A Software Measurement Case Study using GQM

Björn Lindström

Supervisors
Per Runeson, Lund Institute of Technology
Achim Kämmler, HP OpenView Amsterdam

Abstract

Software measurement is about measuring attributes of the process, product, and resources in software projects, and the overall objective for it is to get information about the work to be able to improve and make it predictable. Software measurement is far from standard procedure in the software industry, mainly due to lack of immediate results. HP OpenView in Amsterdam and the team Special Products Engineering in is one example of this when they have not used software measurement in their previous projects. Hence the manager saw a need to evaluate the completed projects with measurement.

Measurement always implies some effort and it is therefore important to know why you measure, for what purpose, and for whom. The Goal Question Metric (GQM) approach is a way to find out why and what to measure and this approach is used. The work presented in this report is based on existing information only, which tightly limits the possible measurements. The conclusion of using GQM is that it is not suitable when the existing data is very limited when it results in too many questions and metrics that are not possible to answer and measure.

The main conclusion of these measurements is that the way the team has been working is not sufficient for reliable measurement. Most measures are not possible to validate and one should therefore be careful to draw general conclusions from the result. Still the results indicate that the organisation is immature and the productivity low.

Table of contents

1 Introduction

1.1 Background

Hewlett-Packard (HP Invent) is a technology solutions provider to consumers, businesses and institutions globally. The company's offerings span for example IT infrastructure, personal computing, imaging and printing for consumers, enterprises and small and medium businesses. HP has about 142 000 employees and offices all over the world, and among them one in Amsterdam, The Netherlands. [1]

At the office in Amsterdam the main work is regarding a software product called Service Desk. The product is mainly used to manage organisations IT equipment, and the purpose is to have a structured way to handle problems, or with other words deliver a service. Service Desk is developed at multiple sites, and in Amsterdam three teams are developing it. Current Products Engineering is maintaining the existing version, Future Products Engineering is developing components for a future version, and Special Products Engineering is developing and maintaining special versions for customers. Most data in this report is from the team Special Products Engineering, which also is called SPE. The SPE team has gone through a rapid expansion, from three people one and a half year ago, they now count to fourteen.

In traditional engineering measurement play a central role to ensure and improve product and process quality. In software engineering this is not yet the case, because there are still companies where measurement has very little or none importance. There are differences that make it considerably harder to measure in software engineering, but this is not the only reason. A reason closer to the truth is that software engineers and managers expect too much from measurements. They expect measurements to be precise and objective, and to answer far too many questions about the product and organisation. Misperceptions of measurement in software engineering and lack of immediate results have resulted in that many measurement programs are abandoned too early [2].

1.2 Problem

A manager needs information about projects to be able to make decisions, plan and schedule, and allocate resources for the different activities. Sources of information are documents produced during the development and direct contact with the developers. However these sources are not sufficient and the manager must rely on experience and estimations. It would be better to know instead of estimating, but when this is not possible the approach has to be to make as good estimations as possible. To be able to make good estimations the manager needs to have in-depth information about the organisation and the staff. Also there is a need for validation of the estimations. The underlying problem for a manager is that it is very difficult to control something that one has little knowledge about.

1.3 Objective

The first objective of this thesis is to find out how software measurements can help a software development team and its manager to get a better understanding of their work performance. The second objective is to see how and if software measurement can provide a manager with information that results in better estimations.

1.4 Scope

The scope of this report is that it includes measures from seven completed projects, all done by the team Special Projects Engineering. It also includes two projects that were completed before the SPE team existed. The completed projects vary in size from two to about nine

people involved, and in duration from about three to eight months. None of the projects developed anything from scratch, but they were built upon a large code base. The SPE team has not implemented a measurement program so the data that exist come from documents like specification, requirements, project schedule, etc. It is also possible to track defects to some extent.

1.5 Outline

Chapters two to four cover the theory relevant for this report. In chapter two the concepts of *Software engineering* are introduced, which among others include software processes, project management, and software quality. The third chapter, *Measurement theory*, covers for example scales, direct and indirect measures, and validation. The fourth chapter is *Software metrics* and it holds information about how to decide what to measure, different type of measures, and practical issues to think about when measuring.

Chapter five is called *State of practise at HP OpenView in Amsterdam* and it briefly describes the team's process, metrics, development, and tools. In chapter six, *GQM analysis*, the Goal Question Metric approach is applied and iterated to find out what to measure. The following chapter, *Gap analysis*, describes the characteristics of the projects to be measured, followed by a presentation of the existing information. Then the gap between the existing and desired metrics according to the GQM analysis is presented.

In chapter eight, *Results*, the results are presented together with comments from the Special Projects Engineering manager. *Conclusion* is the last chapter and it contains the findings of this work.

2 Software engineering

Early experiences of building computer systems made it clear that an unstructured approach is not sufficient; many early projects went far over budget, and some were years late [3]. In the aftermath of this, software engineering evolved. IEEE (Institute of Electrical and Electronics Engineers) defines software engineering as:

"(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering of software. (2) The study of approaches as in (1)." [4]

An informal definition of software engineering is: a structured approach to efficiently build and evolve large and complex computer systems [3]. In practise software engineering includes everything in the production of software. Software engineering is related to computer science but computer science is instead concerned with theories and methods for computer systems. The areas are closely related which makes it is important for a software engineer to have some knowledge of computer science, and vice versa.

The overall goal with software engineering is to produce software with the right quality at reasonable costs. The development costs are depending on what approach the team is using, but roughly the costs can be divided into: specification, design, implementation, and integration and testing. In general, integration and testing is the biggest part and sometimes it can be close to 50 percent of the development costs [3]. A software product with poor quality will result in many hours spent on testing, which mean high costs. Poor quality will also make the product hard to sell, or generate unsatisfied customers. What high quality is differs from product to product, but in general it means that software that is maintainable, dependable, efficient, and usable.

The term software is not only the computer program but also all documentation and configuration files that the program needs to function correctly [3].

2.1 System engineering

Software engineering is not the same as system engineering, but a subset of it. System engineering is concerned with systems as a whole. For example, a computer-based system consists of hardware, software, and in most cases a user interface. This kind of system is more complex than a single computer-based system because they consist of and require input from multiple engineering disciplines. This report only covers computer-based systems, which also is referred to when mentioning system.

2.2 Software processes

To develop software products efficiently it is important to have a structured approach. A software process is a structured approach that describes the different activities that will lead to a developed product. Software processes are complex and no two projects are completely the same hence there is not one process that is applicable in all cases. Many organisations use tailoring (modifying process elements and changing the workflow) to develop organisation and project specific processes. It is not uncommon for a project to use different processes for different components of a product [5]. When developing large systems it is often not enough with one, so a hybrid has to be used. There are a number of generic process models for example the waterfall model, evolutionary development, formal systems development, and re-use development.

The fundamental activities are the same in all processes and they are: *specification*, *design and implementation*, *validation*, and *evolution* [3]. The *specification* of the software is critical for the further development, because a mistake here will lead to difficulties in the design and implementation. The specification of the software should define its functionality and constraints. This activity is also known as requirements engineering.

The *design and implementation* activity is to design and program according to the specification, and it will result in an executable system. If the development process is evolutionary, the specification may also be changed. During the design, the designers decide the structure of the software, the interfaces, the components, and sometimes also the data structures and algorithms. The later part of the design is interleaved with the implementation, and that is why design and implementation is stated as one activity. Some software projects put little effort on design, and instead start to implement almost immediately. This approach is not to recommend, because the lack of structure may create a software that is hard to maintain. There are no general implementation guidelines to follow, but all programmers develop their own style. The programmers do not only program, but also some testing and debugging. Testing is to discover failures, and debugging is to find and correct the place in the code that caused it. [3]

Software *validation*, which is also known as verification and validation (V & V), is an activity to make sure that the system meets the specification and the expectations from the customer. After the implementation, different modules of the system work independently, and the next step is to test the modules together. After this test it is time to test the whole system. The system test includes to validate the functional- and non-functional requirements, and to test the most important properties. The final step in the validation process is the acceptance test. This means to test the system with data from the customer instead of simulated data. The acceptance test will reveal whether it meets the requirements, and if the performance is acceptable.

The *evolution* activity starts when the software is delivered to the customer, and this is also called the maintenance of the software. Software maintenance is to change the delivered system during its lifetime. The changes can be to correct coding errors, design errors, specification errors, but also to handle new requirements. Maintenance is often more costly than the development, and can continue for years after the development finished. The importance of evolution has increased with the number of legacy systems, which an old system that is essential for everyday business. The problem with legacy systems is that after years of maintenance there exists most likely no specification that is accurate, which would make it hard to specify a new system with the exact same functionality. A new system is a risk in itself, and also the whole business process might need to be changed if the old one was intervened with the legacy system.

2.3 Requirements

The requirements for a system are descriptions of its services and constraints. Sommerville [3] distinguishes between three different description levels: *user requirements*, *system requirements*, and *software design specification*. Depending on what a requirement describe, it is classified as functional or non-functional. A *functional requirement* describes the function or service of the system. A *non-functional requirement* is not directly connected to the functionality of the system, but can be such as reliability and response time.

User requirements are descriptions of the functional and non-functional requirements for the system. They should be understandable for a system user without technical knowledge, and must therefore be written in a natural language. To write in a natural language can cause

problems, because it is hard to write so precise that it cannot be misinterpreted. To avoid misunderstandings it is advisable to use some kind of rules of how to write and express the requirements.

The *system requirements* are specifications of the whole system, and they are a more detailed description of the user requirements. System requirements are many times written in a natural language but they do not need to be understandable for a system user. The software engineers often use the system requirements as a starting point for the design, but they should still not include implementation information.

The *software design specification*, or software requirements specification, is the official document of the description of the system. It should include both the user and the system requirements, either as separate or as merged into one description. Both software developers and customers use the software requirements specification so it should be organized with that in mind.

2.4 Software architecture

The first design of a system is to establish the structure, identify the sub-systems and their communications. This activity is called architectural design and it results in a description of the software architecture. An advantage of explicitly designing software architecture is that it is a high-level representation of the system and is useful when discussing with stakeholders. The architecture also allows early analysis of the system, and this is important because the architecture affect requirements such as performance and reliability. Another advantage is that the architecture supports large-scale reuse when it can be transferred to other systems with similar requirements.

There are different styles and models for architectures, but most large systems do not rely on one model. Different parts of a system might have different key issues that need a special design. Performance, security, safety, availability, and maintainability are non-functional requirements that affect which style to use. It is possible that some of these requirements cause design conflicts, for example performance is improved with large-grained components but large grained components make the system harder to maintain.

2.5 Verification and Validation (V & V)

Verification and validation are most times used in the same context, but it is important to remember that they have a different meaning. Boehm [6] makes the following definition:

- “Validation: Are we building the right product?”
- “Verification: Are we building the product right?”

With other words verification is to make sure that the product meets its specified functional and non-functional requirements. Validation is to make sure that the product is functioning the way that the customer wants. The objective with verification and validation is not to make the system completely defect free, but to make it good enough for its intended use.

Verification and validation is a continuing process throughout the development. Software inspection and software test are the two methods used to verify and validate the software during the development. Software inspection does not require an executable program and can therefore be used throughout the whole development. Software testing does on the other hand require an executable program and can only be used in the later stages.

Software inspection is the activity of looking for defects in for example requirements specification, design documents, or source code. The advantages of software inspection are that many defects may be discovered in one session, and that it is cheaper than testing [3]. The people who are performing the inspection also get familiar with what kind of defects to expect and thereby also learn something.

Even though inspection has advantages it cannot replace testing. Non-functional requirements such as performance and reliability have to be tested when the system runs. A disadvantage with testing is that you normally only find one fault at a time. After a fault has been corrected the whole system has to be tested again, so testing is both time-consuming and expensive. Testing is despite this something that is inevitable in all software development.

At an abstract view testing consist of component (unit) testing and integration testing. Component testing is concerned with testing on functions, or groups of methods as an object or a method. Integration testing is to test how these components function together as sub-modules or a whole system. The developers are normally responsible for the testing of the components, but a special team has the responsibility for the integration testing. At testing of object-oriented systems the boundary between component testing and integration testing is not clear and other approaches can be necessary.

2.6 Software project management

Project management in software engineering is different from project management in other engineering disciplines. One difference is that the progress, the developed product, is not visible. The manager can probably see that the team has produced a certain number of lines of code, but that does not tell much about the actual progress when the final size is unknown. Another difference is that the software engineering process is not standardized, and it is up to management to decide which process to use. Due to the fast technological evolution it is possible that experience from one project is not transferable to another, and that is more likely to occur in software engineering than in more traditional engineering disciplines. [3]

The tasks a software manager complete depend on both the organisation and what kind of a project it is. There are some typical tasks that appear in most projects and they are: proposal writing, planning and scheduling, project costing, report writing and presentations, personnel selection and evaluation. Not all persons are suitable to become a manager, because it requires a person with very good communication skills, not just oral but also written.

A typical task for a manager is to make the plan for the project. The planning is an iterative process that is followed throughout the work. During the planning the milestones and deliverables are set. The milestones indicate when activities should be finished, and the deliverables are when the customer should get a delivery. When the manager is scheduling the project he or she decides who is doing what and for how long. The manager also has to consider the risks that exist, and have a plan if they occur.

2.6.1 Project estimation

Project estimation is carried out together with project scheduling and is a task for the project manager. The manager has to estimate how much the project will cost and how long it will take to develop the software. Software cost estimation is used to set up a budget for the project, and also to give a price to the customer. How the price of the software relates to the actual cost of the development is complicated, but some dependent factors are: market opportunity, financial health, estimate uncertainty, and terms of the contract.

When managers estimate projects they may also need an estimate of the programmer productivity. The productivity is usually based on measures of some attribute of the software, which is divided by the total effort. There are two kinds of productivity measures, size-related and function-related. The size-related measures are related to the size of the output from an activity. Examples of size-related measures are lines of source code and pages of system documentation. The functional-related measures are instead related to the overall functionality of the software. Examples of function-related measures are function points and object points, which are described further in 4.4.3 and 4.4.4.

There are different techniques for project estimation, and none of the techniques has shown to be generally more accurate than the other. In big projects it is advisable to use more than one technique and compare the results. If the results differ considerably, it is a sign of lack of information.

The most systematic estimation technique is algorithmic cost modelling, and it is based on information from completed projects. The technique is built on a mathematical formula that includes estimates of effort, size, and other process and product factors. Most of the algorithmic modelling techniques have a similar formula, and they also have the same difficulties. Their difficulties are due to that they rely on an estimation of size, and other subjective factors.

2.7 Software team

For a software group, or team, to be effective the communications within it must be good. The size of the group should therefore be limited. The members of the group should also feel that they are a part of the group and not just a number of individuals put together. If they feel like a group it is likely that they are motivated by the success of the group as well as their own success. There are a number of ways to promote the cohesion in a group. Some simple but effective means are to trust the members, give them responsibility, and to give them access to information.

A group also has to consist of the right type of persons to become successful. From a psychological point of view professionals can be classified into three different types [3]. The task-oriented is motivated by the technical challenge. The self-oriented is motivated by personal success. The interaction-oriented is motivated by the work in a group. All three types are important but especially the interaction-oriented because they help to improve the communication within the group.

Other success-criteria for a group are to have knowledge and experience. Everybody in the group should have the basic knowledge needed, but it is not necessary that all members are experienced. If all have experience it can cause conflicts because they all want it their own way. For the members to be productive they also need a good working space. Privacy and outside awareness are two factors that have shown to be important. [3]

2.8 Software quality

To achieve high quality of the product is the objective of most software companies, and the main reason for this is that the customers no longer accept poor quality. To achieve high quality is not easy but there are ways to work that are helpful.

2.8.1 Quality management

To achieve the desired quality of a product, an organisation should have some kind of quality management. The responsibility of the quality management is to create quality thinking in the

development process, which means that they are responsible for the quality assurance. This means to develop standards to use and to make sure that the developers follow them. A project manager should not also be quality manager, because it can create a conflict between the quality and project budget and schedule.

Quality assurance
Quality assurance (QA) is a process of defining and selecting standards and procedures that helps to reach a certain level of quality. Standards are either applied to the software process or the software product, and they are called product standards and process standards. The reasons why standards are important are that they provide a practise that is suitable for the current environment, and the likelihood of repeating mistakes decreases. Standards also create continuity in the organisation or department which makes it easier for the developers to take on someone else's work. Furthermore continuity also makes it also easier for new employees to learn the way of work and become productive.

Quality planning
A quality plan should be constructed early in the software process and it should indicate what the quality goals are. It should state what high quality is for the product, and this is important for the developers to know so they work to optimise the same product attributes. The quality plan should also define the appropriate standards to use for the current project. An important characteristic for a project plan is that it should be short, because otherwise no one will have time or willingness to read it.

Quality control
Quality control is about making sure that the developers follow the quality assurance standards and procedures. There are mainly two procedures used for quality control and they are quality review and automated assessment. Automated assessment is when software is used to compare the produced documents with the standards. It may also include measurement of software attribute. A review is when a group of people controls the documents produced and the process followed to see if the standards are followed.

Software measurement
Software measurement and metrics are part of quality management, and this is described in chapter 4.

2.8.2 Process improvement

Process improvement relies on the assumption that a good process will lead to a high quality product. This relationship is complex, and a good process does not necessarily result in a high quality product. When improving a process it is not possible to improve all attributes simultaneously. For example the attributes visibility and speed are not possible to improve simultaneously, because when document has to be produced regularly the process gets slower.

Product quality is not only affected by process quality, but also by the technology used during the development, quality of the people, and how much money and time that is assigned to the project. How these factors influence on the product quality depends on the size and characteristics of the project. For example in a large project the process used is more important than skilled people, because if communication and integration do not function it is irrelevant if some component is of excellent quality.

To analyse a process there are two techniques to use: questionnaires and interviews or ethnographic studies [3]. Questionnaires and interviews can be both quick and simple if one knows what to ask. On the other hand, inappropriate questions will lead to an incorrect model of the process. Engineers might also answer the "right" answer instead of the truth. An ethnographic study is an external observation of the process, and it should continue

throughout the whole project. It has a bigger chance of capturing a correct model of the process, but it is rather impractical and mostly used on fragments of a process.

When a model exists of a process, it is time to identify where the process might affect the product quality in a negative way. Improvements of problem-areas should propose new methods or other ways to solve them. These improvements are then merged into the process. It will take time for the changes to have any effect because the developers need a learning period and be able to do the modifications to the model.

Process measurement is an important part of process improvement. Process measurement cannot measure if product quality has improved, but it can measure if the process changes have implied any real improvement of the process.

Capability Maturity Model

Since the Capability Maturity Model (CMM) was developed in the eighties, process improvement is a serious matter in the software engineering community. CMM defines five levels of process maturity, and the main process activities are what differentiate the levels. The maturity is also an indication of how visible the development process is. A mature process is highly visible so the managers and developers are able to understand and control it. The five different levels are as follows [3, 7]:

1. *Initial*. This level means a process that lacks structure and control, and there is also lack of effective management and project plans. Projects may differ widely in quality and productivity. The visibility is zero so it is hard to measure anything meaningful. An organisation at this level should concentrate on structuring the process and thereby make meaningful measurement possible.

2. *Repeatable*. At this level the organisation has functional management but a formal process model does not exist. The level is called repeatable because the organisation can successfully repeat a project of a previous type. If a project is of a new type its success depends on the team. Management measurements are most suitable for a process at this level, and that is measurement on input like budget, schedule, requirements, staff, and output like code and documentation.

3. *Defined*. An organisation at the defined level has a well-defined and documented process. The work follows standards and procedures, and responsibilities are understood. Management have a good insight into the technical progress so cost, schedule, requirements, and quality are under control. Measurements on product attributes are suitable at this level.

4. *Managed*. At this level the process is managed and feedback from early activities is used to set priorities in current activities. Both product and process metrics are collected and used as feedback. This makes the development predictable and the quality of the products high.

5. *Optimising*. This is the highest level, and the whole organisation is focused on continuous process improvement. Measurements from activities are used to improve the process, by removing and adding activities and changing the structure of the process. Improvement also occurs by innovations and the use of new techniques.

CMM was developed for large organisations with projects that last for a long time [3]. This does not make suitable for all kind of organisations, for example it is too bureaucratic for

small ones. That CMM should not be applied in all organisations has been recognised but not explicitly stated, which have resulted in an overuse of it as a software process problem solver.

2.8.3 Dependability

A dependable system performs what the user expects it to do and it does not fail at normal usage. The *dependability* property of a system is composed of four sub-properties that all add up to how dependable a system is [3]. These sub-properties are: *availability*, *reliability*, *safety*, and *security*. In software development the goal is to make software as dependable as possible, but to do this is associated with high costs. The cost of increasing the dependability can increase exponentially due to additional design, implementation, and validation. An increase can also reduce the performance because the system has to control and stop itself from performing illegal actions. Usually dependability is considered more important than performance because undependable systems are often unused and the costs of a failure can become enormous. It is also possible to compensate for bad performance, and undependability may cause information loss.

The properties reliability and availability are closely related and the definitions are as follows [4]. *Reliability* is the probability of failure-free operation over a specified time in a given environment for a specific purpose. *Availability* is the probability that a system, at a point in time, will be operational and able to deliver the requested services.

This raises the question what failure-free is, or perhaps better how a failure is defined. Table 2.1 below describes Sommerville's [3] classification of different problems. Fenton and Pfleeger [7] make a different definition, but remarks that everybody does not need to use the same definitions, but for each product the definitions must be clearly defined so they can be translated. This is discussed further in section 4.4.2**.**

Failure	An event that occurs at some point in time when the system does not act as expected by its users.
Error	Erroneous system behaviour where the behaviour of the system is not consistent with the specification.
Fault	An incorrect system state, for example a state that is unexpected by the designers of the system.
Human error or mistake	Human behaviour that results in a system fault.

Table 2.1 Classification of different problems. [4]

Reliability and availability are considered as most important of the four sub-properties. An unreliable system has difficulties to ensure safety and security. A system that is not available can result in big economic loss.

The *safety* property of a system is to make sure that it neither hurts nor threatens the environment or people. Safety is not a big issue in all software development but there are safety-critical systems like control system in airplanes and chemical plants. Safety-control is easier to implement in hardware, but the complexity of today's system makes software control necessary.

The last sub-property of dependability of a system is security. The *security* of a system is how it can protect itself from external attacks, which can be viruses, unauthorized use of service, unauthorized modification of data, and so forth. Security is important for all critical systems, because a non-secure system affects the availability, reliability, and safety.

