Software Complexity and Project Performance

Master thesis
by
Sofia Nystedt

Bachelor thesis
by
Claes Sandros

Department of Informatics
School of Economics and Commercial Law at the University of Gothenburg

Spring Semester 1999

Supervisor Birgitta Ahlbom

Abstract

Software complexity is the all-embracing notion referring to factors that decide the level of difficulty in developing software projects. In this master thesis we have examined the concept *software complexity* and its effect on software projects, concerning productivity and quality, for our assignor, Ericsson Mobile Data Design AB (ERV). Our literature studies have compelled to the development of our own model of complexity, as no such comprehensive model existed. With this model as a starting point we have examined existing methods of different software metrics. According to our model of complexity the metrics we found focused either on software quality or software size/effort. In our suggestion to ERV, of how they should measure their projects, we have combined both of these attributes in a metric called performance. Two of the functional size methods we found, Function Point Analysis (FPA) and Full Function Points (FFP), were applied on a completed software project to determine which method suited best for the projects at ERV. Our hypothesis was that FFP was most suitable and our tests of FFP and FPA proved that this hypothesis was right. Based on our theoretical studies of the quality metrics we suggested that a structural complexity measure (or a combination of several of them) and error density should each be combined with FFP, at different stages of the development process, to present the measure of performance.

Preface

It has been a privilege for both of us to be able to write our final exam at Ericsson Mobile Data Design AB (ERV). We would like to thank the staffs at JN and JK that have helped us whenever we have needed their knowledge. Special thanks to Anna Börjesson and Mikael Törnqvist, who had the good sense of person knowledge to bring the two of us together for this occasion. Thank you also Anders Klint, our supervisor at ERV, and of course Birgitta Ahlbom, our instructor at the Department of Informatics at the University of Gothenburg.

Sincerely

Sofia Nystedt Claes Sandros

TABLE OF CONTENTS

FIGURES & TABLES

Figures

Tables

1 Introduction

1.1 Background

This report is of special interest for those who are involved in software management, but also for others who are interested in how to measure the software development process. Good management of software projects demands good methods for measuring the process. Measures are needed for two main reasons - to know about the actual outcomes of the project and for making better estimations of new projects.

At ERV there has been an ever-increasing need to measure and compare the productivity in different projects. To accomplish this they have noticed that the complexity of the project task is an important factor to take into account. Up until now there has not existed any method in operation that measures the software complexity at ERV. However, if ERV knows the level of complexity of their work they have good chances of combining this factor with other attributes of the software product and process to create an overall measure of project performance.

Common metrics for software projects are for example man-hours spent on the project, produced number of lines of code, total development time for the project, and total development costs. These metrics are relative easy to measure since they are concrete. Of course the results of the measures can be influenced by external factors but there are no uncertainty in what is measured. Other metrics, also of interest for the management of software projects, are quality and productivity. But how do you measure quality and productivity in a software project? In contradiction to the other metrics mentioned, quality and productivity are very subjective metrics. What is good quality and what exactly do we mean by productivity? Is good quality when there are absolutely no bugs in the program, or if the customer is satisfied even though there are some bugs? Is good quality when the code structure is compressed and effective or if the program is easy to read and maintain? Has the project been productive if many lines of code have been produced in a short time, or if the project took longer time but produced advanced functionality? These questions are reminding of the importance to define why to measure a project and what the expected achievements of the measurements are.

If all metrics were as concrete as total cost etc, it would not be that difficult to measure a software project. The range of factors that influences software projects are however very wide, and it is perhaps impossible to develop software metrics that can comprise them all into one measure. Developing software is in many ways a very creative process and therefore it is not very easy neither to measure quality nor productivity. As a simile - How do you measure if a poet is productive and creates poems of good quality? However, as the demand of improved management increases, as well as the demand of more accurate reports of productivity, organizations have to develop and refine their management techniques. This can be achieved by using more accurate methods to measure project performance.

1.2 Research area

Measuring software attributes with the purpose of improving software product quality and project team productivity has become a primary priority for almost every organization that relies on computers. As computers grow more powerful, the users demand more sophisticated and powerful software. The process of developing new software and maintaining old systems has in many cases been poorly implemented, resulting in large cost overruns and squandered business opportunities. The software problem is huge, influencing many companies and government organizations. One of the most frustrating aspects of today's software development problem for business executives is the large number of projects, which are delivered to customers behind schedule.

The application of software metrics has proven to be an effective technique for improving software quality and productivity. The groundwork research for the origins of the application of quantitative methods to software development was established in the 1970s (Coté, Bourque, Oligny, & Rivard, 1988). There were four primary research trends that occurred at that time, which have evolved into the metrics practices used today:

- *Code Complexity Measures*. In the mid-1970s, there was significant research activity for developing measures of code complexity. These code metrics was easy to obtain since they could be calculated by automated means from the product code itself. Early examples include McCabe's cyclomatic number (McCabe, 1976) and Halstead's measure of complexity (Halstead, 1979).
- *Software project Cost Estimation.* These techniques were developed in the mid-1970s for estimating the effort and schedule that would be required to develop a software product, based upon an estimate of the number of lines of code necessary for implementation and other factors. Early examples include Larry Putnam's SLIM Model (Putnam, 1980) and Barry Boehm's COCOMO Model (Boehm, 1981).
- *Software Quality Assurance*. The techniques of Software Quality Assurance were significantly improved during the late 1970s and early 1980s. Of particular interest to quantitative methods is the emphasis that was placed on the collection of fault data during the various phases of the software life cycle (Möller, 1988).
- *Software Development Process*. As software projects became larger and more complex, the needs for a controlled software development process emerged. This process included defining the development life cycle by finite sequential phases, and placing more emphasis on software project management with better control of resources (Basili, 1980). The measurement of resources and resulting development costs were collected using corporate cost accounting systems.

In the 1980s, these four software engineering technology trends provided the foundation and impetus for the improved management of software projects by quantitative methods. Leading-edge practitioners and researchers began applying metrics for the purpose of improving the software development process. One may compare this approach with the analogous situation of a factory production process in which statistical quality control measurements are used to manage and improve the production process.

Today's practices of software metrics utilize global indicators, which provide insights into improving the software development and maintenance process. The improved process helps to achieve organizational goals established for improving software quality and project team productivity. Thus, even though the foundation for the methods used today were established more than twenty years ago, there has been a continuous development of these methods, and impulses have been drawn from disciplines such as statistics, mathematics and business economics, in order to create software metrics that are useful and applicable.

1.3 Disposition of the report

In Chapter 2, *Purpose*, we establish the purpose with this report and in Chapter 3, *Problem*, we state the set of questions that are answered in our report together with the limitations for our study. Moreover, Chapter 4, *Method*, spells out the method we have used to carry out our master thesis. In Chapter 5, *The Situation at Ericsson Mobile Data Design*, we are presenting the organization that is the object of our study. We will try to focus at the prerequisites that are important for our further discussion. Chapter 6, *Software Metrics*, is dealing with the software metrics' field of study. We are placing software complexity measures in a context, and are stating the reasons for using software metrics' at all. It serves as a frame of reference for our master thesis.

However, the main theoretical foundation for our discussion is established in Chapter 7, *Software Complexity*, where we investigate the concept of software complexity, its sources, nature and effects. This study leads to a more profound survey of different software complexity measures and methods in Chapter 8, *Structural Measures of Software Complexity*, and Chapter 9, *Algorithmic Complexity and Size Measures*. In Chapter 10, *Productivity, Quality and Performance*, we will try to explain how software complexity measures can be combined with other factors to find an overall measure of software productivity and quality. The field tests at Ericsson Mobile Data Design AB (ERV) of two measurement methods are outlined in Chapter 11, *Field Tests at ERV*. Finally, in Chapter 12, *Conclusions and Recommendations*, the work as a whole is discussed and recommendations for ERV are left together with our conclusions and suggestions for future studies.

In the end of Chapter 5-11 there is a short summary of the most important findings and contents of that chapter. It is supposed to serve as a reminder for the reader as well as a guide of how the chapter is related to the rest of the master thesis. Moreover, at the end of the report we have added appendixes for specific issues that we are approaching in this master thesis. We have also added a wordlist of the abbreviations used in the report.

2 Purpose

This master thesis was initiated from Ericsson Mobile Data Design AB (ERV). The purpose of our assignment at ERV was to investigate methods for measuring the complexity and productivity of software development projects. However, in our pre-studies we found that the quality of software projects also was an important aspect to consider. Therefore, our focus came to surround that field too.

Software projects are influenced by several external and internal factors generally gathered in the term s*oftware complexity*. Part of our mission was to define the different parts of software complexity and their effects on software productivity and quality. Next part of our mission was to choose some of the complexity attributes we found most essential for software measures. Since ERV develop real-time software systems, which have special characteristics, this had to be considered during the whole work. With our chosen complexity attributes as a starting point, we would examine existing software tools or methods that could measure these attributes. This would in the end lead to suggestions of how to measure productivity and quality of software projects at ERV.

This thesis leads to the final exam at the Department of Informatics for both of us. Due to different education programs, though, the graduate degree will be a Master Degree for Sofia Nystedt, and a Bachelor Degree for Claes Sandros. Since this assignment is a 20 week full time study project we will simply refer to it as a master thesis.

3 Problem

The problem of this master thesis consists of three logically related parts. By answering the questions of these three problem areas, the purpose of this master thesis can be reached. The scope of the problem narrows as we move from a general view of software complexity to a more ERV-specific one. Firstly, since ERV is interested in measuring software complexity, it is necessary to define this concept, with regard to its sources, nature and consequences. Specifically, we are interested in its relationship with other attributes of the software product and process, mainly productivity and quality. This leads us to the formulation of our first problem:

- How can software complexity be defined? Where does software complexity come from and what does it lead to? How is software complexity related to software productivity and quality?

Secondly, our interest is directed to available measures of software complexity. This means that ERV needs a comprehensive survey of which methods that are possible to apply to the projects at ERV. Therefore, our second problem is:

- How can software complexity be measured? Which methods and measurements are available for capturing different aspects of software complexity?

Thirdly, ERV has requested an analysis of which method or measurement that is best suited for ERV's purpose. The last part of our mission is therefore to create the basis for a choice of a software complexity measure that can be used for productivity and quality comparisons between projects. The third problem analyzed in this report is formulated as follows:

- Which method or measurement of software complexity should be chosen with regard to ERV's need of a comparative measure of project productivity and quality? How is this method or measure going to be implemented in ERV's system development process?

Our expectations of this field were that we would find a rather intricate picture of software complexity, with many different views of how it is influencing project productivity and product quality. We also suspected that there would be a wide range of methods available for measuring software complexity, but that each of them measured only some aspect of software complexity. Thus, the prospect of finding one method or measure that encompasses all parts of this concept was not so bright. The choice of one method would therefore be a question of priorities, and these priorities had to be based on the prerequisites at ERV.

