project our first baseline project (referred to as the baseline A project from now on).

During the fall of 1996 and through 1997, the main effort went into collecting and analysing metrics from the baseline A project. The collection was done mainly by Nera TMN, and we were advised by SINTEF in the analysis, just to make sure that the analysis was not coloured by our opinions and paradigms.

The analysis of the baseline A project showed several anomalies, and this is a short list of our issues:

1 There is no correlation between the preparation time and the number of remarks found in the review. See Figure 1.

2 The size (volume) of the document is not correlated with the time needed to do the review. See Figure 2.

3 There is no significant difference between cost of review per page if the review is planned or not. See Figure 3.

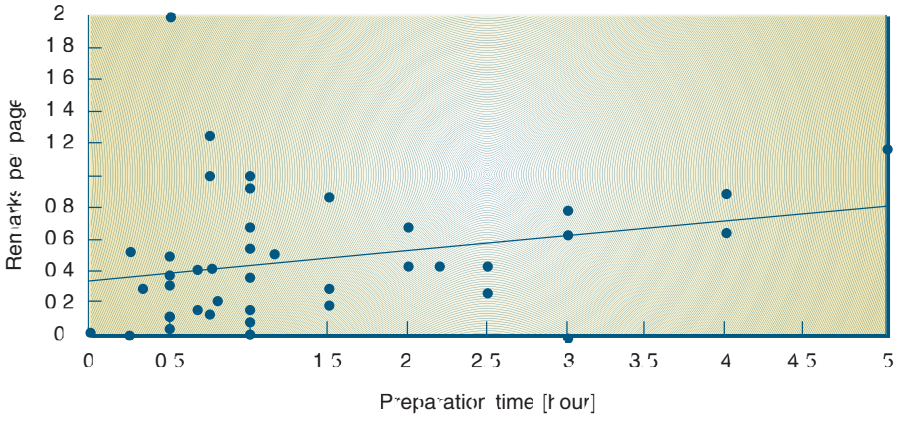


Figure 1 Remarks per page vs. preparation time for the first baseline project (A)

4 The cost of the review does not decrease when the lead time for review increases. See Figure 4.

The following analysis found that the preparation time was very low or non existent. The decision was therefore to increase the preparation time for the next baseline project and see what happened. The preparation time was brought into focus and the participants in this experiment were instructed to use at least four minutes per page. (One code page was defined to be 50 lines of code.)

Experience from the second baseline project

In 1997 we started a similar project to the first baseline project. This time it was a Norwegian telecommunication company, which signed a contract for development and delivery of a Telecommunication Management System. Since this was a similar project to the first baseline project (A), we decided to make this our second baseline project. This project will be referred to as the baseline B project from now on.

The first analysis of the baseline B project showed the following related to the issues of the first baseline project:

1 There is now a correlation (small but significant) between preparation time and the number of remarks found. See Figure 5.

2 There is now a correlation (this one is small also) between the size (volume) of the review and the time needed to perform the review. See Figure 6.

3 There is no significant difference between cost of review per page if the review is planned or not. See Figure 7.

4 The cost of the review decreases with increased lead time for reviews. See Figure 8.

As can be seen, issue 3 is the same for both systems (more on this later), but the situation for the rest of the issues has changed. From this (even though this is a rather small sample) we can conclude:

- The preparation time seems to have some effect on the number of remarks found, the correlation between size of the review and the time needed to perform the review, and the total cost of the review.
- The number of remarks seems to be affected directly.

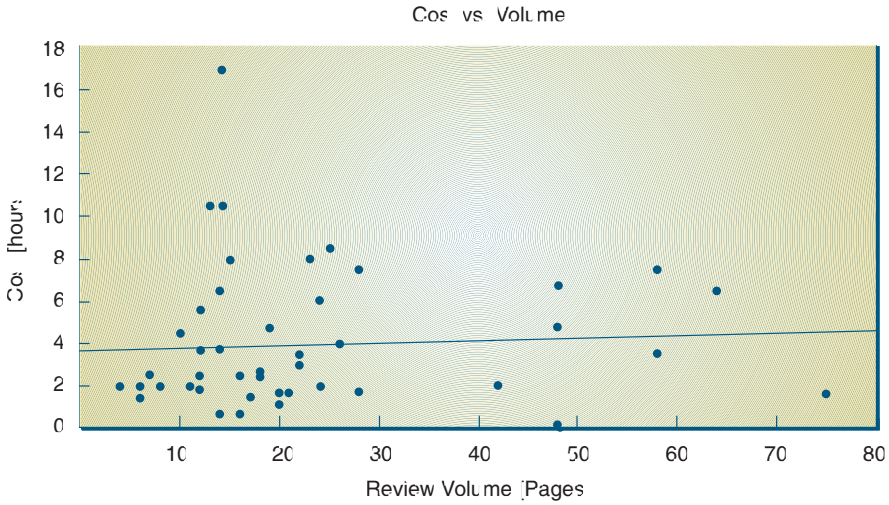


Figure 2 Relationship between cost (time needed) and review volume for the first baseline project (A)

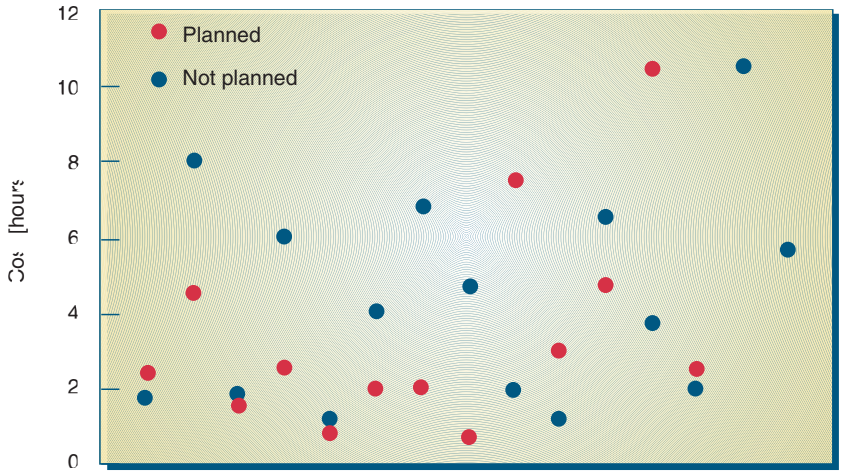


Figure 3 Scatter plot of cost for planned and unplanned reviews for the first baseline project (A)

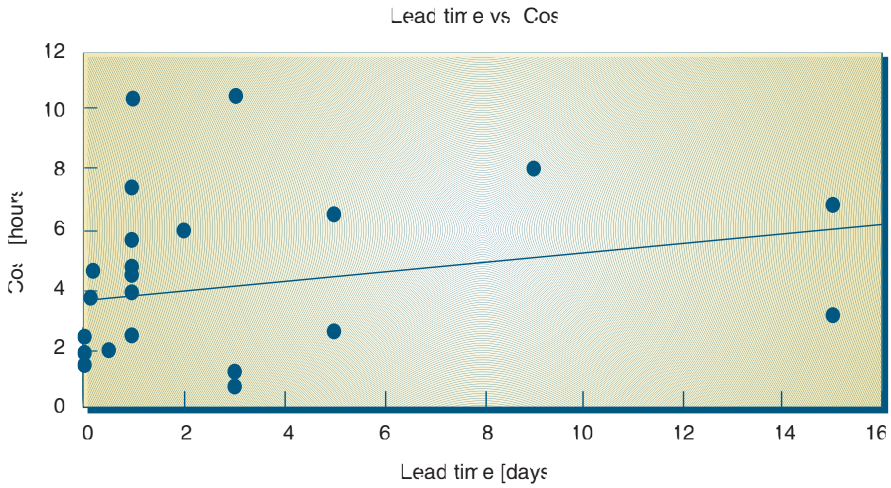


Figure 4 Relationship between cost and lead time for the first baseline project (A)

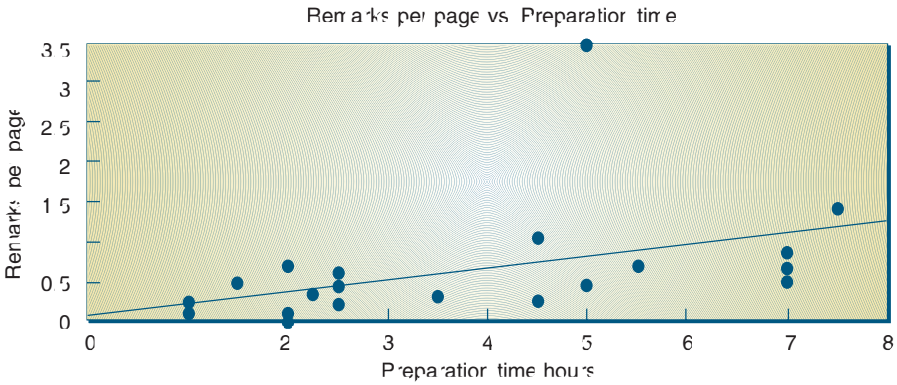


Figure 5 Remarks per page vs. preparation time for the second baseline project (B)

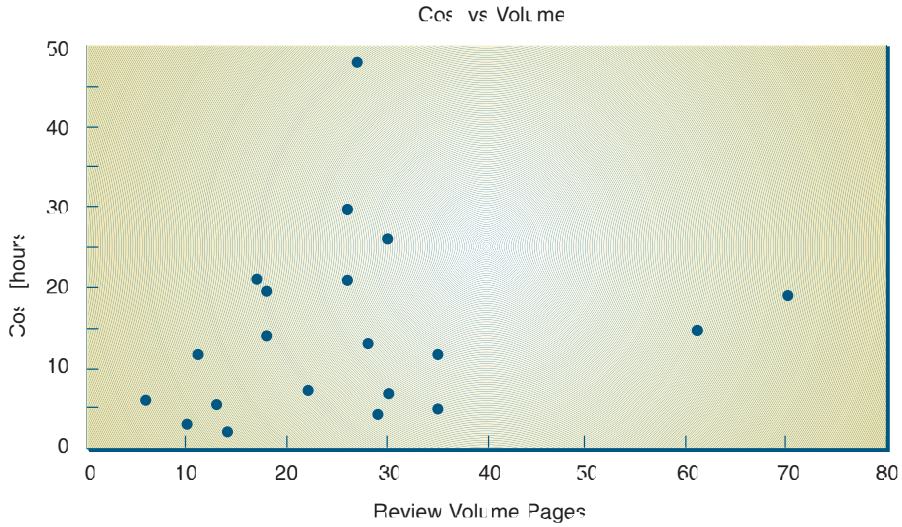


Figure 6 Relationship between cost (time needed) and review volume for the second baseline project (B)

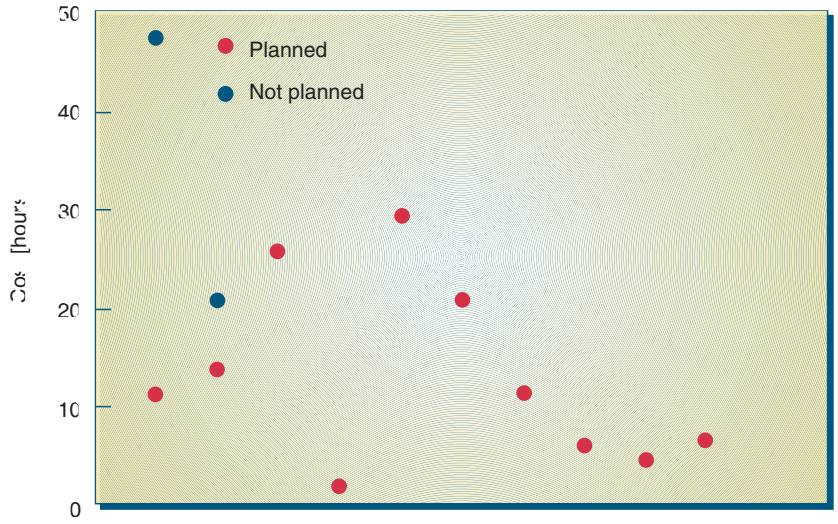


Figure 7 Scatter plot of cost for planned and unplanned reviews for the second baseline project (B)

- The correlation of size vs. time needed and the total cost of the review seems to be indirectly affected.
- Changing the preparation time does not produce a significant difference between reviews that are planned and reviews that are not planned.

To get more clear and precise measurements, these measurements must be taken on several other projects as well, with different preparation time. This is necessary in order to see the importance of the preparation time in our development process.

Why is there no significant difference between cost of review per page if the review is planned or not? There can be several reasons for this:

1 Our hypothesis may be wrong. It says that if a review is properly planned, the cost of the review would be less than for a review that is not planned.

2 The sample may be too small so that the difference does not become significant.

3 Maybe large critical documents are properly planned and the small and less critical ones are not. This will give an abnormal distribution of the samples and our analysis method will not spot this.

4 There may be no difference between planned and unplanned reviews regarding the cost.

There may be even more reasons why there is no difference between planned and unplanned reviews. Our action at this point is to begin experimenting and doing a deeper analysis of this particular problem, to see if we can find the real reason. Currently we have no conclusion for this deviation from our hypothesis – we can only guess. What we will pursue to begin with is item number 3 on the reason list, and in the coming months we will try to design experiments to see if we are correct or not.

Our experiences with this measurement system

The measurement system that was established has served us well regarding effective collection and analysis of the metrics. The GQM plan has also given us a good way to establish good metrics with measurable goals.

Up until today, we have registered some improvements on our development process, and these metrics will be used even more in the future for our improvement programs.

As can be seen from our results, even a minor adjustment gives very good results.

Collection and analysis of data

In the beginning we collected the data using simple forms that were sent to a mailbox. The owner of this mailbox took the form and punched this data into a spreadsheet (Excel). When we had a large enough sample (we decided that we needed at least 10 measurements), we started doing the analysis. This was (and is) an easy way of starting the collection and analysis process. The reason for this is that it is simple and easy. The changing of metrics and methods of analysis can be done relatively fast. But there is a downside to this as well, as we discovered. This becomes apparent when the analysis must be carried out on a regular basis. Then this process is too time consuming.

The solution to the problem was to implement the analysis process in a database. (We ported the collection process as well into the same database, but this is not necessary.) The implementation is time consuming, but it saves a lot of time on the analysis. When we did the analysis in Excel we used one to two days for each analysis. Today we use just 15 minutes. This means that today we can do a weekly analysis on a specific project if we want to.

Conclusion

The time we invested in this measurement system was time well spent. Currently we have some improvements on our development process, and we know that we will be getting even more in the future.

Reference

1 Pulford, K, Kuntzmann-Combelles, A, Shirlaw, S. *A quantitative approach to software management : the ami handbook.* Wokingham, Addison-Wesley, 1996. ISBN 0-201-87746-5.

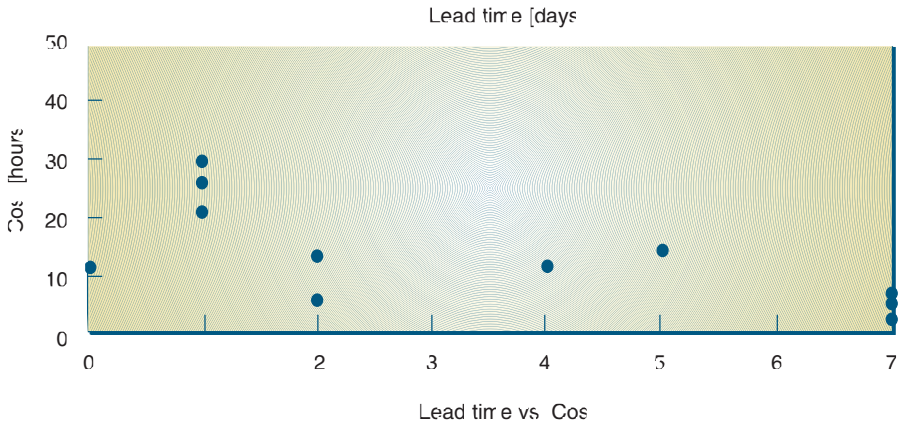


Figure 8 Relationship between cost and lead time for the second baseline project (B)

Hans Erik Stokke (29) is QA-Manager at Nera TMN AS. He received his M.Sc. from the Norwegian Institute of Technology, and has experience from quality assurance (software and hardware), working on quality assurance of the development of large scale telecommunication management systems for network operators and service providers.

e-mail: hes@ts.nera.no

Reidar Palmstrøm (40) is SW Development Manager at Nera TMN AS, and has experience from SW development management of large scale telecommunication management systems for network operators and service providers.

e-mail: rp@ts.nera.no

Surviving software testing under time and budget pressure

HANS SCHAEFER

Testing is the planning and execution of test cases. The goal is to get information about the quality of the system, by running a set of test cases, and by comparing the expected output with the real output. In this paper, 'testing' denotes the test execution phase. This phase is often called 'the testing phase'. The problem is often that everything else, which is to be done before the testing, is delayed. This leads to the test execution being put under severe time and budget pressure. We therefore need a prioritization strategy in order to do the best possible job with limited resources.

Which part of the system requires most attention? There is no unique answer, and decisions about what to test have to be risk-based. There is a relationship between the cost of testing and the cost of defects, and there are other choices to be made. This paper presents some approaches for how to survive the bad game of testing under pressure.

Disclaimer: The ideas in this paper are not intended to be used with safety critical software. Some of the ideas may be useful in that area, but due consideration is necessary.

The scenario is as follows: You are the test manager. You made a plan and a budget for testing. Your plans were, as far as you know, reasonable and well founded. When testing time approaches, the product is not ready, some of your testers are not available, or the budget is just cut. You can argue against these cuts and argue for more time or whatever, but that doesn't always help. You have to do what you can with a smaller budget and time frame. Resigning is no issue. You have to test the product as well as possible, and you have to make it work reasonably well after release. How to survive?

There are several approaches, using different techniques and attacking different aspects of the testing process. All of them aim at finding as many defects as possible, and as serious defects as possible, before product release. Different chapters of this paper show the idea. At the end of the paper, some ideas are given that should help to prevent the pressured scenario mentioned before.

In this paper we are talking about the higher levels of testing: Integration and system test. We assume the basic level of testing every program has been done by the programmers. We also assume the programs and their designs have been reviewed in some way. Still, most of the ideas in this paper are applicable if nothing has been done before you take over as the test leader. It is easier, however, if you know some facts from earlier quality control activities such as design and code reviews and unit testing.

1 The bad game

You will lose the game anyway, by bad testing, or by requiring more time to test. By doing bad testing you will be the scapegoat for lack of quality. By doing reasonable testing you will be the scapegoat for the late release.

How to get out of the game?

You need some creative solution, namely you have to change the game. You need to inform management about the impossible task you have, in such a way that they understand. You need to present alternatives. They need a product going out of the door, but they also need to understand the risk.

One strategy is to find the right quality level. Not all products need to be free of defects. Not every function needs to work. Sometimes, you have options to do a lot about lowering product quality. This means you can cut down testing in less important areas.

Another strategy is priority: Tests should find the *most important defects* first. Most important means often "in the most important functions". These functions can be found by analyzing how every function supports the mission, and checking which functions are critical and which are not. But you can also test more where you expect more defects. Finding the worst areas in the product soon and testing them more will give you more defects. If you find too many serious problems, management will often be motivated to give you more time and resources. Most of this paper will deal with a combination of most important and worst areas priority.

A third strategy is to make testing cheaper in general. One major issue here is automation of test execution. But be cautious: Automation can be expensive, especially if you have never done it before or if you do it wrong! However, experienced companies are able to automate test execution with no overhead compared to manual testing.

A fourth strategy is to get someone else to pay. Typically, this someone else is the customer. You release a lousy product and the customer finds the defects for you. Many companies have applied this. For the customer this game is horrible, as he has no alternative. But it remains to be discussed if this is a good strategy for long term success. So the someone else should be another department in your company, not the testers. You may require the product to fulfil certain entry criteria before you test. Entry criteria can include certain reviews being done, a certain test coverage in unit testing, and a certain level of reliability. The problem is: you need to have high level support to be able to enforce this. Entry criteria tend to be skipped if the project gets under pressure.

The last strategy is prevention, but that only pays off in the next project, when you, as the test manager, are involved right from the project start.

2 Understanding necessary quality levels

Software is embedded in the larger, more complex business world. Quality must be considered in that context (see ref. [8]).

The relentless pursuit of quality can dramatically improve the technical characteristics of a software product. In some applications – medical instruments, air-navigation systems, and many defense-related systems – the need to provide a certain level of quality is beyond debate. But is quality really the only or most important framework for strategic decision making in the commercial marketplace?

Quality thinking fails to address many of the fundamental issues that most affect a company's long-term competitive and financial performance. The real issue is which quality will produce the best financial performance. Quality metrics per se, such as performance measures or defect rates, do not relate to economy. It is in general unknown whether customers will pay higher prices for more quality. In our industry, we must always take into account the rapid evolution of the software's underlying technology and the

relatively short life cycle of our products. The speed of a product, for example, is not quite interesting when hardware speed doubles every two years. Other qualities have grown in their importance.

From a strategic perspective, we should evaluate all investments in quality with respect to their contribution to building a competitive advantage. There are two primary drivers of competitive advantage – lower production costs and product differentiation (the ability to set a premium price for a product because it offers a meaningful advantage over its competitors).

Quality in itself can mean more reuse and less maintenance. But will it be so much better than someone else's software? Will customers pay for this? The main factor in getting a higher price will often be product differentiation. Quality in general does not provide many options for that.

Beware of fanaticism! Language like total quality, and zero defects may have its place in pep talks – but once it is taken seriously and literally, we are in trouble.

Sometimes, quality is a prerequisite. Especially in the safety critical field, a certain level of quality control is necessary to participate in the market at all. However, in such projects you will not often find yourself in the trouble this paper is dealing with.

Many who implement a quality program these days focus on customer satisfaction. Surely, they reason, happier customers must lead directly to higher profits. This is not necessarily the case. There are four possible scenarios, in which the value offered to the customer may be more aligned or less aligned with the economic benefits received by the company.

Scenario 1. Satisfied customers mirror the company's financial gains. In 1990, Microsoft introduced Windows 3.0, which was enormously successful with its satisfied users and enormously profitable for the company. This is a typical scenario for a completely new product which customers perceive as high value for money.

Scenario 2. The value offered to customers is greater than the return on investment made by the company. The product includes new functions or properties which have been expensive to create. However, the price paid does not recover this. This is the typical case of all the "whistles and bells" added to an otherwise useful software product. But the customers only want to pay for what is useful for them. See [13] for an example.

Scenario 3. The product offers more value than customers will pay for. An advanced product is not always as popular as a less advanced product for a lesser price. Even at the same price, the less advanced product may be more popular, because of easier learning and usage.

Scenario 4. Declining customer satisfaction matches a decline in the company's fortunes. Examples include the fall of Norsk Data, and the decline of Digital Equipment.

To sum up: You have to be sure which quality strategy you want to choose, which qualities and functions are important. Less defects do not always mean more profit! You have to research how quality and financial performance interact. Examples of such approaches include the concept of Return on Quality used in corporations such as AT&T [9]. ROQ evaluates prospective quality improvements against their ability to also improve financial performance. Be also aware of approaches like Value Based management. Avoid to fanatically pursue quality for its own sake. Thus, more test is not always needed to ensure product success!

3 Priority in testing: Most important and worst parts of the product

Testing is always a sample. You can never test everything, and you can always find more to test. Thus you will always need to make decisions about what to test and what not to test, what to do more or less. The general goal is to find the worst defects first, and to find as many such defects as possible. This means the defects must be important. A way to assure this is finding the most important functional areas and product properties. Finding as many defects as possible can be improved by testing more in bad areas of the product. This means you need to know where to expect more defects.

When dealing with all the factors we look at, the result will always be a list of functions and properties with an associated importance. In order to make the final analysis as easy as possible, we express all the factors in a scale from 0 to 5. Five points are given for 'most important' or 'worst', or generally for something which we want to test more, 0 points is given to areas where we do not see it important to include them in the testing effort. The details of the computation are given later.

3.1 What is important?

You need to know the most important areas of the product. In this section, a way to prioritize this is described. The ideas presented here are not the only valid ones. In every product, there may be other factors playing a role, but the factors given here have been valuable in several projects.

Important areas can either be functions or functional groups, or properties such as performance, capacity, security, etc. The result of this analysis is a list of functions and properties that need attention. I am concentrating here on sorting *functions* into more or less important areas. The approach, however, is flexible and can accommodate other ideas.

Major factors to look at include:

- *Critical areas (cost and consequences of failure)*

  You have to analyze the use of the software within its overall environment. Analyze the ways the software may fail. Find the possible consequences of such failure modes, or at least the worst ones. Take into account redundancy, backup facilities and possible manual check of software output by users, operators or analysts. Software that is directly coupled to a process it controls is more critical than software whose output is manually reviewed before use. If software controls a process, this process itself should be analyzed. The inertia and stability of the process itself may make certain failures less interesting.

  Output that is immediately needed during working hours is more critical than output which could be sent hours or days later. On the other hand, if large volumes of data to be sent by mail are wrong, just the cost of re-mailing may be horrible.

A possible hierarchy is the following:

A failure would be catastrophic

The problem would cause the computer to stop, maybe even take down things in the environment (stop the whole country, business or product). Such failures may deal with large financial losses or even damage to human life.

A failure would be damaging

The program may not stop, but data may be lost or corrupted, or functionality may be lost until the program or computer is restarted.

A failure would be hindering

The user is forced to work around to do more difficult actions to reach the same results.

A failure would be annoying

The problem does not affect functionality, but rather makes the product less appealing to the user or customer.

- *Visible areas*

The visible areas are areas where many users will experience a failure if something goes wrong. Users do not only include the operators sitting at a terminal, but also final users looking at reports, invoices, or the like, or dependent on the service delivered by the product which includes the software.

A factor to take into account here is the forgiveness of the users, i.e. their tolerance against any problem. It relates to the importance of different qualities, see above.

Software intended for untrained or naïve users, especially software intended for use by the general public, needs careful attention to the user interface. Robustness will also be a major concern. Software which directly interacts with hardware, industrial processes, networks, etc. will be vulnerable to external effects like hardware failure, noisy data, timing problems, etc. This kind of software needs thorough validation, verification and retesting in case of environment changes.

- *Most used areas*

Some functions may be used every day, other functions only a few times. Some functions may be used by many, some by few users. Give priority to the functions used often and heavily. The number of transactions per day may be an idea for helping to find priorities.

A possibility to prioritize down some areas is to cut out functionality which will only be used once per quarter, half-year or year. Such functionality may be tested after release, before its first use.

Sometimes this analysis is not quite obvious. In process control systems, for example, certain functionality may be invisible from the outside. It may be helpful to analyze the design of the complete system.

A possible hierarchy is outlined here (from [3]):

Unavoidable

An area of the product that most users will come in contact with during an average usage session (e.g. start-ups, printing, saving).

Frequent

An area of the product that most users will come in contact with eventually, but maybe not during every session.

Occasional

An area of the product that an average user may never visit, but that deals with functions a more serious or experienced user will need occasionally.

Rare

An area of the product which most users will never visit, which is visited only if users do very uncommon steps of action. Critical failures, however, are still of interest.

An alternative method to use for picking important requirements is described in [1].

3.2 What is (presumably) worst

The worst areas are the ones having most defects. The task is to predict where most defects are located. This is done by analyzing probable defect generators. In this section, some of the most important defect generators and symptoms for defect prone areas are presented. Many more exist, and you have to always include local ideas in addition to the ones mentioned here.

- *Complex areas*

Complexity is maybe the most important defect generator. More than 200 different complexity measures exist, and research into the relation of complexity and defect frequency has been done for more than 20 years. However, no predictive measures have until now been validated for general validity. Still, most complexity measures may indicate problematic areas. Examples include long modules, many variables in use, complex logic, complex control structure, a large data flow, central placement of functions, and even subjective complexity as understood by the designers. This means that there may be done several complexity analyses, based on different aspects of complexity and finding different areas of the product that might have problems.

- *Changed areas*

Change is an important defect generator [5]. The reason is that changes are subjectively understood as easy, and thus not analyzed thoroughly for their impact. The result is side-effects. Advocates for modern system design methods, like the Cleanroom process, state that debugging during unit test is more detrimental than good to quality, because the changes introduce more defects than they repair.

In general, there should exist a protocol of changes done. This is part of the configuration management system (if something like that exists). You may sort the changes by functional area or otherwise and find the areas which have had exceptionally many changes. These may either have a bad design from before, or have a bad design after the original design has been destroyed by the many changes.

Many changes are also a symptom of badly done analysis (see study [5]). Thus, heavily changed areas may not correspond to user expectations. Defects are more likely when code changes.

- *Impact of new technology, solutions, methods*

Programmers using new tools, methods and technology experience a learning curve. In the beginning, they may generate many more faults than later. Tools include CASE tools, which may be new in the company, or new in the market and more or less unstable. Another issue is the programming language, which may be new to the programmers, or Graphical User Interface libraries. Any new tool or technique

may give trouble. A good example is the first project with a graphical user interface. The general functionality may work well, but the user interface subsystem may be full of trouble.

Another factor to consider is the maturity of methods and models. Maturity means the strength of the theoretical basis or the empirical evidence. If software uses established methods, like finite state machines, grammars, relational data models, and the problem to be solved may be expressed suitably by such models, the software can be expected to be quite reliable. On the other hand, if methods or models of a new and unproven kind, or near the state of the art are used, the software may be more unreliable.

Most software cost models include factors accommodating the experience of programmers with the methods, tools and technology. This is as important in test planning as it is in cost estimation.

- *Impact of the number of people involved*

  The idea here is the thousand monkeys syndrome. The more people are involved in a task, the larger is the overhead for communication and the chance that things go wrong. A small group of highly skilled staff is much more productive than a large group with average qualification. In the COCOMO [10] software cost model, this is the largest factor after software size. Much of its impact can be explained from effort going into detecting and fixing defects.

  Areas where relatively many and less qualified people have been employed, may be pointed out for better testing.

  Care should be taken in that analysis: Some companies [11] employ their best people in more complex areas, and less qualified people in easy areas. Then, defect density may not reflect the number of people or their qualification.

- *Impact of turnover*

  If people quit the job, other people have to learn the design constraints before they are able to continue that job. As not everything may be documented, some constraints may be hidden for the new person, and defects result. Overlap between people may also be less than desirable. In general, areas with turnover will experience more defects than areas where the same group of people has done the whole job.

- *Impact of time pressure*

  Time pressure leads to people making short-cuts. People concentrate on getting the problem solved, and they often try to skip quality control activities, thinking optimistically that everything will go fine. Only in mature organizations, this optimism seems to be controlled.

  Time pressure may also lead to overtime work. It is well known, however, that people lose concentration after prolonged periods of work. This may lead to more defects being introduced. Together with shortcuts in applying reviews and inspections, this may lead to extreme levels of defect density.

  Data about time pressure during development can best be found by studying time lists, or by interviewing management or programmers.

- *Areas which needed optimizing*

  The COCOMO cost model mentions shortage of machine time and memory as one of its cost drivers. The problem is that optimization needs extra design efforts, or that it may be done by using less robust design methods. Extra design efforts may take resources away from defect removal activities, and less robust design methods may generate more faults.

- *Areas with many defects before*

  Defect repair leads to new defects, and defect prone areas tend to persist. Experience shows that defect prone areas in a delivered system can be traced back to defect prone areas in reviews and unit and subsystem testing. Evidence in studies [5] and [7] shows that modules that had faults in the past are likely to have faults in the future. If defect statistics from design and code reviews, and unit and subsystem testing exist, then priorities can be chosen for later test phases.

- *Geographical spread*

  If people working together on a project have a certain distance, communication will be worse. This is true even on a local level. Here are some ideas which have proven to be valuable in assessing if geography may have a detrimental effect on a project:

  People having their offices in different floors of the same building will not communicate as much as people on the same floor. People sitting more than 25 meters apart may not communicate enough. A common area in the work space, such as a common printer or coffee machine improves communication. People sitting in different buildings do not communicate as much as people in the same building. People sitting in different labs communicate less than people in the same lab. People from different countries may have difficulties, both culturally and with the language. If people reside in different time zones, communication will be more difficult.

  In principle, a geographical spread is not dangerous. The danger arises if people with a large distance between each other *have to* communicate, for example, if they work with a common part of the system. You have to look for areas where the software structure implies the need for good communication between people, but where these people have geography against them.