2.8.4 Configuration management

Software development is an evolving process, and during the development many versions of software may be used and under development. Different versions exist for different hardware, operating system, corrected faults, and proposals for change. To keep track of the versions there has to be some standard procedure of how to manage the evolving product, this is what configuration management is about.

Configuration management is related to software quality in the following way. After the developers have finished a new version they pass it over to the testers that make sure that the quality is acceptable. The configuration management team then takes over and controls the changes of the software.

The responsibilities for a configuration manager are to keep track of the different versions, but also to ensure that new versions are delivered to the right customer at the right time. Configuration management should rely on standards and procedures and the configuration management plan should describe these. Every project should have its own specific plan, and an important part is that it should clearly define who is responsible for the delivery of each document (configuration item).

Configuration management involves large amounts of data and requires attention to detail. CASE (Computer-Aided Software Engineering) tools to support the process are essential, and there exist a variety of tools for the different activities.

3 Measurement theory

Measurement is something we meet every day, and it is about describing things so they become comparable. Fenton and Pfleeger [7] make the following formal definition:

"Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules."

According to the definition an entity can be: a physical object, an event that occurs at a specific time, or an activity that lasts during a time interval. Examples of entities are a program, a milestone, and the testing phase of a software project. An attribute is a characteristic of an entity, for example code size of a computer program, or effort spent on an activity. [10]

To make a distinction between *measure* and *measurement*, the notation in [7, 10] is followed. *Measurement* is the process when values are assigned to attributes of entities of the real world. A *measure* is the result from a measurement, so it is the assignment of a value to an entity with the goal of characterising a specified attribute. A measure should be seen as a function that associates a value with an entity.

In measurement one has to specify both the entities and attributes, and to measure an entity or attribute alone is the wrong approach. For example to measure only a program does not make any sense, but to measure the size of a program does. There are a number of issues in measurement that are important to think about. These are:

- *Scale*. Different scales can measure the same attribute, which one is appropriate in this case?
- *Accuracy*. Are the measures clearly defined and are the measuring instruments working properly?
- *Error*. Which are the possible error sources, and how big errors are acceptable?
- *Conclusion*. Are the results sufficient to draw reliable conclusions from?
- *Influence*. Are there other factors that influence the results?

3.1 Direct and indirect measurements

In measurement theory one separates between direct and indirect measurements. A *direct measurement* is a measure of an attribute of an entity with no other entities involved. For example size of source code is measurable without any attributes of other entities and is hence a direct measurement. The *indirect measurements* are the ones that are not possible to take direct. For example programmer productivity is an indirect measurement, when the size produced is divided by the effort.

3.2 Scales

When measuring it is important to know which representation that is most suitable for the attribute. In measurement theory there are five different scales: nominal, ordinal, interval, ratio, and absolute. There are other scales but these five are the ones that are most used in measurements [7, 10].

The *nominal* scale is the most basic scale and it only places the entities in different categories or classes. The classes are identified by unique symbols or numbers, and cannot be interpreted as anything else than just identifiers.

The *ordinal* scale is richer than the nominal scale, which means that all relations in the nominal scale are contained in the ordinal. The ordinal scale places the entities in classes that are ordered with respect to the attribute. A unique number must represent each class, but the numbers are only used for ordering. As long as the ordering stays the same, it is allowed to use mapping.

The *interval* scale contains all information that the ordinal has, but also the interval between the classes. The size of the interval is the same between all classes, and any mapping that preserves this size is allowed to use. Addition and subtraction is therefore allowed, but neither multiplication nor division. Celsius and Fahrenheit are two examples of interval scales.

The *ratio* scale is the most useful scale of measurement. It preserves the ordering, the size of the intervals, and also the ratios between entities. It has also a fixed reference point, a zero element, which is 0. The scale must start at zero and increase at equally big steps. Addition, subtraction, multiplication, and division are applicable to the ratio scale, which makes it especially useful. To measure length is to use the ratio scale. Another example is the Kelvin scale, which has a clearly defined zero element and consists of equally increasing steps.

The *absolute* scale is used when there is only one possible way to measure an attribute. Absolute measures are always counts of the number of occurrences of something in the entity. The absolute scale is common in software engineering, for example number of people in a project, and number of defects found during testing.

The reason why scales are important is that they constrain the analysis of the measures. In software engineering it is many times hard to be certain of the scale type for a measure. Briand et al. [11] recommends caution even when following commonly found directions for which statistics to use. They also urge for the use of parametric statistics in areas where the scale is not completely known, but only with care and after thorough consideration. The reason for this is that parametric statistics are much more powerful but riskier than non-parametric statistics. It is possible to miss important relations if one only uses only non-parametric statistics in insecure areas. The difference between parametric and non-parametric statistics is that the parametric assumes an underlying distribution, when the non-parametric does not. For example a variance analysis assume a normal distribution and is therefore parametric.

3.3 Classification

The first thing to do when measuring software is to identify the entities and attributes to measure. In software each entity and attribute classifies to one of the three classes: process, product, or resource [7].

1. *Process*. The process class includes all the measurements that are related to software activities. Typical process entities are to write specifications and to test, and some of their attributes are time, effort, and defects found.

2. *Product*. The product class includes all the documents and deliverables that result from the process activities. For example measurements on the entities specification, design, and code belong to this class.

3. *Resource*. The resources in a software project are personnel, hardware, software, office, and other things that are needed in the everyday work. Measurement of attributes of these entities fall into the resource class.

3.4 Internal and external attributes

It is common to separate between internal and external attributes within the classes. The *internal attributes* are those that can be measured on a process, product, or resource without caring about the external behaviour. Examples of internal product attributes are source code size, and code complexity. The *external attributes* are measurements of how a process, product, or resource is related to its environment. Examples of external process attributes are quality, and requirements stability.

3.5 Validation

Validation of software measurement is different for a measurement system and prediction system. A *measurement system* is concerned with measuring attributes of existing entities. A *prediction system* is used to predict attributes of entities, and involves a mathematical model for the procedure. Fenton and Pfleeger [7] say that a measure is valid if it accurately characterizes the attribute it claims to measure, and a prediction system is valid if it makes accurate predictions. This is not the only definition, for example Kitchenham et al. [13] assume the following conditions to be true for a valid measure:

1. A measure must not violate any necessary properties of its elements.
2. All models used in the measurement process must be valid.

There has been some disagreement of how to validate software metrics. In the past many practitioners and researchers did not pay as much attention to validation. It was not uncommon to start to use measures that only had been validated by the person proposing them. This is not acceptable but one has to find out whether the measure captures what it claims to describe.

To validate measures by showing a correlation to existing valid measures should be used with extra care. A statistical correlation does not mean that the values have an explicit relationship. If we for example want to measure obesity by measuring height in centimetres, we can measure height and weight for a large number of people. We can then show that height correlates strongly with weight but still height is not a valid measure of obesity because there are other factors to take into account.

To use experiments and to test hypothesises is how prediction systems are validated. How big prediction errors that are acceptable, *acceptance range*, depend on various things and are to some extent subjective. The acceptance range must be specified before using a prediction system.

A measure can be valid in one sense but not for another, so the purpose of the measurement has to be taken into account. The notation is that if a measure is valid for assessment it is valid in the narrow sense, or internally valid. If it also is a component of a valid prediction system it is valid in the wide sense.

4 Software metrics

Measurement in software engineering is called *software metrics*, or more precise software metrics are any type of measurement that relates to a software system, process or its documentation. The main reason for measuring a software project is to get information about it and the organisation, and be able to control the projects better. There are many more specific reasons for measuring and they differ between the manager's and the developer's perspective. The managers are concerned about issues like: what does the process cost, how productive is the staff, how good is the code, is the customer satisfied, and how can we do better? The developers care more about: are there any more faults, can we test the requirements, have we achieved our process and product goals, what will happen in the future? Software measurement can help to keep the managers and developers informed about their concerns, but it does not claim to give any absolute solutions.

4.1 What to measure

In most software projects time is a scarce resource, and it is therefore important to make sure that the measurement taken are meaningful. In order to make the measurements effective and meaningful they should be [9]:

- Goal oriented
- Applied to products, processes, and resources.
- Interpreted based on an understanding of the organizational context, environment, and goals.

This implies that the measurements should be defined from top to bottom. One first defines the goals, then measures, and at last interprets the measures. How to interpret a measure depends on the goal, and to leave this fact out can lead to misinterpretations. But to do it top down is not always practical and see more about this in V-GQM below.

4.2 Goal Question Metric

Goal Question Metric (GQM) is a goal-oriented approach that helps to decide why and what to measure, and it is based on the idea that all measures in a measurement program should be meaningful. GQM has a hierarchical structure where first the goals are defined. The goals are then refined into questions that are refined into metrics, see figure 4.1 below. The figure also indicates that the interpretation should be done bottom-up, for example metric 5 (m5) should only be used to answer question 7 (q7) that will help to achieve goal 3.

The goals should state what an organisation wants to achieve with the measurements. It should specify five, (sometimes only four), dimensions [10]:

- *Purpose*: the reason for the measurement.
- *Object of study*: the entity or entities that should be studied.
- *Quality focus*: the attribute or attributes that should be studied.
- *Viewpoint*: the viewpoint from which the measure is taken.
- *Environment*: the specific project or environment where the measurement takes place. This dimension is sometimes left out.

When a goal is defined, it is refined into several questions that help to divide the issue into its components. The questions are then refined into metrics that will help to answer the questions. Metrics are either objective or subjective. An objective metric depends only on the measured entity, but a subjective metric also depends on the viewpoint it is taken from.

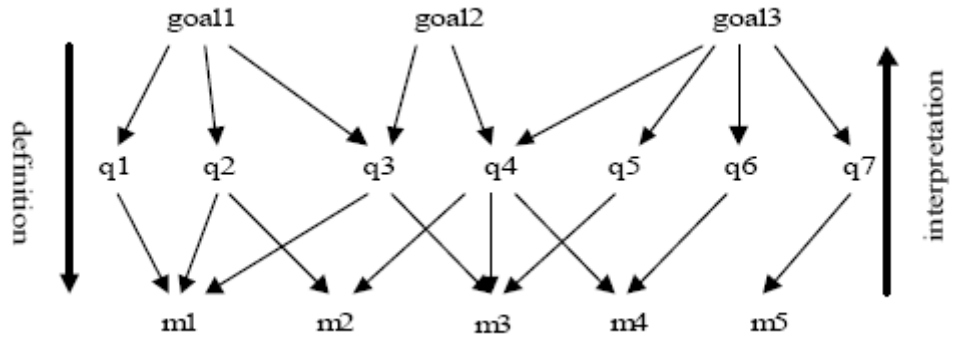

Figure 4.1 A GQM example. [10]

The GQM approach is not static but might need refinement. It is therefore important that the metrics not only help to evaluate the entities but also the reliability of the whole model, which is that the measures should be mature. The maturity of measures also gives an indication of how to evaluate them. Measurement of stable and mature entities can be evaluated objectively while unstable and informal entities should be evaluated subjectively.

GQM is a part of a structured approach for improvement of software products and processes, which is known as the Quality Improvement Paradigm (QIP). The basic idea of QIP is that improvement can be achieved through empirical knowledge of software processes and products. The QIP is out of scope for this project and will not be described any further.

The GQM approach is well known and used in practise but there is still some criticism, there are basically three objections. The first objection is that the approach is not repeatable. Two persons, who start with the same goals, will refine them differently and end up with different questions and metrics. A second objection is that it is difficult to know when to stop create questions, and to start defining metrics. A third objection is that it is not practical, when some of the questions may be impossible to answer with the current organisation [16]. Despite this, GQM is currently the best approach and it has been successfully used in many software organisations [10]. But due to its shortcomings researches have proposed a number of improved GQM approaches. One of them is V-GQM that is described below.

In [31] Olsson and Runeson present an extended GQM, which they call V-GQM (Validation Goal Question Metric). The purpose of the V-GQM is to take unforeseen benefits of the metrics into account and to improve subsequent GQM studies. When the original GQM stops after the analysis of the gathered data, V-GQM has three additional steps, which are metric validation, question analysis, and goal refinement. Figure 4.2 is a visual presentation of the model.

The first additional step is metric validation. This means to analyse the collected metrics and classify each one in one of the categories: unavailable, extended, possible to generalise, and sufficient. These categories make the base for the analyses of the metrics. For example metrics that classify as extended collect more data than necessary (compared to the plan). The question is why this happened, and why it was not included in the original plan. The empirical knowledge gained here can be used in the subsequent GQM.

To analyse the questions means to improve and derive new questions, with input from the metric validation step. Removal of questions is only allowed to those that are obviously irrelevant. In this step the people analysing the questions should also estimate the effort

required to answer them, and the benefits of collecting them. This information is helpful in the next step to see the extent of the measurements.

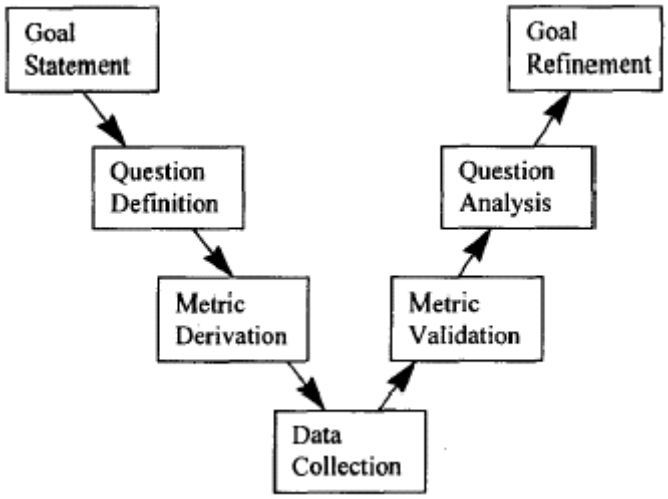

Figure 4.2 The V-GQM model. [31]

Goal refinement is the last step, but also the first step in the subsequent GQM. The new goals depend on if there are any new external facts, and the analysis of the last study. This step also includes removing questions that not conform to the organisational goals.

Olsson and Runeson [31] do note that there are problems with V-GQM, first of all it is an empirical model and might not be perfectly objective. Another risk is that the whole study may evolve against a certain result.

4.3 Measurement program

A measurement (or metrics) program is a program for how to integrate software measurement in an organisation. The program does not need to apply to the whole organisation, but it can as well just apply to one project.

4.3.1 Measurement-based improvement

Niessink and van Vliet present in [12] a generic process model for what they call measurement-based improvement. They assume that measurement itself is not a goal, but the goal is organisational, or to solve an organisational problem. They also assume that the measurement activities are performed in combination with improvement activities to reach the goal. The model is visualized in figure 4.3.

The process starts at the leftmost dot with an organisational problem or a goal. The organisation analyses the problem and arrive in the middle, with either a solution or a cause to the problem. If they have enough information to solve the problem they implement it and arrive at the goal (leftmost dot). If they have not enough information they need to implement a measurement program or design an experiment (right dot). Analysing the gathered information takes the organisation back to the middle with a solution. They then implement the solution and arrive at the goal (left dot). This model is very simplified and it might be that the organisation has to loop the right part many times to find a solution.

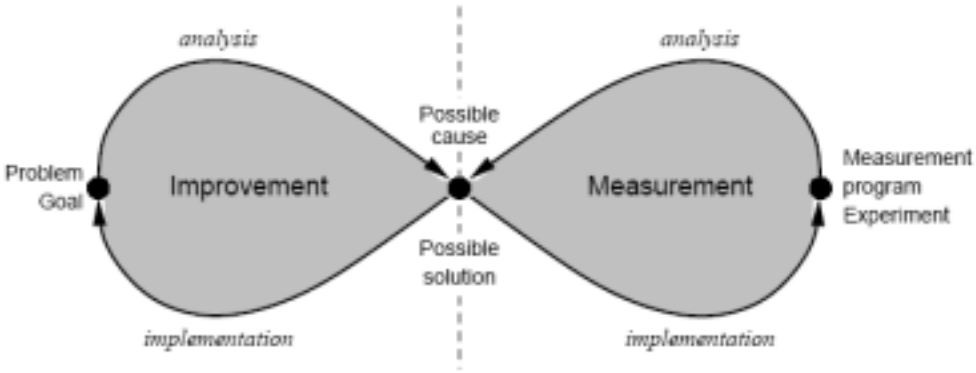

Figure 4.3 The Measurement-based improvement model. [12]

The authors make a clear distinction between GQM and the measurement-based improvement model. The goal of GQM is a measurement goal (can be organisational but does not need to be) while the goal of this measurement-based improvement model is organisational. The GQM approach can be used in this model and that would be during the design of a measurement program (lower right arrow).

4.3.2 Metrics plan

A metrics plan should describe *why*, *what*, *where*, *when*, *how*, and *who* of the metrics. The reader of the plan can then understand why the organisation collects the metrics, how they use them, and how the metrics fit into the bigger picture of software development or maintenance.

The question of *why* and *what*, GQM can answer as described in section 4.2. The *where* and *when* exactly describes how measurement is integrated with the everyday development activities. Some organisations use a process model that allow the developers to decide the *when*, *how*, and *who*.

The *how* regards how to collect the metrics. This step is important because if the collected data is not good (see section 4.8.1) the analysis of it can be meaningless. The metrics plan should describe the collection-process of the specified data and how it relates to the goals of the program. Fenton and Pfleeger [7] state two principles that the collection process should adhere to. It should be simple and minimise disruption of everyday work, and the metrics data should be stored in a database. To use tools help to make the collection simple, and can also help in other aspects like planning, prediction, data analysis, and data storage.

The *who* in the metrics plan consider the people involved in the program. As described in the introduction to section 4 the perspectives for measurement differ for a developer and a manager. The metrics plan should take many perspectives into account to make it useful and interesting for as many as possible. The plan should also make responsibilities and roles clear.

A metrics plan is not static but should be revised periodically to meet changes in the organisation.

4.3.3 Measurement team

Organisations often form a measurement team that helps the managers and developers to understand and use the metrics [7]. The team operates best at a higher organisational level where it gets the bigger perspective. It is the team's responsibility to handle the data and make the analyses. The advantage of this is that the developers can concentrate on their primary tasks. Some other responsibilities of the team are to make the measures and analyses available

as soon as possible to the developers that took them, but they should only be available to the right people. The members of the team should have development experience to make it credible, otherwise the developers might not be as willing to participate.

4.3.4 Successful measurement programs

Most of the measurement programs implemented do not last for two years and are considered unsuccessful. Dekkers et al. report data that about 80 % of the analysed measurement programs in 1998 were unsuccessful [25]. Some of them even caused harm by misusing the measurements.

There are examples of successful measurement programs and they share some characteristics. For example the following is recommended [7, 25, 26]:

- Start small and simple, and increase the measurement when the organisation and team have more experience. This approach takes longer time than a more ambitious approach, but the likelihood of success is greater.
- Set solid objectives and plans. Dekkers et al. recommend implementing the measurement program as a normal software project with requirements, design, build phases, and formal management [25].
- Create an environment where the collection process is safe and the data is correct. Developers must have incentives to not collect incorrect data.
- Empower and encourage developers to use measurement information.
- Information from a good measurement program is interesting for a limited time so the right people must get it in time.

4.4 Measures of internal product attributes

Most measurement of internal product attributes regards the software program. Reasons for this are that all software development includes a program, and that code is very formal. Specifications and requirements are artefacts that are less formal, and measures of them might also be subjective to some extent. Measurement of non-formal artefacts exists after all and to measure early activities like specification and requirements are important because they impact the whole development process.

4.4.1 Requirements

A frequently used measure is to count the number of requirements [14]. The requirements count is not very useful in itself, but for estimation of size, duration, and effort. The estimation can be a mathematical model that is based on data from previous projects, in example average time to code a requirement. It is also common to calculate the volatility of the requirements, but this is an external product attribute. To use requirements count requires requirements of somewhat the same size or to use weights for different requirements.

4.4.2 Code

The most widely used measure of program size is source lines of code. When measuring this it is crucial that the definition of what counts as one line is clearly defined. Fenton and Pfleeger [7] recommend recording of non-commented, *NCLOC*, and commented *CLOC* lines of code separately. They then define: lines of code *LOC = NCLOC + CLOC*. This is only one definition of a code measure and other can be useful as well. *NCLOC* is also called effective lines of code, *eLOC*. Non-commented source statements, *NCSS*, is another common measure and as the name indicates it counts the number of non-commented lines with source statements, this means that it does not count a line with a single curly brace or parenthesis. [3] The main advantages of measuring lines of code are that it is easy to do automatically, and that it correlates in some way with programming effort. The disadvantages are that it is

language dependent, developer dependent, and that it encourages more is better as a programming style.

An alternative to use number of lines of code is to use function points as a size measure. The advantages of this are that they more accurate reflect the output, they can be used during the whole development, and they can be used to measure progress by comparing complete with incomplete ones. The obvious drawback is that function points are harder to compute than lines of code. See more about function points in 4.4.3.

A general problem with text-based length measures is that one line of code becomes less meaningful at usage of automated code generators, and visual programming. Object points are an alternative size measure, and are described in 4.4.4. Another alternative is to take the reused code into account that will give a more accurate picture of the lines of code produced.

To define code reuse is not as simple as it may sound. Sometimes whole files are reused but more often the reuse only covers a unit of code (a function, module, or procedure). The question is what to count as reused code. Some suggestions do not count an exact number but only to what extent the code is reused. One definition used at NASA/Goddard's Software Laboratory is [7]:

1. Reused verbatim. The code in the unit was reused without any changes.
2. Slightly reused. Fewer than 25% of the lines of code in the unit were modified.
3. Extensively modified. 25% or more of the lines of code in the unit were modified.
4. New. None of the lines of code is reused.

A simplified version of code reuse is to call level 1 and 2 reused and level 3 and 4 new.

4.4.3 Function points

Function points are a functional size measure of a system. Allan Albrecht at IBM lead the development of it, and his intention were to find a measure of size independent of technology and programming language.

The value of Function Point (*FP*) is the product of Unadjusted Function Points (*UFP*) and Technical Correction Factor (*TCF*), *FP = TCF * UFP*. To calculate the unadjusted function points it is necessary to count the number items of the following types (from the system specification):

- External inputs. Inputs from the user that provides application oriented data to the software. For example file name and menu selection.
- External outputs. Outputs to the user that provides application oriented data. For example reports and error messages.
- External inquiries. Input that requires a response.
- External files. Machine-readable interfaces that are used to transmit information to other systems.
- Internal files. Files with the main logic in the system.