3.1 Scope and limitations

It is obvious that it is not possible to cover the whole area of software metrics' methods. The time and the scope of this report do not grant us the privilege to make a fully comprehensive study of this subject. As the problem indicates, we concentrated on the parts of software metrics that were of immediate interest to ERV. This means that **the focus of this master thesis is on the concept of software complexity and its relationship with software development productivity and quality**. For reasons

spelled out in the following chapters we also touched upon other aspects of software metrics, but these parts were subordinated.

The time and resources at our disposal has also limited the depth of our analysis. It is difficult to describe software measurement methods without going into details. The reader may find some parts of this report harder to understand, since we have tried to limit the number of details in order not to digress too much from the subject. The time limit, 20 weeks of full study, has also meant that we have not been able to make all the detailed studies we needed to achieve an exhaustive analysis of the area of interest.

Moreover, in a small country like Sweden, the literature on software metrics in general, and especially software complexity, is not easy to find. However, we have had access to the literature available at the Scandinavian research libraries, and we have received the impression that this literature is representative of different views of the software metrics research area. Finally, at Ericsson Mobile Data Design (ERV) we have met interested and committed people, but sometimes our contact persons have had to give priority to other things. This meant that when we performed field tests at ERV we sometimes wished that we could engage more people for experiments and reference than were possible.

4 Method

The initial ambitions of our supervisors at ERV was that we would define the attributes of complexity, then to chose the attributes most essential for ERV's projects, and then calculate a complexity factor of software projects, to be able to compare complexity differences between projects. This numerical value, that would be generated automatically, in addition with the given values of Lines of Code (LOC), man-hours and error density would then be put together in a formula created by us for calculating software productivity.

As our knowledge of the field increased we arrived to the conclusion that we had to change the approach to the problem. First of all, our supervisors wanted us to find and use existing tools on the market for our measures. They had hoped for tools that could auto-generate complexity values from code. Those that exist are not really suitable for productivity measures but more for predicting error proneness. Then we also found out that there were alternative code size metrics to LOC which also included some complexity attributes. These functional size metrics were rather different methods, and therefore very well suited for productivity measures. Even though we found that these methods had to be counted manually we felt that they had to be included in our method of measuring project productivity. Due to these facts we changed our disposal of the work.

However, to describe how our work with this assignment evolved through time we would like to start from the beginning with our method description.

4.1 The preliminary plan

The project was originally divided into four part goals that signified a meeting with our supervisors at ERV to report on our progresses. These part goals were the following:

Stage 1. *Defining the attributes of software complexity*
After research on software complexity we would report our results and in collaboration with our superiors choose some of the factors of complexity interesting to focus on with concern to our further research of productivity measures.

Stage 2. *Examining and choose tools*
In stage 2 we would give a report on existing tools available on the market that can measure complexity and suggest two or three tools suitable for further examination. After more thoroughly examination of these tools we would choose the one ERV should use. The continuing work would be to use this tool on existing code from a completed software project at Ericsson.

Stage 3. *Complexity formula*
From our tests we would have a good foundation for creating a formula that calculated the complexity of software projects at ERV. This could be done simply by using one an existing tool or, if proven to be more accurate, a combination of several tools or other important attributes for software complexity.

Stage 4. *Productivity formula*
In the final stage we were supposed to report the results of our measures from the completed project and give suggestions of a formula on software productivity. With the given parameters of Lines of Code, error density and man time from ERV, and the complexity metric from our complexity formula, we would create such a productivity formula.

This was the preliminary plan of the method for our work when we started. With increased knowledge of the field we had to adjust our plans to fit in with existing methods of software metrics. Our refined stages turned out to be these:

Stage 1. *Defining the attributes of software complexity*
Stage 2. *Examining existing methods*
Stage 3. *Counting a completed project*
Stage 4. *Determine suggestions for Ericsson*

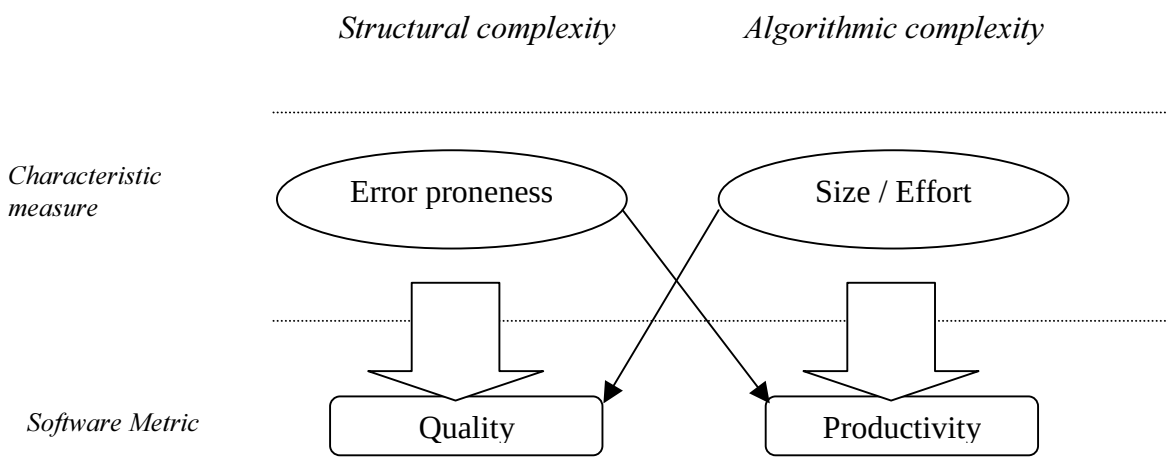
The following sections give a chronological and thorough description of our continued work with these stages.

4.2 The settled plan

4.2.1 Defining the attributes of software complexity

To establish a knowledge base of the subjects of software complexity and software productivity we started up by searching through the literature. Our keyword for the research of literature was s*oftware metrics*, which proved to be the primary word for software measurements as there were barely any hits when we searched on s*oftware complexity*. Several books where found that focused on the software development process and software metrics, but not so many that primarily discussed the attributes of software complexity. However, from our collected knowledge we learned that there was no summed up compilation of what the term complexity contains. We therefore developed our own model of complexity from which we initiated all further discussions. The model is divided into two main tracks where one focus on structural complexity and the other on algorithmic complexity. The characteristic measure of structural complexity is error proneness and for algorithmic complexity it is size and effort. As the simplified model of complexity shows in Figure 4:1, the track of error proneness lead to software quality and size/effort to software productivity. The crossing thin arrows indicate that one side also has effect on the other. The model is fully explained in Chapter 7, *Software Complexity*.

Figure 4:1 Simplified model of software complexity



4.2.2 Examining existing methods

In connection to our literature studies on software complexity we found existing methods for measures on error proneness and size/effort. In contradiction to what we had hoped there were no tools that automatically generated values from code. Some tools, though, that measured error proneness could be generated automatically from the data structure.

As our focus was on both productivity and quality, we wanted to examine the different methods we had found for both fields. However, our main interest was to find some tools that could be used to measure productivity, which meant that we searched for alternative measures of code size. We found a few interesting prospects to this kind of measure gathered under the term functional size measures. They were rather alternative measurement methods than actual tools, but measured software size as well as part of the algorithmic complexity. This was a very interesting progress for us. Unfortunately we learned that these methods only could be used by counting manually. After an extra meeting with our supervisors at ERV we determined to proceed with these methods, as reverting to LOC again was no alternative.

The methods picked out for further examination are shown in the Table 4:1. Each choice of method was based on their market use and acceptance in the software community according to literature (Fenton & Pfleeger, 1996; Jones, 1996).

Table 4:1 Methods discussed in this report

Structural complexity measures	**Algorithmic complexity measures**
McCabe's cyclomatic number	Lines of Code
Henry and Kafura's information flow measure	IFPUG's Function Points
Halstead's measure of complexity	SPR's Feature Points
	Full Function Points

Some of these methods were found in our literature and some were found on the Internet, especially information about Full Function Points, which is a quite new method under development.

4.2.2.1 Development of the performance concept

Our final goal with this assignment was to improve measurement methods of project productivity in some way. Traditional formulas of productivity only include software size and man-hours but not issues about algorithmic complexity or quality. In Garmus and Herron (1996) and Goodman (1993) we found that by comparing a productivity measure with a quality measure we could get a performance measure instead. This way we would take both aspects of complexity into consideration in our metric.

We chose one or two quality metrics and one productivity metric (functional size metric) to use in our performance measure. To decide which one of the functional size metrics to use we were supposed to test the functional size methods we had found on existing code from a completed software project at ERV. The choice of quality metrics was based on theoretical studies and considerations.

4.2.3 Counting a completed project

Our initial ambition was to measure the code with the three functional size measures, Function Point Analysis (FPA), Feature Points and Full Function Points (FFP). To be able to count the project we had to learn what the system did. To our help we had all the documentation of the system. It proved to take longer than we had expected to learn about the system, so we decided to not use the Feature Point Method as we began to run out of time.

4.2.3.1 Why choose FPA and FFP, but not Feature Points?

IFPUG's (International Function Point User Group) Function Point Analysis (FPA) is the most accepted and used method on the market. It is also continuously developing under the supervision of the IFPUG, which handles standardization of the method. Full Function Point (FFP) is an extension of the FPA, but more adjusted to scientific software. We believed that these methods were the most interesting methods to continue with. After continuously presenting to our supervisors the several methods we found during our work, they agreed that we should continue with FPA and FFP.

4.2.3.2 Counting procedure

The actual counting signified that we divided the project in accordance to the modules (or sub-systems) and counted each module as an application. This way we would be able to compare the results from each module and determine, in collaboration with the system developers, if the counting results were reasonable. If one system was twice as difficult to develop as another was, the results of the counts should show this by giving twice as many points.

Our counting of the modules were made according to an established pattern. First, the FPA and FFP methods were applied to a general type of documentation called Interwork Description (IWD). Included in this document were only the external parts of the modules, i.e. the communication and relationships with other modules. To get a more comprehensive picture of the modules we had to turn to individual documents for each module, called Implementation Specification (IS). These usually described the external inputs and outputs (I/O) to the module, and the internal parts, such as updating and erasing of local module data. The counting procedure was then made all over again based on the IS for each module. When applying the methods on this documentation we also used the actual source code as a support when the IS's passed over some important details.

The reason for choosing this approach with two separate counts was based on a request from people at ERV. They expressed their wish that we should investigate the possibilities of applying the methods on the more general documentation (IWD's) or if it was necessary to make a more detailed survey of the IS's and source code in order to get an accurate estimation of the software size and complexity.

During the tests we consulted some of the people involved in the project in order to receive guidance in some details of the system. Most of our work was devoted to understand the functionality of the different modules. The counting of the project took about one week, but to grasp the system in order to apply the methods correctly we had to initiate ourselves in the project for about three or four weeks before then.