- *History of prior use*

  If software has been used before by many users, an active user group can be helpful in testing new versions. Beta testing may be possible. For a completely new system, a user group may need to be defined, and prototyping may be applied. Typically, completely new functional areas are most defect prone because even the requirements are unknown.

- *Local factors*

  Examples include looking at who did the job, looking at who does not communicate well with someone else, who is new in the project, which department has recently been reorganized, which managers are in conflict with each other, the involvement of prestige and many more factors. Only imagination sets boundaries. The message is: You have to look out for possible local factors outside the factors having been discussed here.

What to do if you do not know anything about the project, if all the defect generators cannot be applied?

You have to run a test. A first test should find defect prone areas, the next test will then concentrate on them. The first test

should cover the whole system, but be very shallow. It should only cover typical business scenarios and a few important failure situations, but cover all of the system. You can then find where there was most trouble, and give priority to these areas in the next round of testing.
The next round will then do deep and thorough testing of prioritized areas.

This two phase approach can always be applied, in addition to the planning and prioritizing done before testing.

3.3 How to calculate priority of test areas

The general method is to assign weights, and to make a weighted sum for every area of the system. Test where the sum is highest!

For every factor chosen, assign a relative weight. You can do this in very elaborate ways, but this will take a lot of time. Most often, three weights are good enough. Values may be 1, 3, and 10. (1 for 'factor is not very important', 3 for 'factor has normal influence', 10 for 'factor has very strong influence'.)

For every factor chosen, you assign a number of points to every product requirement (every function, functional area, or quality characteristic). The more important the requirement is, or the more alarming a defect generator seems to be for the area, the more points. A scale from 0 to 3 or 5 is normally good enough.

The number of points for a factor is then multiplied by its weight. This gives a weighted number of points between 0 and 50. These weighted numbers are then summed up. Testing can then be planned by assigning most test to the areas with the highest number of points. An example is shown in Table 1.

The table suggests that invoicing is most important to test, thereafter order registration and performance of order registration. The factor which has been chosen as the most important is visibility.

Computation is easy, as it can be programmed using a spreadsheet. A more detailed case study is published in [4].

4 Making testing cheaper

A viable strategy for cutting budgets and time usage is to do the work in a more productive and efficient way. This normally involves applying technology. In software, not only technology, but also personnel qualification seem to be ways to improve efficiency and cut costs. This also applies in testing.

Table 1

Area to test	Business criticality	Visibility	Complexity	Change frequency	SUM
Weight	3	10	3	3	
Order registration	2	4	5	1	64
Invoicing	4	5	4	2	78
Order statistics	2	1	3	3	34
Management reporting	2	1	2	4	35
Performance of order registration	5	4	0	1	58
Performance of statistics	1	1	0	0	13
Performance of invoicing	4	1	0	0	22

Automation

There are many test automation tools. Tools catalogues list more tools for every new edition, and the existing tools are more and more powerful while not costing more [12]. Automation can probably do most in the area of test running and regression testing. Experience has shown that more test cases can be run for much less money, often less than a third of the resources spent for manual testing. In addition, automated tests often find more defects. This is fine for software quality, but may hit the testers, as the defect repair will delay the project ... Still, such tools are not very popular, because they require an investment into training and learning at start. Sometimes a lot of money is spent in fighting with the tool. For the productivity improvement, nothing general can be said, as the application of such tools is too dependent on platforms, people and organization. Anecdotal evidence prevails, and for some projects automation has had a great effect.

An area where test is nearly impossible without automation is stress, volume and performance testing. Here, the question is either to do it automatically, or not to do it at all.

Test management can also be improved considerably using tools for tracking test cases, functions, defects and their repairs. Such tools are now more and more often coupled to test running automation tools.

In general, automation is interesting for cutting testing budgets. You should, however, make sure you are organized, and you should keep the cost for start-up and tool evaluation outside your project. Tools help only if you have a group of people who already know how to use them effectively and efficiently. To bring in tools at the last moment has a low potential to pay off, and can do more harm than good.

The people factor – Few and good people against many who don't know

The largest obstacle to an adequate testing staff is ignorance on the part of management. Some of them believe that "development requires brilliance, but anybody can do testing".

Testing requires skill and knowledge. Without application knowledge your

testers do not know what to look for. You get shallow test cases which do not find defects. Without knowledge of common errors the testers do not know how to make good test cases[1)]. Again, they do not find defects. Without experience in applying test methods people will use a lot of unnecessary time to work out all the details in a test plan.

If testing has to be cheap, the best thing is to get a few highly experienced specialists to collect the test candidates, and have highly skilled testers to improvise the test instead of working it out on paper. Skilled people will be able to work from a checklist, and pick equivalence classes, boundary values, and destructive combinations by improvisation. Non-skilled people will produce a lot of paper before having an even less destructive test.

The test people must be at least equally smart, equally good designers and have equal understanding of the functionality of the system. One could let the Function Design Team Leader become the System Test Team Leader as soon as functional design is complete. Pre-sales, Documentation, Training, Product Marketing and/or Customer Support personnel should also be included in the test team. This provides early knowledge transfer (a win-win for both development and the other organization) and more resources than exist full-time. Test execution requires lots of people that do not need to be there all the time, but need to have a critical and informed eye on the software. You probably also need full-time testers, but not as many as you would use in the peak testing period. Full-time test team members are good for test design and execution, but also for building or implementing testing tools and infrastructure during less busy times.

If an improvised test has to be re-done, a problem will occur. But modern test automation tools can be run in a capture mode, and the captured test may later be edited for documentation and rerunning purposes.

The message is: Get highly qualified people for your test team!

[1)] *Good test cases, i.e. test cases that have a high probability of finding errors, if there are errors, are also called 'destructive test cases'.*

5 Cutting test work

Another way of cutting costs is to get rid of parts of the task. Get someone else to pay for it or cut it out completely!

Who pays for unit testing? Often, unit testing is done by the programmers and never turns up in any official testing budget. The problem is that unit testing is often not really done. Test coverage tool vendors often report that without their tools, 40 – 50 % of the code is never unit tested. Many defects then survive for the later test phases. This means that later test phases have to test better, and they are delayed by finding all the defects which could have been found earlier.

As a test manager, you should require higher standards for unit testing!

What about test entry criteria?

The idea is the same as in contracts with external customers: If the supplier does not meet the contract, the supplier gets no acceptance and no money. Problems occur when there is only one supplier and when there is no tradition in requiring quality. Both conditions are true in software. But entry criteria can be applied if the test group is strong enough. Criteria include many, from the most trivial to advanced. Here is a small collection of what makes life in testing easier:

- The system delivered to integration or system test is complete;
- It has been run through static analysis and defects are fixed;
- A code review has been done and defects have been corrected;
- Unit testing has been done to the accepted standards (near 100 % statement coverage, for example);
- Any required documentation is delivered and is of a certain quality;
- The units compile and can be installed without trouble;
- The units may even have been run through some functional test cases by the designers;
- Really bad units are sorted out and have undergone special treatment like extra reviews, reprogramming, etc.

You will not be allowed to require all these criteria. You will perhaps not be allowed to enforce them. But you may turn projects into a better state over time by applying entry criteria. If every unit is reviewed, statically analyzed and unit tested, you will have a lot less problems to fight with later.

Less documentation

If a test is designed by the book, it will take a lot of paper to document. Not all of this paper is needed. A test log made by a test automation tool may do the service. Qualified people may be able to make a good test from checklists, and even repeat it. Check out exactly which documentation you will need, and prepare no more.

Cutting installation costs – strategies for defect repair

Every defect delays testing and requires an extra cost. You have to rerun the actual test case, try to reproduce the defect, document as much as you can, probably help the designers debugging, and at the end install a new version and retest it. This extra cost is impossible to control for a test manager, as it is completely dependent on system quality. The cost is normally not budgeted for either. Still, this cost will occur. Here, some advice about how to keep it low.

When to correct a defect, when not?

Every installation of a defect fix means disruption: Installing a new version, initializing it, retesting the fix, and retesting the whole. The tasks can be minimized by installing many fixes at once. This means you have to wait for defect fixes. On the other hand, if defect fixes themselves are wrong, then this strategy leads to more work after finding new defects. The candidate for being wrong is not that easy to find. There will be an optimum, dependent on system size, the chance to introduce new defects, and the cost of installation. For a good description of practical test exit criteria, see [2]. Here are some rules for optimizing the defect repair work:

Rule 1: Repair only important defects!

Rule 2: Change requests and small defects should be assigned to next release!

Rule 3: Correct defects in groups! Normally only after blocking failures are found.

6 Strategies for prevention

The starting scenario for this paper is the awful situation where everything is late and where no professional budgeting has been done. In most organization, no experience data exist and there is no serious attempt at really estimating costs for development, testing, and error cost in maintenance. Without experience data there is no way to argue about the costs of reducing a test.

The imperatives are:

- You need a cost accounting scheme;
- You need to apply cost estimation based on experience and models;
- You need to know how test quality and maintenance trouble interact.

Measure

- Size of project in lines of code, function points, etc.;
- Percentage of work used in management, development, reviews, test preparation, test execution, and rework;
- Amount of rework during first three or six months after release;
- Fault distribution, especially causes of user detected problems;
- Argue for testing resources by weighting possible reductions in rework before and after delivery against added testing cost.

Papers showing how such cost and benefit analyses can be done, using retrospective analysis, have been published in several ESSI projects run by Otto Vinter from Bruel&Kjær [6]. A different way to prevent trouble is incremental delivery. The general idea is to break up the system into many small releases. The first delivery to the customer is the least commercially acceptable system; namely, a system which does exactly what the old one did, only with new technology. From the test of this first version you can learn about costs, error contents, bad areas, etc., and then you have an opportunity to plan better.

7 Summary

Testing in a situation where management cuts both budget and time is a bad game. You have to endure and survive this game and turn it into a success. The general methodology for this situation is not to test everything a little, but to concentrate on high risk areas and the worst areas.

Priority 1: Return the product as fast as possible to the developers, with a list of as serious deficiencies as possible.

Priority 2: Make sure that, whenever you stop testing, you have done the best testing in the time available!

References

1 Karlsson, J, Ryan, K. A cost-value approach for prioritizing requirements. *IEEE Software,* 14 (5), 67–74, 1997.

2 Bach, J. Good enough quality : beyond the buzzword. *IEEE Computer,* 38 (8), 96–98, 1997.

3 Risk-based testing. STLabs Report, 3, (5). (info@stlabs.com)

4 Amland, S. Risk based testing of a large financial application. In: *Proceedings of the 14th International Conference and Exposition on Testing Computer Software,* June 16–19, 1997, Washington, D.C., USA.

5 Khoshgoftaar, T M et al. Using process history to predict software quality. *IEEE Computer,* 31 (4), 66–72, 1998.

6 Several ESSI projects, about improving testing, and improving requirements quality, have been run by Otto Vinter. Contact the author at ovinter@bk.dk.

7 Levendel, Y. Improving quality with a manufacturing process. *IEEE Software,* 8 (2), 13–25, 1991.

8 Favaro, J. When the pursuit of quality destroys value. Testing Techniques Newsletter, May-June 1996. (*http://www.pisa.intecs.it*) in Pisa, Italy. He may be contacted at favaro@pisa.intecs.it.)

9 Quality : how to make it pay. *Business Week,* 3384, 54–59, 1994.

10 Boehm, B W. *Software engineering economics.* Englewood Cliffs, NJ, Prentice Hall, 1981.

11 Jørgensen, M. 1994. *Empirical studies of software maintenance. Thesis for the Dr. Scient. degree.* University of Oslo, 1994. (University of Oslo Research report 188.)

12 Lots of test tool catalogues exist. The easiest accessible key is the Test Tool FAQ list, published regularly on Usenet newsgroup *comp.software.testing.*

13 Matsumoto, C. General Magic's Motorola : AT&T Accounts Go Poof. *San Francisco Business Times,* Sept. 8, 1995.

Hans Schaefer (46) holds a Civ.Eng. degree in computer science from the Technical University of Braunschweig, and is an independent consultant in software testing matters. He has previous experience from Fraunhofer-Institut in Karlsruhe and the Center for Industrial Research in Oslo in work relating to real-time process control software, CASE-tool developing and quality improvement efforts. He is currently test co-ordinator in Telenor's Y2K project, in addition to guest lecturing at several universities in Norway.

hans.schaefer@ieee.org

Quality by construction exemplified by TIMe – The Integrated Methodology

ROLV BRÆK, JOE GORMAN, ØYSTEIN HAUGEN, GEIR MELBY, BIRGER MØLLER-PEDERSEN AND RICHARD SANDERS

This article gives an introduction to some state of the art methods for constructing quality communication software. Emphasis is on formality, abstraction, object-oriented techniques and quality by construction. As an illustration and frame of reference the integrated methodology TIMe is presented.

Introduction

The combination of high complexity with high reliability forced the communication software industry to take a pro-active approach to software quality from the very beginning. As a result, the industry has been a driving force in the research, development and use of software engineering techniques. Communication software has always been highly concurrent, distributed, heterogeneous, and real-time. Therefore, solutions have been developed to attack these problems in particular.

As the software industry in general moves towards distributed heterogeneous solutions we see a convergence towards similar basic principles for the software industry at large. This convergence leads to considerable cross-fertilisation and integration of previously different disciplines such as control systems, user interfaces and databases.

Consequently, we see a trend towards harmonisation and integration of techniques, where traditional boundaries disappear. This development can be illustrated by The Integrated Method, TIMe, which aims to integrate and combine techniques to a whole that cover the totality of disciplines and life cycle steps required in modern systems engineering.

Description techniques

The essence of systems engineering is to understand needs and to design systems having properties that satisfy the needs in a cost effective way. Without descriptions, this is impossible. Descriptions are indispensable in *systems engineering* and all other engineering disciplines. To a large extent systems engineering is a matter of creating, understanding, analysing and transforming descriptions. Consequently, the core (perhaps the soul) of systems and software engineering is description techniques.

This was early realised by the software engineering community and techniques were developed that aimed to increase the abstraction and the formality of descriptions. Abstraction was sought in order to facilitate human conception and understanding, formalisation in order to give preciseness and tool support. The idea was to help prevent errors by improving problem understanding, team communication and by automating manual tasks, and to discover errors by allowing extensive simulation and analysis to take place at early stages of development. To the extent they succeed in these areas, description techniques contribute to software quality both by construction and by correction.

The early techniques developed for software engineering in general, such as Structured Analysis / Structured Design [23], tended not to deal with sequential behaviour, concurrency and distribution, and emphasised abstraction and human understanding more than formality. They had no formal semantics, and therefore it was not possible to simulate and analyse the system behaviour before it was implemented. The mapping from abstract model to concrete design was unclear, and therefore the documentation value of abstract models was limited. Also, the action oriented approach taken by those techniques seemed not to give all the benefits expected.

Later developments have focused more on data modelling, and these have been considerably more successful, especially for data-intensive applications. In recent years the trend has been towards object-orientation and more formality. The Unified Modeling Language, UML [3], is the latest and most notable development in this direction. It combines a set of graphical notations with a partial semantics that makes its meaning more precise. It has a formalism for sequential and concurrent behaviours based on State Charts [8] that enables a partial simulation of behaviour before it is implemented.

The techniques developed for communication systems on the other hand, emphasised formality and dealt explicitly with sequential behaviour and concurrency from the beginning. The FDTs – formal definition techniques – ESTELLE [10], LOTOS [9] and SDL [12, 14] all had state transition based semantics that enabled simulation and analysis to take place before implementation. SDL had the additional benefits of a graphical notation that supported human comprehension combined with an underlying finite state machine semantics that could be implemented effectively. For this reason SDL now seems to be the most successful of the FDTs, with a good track record from numerous practical systems development projects.

SDL as a language was object-based already when first recommended in 1976, and since 1992 [12] it has been a full fledged object-oriented language. It has a semantics that enables complete simulation to take place before implementation, and also to generate complete and efficient implementation code automatically. These properties enable development organisations to move from an implementation oriented development paradigm to a design oriented development paradigm, see below.

In addition to SDL, three other languages are much used for communication systems:

- Message Sequence Charts, MSC [15], which describes interactions by example. MSC provides a useful complement to SDL;
- Abstract Syntax Notation One (ASN.1) which is used to describe data structures, especially in connection with protocols. ASN.1 is combined with SDL [13];
- Tree and Tabular Combined Notation, TTCN [11], which is used to describe test cases. TTCN may be generated from SDL and MSC.

Together these languages complement each other to cover almost everything needed to develop distributed reactive systems. What they lack is the ability to describe general conceptual structures and data models in a graphical notation. This is where UML has its strong point, and fits in as a complement to SDL and MSC.

There is a general trend now to use:

- Object Orientation as a common approach to analysis, design and implementation, with concurrent processes as objects;
- Interaction Scenarios for the specification of communication between users and systems (use cases) and between objects of systems;
- State/Transition based specification of behaviour of individual objects.

This is not very surprising, considering that more and more applications tend to be distributed and reactive. Object orientation helps to master complexity by structuring in terms of objects, and by factoring out common properties in general classes. Objects do not live on their own but communicate with other objects. Interaction Scenarios help to describe and understand even the most complex interaction cases. Describing the behaviour of each object in terms of states and transitions that are triggered by incoming signals from other objects has proven to be of great value and is now adopted in one form or another by most system engineering approaches. As an example of a comprehensive approach, The Integrated Method (TIMe), is presented below.

How FDTs may contribute to quality by construction

In general, most systems engineering methods aims to build quality by making descriptions in the following main areas:

- The problem domain: in order to understand and describe the needs. Understanding the needs is a precondition for achieving quality when quality is understood as 'satisfaction of expectations and needs'. Therefore, domain descriptions that help to get the needs right, and to communicate them precisely, contribute to quality by avoiding errors in the understanding of needs, which is the foundation for quality itself.
- The system properties: in order to specify and understand the characteristics of the system, in particular as seen from an external user. This is where the most important trade-offs and decisions are made concerning how the system will answer the needs. Property descriptions that enable the externally observable properties to be clearly defined, communicated and understood by everybody involved, are essential to ensure that they will satisfy the needs as expected.
- The system design: in order to construct the system in a way that satisfies the properties. Here the main issue is to map external properties to design solutions without losing, changing or adding properties in an undesirable way.
- The system implementation: in order to realise the system according to the design so that it provides the properties and satisfies the needs as expected.

All these descriptions are considered necessary (in one form or another) in most methodologies in order to:

1 Improve common understanding and communication among the people involved in all areas of concerns;

2 Achieve a controlled process towards quality results.

At the same time the number of descriptions may be a problem in itself because:

1 Each description takes some effort to make;

2 Every mapping between descriptions means an opportunity to introduce errors.

Therefore, each description made must be justified by its ability to eliminate errors and simplify other work. The number of translation errors introduced by having several descriptions should be considerably less than the number of errors avoided by making the corresponding descriptions.

Experience has shown that making abstract design descriptions using a language like SDL has helped to considerably reduce the number of errors in implementations, even when the transformation to implementation has been done manually, and therefore the use of a separate application design in SDL has been justified. With automatic code generation, these figures are improved because the translation errors are removed. In that way, using an FDT like SDL for abstract design can contribute towards quality by construction in two important ways:

- By avoiding to introduce errors in the first place;
- By eliminating translation errors.

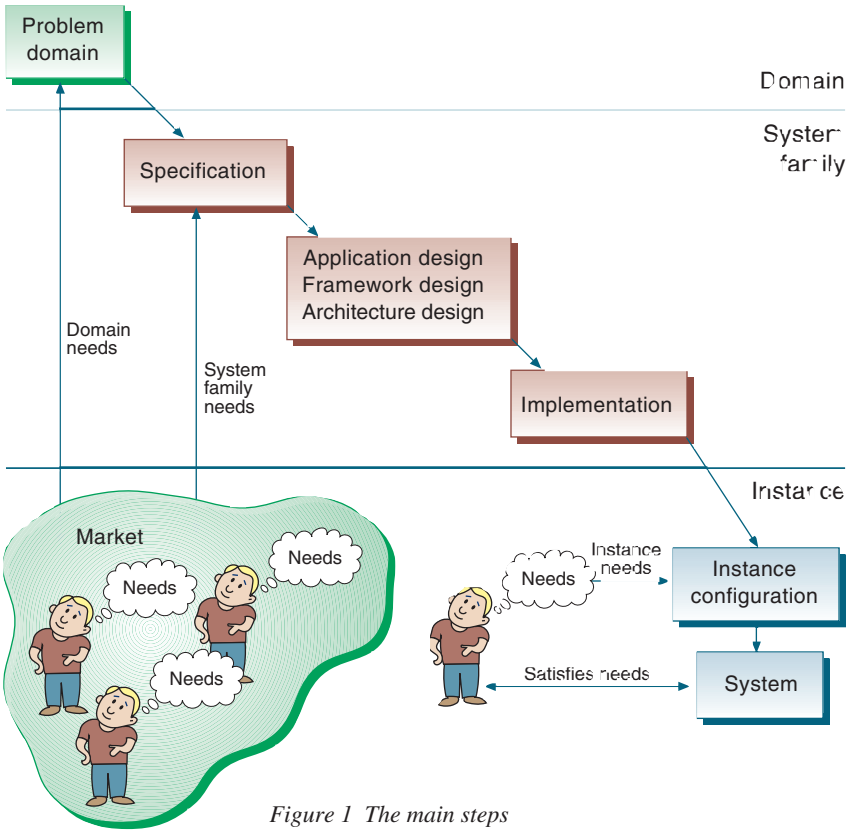


Figure 1 The main steps

As a consequence, many companies have now made the transition from *implementation oriented* development to *design oriented* development. They no longer treat the implementation code, in e.g. C++, as the primary documentation, but as secondary, derived documentation. Application designs expressed in SDL have taken over the role as primary documentation. On this level the application is understood and maintained, and done so in terms that are closer to user understanding. The SDL model can be extensively simulated and validated before implementation takes place, and implementation errors are avoided, particularly by automated transformation. However, there is a snag: the amount of detail required to enable automatic code generation may clutter the description and reduce readability to a level where human errors are likely to increase. Some sort of layering is required to avoid this.

Those that have moved to design oriented development, naturally seek further improvements. In a competitive market place, the ability to add new services (or modify existing ones) with a short time to market while keeping the quality stable, is often sought as the next improvement. In practice, this means to focus more on the domain and the properties. Issues that emerge now are how to model properties separately and how to compose and map properties to designs. The ideal situation would be *property oriented development*, where designs are derived automatically from property descriptions. Although this vision cannot be realised today, some small steps in that direction are already possible using existing languages and tools.

Using MSC to describe behaviour properties, for instance, offers some possibilities. Firstly because MSCs provide a readable and precise way to describe interaction behaviour and thus help to avoid errors, and secondly because MSCs can be used both constructively to partially synthesise application designs, and correctively to verify that application designs satisfy the properties specified in an MSC.

It is interesting to see that UML, which has its origin in the general software engineering community, builds on very similar concepts as SDL and MSC for the behaviour part, namely state charts for object models and event traces for property models. Presently, however, the lack of formal semantics and tool support prevents the possibility of performing design oriented development in UML.

An overall approach – TIMe

The TIMe methodology [19] is designed for systems that are reactive, concurrent, real-time, distributed, heterogeneous and complex. It is centered around a set of models and descriptions capable of expressing domain knowledge, system specifications in terms of external properties, system designs in terms of structure and behaviour, implementation mappings and system instantiation.

Like other similar methods [4], [18], [21], [22], the methodology distinguishes between analysis (of the domain and the requirements), design, implementation and instantiation (see Figure 2).

What is special about the methodology is that:

- Design is split between:
  - *Application design,* where the functionality of the system is designed on an abstract level;
  - *Architecture design,* where the implementation mapping satisfying the non-functional properties is taken care of; and
  - *Framework design,* that defines types of systems with the same infrastructure (e.g. supporting distribution) where the application specific parts are singled out to be redefinable in specific systems.
- It uses a combination of object models and property models both for domain and system analysis, and for design.

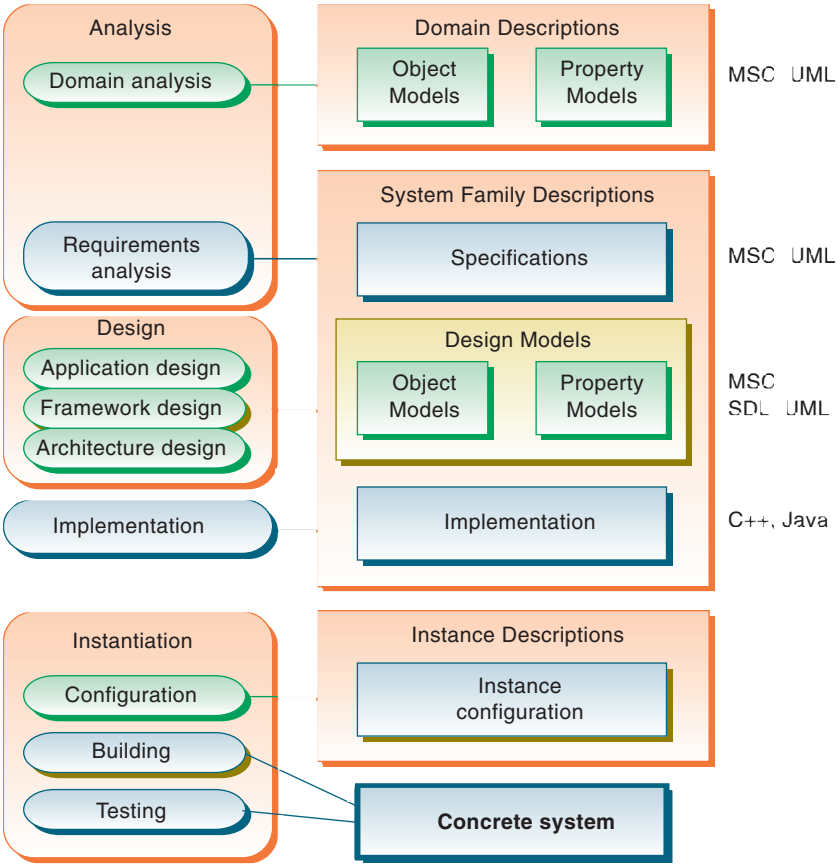


Figure 2 TIMe activities, descriptions and languages

This is done in order to:

- Achieve *flexibility* in services and system designs;
- *Minimise* cost and lead times and to *increase* reuse;
- Enable application evolution to take place in terms of application design;
- Move one step towards property oriented development.

The methodology supports the integrated use of:

- The Unified Modeling Language (UML) for object model analysis;
- Message Sequence Charts (MSC) for interaction scenarios;
- Specification and Description Language (SDL) for specification and design of behaviour.

UML and SDL support object orientation, there are tools that integrate them, and the same tools also support MSC (as well as ASN.1 and TTCN). UML has recently been adopted by OMG – the Object management Group, while SDL and MSC are ITU-T standards.

The methodology promotes:

- Completely analysable application and framework models;
- Automatic transformation from design to code;
- Formal relationships between properties and objects.

This is the setting of the methodology; now let us look at some of its foundation.

Systems and system families

A *system* is defined as a part of the world that a person or group of persons during some time interval and for some purpose choose to regard as a whole, consisting of interrelated components, each component characterised by properties that are selected as being relevant to the purpose. A system is not a description on a piece of paper, but something actually existing as a phenomenon in the real world. This puts the system apart from the description of the system. The system actually exhibits behaviour, while its description is a dead pile of paper.

Systems consist of objects. In order to describe them, classes of objects are defined and described. In short, the methodology consists of approaches, guidelines and techniques for identifying and describing classes of objects, and for deriving, analysing and composing descriptions.

It is fruitful to think in terms of families of systems and really make 'system family specifications' and 'system family designs'. The idea is to focus development and maintenance effort mainly on the system families, in order to reduce the cost and time needed to produce each particular instance, and to reduce the cost and time needed to maintain and evolve the product base.

A system family is a generalised system or set of component types (classes) that can be configured and instantiated to fit into a suitable range of user environments. They represent the product base from which a company can make a business out of producing and selling instances.

The methodology provides guidelines on how to make system families in addition to single systems. Where practical, system types and classes will be defined from which complete system instances may be generated.

Properties and objects

The methodology has the two dimensions *properties* and *objects* as integral parts of the method.

A system consists of a set of objects. Objects are described by:

- *Object models,* that model how a system or a set of related classes are composed from objects, connections and relationships, and how these objects behave.

Systems and objects have properties. Properties are described by:

- *Property models,* that model the properties of a system or object without prescribing their internal construction.

Object models are constructive in the sense that they describe how an object is composed from parts, and is the perspective of designers. Property models are not constructive, but are used to characterize an object or collection of objects from the outside: behaviour properties, performance properties, maintenance properties, etc. This is the perspective preferred by users and sales persons. It is also the main perspective in specifications.

A central idea in the methodology is that every object (and system) is characterized by *provided properties* that can be matched against *required properties*, see Figure 3. The terms verification and validation refer to different aspects of this matching. Verification seeks to establish the truth of correspondence between a description of required properties and the provided properties. In practice, this can be to check that a specified MSC actually may be executed by the system. Validation seeks to check that the provided properties satisfy the real needs of the en-

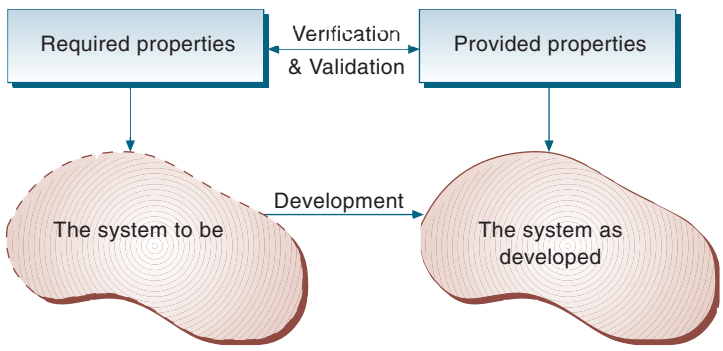


Figure 3 Required and provided properties

vironment, for instance that the needs of a user or another system are satisfied. If the needs can be fully expressed in a description of required properties, then verification is the same as validation, but usually this is not the case. Therefore, validation often involves the user or other systems in the environment. Protocol validation can be seen as a special case where two protocol entities are checked against each other in order to ensure that they behave as expected without errors when working together.

Of special interest are interaction properties, where a property involves the interaction between the system and one or more users of the system or other systems in the environment, or between objects in the system. Figure 4 illustrates some interaction properties specified in MSC. This diagram identifies two instances: the User and the AccessGranting and a simple interaction where the User sends Card and the AccessGranting responds by sending OK. This must be understood as one possible interaction and not as all possible interactions. The property expressed is that the User *may* send Card, and the AccessGranting *may* respond by sending OK. Other properties may be expressed in other MSCs, for instance that the AccessGranting also may respond by sending NOT_OK.

MSCs like the one in Figure 4 need not be tied to particular objects. The User and the AccessGranter may well be understood as roles that some objects play, without being tied to particular objects, and this enables us to define property models separately from object models, and then associate roles with objects. When object models and property models are associated, the general model organisation depicted in Figure 5 is used. The property descriptions describe specific aspects of the system model on different aggregation levels, and is normally organised with a role structure diagram, a concise textual explanation and an MSC document.