Every item should also be classified according to its complexity as low, average, or high. The *UFP* is the sum of all items weighted as in table 4.1.

$$UFP = \sum_{i=1}^{15} weight_i * \#items_i$$

Complexity Item	Low	Average	High
External inputs	3	4	6
External outputs	4	5	7
External inquiries	3	4	6
External files	7	10	15
Internal files	5	7	10

Table 4.1 Unadjusted function points' complexity weights. [7]

TCF consists of 14 complexity factors presented in table 4.2. These factors are subjectively ranked 0-5 based on their influence, and the value of TCF is:

$$TCF = 0.65 + 0.01\sum_{i=1}^{14} F_i$$

It is then trivial to calculate FP = TCF * UFP.

F1 Reliable back-up and recovery	F2 Data communications
F3 Distributed functions	F4 Performance
F5 Heavily used configuration	F6 Online data entry
F7 Operational ease	F8 Online update
F9 Complex interface	F10 Complex processing
F11 Reusability	F12 Installation ease
F13 Multiple sites	F14 Facilitate change

Table 4.2 Components of the technical complexity factor. [7]

Function Point Analysis (FPA) has become an accepted method for estimation of project size and development effort, but it is most used in the area of information systems [30]. A large community called IFPUG (International Function Point User Group) offers support to set up a (FPA) program. There is a desire to standardise function points, or functional measurement, and it is called Functional Size Measurement. A concern about the consistency of FPA is that it is not sure that two persons will come up with the same result for the same system [21]. There exist a number different function point counts but the different results may vary significantly [30]. The FPA also has a tendency to underestimate the size when the specification is not detailed enough [7].

In [19] Jones introduced a technique called Backfiring, and it consists of bi-directional equations to convert between function points and lines of code. This can be useful when using metrics both with lines of code and function points. The Backfiring technique relies on the assumptions that there is a direct relationship between lines of code and function points, and that this relationship is fairly stable. There is no empirical evidence for those assumptions [20] so to generalise the Backfiring technique is not advisable [21].

4.4.4 Object points

Object points are a size measure used earlier than function points in the development process and it is only applicable to object-oriented systems. To compute object points, the first step is to count the number of screens, reports, and third-generation components that will be part of the application. Every screen and report should also be classified as simple, medium, or difficult, using the tables 4.3 and 4.4 as guide. The third-generation components do not need to be classified because they have the same weight. The screens and reports are weighted according to table 4.5. The sum of all weighted screens, reports, and third-generation components is the single object point number. The object points also take reuse into account. When r% of the object points will be reused from previous projects, the new object points are:

*New object points = (object points) * (100 – r) / 100.*

Object points are used in the COCOMO II estimation model that is described in section 4.7.3.

	Number and source of data tables		
Number of views contained	Total < 4 (<2 server, <2 client)	Total < 8 (2-3 server, 3-5 client)	Total > 7 (>3 server, >5 client)
<3	Simple	Simple	Medium
3-7	Simple	Medium	Difficult
>7	Medium	Difficult	Difficult

Table 4.3 Complexity level for screens. [7]

	Number and source of data tables		
Number of sections contained	Total < 4 (<2 servers, <2 clients)	Total < 8 (2-3 servers, 3-5 clients)	Total > 7 (>3 servers, >5 clients)
0 or 1	Simple	Simple	Medium
2 or 3	Simple	Medium	Difficult
>3	Medium	Difficult	Difficult

Table 4.4 Complexity level for reports. [7]

Object type	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component	-	-	10

Table 4.5 Complexity weights for object points. [7]

4.4.5 Cyclomatic complexity

Thomas McCabe introduced the measure cyclomatic complexity, *CC*, and it is also known as McCabe's complexity or just program complexity. It measures the number of linearly independent graphs in a program module, when a connected graph represents the module. This is also known as the degree of logical branching of a function, which appears at every *for*, *while*, *case*, *if*, and *goto*. Technically the calculation of cyclomatic complexity is $CC = E - N + p$, where E is the number of edges, N is the number of nodes, and p is the number of connected components of the graph. The cyclomatic complexity is useful for risk analysis during the development. A high *CC* value for a module means higher risk, but not automatically that the module needs a new design, because there are other facts about the application to take into consideration. The *CC* value can also approximately tell how much testing that is necessary; a complex module needs more testing than a simpler one [17].

4.4.6 Object oriented metrics

When software metrics first evolved functional programming was all there was. When object-oriented programming increased in importance and soon became standard, there was a desire for object-oriented metrics. Chidamber and Kemerer introduced the following six object-oriented metrics [4, 22]:

1. Weighted Methods per Class, *WMC*. *WMC* measures the complexity of a class. For a class with methods M1, M2… Mn, which are weighted respectively with complexity c1, c2, …, cn, the *WMC* is:
$$WMC = \sum_{i=1}^{n} c_i$$

2. Depth of Inheritance Tree of a class, *DIT*. *DIT* is the maximum depth of the inheritance tree of each class. In case of multiple inheritances, it is the maximum

length from the node to the root of the tree. *DIT* is a measure of how many ancestor classes a class can potentially affect.

3. Number Of Children, *NOC*. *NOC* is the number of immediate subclasses subordinated to a class in the class hierarchy.

4. Coupling Between Object classes, *CBO*. *CBO* for a class is a count of with how many other classes it is coupled with. An object is coupled to another object if it acts on the other, for example when a method in one object uses a method in the other object.

5. Response For a Class, *RFC*. *RFC* is the number of methods that can potentially be executed in response to a message received by an object of that class.

6. Lack of Cohesion On Methods, *LCOM*. *LCOM* is the number of pairs of member functions without shared instance variables, minus the number of pairs of member functions with shared instance variables.

An object-oriented threshold is a range of values on a product metric that is between acceptable and unacceptable values [18]. A threshold could for example be that the average number of attributes per class should not be more than 6, or otherwise the class is doing too much. The problem is to identify meaningful thresholds.

4.5 Measures of external product attributes

In software measurement it is more common to measure on internal than external attributes. The main reasons for this are that internal attributes are available for measure early in the life cycle, and that they are easy to measure. Although external product attributes should not be neglected.

4.5.1 Quality models

Software quality consists of many characteristics and they can be measured on both internal and external attributes. To get an understanding of quality it is helpful to make a quality model that visualises the characteristics and their relationships. In most models the focus is on the final product and the notion of quality is from the user's perspective.

Software quality is often expressed in high-level external attributes, or quality factors, like reliability, usability, and maintainability. These *factors* are too abstract to be measurable directly and need to be refined into lower-level quality *criteria*. Examples of criteria are correctability and testability, but neither these can be measured but must be refined once more. The criteria are refined to quality *metrics*, for instance fault count, and degree of testing, see figure 4.4.

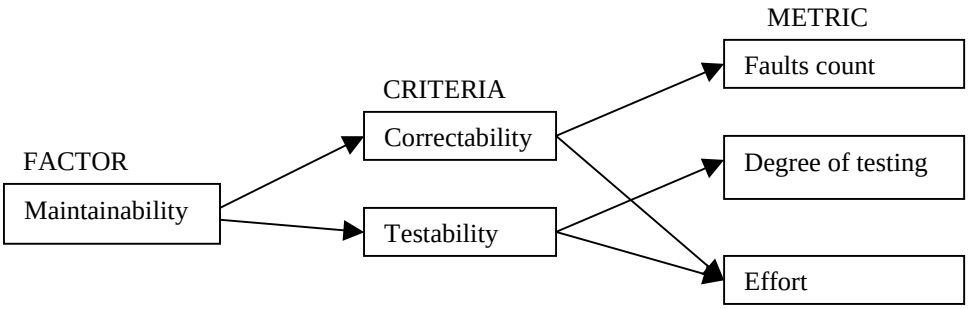

Figure 4.4 Example of a quality model for maintainability. [7]

Since 1992 there is a standardised quality model (ISO/IEC 9126). This standard composes software quality in six characteristics that are *functionality*, *reliability*, *efficiency*, *usability*, *maintainability*, and *portability* [29]. These are refined into sub-characteristics, which can be measured by internal or external attributes. The standard is not widely accepted but one step toward a general view of software quality [7].

4.5.2 Quality measures

To measure quality is difficult because it is somewhat subjective, for instance two persons can have totally different opinion of how user-friendly a program is. Here follows a description of some views of quality and measures [29].

Reliability
IEEE defines reliability as a systems ability to perform its required functions under stated conditions for a specified period of time [33]. Most reliability measurement are measures of defect detection and removal, like mean time to failure (MTTF*)* that measures how long time the system on average runs before a failure. Rosenberg et al. [33] stress the need for measurement that helps to prevent errors during the development, and not just measurement of errors when they occur. One of their findings that can be used during the development is that classes that are large and have high complexity tend to be the most unreliable. Also very small classes that have high complexity are a reliability risk because they are hard to change.

Usability
Usability measures are concerned with how easy it is to use a product, or with other words how user-friendly it is. Most people have an intuitive feeling of what user-friendly is, but there is not a simple measure for this. Instead assumptions often are made that a well-structured manual, good use of menus and graphics, informative error messages, etc. are evidence of good usability. Unfortunately there are no explicit relationship between these internal attributes and good external usability. An example of a measure of usability is how long time it takes to learn a program.

Maintainability
Maintainability measures can help to tell whether the software is easy to understand, enhance, or correct. Measures of maintainability are either external or internal. External ones measure the maintenance process, which assumes that if the process is effective then the product is maintainable. Internal measures of maintainability are measures of internal product attributes. Mean time to repair (MTTR*)* is a widely used maintainability measure and it is the average time it takes to implement a change and get the system up and running again.

Defects as a quality measure
To use external attributes for quality measurement takes planning and data collection, and this costs money. Project managers that are not willing to spend these extra resources have created a view where they interpret lack of defects as good quality. The reason why this is cheap is that defects normally are reported anyway.

Defect density is a measure that is widely used, and considered as an industry standard, and it is defined as the number of found and known defects divided by the product size. Product size is measured in lines of code, or any other size measures like function points or object points. Another quality measure based on defects is system spoilage. System spoilage is defined as the time to fix post release defects divided by the total system development time.

In lack of measures of quality attributes defect counts is acceptable, but one must remember that there are some issues to consider.

- During what activity or activities are the defects counted?

- The defect density can be a measure of how good the process of finding defects is, instead of being a product measure.
- The number of defects does not say anything about the seriousness of them.

Defect classification
Defect classification means to classify defects according to their attributes, and this is necessary to be able to analyse them properly. The classification should be orthogonal, which means that the classes should not affect each other. IBM developed orthogonal defect classification, *ODC*, which is a scheme to capture the semantics of each defect quickly [27]. With *ODC* one classifies activity, trigger, and impact at discovery of a defect and target, defect type, qualifier, source, and age at closing of it. There are standard values for the attributes but each organisation decides exactly which to use.

Another defect classification model is Hewlett-Packard's defect origins and types [28]. Figure A.2 in appendix A visualises this model. Origin is the first activity in the lifecycle where someone could have prevented the defect, and not where it was found. Type is the area within a particular origin that is responsible for the defect. Mode is the reason why the defect occurred.

Fenton and Pfleeger [7] record the following from defects: location, timing, symptom, end result, mechanism, cause, severity, and cost. Exactly what an organisation decides to measure depends on how they will analyse the defects.

4.6 Measures of resources

As mentioned in section 3.3 the resources in a project are the personnel, the office, the software, etc. Personnel productivity is a resource measure and it is measured as the produced size divided by the effort spent. This is a product measure divided by a process measure, but considered as a resource measure when it measures the personnel. Other resource measures are on teams and tools.

4.6.1 Productivity

Productivity is measured as the amount of output divided by the input. In software engineering it is common to use input as the number of person months spent, and the output as number of lines of code produced. Productivity measures can help managers to make estimates of project effort, but this requires a size estimation of the final product which still is very difficult to do.

The productivity measure source lines of code per programmer month was introduced when FORTRAN, COBOL, and assembly programming was used. In these languages it is easy to count the number of lines of code, which is the same as the number of statements. In languages like Java and C++ there is not a simple relation between the number of statements and the number lines of code, which makes it possible to count in different ways. Without the exact same definition of what counts, comparison of the productivity is not meaningful. To compare productivity between different programming languages is also connected with difficulties. One language might need five lines to express what another language can express in one line. It is not even safe to compare productivity with the same language, because all programmers have their own style, which generates different number of lines of code. Functional measures like function-points and object-points are independent of the implementation and therefore avoid these problems.

The personnel productivity only measures the productivity during the implementation, and therefore Fenton and Pfleeger [7] suggest using other measures of productivity as well. When

measuring personnel productivity it must be clear that the data relate to the individual. The productivity does neither say anything about the quality of the code, which is more important than just a high productivity.

4.6.2 Process productivity

How much time and effort that is spent on a software product should reflect how much functionality it has. Putnam and Myers [8] introduced the *software equation* as:

*Product = Constant * Effort * Time*

Product represents the functionality, effort is the effort that all personnel spent on it, and time is the duration of the project. The constant term is balancing the other terms. For example if two projects spend the same amount of time and effort but the result has not the same functionality, the constant term is balancing the equation. The constant term is therefore more an indication of the productivity, and a better way of expressing the equation is:

*Product = Productivity * Effort * Time*

The productivity term is still not precisely defined, and that is because the productivity in a software project depends on many factors. Some of the contributing factors are management style, programming language, complexity of the application, staff skills, technology used. To give the term productivity an even more descriptive name, Putnam and Myers call it process productivity parameter.

The equation is still too general to be useful. After analysing a large number of software projects Putnam and Myers came up with the following empirical equation:

$$Product = Process\ productivity\ parameter * (Effort/B)^{(1/3)} * Time^{(4/3)}$$

Product is the number of lines of source code, new and modified but without commented lines and blanks. Process productivity parameter is a number obtained by calibrating old projects. Effort is the years of effort spent on main build phases (design, implementation, inspection, testing, documentation, and supervision). B is a function of system size, see table A.1 in appendix A. Time is the duration of the main build phases expressed in years. If a project only involved one, two, or three people there are special concerns. If this is the case and the functional design and main build phases ran together, the time and effort should be scaled down by 20 percent. The interpretation of the process productivity parameter should also be handled with care, because in small projects it is more a measure on the staff than the organisation. This is the related to the fact exemplified in process improvement in section 2.8.2; which process to use is more important in big projects than in small.

During the study of previous projects Putnam and Myers noticed that the process productivity parameter values clustered at certain levels. They assigned a number to the clustered levels and created a productivity index. See the index in table A.2 in appendix A.

The software equation seems to make sense but there are objections. If an organisation develops a product with a certain level of functionality, the effort and time are the only variables. This means that if the duration decreases the effort increases, but also that the effort decreases when the duration increases. In the equation product is the functionality of the program, but how do you measure functionality? Practically they use lines of code as a measure of functionality, but the relation between functionality and size is far from clear.

Maintenance projects

Maintenance of previous projects is different from developing new projects. It usually includes writing new code, and changing and removing existing code. The software equation takes the number of new lines of code so for a maintenance project the work has to be

translated into the corresponding new lines of code according to table 4.6. Note that the number of added, changed, and deleted lines of code is a subset of reused, and they should sum up to equal to or less than the number of reused lines.

Type of modification	Effort ratio (modify to new)
New code. The lines of code in new modules.	1.0
Reused. The lines of code in modules that will be reused, but modified (added, changed, or deleted).	0.27
Added. The lines of code that are added to reused modules.	0.53
Changed. The lines of code that are changed in reused modules.	0.24
Deleted. The lines of code that are deleted from reused modules.	0.15
Removed. The lines of code in modules that are completely removed. Testing must still take place which takes some effort.	0.11
Tested. Lines of code from reused modules that needed no modification still require testing with the new code.	0.12

Table 4.6 Translation table for maintenance projects. [8]

4.6.3 Team, experience, and tools

How productive the developers are in a development team depend on many factors, for example team structure, tools, and methods. In the seventies empirical results indicated that team organisation and complexity affect the productivity [7]. The results claimed that complex team structures lead to low productivity. There is still little published evidence of how team structure affects productivity or quality, and therefore we must rely on anecdotal evidence and expert judgement. One study at Hewlett-Packard introduced a measure of communications complexity within a team. But Fenton and Pfleeger [7] reject this single measure because it does not consider the underlying factors of complexity.

Most people would agree that experience is one factor that contributes to productivity, but it is hard to measure. It is also important to distinguish experience of individuals and experience of the team. Some teams do not function well, and although the individuals are experienced the productivity can be low [7]. An ordinal scale is sufficient to measure individual experience, but one should also be aware that experience gets out of date.

Many tool vendors claim that their tool can increase productivity, but most times there is no consensus behind it [7]. In fact a tool used in the wrong way might lower the productivity. In cost estimation models the use of tools is often measured on a binary nominal scale as used, or not used. Reasons to measure use of tools are to relate it to other project variables and increase the understanding of the whole project.

4.7 Software cost estimation

Software cost estimation is the process of predicting the effort needed to develop a software system. It does not only involve prediction of the effort, but perhaps also size, project duration, and cost. Software cost estimation is important because it affects the whole project planning. An underestimated project will lead to under-staffing which might cause staff burnout, and setting too short a schedule might lead to missed deadlines. To overestimate a project is as serious, because it leads to lost opportunities and wasted resources. [23, 24]

4.7.1 Predictions

Predictions are hard to make so one cannot expect them to be exact. A prediction can either be a point estimate, that is one value, or an interval estimate that is a lower and upper bound. For an ongoing or completed project there are several ways to compare the actual values with the

predictions. Below are five ways to calculate the estimation accuracy. A is the actual and E is the estimated value.

Relative error $$RE = \frac{A - E}{A}$$

Magnitude of the relative error $$MRE = |RE|$$

Mean magnitude of relative error for n projects $$MMRE = \frac{1}{n}\sum_{i=1}^{n} MRE_i$$

Median magnitude of relative error for n projects $$MdMRE = MRE_{median(n)}$$

Prediction quality, where n is a number of projects, k is the number of them whose mean magnitude of relative error is less than or equal to q $$PRED(q) = \frac{k}{n}$$

It is suggested that a model is acceptable if *PRED (0.25) = 0.75*, which means that 75 % of the projects should have a mean magnitude relative error that is less than or equal to 25 %. Unfortunately most models are insufficient according to this criterion. [7]

4.7.2 Algorithmic and non-algorithmic methods

There are two types of cost estimation methods: algorithmic and non-algorithmic. Below follows a description of models that are algorithmic, and expert opinion, analogy, and decomposition that are non-algorithmic. Expert opinion is a widely used method even though it is not repeatable, which is a big drawback. [24]

- *Expert opinion*. A manager or developer describes a proposed project, and an expert makes predictions based on past experience. The person making the estimate can use a tool, model, or method but that is not visible to the requester. The quality of the estimates is based on the estimator's experience.
- *Analogy*. A proposed project is compared with past projects to identify the similarities and differences. The person or persons estimating use these facts to come up with an effort estimate for the proposed project. This approach makes the estimators analyse the characteristics of the projects that may lead to an underlying model of the effort. The analysis is most likely documented, so it is possible to use again.
- *Decomposition*. Decomposition is a thorough analysis of the characteristics that affect the project. The analysis is either based on the product or the activities needed to build it. If it is based on the product it is decomposed into its smallest parts, and the estimators estimate the effort for each part. If the analysis is based on the activities, they are decomposed into smallest possible, and then the estimators estimate the effort for each activity. In both cases the small estimates are combined in some way to produce an estimate for the whole project.
- *Models*. Models are built on a mathematical formula and it is described further in 4.7.3.

Pricing to win is a somewhat different method that should be mentioned. With this method the cost estimation is whatever the customer has to spend, and the effort also depends on the budget and not on the functionality of the software.

4.7.3 Models

There are many models for software cost estimation, and they are either based on data from previous projects or global assumptions like the rate at which a developer solves a problem and the number of problems available [23, 24].

Most organisations prefer decomposition and modelling to expert opinion and analogy [7]. This is due to the fact that the purpose of making predictions is not only to provide an estimate but also to provide a process of how to make accurate predictions. An organisation can use a model and adjust it over time so it makes more accurate estimations.

Regression model

Early attempts to find a cost model were based on regression. The engineers tried to find a relationship between the measured attributes that could be described by a formula. They plotted the logarithm of project effort (person months) on the y-axis and the logarithm of the size (thousand lines of code) on the x-axis. The equation $\log E = \log a + b \log S$ is linear in the log-log domain and it corresponds to $E = aS^b$ in the real domain. If this model is perfect all the plotted points lie on a straight line, but this is never the case. Instead the next step is to find other factors (for example experience and type of application) that make the predictions differ from the actual value. These factors are added to the model so the final equation is $E = (aS^b)*F$, where F is the adjustment factor weighting the model according to other factors.

COCOMO

Barry Boehm developed the *constructive cost model* (COCOMO or COCOMO 81) in the 1970s, and he was the first one to look at software engineering from an economic point of view. COCOMO is an empirical model and it was derived through analysis of large amounts of data from completed software projects. Later on Boehm and his colleagues revised the model into COCOMO 2.0 that is better suited for recent technology.

COCOMO 81 consists of three different models, a basic model for early prototyping, an intermediate model for early design, and a detailed model when design is complete. To estimate effort the model uses $E = aS^b F$, where E is effort in person months, S is code size in thousand delivered source instructions, *KDSI*, and F is a correction factor. The parameters a and b depend on the complexity of the project.

Project complexity	a	b
Simple	2.4	1.05
Moderate	3.0	1.12
Embedded	3.6	1.2

Table 4.7 Parameters for effort estimation for the basic model. [7]

The correction factor F is not applied (it is 1) for the basic model because very little is known about the project. For the intermediate model the correction factor depends on an ordinal rating of fifteen cost drivers that is attributes of product, process, and resources. The values of each cost driver must be adjusted to fit the specific organisation. In the detailed model the intermediate model is applied at a component level, and the costs drivers are rated at each step (analysis, design, etc.) in the development process.

To estimate duration COCOMO uses the model $D = aE^b$, where D is duration in months, E is estimated person months, a and b are parameters depending on project complexity (see table). COCOMO assumes that the waterfall process is used and that most of the code is developed from scratch.

Project complexity	a	b
Simple	2.5	0.38
Moderate	2.5	0.35
Embedded	2.5	0.32

Table 4.8 Parameters for duration estimation. [7]

COCOMO 2.0

COCOMO 2.0 consists of three levels that reflect the development process: the early prototyping, the early design, and the post-architecture level.