4.2.3.3 Validation of our results

It is important to point out that the purpose of these tests were not to make a fully scientific examination of the suitability of the FPA and FFP methods on real-time systems. These studies have already been made *en masse* (Conte, Shen and Dunsmore, 1986; Jones, 1996; Grady, 1992; Desharnais, Maya, St-Pierre and Bourque, 1998). Our objective were rather to see which method is the most applicable and practicable when it comes to applying it to the systems developed at ERV. For this reason our only alternative to gain such knowledge would be to validate our results based on the opinions of the people involved in the system development. Three respondents, involved in the system development project, were therefore picked to perform this validation.

The reason for only choosing three respondents was two-fold. Firstly, some of the people involved in the project were no longer available at ERV. Secondly, many people who took part in the project did not have a complete picture of all parts of the project. They claimed that they could only have opinions of the modules they had actively been working on. This meant that only a few persons were possible as subjects of these tests.

4.2.4 Determine suggestions for Ericsson

From the knowledge base that we built up on software complexity; from literature studies, examination of existing methods and our tests of the methods on the completed project, we would give suggestions of how ERV could apply a measurement program from the conclusions we made.

One important factor that would effect our suggestions was that they had to be applicable almost at once after our work at ERV was done. This meant that they had to be easy to implement. Our approach was therefore to give suggestion applicable without too much effort. This implied that the suggested measurements might not be complete. However, as ERV become used to the measures they are free to develop their measurement methods further. Due to this tactic we would give ERV both short-term and long-term suggestions of the measurement methods we came up with.

5 The Situation at Ericsson Mobile Data Design

Ericsson Mobile Data Design (ERV) is established in the field of mobile data system design, as the name indicates, and is also the competence center for Ericsson as a corporation within this area. The company has experience of designing mobile data systems since the beginning of the 1980's but it was not until 1988 that the company itself was founded. Then it was a cooperation between Ericsson and Telia, and owned by them jointly. 1994 Ericsson took over the complete financial and operational responsibility for the company and it was incorporated in the Ericsson group of companies. ERV is situated in Gothenburg, but it is also operating in collaboration with other units within Ericsson all over the world (ERV, 1999a).

In order to give an explanation of our argumentation of which measures should be used, and to support the suggestions being made, we will try to sort out those conditions at ERV that are relevant. Specifically, we will give a picture of which kind of systems and products the company is dealing with, how their projects are working and finally how one can characterize their model of software development. This information will then form the basis of which measures that should be performed, in what way, and at what time.

5.1 Characteristics of the products at ERV

The business concept of ERV is to develop systems for future mobile data and telecommunications. One of the best-known products ERV has developed, a developing process that started in the early 1980's, is Mobitex. It is a land-based system for mobile data communications that has been installed on all continents and has in practice formed a standard for mobile data systems (ERV, 1999a).

To be able to understand what mobile data systems are, it may be useful to get a notion of in what kind of business the systems can be used. In the following section we will try to give some examples of who the users of the mobile data systems can be and where we can find areas of applications for these systems.

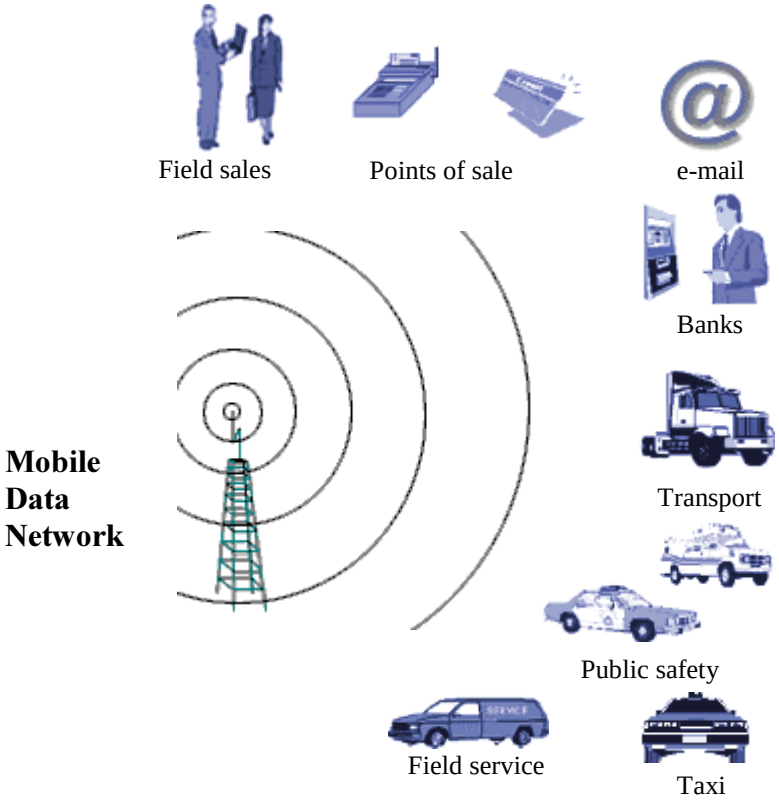
5.1.1 The users of mobile data systems

Mobile data systems are an increasingly important support for many activities in our society. By combining the datacom and telecom opportunities that the mobile telephone standards are giving us today there is an ever-growing need for applications that use mobile networks to transmit data in one form or another. In Figure 5:1 we show some examples of how mobile data systems can be and are being used.

For salesmen using laptops and mobile telephones the mobile data systems can help them for example to report orders to the head office. At *points of sale* the systems transmit information about the customer and his/her credit card to the bank that updates the account of the customer. The technique can also be used for sending *e-mail* messages in mobile networks and there are also prospects of using the systems for browsing the Internet. Today mobile automatic teller machines (ATM's) are not as common as the stationary ones, but as the mobile standards develop these and other services can be offered to *banks* by mobile systems. *Transports*, *taxis* and *field service* have an obvious use of mobile systems, since they are literally mobile. The systems can offer effective means for transmitting information about the traffic situation,

possible routes to the customer, information about orders etc. In the service of *public safety* mobile systems can even be used to save lives. Examples of such systems are the applications that are implemented in ambulances and use mobile networks to transmit information (for example ECG's and EEG's) about the patient to the hospital, where the physicians can make a diagnosis of the patient before he or she has arrived.

Figure 5:1 Use cases for mobile data systems



(ERV, 1999b (modified))

This picture of mobile data systems is of course not all-embracing. There are many fields of application that have not been mentioned here. In fact, finding and developing the needs for mobile data systems is one of the most important assignments for ERV today and even more so in the future. When new and more powerful standards are evolving, it is probable that still unexplored fields of application can be targets for mobile data systems.

It is obvious that business considerations are depending on the product that is developed. For some of the mobile data systems, such as those implemented in public safety vehicles, the reliability of the product is extremely important. It may even be necessary to exceed budget and time-plan in order to ensure that the system is reliable and of high quality. However, in other situations the timeliness of the product is the superior attribute. If we know that the market is ripe for the product in a year, we are forced to develop it in that year, otherwise we will end up with an obsolete product that our competitors already have developed. In that situation, we can consider to lower our quality requirements, in order to deliver the system in time. We will get back to this discussion in Chapter 10, *Productivity, Quality and Performance*, when

we are developing our notion of performance, an overall measure of productivity and quality.

We are now turning to the technical characteristics of this type of system, and the characteristic that is interesting from the viewpoint of this master thesis is its real-time feature.

5.1.2 Real-time systems

In this section we will use the same approach to explain the concepts in question as in the last section, i.e. by exemplifying. Later we will also give a more precise definition of real-time systems and point at some principal characteristics of this type of system.

The category of real-time systems is evolving rather than static. Examples might include:

- Radar systems
- Missile guidance systems
- Navigation systems
- Safety systems
- Telephone switches
- Satellite communications
- Automated process control systems

Real-time software and the real world are inseparably related. Real time cannot be turned back and thus, the consequences of previous influences may last for a long time and the undesired effects may range from being inconvenient to disastrous in both economic and human terms.

The main distinguishing characteristic of a real-time system is that the software must execute almost instantly, at the boundary of the processing limits of the Central Processing Unit (CPU). Secondly, the inputs could occur at either fixed or random intervals, selectively or continuously, which means that the processing pattern of a real-time system in most cases is parallel. The inputs can also require interruption to process input of a higher priority, often utilizing a buffer area or queue to determine processing priorities. Therefore the developers have to consider synchronization aspects. Thirdly, it is an on-line, continuously available, critically timed system that generates event driven outputs almost simultaneously with their corresponding inputs. This demand comes from the fact that the system must meet deadlines in order to satisfy real physical time constraints. Finally, real-time systems typically have long mission times. When they have been implemented and delivered to the customer it is supposed to be used continuously during several years, maybe decades. This implies that the system on the one hand has to deal correctly with situations that is expected to happen, and on the other to recover quickly and properly from extraordinary ones (Quirk, 1985).

We have chosen to summarize these characteristics and describe them in different and more concrete terms. Moreover, we encourage the reader to recall these characteristics further on when we are describing different measures of complexity and size, and above all when we present our ERV-specific conclusions and suggestions. These characteristics of real-time systems include:

- **Algorithms (mathematical and logical):** We will define the notion of algorithms in the following chapters that deal with the measures of software complexity. At this stage we content ourselves with the statement that algorithms are the solution of a given system development problem, in our case implemented in source code. There are usually more algorithms, and more complex ditto, in a real-time system compared to a MIS (Management Information System).
- **Memory constraints:** Data is frequently, but not always, stored in memory. Since real-time software sometimes is implemented in devices with limited resources, e.g. memory, the systems have to take account of these restrictions.
- **Timing constraints and execution speeds:** We have already touched upon this problem. The time is, as the name of the system type indicates, very important. The real world sets this limitation.
- **Continuous availability:** This characteristic has also been addressed earlier in this chapter and means that the demands on the system may occur in parallel rather than in sequence.
- **Process/event driven:** This aspect of real-time software is perhaps the most important when it comes to which size and complexity perspective we should choose. Usually real-time systems are constructed to wait for system calls and signals during the main part of the operation time. This also means that a real-time system consists of a large number of processes that in their turn is made up of even a larger number of sub-processes. We will see that this is an important feature when we want to measure size or other attributes of real-time software.

Now that we have tried to give a more comprehensive picture of what type of software that is produced at ERV, we will turn to the methods and organization that form the basis of the business. This will give us a starting-point for the discussion of how software metrics should be adapted to the prerequisites at ERV.