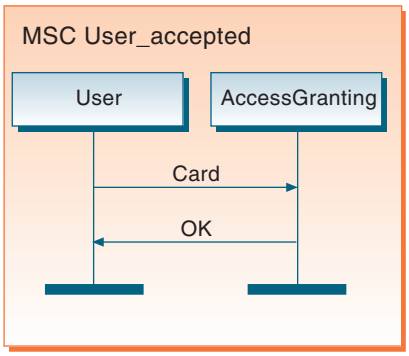


Figure 4 Simple interaction property model expressed using MSC

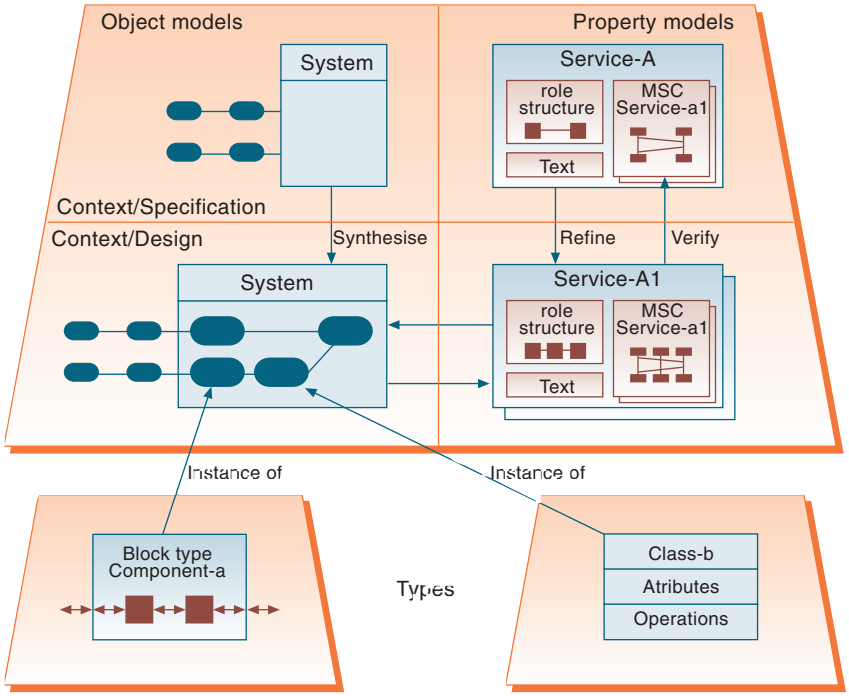


Figure 5 General model organisation

Analysis will produce *specifications* of objects, while design and implementation activities will produce designs of objects. In specifications the object context and external properties are defined. Some limited parts of the content may also be specified, see the wiggly line in Figure 6. The specification of an object includes what is needed in order to use the object – and that may be more than just an interface specification. In the design the remaining content is defined. Using this organisation, specifications are not considered as special models but as an integral part of any model. Before design is made, specifications are validated against user requirements. They are then used as input to synthesising a design. The methodology has guidelines for constructing object designs that will satisfy the properties, and tools exist that can verify that object designs indeed satisfy the specified properties.

When a design is finished the specification can be used to characterise the design as a reusable component and to simplify validation of interfaces when it is connected with other components. As the reader may appreciate, this organization contributes to quality by construction by supporting design synthesis, and quality by correction by supporting validation and verification.

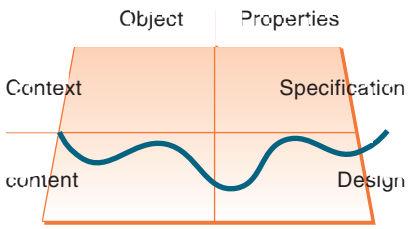


Figure 6 Specification and design

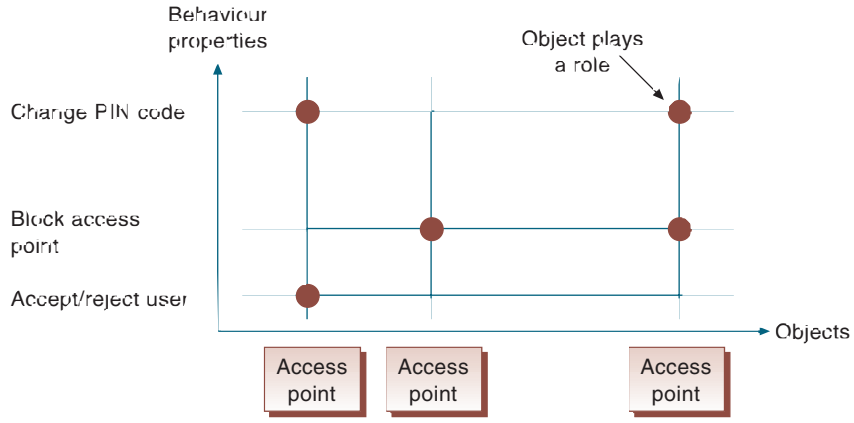


Figure 7 Matching objects and properties in an Access Control System

Quite often a development starts by identifying and describing a number of services that are to be provided, and by defining property models for each of them. Each service will typically involve several object roles, as illustrated in Figure 7. During design, these roles are given to actor objects, where each actor often plays several roles. In this case design synthesis means that the role behaviours assigned to each actor must be composed together into a complete and consistent object behaviour and that the object behaviour should be verified against the assigned role behaviours. In this way the methodology provides some of the answers to the challenge of property oriented system development: to separately model properties, to identify objects and synthesise object behaviours so that they contribute to the properties required of the whole system.

Abstract and concrete descriptions

Descriptions suitable for execution by existing platforms contain a lot of detailed, *concrete* description elements (implementation details, platform specific details, etc.). Descriptions suitable for system developers in their strive to match required properties expressed by users, owners etc. are preferably more *abstract* in the sense that they describe systems in terms that reflect established concepts within a given domain.

The methodology achieves abstraction by supporting UML and MSC for analysis models, and SDL and MSC for design models. UML is a notation that enables informal, abstract object models, MSC describes use cases and interaction between objects, and SDL supports abstract descriptions that (by including concrete description elements) automatically may be transformed to concrete implementations.

Concrete models describe the *implementation architecture*. This is a high level description of the physical implementation. The purpose is to give a unified overview of the implementation and to document the major implementation design decisions. An important benefit of abstract models is that they can allow for many alternative implementations. Therefore they provide a good starting point for trade-offs between alternative technologies and ways to satisfy non-functional properties. The results are documented in the implementation architecture.

Frameworks

Abstract descriptions are organised in two main parts:

- An *application* part that describes what the user environment wants the system to do;
- An *infrastructure* part that describes additional behaviour and supporting functionality that needs consideration, e.g. in order to fully simulate its behaviour. This may e.g. include support for distribution, exception handling, etc.

The reason for this distinction is that systems will often have the same infrastructure part, but different application parts. Reuse of infrastructure is eased by keeping them separate, and application evolution is simplified.

The SDL descriptions will be organised according to the distinction between application and infrastructure. It is normally the case that different systems within a system family will have the same infrastructure but slightly different application parts, and when making different application systems it is desirable not to change or even consider the infrastructure part (besides what it offers). A framework defines the composition of the infrastructure parts and application parts in such a way that different systems can be made by only changing the application parts.

The methodology adapts this idea to SDL, showing how a framework can be defined as an SDL system type, and the different systems as instances of subtypes of this system type. The methodology provides detailed guidelines for how to do this, see [2].

We have above presented some of the foundation of the methodology; we shall now present some aspects that are related to how the languages and FDTs are combined on order to produce quality systems.

From UML models to SDL models

In a mapping from UML to SDL a number of decisions must be taken. The methodology provides guidelines on this mapping – some of them are given below. Most of them are given in a short form just to give an impression of what kind of guidelines that are provided.

UML classes map in general to types in SDL. Classes with their own behaviour and with communication with other objects map to SDL processes types, container classes map to block types,

and data object classes map to SDL data types.

Attributes of objects generally map to SDL variables of data types, unless the object attributes are classes themselves, in which case it is these classes that are mapped as stated above.

Operations are either mapped to remote procedures, to signals in combination with the corresponding transition and optional reply signal, or to operations on variables.

Relations are not easily mapped to SDL. The methodology makes a distinction between *constructive* and *illustrative* relations. Being aware of this distinction when defining relations helps during the mapping. Constructive relations will typically be implemented by corresponding relations in a database part of the system, while illustrative relations need not necessarily be implemented at all.

Connections are mapped to signal routes/channels and corresponding gates on the types involved.

The relations in Figure 8 are not visible in the mapping of the UML AccessPoint class to the SDL AccessPoint block type, while the connection between AccessPoint and User maps to a gate ***e***. The User class is 'mapped' in the first round to processes in the environment of the AccessPoint, and in the second round to processes in the environment of the system.

In a further mapping of the classes in Figure 8, the passive classes will be mapped to classes of objects in a database, describing which users may enter which access zones through which access point. In that mapping the relations are not just illustrative but map to corresponding relations in the database.

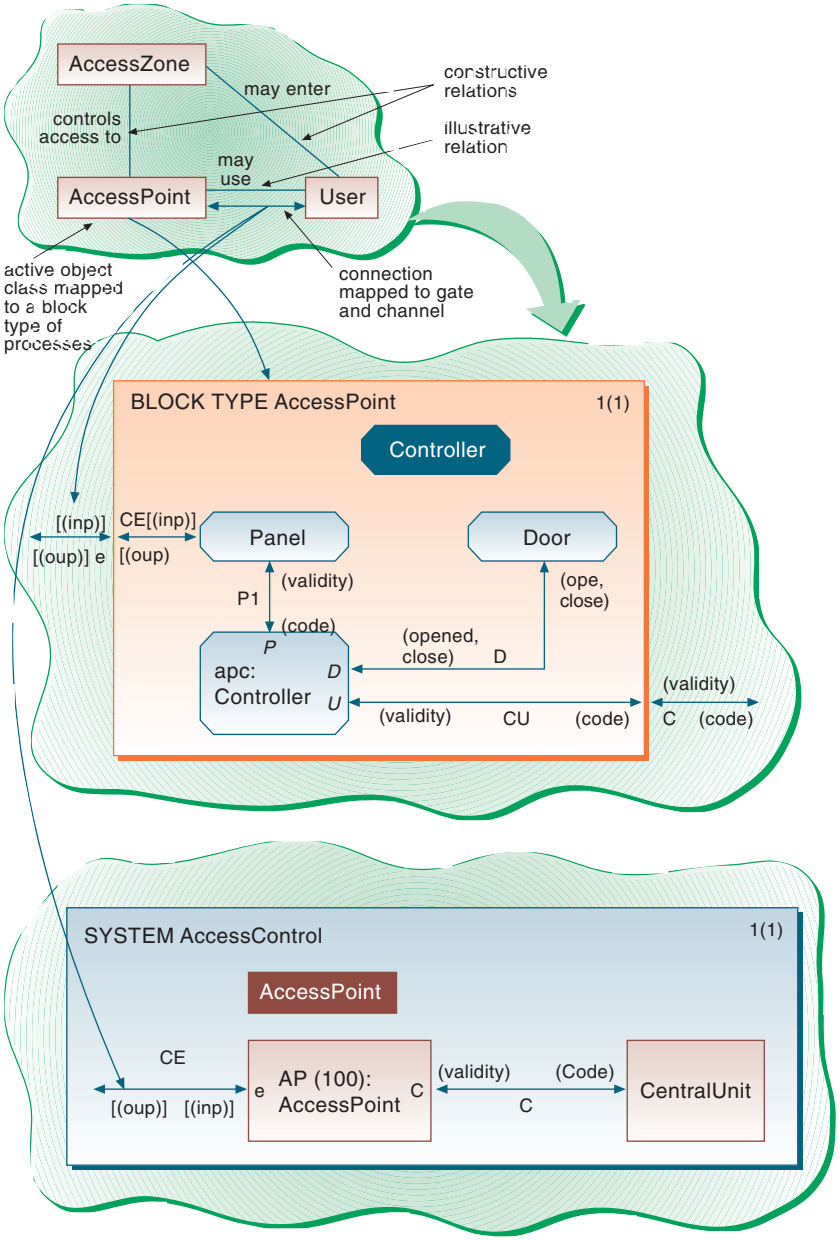


Figure 8 Mapping UML classes, relations and connections to SDL

Aligning SDL and MSC

The property descriptions describe specific aspects of the object model, on different aggregation levels and on different abstraction levels. An important issue is to be able to assess that the different descriptions of the same system actually talk about the same thing.

In some cases this is a simple task, while in other cases it is not so simple. The most difficult situation is when the alignment appears to be simple, but in fact is more intricate. If various aspects of property models are not continuously aligned, a situation like the one in Figure 9 may easily arise.

In Figure 9 it is easy to see that the three different descriptions all describe the same situation in the Access Control system. Still, the three perspectives do not cover each other completely.

The prose says very little about the exact sequencing of the messages, and little about the requirements which the user must fulfil (such as opening the door himself and closing it afterwards) before the system is again ready for another card.

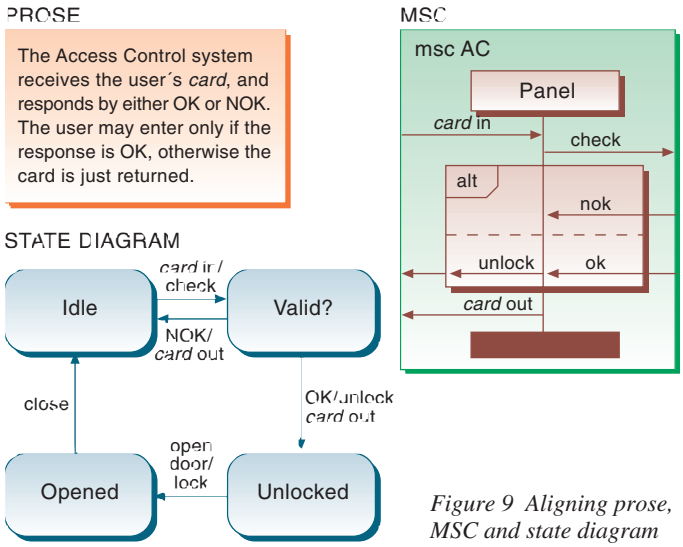


Figure 9 Aligning prose, MSC and state diagram

The MSC says little about the fact that there is a User, and it has no indication of the states which the system passes through. Furthermore, the (visual) response to the user is omitted.

The state-oriented diagram is the most complete one, but it is still not formal enough to be used for automatic code generation (a complete SDL description must be made).

The informal alignment is basically given by identical identifiers. *Card* has been highlighted as one such identifier.

In order to align the three descriptions more formally one runs into questions such as:

- Where are the states (of the state diagram) in the other descriptions?
- Where is the Panel (mentioned in the MSC) in the others?
- Where is the AccessControl system (mentioned in the prose) in the others?

Such differences may not be important, as it is obvious that each perspective will have its own 'aids for thought'. The state diagram uses states as a major means for expression, while MSCs need instances which produce and consume messages. A process algebra description (and other notations based on logic) often uses auxiliary functions.

Where the object model is a complete SDL specification and important properties are expressed by MSC, there is a fairly straight forward alignment mapping. When an MSC document and a supposedly corresponding SDL system is defined, it is necessary to align the two descriptions. By aligning we mean to make explicit how the two descriptions correspond. Which message corresponds to which signal? Which SDL block corresponds to which MSC instance? The advice is to let the names coincide and make this part of the mapping simple.[1)]

Both MSC and SDL may describe non-terminating systems. SDL has initial transitions to define the starting state, while MSC documents do not necessarily have any explicit start at all. Since the MSC document normally does not describe a system completely, the corresponding execution points between the MSCs and the SDL system must be specified.

We recommend that in defining this execution correspondence, the developer should map SDL system states[2)] into MSC conditions. The developer must be aware that MSC conditions do not imply synchronization. Therefore, it may be necessary and advisory to add state invariants as comments in both the MSC and SDL descriptions.

Model checking can face more initial problems. Firstly, there is the partiality problem. The MSC document may not describe all the messages which the SDL system finds adequate to introduce as signals, or the opposite way round.

Secondly, the SDL system and the MSC system may not agree on what objects are in the environment. The MSC document might describe the *User* as an instance while the SDL system defines the user in the environment. Conversely, the SDL system might define the *Door* as a block while it is considered in the environment by the MSC document.

To overcome these discrepancies it is necessary to perform some *alignment modifications*. Some of the alignment modifications will be a permanent change to the specifications while others are only modifications which are necessary for the model checking to perform. For example, the message name *Card Out* could be substituted with *Eject Card*. This could be made permanent. A *Push-Door* message could be eliminated temporarily so that its existence will not confuse the model checking.

Such temporary modifications are often what we call *reductions.* A reduction is a simplification which has no effect on the result of the verification. Put differently, the simplification should be truthful to the original with respect to the purpose of the verification. Reductions may either be mandatory in order to make the model checking work at all, or they may reduce the amount of resources needed to perform the check. One may reduce either the SDL description or the MSC description or both to achieve the most practical correspondence.

[1)] *Integrated MSC/SDL tools often make sure that the mapping of instances and messages are trivial since they demand that the MSC part uses the SDL names.*

[2)] *An SDL system state is the tuple of all process states in the system. In some cases the internal queues should also be included in the system state.*

Reductions may be static or dynamic. Static reductions are changes in the descriptions which are based on the static semantics of the description. Such reductions are e.g. elimination of messages and transitions which communicate with instances which are not in the picture for the verification. See [16]. Dynamic reductions take into account the actual execution of the system. Truthfulness can be achieved more accurately, but the effort needed in the reduction is comparable to performing a reachability analysis. See [20].

There are no adequate tools available to aid in this alignment phase. Therefore manual effort will be necessary to ascertain the consistency of the simplifications. It is especially critical that the statements of truthfulness are made explicit and checked with scrutiny.

Conclusion

FDTs, when used systematically, offer the opportunity to move up in abstraction from implementation-oriented through design-oriented towards property-oriented development, and by doing so harvest quality improvements. The quality improvements are caused partly by constructive means that avoid making errors in the first place, and partly by corrective means that detect errors.

This potential is exemplified by TIMe – The Integrated Method – a development methodology that unites the informal ease of expression of concepts and objects in UML with interaction scenarios in the more formal MSC language, and detailed design of structure and behaviour in formal SDL. From the latter, running implementations may be derived automatically. TIMe provides guidelines on how to use these notations and languages together, both constructively and for verification and validation, to produce high-quality systems. TIMe also exemplifies the convergence of methodologies that are driven by the general convergence of communication and information processing technology.

Bibliography

1 Bræk, R, Haugen, Ø. *Engineering real time systems.* Hemel Hempstead, Prentice Hall International, 1993. ISBN 0-13-034448-6.

2 Bræk, R, Møller-Pedersen, B. *Making frameworks by means of virtual types exemplified by SDL.* FORTE/PSTV, Paris, 3–6 November 1998.

3 UML Partners. *The unified modeling language, version 1.1.* [Online]. URL: http://www.omg.org. (September 1997.)

4 Douglass, B P. *Real-time UML : developing efficient objects for embedded systems.* Reading, Mass., Addison Wesley Longman, 1998. ISBN 0-201-32579-9.

5 Haugen, Ø, Bræk, R, Melby, G. The SISU project. SDL'93 Using Objects. In: *Proceedings of the Sixth SDL Forum,* Darmstadt, Germany, 12–16 Oct. 1993. North-Holland, Elsevier, 1993. ISBN 0-444-81486-8.

6 Haugen, Ø. Using MSC-92 effectively. SDL'95 with MSC in CASE. In: *Proceedings of the Seventh SDL Forum,* Oslo, Norway 26–29 Sep. 1995. North-Holland, Elsevier, 1995.

7 Haugen, Ø. The MSC-96 distillery. SDL'97 Time for Testing : SDL, MSC and Trends. In: *Proceedings of the Eighth SDL Forum,* Evry, France 23–26 Sep. 1997. Elsevier, 1997. ISBN 0-444-82816-8.

8 Harel, D. Statecharts : a visual formalism for complex systems. *Science of Computer Programming,* 8 (3) 231–274, 1987.

9 ISO. *Information processing systems : Open System Interconnection. LOTOS : a formal description technique based on the temporal ordering of observational behaviour.* Geneva, 1989. (ISO 8807.)

10 ISO. *Information processing systems : Open System Interconnection. ESTELLE : a formal description technique based on an extended state transition model.* Geneva, 1997. (ISO 9074.)

11 ISO. *The Tree and Tabular Combined Notation (TTCN).* ISO/IEC JTC 1/SC 21 ("TTCN"). Geneva, 1991. (ISO 9646-3.)

12 ITU. *Z.100 ITU Specification and Description Language (SDL).* ("SDL-92"). Geneva, 1994. (ITU-T Z.100.)

13 ITU. *SDL combined with ASN.1.* Geneva, 1994. (ITU-T Z.105.)

14 ITU. *Addendum to Recommendation Z.100 : CCITT Specification and Description Language.* ("SDL-96"). Geneva, 1996. (ITU Z.100.)

15 ITU. *Message Sequence Charts (MSC).* ("MSC-96"). Geneva, 1996. (ITU-T Z.120.)

16 Juul-Wedde, K. *Consistency control methods and evaluation.* Oslo, ITF, 1994. (SISU Note L-1312-5.)

17 Olsen, A et al. *Systems engineering using SDL-92.* North Holland, Elsevier, 1994. ISBN 0 444 89872 7.

18 Rumbaugh, J et al. *Object-oriented modeling and design.* Englewood Cliffs, NJ, Prentice Hall, 1991. ISBN 0-13-629841-9.

19 SINTEF. *TIMe : The Integrated Method. Electronic Textbook.* [Online].URL: http://www.sintef.no/time. (1998.)

20 Spurkland, S, Haugen, Ø. *Rudimentary SDL Verifier in Prolog.* Oslo, ITF, 1994. (SISU Note L-1313-3.)

21 *Methodology guidelines. The SOMT method.* Malmö, Telelogic, 1998. (Telelogic Manual SDT 3.3.)

22 Verilog. *ObjectGEODE : Method Guidelines.* Verilog Toulouse, France, 1997.

23 Yourdon, E, Constantine, L. *Structured design.* Englewood Cliffs, N.J., Prentice Hall, 1979.

Rolv Bræk (54) is Principal Research Scientist at SINTEF Telecom and Informatics, as well as Adjunct Professor at the Institute for Telematics at the Norwegian Univ. of Science and Technology. He has extensive experience from teaching, consulting and introducing systems engineering methodologies to industry. He has played a central role in the national technology transfer program SISU and its follow-up SISU II, a collaboration between several companies to improve their systems engineering practices.

e-mail: braek@informatics.sintef.no

Joe Gorman (41) studied Computing Science at the Univ. of Glasgow, where he gained his Honours Degree in 1977. After working in Scottish Universities, he started work at SINTEF in 1986. He is involved in contract research work with Norwegian industry, and in international co-operative research funded by the European Commission. His main research interests are software engineering, software development methodologies, compiler techniques and configuration management.

e-mail: joe.gorman@informatics.sintef.no

Øystein Haugen (44) has been working for two years at Ericsson NorARC, where he participates in projects relating to the introduction of TIMe methodology to Ericsson development projects following his earlier activity in the SISU project. He is also part time associate professor at the Inst. for Informatics at the University of Oslo, giving a course in TIMe methodology. Haugen is also heading the group within ITU that standardizes the MSC language, whose next standard is due for approval end 1999. His doctorate thesis dealt with Practitioners' verification of SDL systems.
e-mail: oystein.haugen@ericsson.no

Geir Melby (44) has been working for two years at Ericsson NorARC, where he has the responsibility for the NorARC research program. Before that he was project manager for the SISU project (1988–1997) which has been considered one of the most successful research projects in the software technology area in Norway. He took part in the development of this TIMe methodology based on his earlier experiences with development of real time systems. He has also given several courses in object oriented methods (TIMe) and languages.
e-mail: Geir.Melby@ericsson.no

Birger Møller-Pedersen (49) is Senior Research Scientist at NorARC, Applied Research Center, Ericsson Norway, Software Engineering.

e-mail: Birger.Moller-Pedersen@ericsson.no

Richard Sanders (39) is Research Scientist at SINTEF Telecom and Informatics, and lecturer at the Norwegian University of Science and Technology, from which he graduated in 1984. He has previously worked as a consultant with CAP Gemini and as a software designer and manager at Stento, developing intercom systems and participating in the SISU project. He joined SINTEF in 1995, and helps introduce new development methods in companies, holds courses and carries out research in the field of system development methodologies.
e-mail: richard.sanders@informatics.sintef.no

Software in system perspective

PAUL HOLDER

1 Introduction

Reliability prediction within the telecommunications industry through necessity, is an evolving process. Until the early 1970s the focus had been on the mechanical attributes of uniselectors and relays, a technology that had been evolving over the past 30 years. At this time there were of course thermionic valves whose properties and 'wear out' characteristics were accurately analysed for use in under sea repeaters and other line plant. The 1970s saw a population explosion in the use of transistors, TTL logic and the microprocessor. The size and versatility of the new devices coupled to their adaptability to automated manufacturing methods meant an explosion in their deployment within the telecommunications networks around the world. Through the repeated use of what by today's standards were simple devices, a more complex functionality of the network was achieved. The risk of common failure modes now became a growing factor and it was during the late 1970s and early 1980s that telecommunication companies began to expand reliability departments to include experts in the fields of materials, manufacturing, process control and component testing. In many instances the work carried out was in parallel with that of the supplier and not sustainable, the growth of supplier management and quality standards such as ISO9000 has since allowed such departments to refocus on the core activity of systems and network reliability.

Recently the concept of reliability is gaining a harder edge, it is now becoming the differentiator between competing services and is synonymous with perceived quality. To the reliability engineer this now means that reliability calculations extend to the performance of customer support, fault identification, repair statistics and other aspects of network management. In contrast with the familiar world of circuit reliability, redundant paths and failure mode predictions we must now enter the domain of software reliability and its associated processes.

Figure 1 shows the various factor contributing to system reliability.

It is difficult to directly include software reliability into 'hard' reliability calculations due to its attributes. This paper looks at an emerging service and asks the basic questions:

- Are software faults effecting the service?
- How do their numbers compare to hardware faults?
- Where do they occur?

2 Software reliability

The intrinsic reliability of software is at best a contentious subject, there are of course the jewels in the crown such as the software written for the NASA space shuttle. This level of assurance and testing is not viable in the telecommunications industry where most initiatives are directed at the evaluation of the suppliers development process backed by an extensive testing programme. Figure 2 shows the typical telecommunications view on software.

The types of software testing fall into the general categories of module testing and validation and network simulation. The module testing is usually done by the supplier who tries to ensure it meets its design specification. Some of the modules produced will be based upon those already in existence, others will be new. Network software invariably interacts with other software from various suppliers within its operational environment. To this end it is in the users' interest to evaluate the software on a test network before it is launched on the real system. The suppliers generally support many customers and are not expected, or able, to build network test models to represent every possible application of their software. This applies equally if the software is supplied as part of an equipment or as a standalone package, i.e. network management software. The model testing therefore often falls to the end user.

The type of testing carried out by the end user is dramatically different from that of the supplier, indicators of reliability from the supplier's point of view such as 'estimates of complexity', 'module size' and 'number of patches', are normally not available or applicable. The focus now turns to 'does it perform to requirements?' and 'can we cause it to fail?'.

The supplier is often present at such evaluations to translate the observed problems into terms his software engineers

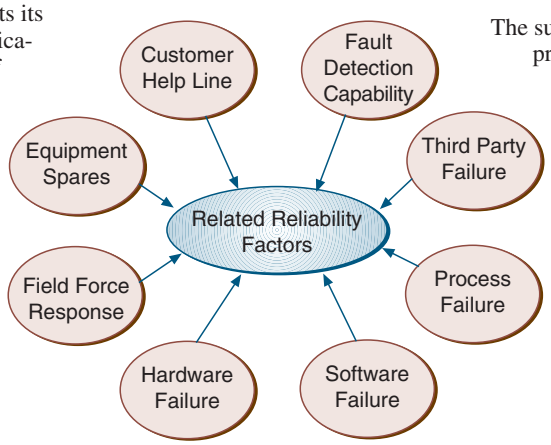


Figure 1 Related reliability factors

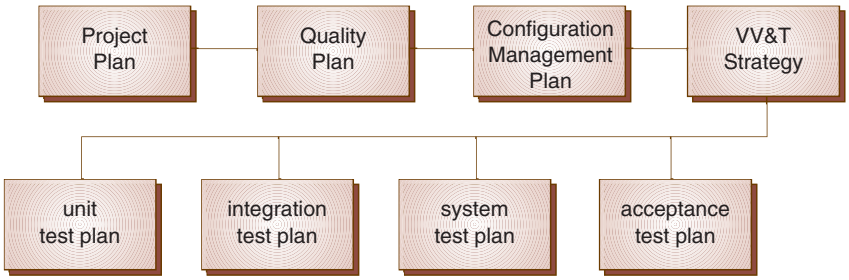


Figure 2 Software development and testing

can understand. When the software is accepted by the customer it is introduced onto the live network. Faults occurring after this stage should be quite rare and due to some unforeseen condition or 'deep seated' error.

3 Software in service

Software does fail in service, even after stringent testing. The software will show a decline in reported faults as it matures and the 'bugs' are fixed. In the telecommunications industry the functionality requirements of the software change rapidly, and major new releases of software can reset the stabilisation process. This is illustrated in Figure 3.

3.1 Operational environment

The roll out of new software into a live network is in itself a major operation requiring planned co-ordination and a rehearsed fall back procedure should it not work. The network itself is operated to conform within specified limits governed by the stated quality of service requirements. The network management system is supported through procedures that provide specific and directed reactions to network problems. The procedures mentioned have a dual purpose, from the point of view of the network manager; they ensure the problem is addressed in a uniform predictable manner whose progress can be easily tracked. The procedures also assign responsibility for clearing the problem to the associated support services who themselves may have their own procedures for use of the resources of others. This is particularly easy to appreciate when one considers the hierarchy of services such as IP over ATM, itself interconnected by an SDH bearer network. In these circumstances the cause of the fault can be lost, particularly if it is transient in nature. In the operational environment such transient failures are often encountered and are worked around, the classic term is FNF, fault not found.

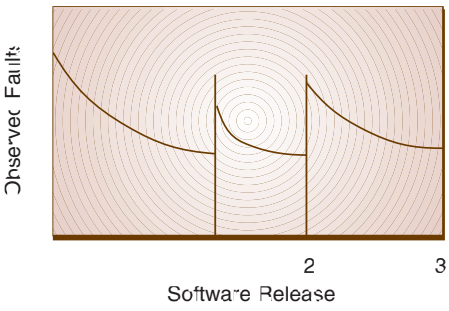


Figure 3 Observed faults with new software release

3.2 The network management process

The objective of the network management system is to maintain the operability of the services and to support accurate billing. The ability of the network management process to detect faults, re-route, without losing traffic, while maintaining billing is a key driver. The complexity and shear dynamics of these networks restrict the ability to diagnose every fault that occurs and this, while regrettable, is a fact of life. Such FNF can manifest themselves in many ways. In the maintenance arena cards returned to the supplier as faulty are often diagnosed as being satisfactory when tested.

In some instances faults can be caused by unplanned work by customers on their own premises or through temporary loss of bearer networks run by other operators. Many network management systems have difficulty in standardising terminology or descriptions for non standard faults, how many times do cards show faults that are cleared through removing then reinserting them? The same thing exists for computers which lock up or crash only to work perfectly when rebooted. The most disturbing fact is that we have become conditioned to accept such failures without taking too much notice, particularly when equipment is widely dispersed and the extent of these instances is not experienced too frequently by the same person.

The following chapters document an attempt to evaluate the possible extent or existence of the problem and associate them to software or hardware sources.

4 Measurement method

In the previous chapter some of the general problems associated with network management were discussed, the purpose of this investigation was to try and identify the magnitude of software problems and present them in proportion to those related to hardware.

The major obstacle was to clearly identify the fault and register it in a uniform way. During the investigation, it was noted that in the particular Network Management Centre being analysed, that in addition to the formal system, a local log of faults was being kept; its purpose was to keep a record of problems and the follow-up actions taken. It became apparent that this local log might be refined to be more precise in its classification of faults, the following actions being taken:

- The log was written in a 'data base' format;
- Fault descriptions were entered from a 'pick list', new fault types could be added by the user;
- Dates and times were added automatically;
- All fields had to be entered;
- An exceptions list was automatically produced at the end of the day for uncleared faults.

The method allowed the results to be gathered in a uniform way and the exceptions list reduced the number of incomplete records.

5 Observations

The data Service chosen for consideration consisted of a core Network of switches interconnected by a bearer network. The 'Network Management' systems for the bearer network operate independently from that of the data service. The service was in its set-up phase and all percentages were calculated from a total number of 151 reported faults over a four month period.

The breakdown of faults within the network is shown in the following sections. The chart shown in Figure 4 indicates the broad areas of faults found across the whole network services.

The above chart addresses all faults detected and suggests that many of the faults encountered are not resolved, at least from the point of view of the Network Operations Unit (NOU). This could be due in part to faults lying in the domains of other NOUs who have passed the fault but do not make a reply to its cause. There is however strong evidence that faults do occur that cannot be traced with any certainty.