In the early prototyping too little is known about the project to estimate size in lines of code, instead COCOMO 2.0 estimates size in object points. When the size in object points is known the productivity is estimated with help of table 4.9 below. The productivity estimate is the mean of the subjective rating of the developer's experience and capability and the CASE maturity and capability [3]. The effort is the estimated number of object points divided by the estimated productivity (NOP/month).

Developer's experience and capability	Very low	Low	Nominal	High	Very high
CASE maturity and capability	Very low	Low	Nominal	High	Very high
Productivity (NOP/month)	4	7	13	25	50

Table 4.9 Object point productivity. [7]

The estimations at the early design level use function points as size measure. Function points are derived from the requirements so they are more descriptive than the object points, and can therefore make a better prediction. The equation in use here is the same as in COCOMO 81, $E = aS^b F$. The size in this equation is thousand delivered source instructions (KDSI), and therefore the function points are converted to this according to an empirical table. At this level Boehm proposes a = 2.5. The exponent b depends on ratings from extra high to very low of the following five factors: precedentedness, development flexibility, architecture/risk resolution, team cohesion, process maturity. The correction factor F is the rating from 1 to 6 of seven cost drivers that are product reliability and complexity, reuse required, platform difficulty, personnel capability, personnel experience, schedule, and support facilities. There is an additional term in the formula when a significant part of the code is automatically generated, and the detail about this is for example available in [3].

The estimations at the post architecture level are based on the same equation as at the early design level. When this level is later during the development the size estimates should be more accurate, and another difference compared to the early design level is that there are 17 cost drivers instead of 7, see appendix A. The size at this model should be adjusted with the following equation: $ESLOC = ASLOC * (AA + SU + 0.4DM + 0.3CM + 0.3IM)/100$. ESLOC is equivalent number of lines of new code, ASLOC is the number of lines of reusable code which must be modified, DM is the percentage of design modified, CM is the percentage of the code that is modified and IM is the percentage of the original integration effort required for integrating the reused software. SU is a factor reflecting the cost of understanding the code, and AA reflects the initial assessment costs deciding if the software may be reused. [3]

Software life cycle model

Putnam's [8] work that lead to the software equation, see section 4.6.2, is based on the assumption that the manpower build-up and the effort can fit a Raleigh distribution. He formulated the following equation about the relationship between the manpower build-up and the effort:

$Total\ Effort = D * Time^3$,

where total effort is the total effort to the end of the project, D is a manpower build-up parameter that ranges from 8 (entirely new software with many interfaces) to 27 (rebuilt software), and time is the duration of the project in months. The software equation and the manpower build-up parameter can be used for estimations and they are then rearranged as:

$$(Effort/B)^{(1/3)} * Time^{(4/3)} = (SLOC)/ (Process\ productivity\ parameter)$$

$$(Total\ effort)/Time^{3} = Manpower\ build\text{-}up\ parameter$$

The manpower build-up parameter and productivity parameter are calculated from previous projects. SLOC exists as an estimation of the size and the two equations are used simultaneously to estimate the effort and time. This model is called the software life cycle model (SLIM), and it is widely used in practise.

Problem with models

Problems with models like COCOMO and SLIM is that they rely on an estimation of size, which is not available early in the life cycle. COCOMO and SLIM are general models which make them overly complex because they need adjustment factors when they do not rely on local calibrations. Applications show that the cost drivers in COCOMO do not always improve the estimates [7].

4.7.4 Estimation process

A general estimation process for entirely new projects is visualised in figure 4.4. The process starts with the collection of the initial requirements, and all other documents that are available. At an early stage there is not much information available so it is important to communicate the uncertainty in the estimate. The second step is to estimate the product size, and this is usually done by analogy, expert judgement or an algorithmic approach like function points. The estimation should be compared and validated against previous projects. Then follows the estimations of effort and schedule, and these are often highly dependent on the size estimation. The best way to convert the size estimate to effort is to see if any previous project is of the estimated size. This of course depends on if the organisation has been documenting projects, and if any project is of similar size. The other option is to use an algorithmic approach like COCOMO or SLIM.

At scheduling it is also advisable to use data from previous projects, and if this is not possible the alternative is to use a model. A rule of thumb for schedule estimation is [23]

$$Schedule\ in\ months = 3 * (effort\text{-}\ months)^{1/3}$$

This is a very rough estimate and the 3 can be anything between 2 and 4 and it has to be tried out for the current organisation. But this rule is not really much help because one needs to know or estimate the effort, which is as hard as to estimate the duration. The cost estimation depends on how the organisation makes their estimations and again data from previous projects can help.

When the estimates are approved the development can start. During the development the estimations should be re-estimated to get more accurate values. When the development is over the actual values of the size, effort, schedule, and cost should be analysed and compared against the estimated values. These empirical conclusions should be used when estimating new projects and they are essential to be able to improve the estimations.

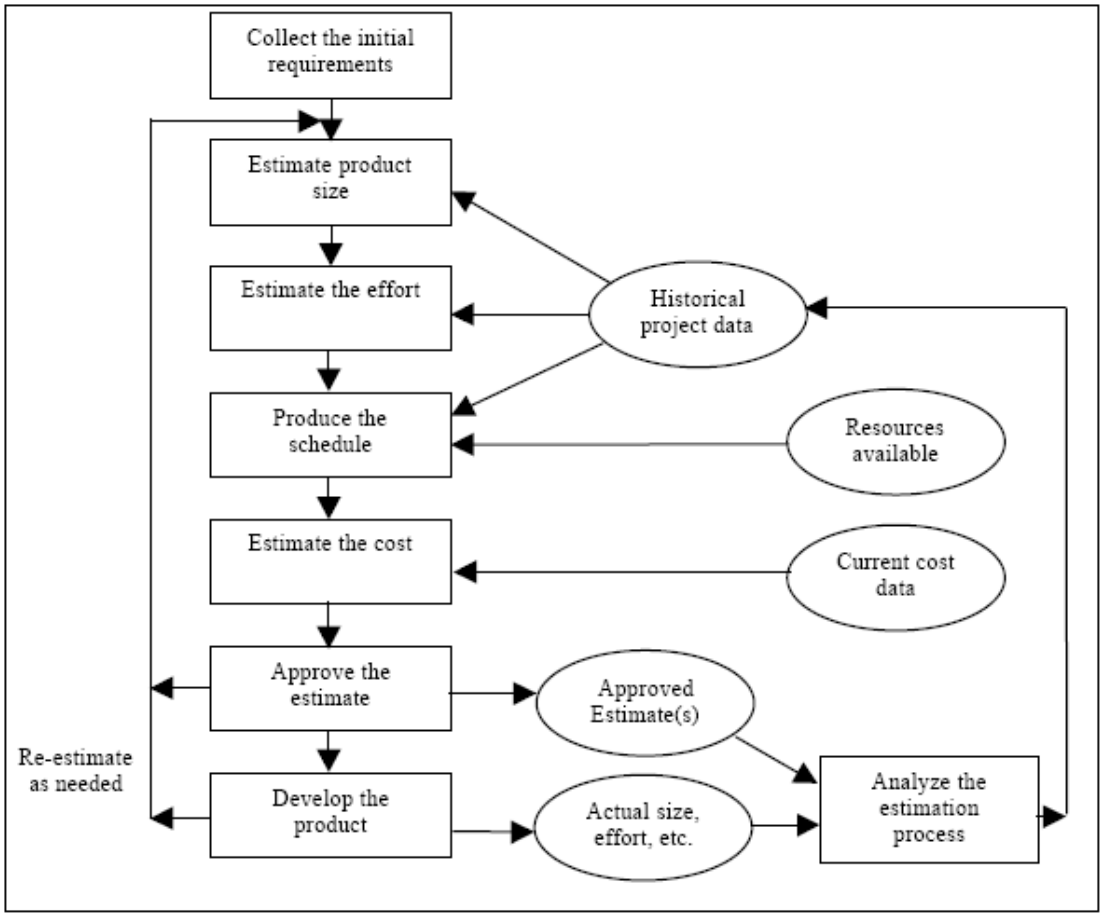

Figure 4.4 A general project estimation process [23]

4.7.5 Maintenance and enhancement projects

The software industry does more evolution than completely new development. Almost all maintenance projects are a combination of new development and adaptation of existing code. The general estimation process described above is designed for completely new development. To estimate evolution projects there are special issues to consider. To insert new functionality requires that the architecture is designed for it. If not, the size estimations are hard to do and the estimates can become very inaccurate. The common thing to do in maintenance projects is to have an experienced individual to estimate the effort by analogy. Most schedule and effort estimation models are based on that projects are developed from scratch so using these models tends to cause over-estimations. [23]

4.7.6 Problems with estimations

Estimation of projects is necessary but it is one of the most difficult software development activities. Here are some explanations to why it is so hard [23]:

- The best way to generate accurate estimations is to use data from previous projects. The problem is that most organisations do not collect and analyse data from the completed projects.
- To estimate size early in the process is difficult and many organisations skip it and go directly to produce a schedule. This is a problem because if they have not thoroughly thought through what they will build, the schedule will not be as good and it is hard to evaluate a scoop change.

- Managers and customers often want a schedule that is unrealistic, so the developers must make clear what is possible to do in a given period of time. An unrealistic schedule might succeed but the risk for a slippage is substantial.
- Many customers demand an exact date when the development should be finished. The estimation is still a guessing so it should rather state a time period when the project will be complete. If a date is needed, it should include the probability, for example it is an 85 % probability that the project will finish on or before May 15.

4.7.7 Dealing with problems

There are several ways to improve the accuracy of estimations. The most important way to improve a particular environment is to use measures of size and effort that are defined consistently across the environment [7]. The model used should be calibrated for the specific environment, and this is a periodic activity to keep the model up to date. Subjectivity is a contributing factor that may distort the accuracy, but it is many times necessary. Locally developed models appear to have higher accuracy than general ones [7]. One reason for this is that they often need less adjustment factors, which are subjective.

Local cost models
Tom DeMarco [7] suggests the following five steps when constructing a local cost model:

1. Decide which activities that are part of the project cost estimation. Some organisations chose to estimate only the work after a complete specification exists, while other includes the requirements elicitation and specification. Maintenance activities are also included or not depending on the case.
2. Formulate the cost model, and describe how the inputs relate to the outputs. If there are several models, an overall model has to explain how they relate to each other. Local cost models often include cost drivers and adjustment factors, but also the effort spent on completed tasks. The reason for this is that empirical results show that effort spent on completed tasks correlate with the remaining effort in the project.
3. To validate the model it is necessary to check it with data from past projects in the specific environment. To use data from completed projects that had no formal collection process might be risky. The data can be biased or inaccurate. To test a model theoretically it is necessary to have n+2 past projects, where n is the number of input variables. In practise this is not enough, but 7-10 for each input is necessary.
4. Analyse the data with regression and test the significance of the results.
5. Check if the model estimates acceptable values with mean magnitude of relative error, and prediction quality. If the model lacks accuracy it has to be re-evaluated thoroughly. This last step is very important not only to validate the model but also to reflect changes in process, resources, and techniques.

4.8 Measurement in practice

Measurement theory is not really useful without real numbers, and those numbers have to be collected somehow, somewhere, and by somebody.

4.8.1 Good data

A measurement might be well defined and meaningful, but the data must still be good to give good results. To be good, the data should be:

1. *Correct*. The data should be collected according to the definition of the metric.
2. *Accurate*. The data should correspond to the actual values.
3. *Appropriately precise*. The precision used should be reasonable.
4. *Consistent*. The result should not differ much when different devices or persons are measuring.

5. *Time-stamped*. If the data is associated with a time period or activity it should be time-stamped, to be able to know exact when they were collected.
6. *Replicable*. The data, project history, and results should be stored so the measurement can be replicated.

It is important to consider these issues before the data collection begins, for example what precision to use and during which activities to collect the data.

4.8.2 The collection process

There are two kinds of data: raw and refined. Raw data is the result from the data collection, while refined data is the extraction from the raw data. Collecting raw data can be done either automatically or manually. It is advisable to do as much as possible automatically but often there is no other option than to do it manually. The collection process should be kept as simple as possible otherwise the programmers, testers, managers, or whoever is collecting the data will not do it properly. For the same reason one should avoid unnecessary collecting. To train and motivate the staff to do the data gathering is also a part of the collection process. The motivation of the staff can be as simple as to provide the analysis of the results so they can see how it can help them in their work. The goal should also be to make the collecting a natural part of the development process, and not something that is considered as extra work.

4.8.3 Best practices of software metrics

In [14] Kulik and Weber present a survey about best practices regarding software metrics. The survey covers over 500 people from mainly North America, Western Europe and India. The results show that the most common used metrics are schedule (59% of participants use them), requirements (49%), lines of code (49%), and fault density (39%). The most popular tool is without question Microsoft Excel that more than 60 % of the organisations use. Other tools are RationalSuite, ClearQuest, and QSM SLIM. One notable thing about the tools is that none of them had significant better user satisfaction than 3 on a scale from 1 to 5. The most significant benefits (more than 50% or more of total respondents) with metrics are more predictable schedule, better ability to understand the project schedule, better estimates of size and cost, and improved communication with management.

5 State of practice at HP OpenView Amsterdam

The software measurement case study is performed at HP OpenView in Amsterdam and in the team Special Products Engineering (SPE). The organizational structure in Amsterdam is that there are three different teams developing the product Service Desk. The three teams are Special Products Engineering (SPE), Current Products Engineering (CPE), and Future Products Engineering (FPE). When referring to *team* or *the team* it means the SPE team. The SPE team is developing special versions of the product for customers and the work is performed in projects. The projects are dividing the team into smaller units and the current practise is that there is one SPE manager and two project leaders. The SPE manager does not manage any project directly but that does the project leaders. Most developers work in projects with a project leader but there are also a few developers that manage their own project.

Below follows descriptions of the SPE team's state of practise in process, metrics, development, and tools.

5.1 Process

There exists an organisational process but it is extensive and the team has not followed it in the previous projects. Currently the team is tailoring the organisational process so it better will the suit their environment. According to the project schedules the previous projects had a structure like below:

1. Project setup.
2. Analysis.
    a. Requirements.
    b. External specification.
3. Design
    a. Functional specification.
    b. Technical specification.
4. Implementation.
    a. Implement feature 1.
    b. Peer review.
    c. Unit test.
    d. Fix code.
    e. Implement feature 2.
    f. Etc.
5. Integration test.
    a. Build package.
    b. Test.
    c. Fix code.
    d. Retest.
6. User acceptance test.
7. Final fixes, build, and test.
8. Delivery.

There are differences between projects and for some projects the delivery is divided into several drops. This means that when some feature is implemented it is integrated and tested and then delivered to the customer for user acceptance test, and at the same time the implementation continues with other features.

The fact that no process is defined makes it harder to do measurement. Some measurement is impossible because of differences between the processes. To define a process would also create more consistency of the projects within the team. It is not necessary for all projects to

adhere to one process, but it would make sense to create a generic one and then tailor it for different customer projects.

5.2 Metrics

The team has not collected any data with the purpose of using it for evaluation. The documents that exist differ slightly from project to project, but what exist in most projects are: requirements, external specification, functional specification, technical specification, project schedule, and test report. Source code exists obviously for all projects. More about what measures that exist will follow in the gap analysis.

5.2.1 Cost estimation

The current cost estimation technique is expert judgement. The project manager takes help from an experienced developer or designer that returns a duration estimate in number of hours, days, or weeks. When a project is started the project schedule is constructed and it contains more detailed estimations of different activities. The schedule contains who works on an activity and the duration of it. The project schedule is not followed slavishly, and the order when the activities are performed might change but this is normally not documented.

5.3 Development

The evolution of Service Desk and the projects completed by the SPE team is not straightforward. At several occasions code from an SPE project has been merged into a service pack of the main product, and code from service packs (or the future product) is often used in the SPE projects. This makes it very hard to find out how much that actually was developed during some of the projects. The main development is done in Java and SQL. The Java-files are written in a standard way, but the SQL-files can be generated.

5.4 Tools

ClearCase is used to for version control of code and other documents. The project schedules are created with Microsoft Project. Other documents are mainly produced with Excel and Word. Requirements management is done with a tool called Caliber and Service Desk is used for bug tracking.

5.4.1 SOME

SOME is a static code analyser to provide source code measures for any defined product or component [32]. The analyser mainly provides size measurements as NCSS, PLOC, comments, and blanks, but also cyclomatic complexity (McCabe's). This complexity measures the complexity of entire files, but it should rather measure the complexity of methods. It is also possible to measure turmoil between two code bases, or start and finish of projects. The turmoil reports additionally include: new, modified, deleted, files and lines of code, total turmoil and turmoil rate. All the reports are separated in Java, C, C++ files and other files. With help from the administrator it is possible to customise settings as what file-types to count and to exclude directories.

Definitions
SOME measures the code according to the following definitions:

NCSS: all lines that not are comments, blanks, or a standalone brace or parenthesis.
PLOC: all physical lines.
Blanks: all lines that only contain a new line character.
Comments: the number of lines that start with # or are surrounded by /* */.

Cycl.Compl.: the degree of logical branching within a function. Logical branching is when while, for, if, case, or goto appears in a function.

Limitations
There are some limitations with SOME that makes it hard to get some desired values. All the projects are maintenance projects so the turmoil measure has to be used. Figure 5.1 shows how SOME reports turmoil for Java, C, C++ files, and figure 5.2 for non Java, C, C++ files.

	NCSS	PLOC	Blanks	Comments	Cycl.Compl.	New lines	Mod.lines	Del.lines	Turmoil
Totals	5306	16861	2533	8071	680	1508	182	136	1826

Figure 5.1 Turmoil report for Java, C, C++ files.

	PLOC	Blanks	Comments	New lines	Mod.lines	Del.lines	Turmoil
Totals	3433	238	28	3429	2		3431

Figure 5.2 Turmoil report for non Java, C, C++ files.

NCSS counts the total number of NCSS from files that have been added, modified, and deleted. This is a problem because what really is desired is to know how many of the new, modified, and deleted lines that are NCSS. The similar problem exists for non Java, C, C++ files where the desire is to know the PLOC, blanks, and comments for new, modified, and deleted files respectively, but it only reports PLOC, blanks, and comments in total. Another thing to think about is that if a file is moved during the project it is reported as deleted and added.

5.4.2 Service Desk

Service Desk is a tool mainly used to manage organisations' IT equipment, and the idea is to have a structured process to solve problems, or with other words how to deliver a service. A process can start with that a client reports about an issue, which implicitly is a request for a service. The organisation starts to investigate the issue and soon they have to answer questions like: is this a known error or is it a new problem, can we solve it, if we solve it how do we implement the change, and when is the issue solved? ITIL (IT Infrastructure Library) is a framework for how to manage IT services (which Service Desk is compliant with) and it describes how to work with issue like this (for more about ITIL see for example [34]).

The product is structured into four main sections: Service Desk, Organisation, CMDB, and SLM. Service level management, SLM, is where service agreements with customers are defined. CMDB means configuration management database and here all configuration items are defined. Organisation describes all people in the organisation, how it is structured, and it is also possible to define workgroups. The part called Service Desk is the major part of the product and it consists of service calls, incidents, problems, changes, and work orders. Service calls and incidents are similar but with the difference that incidents are automatically generated by a system when a person reports service calls. Service calls and incidents are what above is called an issue.

6 GQM

To decide what to measure from the completed projects the goal question metric approach is applied and iterated. The iteration is done G-Q-M-Q-G-Q-M but only the result after the two M steps are shown. The complete iteration is available in appendix B. All questions are in present tense even though some of them only can be measured on completed projects.

6.1 Iteration 1

After the first iteration G-Q-M the goals, questions, and metrics are as below. To make it clear both the direct and indirect metrics are explicitly stated. Duplicate metrics exist, and this will be considered later on.

Goal 1. **Improve the accuracy of the estimations**.

Q1. *How accurate are the project estimations (cost, effort, duration)?*

- M1. Actual effort of the project.
- M2. Estimated effort of the project.
- M3. Actual cost of the project.
- M4. Estimated cost of the project.
- M5. Actual duration of the project.
- M6. Estimated duration of the project.
- M7. Magnitude of relative error of effort.
- M8. Magnitude of relative error of cost.
- M9. Magnitude of relative error of duration.

Q2. *How accurate are the effort and duration estimations of the activities?*

- M10. Actual effort for each activity.
- M11. Estimated effort for each activity.
- M12. Actual duration for each activity.
- M13. Estimated duration for each activity.
- M14. Magnitude of relative error of the effort for each activity.
- M15. Magnitude of relative error of the duration for each activity.

Q3. *Does the duration of a project affect the accuracy of the total estimations?*

- M16. Actual duration of the project.
- M17. Magnitude of relative error of the duration for the project.

Q4. *What is an acceptable level of accuracy for the total estimations (cost, effort, duration)?*

- M18. Acceptable accuracy for estimation of total duration.
- M19. Acceptable accuracy for estimation of total effort.
- M20. Acceptable accuracy for estimation of total cost.

Q5. *Do the previous projects fit any existing model?*

- M21. Actual size.
- M22. Actual effort.
- M23. Regression of size and effort.

Q6. *How confident is the project manager with the estimations?*

- M24. Confidence rating of the estimations.

Goal 2. **Reduce the risks in the project.**

Q7. *What are the main risks in the project?*

M25. The identified risks.

Q8. *How serious are the risks if they occur?*
M26. The seriousness of the risks.

Q9. *How likely are the risks to occur?*
M27. The likelihood of the risks.

Q10. *What to do to prevent the risks?*
M28. The plan to prevent the risks.

Q11. *Do the developers have the right skills?*
M29. Complexity rating of the project (or for each requirement).
M30. Developers' experience and knowledge.

Goal 3. **Ensure the predictability of the costs in the project.**
Q12. *Are there any unexpected personnel or travelling costs?*
M31. The reason for unexpected costs.
M32. The amount of unexpected costs.

Goal 4. **Ensure the timeliness of the activities in the project.**
Q13. *Which activities do not meet the schedule?*
M33. The type of activities that needed more effort (at least one day) than estimated.

Q14. *Is the total effort on finding and fixing defects under control?*
M34. Effort spent on fixing defects.
M35. Number of issues found during peer review.
M36. Effort spent on peer review.
M37. Effort spent on supporting and maintaining old projects.

Q15. *How many days are the personnel on unplanned vacation or sick on average per month?*
M38. Number of sick days per month.
M39. Number of days on unplanned vacation per month.
M40. Average number of unplanned absence days per month.