5.2 ERV project organization

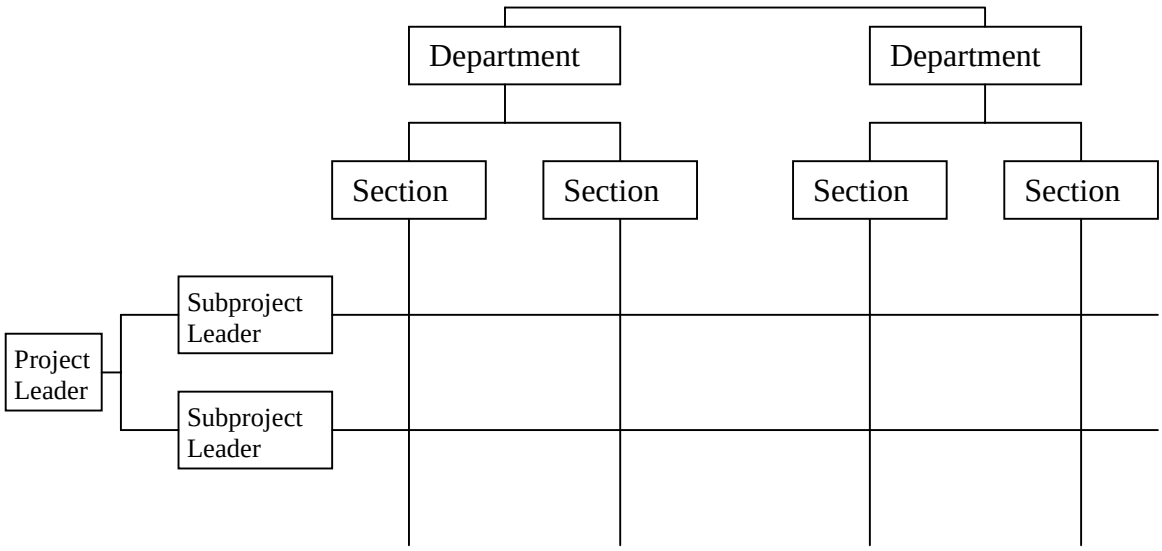
ERV as a company is divided into departments. Each of them is responsible for a separate product or area of competence. Apart from staff and more administrative departments there are today eight departments that are directly connected to the principal business. These are (ERV, 1999a):

- M: Responsible for the Mobitex Infrastructure
- K: Responsible for Mobitex Modems
- A: Responsible for design of systems associated with American Standards for mobile data communication (CDPD)
- N: Responsible for Switch Technology and design of systems associated with the 2nd generation mobile data communication standard in Europe (GPRS)

- Y: Responsible for Router Technology
- J: Responsible for design of systems associated with the 3rd generation mobile data communication standard in Europe (UMTS)
- S: Responsible for Systems & Air Interface
- L: Responsible for Network Management Systems and Information Design

These departments are further divided into a number of sections. Usually there are at least one design section and one section for system verification at each department. In addition, special functions such as user documentation, system development methods etc., are distributed to separate sections at the different departments, so that each department is responsible for at least one such special competence area.

Figure 5:2 Project organization at ERV



(ERV, 1999c (modified))

All development work is effected in the form of projects. A typical project consists of 40 to 120 people, normally lasts for about one year and is divided into several sub-projects. The projects are fully responsible for their work, on time delivery, product quality and functionality. Moreover, they have the financial and budget responsibility. The people that make up a project are drawn from different sections and departments. Usually is staff from other Ericsson companies also participating in the projects. This means that people that work at the same section do not necessarily have to work at the same project. Each project also has a project leader appointed from one of the participating sections or drawn from some other department at ERV. Moreover, people responsible for the different subprojects, subproject leaders, are appointed. Thus, even if it is a rather strict hierarchic organization the project organization at ERV is a complex one with people from different parts of ERV, from different Ericsson companies, and from different parts of the world (ERV, 1999a).

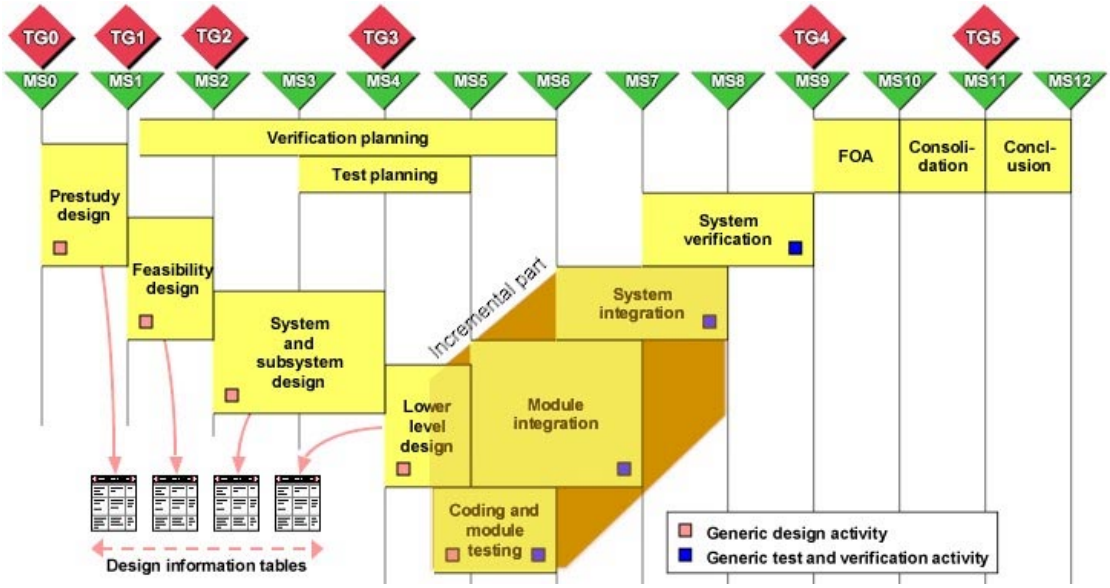
The tools used for software development are usually the same for every project. Moreover, the framework for managing the development process is standardized. ERV is an organization with a large group of new employees due to its recent expansion. This means that inexperienced personnel are mixed with experienced ditto. However, the mix is rather similar from case to case, and this implies that there are

usually small differences in the total project "experience quota" when comparing different projects. This is an important assertion to make when defining how we should measure the effort of our project. We will come back to this in Chapter 10, *Productivity, Quality and Performance*.

5.3 Darwin – ERV's model for system development

Before we begin to describe the Darwin model, we would like to state that this section is based entirely on internal oral and written information, that we have received at ERV (M. Timmerås, personal communication, 22nd March, 1999; ERV, 1999d). ERV has developed a model, called Darwin, for how project work should be arranged with regard to planning and controlling. It is based on a model common for Ericsson companies, but some changes have been made in order to adjust it to the prerequisites at ERV. We will give a quick overview of the model and focus on those parts that are important in order to understand our further discussion, especially our ERV-specific recommendations.

Figure 5:3 Darwin



(ERV, 1999d)

Darwin can be illustrated in a two-dimensional figure, in the shape of a V. The horizontal axis represents time, with the start of the project at the left end. The vertical axis represents the level of abstraction, which increases as we move upwards in the figure. The most concrete parts of a project, for example low-level design and coding, are situated in the bottom of the V. Activities, where knowledge of details is less important, are pictured in the upper part of the V. In other words, they are located at the start and end of the project.

The activities are not different from those normally identified in a system development process. First a prestudy is made to collect information about the general conditions for the project. A feasibility study is then made in order to see if the project is technically and economically practicable. When these studies are done the more concrete design activities are performed; first system and subsystem design and then

lower level design and coding. During the time that these activities are performed the planning of verification and test activities are also made. During coding some module testing is carried out. Moreover, during and after the code phase is completed, module integration and system integration can be executed. In the end of a project, focus is put on the system verification and installation of the system at the customer, before the project is consolidated and concluded.

The so-called milestones (MS) and tollgates (TG) guide the management of the project. Milestones are focusing on the project history, i.e. what has been accomplished, to be able to issue early warnings if something is delayed or not performed as planned. Tollgates are instead future oriented, and they are supposed to work as points in the development process where decisions of the further direction of the project are made. The idea behind this construction is that there is a passage criteria for each milestone, which specifies which actions and documents that must be performed before the project can pass the milestone. The model also specifies which measurements that should be made between two milestones.

Moreover ERV is applying incremental development. Simplified this means that the product is created by adding functionality in partly parallel processes. The design and test phases of the development process are in a way run many times. One can illustrate this by comparing the software with an onion. Just like an onion consists of many layers, incremental development of software mean that you start with a kernel of functions that you complete before you add “layers” with more functionality. In this way you have a product ready for release each time you have accomplished one “layer”, and you do not run the risk to have a half-finished product in your hands when the project is over.

Darwin has been newly developed by ERV, and is going to be implemented in the projects of the company. It has been the expressed wish of the company that we should connect our findings and recommendations to this model in one way or another. In the following chapters we are going to lay the theoretical foundation for these conclusions.

5.4 Summary

In this chapter we have spelled out the circumstances at ERV that we find important for the further discussion in this report. We have defined the area of business for ERV, by exemplifying the users of mobile data communication systems. Applications can be found in such areas as field sales, point of sales, e-mail and Internet, bank transactions, transport, public safety, taxi and field services. Moreover the systems designed at ERV can be described as real-time systems. The main characteristics of such systems are the great number of mathematical and logical algorithms, memory constraints, timing constraints and execution speed, the continuous availability and most importantly that they are process or event driven.

The line organization is hierarchical with departments as the main unit, which in turn are made up of sections. Each project can consist of up to 120 persons and lasts for a long period, usually a year or more. The people in a project are drawn from different departments and sections at ERV, as well as from other Ericsson companies, which means that a project is a rather complex structure, and takes the form of a matrix

organization. ERV is using their own model of system development, called Darwin, which specifies the activities and documents that should be accomplished at a specific time during the development process. Now we are turning to the theoretical parts of this master thesis, beginning with defining software metrics.

6 Software Metrics

When you can measure what you are speaking about, and express it into numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: It may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science.

Lord Kelvin (1824-1904)

Before we begin our discussion of complexity and its nature we need a more comprehensive understanding of what software metrics is and why we would like to use measurements to evaluate the software process. This section is trying to answer questions such as: Why are we measuring software projects? What kind of information do we expect to get from the measurements? Who would want this information? To reach this perception we need some kind of definition of the concept "software metrics". That is where we start our investigation.

6.1 Definitions of software metrics

Intuitively one could guess that "software metrics" is involved with numbers and measuring different aspects of the software development process. But to structure our own minds, and to give a more precise idea of what we mean by "software metrics", we need a definition. If we turn to the literature we can find several such definitions, which give more or less the same interpretation of the term.

Goodman (1993) defines software metrics as "the continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products" (p. 6). As one can understand, this definition covers quite a wide field of application, but the main focus is on improving the software process and all the aspects of the management of that process.

The main situation for use of software metrics is in decision making, which is emphasized by Grady (1992): "Software metrics are used to measure specific attributes of a software product or software development process … they help us to make better decisions" (p. 1). This definition also pinpoints one of the problems of software development today: the lack of information for predicting and evaluating software projects. We will come back to this in the following sections.

Some of the authors make a distinction between *software measurement* and *software metrics*. They mean that the terms measure and measurement are more mathematical correct when discussing the empirical value of different measured objects in the software development process. Further they establish that a metric is a criterion used to determine the difference or distance between two entities, for example the metric of Euclid, that measures the shortest distance between two points. These authors claim that the use of the term software metrics is built on a misunderstanding of the correct meaning of it, and that software measurement should be used instead (Zuse, 1991). Despite this formal difference between the terms, we have chosen to use software

metrics, for the simple reason that we feel it is the most applied and acknowledged term in the literature as a denomination of the phenomenon we are going to study.