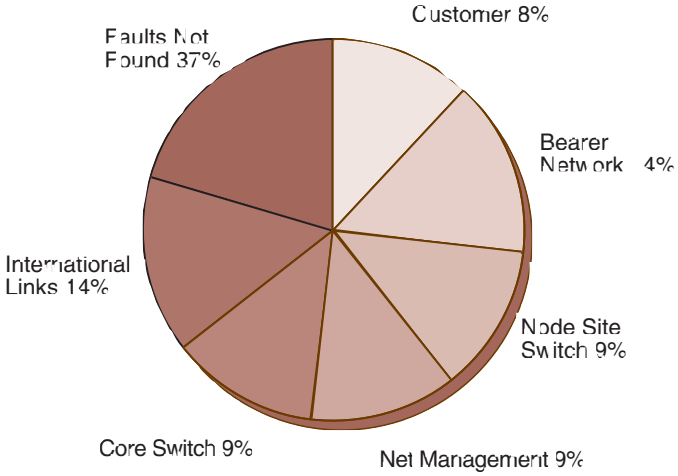


Figure 4 Fault profile – 4 month period

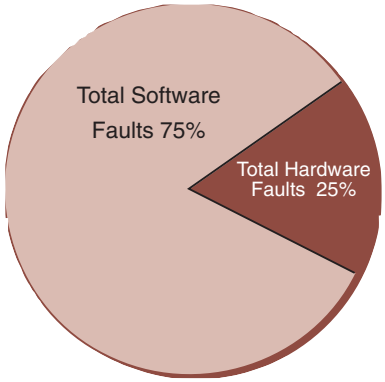


Figure 5 Hardware vs. software

5.1 Hardware and software faults in the network

The chart in Figure 5 indicates that of the faults found, 25 % are attributed to software problems. The FNF are not included in these figures.

5.2 Distribution of software faults

Figure 6 shows the distribution of software faults. The chart strongly suggests that the majority of problems relate to the Network Management System with over 55 % of the reported faults. The core switches contribute to 39 % of the faults with 6 % of the faults arising in the bearer network. It is possible that all of the 'internal' bearer network software faults are not captured here but only the ones that effect our service. If this is the case then 6 % may be considered quite high.

6 Discussion

It is clear that software faults form a large contribution to the overall reliability of this Data Service. It was not proven but one might suspect that many of the FNF failures would also be related to software faults. In order to gain a better understanding of the occurrence and effect of software failures we need to take more care in identifying and logging them. The Network management systems themselves need to be within this logging process as many of the service faults occur within those systems. Testing of systems needs to be thorough and a complete VV&T life cycle process is required.

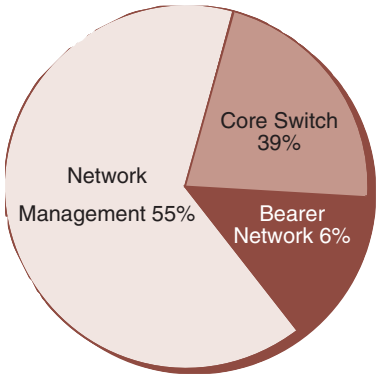


Figure 6 The distribution of software faults

Joining British Telecom in 1974, Paul Holder (41) completed an Apprenticeship in Electronics and Telecommunications before gaining a degree in Engineering. Initially his career was focused on predicting the reliability of systems using Finite Element (FEA) and Finite Difference (CFD) techniques. In later years after further studying he focused on software reliability and currently manages the Verification Validation and Test of network "Plan and Build" software.

e-mail: paul.he.holder@bt.com

Implementing a quality measurement system and the role of EIRUS

JAN WILLEMS

In today's business of a PNO (Public Network Operator) quality is becoming a big issue. It is generally perceived that a PNO can only be successful if the quality of service offered to its customers is on a high level. In other words, quality increases revenues.

There are several inputs in order to obtain the desired quality of service level. Some aspects have to do with the internal organisation and know-how of the PNO. But another very important aspect is the so-called incoming quality, the quality of the telecommunication products that a PNO buys from its suppliers.

In order to be able to assess the quality of these telecommunication products a measurement system should be in place. Such a system should provide a general overview of the quality and reliability of the product and also of the quality of the processes of the supplier.

This paper will start with a presentation of the quality measurement system implemented in Belgacom for a particular telecommunication product, namely public switching elements. The system consists in fact of two different quality measurement systems. Depending on the phase in the product lifecycle a different approach is indeed needed. The first system contains in-process quality metrics that evaluate the processes of the supplier mainly during the development phase of the product. The second system contains measurements that analyse the quality and reliability during the operational phase.

The second part of this paper will focus on the role of EIRUS during the implementation of these quality measurement systems.

1 Quality metrics during the development phase

1.1 In-process quality metrics

A definite goal for a PNO is to get from the supplier a product with a high quality, at low cost and on time. In the case of switching elements, this is not always an easy objective, because the complexity of switching software is often underestimated. Therefore it happens that suppliers have to announce delivery delays during the development phase. But it also happens that the quality level suffers.

In-process quality metrics allow the PNO to follow the development phase in detail and to get a guarantee that no compromises are made with respect to quality. Metrics are also a major input for supplier monitoring activities. Practice shows that possible delays in the product delivery are detected and announced earlier. This means that the internal organisation of the PNO can react in an earlier stage and resources can be rescheduled without excessive costs.

Moreover, these in-process quality metrics help not only to follow the development of a product already bought, but give also an idea of the ability of the supplier to manage development projects. This can be useful for buying decisions on future products of this supplier.

1.2 Implementation issues

The metrics that are used in Belgacom for this purpose, are defined in a Bellcore Generic Requirement document, namely IPQM [1].

The detailed Bellcore definitions of these metrics serve only as a basis. Hard requirements for the supplier to provide IPQM metrics exactly as they are defined in the IPQM document, can be relaxed in many cases. Especially when the supplier performs already IPQM-like metrics internally. In that case they are analysed and taken over as they are or slightly adapted where needed. The underlying idea for the PNO is to get a good picture on how the supplier masters its processes during the development of the product. In other cases, where no equivalent in the existing metrics can be found, the supplier may have to install new tools and is asked to provide the metrics.

The benefit for the supplier is that the metrics are generally accepted as being useful for internal use as well, and that the results can be used to define improvement plans. For Belgacom it is mainly a useful supplier monitoring tool.

1.3 Examples

Two examples of IPQM metrics will be given here: Milestone Monitoring and Test Tracking. Together with other IPQM-metrics, these are provided to

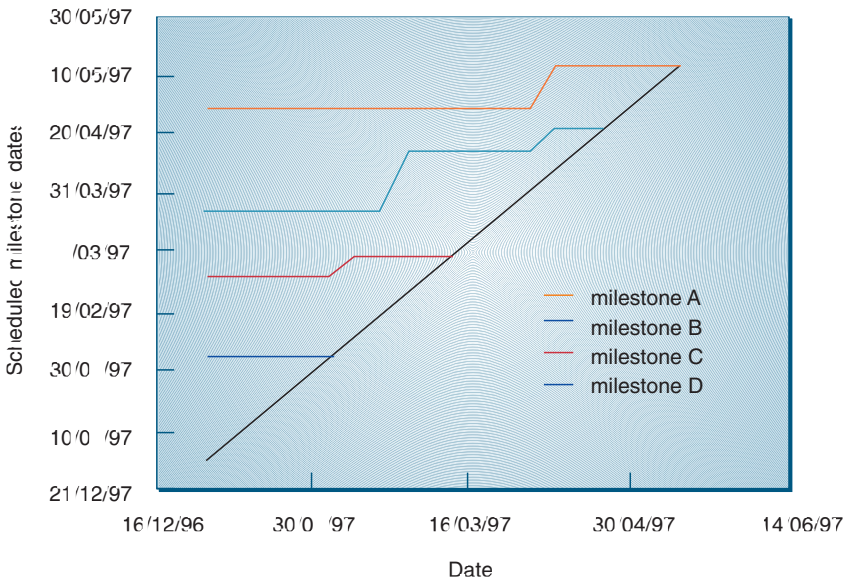


Figure 1 A graphical representation of the Milestone Monitoring metric in IPQM. During the development of a product, this metric clearly shows slips in the internal milestones defined by the supplier

Belgacom by the suppliers on a formal and regular basis, at least monthly.

Milestone Monitoring

This is a simple but important metric. The project milestones are all listed and the planned and the actual completion dates for each milestone are compared. The milestones can also be defined as a set of goals that must be completed before approaching the next sets of goals that are necessary for the completion of the project.

On the EIRUS forum a graph has been proposed which gives an overview of these planned and actual milestone data. An example of such a graph is shown in Figure 1. The planned date of each milestone is shown on this graph. This is done as a function of time, because at some moment in time it may be needed to shift the planned date of a particular milestone towards a later date. In an ideal project the planned dates remain constant and the graph shows horizontal lines.

Test Tracking

The Test Tracking metrics assess the progress of test planning and test activities. The information is used by Belgacom to track the testing activities of the supplier during integration, regression and system testing. The main reason here for the PNO is to gain confidence in the product of the supplier and to detect weak areas. The underlying idea is that with a follow-up of these metrics the validation period (i.e. the period between the official delivery to the PNO and the first office application) can be kept very short and efficient.

Data that are collected and provided by the supplier, are the total number of test cases planned, executed, resp. passed, for each test life cycle phase. This yields an execution percentage of planned tests, a pass percentage related to the executed tests and a pass percentage related to the number of tests planned. The metric is performed for the overall product as well as per group of features in the software release.

2 Quality and reliability measurements in the operational phase

2.1 Tactical Report Card

After the development phase the switching software is officially delivered to the PNO. A short validation period takes place and afterwards the operational phase starts with the first office application. The software introduction in all the exchanges of the network is part of the operational phase. An excellent quality of the switching software should be reached by that time, because the customers of the PNO are now directly affected if the quality level is too low.

In order to measure the quality of the supplier's product during the operational phase, the concept of a so-called 'Tactical Report Card' was introduced in Belgacom. Such a Tactical Report Card is issued every three months and evaluates the quality of the supplier's product in a quantitative way by means of a global score. This score is a value between 1 and 5 and reflects the overall quality of the product. A value 1 corresponds with an unacceptable quality level, while a value 5 corresponds with an excellent quality level.

Since this score is calculated every quarter, it is a unique instrument for the management of both the PNO and the supplier to keep track of the quality evolution and to initiate improvement actions whenever needed.

At the end of every quarter one or two meetings are organised between Belgacom and the supplier. First the correctness of the measurement values are checked and analysed, and afterwards the Tactical Report Card is discussed on a higher management level between Belgacom and the supplier. On these follow-up meetings, weak and strong points are highlighted and action points are defined.

2.2 Measurements

The Tactical Report Card contains up to 30 different measurements, all based on the Bellcore Generic Requirement document RQMS [2]. In Belgacom a subset of the RQMS measurement was chosen. The measurements are grouped into 7 items: System Outage Performance, Patches, Problem Reports, Fault/Fix History, Fix Response Time, Circuit Pack and Release Application.

For each of these measurements the actual outcome is compared with predefined target values. This comparison determines the score for each measurement. The global score is calculated as a weighted average of these individual measurements scores. The weights are fixed values, chosen by Belgacom, and not by the suppliers. They allow the PNO to indicate that some measurements are more important than others. The System Outage Performance measurement, for example, is considered in Belgacom more important than the 'Circuit Pack' measurement.

System Outage Performance

The objective of the PNO is to minimise or even to prevent system outages. The history of system outages with data on duration and frequency helps to understand the performance of the switching element over time.

For the implementation in Belgacom the exact measurement definition was copied from Bellcore, with some small exceptions, e.g. a more strict distinction between outages of host and remote units.

A difficulty in the implementation is the fact that the data collection and reporting for outages in the exchanges is not yet fully automated. Practice shows that outage reporting has to be described in strict procedures for the operational people. Only then can reliable statistics be derived from these data.

An example of the Total Downtime measurement is shown in Figure 2. The downtime, expressed in minutes/system/year, is depicted for the last 12 months. This downtime value is a six-month rolling average of the downtime of the exchanges. Such a calculation smooths the graph and prohibits exceptional outages disturbing the statistics. Every outage report contains also whether the outage in the exchange was attributable to the supplier or not and whether the outage was scheduled or not. This makes it possible to split up the overall downtime into supplier attributable outages and scheduled outages.

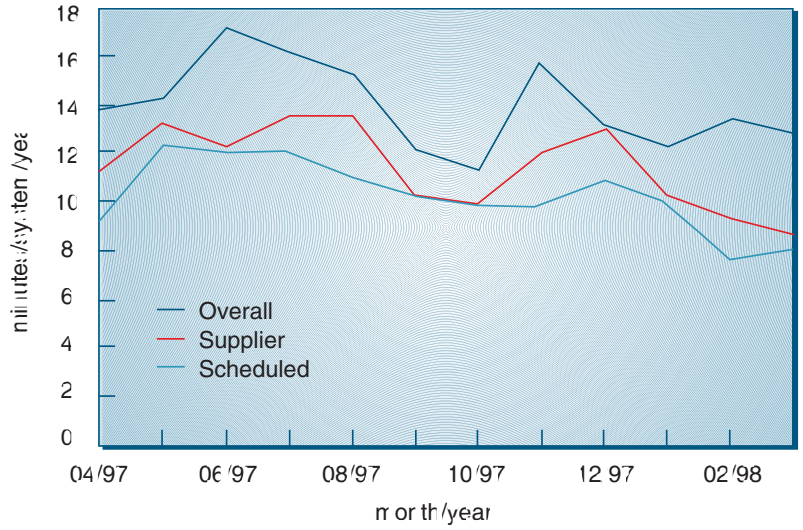


Figure 2 A submeasurement of the System Outage Performance measurement in RQMS. A distinction is made between the overall downtime, the supplier attributable downtime and the scheduled downtime

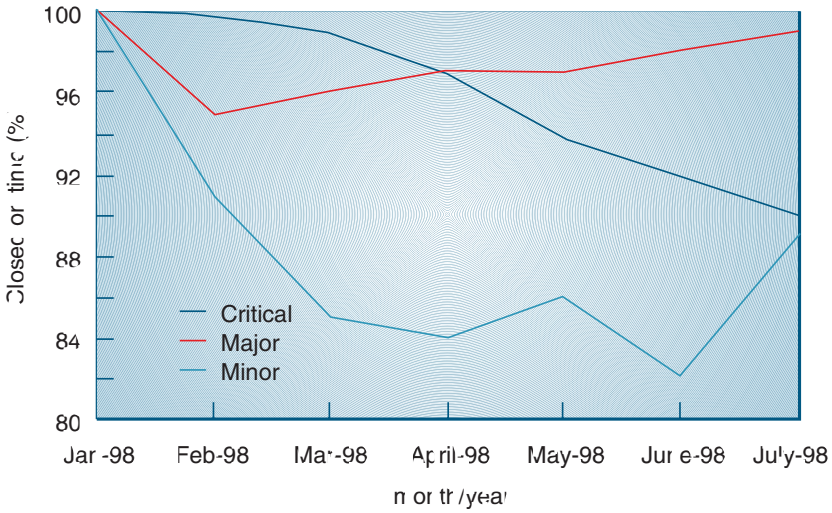


Figure 3 A submeasurement of the Fix Response Time measurement in RQMS

Patches

This measurement helps the PNO to understand on the one hand the extent to which operational people have had to deal with patching activity and on the other hand the stability of the product, i.e. the extent to which the software is being changed in the field.

The switching suppliers of Belgacom are providing detailed information about each patch (introduction date, IDs of problem reports that have been fixed, feature or corrective patch, ...). Two important measurements are the number of released corrective patches as a function of the cumulative field operation months and the percentage of patches found defective.

Problem Reports

The number of incoming problem reports, reported by the people in the PNO during the operational phase, is a measure for the pain experienced in the field. The measurement distinguishes between critical, major and minor problem reports.

The measurement is not provided by the supplier, but is done by Belgacom, mainly because the input data are based on the problem reporting tool of Belgacom, and this tool can produce statistical results in an easy way.

Fault/Fix History

This measurement shows the number of cumulative faults found during the supplier test activities, during the PNO validation test period and during first office applications. The number of fixed and still unfixed faults enables us to assess the effectiveness of the supplier's fault detection and removal process.

The fact that many faults are found does not necessarily indicate either bad or good quality. The more faults are found and resolved before deployment, the better. The importance of this measurement is that it helps to understand whether the release is ready for deployment or not.

Fix Response Time

An example of one of the measurements related to Fix Response Time is shown in Figure 3. The measurement values are calculated as six-month rolling averages of the percentage of fixes delivered on time to the total number of fixes delivered that month. In Figure 3 this is done per severity (critical, major and minor problem reports). On time means fixed within 24 hours for critical problem reports, within 30 days for major problem reports and within 180 days for minor problem reports.

This is one of the most important measurements for Belgacom, because it allows us to examine the supplier responsiveness and promptness with which he responds to problem reports.

A high responsiveness of the supplier is needed, especially when the customers of the PNO are affected by these problems of the switching elements.

The measurement results are regularly compared with the target values. It turns out that this comparison is a strong driver for the supplier to increase the responsiveness for delivering fixes.

Circuit Pack

The sixth group of measurements are the Circuit Pack measurements. These are more focused on the hardware performance of the switching elements and calculate the number of circuit packs that were returned for repair. A high number of returns increases the hardware maintenance cost. Reliable circuit packs are needed and this measurement gives a good indication of this reliability. Thanks to the Circuit Pack measurement appropriate improvement actions can be defined if needed.

Release Application

The measurement data for the Release Application measurement are provided by the supplier and give an idea on the extent to which aborts and problems are encountered during the application of a new release. The percentage of the cumulative number of application attempts for which an abort occurred, and the percentage of the cumulative number of application attempts with one or more problems, are the two submeasurements. Belgacom is working with yearly switching software releases, which means that each year the measurement output can be compared with previous releases. This is clearly a motivation for the supplier to improve the performance on this domain on a yearly basis.

2.3 Future development of Tactical Report Card

Within Belgacom the concept of a Tactical Report Card has been quite successful. Suppliers are now very concerned about the quality of their products. Of course, this was already the case before, but it is felt that this attention increased.

It has been a clear decision in Belgacom to keep this Tactical Report Card as a management tool for assessing the quality of the supplier. On a high management level quality improvement actions can now be discussed more easily.

Although the primary purpose was not to compare the performance of different suppliers, it is very tempting to do this. Comparisons are however delicate and the differences in the global scores of two suppliers are to be interpreted very carefully. The main objective of Belgacom, however, is to see an improvement over time for each individual supplier. In order to force the suppliers to really improve over time, it has been agreed with the suppliers to make the score calculation a little bit more severe every year. This is done by every year changing the target values to which the measurement values are compared.

Because of the positive experience with this system for switching elements, activities have already started in Belgacom to extend the Tactical Report Card concept to other telecommunications products, e.g. SDH equipment and even into the GSM world.

In the future, further measurements will be added to this report card, because for the time being only a subset, although a large one, of the RQMS measurements has been implemented.

The experience showed also that some details need to be finely tuned in order to increase the usefulness, the correctness or the understandability of the measurements. The evolution in EIRUS and in the Bellcore documents will be a guideline here.

3 Role of EIRUS

3.1 A user group for quality and reliability measurements

The EIRUS group (see Figure 4) consists of PNOs and suppliers who apply a uniform quality measurement system for telecommunication products [3]. A study for such a uniform quality measurement system that would satisfy the needs of the European PNOs, was conducted by EURESCOM back in 1995. The main result of this EURESCOM study was that the combination of two measurement sets from Bellcore fulfils the criteria. These Bellcore measurement sets are IPQM and RQMS. In the EURESCOM study these measurement sets were tailored to European use, resulting in the E-IPQM and E-RQMS system. Under the impulse of this EURESCOM study, EIRUS (= E-IPQM and E-RQMS Users) was formed in 1995 as a user group of PNOs and suppliers with the following objectives:

- To implement the E-IPQM and E-RQMS measurements on a wide scale;
- To formulate a consensus on change requests to E-IPQM and E-RQMS to preserve uniformity;
- To find best solutions and practices;
- To learn from experiences of other EIRUS members;
- To provide input to system user groups on the issue of quality measurements;
- To promote the knowledge about E-IPQM and E-RQMS.

EIRUS is now a user group with ten PNO members (Belgacom, BT, Deutsche Telekom, OTE Greece, Swisscom, KPN Telecom, Telecom Finland, Telecom Italia, Telenor and Telia). The supplier members are Alcatel, Ascom Ashler, Bosch Telecom, ECI Telecom, Ericsson, GPT, Italtel, Lucent Technologies, Nortel and Siemens.

3.2 EIRUS Issue List mechanism

EIRUS strongly believes in the uniformity of the measurement definitions, because it can reduce the measurement implementation cost for the PNOs and the suppliers. In order to maintain this uniformity a mechanism was established which manages potential requests for deviation from the Bellcore measurements or potential proposals for changing a particular measurement to a small or a large extent. The need for this mechanism stems from recent experiences during the implementation of the measure-

Figure 4 Logo of EIRUS, a user group of quality and reliability measurements for all kinds of telecommunication products

ments. Indeed, PNOs and suppliers are sometimes faced with implementation problems. In a lot of cases this is due to the fact that the Bellcore measurements, defined in the United States, are not directly applicable to the European situation.

The core of this mechanism is the EIRUS Issue List. This list is an official document to which every PNO or supplier member can add an issue. Issues can be requests for clarification or interpretation of a measurement definition, but also proposals for deviations from the Bellcore measurement definitions. This list is available on the Internet: http://www.eurescom.de/public/newpub.htm.

A fixed item on the agenda of the EIRUS meetings, which are in principle held twice a year, is the EIRUS Issue List. Here each issue is presented and discussed. If necessary, a small working group is created to analyse the issue in more detail. The final decisions on these issues help the PNOs and suppliers to implement the measurements in an efficient way.

This mechanism is rather similar to the process used within Bellcore for creating new versions of the RQMS document. In the United States a BTF (Bellcore Technical Forum) is established where American PNOs and suppliers come together to discuss changes and improvements to the measurement definitions.

3.3 Contacts with Bellcore

The IPQM and RQMS documents from Bellcore are living documents. A result of their BTF process is that on a regular basis new versions of IPQM and RQMS are publicised. The differences between the versions are rather modest, but they are nevertheless important enough for EIRUS to think about how to cope with this evolution.

For the time being EIRUS has decided to adhere to a particular version, at least for some time, even when new versions are issued by Bellcore. An annoying problem is that EIRUS is now not involved in the BTF process. It is believed that EURESCOM could play an important supporting role here. The ideal solution is of course that the evolution in the IPQM and RQMS documents is handled with inputs from both the United States, EIRUS and possibly other interested parties. A standardisation body could play a role here, but today the reality is that the development of the Bellcore documents remains a commercially driven process.

Another topic in the relations with Bellcore are tools. Bellcore has developed an IPQM tool [4], which is now commercially available for different computer platforms. This tool enables the implementation of the IPQM metric system in an easy and standardised way. Bellcore announced that soon there will be a version for RQMS as well. The use of (uniform) tools surely facilitates and speeds up the use of IPQM and RQMS.

3.4 Future developments in EIRUS

A driver for future developments in EIRUS are the recommendations made by EURESCOM Project P619 [5], e.g. about data dissemination and common targets. Data dissemination becomes feasible because of the uniform reporting and means sharing the data that result from the measurements. Several forms of data dissemination have been proposed and the advantages and disadvantages have been listed in this project deliverable. The definition of common targets for each measurement is the process of establishing target values that the products must meet according to the PNOs. Common targets can push forward the quality level.

Another driver for the future comes from the EIRUS members themselves. Their vision will determine for example how the EIRUS Issue List mechanism could evolve. Who knows – maybe one day the development of a separate European quality measurement set without Bellcore will become feasible.

On a short term, however, it is important to extend the application of the E-IPQM and E-RQMS measurements towards more telecommunication product categories. Now the focus is mainly on switching and some transmission products, but the measurements are in principle applicable to a still broader range of products, e.g. TMN, ATM, ...

3.5 Role of EIRUS for the quality measurement system implementation in Belgacom

In the fast changing world of telecommunications it is necessary for a PNO to work better, faster and cheaper. Having these three requirements in mind, it is certain that the role of EIRUS is important for Belgacom with respect to the implementation of the above mentioned quality measurement systems.

EIRUS allows Belgacom to work *better.* Indeed, the E-IPQM and E-RQMS measurement system is, through EIRUS, generally accepted amongst the PNOs and the suppliers. Moreover, this system gives the PNOs an instrument to push quality to a high level, because the system makes the PNO and the supplier understand where the weak and strong points are.

EIRUS also allows us to go *faster.* Because experiences are shared, the learning phase is shorter. Moreover, such a discussion forum leads to better contacts on the subject of quality measurements among PNOs and between PNOs and suppliers. These contacts are paving the way for a rapid implementation.

And finally, it is possible to work *cheaper* because EIRUS assures the uniformity of the measurement definitions and this reduces the cost for the suppliers of making or adapting measurement tools. It is cheaper for a supplier to deliver the same measurements to all his customers, than to deliver completely different measurement data to each of his customers. And a lower cost for suppliers will be reflected in a lower cost of the telecommunication product. But also the fact that internally in a PNO organisation the measurements of different suppliers can be compared and the same tools can be used, is a cost reducing factor.

4 Conclusions

This paper presented the implementation of a quality measurement system in Belgacom for switching elements.

The development phase at the supplier's is covered by a set of IPQM metrics. It allows the PNO to monitor the supplier and it increases the visibility on the product development. A regular reporting of these metrics helps to learn whether the

product will be delivered to the PNO on time and/or according the required quality criteria.

For the operational phase, i.e. when the switching software is operating in the field exchanges, a subset of the RQMS system was chosen. Up to 30 measurements are part of a so-called Tactical Report Card, in which the quality is assessed in a quantitative way by means of a three-monthly global score between 1 (unacceptable quality) and 5 (excellent quality). this turned out to be an important management tool for pushing the overall quality towards higher levels.

The work in EIRUS, being a user group of PNOs and suppliers that are implementing these IPQM and RQMS measurement sets, is of high importance. Such an international user group made it easier for Belgacom to implement generally accepted measurements on a short term and at a lower cost. Evolution and improvements in the measurement system are followed up by EIRUS in order to keep the uniformity in the measurement definitions among the PNOs and the suppliers. Uniformity is indeed substantial because it can reduce the implementation costs. The issue list mechanism in EIRUS is a valuable procedure to maintain this uniformity and at the same time to share experiences.

The positive experience of Belgacom with EIRUS and with the implementation of E-IPQM and E-RQMS hopefully encourages others to start or continue similar activities, because a quality measurement system that is generally applied in Europe and beyond, will force the suppliers even more to deliver high quality telecommunication products. In the end, this is in the interest of the PNO customers.

5 References

1 *Reliability and Quality Measurements for Telecommunications Systems (RQMS).* Bellcore GR-929-CORE, Issue 2, December 1996.

2 *In-Process Quality Metrics (IPQM).* Bellcore GR-1315-CORE, Issue 1, September 1995.

3 Johansson, R. EIRUS : a user group for quality measurements. *Telektronikk,* 93 (1), 86–88, 1997.

4 Ali, S R. Tool based in-process software quality analysis. *Telektronikk,* 93 (1), 83–85, 1997.

5 *European Quality Measurements : Overview of E-IPQM and E-RQMS, Version 2, Deliverable 4.* Heidelberg, EURESCOM, January 1998. (EURESCOM Project P619.)

Jan Willems (32) received his electrical engineering degree from the Univ. of Gent in 1990. He worked 1990 – 1994 as Research Engineer at the Univ. of Gent, dealing with opto-electronic device research for telecom systems. He received his Ph.D. in 1995, joined Belgacom in 1994 in charge of QA and supplier monitoring activities for public switching software until becoming Quality Manager in the Network Services div. in 1998 with the responsibility for improvements in network quality, performance and quality of service.

e-mail: Jan.Willems@is.belgacom.be

Recommendations to improve the technical interface between PNO and suppliers

JAN-ERIK KOSBERG, ØYSTEIN SKOGSTAD AND OLA ESPVIK

1 Introduction

The liberalisation of the European telecommunication market since 1 January 1998 has put new demands on the Quality of service (QoS) provided by Service and Network Providers – hereafter named PNOs (Public Network Operator – PNO). An important basic of controlled QoS is a well structured and – between PNOs and suppliers – agreed upon Quality Assurance framework. The importance of QoS provided to the customer has in later years received a remarkably increased attention. In addition, 'Time to Market' and 'Cost Savings' now represent real competitive issues for the PNOs.

However, a Quality Assurance framework specific to each PNO – Supplier contract on the market would make up a substantial variation in the understanding of quality concepts and make life very ineffective for both suppliers and PNOs. The result could be a lot of ambiguity in the communication on technical matters between PNOs and their many suppliers. It is therefore imperative to all actors in the telecommunication market that PNOs and Suppliers communicate in a well defined and harmonised manner as regards quality issues. Fortunately such a process is well under way in the EIRUS (European IPQM and RQMS Users) organisation by the combined efforts of many leading Suppliers and PNOs.

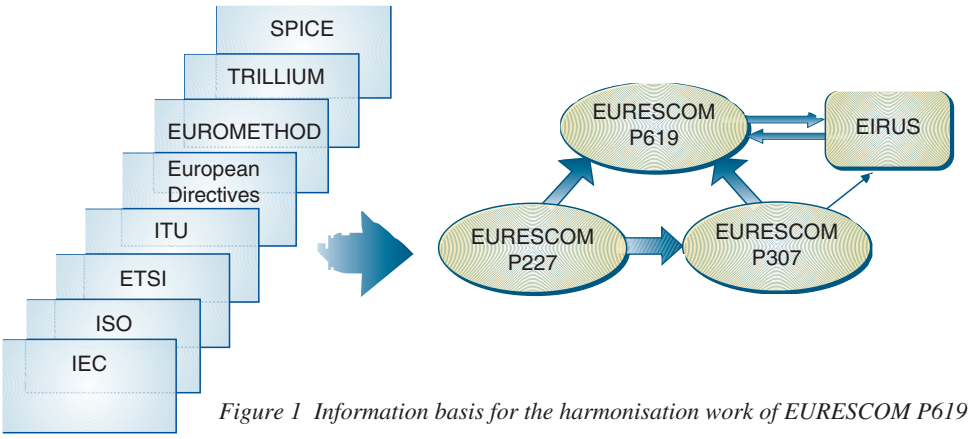
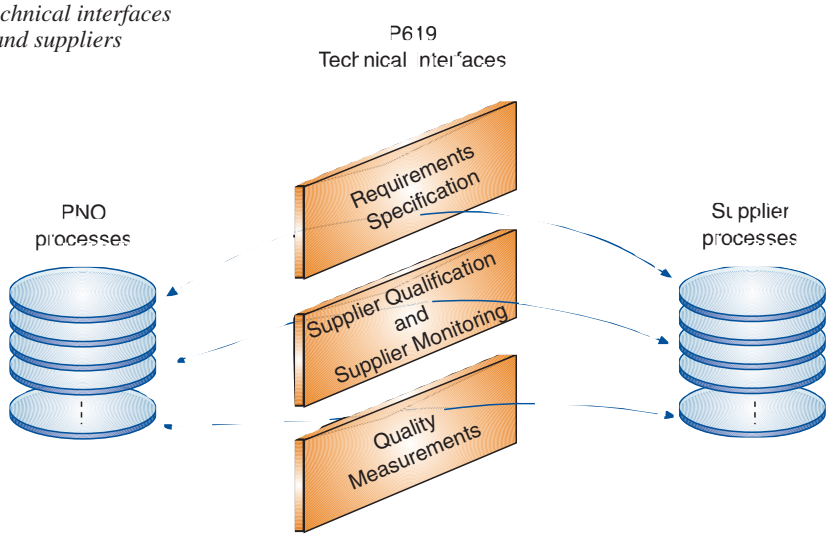


Figure 1 Information basis for the harmonisation work of EURESCOM P619

Figure 2 Key technical interfaces between PNOs and suppliers



Defining PNO – Suppliers Technical Interface as

> *"The communication taking place about technical matters between a PNO and a supplier"*

implies that controlling the incoming quality through an optimisation of the Technical Interfaces is a key issue to a PNO's market success.