Q16. *Do the requirements change during the project?*
M41. Number of initial must requirements.
M42. Number of must requirements added after project start.
M43. Number of completed must requirements.
M44. Volatility of the requirements.

Goal 5. **Analyse the organisational productivity.**
Q17. *What is the organizational productivity according to the process productivity parameter?*
M45. New non-commented source statements (NCSS).
M46. Modified NCSS.
M47. Deleted NCSS.
M48. Duration of the main build phase.
M49. Effort spent on the main build phase.
M50. Process productivity parameter.

6.2 Final iteration

After the iteration G-Q-M-Q-G-Q-M the goal, questions, and metrics are as below. The metrics that are exactly the same have now the same number but because the metrics still are not precisely defined there may be even more that are the same in the end.

Goal 1. **Improve the accuracy of the estimations**.

Q1. *How accurate are the project estimations of duration?*
- M1. Actual duration for the project.
- M2. Estimated duration for the project.
- M3. Magnitude of relative error of the duration.

Q2. *How accurate are the estimations of duration of the activities?*
- M4. Actual duration for each activity.
- M5. Estimated duration for each activity.
- M6. Magnitude of relative error of the duration for each activity.

Q3. *Does the duration of a project affect the accuracy of the total estimations?*
- M1. Actual duration of the project.
- M7. Magnitude of relative error of the duration for the project.

Q4. *Do the previous projects fit any existing model?*
- M8. Actual size.
- M9. Actual effort.
- M10. Regression of size versus effort.

Q5. *How confident is the project manager with the estimations?*
- M11. Confidence rating of the estimations.

Q6. *How big part of total effort is analysis, design, development, and test respectively?*
- M12. Total actual effort.
- M13. Actual effort of analysis, design, development, and test respectively.

Q7. *Does a project plan exist?*
- M14. If a proper project plan exist or not.

Goal 2. **Ensure the predictability of the costs in the project.**

Q8. *Are there any unexpected personnel or travelling costs?*
- M15. The reason for unexpected costs.
- M16. The amount of unexpected costs.

Q9. *How large are the total costs in the project?*
- M17. Total costs of personnel and travel expenses.

Goal 3. **Ensure the timeliness of the activities in the project.**

Q10. *Which activities do not meet the schedule?*
- M18. The type of activities that needed more effort (at least one day) than estimated.

Q11. *Is the effort spent on finding and fixing failures under control?*
- M19. Effort spent on fixing defects.
- M20. Number of issues found during peer review.

M21. Effort spent on peer review.
M22. Effort spent on supporting and maintaining old projects.
M23. Number of old defects fixed during the project.

Q12. *How many days are the personnel on unplanned vacation or sick on average per month?*
M24. Number of sick days per month.
M25. Number of days on unplanned vacation per month.
M26. Average number of unplanned absence days per month.

Q13. *Do the requirements change during the project?*
M27. Number of initial must requirements.
M28. Number of must requirements added after project start.
M29. Number of completed must requirements.
M30. Volatility of the requirements.

Q14. *How much time is lost due to network outage and tool problems?*
M31. Number of hours the network was down.
M32. Number of hours tool problems caused disruption in the work.

Q15. *Do the developers know their responsibilities?*
M33. Developers' knowledge about their responsibilities.

Goal 4. **Analyse the organisational productivity.**

Q16. *What is the organizational productivity according to the process productivity parameter?*
M34. New non-commented source statements (NCSS).
M35. Modified NCSS.
M36. Deleted NCSS.
M37. Duration of the main build phase.
M38. Effort spent on the main build phase.
M39. Process productivity parameter.

Goal 5. **Increase the quality of the product.**

Q17. *How many lines of code are the new class files on average?*
M40. Number of new Java class files.
M41. Number of lines of code in the new Java files.
M42. Average number of lines of code in the new Java files.

Q18. *Is the code standard followed?*
M43. Check whether the code standard is followed or not.

Q19. *Is good quality defined for the product?*
M44. Whether it is documented what good quality is.

Q20. *What is the defect density?*
M45. Number of defects reported in service desk during the project.
M46. Total number lines of Java code.
M47. Number of defects per thousand lines of code.

Q21. *Where in the code do the defects originate from?*
M48. Origin in the code of the defects.

Q22. *How many service calls are created during the project?*
M49. Number of service calls reported in service desk for the project.

Q23. *How many NCSS of JUnit tests are created per new class file?*
M50. Non-commented sources statements of JUnit tests.
M41. Number of new Java class files.

Q24. *How many of the unit tests are actually performed?*
M51. Number of unit tests performed.

Goal 6. **Increase the quality of the process.**

Q25. *Is the usage of standards and templates sufficient?*
M52. Number of existing standards in the organisation.
M53. Number of standards used in the project.
M54. Number of existing templates in the team.
M55. Number of templates used in the project.
M56. Developers' knowledge about standards and templates.

Q26. *Is the process visible?*
M57. Number of documents produced internally during the project.
M58. Activities in the process that are invisible.

Q27. *Is the documentation updated and finalised?*
M59. Whether cleanup after the project is scheduled or not.
M60. Number of times the project plan and schedule were updated.
M61. Dates when the project plan and schedule were created and last updated.
M62. Documents that still are in an early version.

Q28. *Is the process how to report defects defined and well understood?*
M63. Whether a document exists that defines the process of reporting defects.
M64. Developers' knowledge about how to report defects.

Q29. *How long are the service calls open on average?*
M50. Number of service calls reported in service desk for the project.
M65. The time closed service calls for the project was open.
M66. Average time the closed service calls were open.

Q30. *During what activity are the defects introduced?*
M67. The activity where the defect is introduced.

Q31. *During what activity are the defects found?*
M68. The activity where the defect is found.

7 Gap analysis

The metrics resulting from the GQM analysis might not be available in the documentation from the previous projects. Here follows a presentation of the completed projects followed by the existing information. Thereafter the gap between what is needed to answer the GQM-questions and the existing information is presented.

7.1 Projects

The completed projects are presented here to give a picture of how they differ and what to expect from the result.

7.1.1 Ascension

Ascension was an internal project and it was a merge of several completed projects plus some new features. The project was collaboration between SPE and Current Products Engineering (CPE). Throughout the project nine people were involved and it ran from September 2003 to March 2004. The setup was that SPE was doing most of the work in the beginning with one CPE person involved, and then should leave it over to CPE. The communication between the teams was not very good so the hand over did not go smooth.

7.1.2 Boigu

Boigu was a small customer project that only included two people. It lasted for about 100 working days and ran from July to November in 2003. One of the members functioned as leader and the other one as developer. The leading person was experienced but it was the first project for the developer, which made an impact.

7.1.3 Curacao and Hitra

Curacao and Hitra were consecutive projects for one customer. Curacao started in May 2003 and the solution was accepted in October the same year. The setup of the project was one project leader and two developers that implemented some new features. The analysis of Hitra started in August 2003, which was before the solution of Curacao was accepted, and the solution was accepted in May 2004. The developers and project leader were the same in both projects, and the only difference was that a tester was doing the integration testing in Hitra. Curacao was the first project with this customer, which may have affected the work. Both project leader and developers were experienced in both projects. Hitra was a bit affected by Ascension that got priority and pulled out some resources. After the projects the leader and developers were responsible for support of the solution, which affected subsequent projects.

7.1.4 Mathilde

Four Mathilde projects are included in this report and they are for one and the same customer. The first two projects were completed before the SPE team existed. Mathilde 1 and 2 as they are called ran somewhat intertwined from October 2002 to May 2003. Seven people were involved to a larger extent but the number gets even higher if even minor roles are counted. The people involved were all experienced in this setting.

Mathilde 3 started in November 2003 and it ran to around April 2004. The setup was one project leader, 3-4 developers, and one tester. Mathilde 4 started in February 2004 and the last delivery was in June 2004. The setup was similar as in Mathilde 3. Mathilde 3 developed new features, and Mathilde 4 was a big merge plus some new features. In Mathilde 4 the leader was inexperienced and some of the developers were rather new to the product, which affected the work. The experience with this customer was also not extensive.

7.1.5 Venus

Venus was a customer project and it started in June 2004 and it finished in October the same year. The project involved two experienced developers and did not have any formal management. It was the first project with this customer. A previous project for another customer did require some support activities, which made an impact.

7.2 Existing information

The existing information is limiting the possibilities to derive the metrics. The following metrics are possible to derive, or recreate from the existing information. All information does not exist for all projects, and some of the metrics need modifications to be available.

M1. Duration of the project.
M2. Estimated duration of the project.
M3. Magnitude of relative error of duration.
M5. Estimated duration of the activities.
M8. Actual size.
M9. Actual effort of the project.
M10. Regression of size versus effort.
M13. Actual effort of analysis, design, development, and test respectively.
M14. If a project plan exist or not.
M17. Total costs of personnel and travel expenses.
M21. Effort spent on peer review.
M29. Number of completed must requirements.
M34. New NCSS.
M35. Modified NCSS.
M36. Deleted NCSS.
M37. Duration of the main build phase.
M38. Effort spent on the main build phase.
M39. Process productivity parameter.
M40. Number of new Java class files.
M41. Number of lines of code in the new Java files.
M42. Average number of lines of code in the new Java files.
M43. Check whether the code standard is followed.
M44. Whether it is documented what good quality is.
M45. Number of defects reported in service desk during the project.
M46. Total number of lines of code in the project.
M47. Number of defects per thousand lines of code.
M50. Number of service calls reported in service desk during the project.
M51. Non-commented sources statements of JUnit tests.
M57. Number of documents produced internally during the project.
M58. Activities in the process that are invisible.
M59. Whether cleanup after the project is scheduled or not.
M60. Number of times the project plan and schedule were updated.
M61. Dates when the project plan and schedule were created and latest updated.
M62. Documents that still are in an early version.
M63. Whether a document exists that defines the process of reporting defects.
M65. The time closed service calls for the project were open.
M66. Average time the closed service calls were open.

7.3 Gap between existing and wanted information

The missing information to answer each question is presented below. Most of the information from the previous projects lacks in reliability and accuracy but this gap analysis is only concerned with if the metrics are collected or not. The gap is regarding completed projects and it does therefore not perfectly reflect what the team is collecting in ongoing projects.

Goal 1. **Improve the accuracy of the estimations**.

Gap Q2.	M4.	Actual values of duration of the activities.
	M6.	Magnitude of relative error of duration.
Gap Q5.	M11.	Confidence rating of the estimations.

Goal 2. **Ensure the predictability of the costs in the project**.

Gap Q8.	M15.	The reason for unexpected costs.
	M16.	The amount of unexpected costs.

Goal 3. **Ensure the timeliness of the activities in the project.**

Gap Q10.	M18.	The type of activities that needed more effort than estimated.
Gap Q11.	M19.	Effort spent on finding defects.
	M20.	Number of defects found during peer review.
	M22.	Effort spent on supporting and maintaining old projects.
	M23.	Number of old defects fixed during the project.
Gap Q12.	M24.	Number of sick days per month.
	M25.	Number of days on unplanned vacation per month.
	M26.	Average number of unplanned absence days per month.
Gap Q13.	M27.	Number of initial must requirements.
	M28.	Number of must requirements added after project start.
	M30.	Volatility of the requirements.
Gap Q14.	M31.	Number of hours the network was down.
	M33.	Number of hours tool problems caused disruption in the work.
Gap Q15.	M33.	Developers' knowledge about their responsibilities.

Goal 4. **Analyse the productivity of the organisation in the projects.**

When number lines of code are used instead of non-commented source statements there are no gaps.

Goal 5. **Increase the quality of the product.**

Gap Q21.	M47.	Origin in the code of the defects.
Gap Q24.	M51.	Number of unit tests performed.

Goal 6. **Increase the quality of the process.**

Gap Q25.	M52.	Number of existing standards in the organisation.
	M53.	Number of standards used in the project.
	M54.	Number of existing templates in the team.
	M55.	Number of templates used in the project.

	M56.	Developers' knowledge about standards and templates.
Gap Q28.	M64.	Developers' knowledge about how to report defects.
Gap Q30.	M67.	The activity where the defect is introduced.
Gap Q31.	M68.	The activity where the defect is found.

7.4 Special concerns

Some of the metrics are special in the way that the information exists but it is not interesting for completed projects. Other metrics cannot be measured on completed projects, but they are interesting to measure as of today. There are also metrics that might not be interesting to measure more than on the completed projects, and this will be discussed later.

7.4.1 Existing information

The following metrics exist but are not interesting.

M17. Total costs of personnel and travel expenses.
The cost is not interesting to recreate for previous projects because there are no estimations to compare to, and to compare to the actual funding is not possible.

M29. Number of completed must requirements.
It is not perfectly clear which the initial requirements were which makes this metric not interesting.

M63. Whether a document exists that defines the process of reporting defects.
This metric is rather generic and is not interesting for the completed projects, but it is interesting as of today.

7.4.2 Non-existing information

These metrics cannot be measured on the completed projects but are still interesting.

M52. Number of existing standards in the organisation.
If there are any standards in the organisation they are well hidden, and therefore this is interesting.

M54. Number of existing templates in the team.
If it gets more clear which templates that exist it can help in coming projects.

M64. Developers' knowledge about how to report defects.
This says something about the completed projects as well, because many of the developers are still here, but this is only needed if the answer to M63 is positive.

7.5 Questions possible and interesting to answer

Table 7.1 below describes the questions interesting and possible to answer for the projects. Mathilde 1 and Mathilde 2 are not considered for all questions because they are mainly used for comparison. In the end it may be that information for some projects is not good enough to answer a question. Boxes marked with a dash are not project dependant.

	Ascension	Boigu	Curacao	Hitra	Mathilde 1	Mathilde 2	Mathilde 3	Mathilde 4	Venus
Goal 1. Improve the accuracy of the estimations.									
Q1. How accurate are the estimations of duration?	X		X	X				X	
Q3. Does the duration of a project affect the estimations?	X		X	X				X	
Q4. Do the previous projects fit any existing model?	X	X	X	X	X	X	X	X	X
Q6. How big part of total effort is analysis, design, development, and test respectively?		X	X	X					
Q7. Does a project plan exist?	X	X	X	X	X	X	X	X	X
Goal 4. Analyse the organisational productivity.									
Q16. What is the organisational productivity according to the process productivity parameter?	X	X	X	X	X	X	X	X	X
Goal 5. Increase the quality of the product.									
Q17. How many lines of code are the new lines on average?	X	X	X	X			X	X	X
Q18. Is the code standard followed?	X	X	X	X	X	X	X	X	X
Q19. Is good quality defined for the product?	-	-	-	-	-	-	-	-	-
Q20. What is the defect density?	X		X	X	X	X	X	X	
Q22. How many service calls are created during the project?	X		X	X	X	X	X	X	
Q23. How many NCSS of JUnit test are created per new class?	X	X	X	X	X	X	X	X	X
Goal 6. Increase the quality of the product.									
Q25. Is the usage of standards and templates sufficient?	-	-	-	-	-	-	-	-	-
Q26. Is the process visible?	X	X	X	X			X	X	X
Q27. Is the documentation updated and finalised?	X	X	X	X			X	X	X
Q28. Is the process how to report defects defined and well understood?	-	-	-	-	-	-	-	-	-
Q29. How long are the service calls open on average?	X		X	X	X	X	X	X	

Table 7.1 Questions possible and interesting to answer for the projects.

8 Result

In this section the results are presented, and they are presented according to their goals and questions. Most questions include a comment from the SPE manager about the results. The SPE manager was not directly managing the projects but was at one level up although with a good insight of what was going on. For some questions all projects are presented even though result not exists, or it is a result that no information exists. The measurement methods are described in appendix C.

8.1 Accuracy of the estimations (Goal 1)

8.1.1 How accurate are the estimations of duration (Q1)?

The following table describes the accuracy of the estimations.

Duration in weeks	First duration estimate	Actual duration	MRE first estimate
Ascension	22	26	0.15
Curacao	22	24	0.08
Hitra	37	40	0.08
Mathilde 4	12	18	0.33
Average			0.16

Table 8.1 Accuracy of the estimations.

Does this agree with your feeling and what does it tell you?

It is about as expected, but I thought Hitra would be better than Curacao when Curacao was the first customer project for the SPE team. This tells that we are not doing a good job because no project finished on time, and Mathilde 4 was really bad. When this project started I had a rather naive vision of the work here. I thought it was possible to improve but realize now that it is not, or at least not in the way I would like to. The design and coding are many times done in parallel, which makes it almost impossible to make the schedules reliable. We have now started to define a process and to have a rather ambitious approach in one project and we will see how it turns out. Hopefully will a better process result in a more reliable schedule.

8.1.2 Does the duration of a project affect the accuracy of the estimations (Q3)?

Figure 9.1 shows MRE of duration against the duration in years.

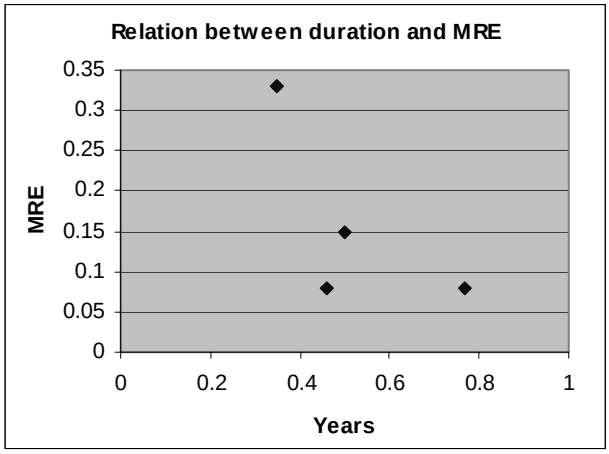

Figure 8.1 Relation between duration and MRE of duration.

Does this agree with your experience of what affect estimations and what does it tell you?
Yes. The points can be thought to fit a 1/x line and that is in line with my experience. In short projects there is no time to adjust when things go a little wrong. When projects last longer it is easier to "hide" problems, and when a customer makes changes it is easier to extend the project more than actually is needed.

8.1.3 Do the previous projects fit any existing model (Q4)?

Regression of size versus effort gives results as in the figures below.

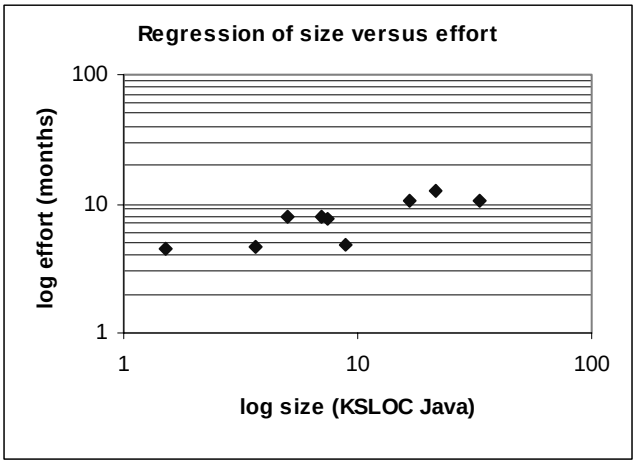

Figure 8.2 Regression of effort versus size.

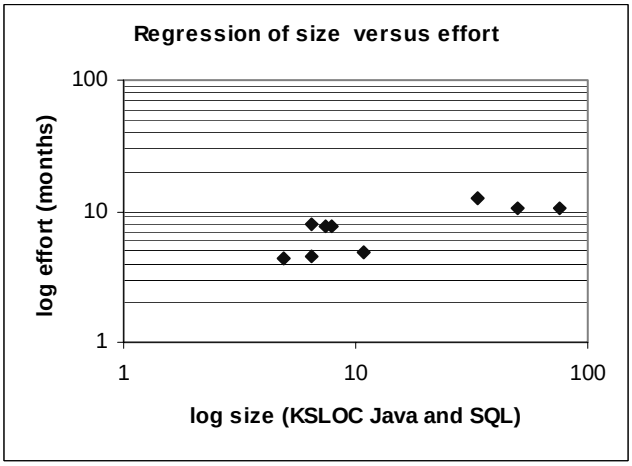

Figure 8.3 Regression of effort versus size.

Is this something you would expect and is it useful?
It does not tell me much. The developers spend a lot of time trying to understand the code but the number of lines that has to be changed is not that large, and this makes it very hard to do size estimates.

8.1.4 How big part of total effort is analysis, design, development, and test respectively (Q6)?

The figure below shows how big part the activities are of the total effort.

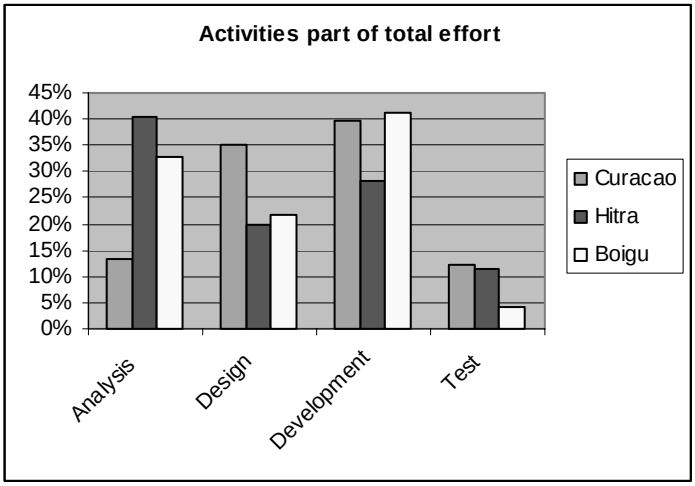

Figure 8.4 Activities part of total effort.

Is this something you would expect and what does it tell you?
Boigu is not really representative when it was so small and also was used to get a developer up to speed. With only two projects it does not tell me much, but it is interesting if the number of data points increases.

8.1.5 Does a project plan exist (Q7)?

None of the previous projects had a project plan.

What does this tell you?
This is of course no new knowledge, but no wonder that the estimates are not that good! We have started to have a project plan in the larger projects, but in projects with only two and three people there is no need to create an extensive project plan.

8.2 Organisational productivity (Goal 4)

8.2.1 What is the organizational productivity according to the process productivity parameter (Q16)?

Table 9.2 shows the result of the process productivity parameter for the projects.