Yet another aspect of software metrics that is often emphasized when defining the term, is that it can be applied during the whole development lifecycle, from initiations, when costs must be estimated, to monitoring the reliability of the end product in the field, and the way that the product changes over time with enhancement. Used correctly and consistently a project can therefore benefit from the software metrics during all stages of the software development (Fenton & Pfleeger, 1996).

Sometimes one can come across attempts to classify different types of software metrics. One such distinction is made between primitive and computed metrics. Primitive software metrics are directly measurable or countable, such as counting lines of code. Computed software metrics use some kind of mathematical formula to calculate the value of an attribute of a software project, such as the productivity measure of non-comment source statements per working month (Zuse, 1991).

To sort out the measurements that are used for high-level decision making the term "global metrics" is often used. This expression is most often used when referring to metrics, which is giving as a result information about size, product, and development process quality that are of interest to the managers of a software development or maintenance activity. This notion implicates that there could also be a such thing as a "local metrics", which is tied up with a more limited practice of software metrics during one phase of the product life cycle, for example code measures for a separate module. These small-scale metrics, usually called "phase metrics", could be combined for a high-level use, and according to our definition, phase metrics are in this way the constituent parts of the managerial global metrics (Goodman, 1993).

This discussion has led us to a definition of software metrics that we are going to use in this report. **Software metrics can be defined as the practice of measuring different attributes of the software development process and products in order to get useful and relevant information for efficient management during the entire development process**. Our view of software metrics is goal-oriented (or top-down). The primary problem for us is not what kind of metric(s) we should use, but rather what the goal of the measurements is and which kind of information we would like to come out of it. Only when we know the context of the metrics, the purpose of it and the people interested in it, we can correctly identify and evaluate the applicable measurements.

6.2 Why software metrics?

Now that we have a more established idea of what software metrics is, we also need to ask ourselves if and why software metrics matters. Why do we need to measure software? One way to answer this question is to identify the problems that could arise if we do not use software metrics in our projects. We have identified at least three groups of difficulties for developers and managers, who do not have a notion of software metrics:

1. They cannot set up measurable goals for their software products, since they do not know if they have reached them. For example, they can promise that this or that product should be user-friendly, reliable and easy to maintain, but as long as they do not clearly and objectively specify what they mean by these terms they do not know if they have met their goals. Tom Gilb (1988) has summarized this in his Principle of Fuzzy Targets. This principle says: *Projects without clear goals will not achieve their goals clearly*.
2. For most projects it is rather easy to establish the total cost, but it is harder to distinguish the costs at different stages of the software development process from each other, for example the cost of design from the cost of coding or testing. One reason for many complaints from the customers is also the failure to give a correct estimate of cost. If the managers cannot measure the components of cost it is almost impossible to control the total cost, and consequently hard to give an accurate quotation to the customer (Bache & Bazzana, 1994).
3. Finally, developers and managers fail to quantify or predict the quality of the products they produce. Thus, if the customer want to know how reliable a product will be, or how much work will be needed to change the product, they cannot give him the answer. The result of this is that the customer, since he is lacking valuable information, that perhaps other companies supply him with, recognizes that he is taking a risk if he chooses their product and therefore purchases a substitute (Fenton & Pfleeger, 1996).

Based on this inventory of software development pitfalls we can list three basic activities for which measurements are important. First, we can identify measures, which are used to *understand* what is happening during the different stages of development and maintenance. Through measurements we can see clearly the relationships among activities, which factors that influence the development process and how they can be influenced. Second, software metrics can help us *control* the activities in our projects. When we understand the relationships, we can use our goals and baselines to try to predict what will happen and make changes to processes and products in order to meet our goals. Third, measurement supports the activity to *improve* our processes and products. For example, by sorting out those parts of the project that does not meet our quality requirements, and deposit more resources to monitoring these parts, we can improve our overall quality.

6.3 Recipients and use of software metrics information

We have already shortly described which members of the software development team that may be interested in the information we get from software metrics. We choose, perhaps somewhat roughly, to separate them into two groups: managers and engineers. However, we acknowledge that there could be many different kinds of managers (economical managers, software quality managers, technical managers, staff manager etc.) and engineers (“design engineers”, “code-generating engineers”, “testing and verifying engineers” etc.). If we leave this distinction out of consideration, we can identify a wide range of useful information for these two groups. In addition, the interests of customers/users/clients can be satisfied by software metrics’ information.

6.3.1 Managers

One question that most managers have is how much a process cost. If we can measure the time and the effort we use in the software production, we can also understand not only the cost of the total project, but also how the different parts contribute to the whole.

When we have a measure of cost or effort we can also combine this with a measure of size, in order to get a notion of the productivity of the project. By collecting productivity information about a large set of projects, we can then make attempts to predict the cost and duration of future projects.

Software metrics' information can also be used to evaluate the quality of the products we use. For instance, different kinds of fault measures can be used to assess the code being produced for comparing, predicting and setting targets for process and product improvement. But the relative occurrence of faults is not the only measure of software quality. In order to satisfy the customer we also need a high level of functionality. If we can measure usability, reliability, response time, and other characteristics we can also find out if our customers will be happy with both functionality and performance.

An overall objective of most managers is to improve their business by focusing on those activities that will generate the largest revenue relative to the cost of the improvement. If we can measure the time it takes to perform each development activity, and calculate its effect on quality and productivity, we can also weigh the costs and benefits of each practice to determine if the benefit is worth the cost. Another way of improving the business is for example to compare two different design methods and measure the results to decide which one is the best (Fenton & Pfleeger, 1996).

6.3.2 Engineers

One obvious advantage of software metrics, that we already have touched upon, is that different attributes of the software process are objectively measurable and can therefore be expressed in numbers. The developers then have the opportunity to translate the requirements from words to testable entities. For example, the reliability requirement can be replaced by one that states that the lower limit for mean time to failure is 15 elapsed hours of CPU time.

If we measure the number of faults during the different phases of the software product life cycle, and build a database on this information, it is also possible to set up models of expected detection ratios. These models can help us to evaluate future testing efforts and decide whether they have been successful or not, i.e. if it is probable that we have found all the faults.

The measurement of the characteristics of the products and processes can also tell us whether we have met our quality standards and our quality goals. For instance, if we have specified that the software should not contain more than 20 failures per time unit, we can easily control this during the test phase. But software metrics are not only useful for a retrospective view on the development process; we can also use it for predicting attributes of the future products and processes, such as probable system size, maintenance problems and software reliability (Fenton & Pfleeger, 1996).

6.3.3 Customers

Software metrics is of obvious interest for the people who will use the system produced. Firstly, it is important for them to have information about quality, cost, time consumption etc., to be able to compare different products or companies with each other. When this information is available for all the alternatives that a customer has, he can make a more rational and well-founded choice. On the other hand, if the information is not available for some alternatives, the customer is more likely to choose those products from companies that have bothered to gather the information.

Secondly, information from a software metrics' program is meaningful for the user, when he would like to evaluate a project developed or product produced by another company (outsourcing). Measurements can then be used to assess the quality of a software system. The question is whether the system have the reliability, maintainability, usability, efficiency etc., that we would like it to have. Software metrics can also be used to discover if a project has not performed as good as we have hoped relative to the cost of the project, i.e. those projects with too low productivity (Möller & Paulish, 1993).

Evidently, there are problems associated with companies giving this type of information to its customers. If competitors can capture this information they can use it for own purposes, for example by discrediting the measured company in marketing. "Bad" figures of e.g. fault density or programmer productivity can do a great harm to a software company, and thus it is likely that it will retain the information if it can suspect that it will be public if it is released to the customers.

6.4 The components in software metrics

Software metrics includes many types of models and measures used in the situations described above. There are many proposals in the literature of how to classify these areas (Möller & Paulish, 1993; Fenton & Pfleeger, 1996; Ohlsson, 1996; Grady, 1992), and we have tried to summarize them in the following categories:

1. *Cost and effort estimation models.* The aim of these models is to predict the total cost of a software development project mainly at the requirement stage, but also to track the costs during the whole product life cycle. An example of such a model is Albrecht's Function Points model that we will return to in the following chapters. The models often share the approach of effort expressed as a function of one or more variables (for example size, capability of the developers and level of reuse). Size is usually computed by counting Lines of Code or number of functions points.
2. *Productivity models and measures.* When combining measures of size and effort or cost there is the possibility to reach a productivity measure. Based on the collection of productivity data from finished projects, managers can also build models for assessing and predicting staff productivity in future projects. These models and measures are on different levels of sophistication from the traditional ones, that divides size by effort, to ones that take more factors into consideration, such as quality, functionality and complexity.
3. *Quality models and measures.* As we have noticed productivity cannot be viewed in isolation. The speed of production is meaningless if the product is of inferior quality. This discovery has led software engineers to develop models of quality whose measurements can be combined with those of productivity models.

4. *Reliability models.* Most quality models include reliability as a factor, but the need, above all generated from the customers, for reliable software has led to the specialization in reliability modeling and prediction. Reliability models are usually statistical models for predicting mean time to failure or expected failure interval.
5. *Structural and complexity metrics.* Some quality attributes, like reliability and maintainability, are not measurable until the operational version of the code is available. To be able to predict which modules in a system that are less reliable than others, different predictive theories have been established to measure structural attributes of the software to support quality assurance, quality control and quality prediction. Examples of such theories are Halstead's measures of effort, difficulty, volume and length, as well as McCabe's cyclomatic number.

6.5 Summary

Although the discipline of software metrics is a rather young one, it has already spread to many areas of the software development process. In this chapter we have tried to give a definition of the concept, and explain in which areas and for whom it is useful. We developed our own definition of software metrics, based on the literature, and we established that software metrics is the practice of measuring different attributes of the software development process and products in order to get useful and relevant information for efficient management during the entire development process.

Three main reasons for using software metrics can be identified. Firstly, if we are not able to measure our products and processes, we cannot set up measurable goals, since we do not know if we have reached them. Secondly, we may want to distinguish costs at different stages of the software development process from each other. This is possible with software metrics. Finally, the customers will not choose our products if we are not able to quantify or predict the quality of the products we produce.

We also distinguished three main recipients of software metrics information: managers, engineers and customers. *Managers* use it mainly for controlling and predicting the cost of projects, but also to evaluate the quality, and calculate the effects of a more cost-effective solution. The *engineers* are more concerned with the quality of the system, and especially if the system is meeting the requirements. *Customers* are also very interested in the quality of the system, but mainly from their own viewpoint and according to their own requirements. Problems with giving this kind of information to the public can also be recognized.

In the end of the chapter, we tried to categorize different components in software metrics, and we came up with the following suggestion: cost and effort estimation, productivity models and measures, quality models and measures, reliability models, performance evaluation and models, and finally structural and complexity metrics.

7 Software Complexity

Most experts today agree that complexity is a major feature of computer software, and will increasingly be so in the future. Some even states that computer programs are the most complex entities among human products (Brooks, 1995). But what is software complexity? From where does it originate? What are the apparent effects of it?