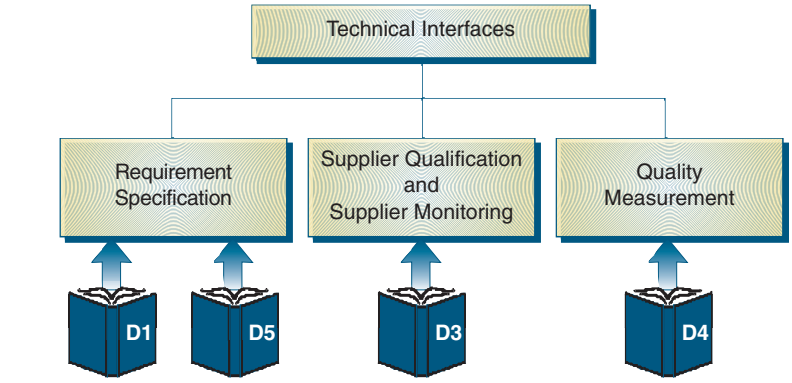
This paper is based on the work and the results from EURESCOM project P619 'PNO – Suppliers Technical Interfaces', that took place during the period February 1996 – March 1998. The project produced six reports (EURESCOM Deliverables) – all publicly available from EURESCOM [1, 2, 3, 4, 5, 6].

PNOs participating in P619 were BT (United Kingdom), CSELT (Italy), Deutsche Telekom (Germany), France Telecom (France), HT (Hungary), KPN Research (The Netherlands), OTE (Greece), RB (Belgium), Telia (Sweden), and Telenor Research and Development (Norway). SINTEF Telecom and Informatics participated in the Telenor team. Project leader was Dr. Marcello Melgara from CSELT and the EURESCOM project supervisor was Juan Siles. The P619 project team kept close contact with EIRUS and reported regularly to the EIRUS meetings.

The purpose of the EURESCOM project 619 was to analyse the technical interfaces and to identify on which levels these interfaces could be harmonised and improved. The commercial interfaces of the procurement environment were not subject to the project analysis. Technical interfaces had been partially analysed by an earlier EURESCOM project P227 'Software Quality Assurance' that produced a deliverable 'Buy IT' [7], upon which some of the work of P619 was based – in addition to a wide range of other international studies, reports, and standards on Quality Assurance. The background information of P619 is shown in Figure 1.

The technical interfaces cover product and process related quality assurance activities agreed upon between the PNO and the supplier. P619 focused on the following three main technical interfaces:

- Requirements Specification;
- Supplier Qualification and Supplier Monitoring;
- Quality Measurement.



D1: 'Guidelines on Requirements Specifications'
D3: 'Guidelines on Suppliers Qualification and Monitoring'
D4: 'European Quality Mesurements: Overview of E-IPQM and E-RQMS'
D5: 'Guidelines on Hardware Traceability'

Figure 3 Technical interfaces and P619 Deliverables

These interfaces are illustrated in Figure 2. The interfaces cover activities both within the PNO and at its supplier.

The selected technical interfaces and the relations to the results of P619 in the form of Deliverables are shown in Figure 3.

2 Requirements specification interface

The Requirement Specification document not only describes technical characteristics of products but also takes into account the product's entire life cycle, placed into the standard life-cycle process model described in ISO 12207 [18] for software.

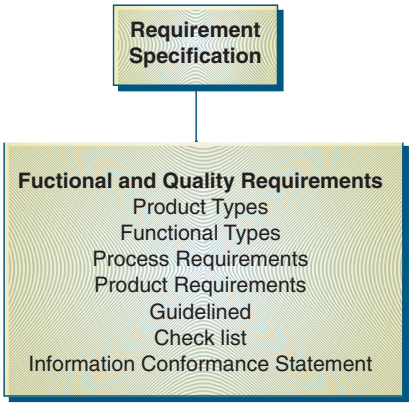


Figure 4 Requirements specification interface aspects

The process of specifying requirements starts after having identified the needs made by the PNO, involves interaction with suppliers, and ends with the inclusion of the Requirements Specification document in a contract. Specification writing steps are not purely technical. They are in fact a combination of all kinds of activities, including organisational and commercial ones. European Council Directive 93/38 [9], harmonising the procurement procedures of entities operating in the water, energy, transport and telecommunications sectors, specifies rules and restrictions to the scope of technical specifications.

The Requirements Specification interface is highlighted in Figure 4.

2.1 Types of requirements

2.1.1 Product types

There are many types of products that can be purchased, and the requirement specifications will differ depending on the various product types. In P619, products were classified as belonging to one of the following three *market* types:

- *Innovative products (IN).* A product is innovative if the purchaser has little experience with this or similar products. This implies that the purchaser will have to do more work in specifying his requirements. Maybe he also is limited to define only some requirements that he estimates to satisfy the purpose of the product, leaving the rest of the requirements to be defined in collaboration with the supplier.
- *Known products (KN).* A product is known to a PNO if the PNO already has experience with this or similar products. This implies that the PNO probably has to do less work in requirements specification than for the innovative product. An example is when the product is a new version of an existing product.

Abbreviations

ATM	Asynchronous Transfer Mode
EIRUS	E-IPQM and E-RQMS Users
E-IPQM	European IPQM
E-RQMS	European RQMS
ETSI	European Telecommunications Standards Institute
EURESCOM	European Institute for Research and Strategic Studies in Telecommunications GmbH
IEC	International Electrotechnical Commission
ICS	Implementation Conformance Statement
IN	Innovative Product
IPQM	In-Process Quality Metrics
ISO	International Standardisation Organisation
ITU	International Telecommunications Union
KN	Known Product
OAM	Operation and Maintenance
OTS	Off-the-Shelf Product
PNO	Public Network Operator
RQMS	Reliability and Quality Measurements for Telecommunication Systems
SPICE	Software Process Improvement and Capability dEtermination
SM	Supplier Monitoring
SQ	Supplier Qualification
Trillium	Model for Telecom Product Development & Support Process Capability

Table 1 Sources for Process Requirements [1]

PROCESS		
Source	**Abbreviation**	**Requirements Specification**
EURESCOM P619 'PNO-Suppliers Technical Interfaces' *(Based on EURESCOM P227, EURESCOM P307)*	P619	D1: Guidelines on Requirements Specification. D2: Update of the P227 Deliverable, in the form of a road map for finding updated base-documents. D3: Guidelines on Supplier Qualification and Monitoring, D4: Quality Measurements according to E-IPQM and E-RQMS Revision 2. The requirements should mention the E-IPQM and E-RQMS quality measurements for use during product delivery and operation and maintenance.
EURESCOM P227 'Software Quality Assurance' *(ISO 12207)*	P227	In its quality assurance activities and with its reference to ISO 12207 [18], the P227 Deliverables contains many detailed definitions of processes, activities and tasks that can be used to define quality assurance as part of the entire life cycle of a software product. P619, D2, added a road map to the P227 Deliverables through which updated base-documents can be found.
EURESCOM P516 'Telecom Software Validation Procedures' *(ISO/IEC 9126)*	P516	When specifying a product, the requirements should use the quality characteristics of ISO/IEC 9126 [15].
SPICE and *(ISO/IEC TR 15504)*	SPICE99	The SPICE project (Software Process Improvement Capability dEtermination) developed a standard for software process assessment [19].
ISO/IEC 12119 'Quality Requirements' (ISO/IEC 9126)	ISO12219	ISO 12119 [17] defines requirements for software packages concerning quality and testing. As such, it provides input for specifying the requirements for a software product. Furthermore, it uses the ISO/IEC 9126 quality characteristics.
ISO 9000 Series Quality Assurance Standards *(ISO 8402, ISO 9000-3, ISO 9001, ISO 9004, ISO 9004-2)*	ISO9000	ISO 9000 Series specifies requirements on the quality system and process used by a company, and requires strict certification by an authorised body [10, 11, 12, 13, 14]. The ISO 9000 Series will be updated in 1999.
European Council Directive 93/38	ECD93/38	This Directive [9] puts restrictions on the scope of Technical Specifications.

- *Off-the shelf products (OTS).* An OTS product is produced by a supplier for a general market, to the supplier's own specifications, and it is simply purchased by a PNO.

Variations or combinations of the above product types are of course possible.

2.1.2 Functional types

It is also useful to make a technical division of telecommunication products into *functional types* like transmission systems, switching systems, operation, administration and maintenance systems, and possibly other systems.

2.1.3 Process requirements

Requirements can be specified for a product and for the process by which the product is created, delivered, and supported.

General *Process Requirements* can be considered as specifications regarding:

- Rules for Quality Assurance:
  - Traceability rules
  - Configuration management rules
  - Change control and corrective actions procedures and rules
- Test definitions
  - Verification/validation test definition
  - Qualification test definition
  - Serial production test definition
  - Compliance to standards and procedures
- Acceptance procedure and criteria
- Required measurement procedures for process quality measurements.

2.1.4 Product requirements

General *Product Requirements* can be considered as specifications addressing:

- Functional and structural requirements
  - Data model requirements
  - Functional requirements
  - Human-engineering requirements
  - Operation and maintenance requirements
  - Design constraints
  - Usability requirements and user interfaces
  - Interfaces to other systems
  - Environment and domain issues
  - Criticality
  - Compatibility
  - Performance
  - Dimensional
  - Capacity, hardware requirements
  - Safety and security
  - Environmental
  - Electro-magnetic compatibility, protection
  - Electro-static protection

- Marking, packaging, labelling
- Shipment, installation

- Quality parameters
  - Incoming product quality indexes
  - Complexity
  - Reliability
  - Availability
  - Maintainability
  - Modularity
  - Upgradeability
  - Portability
  - Operational efficiency
  - Criticality
- Required measurement procedures for product quality measurements.

2.1.5 Guidelines

There is a number of guidelines for making relevant specifications. A selected sample of sources for producing requirement specifications is listed in Tables 1 and 2.

Table 1 includes sources of requirement specifications defining acceptance and operation methods and processes and assuring, as much as possible, process conformity to specified requirements.

Table 2 includes sources of technical specifications of products as well as acceptance and operation criteria for the assurance of product conformity to specified requirements.

More general issues for products like power levels, cable characteristics, protocols, interfaces etc. can be found in various recommendations and standards from ETSI and ITU. Hardware products like electrical components, equipment technology, electrical safety etc. are addressed in standards from ISO. Issues related to reliability, dependability, quality of service, risk management, reliability testing, etc. can be found in standards from IEC.

As Table 2 shows, results from the international non-formal organisations and groups like ATM-Forum, DAVIC, IETF and FSAN are to be considered.

Table 2 Sources for Product Requirements [1]

PRODUCT		
Source	**Abbreviation**	**Requirements Specification**
Standards of the ISO-IEC	ISO-IEC	Such issues as power levels, cable characteristics, protocols, safety requirements, software characteristics etc.
ISO/IEC 9126 'Quality characteristics and guidelines for their use'	ISO9126	Definitions of quality characteristics for software like Functionality, Reliability, Usability, Efficiency, Maintainability, and Portability, and definitions of a process for using these six characteristics in assessing software quality requirements definitions.
ISO/IEC 9646-7 Information Technology – Open System Interconnection – conformance testing methodology and framework *Part 7:* Implementation Conformance Statement	ISO9646	Specifies requirements for the development of Implementation Conformance Statements (ICS) [16]. The supplier of an implementation should make a statement to conform to a given specification, stating which capabilities have been implemented.
Specifications of ad-hoc international groups as ATM-Forum, DAVIC, IETF, FSAN etc.	ATM-Forum, DAVIC, IETF, FSAN	Various technical and functional characteristics.
EURESCOM P518 'Telecommunication and the Environment'	P518	The requirements should mention the Guidelines for Environmentally Responsible Procurement for use during Product Delivery.

2.1.6 Requirements specification check list

Deliverable D1 [1] presents guidelines on producing requirements specification. The guidelines propose a checklist to be used as a reference list for creating requirement specifications. The checklist defines general requirements and applicable additional requirements for each of the types of a product's components types: software, hardware, and services[1)]. Then the relevance of a requirement source to Innovative, Known, and Off-the-Shelf product is indicated with low (l), normal (n), and high (h) – as shown in Table 3.

The table contains both product and process requirements sources.

Chosen or preferred measurement methods suitable for measuring the quality parameters should be included in a requirement specification. PNO and suppliers should also agree on the metrics to be used in measuring the parameters contained in the specification.

2.2 Recommendations

Some of the recommendations in P619 to the Requirements Specification Interface are [1]:

- A Requirements Specification document should contain product and/or process requirements where applicable, related to all groups of categories of products, all levels of analysis in each category, for the entire life-cycle of the product up to and including disposal.
- Requirements should be written according to the guidelines of European Council Directive 93/38. The Directive includes details for technical

[1)] *Services provided by the supplier as part of the product such as training, installation, and operation and maintenance.*

Table 3 Requirements checklist with relevance for Innovative, Known and Off-the-shelf products [1]

CHECKLIST			Relevance		
Applicability	**Source**	**General Requirement**	**IN**	**KN**	**OTS**
Requirements for all product types	P619	Quality Measurements	n	n	l
	P227	Rules for Quality Assurance	n	n	l
	ISO9000	Rules for Quality Assurance	h	n	l
	ECD93/38		n	l	n
	ITU	Functional / structural description	n	n	n
	P619	Supplier monitoring	h	n	l
	P518	Design for the Environment and End-of-Life Options	h	h	h
Additional requirements for Software	ETSI	Functional / structural description	n	h	h
	ISO9646	Test definition	l	h	h
	ISO12219	Test definition	l	n	h
	ISO 9126	Generic quality definitions	n	n	n
	P516	Quality parameters	h	n	n
	SPICE ISO/IEC TR15504	Rules for Quality Assurance	n	n	n
	ATM-Forum, DAVIC, IETF, FSAN, and other ad-hoc international groups	Functional / structural description	n	h	h
Additional requirements for Hardware	ETSI	Functional / structural description	n	h	h
	IEC 747, 748	Reliability / dependability requirements	n	n	n
	CENELEC CECC	Reliability / dependability requirements	h	n	n
	ATM-Forum, DAVIC, IEFT, FSAN, and other ad-hoc international groups	Functional / structural description	n	h	h
Additional requirements for Services	ISO 9004-2	Rules for Quality Assurance	h	n	n

specifications and details for qualification procedures. It is applicable if the product or service to be acquired is to be used for concession or enforced activities in the field of telecommunications. The Directive describes only the first steps of the product life cycle as Technical specification, Call for tender, Supplier qualification, and Product selection. Refer to standards whenever possible.

- For communication with suppliers about requirements, it is recommended to use an approach according to the Implementation Conformance Statement from ISO 9646 (see Table 1). This approach needs to be described in detail.

3 Supplier qualification and supplier monitoring interface

Supplier Qualification and Supplier Monitoring are two activities conducted by the PNO. Both activities are concerned with procurement of telecommunications products and related processes used at the supplier.

The Supplier Qualification and Supplier Monitoring interface aspects and guideline sources are highlighted in Figure 5.

3.1 Supplier qualification

A supplier should have a demonstrated capability to furnish products that meet the PNO's requirements. The method of assessing such a capability before actual purchasing of any product is Supplier Qualification. Qualification can take place within a procurement session, or at any period of time chosen by a PNO. The intention of qualification is to create a list of qualified suppliers for future procurements.

The management of a list of qualified suppliers is an effective way of guaranteeing that all the suppliers meet the basic requirements about quality. By using a supplier qualification system, a PNO is allowed to deal with suppliers having obtained the characteristics considered necessary.

The need for a supplier qualification system is linked to the following goals, common to both the PNO and the supplier:

- Reduction of acquisition cost: the reduction can be achieved through a decrease of incoming cost (less controls, less goods stocked to be tested, less production line interrupts, less reworks and repairs).
- Increasing product quality: if a supplier is reliable concerning organisation, financial issues, processes, flexibility etc., it is also possible to assume a high quality of the product.
- Prevention of non-conformities: repetitiveness in the supplier's processes, both technical and organisational, increases the probability that a product is reliable and meets the requirements.

Supplier qualification focuses both on products and processes. For both products and processes the main document is the Requirements Specification where all the requirements to be met by the products and the processes are listed together with associated measurement methods and techniques.

For the qualification of processes, several reference documents are available. Two such reference documents, SPICE [19] and TRILLIUM [20], are briefly analysed and presented in D3, Annex 1 [3].

The SPICE project (Software Process Improvement and Capability dEtermination) was an international collaborative effort to develop a standard for software process assessment. The work done by the SPICE project is in 1998 published as an international standard (technical report) ISO/IEC TR 15504. The potential usage of the SPICE results is as a base document for Supplier Assessment and for Process Improvement.

The Trillium Model[2)] consists of the following chapters:

- Model overview;
- Implementation Guidelines;
- Model Description;
- Essential Information about Trillium Practices;
- Trillium Capability Areas, Road Maps and Practices.

[2)] *Model for Telecom Product Development & Support Process Capability©Bell Canada, 1994 (release 3).*

A Road Map in this context is a set of related practices that focus on an organisational area or need, or a specific element within the product development process. In total, 508 practices are identified. These practices are mainly in the form of references to practices defined in the underlying documentation. Trillium is claimed by Bell Canada to be used to assess the product development and support capability of prospective and existing suppliers of telecommunications or information-based products.

Measurements on products and processes are normally carried out during all life-cycle phases of the product. Quality measurements (e.g. E-IPQM and E-RQMS) are discussed in Section 4.

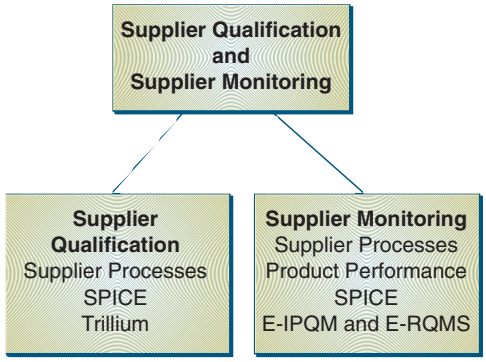


Figure 5 Supplier qualification and supplier monitoring interface aspects and guideline sources

3.2 Supplier monitoring

When procurement is decided and a contract is awarded, the supplier is monitored in order to assess the correct execution of the contract. The monitoring covers all factors involved in the procurement (depending on market types IN, KN or OTS, functional types as transmission systems, switching systems or other systems, technical properties, and life-cycle phases of the products).

Supplier monitoring can be done by measuring the performance of supplier's processes and of the quality of the products and services actually produced.

Although supplier monitoring and supplier qualification are performed in different time phases, they show many similarities because they both deal with product and processes related to a procurement, and they take into account the same factors though seen from different perspectives.

Using the results from supplier monitoring, the PNOs can assess the status of procurement and initiate corrective actions if necessary. Thus the results from the measurements can be used by the PNOs in the supplier qualification process for later purchases and of telecommunication products.

Measurements are normally performed with the life-cycle phases of the products in mind. These are summed up as follows:

- During the development phase, the suppliers' management of a development project can be monitored. Measurements of resources, changes and delays can be captured. In case of software development, characteristics of the produced code, like size and fault content, can be measured.
- At physical delivery, the discrepancy between planned and actual delivery dates can be recorded, as well as non-conforming functionality and quality characteristics.
- When in the operational phase, failure statistics, update and support capability measurements are important.

3.2.1 Supplier monitoring areas

As already mentioned, two groups of monitoring areas can be identified:

1 Supplier processes monitoring consists of:

- Monitoring of supplier processes (manufacturing and organisational);
- Monitoring of the specific development and delivery project (milestones);
- Monitoring of supplier services processes (after sale, installation, etc.).

Monitoring of supplier development processes can be done in the early phases of the development, even before a product is developed. The purpose is to assess and verify that the capability of the supplier is in conformance with the requirements specification.

2 Product performance monitoring
Product performance monitoring consists of:

Non-Official Standardisation Bodies

ATM-Forum	The ATM Forum, a non-profit international organisation formed in 1991, consists of a world-wide Technical Committee; three Marketing Committees for the Americas, Europe and Asia/Pacific; and the User Committee, in which ATM end-users participate. The ATM Forum comprises more than 900 member companies, and remains open to any organisation that is interested in accelerating the availability of ATM-based solutions.
DAVIC	Digital Audio Visual Council DAVIC is a non-profit association based in Switzerland, with a membership of over 175 companies from more than 25 countries. It represents all sectors of the audio-visual industry: manufacturing (computer, consumer electronics and telecommunications equipment) and service (broadcasting, telecommunications and CATV), as well as a number of government agencies and research organisations.
FSAN	Full Service Access Network FSAN is a group of 14 telecommunication companies who are working with their strategic equipment suppliers to agree upon a common broadband access system for the provision of both broadband and narrowband services. This common broadband access system is documented in the FSAN requirement specification. It is a public document, with the contents being presented to relevant standardisation bodies. The companies involved in FSAN are Bell Canada, Bell South, BT, DT, Dutch PTT, FT, GTE, KOREA Telecom, NTT, SBC, Swisscom, Telefonica, Telstra and Telecom Italia.
IETF	The Internet Engineering Task Force is a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet. It is open to any interested individual.

- Monitoring of the product characteristics during development (verification and validation);
- Monitoring of the product during manufacturing (requirements meeting, test results);
- Monitoring of the product performance during operation (reliability, availability) and during the telecommunication network management and maintenance.

During the later stages of the development, when the product exists in a prototype form, it becomes possible to assess the product itself. By reviewing the product during development and manufacturing, quality characteristics can be obtained. During the operational phase, actual failure performance can be measured.

3.2.2 Supplier monitoring procedures

In order to monitor a supplier, a corresponding agreement has to be made between the PNO and the supplier. The agreement is usually included in the business contract and further detailed in the requirement specification. The agreement includes the agreed activities and time schedules for procurement, and the monitoring to be done for relevant aspects. After the contract is signed, the agreement comes into force, and the actual monitoring of the supplier can start.

In the product requirement specification, all required properties are specified, i.e. functional requirements, quality requirements, and other. The fulfilment of all the specified properties can, in principle, be measured during the relevant life-cycle phases.

It is important that collected data relates to a plan with decision points or milestones, so that the supplier (or the PNO) can take actions to correct identified problems.

3.3 Results and application fields

Qualification or monitoring of suppliers can be accumulated in a measure of quality called Supplier Quality Level. A Supplier Quality Level may have many purposes such as ranking of suppliers, selection of the best supplier, contract awarding to a supplier, or quality monitoring.

In P619, a ‘universal’ method was sketched to perform qualification or monitoring of suppliers in an objective way and to obtain the Supplier Quality Level, on both products and processes, for all product categories, and all parameters involved. The method comprises three parts: the Prerequisites, the Analysis Part and the Synthesis Part.

The Prerequisites include the procurement scenario and the requirement specification.

The Analysis Part comprises the analysis of the procurement into simple cases, and the analysis of supplier qualification or supplier monitoring into simple cases.

A simple procurement case is defined as a case that includes only one type of equipment like a switching system, a transmission system, an operation and maintenance system, or some other system. Within the actual type of equipment, there will then be an indication of the ‘component’ category like hardware, software or services, and with indication of the market type category ‘known product’, ‘innovative product’ or ‘off-the-shelf product’.

A simple case for supplier qualification or supplier monitoring is defined as a case that deals with one simple procurement case as defined above. For this simple procurement case, the assessment method – for supplier qualification or for supplier monitoring – to be performed has to be chosen. Once this assessment method is chosen, an aspect, defined as a property or a feature of the telecommunication product, is chosen. And then, at last, the relevant life-cycle phase(s) for that aspect is chosen.

The Synthesis Part, starting with the assessments of simple supplier qualification (or supplier monitoring) cases, results in a value for the overall quality level as follows:

1 Measurements in each simple supplier qualification (or supplier monitoring) case take place.

2 Quality levels for simple supplier qualification or supplier monitoring cases are obtained.

3 Obtained quality levels are combined to higher-level quality levels.

4 Overall quality level for supplier qualification or supplier monitoring is obtained.

More details about this method can be found in D3, Annex 2 [3], together with application examples. The method, its effectiveness, its precision and the results, depend on the factors involved, their status and their evolution in time.

The requirement specification contains all elements about both products and related processes that are considered during qualification and monitoring of the supplier to ensure that the supplier meets all the requirements. The use of the proposed method for supplier qualification and supplier monitoring can therefore provide a means for improving the requirement specification in the appropriate topics.

The measurement methods E-IPQM and E-RQMS [4] can be used during the monitoring phase in order to verify whether the target quality levels are reached, both for processes and products.

3.4 Recommendations

Some of the recommendations to supplier qualification and supplier monitoring interfaces are according to D3 [3]:

- PNOs should create and keep up-to-date a requirements specification adapted to the needs of the proposed method for supplier qualification and supplier monitoring.
- PNOs should introduce the method in stages. Thus both PNOs and suppliers may become familiar with the application of the method and improve it according to practical experiences. In the first stage of this approach, some life-cycle phases may be grouped together, levels of analysis may be limited, and types of equipment may be grouped so that the number of cases and resulting measurements can be limited and better performed.

4 Quality measurement interface

The quality measurement interface between PNOs and suppliers consists of measurement methods and measurement systems where both the PNOs and the suppliers are involved in active co-operation.

During the development or modification of a telecommunications system, the suppliers' performance and progress are evaluated by the PNO by comparing the suppliers' actual process with his project plans. Conformance to these plans ensures that the product is delivered by the agreed time and with the required quality. The supplier is expected to provide measurement reports that define the current status of the work on the process of developing new products, and to report on the actions that they will take if the progress does not proceed according to the plans. The PNO will then evaluate the situation based on the supplier reports.

When in operation, the systems' performance will be monitored and analysed by the PNO by means of measurement reports. Information from some of these measurements will necessarily be supplied by the PNO itself, but the supplier provides the final measurement reports. The results from this analysis can then be passed to the supplier to support the product improvement process.

The quality measurement interface and relations to the major input deliverables are highlighted in Figure 6.

One of the focus points in P619 has been further development of the measurement systems E-IPQM and E-RQMS. The PNO will include in the requirement specification of a new telecommunication system the quality measurements that are to be used in the development phase and in the operational and maintenance phase of the system. The results from the measurements can be used by the PNO in the supplier qualification process as well as in the supplier monitoring process.

4.1 Results and application fields

The Bellcore measurement systems IPQM [24] and RQMS [25] have gone through considerable changes since the initial work of defining the European measurement systems was carried out in P307 in the period 1993 – 1995. There was therefore a need to clarify among the European PNOs participating in P619 and in EIRUS [21], how to handle these changes through defining the new European versions of E-IPQM and E-RQMS.

The main part of deliverable D4 [4] gives an overview of the measurement systems E-IPQM and E-RQMS Version 2 with definition of the measurements by Bellcore, the E-Addendum, data dissemination options, common targets options, application guidelines and system maintenance. In addition, D4 gives a discussion of the aspects of standardisation and of possible forms of co-operation with Bellcore, and of the changes in the status of EIRUS that would be implied. The Annex to D4 gives the detailed definition of E-IPQM and E-RQMS Version 2.

In order to maximise the impact and potential benefits of E-RQMS and E-IPQM, P619 recommended that they should be implemented uniformly by all PNOs. Organisations who apply the systems should become members of EIRUS in order to contribute to the development and update procedures for E-IPQM and E-RQMS.

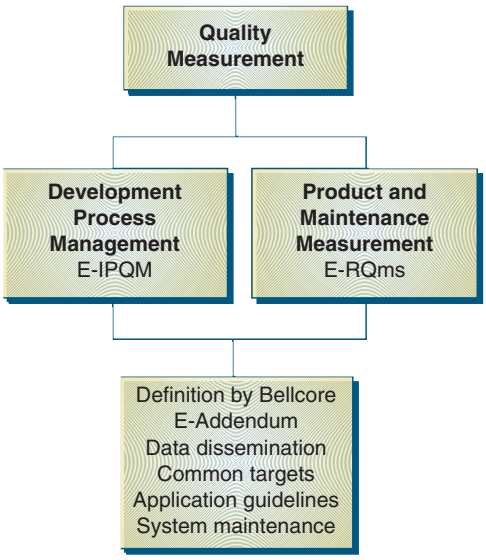


Figure 6 Quality measurement interface aspects

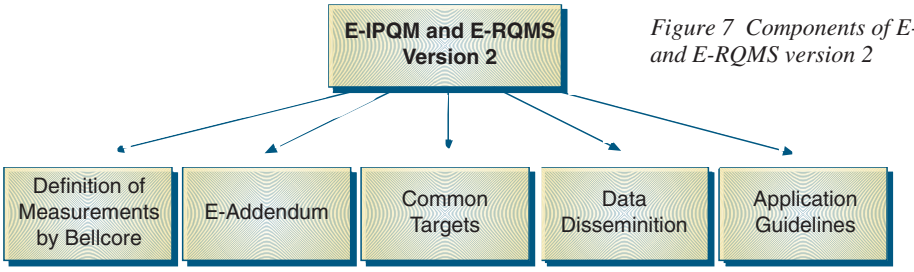


Figure 7 Components of E-IPQM and E-RQMS version 2

Due to its contractual nature, evaluation methods and metrics suitable for measuring the quality parameters should be included in a requirement specification. PNO and suppliers should agree about metrics to be used in measuring the parameters contained in the specification.

4.1.1 Definition of the measurements

The *definition* of Version 2 of E-IPQM and E-RQMS is as shown in Figure 7:

4.1.1.1 Definitions of measurements by Bellcore

The first definitions of the measurements in E-IPQM and E-RQMS are taken from the definitions of measurements by Bellcore [24, 25].

The main purpose of the In-Process Quality Metrics (IPQM) is to keep a customer (e.g. a PNO) informed about the development process of telecommunication software. This enables the PNO to investigate and control the development of the software, and to let the supplier take corrective actions if problems occur.

The main purpose of the Reliability and Quality Measurements for Telecommunications Systems (RQMS) is to measure trends in the reliability and quality of subsequent releases of telecommunication systems and software. Thus, RQMS measures vital quality aspects in the operation and maintenance phases.

From the Bellcore documents, only the definitions of the measurements are taken. Objectives, thresholds and other requirements as defined in the Bellcore documents are not a part of E-IPQM and E-RQMS. As to the E-IPQM and E-RQMS equivalent of objectives and thresholds, see discussion in section 4.1.1.4.

4.1.1.2 E-Addendum

Due to European needs, new or deviation measurements may have to be defined. Any deviations from the Bellcore measurement definitions will be included in the E-Addendum, which contains the information on measurements that are different from or extra to those referred in RQMS and IPQM. A maintenance procedure for the E-Addendum by using an E-Addendum Issue List mechanism and a template facilitating this maintenance has been proposed.

EIRUS was considered to be the most appropriate body to perform the maintenance of the E-Addendum.

4.1.1.3 Data dissemination options

Data dissemination of actual measured data among PNOs can give benefits to the PNOs involved. Careful consideration will be required to the type and detail of any data that is shared. Various options for data dissemination are analysed and described in more detail in D4. The options, which have been determined for, are:

1 Private Data Dissemination: The measurement data are collected, but are not disseminated outside the PNO. The measurement data are for internal use only.

2 Best Practice Data Dissemination: Only the best practices for each particular measurement are public and are published anonymously.

3 Aggregated Data Dissemination: The measurement data are disseminated anonymously in an aggregated form.

4 Full Data Dissemination: All measurement data are distributed among PNOs.

An assessment method is defined to enable the individual PNOs to assess the desirability and feasibility of the options available.

The ultimate decision as to the level of data dissemination practised will rest with the individual PNO. It is most likely that in the first instance, PNOs with limited experience in disseminating measurement results will need to reaffirm their policy on such matters. The implementation process is thus likely to evolve over time as confidence and the benefits of the process are better understood. It can therefore be expected that any preferred option for data dissemination between PNOs will be transitory and subject to change.

It is considered that in reality 'best practice' data dissemination will be the most likely entry point for many PNOs, and it is from this starting point that they will gain experience in the benefits available. It is expected that in certain fields of measurement, 'best practice' dissemination will lead to higher levels of data dissemination. The establishment of some kind of a 'Data Dissemination Centre' would then be required.

4.1.1.4 Common target options

To what degree the PNOs can agree on setting common target values that a product must meet was also an object of the study in P619. A target is, in this context, the value of a particular measurement that can be used as a goal for quality improvement.

The target values should be defined from experience and with care. The common targets should be used as goals; the use of the values in contracts should mirror this.