Project	Productivity parameter (Java and SQL)	Productivity parameter (Java)	Productivity index (Java and SQL)	Productivity index (Java)	Average productivity index
Ascension	61916	42631	19	18	19
Boigu	14389	4627	13	8	11
Curacao	10409	9718	12	12	12
Hitra	8549	6648	11	10	11
Mathilde1	30388	24710	16	15	16
Mathilde2	26407	15179	16	13	15
Mathilde3	15504	15047	13	13	13
Mathilde4	105486	18116	21	14	18
Venus	18106	10522	14	12	13
Average SPE			14.7	12.4	13.9

Standard deviation			3.8	3.2	3.3
Average non SPE			16	14	15.5
Standard deviation			0	1.4	0.7

Table 8.2 Process productivity parameter for the projects.

Does this agree with your picture of the projects and what does it tell you?
Yes. I expected Boigu to be low when it trained an inexperienced developer. Mathilde 1 and 2 should be high because the developers in those projects had been working with Service Desk for a long time. In the other projects the people were rather new to the product. Mathilde 4 and Ascension are hard to say anything about when they are merge projects. I thought Hitra would be better. Mathilde 3 is high because a lot of planning was skipped which is ugly but gives high productivity. It is not sure that a high value is good because when things like testing and writing documentation is skipped the productivity gets higher. I consider the productivity value with only Java code as most interesting. It is obvious that the non-SPE projects have a higher value and I expected this, but not that it would be that clear. It is useful to get it on paper and not just have a feeling about it. It is also interesting to have if we want to do this in a year again with more projects.

8.3 Product quality (Goal 5)

8.3.1 How many lines of code are the new files on average (Q17)?

The figures below show the number of PLOC for the different projects.

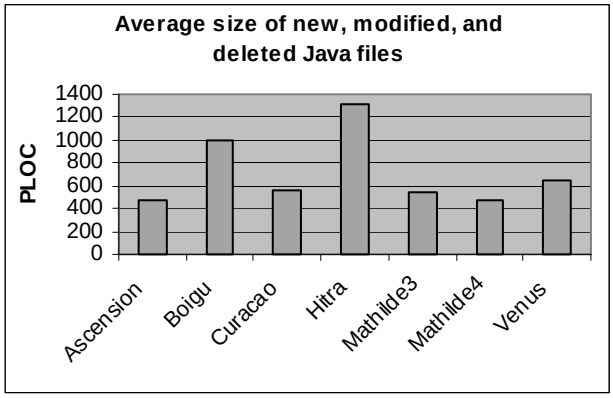

Figure 8.5 Average number of PLOC of the new, modified, and deleted Java files.

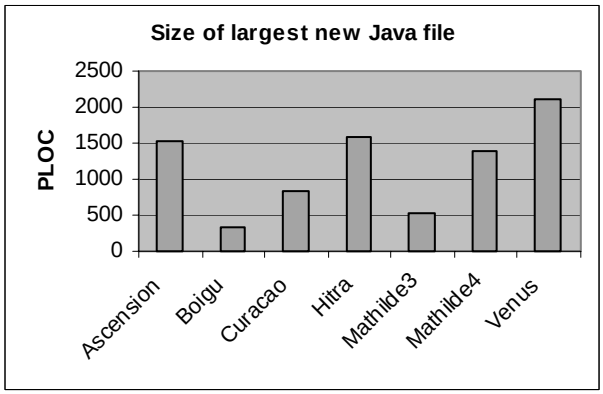

Figure 8.6 Size of largest new Java file in each project.

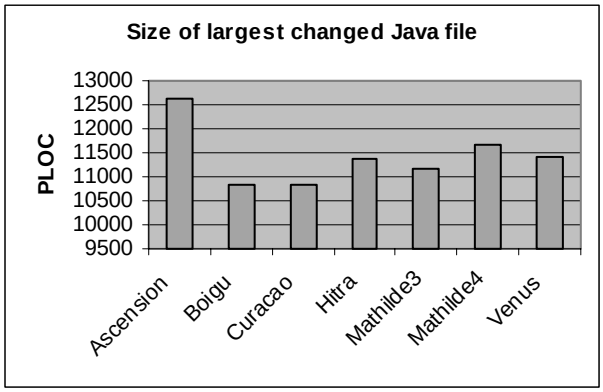

Figure 8.7 Size of largest changed Java file in each project.

Does this agree with your picture of the projects and what does it tell you?
I do not have a good feeling about sizes in Java, but the files should not be too large to be easy to maintain.

8.3.2 Is the code standard followed (Q18)?

The table below summarises the violations to the coding guideline in the new and modified files. X means coding guideline violated. For the exact deviations see appendix E. The truncated headers are indentation, statements, and documentation.

File	Braces	Indent.	Long lines	State.	Blanks	Names	Doc.
Ascension (mod.)	x		x		x		x
Boigu (mod.)							x
Curacao (new)			x		x	x	x
Hitra (new)	x		x		x		x
Mathilde 3 (new)			x				x
Mathilde 4 (new)	x		x			x	x
Venus (new)		x	x				x

Table 8.3 Summary of the deviations from the coding standard.

Was this expected and what does it tell you?
I expected this because the coding standard is not enforced. It looks like they did a good job in Boigu.

8.3.3 Is good quality defined for the product (Q19)?

There is no general quality document. The only document that addresses quality issues is the coding guideline. The guideline does direct or indirect describe the following quality attributes:

- Readability. This should be one objective of the work.
- Reusability. Try as much as possible to reuse packages and design new packages so they easily can be reused.
- Complexity. The complexity of a method or a function should not exceed 10.
- Localizability. Code that is language dependent should be localised.
- Efficiency. First rule of optimization: do not do it. Second rule: do not do it yet, Michael Jackson, Michael Jackson Systems Ltd. If you anyway do optimize, do it last which means to follow the order:

1. Make it compile.
2. Make it run.
3. Make it right.
4. Make it fast.

Is this information useful?
It shows that no projects have a quality plan, which is not good. But a quality plan is not very important, because you do not get quality just because you have a plan. For an upcoming project a quality plan is written.

8.3.4 What is the defect density (Q20)?

The figures below show the problem and service call densities of the projects. To be able to know how to compare the projects one has to know how service calls and problems are reported. Here follows a description how the projects differ and which implicitly means how they should be compared:
Ascension: defects found internally are reported as service calls, no customer so no external defects. Problems not used but instead service calls only result in changes, which not is presented.
Curacao, *Hitra*, and *Mathilde 4*: defects found by customer are reported as service calls and defects found internally are reported as problems.
Mathilde 1, *Mathilde 2*, and *Mathilde 3*: all defects found are reported as service calls, and problems are only a sub-set of service calls.
Boigu and Venus: nothing reported.

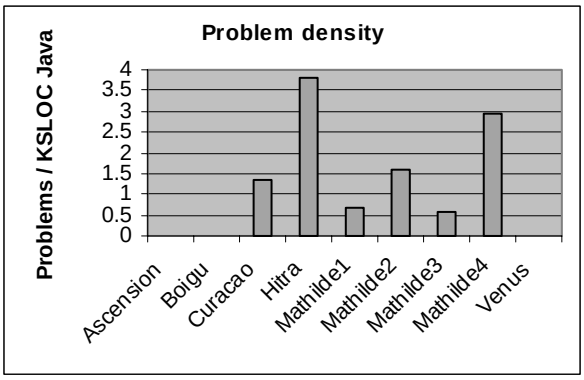

Figure 8.8 Problem density.

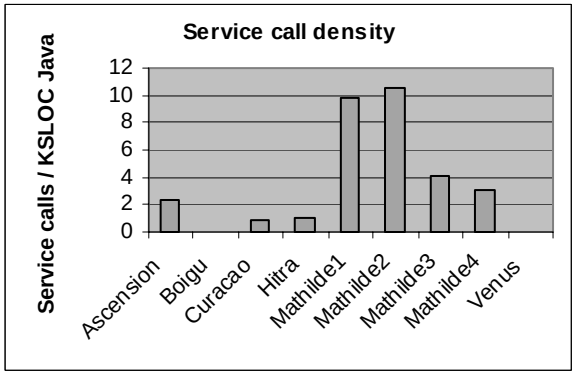

Figure 8.9 Service call density.

Does this agree with your picture of the projects and is the information useful?
I am surprised that Mathilde 4 had high problem density when the impression is that testing was neglected in this project.

8.3.5 How many service calls are created during the project (Q22)?

The figure below shows the number of service calls created for each project. See section 9.3.4 above for a description of how service calls are reported in the projects.

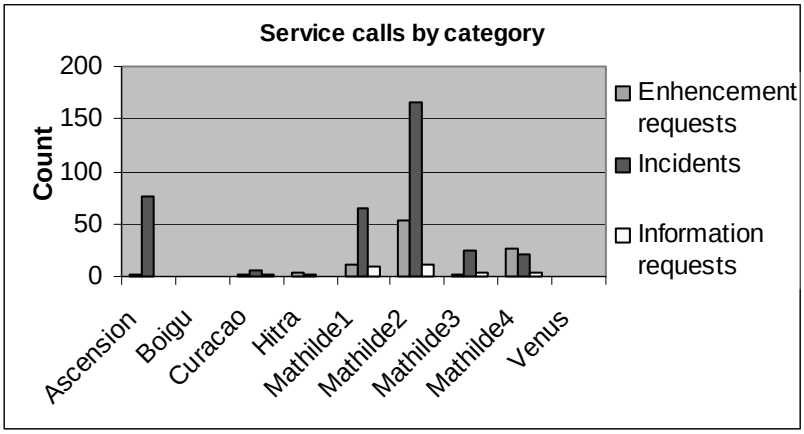

Figure 8.10 Number of service calls created.

Does this agree with your picture of the projects and what does it tell you?
Ascension was a merge that caused many problems. I did not manage Mathilde 1 and 2 so I do not know why they are so high.

8.3.6 How many NCSS of JUnit tests are created per new class (Q23)?

Figure 9.11 below shows the number of NCSS of JUnit tests for each project. But this is not correct so figure 9.12 shows what actually is possible to measure. Ascension and Mathilde 4 are merge projects so it is impossible to measure, which files that were created. Boigu was so small that it is possible to manually calculate which JUnit files that are written. Mathilde 1 and Mathilde 2 ran intertwined so what was created during which project is not possible to say, and neither is it possible to see if some of the tests existed before. Between Curacao and Hitra the test folder was not moved so the diff measure actually reported the new files for Hitra. Mathilde 3 and Venus are the only projects where it is sure that no JUnit tests were written. That Mathilde 3 not produced any JUnit tests is also visible in figure 9.11 where Mathilde 2 and 3 has the exact same number of NCSS.

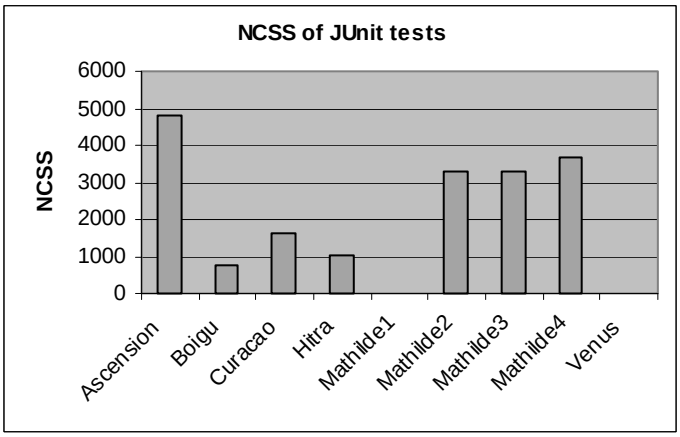

Figure 8.11 The number of NCSS of JUnit tests.

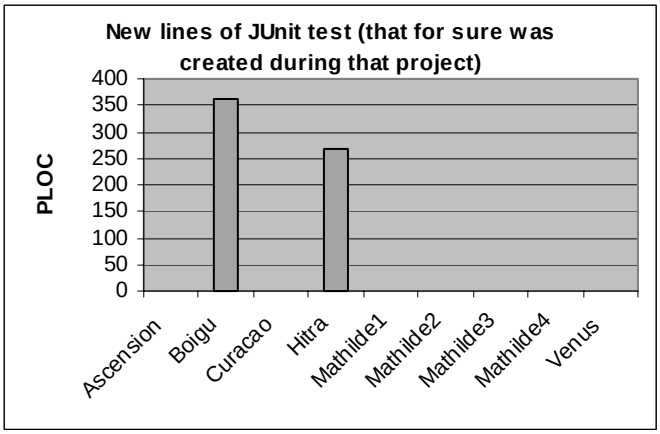

Figure 8.12 Number of new lines of JUnit test that could be determined.

8.4 Process quality (Goal 6)

8.4.1 Is the usage of standards and templates sufficient (Q25)?

In the SPE folder templates there are the following nine documents:

1. Curacao ES OVSD
2. Curacao FS OVSD
3. Design Document Template
4. Eros External Specification
5. External Specification OVSD
6. Hitra ES SD
7. Nias ES OVSD
8. Testdesign
9. Use case template

In one project folder there exists another templates folder which contain two functional specification templates. It is obviously not very clear which the ones to use are. On the organisational level the following are named as common templates:

1. ComponentTransitionBrief (transition ownership of a component)
2. Component_Test_Plan_Template
3. FuncSpecTemplate
4. Production Build Report Template
5. Quality Plan Template
6. Small Project Plan Template

A team called Standards & Technology answered the question about organisational standards. The first thing they pointed out is that the SPE team must follow the organisational standards that are applicable. The standards are rather generic and there are around thirteen different ones. They range from the previously named Java Coding Guideline to things like Shell Coding Conventions, Component Version Labelling Guide (how to identify software during different phases), and Foundation Directory Layout (how the file system is used by the software). When asked about which standards that is applicable in this environment the Standards & Technology team answered that it is obvious i.e. the C++ Coding Guidelines is not applicable here.

8.4.2 Is the process visible (Q26)?

The table below shows the results of how visible the process is. X means document exist, n means not needed, and blank means not found in the project storage place.

The headers in the table below are in order: external specification, functional specification, technical specification, code, peer review result, JUnit code, user documentation, and integration test.

That results of peer reviews exist for Ascension is only half the truth because it is only result from one peer review. If the peer reviews are just invisible or not performed is hard to tell, but they are not considered to be very important. For Mathilde 4 functional specification exist for two out of eight requirements.

	ES	Schedule	FS	TS	Code	Peer	JUnit	User doc.	Int. test
Ascension	n	x	x	x	x	x	x	x	x
Boigu	x	x	x	x	x		x	x	
Curacao	x	x	x	x	x		x	x	
Hitra	x	x	x	x	x		x	x	x
Mathilde 3	x				x				x
Mathilde 4	x	x	x	x	x		x	x	x
Venus	x	x			x			x	x

Table 8.4 Visibility of the process.

Does this agree with your picture and what does it tell you?
After things that mostly look like chaos this looks not too bad.

8.4.3 Is the documentation updated and finalised (Q27)?

Table 9.5 shows information about the project schedules. The answer to question seven (section 8.1.5) gave that none of the previous projects had a project plan so no questions can be answered about them. In table 9.6 ok means if it is clear which the initial requirements are, and complete means that the documents look like they are finished. Complete does not say if the document is correct because it can for example be that the technical specification is not as the implemented solution.

	Versions of schedule	Created	Last update	Cleanup scheduled
Ascension	26	9/17/03	2/11/04	Yes
Boigu	1	-	-	No
Curacao	2	8/19/03	8/19/03	No
Hitra	11	9/18/03	3/12/04	Yes
Mathilde 3	0	-	-	No
Mathilde 4	5	2/4/04	6/29/04	No
Venus	5	6/17/04	10/15/04	No

Table 8.5 Information about the project schedule.

	Requirements	External spec.	Functional spec.	Technical spec.
Ascension	Ok	Complete Draft	Complete Draft	Complete Draft
Boigu	Ok	Complete Draft	Complete Draft	Complete Draft
Curacao	Ok	Complete Released	Complete Released	Complete Draft

Hitra	Not ok (bullet list in workshop document)	Complete Accepted	Complete Draft	Complete Draft
Mathilde 3	Not ok (list in presentation notes)	Complete Review	Not found	Not found
Mathilde 4	Ok	Complete Draft	Complete but exist only for two out of eight req. Draft	Complete Draft
Venus	Not found	Complete Draft	Not found	Not found

Table 8.6 Information about the documents for requirements and specifications.

Does this agree with your picture and what does it tell you?
Mathilde 3 was not a good project so this does not surprise me.

8.4.4 Is the process how to report defects defined and well understood (Q28)?

A process for how to handle defects does exist but it is not documented. It is not explicitly agreed upon in the team, but the process is used in the current projects (at least the major ones). The process is that customers create a service call when they find a problem (bugs or other things) and the developers and tester create a problem when they find one. This means that problems describe internal defects and service calls describe external ones of the product. When controlling Service Desk for the projects that are supposed to use this process, it looks like the developers and tester use it correctly.

Does this agree with your picture and what does it tell you?
I think it is agreed upon. That the small projects not report defects in service desk is a problem and it should be handled.

8.4.5 How long are the service calls open on average (Q29)?

The figure below shows how long time service calls are open. The projects marked with a star have very limited number of data points so they may not give a good indication. Again it matters how service calls are recorded and see 9.4.3 for the description.

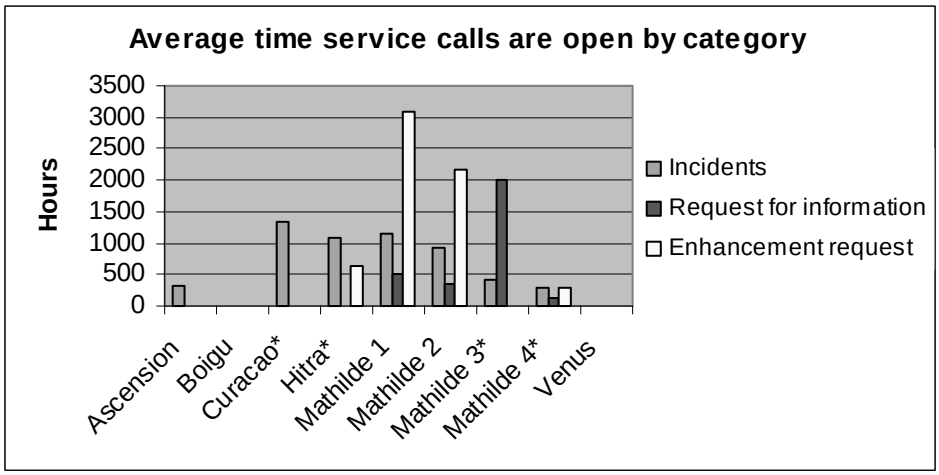

Figure 8.13 Average duration service calls are open by category.

Does this agree with your picture and what does it tell you?
That Ascension is low is not surprising when the work was done under at lot of pressure.

9 Conclusion

The existing information is not reliable because it was not collected for the purpose of evaluation. The information is not either repeatable because some of the measures would probably be different if repeated by someone else, or even if repeated by me. When this is the case the results and the conclusions are subjective.

The most important conclusion of these measurements is that the way the team has been working is not sufficient for reliable measurements. The data that exists is not good and the main reasons are:

- The documentation is not updated, especially the schedules which are important for many measures.
- The usage of Service Desk is not consistent and some projects do not track anything.
- Files and folders are sometimes moved which makes the SOME count incorrect.

Even though the measurements are not good it is possible to see some improvements. The work in Mathilde 4 is more visible than in Mathilde 3, and the way of tracking defects is consistent in Mathilde 4, Curacao, and Hitra.

The first objective of this thesis was to find out how software measurement can help a software team and its manager to get a better understanding of their work performance. One result of this objective is the productivities of the projects. These clearly show that the performance in the SPE team is lower than in the projects completed before its existence. But the productivity parameter could not be validated so one should not jump to conclusions.

The second objective was to see how and if software measurement can provide the manager with information that results in better estimations. The main finding here is that it is not even possible to calculate reliable values of how good the estimations were, which must be a prerequisite for this objective. Although the unreliable values do indicate that the experience of the leader is important. In Curacao and Hitra the leader was experienced and able to keep control, but in Mathilde 4 the leader was inexperienced and the project slipped rather much. The experience of the developers was rather high in all projects so it should not have made a big impact.

In terms of possibility to measure, it is not a big difference between the projects, but most information exists for Curacao and Hitra. This is due to a more defined way of work compared to the other projects, and probably that one leader and two developers is a suitable constellation. In the projects Venus and Boigu it may be that the lack of a formal leader made it possible to not track problems in Service Desk, but this is of course also depending on the nature of a project with few members.

The goal question metric (GQM) approach was used to decide what to measure. The conclusion of this is that it is not suitable to use when the existing data is very limited because it only results in questions and metrics that are not possible to answer and measure. The advantage of GQM is that it makes the manager think about the objectives for the measurements, which possibly can be used in future projects. In the later GQM iteration it is visible that the goals and questions are directed to fit the existing data, which not is its intended purpose**.** To document a GQM iteration is rather tedious and does not bring much when many of the measures in the end are slightly different.

If the results are related to CMM it is clear that the maturity of the organisation is low. A characteristic of the lowest maturity level is that it is hard to measure anything meaningful,

which almost the case is here. Another typical sign of an immature organisation is that the intermediate activities peer review and unit tests are not possible to measure.

10 References

[1] www.hp.com

[2] Carol A. Dekkers, *Making Software Measurement Really Work: #1 Aligning Measurement Expectations*, IT Metric Strategies (Cutter Corporation), April 2000.

[3] Ian Sommerville, *Software Engineering*, sixth edition, Pearson Education, Essex, 2001.

[4] IEEE Std 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Computer Society, New York, 1990.

[5] Peng Xu and Balasubramaniam Ramesh, *A Tool for the capture and use of Process knowledge in process tailoring*, Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS '03), 2002.

[6] Barry W. Boehm, *Research Directions in Software Technology*, Cambridge, 1979.

[7] Norman Fenton and Shari Lawrence Pfleeger, *Software Metrics – a rigorous and practical approach* (revised printing), second edition, PWS Publishing,

[8] Lawrence H. Putnam and Ware Myers, *Measures for Excellence - Reliable software on time, within budget*, Prentice Hall, New Jersey, 1992.

[9] Victor S. Basili, Gianluigi Caldiera, H. Dieter Rombach, *The Goal Question Metric Approach*, Encyclopedia of Software Engineering, Wiley, 1994.

[10] Sandro Morasca, *Software Measurement*. ftp://cs.pitt.edu/chang/handbook/26.pdf

[11] Lionel Briand, Khaled El Emam, Sandro Morasca, *On the Application of Measurement Theory in Software Engineering*, Empirical Software Engineering, 1996.