When studying the literature about the concept software complexity one is struck by the fact that there is not much work done about the origins and nature of software complexity. Almost all attention has been given to which effects it has on software projects, above all the costs and quality of the product. This is a natural development, since the incentive to care about software complexity is derived from the need of software project managers and engineers to control and predict project productivity and quality, as we discussed in the earlier chapters. However, to be able to better understand how the complexity influences other attributes of software projects, we will explain how complexity is created and what it is made up of.

In this chapter we will summarize some of the research about software complexity and its findings. In the end of the chapter we will build a model of software complexity and explain how it is connected to other factors such as error-proneness, size, reliability, maintainability, quality, productivity etc. We are not picturing the model until in the end of this chapter (Figure 7:1), but the reader may want to have a look at it during the reading, to comprehend the relationships established.

7.1 Sources of software complexity

One way to start the quest for a model of software complexity is to examine the origins of the concept. The intention is to be able to define what we mean by software complexity by examining where it comes from. To solve this problem we set out by looking at the different phases of the software product life cycle and try to derive different types of complexity from these stages.

Although there is very little written about the origin and nature of software complexity, some suggestions of a definition can be found. Most of them are based on the notion that **software complexity is the degree of difficulty in analyzing, maintaining, testing, designing and modifying software** (Zuse, 1991; Ohlsson, 1996; Fenton & Pfleeger, 1996; Grover & Gill, 1995). Our approach to the concept of software complexity, in this master thesis, will be similar to the one of these researchers.

With this basis it could be expected that different types of complexity are created during each stage of the product life cycle. Hence, we are suggesting a twofold classification of complexity:

- **complexity of the problem**, which is the inherent complexity, created during the requirements phase, and;
- **complexity of the solution** (also referred to as **added complexity**), which is the complexity being attached to the complexity of the problem. This type of complexity is added during the development stages following the requirements phase, primarily during the designing and coding phases.

Another way to look at software complexity is to see it as the resources needed for the project, or the efficiency of the project. With this approach, the complexity of a problem can be defined as the amount of resources required for an optimal solution of the problem. Then complexity of a solution can be regarded in terms of the resources needed to implement a particular solution. These resources have at least two aspects: time, i.e. computer time and man-hours, and space, i.e. computer memory (Goodman, 1993).

The idea behind this typology is that when we can measure how and when complexity is brought into the project, we will also know which stage in the product life cycle we need to focus on, to be able to control and hopefully reduce the software complexity. The advantage of this approach is also that by concentrating on how to measure complexity at different phases of the development, we can create models for predicting how much complexity will be added to the project later in the process.

The fact that complexity is created over time makes it also hard to define and measure. How are we going to summarize the complexity deriving from the problem, the design and the code in one single number? Researchers have recognized this difficulty, and therefore suggested that complexity should be discussed from several perspectives and that there are several characteristics of software that must be measured to get a general understanding of what complexity is (Zuse, 1991; Fenton & Pfleeger, 1996). This leads us to an attempt of defining what we mean by software complexity.

7.2 Nature of software complexity

That complexity is strongly connected to the amount of resources needed in a project is something that is stated by most of the researchers of software metrics (Jones, 1996; Fenton & Pfleeger, 1996; Möller & Paulish, 1993; Goodman, 1993; Grady, 1992). The notion is that a more complex problem or solution is demanding more resources from the project, in form of man-hours, computer time, support software etc. A large share of the resources is used to find errors, debug, and retest; thus, an associated measure of complexity is the number of software errors.

But since our intuition tells us that a solution that is more complex than another also is more likely to need a larger amount of effort, the notion of complexity is naturally bound to the concept of size. We think that by examining these relationships, firstly between complexity and size and secondly between complexity and number of errors, we can also understand how complexity is connected to productivity and quality.

We have already recognized that there are many categories of software complexity, but that it is hard to combine all these categories in one overall measure of complexity. Rather, our approach is to interpret complexity in different ways, and then try to build a model of how these different aspects of complexity is influencing productivity and quality (we will come back to this later in the chapter).

In that case, how are we going to properly define software complexity? Based on the literature (Fenton & Pfleeger, 1996; Zuse, 1991; Ohlsson, 1996) we would like to suggest that software complexity is made up of the following parts:

1. **Problem complexity** (which is also called **computational complexity**) measures the complexity of the underlying problem. This type of complexity can be traced

back to the requirements phase, when the problem is defined. If the problem can be described with algorithms and functions it is also possible to compare different problems with each other, and try to state which problem is the most complex.

2. **Algorithmic complexity** reflects the complexity of the algorithm implemented to solve the problem. This is where the notion of efficiency is applied. By experimentally comparing different algorithms we can establish which algorithm gives the most efficient solution to the problem, and thus has the lowest degree of added complexity. This type of complexity is measurable as soon as an algorithm of a solution is created, usually during the design phase. However, historically algorithmic complexity has been measured on code, where the mathematical structure is more apparent.
3. **Structural complexity** measures the structure of the software used to implement the algorithm. Practitioners and researchers have recognized for a long time that there may be a link between the structure of the products and their quality. In other words, the structure of requirements, design, and code may help us to understand the difficulty we sometimes have in converting one product to another (as, for example, implementing a design as a code), in testing a product (as in testing code or validating requirements, for instance) or in predicting external software attributes from early product measures.
4. **Cognitive complexity** measures the effort required to understand the software. It has to do with the psychological perception or the relative difficulty of undertaking or completing a system. However, since this aspect of complexity is more an object of study for the social scientists and psychologists, we are not considering it in this report. Nonetheless, it is important to notice that the human mind is a constraint for the software process, and that it influences the attributes of the software we want to measure, such as quality and productivity.

Algorithmic and structural complexity (together with cognitive complexity) can be used to measure the complexity of a solution and especially the added complexity. Problem complexity is instead directly connected to the complexity that is generated by the originally requirements. Therefore, it is obvious that the aspects of complexity that are measurable during the software development process, are algorithmic and structural complexity. But measuring the problem complexity is also of interest to us when we want to predict the effort or resources needed for a specific project. By comparing new problems with old ones and considering the effects of the solutions to the old problems, we are able to predict the attributes of the solution to the new problem, such as cost, fault density, size etc.

Thus, we will neither try to find a general definition of software complexity, nor try to find an overall measure of it. Rather, we will use this typology of complexity in order to sort out those measures of software complexity that can help us understand the effects of complexity and the connections to project productivity and quality. However, to be able to do this we need to specify the consequences of complexity.

7.3 Effects of software complexity

The reason to why we bother about complexity is that it is related in one way or another to more important attributes of the software process. But how does these relationships look more in detail? This is the subject for the following section. According to our view, software complexity is a determinant factor for two of the main attributes of the software product: **error-proneness** and **size**.

7.3.1 Error-proneness

Throughout this master thesis a number of fundamental process entities will be referred to. The definitions in this section are based on the framework of Fenton and Pfleeger (1996). A **defect** is a result of an **error** during the software development process. The error occurs because of the human factor, often of ignorance or negligence. A **fault** is a defect, which has persisted until the software is executable and is at some time discovered. A fault that is discovered manifests itself in a **failure**. **Error-prone** modules are the modules that is most likely, statistically, to have a high proportion of errors, **fault-prone** modules are the modules that have the highest proportion of faults, and **failure-prone** modules are the modules with the highest proportion of faults discovered after release. The idea with identifying error-prone modules is that we have the possibilities and the time to correct the errors and prevent them from resulting in faults and failures. When we are able to measure fault-proneness and failure-proneness we have most often reached a point in the development process where we can not take care of the problem.

The main idea behind the relationship between complexity and error-proneness is that when comparing two different solutions to the same computational problem, we will, provided that all other things are equal, notice that the most complex solution is also generating the most number of errors. This relationship is one of the most analyzed by software metrics' researchers and previous studies and experiments have found this relationship to be statistically significant (Curtis, Sheppard & Milliman, 1979; Henry, Kafura & Harris, 1981; Shen, Yu, Thebaut & Paulsen, 1985).

This confirmation of the existence of a relationship between software complexity and errors, and understanding the characteristics of the relationship, is of important practical benefit. Since a great deal of software development costs are directed to the software integration testing, it is crucial for the project performance to possess the instruments for predicting and identifying the type of errors that may occur in a specific module (Basili & Weiss, 1984; Boehm, 1981).

We have found at least three ways where error-proneness can influence quality and productivity: through the **usability** and **reliability** of the software and through the **need of changing** the system. The concept of *usability* is connected to what the customer expects from the product. If the customer feels that he can use it in a way that he intend to, he will more likely be satisfied and regard it as a product with high quality. Thus, a large number of errors in software are presumably something that would lower the usability of the program.

The *reliability* of a system is often measured by trying to determine the mean time elapsed between occurrences of faults (the result of the software errors) in a system. The idea is that a relatively more reliable product is more stable and has fewer unexpected interruptions than a less reliable product. However, a product impaired by many errors must also be *changed* in one way or another, in order to lessen the number of faults. By trying to prevent the errors from arising or by detecting them and then try to correct them afterwards we can take care of the errors. Either way, a module that has more errors also needs more resources to remedy it (Ohlsson, 1996; McCarty, 1999).

The expectation of the system developer is that the usability and reliability of the software can be improved over time, but in order to do that we need to use more

resources to make improvements in the product. Thus, the error-proneness influences both the quality and the productivity dimension of software. By creating a more reliable and stable product we are building quality into it, and by preventing and correcting errors we are hopefully improving the overall quality. But the need for change is also restraining the productivity through the resources that is used for all the activities associated with changing the software: preventing, detecting and correcting errors. When we spend more time on these activities our overall productivity decreases. Thus, productivity and quality is interconnected by the factor of error-proneness. Products with relatively high quality do not need as much corrections, and therefore the time used for error-correcting and error-preventing activities is shorter. From this viewpoint it is therefore correct to say that project productivity increases as quality increases (Burr & Owen, 1996).

7.3.2 Size

The same amount of effort as have been directed to study the relationship between complexity and error-proneness, has not been aimed at analyzing the connection between complexity and size. However, most researchers in the field of software metrics would agree that the complexity of the problem and the solution is one of the factors that determine the size of the project (Fenton & Pfleeger, 1996; Treble & Douglas, 1995; Symons, 1991). All other things equal, it is obvious that a more complex problem will need more effort to be solved, and a more complex solution will need more resources to be implemented. However, to elucidate the connection between size and productivity, we will need to outline this relationship more in detail.

We can identify three results of software size: it influences the testability or **maintainability** of the software product, the **understandability** of the software product, and the **computational power** (time, memory etc.) needed to implement the software product. Maintainability is a requirement when we want our software to be easy to understand, enhance, or correct. Size is one of the factors that make the *maintenance* more difficult to implement. A larger product is in general also demanding a larger share of resources to modify and correct. An objection can be raised against our separation of maintainability and need of changing, which we discussed as a factor of error-proneness. The reason why we have made this division is that the maintenance activity includes much more than just preventing, detecting and correcting errors. That is, we do not need visible errors to have incentives for maintaining a software product. Rather, the maintenance activity also includes such things as preventive and perfective measures to improve the product in one way or another. The level of maintainability influences both the quality and the productivity. When we create programs that are easy to maintain, it is also easier to detect and correct our mistakes, and thereby reach our quality goals. Further, a maintenance-friendly module does not need as much time and other resources to keep in repair, and thus it is also acting on the project productivity.