The following options, different for E-IPQM and E-RQMS, have been evaluated:

1 No Common Targets.

2 Common Targets for selected E-RQMS measurements for selected product makes.

3 Common Targets for selected E-RQMS measurements for all products of a defined type.

4 Common Targets for selected E-IPQM measurements by selected suppliers.

5 Common Targets for selected E-IPQM measurements for all suppliers.

Restrictions similar to those defined above for data dissemination apply to defining the optimum degree of common targets. It is expected that most PNOs will rank the desired common targets individually to reflect their own requirements. The actual values used would be derived by user groups such as EIRUS and implemented by groups of PNOs with similar interests.

To help the decision process, an assessment mechanism has been provided which covers data dissemination as well as common targets. This mechanism takes account of aspects of 'Quality Assurance' activities such as supplier qualification and monitoring as well as 'Feasibility Aspects', including confidentiality issues and resources needed.

Based upon experience gained within the USA, European PNOs should establish common targets, mainly for the E-RQMS measurement system, since the operators in USA for the time being still are lacking extensive experience from IPQM. The advancement of common targets to cover as many measurements as possible should become a key objective for the PNOs.

4.1.1.5 Application guidelines

The introduction of any quality system puts challenges on the staff of the PNOs concerned, both in the relationship with their suppliers and the integration of the quality system into their current operating procedures. It is recommended as application guidelines that PNOs introduce the measurements and associated procedures in a planned way, such as the one recommended by EIRUS.

The implementation programme devised by EIRUS is based on the two stages proposed by P307, known as Phases 1 and 2 [8]. The EIRUS scheme starts with a Phase 0 with a subset of the measurements that were in the P307 Phase 1. The splitting of the original Phase 1 set of measurements was established in response to difficulties found in applying the measurements that were in Phase 1 in a single step.

It would be prudent for any PNO planning an introduction schedule to approach EIRUS to gain the most current information on the status of measurements introduction within its member organisations. For guidance, the duration

Table 4 Duration of EIRUS phases of introduction

Phase 0	9 months
Phase 1	12 months
Phase 2	12 months

of the original EIRUS introduction of measurements was as described in Table 4.

The measurements contained within each phase are listed in Appendix A of Annex A to D4 [4].

4.1.1.6 Standardisation – co-operation with Bellcore

European parties should work towards a situation in which a European standard can be created for E-IPQM and E-RQMS, similar to the US situation. The experience gained by Bellcore within the USA could be exploited by finding an agreement with Bellcore in keeping a European standard up to date. EIRUS is a possible organisation to be involved in advancing a practical solution to this.

5 Conclusions

The competition between PNOs has increased during the last years. QoS provided to customers by the PNOs receives attention. Since QoS to customers strictly correlates with products and services PNOs receive from their own suppliers, incoming quality, seen from the PNOs, is a key issue to their business success.

The goal of the EURESCOM Project P619 was to improve and harmonise technical interfaces between PNOs and their suppliers. The perspective has been to improve the quality control of telecommunications products and services. The P619 project activities were in line with work carried out earlier by EURESCOM in several projects, starting with Project P227 and continued in Project P307. The results of P619 were presented to the public in a seminar in Heidelberg in March 1998, and all the project deliverables are now available to the public.

P619 has proposed a set of possible guidelines that PNOs should consider, though some of these proposals require further work. A concrete result from the work is the support to EIRUS and the dissemination of experiences gained by PNOs and supplier members of EIRUS.

Acknowledgement

This paper is based on a study done in EURESCOM Project P619. It is a pleasure to acknowledge the contributions from all the colleagues in the project. Especially an acknowledgement to Project Leader Dr. Marcello Melgara from CSELT, the Task Leaders Mr. Werner Deichmann from Deutsche Telekom, Mr. George Tsiamas from OTE, Mr. Chris Aldenhuijsen from KPN Research, and Mr. Paul Holder from BT. And last, but not least, an acknowledgement for the contributions from our colleagues at CSELT in writing P619 D6: Mr. Gianni Benso, Ms. Patrizia Bondi, and Ms. Laura Marchisio.

References

1 EURESCOM. *Guidelines on Requirements Specifications.* Heidelberg, 1998. (EURESCOM P619 D1.)

2 EURESCOM. *Update of the P227 Deliverables.* Heidelberg, 1998. (EURESCOM P619 D2.)

3 EURESCOM. *Guidelines on suppliers qualification and monitoring.* Heidelberg, 1998. (EURESCOM P619 D3.)

4 EURESCOM. *European Quality Measurements: Overview of E-IPQM and E-RQMS version 2.* Heidelberg, 1998. (EURESCOM P619 D4.)

5 EURESCOM. *Guidelines on hardware traceability.* Heidelberg, 1998. (EURESCOM P619 D5.)

6 EURESCOM. *Recommendations to improve the technical interfaces between PNO and suppliers.* Heidelberg, 1998. (EURESCOM P619 D6.)

7 EURESCOM. *Buy IT : recommended practices for Procurement of Telecommunication Software.* Heidelberg, 1995. (EURESCOM P227 D4.)

8 EURESCOM. *European quality measurements : E-IPQM and E-RQMS.* Heidelberg, 1995. (EURESCOM P307 D5.)

9 European Council. *Co-ordinating the procurement procedures of entities operating in the water, energy, transport and telecommunication sector.* 1993. (European Council Directive 93/38.)

10 ISO. *Quality management and quality assurance : vocabulary.* Geneva, 1994. (ISO 8402.)

11 ISO. *Quality management and quality assurance standards : guidelines for the application of ISO 9001 to the development, supply, and maintenance of software.* Geneva, 1992. (ISO 9000-3.)

12 ISO. *Quality systems : model for quality assurance in design/development, production, installation and servicing.* Geneva, 1994. (ISO 9001.)

13 ISO. *Quality management and quality system elements : guidelines.* Geneva, 1994. (ISO 9004-1.)

14 ISO. *Quality management and quality system elements : guidelines for services.* Geneva, 1991. (ISO 9004-2.)

15 ISO. *Information technology : software product evaluation : quality characteristics and guidelines for their use.* Geneva, 1991. (ISO/IEC 9126.)

16 ISO. *Information Technology : open systems interconnection : conformance testing methodology and framework. Part 7 : Implementation Conformance Statements.* Geneva, 1995. (ISO/IEC 9646-7.)

17 ISO. *Information technology : software packages : quality requirements and testing.* Geneva, 1994. (ISO/IEC 12119.)

18 ISO. *Information technology : software life cycle processes.* Geneva, 1995. (ISO/IEC 12207.)

19 ISO. *SPICE99 : Software Process Improvement and Capability dEtermination. Information technology : software process assessment, parts 1–9.* Geneva, 1998. (ISO/IEC TR 15504.)

20 *Trillium : model for telecom product development & support process capability.* Bell Canada, 1994. (Release 3.)

21 Johansson, R. EIRUS : a user group for quality measurements. *Telektronikk,* 93 (1), 86–88, 1997.

22 Groen, H et al. EURESCOM and QoS/NP related projects. *Telektronikk,* 93 (1), 184–195, 1997.

23 Skogstad, Ø. Review of international activities on software process improvements. *Telektronikk,* 93 (1), 101–108, 1997.

24 *In-Process Quality Metrics (IPQM).* Bellcore, 1995. (Generic Requirements GR-1315-CORE, Issue 1.)

25 *Reliability and Quality Measurements for Telecommunications Systems (RQMS).* Bellcore, 1996. (Generic Requirements GR-929-CORE, Issue 2.)

Jan-Erik Kosberg (53) graduated from the Norwegian Univ. of Science and Technology in 1969 and completed his Dr.Ing. thesis in 1973. He joined SINTEF in 1974 and is now Research Scientist at SINTEF Telecom and Informatics. His present research is on Quality of Service, and on infrastructures and services for information exchange in the Norwegian National Information Networks Program.

e-mail: Jan-Erik.Kosberg@informatics.sintef.no

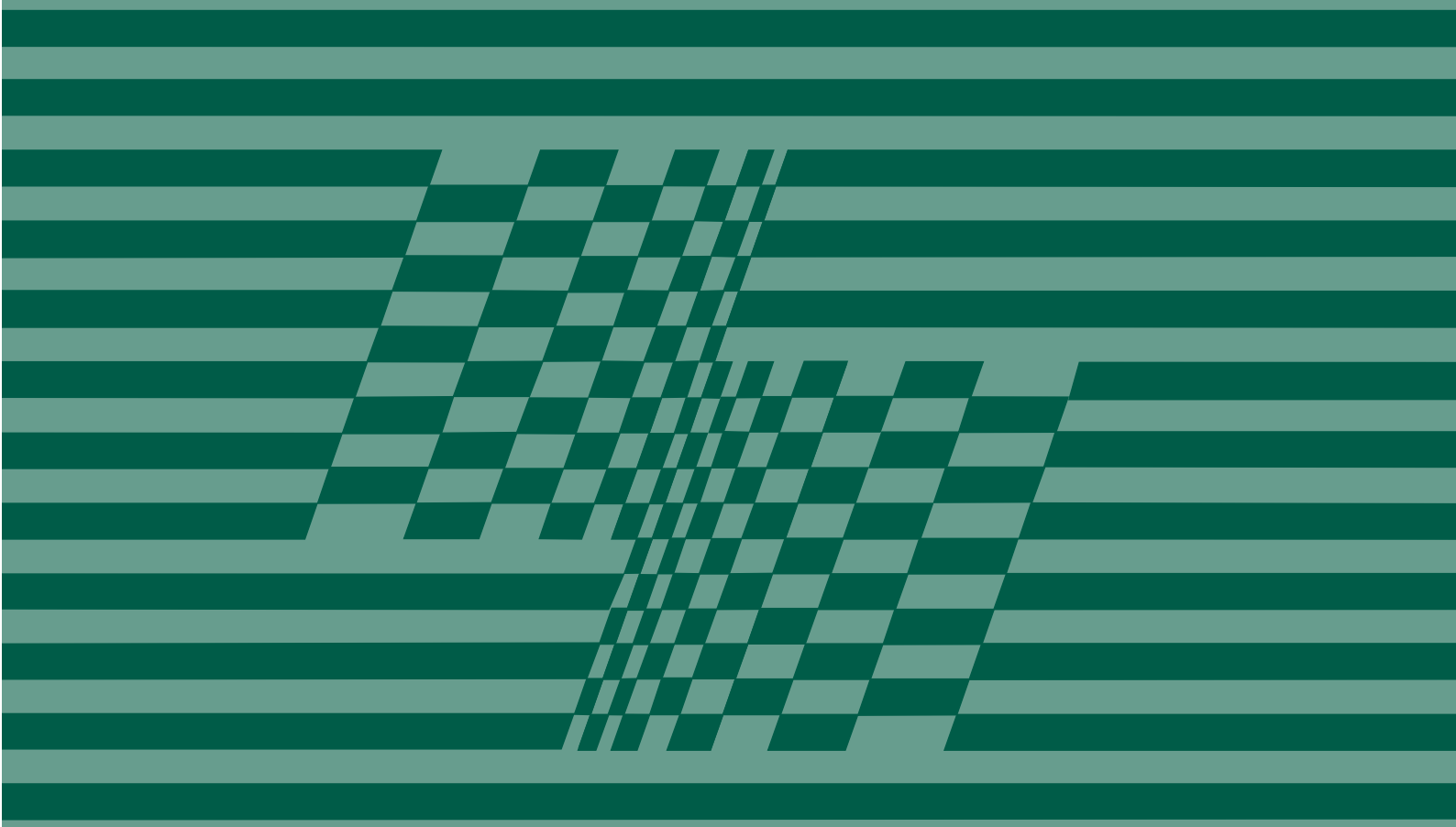
Øystein Skogstad (55) is Senior Scientist at SINTEF Telecom and Informatics, Trondheim. He is working with safety issues relating to electronic railway signalling systems. During his career he has also been engaged in research activities in various other areas including reliability in telecommunication switching, telecom network reliability, software quality assurance and software engineering methods.

e-mail: Oystein.Skogstad@informatics.sintef.no

Ola Espvik (55) is Senior Research Scientist and editor of Telektronikk. He has been with Telenor R&D since 1970 doing research in traffic engineering, simulation, reliability and measurements as well as being active in ITU, EURESCOM and several educational projects. His present research focus is on Quality of Service and Quality Assurance.

e-mail: ola.espvik@fou.telenor.no

Special



User perception of European network tones

FERGUS MCINNES, DONALD ANDERSON, MARK SCHMIDT AND MERVYN JACK

1 Introduction

A previous contribution to this journal [1] reported on work sponsored by the European Telecommunications Standards Institute (ETSI) which reviewed the problems arising out of a possible initiative to harmonise European telephone network tones. A number of technical aspects were reviewed, but in particular some of the human factors issues contributing to usability were dealt with at some length. An analysis of user actions and basic information about auditory perception and association led to the development of a descriptive model for tones which assigned functional characteristics to different phases of a call set-up procedure. The phases were embraced in a so-called SWAT model – named after the possible actions available to the caller: *Signal, Wait, Abort* and *Talk*. Tones were identified as *prompts for action* on the part of the caller or as *feedback*; a feedback tone could indicate that progress was being made and the user should wait, or that there was some problem and the user should abort the call. In most cases existing network tones were a good fit into the model, and it was suggested that the model could be used in a predictive way to help design new tones, provided that the function could be defined.

Although some indications were given in the literature about users' stereotyped associations for both frequency (pitch) and cadence (rhythm), no specific research existed which could confirm assumptions about tone assignment within the model. As an example, tones with high pitch or fast cadence were associated with urgency or importance, which could be interpreted as indicating that the user should do something *(prompt for action)*. Likewise, slow cadence or low frequency could be assigned the meaning *wait* (or take no action because the network is processing your last action). Thus, the model proposed in the ETSI report [2] was based only on theoretical or extrapolated assumptions from laboratory-tested concepts and not confirmed in any user testing in the context of telephony.

The Centre for Communication Interface Research (CCIR) at the University of Edinburgh undertook to investigate some of these behavioural associations. Using a set of existing European tones identified from the ETSI work, and a set of semantic opposites devised by the ETSI Project Team, CCIR carried out a user perception experiment, the results of which are reported here.

2 Aims of the experiment

The purpose of the experiment was to evaluate a number of existing tones with respect to whether they fulfil the basic functions for which they are intended, such as indicating that action from the user is required (e.g. pay tone), or indicating a particular status of the network (e.g. engaged, ringing). The tones were evaluated using a set of adjectives, such as *pleasant, disruptive, calming* or *boring,* which can be seen as contributing to the 'meaning' of a particular tone. This 'meaning' in turn may relate to the behaviour of the telephone user after hearing the tone, which in the end determines a tone's effectiveness and usefulness. A secondary aim was to test a particular methodology for evaluating such tones using semantic differential scales, whose extremes are labelled with two contrasting words or phrases. The goal of the experiment was to answer the following two questions:

- How do British subjects judge the particular network tones along a pre-defined set of semantic differential scales labelled with opposites such as 'pleasant – unpleasant'?
- Are there significant differences in subjects' judgements for different tones?

3 The ETSI model

The model [1, 2] classifies each tone by its functional characteristics. A tone may indicate to the caller that progress with a call is being made *(OK),* or that there is a problem in the network or with the particular call *(not OK)*. Also it may simply provide *feedback* or it may be a *prompt* for action from the user. Tones are further classified according to the phase in the call at which they occur – the *non-talk phase* in which the connection is established, or the *talk phase* in which speech or other information is transmitted between the connected parties – and as indicating *basic* or *sophisticated* functionality. This classification is shown in Figure 1, with examples of tones in different categories. The cells are coloured to represent the spectrum of call status severity noted by the ETSI team and indicated in the final version of their model (Figure 3 in [1]), and the special requirements for the talk phase, where a low duty cycle is recommended so as not to interfere with conversation.

The underlying motivation of the model is to determine what meaning each network tone should attempt to convey to the caller, establish how effectively each tone conveys its intended meaning, and determine whether these functional characteristics have acoustic equivalents in terms of cadence, frequency, intensity and duration. For example, Pollack (1952) showed that tones with long cadences imply 'goodness', whereas tones with short cadences imply 'badness' and elicit a greater feeling of urgency than long cadences [3].

The goal of the ETSI project's assignment of tones is greater harmonisation of existing European network tones and in the long term this may aid the design of new, audibly distinct and more widely recognised network tones.

4 Tones evaluated

The 18 tones evaluated in this experiment are listed in Table 1.[1)] They are classified according to the ETSI model: those occurring in the talk phase are asterisked.

The notation for cadences is as in [1] and [2]: durations of 'on' intervals (tone bursts) are given in bold, and durations of 'off' intervals (silences) in normal type. Where a tone is specified to occur only a limited number of times, this is shown in square brackets in the 'Cadence' column; otherwise the tone is designed to be repeated indefinitely (subject to any applicable time-out). Frequencies separated by '/' within a tone specification occur in succession, while those separated by '+' occur simultaneously. Where simultaneous frequencies have different

[1)] *In fact, 19 tones were evaluated, but it was subsequently discovered that one of them (intended to be the ETSI recommended Special Information/Number Unobtainable tone [4]) did not match its specifications, as it had been generated with the wrong sequence of frequencies; the results for this tone have therefore been omitted from the paper.*

		OK		NOT OK	
		Non-Talk	Talk	Non-Talk	Talk
FEEDBACK	**Basic**	Ring, Comfort, Holding, Caller Waiting	Conference	Busy, Number Unobtainable	Pre-Empt
	Sophisticated	Special Ring, Positive Indication		Congestion, Negative Indication	Intrusion, Warning
PROMPT – CAN	**Basic**	Dial, Second Dial	Call Waiting		
	Sophisticated	Special Dial			
PROMPT – MUST	**Basic**	UPT (PUI/PIN), Record	Pay		
	Sophisticated				

Figure 1 The ETSI model of tone functions

		OK		NOT OK	
		Non-Talk	Talk	Non-Talk	Talk
FEEDBACK	**Basic**	2 (ETSI & UK Ring)		4 (ETSI & UK Busy, UK SI, UK NU)	
	Sophisticated			2 (ETSI & UK Congestion)	1 (ETSI Warning)
PROMPT – CAN	**Basic**	2 (ETSI & UK Dial)	2 (ETSI & UK Call Waiting)		
	Sophisticated	2 (UK Special Dial: System X & System Y)			
PROMPT – MUST	**Basic**	1 (ISO/IEC Record)	2 (UK & Sweden Pay)		
	Sophisticated				

Figure 2 Numbers and descriptions of tones evaluated, classified according to the ETSI model

Table 1 Details of tones evaluated

	Source	Tone	Classification	Cadence	Frequency
1.	United Kingdom	Dial	Prompt – Can	continuous	350+440
2.	United Kingdom	Special Dial (System X; GPT)	Prompt – Can	**0.75** – 0.75	350+440
3.	United Kingdom	Special Dial (System Y; Ericsson)	Prompt – Can	continuous + **0.75** – 0.75	440 + 350
4.	United Kingdom	Ring	Feedback OK	**0.4** – 0.2 – **0.4** – 2.0	400+450
5.	United Kingdom	Busy	Feedback Not-OK	**0.375** – 0.375	400
6.	United Kingdom	Congestion	Feedback Not-OK	**0.4** – 0.35 – **0.225** – 0.525	400
7.	United Kingdom	Number Unobtainable	Feedback Not-OK	continuous	400
8.	United Kingdom	Special Information	Feedback Not-OK	**0.33** – 0.03 – **0.33** – 0.03 – **0.33** – 0.0	950/1400/1800
9.	United Kingdom	Pay	Prompt – Must *	**0.125** – 0.125	400
10.	United Kingdom	Call Waiting	Prompt – Can *	**0.1** – 3.0	400
11.	ETR 187	Dial	Prompt – Can	continuous	425
12.	ETR 187	Ring	Feedback OK	**1.0** – 4.0	425
13.	ETR 187	Busy	Feedback Not-OK	**0.5** – 0.5	425
14.	ETR 187	Congestion	Feedback Not-OK	**0.2** – 0.2	425
15.	ETR 187	Call Waiting	Prompt – Can *	**0.2** – 0.6 – **0.2** – 3.0 [x2]	425
16.	ETR 187	Warning	Feedback Not-OK *	**0.5** – 15.0	1400
17.	Sweden	Pay	Prompt – Must *	**0.2** – 0.2 – **0.2** – 3.6 [x2]	941
18.	ISO/IEC 13174	Record	Prompt – Must	**0.15** – 0.075 – **0.15** [x1]	500/620

cadences (tone no. 3), the respective cadences are separated by '+' in the 'Cadence' column.

The locations of these tones in the ETSI model are shown in Figure 2.

5 Experiment procedure

A total of 100 subjects took part in the experiment and each evaluated all the tones in a listening experiment. Subjects were balanced for age and gender and were presented with each tone after pressing any key on the keypad of the telephone they were using for the experiment. This was the same for each subject as all subjects completed the experiment at CCIR. Simple continuous tones were presented with a duration of 5 seconds. Repetitive tone sequences and chimes with an unspecified number of cycles were presented in cycles of 6 instances. Other tone sequences or chimes with a fixed number of cycles were presented accordingly (e.g. the Swedish pay tone has 2 cycles of a tone sequence). All tones were presented at -6 dBm0, as this takes adequate account of signal loss between the exchange and the handset. The order of presentation of the tones was randomised for each subject.

Following each tone, subjects indicated their judgements on a set of semantic differential scales. Semantic differential scales are used to assess aspects of meaning with the use of adjectival opposites [5]. The scales adopted for this experiment, after discussions between members of CCIR and the ETSI Project Team, are shown in Figure 3. Subjects were asked to evaluate each tone on all seven scales.

Alarming	:___:___:___:___:___:___:___:	Calming
Reassuring	:___:___:___:___:___:___:___:	Worrying
Urgent	:___:___:___:___:___:___:___:	Not urgent
Pleasant	:___:___:___:___:___:___:___:	Unpleasant
Interesting	:___:___:___:___:___:___:___:	Boring
Disruptive	:___:___:___:___:___:___:___:	Not disruptive
Requires action	:___:___:___:___:___:___:___:	Does not require action

Figure 3 Semantic differential scales used to evaluate the tones

6 Results

This experiment yielded mean ratings (obtained by averaging across the 100 subjects) for 18 tones on seven semantic differential scales. These results can be presented and analysed in various ways: taking one *tone* at a time and plotting a profile of its ratings on all the scales, or taking one *scale* at a time and deriving a ranking of all the tones on that scale. In the latter case, pairwise statistical comparisons of tones can also be made, using the individual subjects' responses.

Section 6.1 below presents the results in the form of a profile for each tone. Section 6.2 gives the ranking of the tones on each semantic differential scale, and also on a pooled scale derived after observing strong correlations among six of the seven original scales. Section 6.3 presents a characterisation of the tones on two dimensions, the first corresponding to the pooled scale and the second to the 'Interesting – Boring' scale.

6.1 Profiles of tones across all semantic differential scales

Figures 4 to 6 show the mean ratings on all the semantic differential scales for each of the tones. Tones are grouped according to their context of occurrence and function, as follows:

- Prompt, Non-Talk: dial and record tones (tones 1, 2, 3, 11 and 18) – Figure 4;
- Feedback, Non-Talk: tones occurring after dialling and giving feedback on success (ring tone) or failure (busy, congestion, number unobtainable or special information) (tones 4, 5, 6, 7, 8, 12, 13 and 14) – Figure 5;
- Talk: tones occurring during the talk phase of a call, comprising pay, call waiting and warning tones (tones 9, 10, 15, 16 and 17) – Figure 6.

On each scale, labelled 'A – B' where A and B are the contrasting adjectives or phrases, high scores indicate ratings towards 'A', and low scores indicate ratings towards 'B'. Possible scores range from 1 to 7; a value of 4 is neutral.

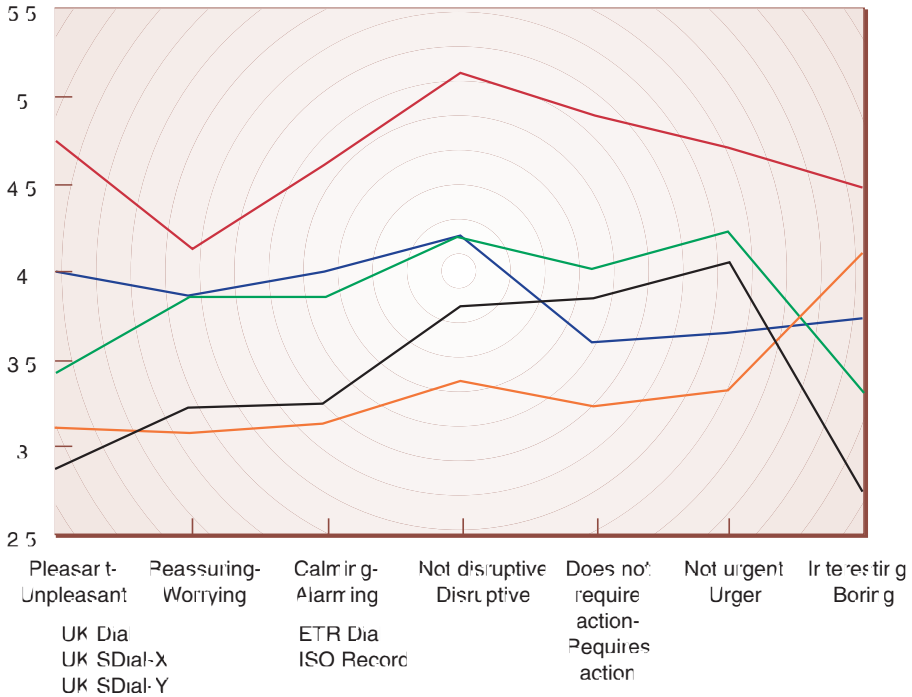


Figure 4 Ratings of 'Prompt Non-Talk' tones

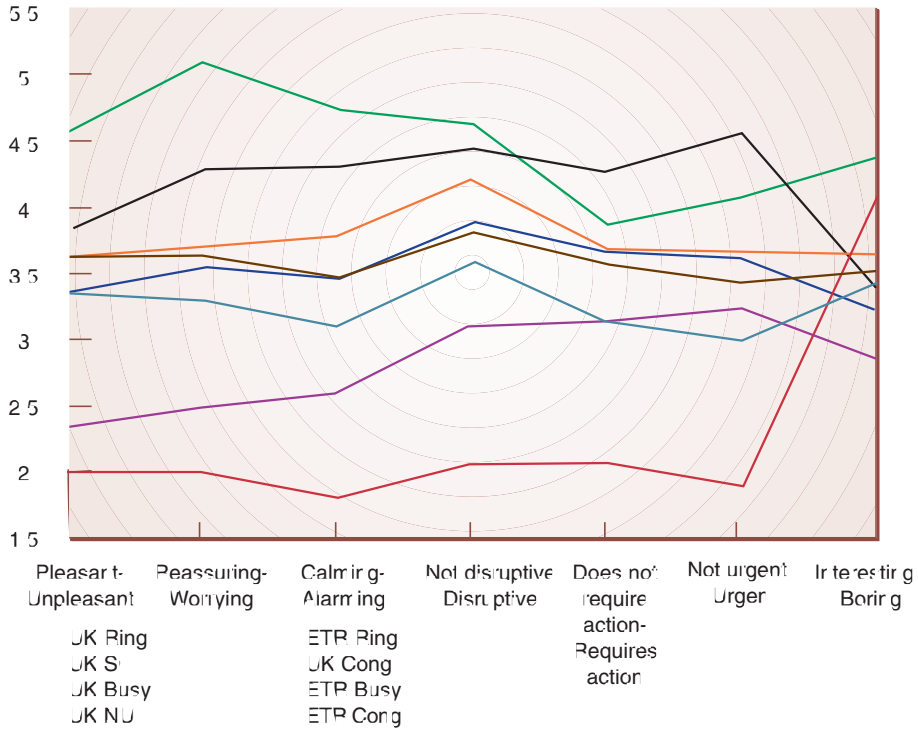


Figure 5 Ratings of 'Feedback Non-Talk' tones

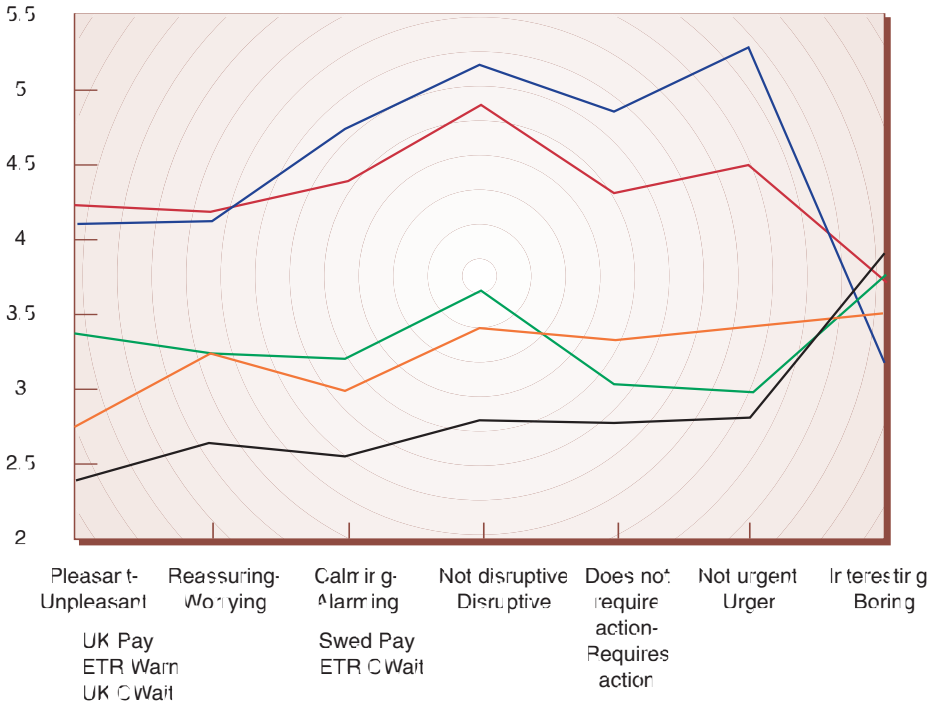


Figure 6 Ratings of 'Talk' tones

6.1.1 'Prompt Non-Talk' tones

As can be seen in Figure 4, the four dial tones evaluated were all rated near or slightly below neutral on all the scales. The UK and System X dial tones were found more pleasant, reassuring and calming and less disruptive than the System Y and ETSI counterparts. The other three scales show a different pattern, with the two basic dial tones (UK and ETSI – both with a simple continuous cadence) judged less urgent and requiring of action and more boring than the System Y dial tone, and the System X tone rated intermediately. In most cases the contrasts were statistically significant, and in some cases highly significant.

The ISO record tone was placed higher than all the dial tones on every scale, and in most cases this was highly significant. For some of the scales (e.g. non-disruptiveness) this can perhaps be attributed to its being a once-only rather than recurring or continuous tone.

6.1.2 'Feedback Non-Talk' tones

As shown in Figure 5, both the ring tones tested were rated above neutral on most of the first six scales, whereas the other tones (indicating failure to make a connection) were scored below neutral to a greater or lesser degree. This is as it should be if the tones have been designed appropriately for their functions as giving positive ('OK') and negative ('not-OK') feedback. However, the wide range of scores across the set of 'not-OK' feedback tones warrants examination.

The most unpleasant, worrying, alarming, disruptive, action-requiring and urgent tone was the UK special information tone, with ratings near 2 on all the first six scales. (On each of these scales, the difference between this tone and any of the others in the group was highly significant, with the one exception that the UK number unobtainable tone was not quite significantly less unpleasant.) Distinctive characteristics of this tone are the wide range of frequencies, including one as high as 1800 Hz, and the lack of silent intervals.

Next lowest on most of the scales was the UK number unobtainable tone. It is interesting to note that this scored significantly lower on all the first six scales than the ETSI dial tone (seen in Figure 4) although the only difference in their specifications is between their frequencies of 400 Hz and 425 Hz respectively. Recognition of the UK tone by UK subjects as having a negative meaning may have played a part here.

The other four tones – the UK and ETSI busy and congestion tone – were all rated just slightly below the neutral point on all the scales, with the exception of the UK congestion tone which was slightly above neutral for non-disruptiveness.