[12] Frank Niessink and Hans van Vliet, *A Pastry Cook's View on Software Measurement*, In: Software Measurement - Research and Practice of Software Metrics, Reiner Dumke and Alain Abran (ed.), Deutscher Universitaetsverlag, Wiesbaden, Germany, 1999, pp. 109-125

[13] Barbara Kitchenham, Shari Lawrence Pfleeger, Norman Fenton, *Towards a Framework for Software Measurement Validation*, IEEE Transactions on software engineering, vol. 21, no. 12, December 1995.

[14] Peter Kulik and Catherine Weber, *Software Metrics Best Practices 2001*, KLCI Research group, 2002.

[15] Jean-Philippe Jacquet, Alain Abran, *From Software Metrics to Software Measurement Methods*, Software Engineering Standards Symposium and Forum, 1997. Emerging International Standards. ISESS 97. Third IEEE International.

[16] Alfonso Fuggetta, Luigi Lavazza, Sandro Morasca, Stefano Cinti, Giandomenico Olando, Elena Orazi, *Applying GQM in an industrial software factory*, ACM Transactions on Software Engineering and Methodologies, 1997.

[17] Edmond VanDoren, *Cyclomatic complexity*, 1997. http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html

[18] Mark Lorenz and Jeff Kidd: *Object-Oriented Software Metrics*, Prentice-Hall, 1994.

[19] Capers Jones, *Backfiring: Converting Lines of Code to Function Points*, IEEE Computer 28, 11 (November 1995): 87-8.

[20] Jean-Marc Desharnais and Alain Abran, *Approximation techniques for measuring Function Points*, in International Workshop on Software Measurement (IWSM), Montreal, 2003.

[21] VanDoren, *Function Point Analysis*, 1997. http://www.sei.cmu.edu/str/descriptions/fpa_body.html

[22] Shyam R. Chidamber and Chris F. Kemerer, *A Metrics Suite for Object Oriented Design*, IEEE Transactions on Software Engineering, vol. 20, No. 6, June 1994.

[23] Kathleen Peters, *Software Project Estimation*. http://www.spc.ca/downloads/resources/estimate/estbasics.pdf

[24] Hareton Leung and Zhang Fan, *Software Cost Estimation*, The Hong Kong Polytechnic University. http://paginaspersonales.deusto.es/cortazar/doctorado/articulos/leung-handbook.pdf

[25] Carol A. Dekkers and Patricia A. McQuaid, *The Dangers of Using Software Metrics to (Mis)manage*, IT Pro, March – April 2002.

[26] Jakob Iversen and Lars Mathiassen, *Lessons from Implementing a Software Metrics Program*, Proceedings of the 33rd Hawaii International Conference on System Sciences, 2000.

[27] IBM Research Group, *Orthogonal Defect Classification*, 2002. http://www.research.ibm.com/softeng/ODC/ODC.HTM

[28] Jon T. Huber, *A Comparison of IBM's Orthogonal Defect Classification to Hewlett Packard's Defect Origins, Types, and Modes*, 1999. http://www.stickyminds.com/sitewide.asp?ObjectId=2883&Function=DETAILBROWSE&ObjectType=ART#authorbio

[29] ISO/IEC FDIS 9126-1:2000(E).

[30] Per Runeson, Niklas Borgquist, Markus Landin, Wladyslaw Bolanwski, *An Evaluation of Functional Size Methods and a Bespoke Estimation Method for Real-Time Systems*, PROFES'00 - Proceedings International Conference on Product Focused Software Process Improvement, pp. 339-352, 2000.

[31] Thomas Olsson, Per Runeson, *V-GQM: A Feed-Back Approach to Validation of a GQM Study*, Metrics ’01 - International Software Metrics Symposium, 2001.

[32] http://ovweb.bbn.hp.com/some/nsmd/htdocs/

[33] Linda Rosenberg et al, *Software Metrics and Reliability*, 9th International Symposium on Software Reliability Engineering, 1998.

[34] http://www.itil.co.uk

[35] http://java.sun.com/docs/codeconv/

[36] http://www-1g.cs.luc.edu/~van/cs330/lect13/

A Appendix – Tables and pictures

Size (SLOC)	B
5-15K	0.16
20K	0.18
30K	0.28
40K	0.34
50K	0.37
>70K	0.39

Table A.1 Constant B for the software equation. [7]

Productivity index	Productivity parameter	Application type	Standard deviation
1	754		
2	987	Microcode	1
3	1220		
4	1597	Firmware	2
5	1974	Real-time embedded Avionics	2 2
6	2584		
7	3194	Radar systems	3
8	4181	Command and control	3
9	5186	Process control	3
10	6765		
11	8362	Telecommunications	3
12	10946		
13	13530	Systems software Scientific system	3 3
14	17711		
15	21892		
16	28657	Business systems	4
17	35422		
18	46368		
19	57314		
20	75025		
21	92736		
22	121393		
23	150050		
24	196418		
25	242786		
26	317811		
27	392836		
28	514229		
29	635622		
30	832040		
31	1028458		
32	1346269		
33	1664080		
34	2178309		
35	2692538		
36	3524578		

Table A.2 Productivity index. [7]

Product attributes	

RELY DATA CPLX DOCU RUSE	Required system reliability Size of database used Complexity of system modules Extent of documentation required Required percentage of reusable components
Computer attributes	
TIME PVOL STOR	Execution time constraints Volatility of development platform Memory constraints
Personnel attributes	
ACAP PCON PEXP PCAP AEXP LTEX	Capability of project analysts Personnel continuity Programmer experience in project domain Programmer capability Analyst experience in project domain Language and tool experience
Project attributes	
TOOL SCED SITE	Use of software tools Development schedule compression Extent of multi-site working and quality of site communications

Table A.3 Project cost drivers for post architecture level COCOMO 2. [3]

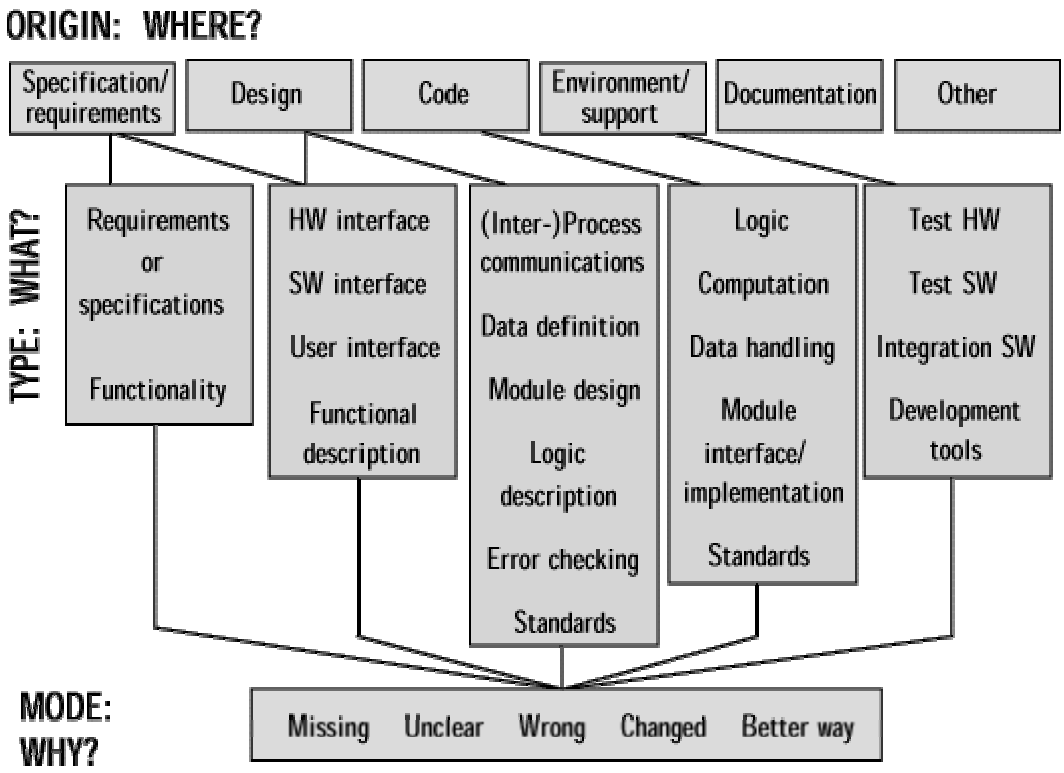

Figure A.1 Hewlett-Packard's defect origin, type, and mode. [36]

B Appendix – Complete GQM analysis

To decide what to measure and for what purpose the goal question metric approach is applied and iterated.

B.1 Goals version 1

The following goals are identified by the project manager:

Goal 1. **Improve the accuracy of the estimations from the project manager's viewpoint.**

Goal 2. **Reduce the risks in the project from the project manager's viewpoint.**

Goal 3. **Ensure the predictability of the costs in the project from the project manager's viewpoint.**

Goal 4. **Ensure the timeliness of the activities in the project from the project manager's viewpoint.**

Goal 5. **Analyse the organisational productivity from the manager's viewpoint.**

B.2 Questions version 1

From the goals the following questions (Q) are derived. The viewpoint of the goals is all from the project manager, and hence it is left out.

Goal 1. **Improve the accuracy of the estimations**.

- Q1. *How accurate are the project estimations (effort, duration, and cost)?*
- Q2. *How accurate are the estimations of effort and duration of the activities?*
- Q3. *Does the duration of the projects affect the accuracy of the project estimations?*
- Q4. *What is an acceptable level of accuracy of the estimations?*
- Q5. *Can a model be used for the estimations?*
- Q6. *How confident is the project manager with the estimations?*

Goal 2. **Reduce the risks in the project**.

- Q7. *What are the main risks in the project?*
- Q8. *How serious are the risks if they occur?*
- Q9. *How likely are the risks to occur?*
- Q10. *What to do to prevent them?*
- Q11. *Do the developers and project manager have the right skills?*

Goal 3. **Ensure the predictability of the costs**.

- Q12. *Are there any unexpected personnel or travelling costs?*

Goal 4. **Ensure the timeliness of the activities**.

- Q13. *What activities do not meet the schedule?*
- Q14. *Is the effort spent on finding and fixing defects under control?*
- Q15. *How many days are the developers on unplanned vacation and sick on average per month?*
- Q16. *Do the requirements change during the project?*

Goal 5. **Analyse the organisational productivity**.

Q17. *What is the organisational productivity according to the process productivity parameter?*

B.3 Metrics version 1

The questions resulted in the metrics (M) presented below. To make it clear both the direct and indirect metrics are explicitly stated. Duplicate metrics may exist, and this will be considered later on.

Goal 1. **Improve the accuracy of the estimations**.

Q1. *How accurate are the project estimations (cost, effort, duration)?*

M1. Actual effort for the project.
M2. Estimated effort for the project.
M3. Actual cost for the project.
M4. Estimated cost for the project.
M5. Actual duration for the project.
M6. Estimated duration for the project.
M7. Magnitude of relative error of the effort.
M8. Magnitude of relative error of the cost.
M9. Magnitude of relative error of the duration.

Q2. *How accurate are the effort and duration estimations of the activities?*

M10. Actual effort for each activity.
M11. Estimated effort for each activity.
M12. Actual duration for each activity.
M13. Estimated duration for each activity.
M14. Magnitude of relative error of the effort for each activity.
M15. Magnitude of relative error of the duration for each activity.

Q3. *Does the duration of a project affect the accuracy of the total estimations?*

M16. Actual duration of the project.
M17. Magnitude of relative error of the duration for the project.

Q4. *What is an acceptable level of accuracy for the total estimations (cost, effort, duration)?*

M18. Acceptable accuracy for estimation of total duration.
M19. Acceptable accuracy for estimation of total effort.
M20. Acceptable accuracy for estimation of total cost.

Q5. *Do the previous projects fit any existing model?*

M21. Actual size.
M22. Actual effort.
M23. Regression of size versus effort.

Q6. *How confident is the project manager with the estimations?*

M24. Confidence rating of the estimations.

Goal 2. **Reduce the risks in the project.**

Q7. *What are the main risks in the project?*

M25. The identified risks.

Q8. *How serious are the risks if they occur?*
M26. The seriousness of the risks.

Q9. *How likely are the risks to occur?*
M27. The likelihood of the risks.

Q10. *What to do to prevent the risks?*
M28. The plan to prevent the risks.

Q11. *Do the developers have the right skills?*
M29. Complexity rating of the project (or for each requirement).
M30. Developers' experience and knowledge.

Goal 3. **Ensure the predictability of the costs in the project.**
Q12. *Are there any unexpected personnel or travelling costs?*
M31. The reason for unexpected costs.
M32. The amount of unexpected costs.

Goal 4. **Ensure the timeliness of the activities in the project.**
Q13. *Which activities do not meet the schedule?*
M33. The type of activities that needed more effort (at least one day) than estimated.

Q14. *Is the total effort on finding and fixing defects under control?*
M34. Effort spent on fixing defects.
M35. Number of issues found during peer review.
M36. Effort spent on peer review.
M37. Effort spent on supporting and maintaining old projects.

Q15. *How many days are the personnel on unplanned vacation or sick on average per month?*
M38. Number of sick days per month.
M39. Number of days on unplanned vacation per month.
M40. Average number of unplanned absence days per month.

Q16. *Do the requirements change during the project?*
M41. Number of initial must requirements.
M42. Number of must requirements added after project start.
M43. Number of completed must requirements.
M44. Volatility of the requirements.

Goal 5. **Analyse the organisational productivity.**
Q17. *What is the organizational productivity according to the process productivity parameter?*
M45. New non-commented source statements (NCSS).
M46. Modified NCSS.
M47. Deleted NCSS.
M48. Duration of the main build phase.
M49. Effort spent on the main build phase.
M50. Process productivity parameter.

B.4 Questions version 2

When the metrics are defined the questions are reviewed to decide if there are any that should be added, modified, or deleted. First the questions and the reason for the change are presented, and then follow all questions with the updates. Q* means new question.

Goal 1.

Q1. Modified. *How accurate are the project estimations of duration?*
Reason: the current estimations are duration estimates but the activities are decomposed and estimated into small parts so duration and effort are almost the same. Effort estimations are therefore not worth doing. Cost estimations are not very interesting at the moment but tracking of costs may be.

Q2. Modified. *How accurate are the estimations of duration of the activities?*
Reason: as above, effort estimations do not provide enough to be worth doing.

Q*. *How big part of total effort is analysis, design, development, and test respectively?*
Reason: if there is a relation between these activities it can help when estimating.

Q*. *Does a project plan exist?*
Reason: if a project plan exists the project is likelier to be structured and managed so slippages can be avoided.

Goal 2.

Q9-11. Removed.
Reason: a risk analysis is hard to do but it is not sure that it provides anything. To have some risk identification question 8 is left.

Q12. Modified. *Do the developers and the project manager have the right skills?*
Reason: if the project manager not has the right skills or experience that is a significant risk.

Goal 3.

Q*. *How large are the total costs in the project?*
Reason: the cost in question 1 was removed so the tracking is added here.

The questions are as follows after the changes and with updated numbering:

Goal 1. **Improve the accuracy of the estimations**.

Q1. *How accurate are the project estimations of duration?*
Q2. *How accurate are the estimations of duration of the activities?*
Q3. *Does the duration of the projects affect the accuracy of the project estimations?*
Q4. *What is an acceptable level of accuracy of the estimations?*
Q5. *Do the previous projects fit any existing model?*
Q6. *How confident is the project manager with the estimations?*
Q7. *How big part of total effort is analysis, design, development, and test respectively?*
Q8. *Does a project plan exist?*

Goal 2. **Reduce the risks in the project**.

Q9. *What are the main risks in the project?*
Q10. *Do the developers and the project manager have the right skills?*

Goal 3. **Ensure the predictability of the costs**.
Q11. *Are there any unexpected personnel or travelling costs?*
Q12. *How large are the total costs in the project?*

Goal 4. **Ensure the timeliness of the activities**.
Q13. *What activities do not meet the schedule?*
Q14. *Is the effort spent on finding and fixing defects under control?*
Q15. *How many days are the developers on unplanned vacation and sick on average per month?*
Q16. *Do the requirements change during the project?*

Goal 5. **Analyse the organisational productivity**.
Q17. *What is the organisational productivity according to the process productivity parameter?*

B.5 Goals version 2

When the questions and metrics are defined it is easier to decide if there are any goals that should be added, modified, or deleted. Two new goals are identified and the goal reduce the risks in the project is removed because a risk analysis should not be a part of a metrics program. Instead the risk analysis is a part of the project manager's tasks and should be documented in the project plan. The current goals are:

Goal 1. **Improve the accuracy of the estimations from the project manager's viewpoint.**

Goal 2. **Ensure the predictability of the costs in the project from the project manager's viewpoint.**

Goal 3. **Ensure the timeliness of the activities in the project from the project manager's viewpoint.**

Goal 4. **Analyse the organisational productivity from the manager's viewpoint.**

Goal 5. **Increase the quality of the product from the project manager's viewpoint.**

Goal 6. **Increase the quality of the process from the project manager's viewpoint.**

B.6 Questions version 3

With the new goals the following changes are introduced:

Goal 1.
Q4. Removed.
Reason: it does not provide anything.

Goal 3.
Q*. *How many hours are lost due to network outage and tool problems?*
Reason: can cause project slippage.

Q*. *Do the developers know their responsibilities?*
Reason: if they not know exactly what to do they may do the wrong things.

Goal 5.

Q*. *How many lines of code are the new files on average?*
Reason: too large files can be hard to maintain.

Q*. *Is the code standard followed?*
Reason: code that follows one notation is easier to maintain.

Q*. *Is good quality defined for the product?*
Reason: the developers should know what to emphasise on.

Q*. *What is the defect density?*
Reason: it is easy to track and gives one view of quality.

Q*. *Where in the code do the defects originate from?*
Reason: perhaps should some part of the product be redesigned.

Q*. *How many service calls are created during the project?*
Reason: another indication of the quality.

Goal 6.

Q*. *Is the usage of standards and templates sufficient?*
Reason: to make the documentation consistent.

Q*. *Is the process visible?*
Reason: a visible process is easier to control.

Q*. *Is the documentation updated and finalised?*
Reason: documents that not are finalised are hard to analyse.

Q*. *Is the process how to report defects defined and well understood?*
Reason: if it is not understood by the developers the defect reports will not reflect the reality.

Q*. *How long are the service calls open on average?*
Reason: if the service calls

Q*. *During what activity are the defects introduced?*
Reason: indicates if some activity should get more time.

Q*. *During what activity are the defects found?*
Reason: says if the methods to prevent and find defects during the process are good enough.

The current questions are:

Goal 1. **Improve the accuracy of the estimations**.

Q1. *How accurate are the project estimations of duration?*
Q2. *How accurate are the estimations of duration of the activities?*

Q3. *Does the duration of the projects affect the accuracy of the project estimations?*
Q4. *Do the previous projects fit any existing model?*
Q5. *How confident is the project manager with the estimations?*
Q6. *How big part of total effort is analysis, design, development, and test respectively?*
Q7. *Does a project plan exist?*

Goal 2. **Ensure the predictability of the costs**.
Q8. *Are there any unexpected personnel or travelling costs?*
Q9. *How large are the total costs in the project?*

Goal 3. **Ensure the timeliness of the activities**.
Q10. *What activities do not meet the schedule?*
Q11. *Is the total effort on finding and fixing defects under control?*
Q12. *How many days are the developers on unplanned vacation and sick on average per month?*
Q13. *Do the requirements change during the project?*
Q14. *How much time is lost due to network outage and tool problems?*
Q15. *Do the developers know their responsibilities?*

Goal 4. **Analyse the organisational productivity**.
Q16. *What is the organisational productivity according to the process productivity parameter?*

Goal 5. **Increase the quality of the product.**
Q17. *How many lines of code are the new files on average?*
Q18. *Is the code standard followed?*
Q19. *Is good quality defined for the product?*
Q20. *What is the defect density?*
Q21. *Where in the code do the defects originate from?*
Q22. *How many service calls are created during the project?*
Q23. *How many NCSS of JUnit tests are created per new class?*
Q24. *How many of the unit tests were actually performed?*

Goal 6. **Increase the quality of the process.**
Q25. *Is the usage of standards and templates sufficient?*
Q26. *Is the process visible?*
Q27. *Is the documentation updated and finalised?*
Q28. *Is the process how to report defects defined and well understood?*
Q29. *How long are the service calls open on average?*
Q30. *During what activity are the defects introduced?*
Q31. *During what activity are the defects found?*

B.7 Metrics version 2

The changes resulted in the following metrics. The reasons for the changes are not presented when most are connected to the changes of the questions. All metrics are presented and duplicates may occur, but the most obvious ones are given the same name.

Goal 1. **Improve the accuracy of the estimations**.
Q1. *How accurate are the project estimations of duration?*
M1. Actual duration for the project.

M2. Estimated duration for the project.
M3. Magnitude of relative error of the duration.

Q2. *How accurate are the estimations of duration of the activities?*
M4. Actual duration for each activity.
M5. Estimated duration for each activity.
M6. Magnitude of relative error of the duration for each activity.

Q3. *Does the duration of a project affect the accuracy of the total estimations?*
M1. Actual duration of the project.
M7. Magnitude of relative error of the duration for the project.

Q4. *Do the previous projects fit any existing model?*
M8. Actual size.
M9. Actual effort.
M10. Regression of size versus effort.

Q5. *How confident is the project manager with the estimations?*
M11. Confidence rating of the estimations.

Q6. *How big part of total effort is analysis, design, development, and test respectively?*
M12. Total actual effort.
M13. Actual effort of analysis, design, development, and test respectively.

Q7. *Does a project plan exist?*
M14. If a proper project plan exist or not.

Goal 2. **Ensure the predictability of the costs in the project.**
Q8. *Are there any unexpected personnel or travelling costs?*
M15. The reason for unexpected costs.
M16. The amount of unexpected costs.

Q9. *How large are the total costs in the project?*
M17. Total costs of personnel and travel expenses.

Goal 3. **Ensure the timeliness of the activities in the project.**
Q10. *Which activities do not meet the schedule?*
M18. The type of activities that needed more effort (at least one day) than estimated.

Q11. *Is the effort spent on finding and fixing defects under control?*
M19. Effort spent on fixing defects.
M20. Number of issues found during peer review.
M21. Effort spent on peer review.
M22. Effort spent on supporting and maintaining old projects.
M23. Number of old defects fixed during the project.