In this context we should also acknowledge the interconnection between maintainability and change. If the maintainability of the system is high this will probably lead to lesser time spent on changing activities. Likewise, changes can be made to the system in order to increase the maintainability of the system. In other words, the maintainability and the level of change of a system are connected with each other, and this connection also influences the productivity and quality of the software.

Since the human mind is limited, the size of software is also influencing the ability for programmers and system developers to *understand* and comprehend how the software is structured. A large unit of code is more time-consuming and resource-demanding to familiarize oneself with, and it is even possible that it is too sizable to be able to comprehend at all. Thus, when added complexity in this way creates unnecessary large software modules it requires resources that could be used for other activities, and thereby it lowers the productivity of the project.

Finally, sizable software also uses more *computer power*, i.e. computer memory and time, for compiling, executing and testing the program. When building large systems the developers may have to sit idle and wait for the computer to make its share of the job, because the computational resources do not keep up with its human counterpart(s). When the human workforce is underutilized in this way the productivity is also suffering.

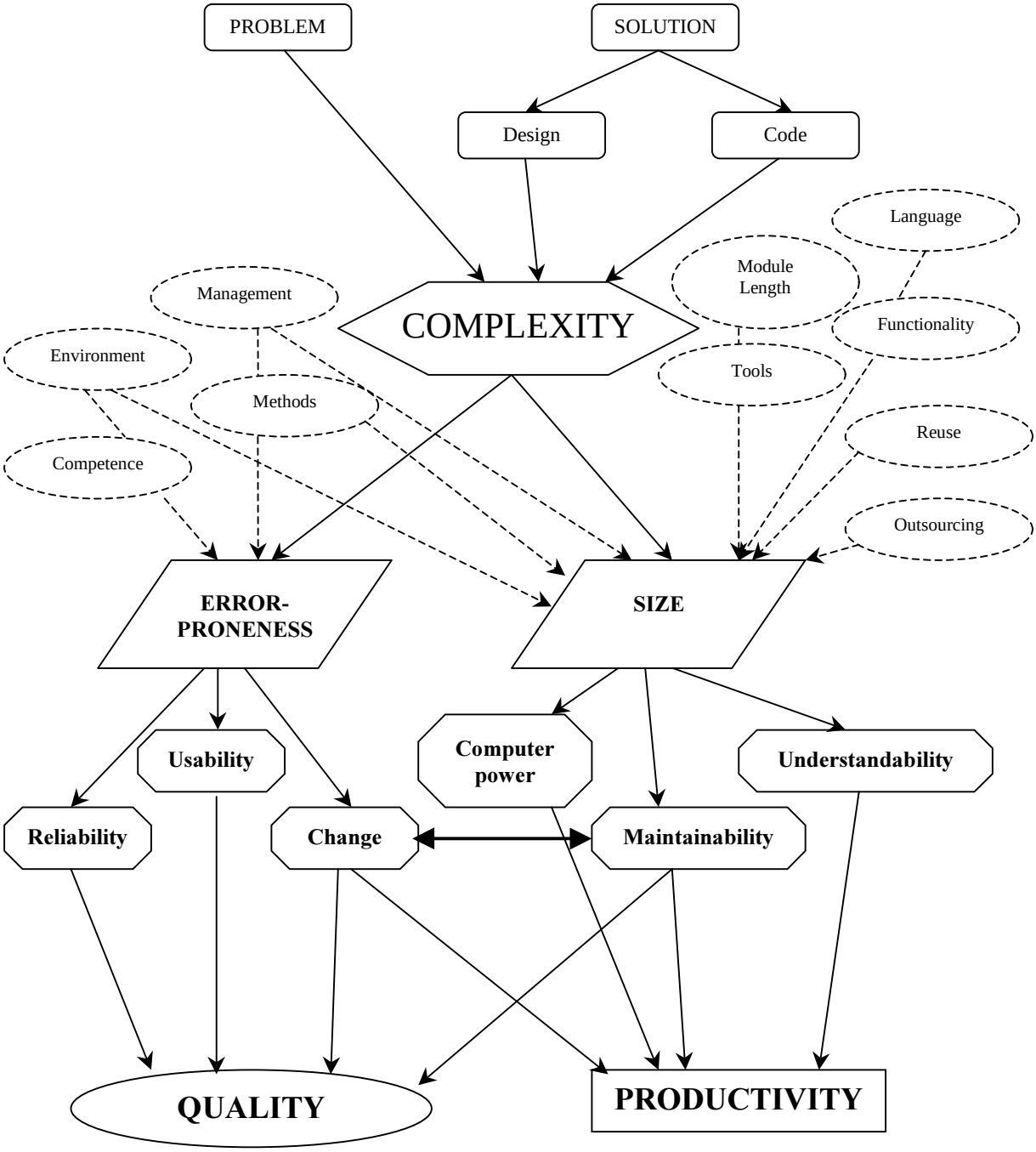
7.4 A model of software complexity

We started this chapter by saying that our goal was to build a model of software complexity. With the discussion of the sources, the nature and the effects of complexity we have also laid down the theoretical framework for a graphical disclosure of our understanding of the concept software complexity (Figure 7:1). It may look confusing in the beginning, but most of the relationships have already been explained in this chapter.

The sources of complexity have been discussed in the beginning of this chapter. The problem gives the project its inherent complexity, and further complexity is added during the design and code phase of the software development process. According to our model complexity is then uniting with other factors in determining the error-proneness and size of the system. We are not claiming that this model is encompassing all thinkable factors, but examples of components that are settling the occurrence of errors in the software and the size of the project are given. The *management* is interesting because a project can suffer as much from a defective leadership as it can benefit from a good one. Our belief is that a manager who has the ability to direct a project also creates better prerequisites for a low level of error density.

The *environment* in itself, e.g. the working place, the colleagues, the workload etc., is also a determinant factor for the error-proneness of the system. A “negative” environment usually means small possibilities of creating high-quality software. The use of good and relevant working *methods* is also important for the ability to reduce the number of errors in a program, as well as the *competence*, i.e. the experience and knowledge, of the people working with the project.

Figure 7:1 A model of software complexity



The methods, the management and the environment of the project also influence the size of software and the effort needed for creating it. But the magnitude of the software is also influenced by many other factors. One of the most obvious of these is the physical *length* of the project, measured for example by pages or lines of text (principally in the requirements and design phases) or code (in the coding phase and later). *Functionality* captures an intuitive notion of the amount of function contained in a delivered product or in a description of how the product is supposed to be. It is indicating that when we are creating a system that is supposed to “do more things” we are also creating a larger system.

The concepts of *tools* and *language* are in fact referring to much of the same phenomena. Most of us have experience telling us that we do not need as much effort when coding a system in a 4GL (4^{th} Generation Language) and with a visual tool, as we may need when implementing the same system in lower level languages, and with the help of a text editor. Moreover, most projects do not start from scratch. Usually some of the work has been done before, and we can *reuse* it in our new projects. Another alternative is to *outsource* some of the work to internal or external companies. Both these practices influence how much effort that is demanded from us in the immediate project.

The effects of errors and size have already been discussed in this chapter, and we will not repeat that analysis here. What we would like to point out once more is the intricacy of the model, how complicated the relationship is between complexity on one hand and productivity and quality on the other. This means that our prospects of finding one single measure, or even a set of measures, that reflects all aspects of complexity, is rather small. Rather, we will try to give an overview of the most important and also widely used measures of complexity, and what aspect of complexity that it is supposed to measure. We have chosen to divide these measures into two categories. The first one is complexity measures that are supposed to be used for predicting the number of errors in software, i.e. it is predicting the *error-proneness* of a program. The second is connected to the *size* and *effort* dimension of complexity and includes measures that combine complexity with other factors (mainly functionality) to predict and establish the size of a software product. The following two chapters will deal with these issues.

7.5 Summary

The concept of complexity is very hard to define, and it is even more so when it comes to software complexity. We proposed a rather loose definition of the concept and said that software complexity is the degree of difficulty in analyzing, maintaining, testing, designing and modifying software, i.e. software complexity is a subject during the entire software development process.

When we established this definition, we turned to the sources of complexity. They were divided in two main classes: complexity of the problem and complexity of the solution. The last-mentioned was also divided in complexity generated during the design and code phases of the development.

To explore the nature of complexity, we can classify the concept, in order to look at it from different viewpoints. One such classification was proposed in this chapter and it divided software complexity into problem complexity (also called computational complexity), algorithmic complexity, structural complexity and cognitive complexity.

The effects of complexity were divided in two main parts: error-proneness and size. A program that is more complex than another is also more likely to contain more errors and generate a larger system. The error-proneness, in its turn, influences such attributes of the system as the usability, reliability and need of change. Size, on the other hand, influences the maintainability, understandability and computational power needed for implementing the system. The discussion resulted in a model of software complexity that will form the basis for our continuous exploration of the subject.

8 Structural Measures of Software Complexity

8.1 Types of structural measures

In our model of complexity we found that the link between software complexity, size and productivity is not as simple and obvious as it seems. We would like to assume, that, all other things being equal, a large module takes longer to specify, design, code, and test than a small one. But we argued that also the structural complexity by means of its effect on the error-proneness of the program is determining the productivity level of the project. Thus, we must investigate characteristics of product structure, and determine how they influence the outcomes we seek. In this chapter, we focus primarily on code measures, but many of the concepts and techniques we introduce here can also be used on other documents produced during the product life cycle.

The notion of structural complexity can also be seen from different viewpoints, each playing a different role. We can identify at least three different aspects of structure (Fenton & Pfleeger, 1996):

1. control-flow structure
2. data-flow structure
3. data structure

The **control-flow** is concerned with the sequence in which instructions are executed in a program. This aspect of structure takes into consideration the iterative and looping nature of a program. Thus, whereas size counts an instruction only once, control flows make more visible the fact that an instruction may be executed many times as the program is actually run.

Data flow follows the trail of a data item as it is created or handled by a program. Many times, the transactions applied to data are more complex than the instructions that implement them; data-flow measures depict the behavior of the data as it interacts with the program.

Data structure is the organization of the data itself, independent of the program. When data elements are arranged as lists, queues, stacks, or other well-defined structures, the algorithms for creating, modifying, or deleting them are more likely to be well-defined, too. So the structure of the data tells us a great deal about the difficulty involved in writing programs to handle the data, and in defining test cases for verifying that the programs are correct. Sometimes a program is complex due to a complex data structure rather than complex control or data flow.