While the UK ring and special information tones were at opposite extremes on most of the other scales, they were similarly rated on the 'Interesting – Boring' scale, both being judged more interesting than any of the others in the group.

6.1.3 'Talk' tones

Figure 6 shows that the tones occurring in the talk phase of a call fall into two groups according to their scores on the first six scales. Above neutral on all these scales are the UK and ETSI call waiting tones; below neutral are the UK and Swedish pay tones and the ETSI warning tone. These ratings seem appropriate in view of the tones' functions: pay tones, being prompts for mandatory action (without which the call will be terminated), ought to be more intrusive and urgent than call waiting tones which are prompts for optional action, while the warning tone (indicating that the privacy of the call may be compromised) likewise conveys important information of which the caller needs to be aware.

Here, as in Figures 4 and 5, the first six scales show broad coherence in their rankings of the tones but the 'Interesting – Boring' scale yields very different results. In this case all the tones in the group were similarly judged to be slightly on the boring side of neutral.

6.2 Ranking and comparison of tones on each scale

For each scale *(adjective1 – adjective2),* the tones are listed in order from the *adjective1* end of the scale to the *adjective2* end, with their mean ratings on the scale. A horizontal line marks the neutral point on each scale.

6.2.1 'Pleasant – Unpleasant' scale

18	ISO Record	4.75
4	UK Ring	4.58
15	ETR CWait	4.24
10	UK CWait	4.11
2	UK SDial-X	4.01
12	ETR Ring	3.85
6	UK Cong	3.62
13	ETR Busy	3.61
1	UK Dial	3.42
9	UK Pay	3.38
14	ETR Cong	3.37
5	UK Busy	3.34
3	UK SDial-Y	3.12
11	ETR Dial	2.88
16	ETR Warn	2.76
17	Swed Pay	2.41
7	UK NU	2.34
8	UK SI	2.01

Comments:

Most tones were rated on the 'unpleasant' side of the neutral point (4 on the scale). Only the ISO record and UK ring tones were placed substantially on the 'pleasant' side, though both UK and ETSI call waiting tones were slightly on the 'pleasant' side.

6.2.2 'Reassuring – Worrying' scale

4	UK Ring	5.08
12	ETR Ring	4.26
15	ETR CWait	4.19
10	UK CWait	4.13
18	ISO Record	4.12
2	UK SDial-X	3.86
1	UK Dial	3.86
6	UK Cong	3.71
13	ETR Busy	3.63
5	UK Busy	3.54
14	ETR Cong	3.30
9	UK Pay	3.25
16	ETR Warn	3.24
11	ETR Dial	3.22
3	UK SDial-Y	3.08
17	Swed Pay	2.64
7	UK NU	2.48
8	UK SI	2.01

Comments:

The UK ring tone was much the most reassuring of all the tones assessed, which is appropriate given its function. However, this may be an effect of the subjects' familiarity with this tone as conveying an 'OK' signal. It would be interesting to see whether subjects in other countries considered it similarly reassuring.

In general, tones with a 'Feedback OK' or 'Prompt – Can' function were considered reassuring or only slightly worrying, and those with other functions were considered worrying. Exceptions are the ISO record tone (placed on the reassuring side of neutral despite being categorised as 'Prompt – Must') and the ETSI and System Y dial tones (considered worrying although intended for a 'Prompt – Can' function).

6.2.3 'Calming – Alarming' scale

10	UK CWait	4.76
4	UK Ring	4.73
18	ISO Record	4.60
15	ETR CWait	4.41
12	ETR Ring	4.29
2	UK SDial-X	3.99
1	UK Dial	3.85
6	UK Cong	3.77
13	ETR Busy	3.47
5	UK Busy	3.46
11	ETR Dial	3.25
9	UK Pay	3.19
3	UK SDial-Y	3.13
14	ETR Cong	3.09
16	ETR Warn	2.99
7	UK NU	2.58
17	Swed Pay	2.56
8	UK SI	1.80

Comments:

The same five tones were rated above neutral on this scale as on the 'Reassuring – Worrying' scale, though the UK ring tone no longer stood out from the rest. The scores for the remaining tones were similar on the two scales, and similar remarks apply.

6.2.4 'Not disruptive – Disruptive' scale

10	UK CWait	5.17
18	ISO Record	5.12
15	ETR CWait	4.90
4	UK Ring	4.62
12	ETR Ring	4.43
6	UK Cong	4.22
2	UK SDial-X	4.21
1	UK Dial	4.19
5	UK Busy	3.87
11	ETR Dial	3.80
13	ETR Busy	3.80
9	UK Pay	3.66
14	ETR Cong	3.60
16	ETR Warn	3.42
3	UK SDial-Y	3.38
7	UK NU	3.10
17	Swed Pay	2.80
8	UK SI	2.04

Comments:

Ratings on this scale tended to be higher than on the preceding ones, and this is reflected in the larger number of tones scoring above neutral. However, the relative scores and rank ordering are similar to those on the 'Calming – Alarming' scale.

The tones considered least disruptive were the two call waiting tones and the ISO record tone, all of which have short 'on' intervals (0.2s or less). However, the Swedish pay tone which also has short 'on' intervals was one of the most disruptive. This suggests that high frequency (found both in the Swedish pay tone and in the UK special information tone) is perceived as a disruptive characteristic.

6.2.5 'Does not require action – Requires action' scale

18	ISO Record	4.88
10	UK CWait	4.86
15	ETR CWait	4.33
12	ETR Ring	4.24
–1	UK Dial	4.00
4	UK Ring	3.87
11	ETR Dial	3.84
5	UK Busy	3.66
6	UK Cong	3.65
2	UK SDial-X	3.59
13	ETR Busy	3.58
16	ETR Warn	3.33
3	UK SDial-Y	3.23
7	UK NU	3.13
14	ETR Cong	3.12
9	UK Pay	3.03
17	Swed Pay	2.78
8	UK SI	2.05

Comments:

The scores and rankings on this scale were broadly similar to those on the preceding two scales, though the UK ring and pay tones and the System X dial tone were placed lower, and the UK dial tone higher, on this scale.

This perceptual scale relates more directly to function than any of the other scales, in that some tones, in their intended uses, do in fact require action and others do not. Those that most clearly do require action are those in the 'Prompt – Must' category, i.e. pay and record tones; 'Prompt – Can' and 'Feedback Not-OK' tones do not *require* action but do at least suggest or invite it; and 'Feedback OK' tones clearly do not require action. When this functional ranking is compared with the perceptual one, some anomalies emerge. In particular, the ISO record tone, designed for a 'Prompt – Must' function, and the two call waiting tones, classified as 'Prompt – Can', were all perceived as less requiring of action than any of the others. The UK and Swedish pay tones, however, were appropriately rated as requiring action, and within the 'Feedback' category all the 'Not-OK' tones were judged to require action more than those (ring tones) indicating an 'OK' status.

6.2.6 'Not Urgent – Urgent' scale

10	UK CWait	5.28
18	ISO Record	4.70
12	ETR Ring	4.54
15	ETR CWait	4.51
1	UK Dial	4.22
4	UK Ring	4.07
11	ETR Dial	4.04
6	UK Cong	3.67
2	UK SDial-X	3.64
5	UK Busy	3.61
13	ETR Busy	3.43
16	ETR Warn	3.42
3	UK SDial-Y	3.32
7	UK NU	3.22
14	ETR Cong	2.98
9	UK Pay	2.98
17	Swed Pay	2.82
8	UK SI	1.88

Comments:

The ranking of the tones on this scale was very similar to that on the preceding scale. Ring, call waiting and record tones and basic dial tones were judged 'not urgent' on average, while all the 'Feedback Not-OK' tones and the two special dial tones were rated as 'urgent'.

6.2.7 'Interesting – Boring' scale

18	ISO Record	4.47
4	UK Ring	4.34
3	UK SDial-Y	4.09
8	UK SI	4.08
17	Swed Pay	3.91
15	ETR CWait	3.76
9	UK Pay	3.73
2	UK SDial-X	3.73
6	UK Cong	3.63
16	ETR Warn	3.52
13	ETR Busy	3.52
14	ETR Cong	3.43
12	ETR Ring	3.42
1	UK Dial	3.30
5	UK Busy	3.23
10	UK CWait	3.18
7	UK NU	2.85
11	ETR Dial	2.74

Comments:

The ranking of the tones on this scale was very different from that on any of the other scales. For instance, the ISO record tone and the UK special information tone, which were near opposite ends of the rank order for each of the other adjective pairs, were here placed close together as both being 'interesting', while the UK call waiting and number unobtainable tones, also placed near the top and bottom respectively for each of the other scales, were both considered 'boring'. Some similarity might have been expected between the 'Interesting – Boring' and 'Pleasant – Unpleasant' scales, and indeed the same two tones appear at the top of both, but in other respects they differ widely, and the 'Pleasant – Unpleasant' scores are on the whole more closely related to those on the other five scales than to those on this one.

Multiple frequencies with different timing (found in tones 3, 8 and 18) and a complex cadence (tones 4, 6, 8, 15, 17 and 18) appear to contribute to making a tone interesting. In contrast, tones with a single frequency presented continuously (tones 7 and 11) are regarded as boring.

Ratings on this scale do not appear to relate to tone function: in several cases different tones with similar functions were scored very differently, including ring tones (4 and 12) and dial tones (1, 2, 3 and 11).

6.2.8 Pooled scale

Scores on the pooled scale were derived by taking the mean across all the original scales other than 'Interesting – Boring'. These scales had been observed to be strongly correlated. (This was confirmed by a factor analysis of the mean scores for the 18 tones on these six scales, in which the first factor was found to explain 91.8 % of the variance and yielded a rank order identical to that on the pooled scale. A factor analysis of the mean scores on all seven scales yielded two factors, the first similar to that in the six-scale analysis and almost independent of the 'Interesting – Boring' score, and the second having weighting 0.975 on the 'Interesting – Boring' score and less than 0.3 on each of the others, which together accounted for 95.7 % of the variance.)

A high score thus indicates a tone judged to be *pleasant, reassuring, calming, not disruptive, not requiring action* and *not urgent,* whereas a low score represents one considered *unpleasant, worrying, alarming, disruptive, requiring action* and *urgent.*

10	UK CWait	4.72
18	ISO Record	4.70
4	UK Ring	4.49
15	ETR CWait	4.43
12	ETR Ring	4.27
1	UK Dial	3.92
2	UK SDial-X	3.88
6	UK Cong	3.77
13	ETR Busy	3.59
5	UK Busy	3.58
11	ETR Dial	3.50
9	UK Pay	3.25
14	ETR Cong	3.24
3	UK SDial-Y	3.21
16	ETR Warn	3.19
7	UK NU	2.81
17	Swed Pay	2.67
8	UK SI	1.97

Comments:

The order of the tones on the pooled scale is very similar to that on the 'Calming – Alarming' scale. Ring, call waiting and record tones scored positively overall, and all the others scored negatively.

Where UK and other tones with the same function were compared, the UK tone was usually given a higher score than the non-UK one. (The only exception is for busy tones, where the UK and ETSI tones scored almost the same.) This may be a familiarity effect, given that the subjects were British; experiments with subjects from other countries would be necessary to elucidate this.

6.3 Characterisation of tones in a two-dimensional space

The observations in the previous section suggest that tones can be characterised perceptually on two dimensions, the first corresponding to the pooled scale defined in Section 6.2.8, and the second corresponding to the 'Interesting – Boring' scale.

Figure 7 shows the 18 tones plotted on these two dimensions. According to recommendations on subjective characteristics in the ETSI report [2], tones towards the left of the plot should be suitable for 'Feedback Not-OK' and 'Prompt – Must' functions, and those towards the right should be suitable for 'Feedback OK' and 'Prompt – Can' functions. By this criterion, most of the tones tested appear to be fairly well matched to their intended functions, the ISO record tone being an exception (as already remarked in Sections 6.2.2 and 6.2.5 above).

7 Summary and conclusions

The study described here set out to explore users' perceptions of a set of telephony tones using an inventory of semantic differential scales, so as to determine whether the perceptions matched the intended functions of the tones (as set out in the ETSI model), and whether differences in perceived characteristics between tones could be reliably detected using the semantic differential technique.

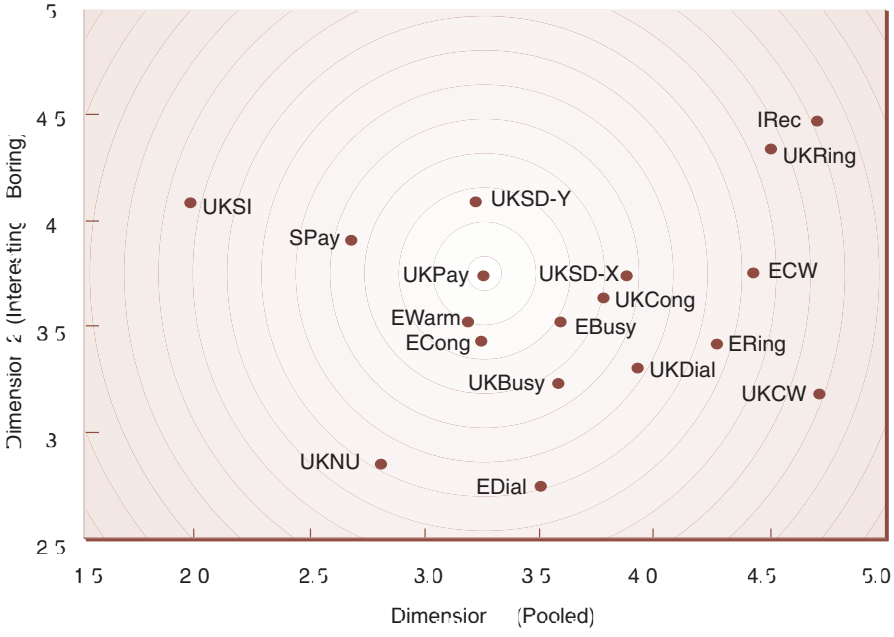


Figure 7 Positions of tones in two-dimensional space

Major differences were in fact found in the judgements of different tones. Most of the comparisons between pairs of tones on specific semantic differential scales attained a high degree of statistical significance on the sample of 100 subjects. Moreover, a clear pattern emerged, in which over 90 % of the inter-tone variance in the mean scores on six of the seven scales was explained by a single factor, so that the results could be summarised by locating each tone in a two-dimensional space.

The first of the two dimensions of perceptual characterisation contrasts tones which are *pleasant, reassuring, calming, not disruptive, not requiring action* and *not urgent* (having high values on the pooled scale) with those considered *unpleasant, worrying, alarming, disruptive, requiring action* and *urgent* (having low values). This dimension seems closely related to the contrasting subjective characteristics recommended in the ETSI report [2] for tones indicating 'OK' and 'not OK' system status.

The second dimension corresponds to the single semantic differential scale from *interesting* to *boring.* Tones incorporating multiple frequencies or complex cadence were judged most interesting, and those with a single frequency played continuously were found most boring. It is not immediately obvious how this dimension relates to the functional classification of tones.

Results on specific tones were as follows:

- The dial tones (UK and ETSI) and special dial tones (System X and System Y) ranged from near neutral to fairly low on the first dimension, but were mostly near the middle of the ranked list, which seems reasonable given their 'Prompt – Can' function.
- The ISO record tone was rated high on the first dimension. This suggests that it may not be suitable for its intended function, in which a response is required from the user within a designated time-out period (i.e. it is in the 'Prompt – Must' category), as it does not convey an impression of urgency. However, in practice this will depend on the context in which it is encountered: for instance a verbal prompt to record a message, immediately preceding the tone, would help to make the purpose clear.
- The UK and ETSI call waiting tones were also high on the first dimension. This seems appropriate given that they occur in the talk phase of a call as prompts for non-mandatory action and should therefore be designed not to be too intrusive.
- Both the pay tones tested (UK and Swedish) had scores well below neutral on the first dimension, appropriate to their function as prompts for urgent and necessary action.
- Both the UK and the ETSI ring tones scored positively on the first dimension – appropriately since they give feedback that the system status is OK. The other feedback tones had scores below the neutral point, appropriate to their function as indicating a 'not OK' condition. The UK special information and number unobtainable tones were found particularly unpleasant and disturbing.

In general the subjects in this experiment rated the UK tones higher on the first dimension (i.e. as more pleasant, less alarming, etc.) than the non-UK equivalents. This may be a result of greater familiarity; to test this hypothesis would require further research with subjects from other countries.

Broader questions include the following.

- To what extent are the perceptions of existing tones, as measured in this experiment, determined by the tones' inherent acoustic characteristics, and to what extent are they modified by users' past experience of the tones in use? This will have a bearing on predictions about users' perceptions of new tones when these are first introduced, and on the degree to which those perceptions can be expected to change with experience.
- Do the two dimensions identified here provide a *complete* characterisation of the salient aspects of users' perceptions of telephone network tones, or are there other dimensions which must be considered so as to design tones appropriate for particular functions?
- To what range of sounds is the two-dimensional characterisation appropriate? This experiment was confined to existing tones which are fairly simple in their acoustic structure (being combinations of not more than three pure tones, with regular cadences), but with modern equipment it might be feasible to use more complex synthesised or real-world sounds; would the same dimensions apply to these, and would any extra dimensions be required to characterise them?

All these questions suggest directions for further experimental work.

To bridge the gap between evaluating tones out of their intended specific contexts, as in this experiment, and experiencing them in context (i.e. at particular points in the process of setting up and continuing a call), comparative trials of tones in their intended contexts should ideally be conducted. However, experiments like the one reported here, which are relatively simple and quick to design and run, will still be appropriate in the early stages of designing a new tone, with evaluations in context perhaps following at a later stage once a promising tone (or a set of plausible candidate tones) for a specified function has been identified.

8 Acknowledgements

This work was carried out as part of the Dialogues 2000 Project, which was made possible by support from Engineering and Physical Sciences Research Council and from BT as part of the SALT LINK Programme. The authors acknowledge with thanks the assistance of their colleagues at CCIR, at BT Laboratories and at ETSI.

9 References

1 Anderson, D M. Feedback tones in public telephone networks: Human Factors work at ETSI. *Telektronikk,* 93 (3/4), 65–77, 1997.

2 ETSI. *Human Factors (HF) : European harmonization of network generated tones, Part 1: A review and recommendations.* Valbonne, 1997. (TR 101 041-1.)

3 Pollack, I. The information of elementary auditory displays. *Journal of the Acoustical Society of America,* 24, 745–749, 1952.

4 ETSI. *Recommendation of characteristics of telephone service tones when locally generated in telephony terminals.* Valbonne, 1995. (ETR 187.)

5 Oppenheim, A N. *Questionnaire design and attitude measurement.* London, Heinemann Educational Books Ltd, 1973.

Fergus McInnes (36) is a Research Fellow at the Centre for Communication Interface Research (CCIR) at the University of Edinburgh. He has 14 years' research experience, mainly in speech recognition and spoken dialogue systems. His current work is on fluent speech dialogues for automated telephone services and on usability issues in video-conferencing systems and virtual environments.

e-mail: Fergus.McInnes@ed.ac.uk

Donald M. Anderson is currently working as Research Ergonomist with Rikshospitalet (National Hospital), Oslo, and as the Principal Consultant, Man Machine Technology AS. He has 45 years experience in ergonomics and his interests include standards, maritime ergonomics, product design and development and aids for the handicapped.

e-mail: mantech@login.eunet.no

Mark Schmidt (32) is currently a manager in the Customer Relationship Management (CRM) practice at Andersen Consulting, specialising in the design and implementation of multi-channel customer contact centres. His specialist skills lie in the area of speech processing, voice processing and dialogue design for man-machine interfaces involving speech. Prior to joining Andersen Consulting in 1997, Mark Schmidt spent 8 years at the Centre for Communication Interface Research at Edinburgh Univ.

Mervyn Jack (49) is Professor of Electronic Systems at the University of Edinburgh. He leads a multi-disciplinary team of twenty researchers investigating human factors and usability engineering of automated telephone services. His main research interests are dialogue engineering and virutal reality systems design for advanced e-commerce and consumer applications.

e-mail: Mervyn.Jack@ed.ac.uk

Status

Introduction

PER HJALMAR LEHNE

Security is a key component in all information technology and telecommunications. Security functions take care of the user's privacy by preventing eavesdropping, and it also protects the network and service providers from unauthorised access to the systems. In telecommunications, security functions were heavily emphasised when specifying the pan-European mobile system GSM. Radio frequencies constitute a 'shared' transmission medium, which means that 'anyone' in principle can eavesdrop on or intrude into the communications channel with simple means. In GSM, this issue was taken seriously, so that advanced algorithms and functions were introduced for both authentication and encryption purposes.

One of the international organisations dealing with security issues is the *International Organization for Standardization – ISO*. In this issue of the Status section, Øyvind Eilertsen reports from the work performed in ISO/IEC[1)] Joint Technical Committee (JTC) 1, Subcommittee (SC) 27, "Security Techniques". The work in SC 27 covers all aspects of security standardisation.

The future mobile communications systems, or third generation mobile systems, as they are often called, are now soon reality. That means that the future we are talking about is the very near future. In Europe, the new generation of mobile communications is called *UMTS – Universal Mobile Telecommunications System*. The European Telecommunications Standards Institute – ETSI , with its Technical Committee (TC) Special Mobile Group (SMG), is responsible for the standardisation of UMTS. The introduction of UMTS is defined to be in two steps, or phases. Phase 1, which is based on the existing GSM core network, is planned for introduction and roll-out from 2001 – 2002. UMTS Phase 2 is planned to start in 2005 using a broadband core network, probably based on an evolvement of the Internet Protocol (IP).

The single most important step towards providing full multimedia access to personal, mobile terminals is the specification of a new radio interface capable of carrying 2 Mb/s services in dense areas. The new terrestrial radio interface for UMTS is called *UTRA,* which stands for *UMTS Terrestrial Radio Access*. In the second paper of this section, the basic concepts of UTRA are described, emphasising the physical layer. It is important not to forget that UMTS also incorporates a satellite component, for which a radio interface called USRA (UMTS Satellite Radio Access) is being discussed.

Per Hjalmar Lehne (40) is Research Scientist at Telenor Research & Development, Kjeller. He is working in the field of personal communications, with a special interest in antennas and radio wave propagation for land mobile communications.

e-mail: per.lehne@fou.telenor.no

[1)] *International Electrotechnical Committee.*

Security standardization in ISO

ØYVIND EILERTSEN

JTC 1

The International Organization for Standardization (ISO) and the International Electrotechnical Committee (IEC) have established a joint technical committee (JTC 1) to facilitate standardization in the field of information technology. In this context, information technology includes the "specification, design and development of systems and tools dealing with the capture, representation, processing, security, transfer, interchange, presentation, management, organization, storage and retrieval of information."

As of January 1999, JTC 1 has 19 subcommittees, and the work areas include terminology, user interfaces, audio/video coding, software engineering, and security.

SC 27

Subcommittee (SC) 27 *'Security Techniques'* takes care of all aspects of security standardization, including

- Identification of generic requirements for IT system security services;
- Development of security techniques and mechanisms;
- Development of security guidelines;
- Development of management support documentation and standards.

Specifically excluded is the actual embedding of security mechanisms in applications. The standardization of cryptographic algorithms for confidentiality services was excluded for some time, but is now a part of the work area.

A large number of security-related standards from a variety of standards organizations have already been developed. To avoid duplicating existing work, SC 27 has liaison with groups doing security work, including ECMA (TC 36), ETSI (TC security), ITU-T (SG7/Q20) and IEEE (P1363). In addition, SC 27 has liaison to other ISO committees, including TC 68 ('Banking, securities and other financial services').

Working groups

The activities of SC 27 are divided into three working groups or WGs.

Working Group 1: 'Requirements, security services and guidelines'

The terms of reference of Working Group 1 are

- Identification of application and system requirement components;
- Development of standards for security services (e.g. authentication, access control, integrity, confidentiality, management and audit) using techniques and mechanisms developed in WG 2;
- Development of supporting interpretative documents (e.g. security guidelines, glossaries, risk analysis).

Current work:

- Trusted Third Parties (TTPs): The technical report 'Guidelines for the use and management of TTPs' is currently under PDTR ballot. The target date for publishing is May 2000. 'Specification of TTP services to support the application of digital signatures' is still a Working Draft.
- Guidelines for the management of IT security (GMITS): Parts 4 and 5 are currently under PDTR ballot, and the target date for publishing is May 2000.
- Other work items in WG 1 still on the working draft level are 'Time stamping services and protocols', 'Security Information Objects' and 'IT intrusion detection framework'.

Working Group 2: 'Security techniques and mechanisms'

WG 2 terms of reference:

- Identify the need and requirements for security techniques and mechanisms in IT systems applications;
- Develop terminology, general models and standards for these techniques and mechanisms for use in security services.

The mechanisms in general include several options, including both cryptographic (symmetric and asymmetric) and non-cryptographic techniques.

Current work:

A number of standards are currently being revised, including

- IS 10118 'Hash functions' parts 1 and 2;
- IS 9797 'Message Authentication Codes' parts 1 and 2;
- IS 9798 'Entity authentication' parts 2 and 4;
- IS 9796 'Digital signature mechanism giving message recovery' part 1;
- IS 7064 'Check character systems'.

A new work item is 'Cryptographic techniques based on Elliptic Curves', parts 1 to 3, which is a really hot topic in the cryptographic community. All three parts are still Working Drafts with a publication target date of 2000/2001. A new working draft on random number generation and validation is still in a very early phase.

Working Group 3: 'Security evaluation criteria'

WG 3 terms of reference (excerpt):

- Develop standards for security evaluation and certification of IT systems, components and products. This will include consideration of computer networks, distributed systems, associated application services, etc. Three aspects may be distinguished:
  1 Evaluation criteria;
  2 Methodology for application of the criteria;
  3 Administrative procedures for evaluation, certification and accreditation schemes.

Current work:

WG 3 was initially established to develop the IT evaluation criteria. This project has been carried out in close co-operation with the Common Criteria project, whose main goal has been to align existing evaluation criteria from Canada, Europe and the US. The ISO/IEC standard will be technically aligned with the final version of the Common Criteria.

The WG 3 evaluation criteria project is now approaching its conclusion, as all three parts of the standard are undergoing Final DIS ballot. To facilitate the implementation and widespread use of the evaluation criteria, WG 3 has initiated a number of new projects, including 'Protection Profile registration', 'Framework for IT security assurance' and 'Guide for production of Protection Profiles and Security Targets'.

SC 27 schedule

SC 27 working group meetings are held every six months, usually in April and October. The 1999 meetings will be arranged in Madrid, Spain (April) and Columbia, MD, USA (October). The SC 27 meeting calendar, as well as the an overview of security terminology and the project catalogue can be downloaded from the Internet address

http://www.iso.ch:8080/jtc1/sc27/

Norwegian work in SC 27

Norway has been an active member of SC 27 since the creation of the committee in 1990. Svein J. Knapskog has been convenor of WG 3 from the start, and there has been consistent Norwegian representation in WG 2.

It has generally been difficult to find Norwegian companies willing to support participation in WG 1, but the recent work on TTPs has persuaded both Posten SDS and Telenor to send representatives to the working group meetings.

Norsk Teknologistandardisering (NTS) is the Norwegian member body for SC 27, and all membership issues are handled by NTS and the reference group K 171, which at present is headed by Øyvind Eilertsen of Telenor Research and Development. K 171 generally meets prior to the Working Group meetings to co-ordinate Norwegian voting and comments to documents, and participation is not limited. Interested parties are strongly encouraged to attend the K 171 meetings.

Abbreviations

CD	Committee Draft
DIS	Draft International Standard
DTR	Draft Technical Report
FCD	Final Committee Draft
FDIS	Final Draft International Standard
IEC	International Electrotechnical Committee
ISO	International Organization for Standardization
JTC	Joint Technical Committee
NTS	Norsk Teknologistandardisering
PDTR	Proposed Draft Technical Report
SC	Subcommittee
TR	Technical Report
TTP	Trusted Third Party
WD	Working Draft
WG	Working Group

Øyvind Eilertsen (32) has been employed as Research Scientist at Telenor R&D since 1992. He is attached to the security group and is currently working on a project related to GSM security. Interests include cryptographic algorithms and Internet security.

e-mail: Oyvind.Eilertsen@fou.telenor.no

UTRA – the radio interface for UMTS

PER HJALMAR LEHNE

Background

In January 1998, ETSI[1)] *Special Mobile Group – SMG*, held its 24th meeting in Paris. The main task for this meeting was to decide the new radio access method to be adopted for UMTS, namely *UTRA – UMTS Terrestrial Radio Access*. Before the ETSI decision, 5 candidates were competing for UTRA:

- α: Wideband Code Division Multiple Access (W-CDMA);
- β: Orthogonal Frequency Division Multiple Access (OFDMA) with slow frequency hopping (SFH);
- δ: Time Division / Code Division Multiple Access (TD/CDMA);
- γ: Wideband Time Division Multiple Access (WB-TDMA);
- ε: Opportunity Driven Multiple Access (ODMA).

ETSI SMG's decision was a compromise:

- In the paired frequency band the α-concept (W-CDMA) was chosen.
- In the non-paired frequency band the δ-concept (TD/CDMA) was chosen together with elements from the ε-concept (ODMA).
- The concepts should be further developed to be harmonised for best interoperability.

[1)] *European Telecommunications Standards Institute.*

Table 1 Main parameters of the new UTRA concept [1]

	Mode:	
	FDD	**TDD**
Access method	Direct Sequence (DS)-CDMA	
User bit rates	up to 2.048 Mb/s	
Chip rate	4.096 Mcps, extendible to 8.192 and 16.384 Mcps	
Channel separation	5 MHz, adjustable with a channel raster of 200 kHz The carrier frequency must be a multiple of 200 kHz	
Duplex distance	Variable (TBD)	n/a
Frequency bands	1 920 – 1 980 MHz (UL) 2 110 – 2 170 MHz (DL)	any
Frame length	10 ms	
Spreading codes	Orthogonal Variable Spreading Factor – OVSF Extended Very Large Kasami Orthogonal Gold codes	
Channel coding	Convolutional codes, Turbo codes, RS codes, interleaving	

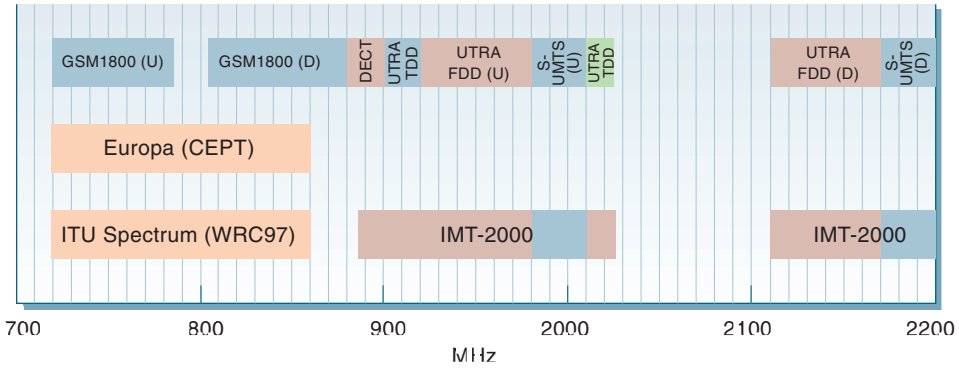


Figure 1 Frequencies for IMT-2000 and UMTS

UTRA in a 'nutshell'

After the decision, the Physical Layer Expert Group of SMG2 has worked intensively to develop and harmonise the concepts. In Table 1 the main parameters of UTRA are listed [1]. These seem to be fairly stable. Most of the parameters and modes are described in this article.

Frequencies

The ITU World Radio Conference in 1997 – WRC '97 – allocated the frequency bands for the third generation mobile system – IMT-2000. The European counterpart – UMTS – was allocated nearly the same bands from the European Radio Office – ERO. Figure 1 shows these frequency bands.

Duplex methods

The duplex method is the technique used for separating the two transmission directions, denoted uplink and downlink. The UTRA concept contains two different duplex modes to be used in different scenarios:

- *FDD – Frequency Division Duplex:* Up- and downlink are separated in the frequency domain. Different frequencies are used for each direction.
- *TDD – Time Division Duplex:* Up- and downlink are separated in the time domain. The same frequency is used, but the two directions use different time slots.