Q12. *How many days are the personnel on unplanned vacation or sick on average per month?*
M24. Number of sick days per month.
M25. Number of days on unplanned vacation per month.

M26. Average number of unplanned absence days per month.

Q13. *Do the requirements change during the project?*
- M27. Number of initial must requirements.
- M28. Number of must requirements added after project start.
- M29. Number of completed must requirements.
- M30. Volatility of the requirements.

Q14. *How much time is lost due to network outage and tool problems?*
- M31. Number of hours the network was down.
- M32. Number of hours tool problems caused disruption in the work.

Q15. *Do the developers know their responsibilities?*
- M33. Developers' knowledge about their responsibilities.

Goal 4. **Analyse the organisational productivity.**

Q16. *What is the organizational productivity according to the process productivity parameter?*
- M34. New non-commented source statements (NCSS).
- M35. Modified NCSS.
- M36. Deleted NCSS.
- M37. Duration of the main build phase.
- M38. Effort spent on the main build phase.
- M39. Process productivity parameter.

Goal 5. **Increase the quality of the product.**

Q17. *How many lines of code are the new class files on average?*
- M40. Number of new Java class files.
- M41. Number of lines of code in the new Java files.
- M42. Average number of lines of code in the new Java files.

Q18. *Is the code standard followed?*
- M43. Check whether the code standard is followed or not.

Q19. *Is good quality defined for the product?*
- M44. Whether it is documented what good quality is.

Q20. *What is the defect density?*
- M45. Number of defects reported in service desk during the project.
- M46. Total number lines of Java code.
- M47. Number of defects per thousand lines of code.

Q21. *Where in the code do the defects originate from?*
- M48. Origin in the code of the defects.

Q22. *How many service calls are created during the project?*
- M49. Number of service calls reported in service desk for the project.

Q23. *How many NCSS of JUnit tests are created per new class file?*
- M50. Non-commented sources statements of JUnit tests.
- M41. Number of new Java class files.

Q24. *How many of the unit tests are actually performed?*
- M51. Number of unit tests performed.

Goal 6. **Increase the quality of the process.**

Q25. *Is the usage of standards and templates sufficient?*
- M52. Number of existing standards in the organisation.
- M53. Number of standards used in the project.
- M54. Number of existing templates in the team.
- M55. Number of templates used in the project.
- M56. Developers' knowledge about standards and templates.

Q26. *Is the process visible?*
- M57. Number of documents produced internally during the project.
- M58. Activities in the process that are invisible.

Q27. *Is the documentation updated and finalised?*
- M59. Whether cleanup after the project is scheduled or not.
- M60. Number of times the project plan and schedule were updated.
- M61. Dates when the project plan and schedule were created and last updated.
- M62. Documents that still are in an early version.

Q28. *Is the process how to report defects defined and well understood?*
- M63. Whether a document exists that defines the process of reporting defects.
- M64. Developers' knowledge about how to report defects.

Q29. *How long are the service calls open on average?*
- M50. Number of service calls reported in service desk for the project.
- M65. The time closed service calls for the project was open.
- M66. Average time the closed service calls were open.

Q30. *During what activity are the defects introduced?*
- M67. The activity where the defect is introduced.

Q31. *During what activity are the defects found?*
- M68. The activity where the defect is found.

C Appendix – Measurement method

In this section follow descriptions of the measurement methods, error sources, and the need of validation for the different metrics. When referring to project manager it is the SPE manager or the project leader and in this section no separation is done between the different roles. The structure is that the metrics are presented according to their goals and questions.

C.1 Accuracy of the estimations (Goal 1)

C.1.1 How accurate are the estimations of duration (Q1)?

The necessary direct metrics to answer this question are the actual duration (M1) and estimated duration (M2). The necessary indirect metric is the magnitude of relative error (M3), which is calculated from the direct ones. The estimated duration (M2) is derived from the first version of the project schedule. The date when the project ended (or when some late activity ended) is acquired from the project manager and with the start date from the project schedule the actual duration (M1) is calculated. M1 and M2 are stated in whole weeks. The magnitude of relative error for duration is calculated with the formula in section 4.7.1, which is: $|(Actual - Estimated)/Actual|$.

A possible source of error for the estimated duration is that the schedule is not estimations but actual duration. As long as the schedule is version controlled and it is possible to control when it is created and updated this is only minor. A larger error source is the actual duration that in most cases is acquired from the project manager because the project schedule is not up to date. The manager relies on sources like emails to customer, and when the product is built. These sources are taken out of context, which makes them less reliable. The late activities that the project manager gives dates to are neither perfectly consistent, for example they can be: testing ended, or product delivered to customer.

To validate the estimated duration the schedules need to be checked for when they are created during the project. See appendix D for details of validation of the schedules. The actual duration cannot be validated when it is acquired the way it is. The magnitude of relative error cannot be validated, but is valid if the inputs are. In this case the magnitude of relative error is not valid.

C.1.2 Does the duration of a project affect the accuracy of the estimations (Q3)?

One indirect and one direct metric are needed to answer this question, which are the magnitude of relative error (M3) and the actual duration (M1). They are introduced in question one (Q1) above so the method is not described again. The magnitude of relative error of the first estimation of duration is plotted against the duration of the project. For error sources and validation see question one (Q1).

C.1.3 Do the previous projects fit any existing model (Q4)?

To see if it is advisable to use a model for estimations the data from previous projects is applied to regression of size versus effort. To construct a local cost model takes time and requires substantial and accurate measurements, and therefore it is not considered here. COCOMO is designed for projects that start from scratch and is therefore left out.

Regression of size versus effort

The effort is calculated from the project schedule by multiplying the duration for each activity with the number of people working on it. The sum is the total effort (M9). Deployment and project setup is not included in the total effort when they are not always scheduled. The number of thousand lines of code is calculated with SOME (M8). For all projects the total effort in months is plotted against the number of thousand of new lines of code in a log-log

graph (M10). Two graphs are presented, one with Java and one with Java and SQL lines of code.

The largest source of error is that the effort is calculated from schedules that are not up to date. The schedules also look different which makes it hard to be consistent of what to count as the effort. The size in lines of code has the error source that some files may have been moved during the project that will count them as new, even though they are not. To include or exclude SQL lines of code can give an indication of what gives best result. For validation of that SOME counts correctly see appendix D. The actual effort cannot be validated when it is calculated as it is.

C.1.4 How big part of total effort is analysis, design, development, and test respectively (Q6)?

To find the answer to this question the effort of the analysis, design, development, and test are calculated from the schedule. The schedules are not explicitly stated in these activities so the sub-activities are divided into these (M13). The total effort (M12) is the sum of these four activities, and the effort is as before calculated as the duration times the number of people working with it.

One error source is that when dividing sub-activities into the four main activities, some activities are left out to make it somewhat consistent between the schedules. Although the biggest problem is as before that the effort is not the actual effort. Another large error source is that analysis most times is scheduled without sub-activities that make it hard to know what really was done. This measure is only meaningful if there is a clear separation between the different activities, but in the SPE team the design and development run not perfectly separated. The actual effort can as mentioned before not be validated.

C.1.5 Does a project plan exist (Q7)?

To answer this question the project documentation is searched through to see if there are any project plans (M14).

C.2 Organisational productivity (Goal 4)

C.2.1 What is the organizational productivity according to the process productivity parameter (Q16)?

To calculate the process productivity parameter (M39) the necessary metrics are new (M34), modified (M35), and deleted (M36) lines of code, and duration (M37) and effort (M38) of the main build phase. When calculating lines of code in the equation the new ones are adjusted (multiplied) with 1 (not adjusted), modified with 0.53, and deleted with 0.11 according to table 4.6. The values for effort and duration are derived from the project schedules, except from Mathilde 3 where the values are based on information (start and end date of the project and when people had vacation) from the project manager. Effort and duration are in years with two decimals precision. The constant B is from table A.1 in appendix A. The equation is as follows:

$$\text{Process productivity parameter} = \frac{(\text{new}) * 1 + (\text{modified}) * 0.53 + (\text{deleted}) * 0.11}{\left(\frac{\text{duration}}{\text{B}}\right)^{\frac{1}{3}} * \text{effort}^{\frac{4}{3}}}$$

New, modified, and deleted lines of code are used instead of the non-commented source statements. This error source makes it less sensible to compare to external projects. The error source that files may have been moved during the project and then becomes counted as new is still present.

All completed projects are maintenance (evolution) projects and according to table 4.6 it takes some effort to test the new code with the existing one. The code base is large and for example the Boigu project contained almost 269 000 non-commented source statements when it started. According to the theory the effort is 0.12 per line of code to test all existing lines with the new ones. Hence this would result in very high values of the productivity for all projects, so to get away from this problem this effort of testing existing code is simply neglected. This error source is large but when the same approach is used for all projects it is at least consistent.

As mentioned before the main development is done in Java but SQL code is also produced. It is plausible that it takes more effort to develop one line of Java than one line of SQL code, but this relation is unknown. When this is a large uncertainty two values of the productivity parameter are calculated, one when the Java and SQL lines are weighted the same, and one when only the Java lines are counted. The average of these is also calculated. The average of SPE and non-SPE projects and their standard deviation are also calculated to be able to compare the differences.

One error source with the effort and duration of the main build phase is that it is hard to extract it from the project schedules in a consistent way. As before the project setup and deployment are not included. The fact that the actual effort is not recorded is also an issue here. Mathilde 1 and 2 ran intertwined which makes their values possibly different from the other projects. For validation of the process productivity parameter see appendix D.

C.3 Product quality (Goal 5)

C.3.1 How many lines of code are the new files on average (Q17)?

To answer this question the number of physical lines of code and the number of new, modified, and deleted files (M40) are collected from SOME. Due to limitations of SOME it is not possible (or not without too much manual work) to get only the lines of code in the new files, therefore the modified and deleted files are also counted. The lines of code in the files are collected from SOME (M41). The average (M42) of how many lines the files contain are calculated and showed in a graph. When the work includes changing in many existing files this may not be a good measure of the current work. Therefore two other measures are collected, which are the size of the largest new file in PLOC and the size of the largest modified file in PLOC.

That also modified and deleted files are counted is a large error source, but the two other measures can help to give a more correct picture. Other error sources of counting the number lines of code have been presented before. For validation of SOME see appendix D.

C.3.2 Is the code standard followed (Q18)?

To see if the code standard is followed (M43) the new files are identified with SOME, and one new file of reasonable size (around 100 PLOC or only that many lines are checked) is selected. In cases when there is no new file with appropriate size, a modified one is selected. The existing coding guideline follows in most cases Sun's Code Conventions for Java [35], but there are exceptions. The general rule is that teams are allowed to diverge from the guideline as long as they not disagree with the Sun's Code Conventions for Java. The

guideline says that one file should have the same standard. In case of additions or modifications to a file it should be consistent to the old code or the whole file should be changed to follow the coding guideline. New files shall follow the guideline.

To make it practical the check is not complete but the following issues are considered:

1. Curly braces.
   Opening curly braces should be ending a row and not on a new one. Use braces when it increases readability, even if it is not necessary.
2. Indentation.
   Indent nested code with four blanks.
3. Long lines.
   A line should not be longer than 80 characters. Long lines should be broken up in a correct way.
4. Statements.
   Do not use multiple statements on one line.
5. Blanks.
   Use blanks before and after binary operations. No blanks with unary operations. Use one blank after keywords. Do not put a blank after an opening and before a closing parenthesis.
6. Names.
   Use names that are meaningful on variables and methods. Class names should only have the first letter capitalised even if it is an abbreviation.
7. Documentation.
   The documentation should describe why the code is doing what it does, and not just what it does. @author and @version should not be included in the comments because they can be obtained elsewhere. Classes should have the following comment:

```
/**
*     Description
*
*     @see
*     @since
*     @deprecated
*/
```

   Methods should look like below, but @param, @return, and @throws are obviously only needed if there are parameters, a return other than void, and if something is thrown.

```
/**
*     Description
*
*     @param
*     @return
*     @throws
*
*     @see
*     @since
*     @deprecated
*/
```

The files differ in size and therefore it is only checked if a point (as defined above) is violated or not, and not the number of them. The coding guideline also includes object oriented thresholds that are: code complexity should not exceed 10, a file should not exceed 1500

lines, and a function should not exceed 60 lines. It is not clear where these numbers come from and they are not considered.

C.3.3 Is good quality defined for the product (Q19)?

This question is answered by looking in all possible documents that might contain something about the quality (M44).

C.3.4 What is the defect density (Q20)?

To calculate the defect density is not straightforward because the way to report defects has changed with time. The measures closest to defect density are problem density and service call density, and why this is will become clear in the answer to question 28. The number of thousand lines of code (M46) is measured with SOME but only the Java lines of code are counted. The number of problems and service calls (M45) are collected from Service Desk and the densities (M47) are calculated by dividing those with the number of thousand lines of code. The error sources are that there may be duplicates of service calls and problems. Validation of the data from Service Desk is not possible.

C.3.5 How many service calls are created during the project (Q22)?

To answer this question the number of service calls (M49) is collected from Service Desk and showed in a diagram. This is a measure of how many service calls that are related to a project and not only how many that were created during it but also afterwards. Error sources are again that service calls may be duplicates of already existing issues, and the way Service Desk is used is different.

C.3.6 How many NCSS of JUnit tests are created per new class (Q23)?

To answer this question the necessary metrics are the number of non-commented source statements of all files in the test directory (M50), and the number of new files (M41). The folder with JUnit tests is in most projects not stable which cause some difficulties. In cases when the folder is moved SOME counts all files in the test folder as new, even though most of them are from previous projects or part of the JUnit framework. The error of this metric is so large that it should only be seen as the number of non-commented source statements in the test folder. The first figure shows only this, and per new class is left out when the error is so large. A second figure shows new lines of code of JUnit tests where it could be determined easily, which only was for two projects.

C.4 Process quality (Goal 6)

C.4.1 Is the usage of standards and templates sufficient (Q25)?

To find out the organisational standards (M52) and templates the Standards & Technology team in the organisation is asked for them. The documentation and team members are consulted for the team templates (M54).

C.4.2 Is the process visible (Q26)?

The visibility of the process is analyzed by checking what is produced during it. During the four main activities analysis, design, implementation, and test the following documents should normally be produced and available (M58):

Analysis:	schedule, external specification.
Design:	functional specification, technical specification.
Implementation:	code, peer review result, JUnit code, user documentation.
Test:	integration test result.

Some of this was covered in the gap analysis and in other questions, but this summarizes it. If documents exist but are not in the project folder, a person involved in the project is asked if they may be somewhere else. One of the projects is internal, which means that external specification is not needed.

C.4.3 Is the documentation updated and finalised (Q27)?

To see if the documentation is well kept the creation date, last update (M61), and how many times the schedule and project plan are updated (M60) are acquired from the version controlled software. To see if cleanup is scheduled or not (M59) the project schedule is checked. The requirements document is checked for whether it is clear which the initial requirements are. The specifications are checked if they look complete or not, and what status they have (M62). The status of a document may not be very important but if it used it should be used correctly.

Whether a document is complete or not is a subjective judgement. Some document may be missed but this error source is only minor. Validation of these metrics is not possible. The number of documents produced internally is neglected because it is after all not interesting.

C.4.4 Is the process how to report defects defined and well understood (Q28)?

To answer this question the project manager is asked about if there is a process how to handle defects in Service Desk (M63). To see if the developers and tester use it correctly the data in Service Desk is controlled (M64). To check the data instead of asking the developers was chosen because it is more objective. An error source is that the check of data in service desk may only cover a small number of the developers but this is not very serious.

C.4.5 How long are the service calls open on average (Q29)?

The time service calls are open (M65) is collected from Service Desk, as well as the number of them (M50). The average (M66) is calculated and presented in a graph. When it makes a big difference what a service call is categorized as, they are presented in the three categories: enhancement request, incident, and information request. An error source is that service calls are categorized as incidents by default, and people may miss to change it. An error source is that how to handle service calls can differ from project to project. In some projects service calls are kept open longer on purpose so that the customer can see what is happening with it. Another error source is that the number of service calls related to a project may be sparse which can give a wrong picture. Validation of this is not possible.

D Appendix – Validation

As described in section 3.5, a definition of a valid measure is that it accurately characterizes the attribute it claims to measure. The validation of the different measures and information sources are described here.

D.1 Code

To validate that SOME counts correctly, one Java-file and one SQL-file are counted manually according to the definitions in section 5.4.1. Table 10.1 shows the manually counted numbers. When the cyclomatic complexity not is used it is not counted.

	PLOC	Comments	Blanks	NCSS
The Java-file	53	38	3	9
The SQL-file	56	0	10	-

Table C.1 Numbers of two manually counted files.

The figure below is an extract from SOME for the selected Java-file. The missing labels are in order Name (trimmed), NCSS, PLOC, Cyclomatic Complexity, Comments, and Blanks.

Figure C.1 Extract from SOME of measured Java-file.

Figure 10.2 is the equivalent SOME extract for the SQL-file. The missing labels are in order Name (trimmed), PLOC, Blanks, and Comments.

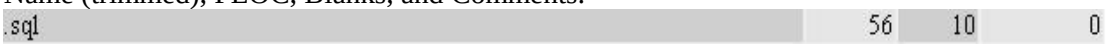

Figure C.2 Extract from SOME of measured SQL-file.

The manual count agrees with the SOME count. The measures PLOC, comments, blanks, and NCSS accurately characterises the attributes they claim to measure and therefore they are valid measures.

To validate that the turmoil accurately measures the difference between two code bases they are manually compared. From the turmoil report one new, one modified, and one deleted file are selected. The new file is controlled so it not exists in the old code base. The modified file is compared to see if it really is modified. The deleted file is checked whether it really is deleted.

The new and deleted files are controlled and they are new and deleted. There is no way to show this and it will not be discussed further.

The manual count of the selected modified file in the first code base gives the following.

First code base	PLOC	Comments	Blanks	NCSS
The Java-file	101	41	13	43

Table C.2 Result of the manual count of the selected Java-file in the first code base.

The manual count of the file in the second code base gives the following.

Second code base	PLOC	Comments	Blanks	NCSS
The Java-file	103	41	13	45

Table C.3 Result of the manual count of the selected Java-file in the second code base.

From these numbers it is only possible to say that the number of new lines should be at least 2. If lines were deleted it is higher. The two files are also compared for modifications but none are found. This gives that no lines are deleted, and the number of new lines are 2.

The turmoil measure reports the following between the code bases. The missing labels are in order Name, NCSS, PLOC, Blanks, Comments, Cyclomatic Complexity, New lines, Modified lines, Deleted lines, and Turmoil.

Figure C.3 Extract from SOME showing the turmoil report for the selected Java-file.

The turmoil report shows the actual difference and the turmoil measure is therefore a valid measure of the difference in lines of code between two files. It is not possible to validate that the files in the turmoil report really are the right ones, because when files are moved between different folders they are counted as new.

D.2 Project schedule

The project schedule is a source of information for many measures and if a measure is valid depends both on the schedule and the measure. First of all it must be clear if the schedule expresses effort or duration, or both. A valid measure for duration cannot be extracted from a schedule expressing effort, unless it is possible to show a valid relation between the two. What is measurable from a schedule also depends on if it is updated during the project. A project schedule created in the beginning of a project can only claim to be estimations of the activities. It cannot be validated for measures that claim to characterise the actual values. A schedule that is updated can on the other hand be validated to measure actual values.

The table below describes if the schedules from previous projects are valid. This is done by checking when the schedules are created compared to start date of the project, and how well they are updated. All previous project schedules are expressed in duration and therefore the actual effort is not possible to validate.

	Valid measures of estimated duration	Valid measures of actual duration	Valid measures of actual effort
Ascension	Yes	No	No
Boigu	Yes	No	No
Curacao	Yes	No	No
Hitra	Yes	No	No
Mathilde 3	No (no schedule exist)	No	No
Mathilde 4	Yes	No	No
Venus	Yes		

Table C.4 Validity of project schedules.

D.3 Process productivity parameter

The process productivity parameter claims to measure the productivity within an organisation, but if it really does is hard to validate. When the formula is based on lines of code, effort, and duration the first criteria must be that those measures are valid. According to above neither the actual effort nor duration are valid measures and therefore the process productivity is not possible to validate.

E Appendix – Coding standard

How the files deviates from the coding standard.

Modified file. Ascension

1. Opening curly brace on a new row.
3. Long lines broken up at the wrong place, a comma should not start a new line. Lines that are more than 80 characters.
5. Blank after an opening parenthesis and before a closing parenthesis.
7. Methods comments are missing @return, @throws, @see, @since, and @deprecated.
   Class comments are missing @see, @since, and @deprecated.
   @author in the file.
   The file contains author, date, and revision in the comments.

Modified file (new files too small). Boigu

7. Method comments are missing @param, @return, @see, @since, and @deprecated.
   Class comments are missing @see, @since, and @deprecated.
   Comments that are not descriptive enough.
   The file contains author, date, and revision in the comments.
   One method completely without comments.

New file. Curacao

3. A long line broken up in an incorrect way.
5. Blanks after opening parenthesis and before closing parenthesis.
6. Variable name that is non-descriptive where it easily could and should be.
   Class name with more than the first letter capitalised.
7. One method without comments.
   Method comments are missing @param, @return, @see, @since, and @deprecated.
   Class comments are missing @see, @since, and @deprecated.
   @author in the file.

New file. Hitra

1. Opening curly brace on a single row.
3. Several lines have over 80 characters.
5. Blanks after opening parenthesis and before closing parenthesis.
   No blank after keywords.
7. Method comments are missing @return, @see, @since, @deprecated.
   Class comments are missing @see, @since, and @deprecated.

New file. Mathilde 3

3. One line is longer than 80 characters.
7. Methods without comments, but they are one or two lines and easy to understand.
   Methods comments are missing @param, @return, @see, @since, and @deprecated.
   Class comments are missing @see, @since, and @deprecated.

New file. Mathilde 4

1. Openings curly braces on new rows.
3. Lines longer than 80 characters.
6. Class name with more than one initial capital letter.
7. Methods without comments.
   Method comments are missing @return, @see, @since, and @deprecated.
   Class comments are missing @see, @since, and @deprecated.

@version in the class.
The file contains author, date, and revision in the comments.

New file (only looked at the beginning of the file). Venus

2. Indentation sometimes eight spaces.
3. Lines longer than 80 characters.
7. Method without comments.
   Methods comments are missing @return, @see, @since, and @deprecated.
   Class comments include @author and are missing @see, @since, and @deprecated.