CDMA – Code Division Multiple Access

A novel method for channel access and resource sharing is used in UTRA. This is Code Division Multiple Access – CDMA. In UTRA, a technique called Direct Sequence CDMA (DS-CDMA) is employed. The principle is shown in Figure 2. More comprehensive descriptions of the CDMA principle are given by [2] and [3].

The coded information bit stream is multiplied with a *spreading code* before it is fed to the RF-modulator. The smallest unit of the spreading code is called 'chip'. The chip rate is much higher than the information bit rate. For UTRA it is up to 64 times higher.

CDMA is thus used to allocate several connections and users to the same time- and frequency-domain resource. In UTRA different spreading codes are used for different purposes:

- *Channelisation code* – allocated to the connection;
- *Scrambling codes* – allocated to the cell;
- *Random access preamble spreading code;*
- *Synchronisation code.*

Table 2 lists the spreading codes chosen.

Table 2 Spreading codes used in UTRA

Channelization code	Orthogonal Variable Spreading factor – OVSF
Scrambling codes	Short Scrambling Code: Extended Very Large Kasami set (256) Long Scrambling Code: 40960 (10 ms) segment of a 2^{41} - 1 Gold code
Random access preamble spreading code	256/128 chip orthogonal Gold code
Synchronisation code	256/128 chip orthogonal Gold code

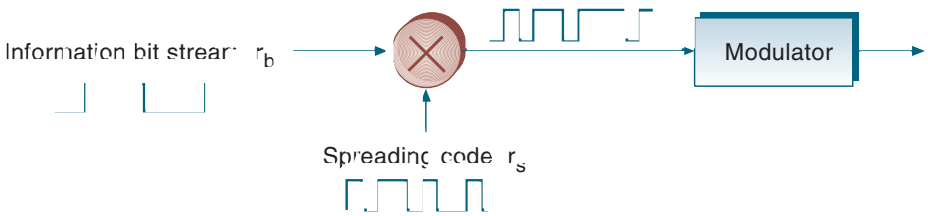


Figure 2 Principle for Direct Sequence CDMA

Radio Protocol Architecture

Channel hierarchy

The radio protocol is organised in so-called 'channels' on three levels on layers 1 and 2. From top to bottom these are: *Logical channels, transport channels* and *physical channels*.

The logical channels handle the layer 2 Medium Access Control (MAC) and Radio Link Control (RLC), as well as control and user information. The transport channels constitute the interface between layer 1 and layer 2 MAC, both for signalling and data.

The physical channels are the physical bearers for the transport channels. They describe the physical resources, like *carrier frequency, code* and (uplink only) relative *phase* (0 or $\pi/2$); in TDD mode also *time slot*. In this context, it is necessary to define the difference between the *Mobile Station – MS*, and the *User Equipment – UE*. The MS (or ME – Mobile Equipment) is the radio transceiver equipment. The MS together with a UMTS Subscriber Identity Module (USIM) is called a UE. A UE can have several USIMs.

The different logical, transport and physical channel types defined in UTRA are listed in Tables 3, 4 and 5.

Table 3 Logical channel types in UTRA

Control Channels (CCH):	**Control plane information only**
Synchronisation Control Channel (SCCH)	A downlink channel used for broadcasting synchronisation information (cell ID, optional information) in case of TDD operation.
Broadcast Control Channel (BCCH)	A downlink channel for broadcasting system control information.
Paging Control Channel (PCCH)	A downlink channel that transfers paging information. It is used when the network does not know the location cell of the UE.
Dedicated Control Channel (DCCH)	A point-to-point bi-directional channel that transmits dedicated control information between a UE and the network.
Common Control Channel (CCCH)	Bi-directional channel for transmitting control information between network and UEs.
Traffic Channels (TCH):	**User plane information only**
Dedicated Traffic Channel (DTCH)	A point-to-point channel dedicated to one UE, for the transfer of user information.

Table 4 Transport channel types in UTRA

Common Channels:	
Broadcast Control Channel – BCCH	A downlink channel used to broadcast system- and cell-specific information over the entire cell.
Forward Access Channel – FACH	A downlink channel used to carry control information to an MS when the location cell is known. Transmitted over the entire cell or over only a part of the cell using lobe-forming antennas.
Downlink Shared Channel – DSCH	A downlink channel used to carry control information to MSs. Can be standalone or associated with a DCH. Transmitted over the entire cell or over only a part of the cell using lobe-forming antennas.
Paging Channel – PCH	A downlink channel used to carry control information to a mobile station when the location cell is unknown. Transmitted over the entire cell.
Random Access Channel – RACH	An uplink channel used to carry control information from a mobile station. Always received from the entire cell.
Dedicated Channels:	
Dedicated Channel – DCH	A downlink or uplink transport channel used to carry user or control information between the network and a mobile station.

Table 5 Physical channel types in UTRA

Common Physical Channels:	
Primary Common Control Physical Channel – CCPCH	Downlink physical channel used to carry the BCCH.
Secondary Common Control Physical Channel – CCPCH	Downlink physical channel used to carry the FACH and PCH.
Synchronisation Channel – SCH	A downlink signal used for cell search. Consists of two sub-channels, the Primary and Secondary SCH.
Physical Random Access Channel – PRACH	Uplink physical channel used to carry the RACH. Based on a slotted ALOHA scheme.
Dedicated Physical Channels:	
Dedicated Physical Channel – DPCH	Downlink physical channel used to carry dedicated data generated at Layer 2 and above. It is time-multiplexed with control information generated at Layer 1.
Dedicated Physical Data Channel – DPDCH	Uplink physical channel used to carry dedicated data generated at Layer 2 and above, i.e. the DCH. Each Layer 1 connection may have zero, one or several uplink DPDCHs.
Dedicated Physical Control Channel – DPCCH	Uplink physical channel used to carry control information generated at Layer 1.

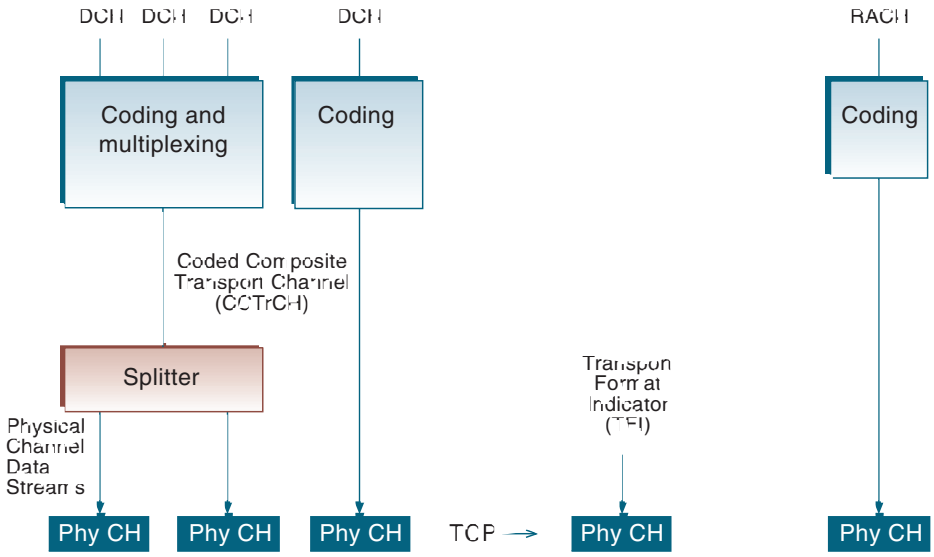


Figure 3 Model of the MSs physical layer – uplink

Physical Layer Model

The physical layer model, with an example of the uplink as seen from the MS is shown in Figure 3.

Frame Structure

The basic unit in the frame structure of UTRA is called a *frame* of 10 ms duration. It consists of 16 *slots,* each 625 µs. Each slot consists of a variable number of bits, depending on the service carried. By using rate matching and OVSF codes the bits are coded into a chip rate of 4.096 Mchips/s, giving 2 560 chips/slot.

UTRA FDD also defines a *super frame* of 720 ms, consisting of 72 frames. The frame structure is shown in Figure 4. An example of how the slot is organised for the dedicated physical channel (DPCH) is shown in Figure 5.

Random Access – FDD mode

The random access is based on a slotted ALOHA scheme as shown in Figure 6. Each frame period is organised into 8 access slots, each offset by 1.25 ms. The access burst is of 11.25 ms duration, consisting of a 1 ms preamble and a 10 ms data part as shown in Figure 7.

Asymmetric Duplex in TDD mode

Multimedia communications, like Internet browsing, has a very asymmetric nature, i.e. the effective bit rate is much higher in one direction than in the other. Conventional FDD networks typically allocate equal bandwidths in both directions, and are thus far from optimal in resource efficiency. In UTRA, this aspect has been taken into account, especially in the TDD part. The asymmetric duplex method specified has a high degree of flexibility in the asymmetry rate, ranging from 1:1 to 1:15. Some examples are shown in Figure 8.

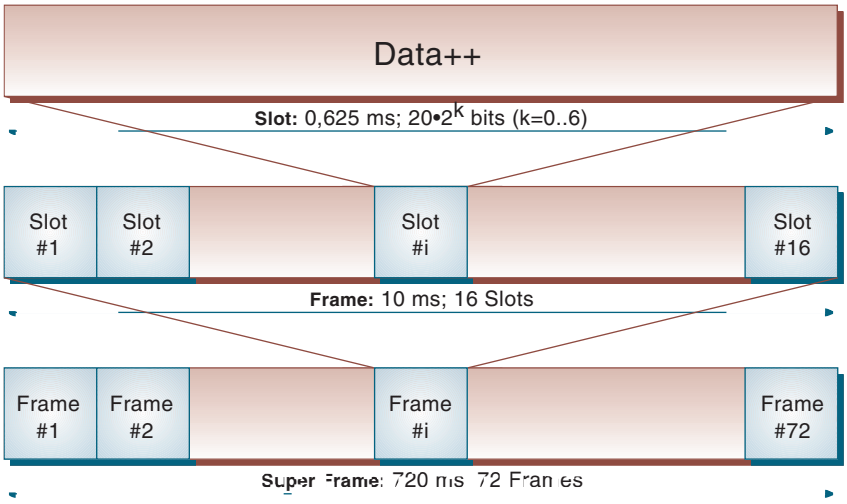


Figure 4 Frame hierarchy for UTRA

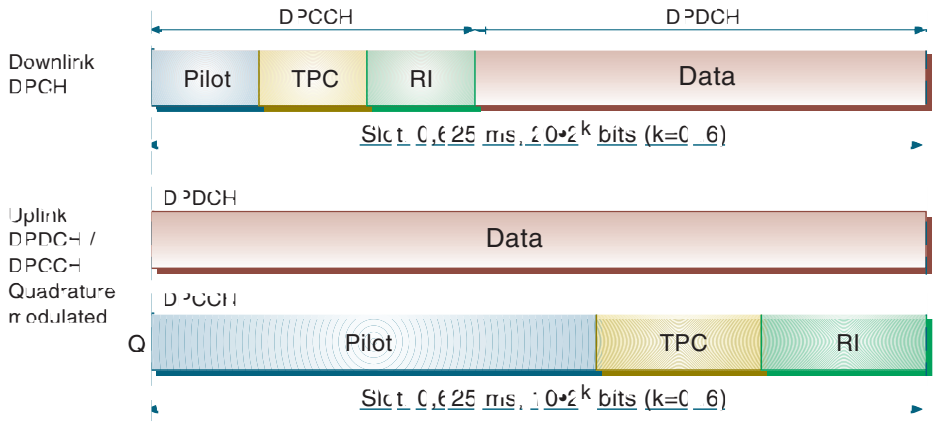


Figure 5 Frame structure for the dedicated physical channel in UTRA FDD

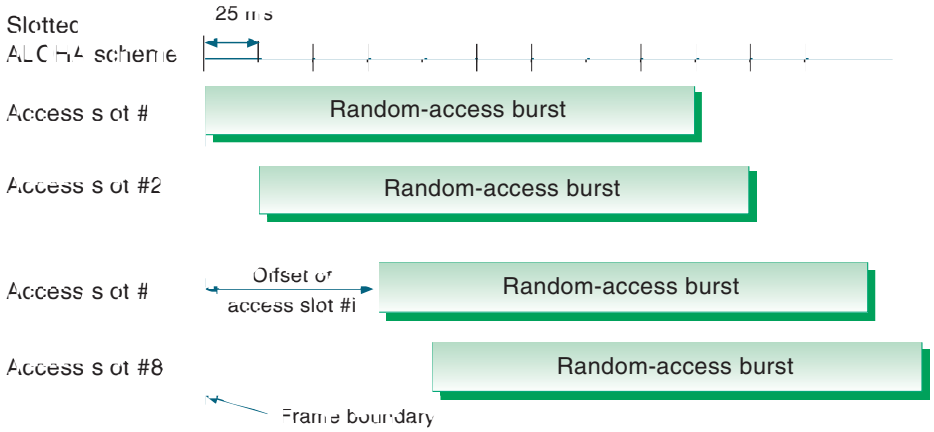


Figure 6 Slotted ALOHA scheme for UTRA.FDD

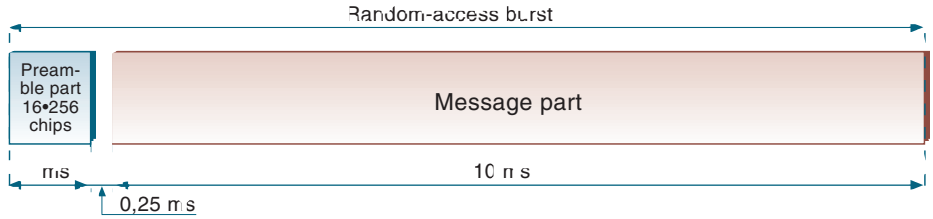


Figure 7 Random-access burst

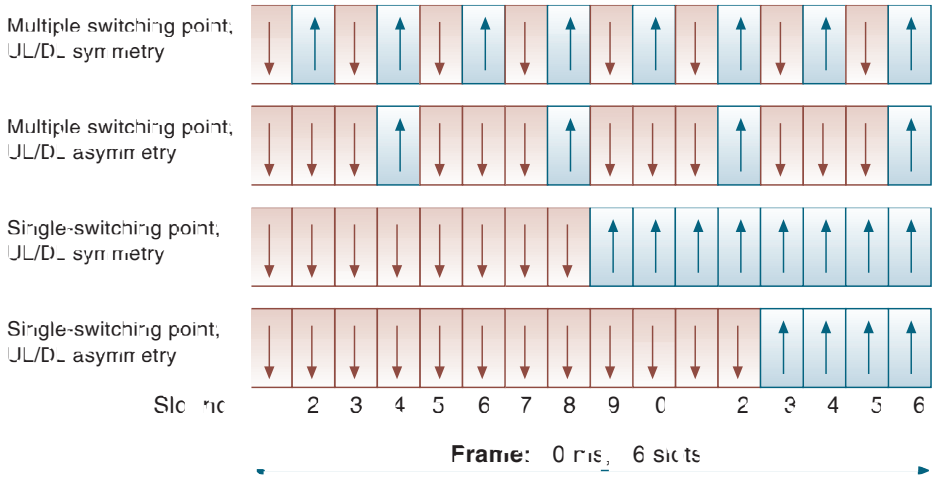


Figure 8 Asymmetric duplex in UTRA TDD

Frame Structure – TDD mode

In TDD mode, the basic frame structure is equal to FDD, however a different slot structure is used. A mid amble of 256 or 512 chips is introduced, and the slots are separated by a 96 chip guard period. The transmission within a slot period in UTRA TDD is called a 'traffic burst'. Two different bursts are defined as shown in Figure 9.

The transmission of control channels in TDD is also different from FDD. In Figure 10, the organising of a frame is shown, with the structure of the access burst, which clearly is different from the FDD access burst.

Channel Coding and Multiplexing

In general, the channel coding in UTRA is performed as shown in Figure 11. The coding and multiplexing are done on the transport channel level. Channel coding is performed in several stages, and the coding is different according to the different service requirements. This is shown for the different options of UTRA FDD in Figure 12. The coding schemes for TDD is similar.

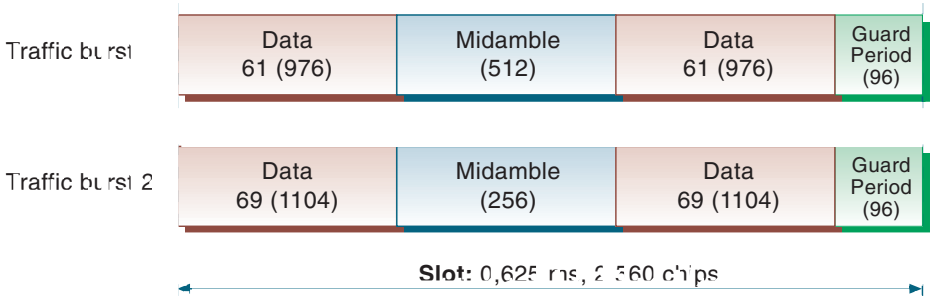


Figure 9 Traffic burst structure in UTRA TDD

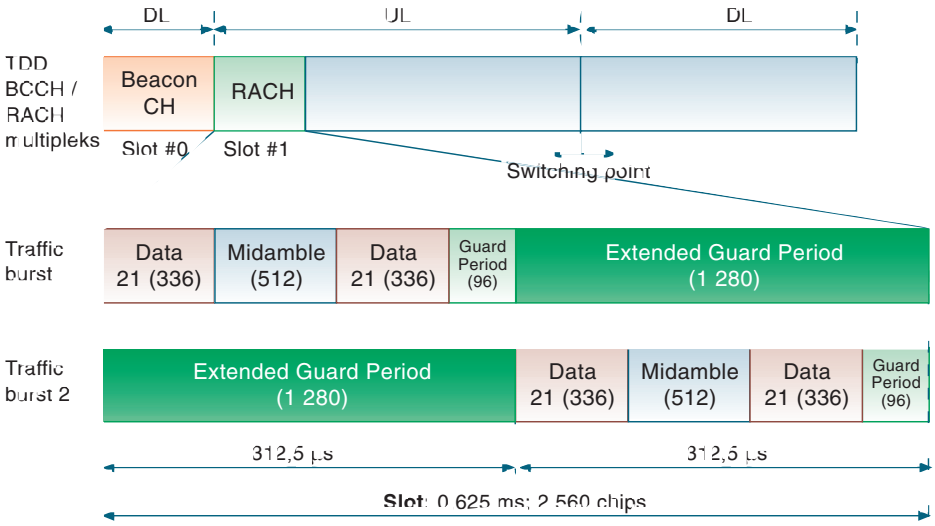


Figure 10 Control channel frame structure in UTRA TDD

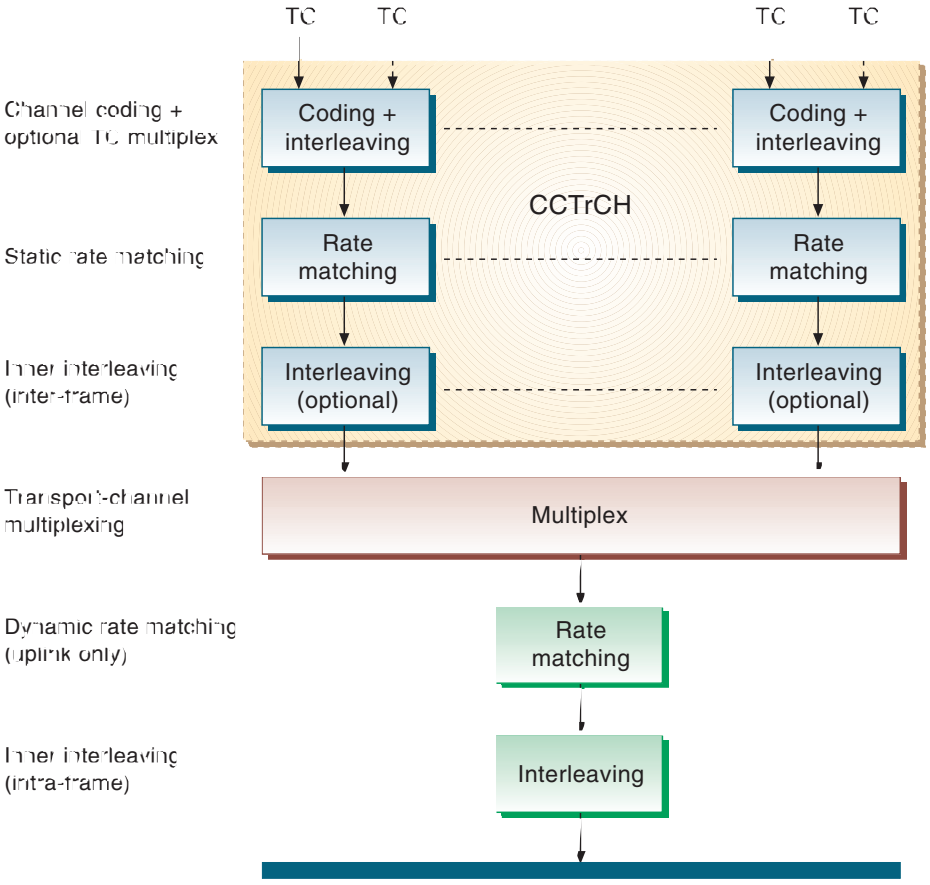


Figure 11 Coding and multiplexing of transport channels

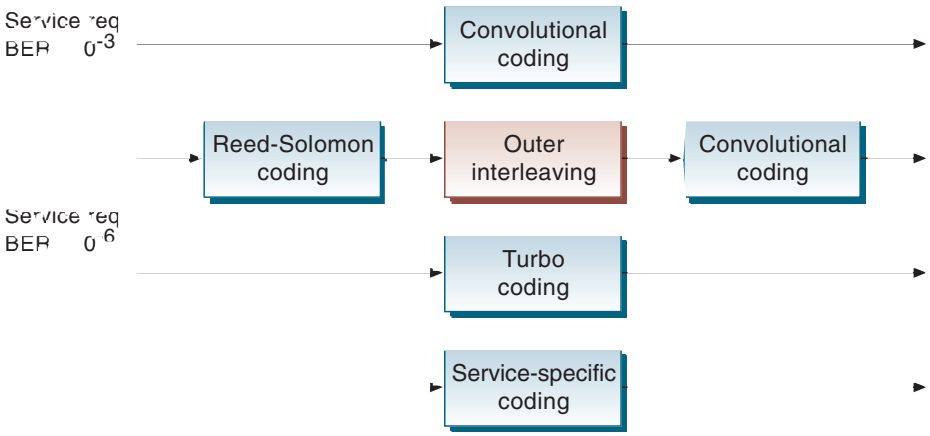


Figure 12 Channel coding for UTRA FDD

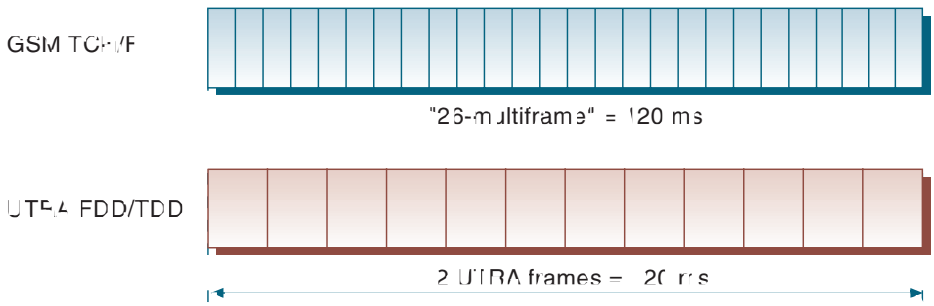


Figure 13 UTRA – GSM interoperability on the radio protocol level

The different codes used are:

Inner coding and interleaving of layer 2 data:

- Convolutional code, code rate 1/2 or 1/3;
- Low-delay services: intra-frame (10 ms) interleaving;
- Higher allowable delay: inter-frame (15 frames, 150 ms).

Outer coding and interleaving of layer 2 data:

- Rate 4/5 Reed-Solomon code over GF(28);
- Symbol wise, inter-frame block interleaving.

Turbo coding:

- Rate 1/2 or 1/3.

Rate matching

Rate matching is a technique used to adapt the varying offered bit rates of the transport channels to the available bit rate of the uplink and downlink dedicated channel.

Two types of rate matching are implemented in UTRA:

- *Static rate matching* is carried out on a slow basis every time a transport channel is added to or removed from the connection. It is used to adjust the coded transport channel bit rate in order to fulfil the minimum transmission quality and to match the total bit rate after multiplexing to the channel bit rate of the up- and downlink DPDCH.
- *Dynamic rate matching* (uplink only) is carried out on a frame-to-frame (10 ms) basis. It is used to match the total instantaneous rate of the multiplexed transport channels to the channel bit rate of the uplink DPDCH.

Modulation

The modulation method specified for UTRA is Quadrature Phase Shift Keying – QPSK, with pulse shaping: The pulse shaping filter specified is a *root raised cosine* (RRC) with roll-off $\alpha = 0.22$ in the frequency domain.

UTRA – GSM interoperability

One of the goals for UTRA has been the ability to inter-operate with the existing GSM access method. When talking about inter-operability, not only co-existence has been required, but it should also be possible to roam and perform handover between networks operating on the two access techniques. The handover requirement necessitates a common entity on the radio frame structure. This is shown in Figure 13. A so-called 26-multiframe in GSM is used to carry the full rate traffic channel, TCH/F. This can be synchronised to a UTRA multiframe of 12 frames, which enables multimode mobile stations to monitor both networks in a structured manner.

Power Control

Power Control (PC) is crucial for a CDMA network to function properly due to the so-called near-far problem. Three types of power control are implemented in UTRA:

The *closed loop PC* (both up- and downlink) is used to adjust the MS output power to meet uplink SIR target. It is performed on time slot level (0.625 ms). It has a variable step size of 0.25 – 1.5 dB, with a dynamic range of 80 dB on uplink and 30 dB on downlink.

The SIR target adjustment (both up- and downlink) is performed by the *outer loop PC*, and based on estimated connection quality. An *open loop PC* also exists (uplink only) in order to adjust the output power on the physical random access channel (PRACH).

Handover

Handover (HO) has been mentioned earlier when discussing the UTRA – GSM interoperability. A total of three different handover concepts are defined in UTRA:

The principle of *soft handover* is shown in Figure 14. This is the really novel technique in UTRA compared to GSM. In normal traffic, the mobile station communicates with more than one base station. This set of BSs is called an *active set* of base stations. This technique can be used within one network.

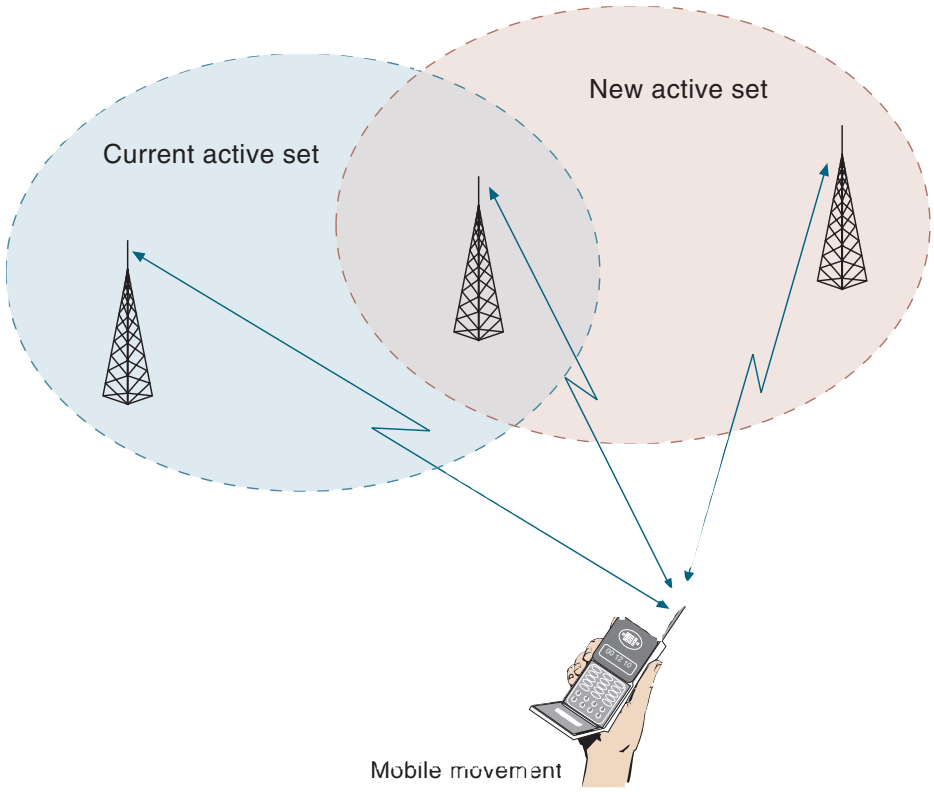


Figure 14 Soft handover (HO) in UTRA

Additionally, handover can be performed between cells to which different number carriers have been allocated, between cells of different overlapping layers using different carrier frequencies, and between different UTRA operators/systems using different carrier frequencies. Then, so-called *UTRA to UTRA hard handover* is used.

And, finally, as mentioned earlier, *hard handover from UTRA to GSM* is possible.

Additional features and options

A lot of things can be said about the new radio interface for UMTS, and some features and options are mentioned here.

Adaptive antennas

Adaptive or 'smart' antennas are spatio-temporal filters that can be implemented at the base station. This makes it possible to reduce interference, extend the range, or in the most sophisticated technique, introduce an additional dimension in the access space; so-called *Space Division Multiple Access – SDMA*. Describing adaptive antenna techniques is beyond the scope of this article. A good overview can be found in [4].

Downlink transmit diversity

Transmitter diversity is a means to significantly improve capacity and coverage on both FDD and TDD, without the need of a second receiver chain in the MS as conventional receiver diversity implies. Transmit diversity means sending on two (or more) antennas at the base station in order to introduce diversity in the received signal. In many ways, this is the reciprocal method to conventional receiver diversity. In UTRA, two different techniques are defined:

Orthogonal Transmit Diversity (OTD) utilises *code division transmit diversity*. In this method the coded bits are split into two data streams to be transmitted via two separate antennas. By using different orthogonal channelisation codes on the two antennas, the orthogonality between the two output streams is maintained. A small additional processing is required at the mobile station.

Time Switched Transmission Diversity (TSTD) and *Selection Transmit Diversity* (STD) are both so-called *time division transmit diversity* methods. This means that the signal is switched between the antennas in one of two ways. In TSTD the signal is switched according to a pattern decided by the base station. In STD the switching is dependent on signalling received by the mobile station.

MS – MS transmission; relaying

The UTRA TDD design is sufficiently flexible to support both simple relaying and advanced relaying protocols such as *Opportunity Driven Multiple Access (ODMA)*. Relaying is a widely used technique for packet radio data transmission in both commercial and military systems. The potential of relaying is among other things to improve coverage by reduced effective path loss and to increase capacity by lower transmission power and associated inter-cell interference. Relaying means that MS – MS transmission must be enabled. This is possible within the UTRA TDD frame structure.

Final comments

The new radio access technique for UMTS, namely UTRA as defined by ETSI, has inherent great potential and flexibility to support the needs for future mobile multimedia services. Simple trials are already taking place, and it is believed that early commercial deployments will commence already in 2001 – 2002.

References

1 ETSI. *The ETSI UMTS terrestrial Radio Access (UTRA) ITU-R RTT Candidate Submission.* Attachment 2 – Updated System Description. September 1998.

2 Eriksen, J, Svebak, O D. Code Division Multiple Access – hot topic in mobile communications. *Telektronikk,* 91 (4), 99–108, 1995.

3 Prasad, R, Ojanperä, T. An overview of CDMA Evolution toward Wideband CDMA. *IEEE Communications Survey. Fourth Quarter 1998.* 25.02.99. [online]. URL: http://www.comsoc.org/pubs/surveys/4q98issue/prasad.html

4 Pettersen, M, Lehne, P H. Smart antennas – the answer to the demand for higher spectrum efficiency in personal communications systems. *Telektronikk,* 94 (2), 54–64, 1998.

Per Hjalmar Lehne (40) is Research Scientist at Telenor Research & Development, Kjeller. He is working in the field of personal communications, with a special interest in antennas and radio wave propagation for land mobile communications.

e-mail: per.lehne@fou.telenor.no