We will discuss each of these categories in the following sections. Moreover we will give examples of measures that are used to characterize these aspects of structural complexity. Since over hundred measures of structural complexity have been proposed (Zuse, 1991), we will have to limit ourselves to three types, one for each category. The reason why we have chosen these three measures is that they are the most discussed in the literature, the most frequently used, and, not least important, they are the original measures that most of the other measures have been developed from (Zuse, 1991; Fenton & Pfleeger, 1996; Ohlsson, 1996).

8.2 Control-flow structure

Measures of control-flow structure have been the interest of software metrics work since the beginning of the 1970's. In fact, they were the first ones proposed for measuring software attributes in general, and especially software complexity. The control flow measures are usually modeled with **directed graphs**, where each node (or point) corresponds to a program statement, and each arc (or directed edge) indicates the flow of control from one statement to another (Fenton & Pfleeger, 1996). We call these directed graphs **control-flow graphs** or **flowgraphs**. An example of one such graph can be seen in Figure 8:2.

Thus, directed graphs are depicted with a set of nodes, and each arc connects a pair of nodes. The arrowhead indicates that something flows from one node to another node. The in-degree of a node is the number of arcs arriving at the node, and the out-degree is the number of arcs that leave the node. We can move from one node to another along the arcs, as long as we move in the direction of the arrows. A **path** is a sequence of consecutive (directed) edges, some of which may be traversed more than once during the sequence. The reason why we explain these concepts is that this terminology is used in the measure we are going to describe. To be able to understand description of McCabe's cyclomatic number it is necessary to recognize these notions.

Normally we can construct a program using the structural primitives: **sequence**, **selection** and **iteration**. Since most programming languages have formalized ways of expressing these constructs, it is possible to extract a flowgraph from existing code. Most automated tools that are applied to source code also work in this way. They try to identify the structural primitives, then build a flowgraph model of this program, and by this time it is rather easy to compute the equation used by the measure. We will describe one of these measures, McCabe's cyclomatic number, which also was the first measure of software complexity proposed, introduced in 1975.

8.2.1 McCabe's cyclomatic number

McCabe (1976) proposed that program complexity could be measured by the cyclomatic number of the program's flowgraph. The measure is expressed in the equation:

$$\mathbf{v(F) = e - n + 2}$$

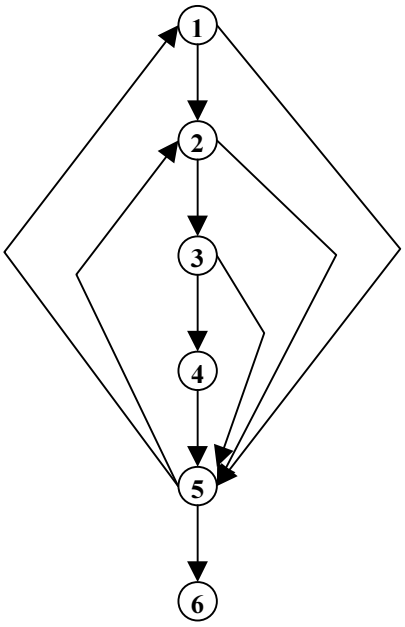
where v stands for McCabe's cyclomatic number, F is the flowgraph that is studied, e is the number of directed edges and n is the number of nodes in the flowgraph. We will try to illustrate this measure by an example.

Figure 8:1 Example for McCabe's cyclomatic number

```
        Procedure sort (var x: array of integer; n: integer);
        var i, j, save : integer;
        begin
(1.)        for i:=2 to n do                    (node 1)
(2.)        for j:=1 to i do                    (node 2)
(3.)            If x[i]<x[j] then               (node 3)
(4.)            Begin                           (node 4, including rows 4-7)
(5.)                save:=x[i];
(6.)                x[i]:=x[j];
(7.)                x[j]:=save;
(8.)            end                             (node 5)
(9.)    end;                                    (node 6)
```

The flowgraph for this module would be:

Figure 8:2 Flowgraph of example in figure 8:1



In this module McCabe's cyclomatic number would be 6, i.e. there are 10 directed edges and 6 nodes (v(F) = 10-6+2 = 6). On the basis of empirical research, McCabe claimed that modules with high values of v were those most likely to be error-prone and unmaintainable. He proposed a threshold value of 10 for each module; that is, any module with v greater than 10 should be redesigned to reduce v. However, as we already have mentioned, the cyclomatic number presents only a partial view of complexity. It can be shown mathematically that the cyclomatic number is equal to one more than the number of decisions in a program, and there are many programs that have a large number of decisions but are easy to understand, code and maintain. Thus, relying only on the cyclomatic number to measure actual program complexity can be misleading.

8.2.2 Strengths and weaknesses of McCabe's measure

Among the advantages of this measure one can notice that it is rather easy to compute from the program text, and flowgraph as well. Most of the software tools available for code measuring, support the use of McCabe's cyclomatic number, which is a sign of its user-friendliness. The measure also supports a top-down development process to control module complexity in the design phase, i.e. before actual coding takes place. As we mentioned in the previous section it can also be used to *control* the complexity of program modules, with regard to McCabe's recommendation of an upper bound of 10. Moreover it can be adapted to *evaluate* alternate program design to find the simplest possible program structure, and also as a guide for allocating testing resources (Fenton & Pfleeger, 1996).

The critics against the measure are mainly of a theoretical nature. Researchers have argued that it measures only the psychological complexity and not the computational complexity (Zuse, 1991), i.e. it may be hard for a human to understand a program with a high number of cyclomatic complexity, but it may not be as hard for a computer. Moreover, it views all predicates, either selection or iteration, as contributing the same amount of complexity. Other objections that have been raised against McCabe's measure are that it is insensitive among other things to the frequency and the types of input and output activity, to the size of purely sequential programs, to the number of variables in the program and to the intensiveness of fatal operations in the program, i.e. things that is captured by other measures (Ohlsson, 1996).

Despite all this criticism it is still one of the most practicized measures of software complexity, probably because it is simple to understand and implement. Empirical investigations do not give clear evidence of the usefulness of the measure. Some of them claim that McCabe's cyclomatic number correlates strongly with the error-proneness of a module, but others assert that simple measures of program size, such as Lines of Code, can be used as just as good estimates of error-proneness as the cyclomatic number (Zuse, 1991).

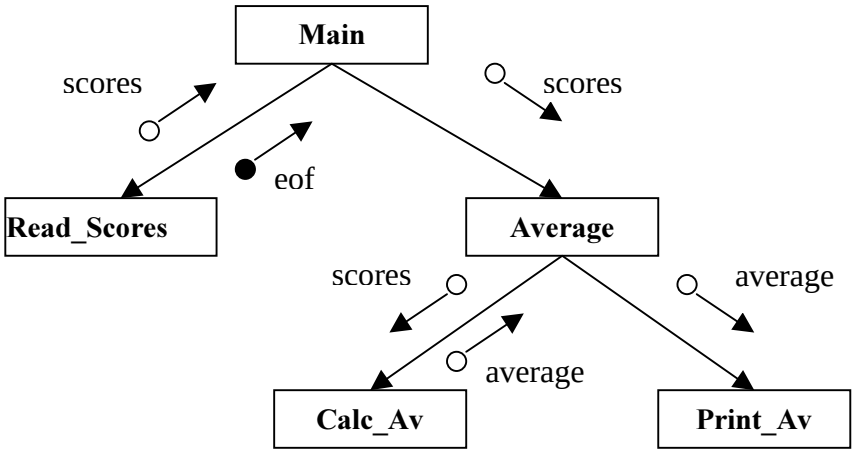
8.3 Data-flow structure

When the measures of the control-flow structure focus on the attributes of individual modules, the measures of data-flow structure emphasize the connections between the modules. The main object for the study is a collection of modules, either at the design stage or when the program is implemented in code. It is the inter-module dependencies that interest us, and measures of these attributes are called **inter-modular measures** (Fenton & Pfleeger, 1996).

However, first we have to establish a definition of a module, and in this context we rely on the one suggested by Yourdon & Constantine (1979): **"A module is a contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier"** (p. 37). This deliberately vague definition permits a liberal interpretation. Thus, a module can be an object that, at a given level of abstraction, you wish to view as a single construct. We insist additionally that a module is (at least theoretically) separately compilable.

To describe inter-modular attributes, we build models to capture the necessary information about the relationships between modules. Figure 8:3 contains an example of a diagrammatic notation capturing the necessary details about designs (or code). This type of model describes the information flow among modules; that is, it explains which variables are passed between modules. However, when measuring some attributes we do not need to know the fine details of a design, so our models suppress some of them. For example, when we just have to know if a module calls (or depends on) another module we use a more abstract model of the design, a directed graph known as the module call-graph.

Figure 8:3 Graph of module interaction; design charts.



(Fenton & Pfleeger, 1996, p. 303)

Information flow metrics can be applied to any functional decomposition of a software system. Examples of these include structure charts and dataflow diagrams. For example, in a dataflow diagram we do not have 'calls', instead we have data flows between processes. Given that information flow metrics apply to these forms of functional decomposition, they come into play from as early as the high-level design stage, and serve a useful purpose right the way down to low-level design when you can start to use McCabe metrics, or some other intra-modular measure (Fenton & Pfleeger, 1996).

What we are concerned with is consequently the information flow between modules. Researchers have attempted to quantify several aspects of information flow (Boehm, Brown & Kaspar, 1978; Hausen, 1989; Henry & Kafura, 1984), including:

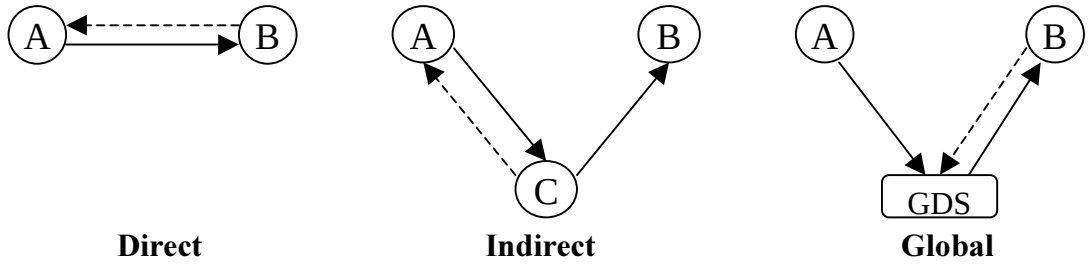
- The total level of information flow through a system, where the modules are viewed as the atomic components (an inter-modular attribute); and
- The total level of information flow between individual modules and the rest of the system (an intra-modular attribute).

The measure we are about to study is recognizing both of these aspects but is focusing on the latter. However, we start by explaining a few of the concepts used by Henry and Kafura (1981) in their information flow complexity metrics.

8.3.1 Henry and Kafura's information flow measure

To understand this measurement, we will have to consider the way in which data move through a system. The central concept is **local flow**, which can be **direct** or **indirect**. It is direct if: a) a module A invokes module B and passes information to it (active), or b) the invoked module B returns a result to the caller, module A (passive). Similarly, we say that a local indirect flow exists if a third module C invokes A and moves the result to B. A **global flow** exists if information flows from module A to module B through a global data structure, that A writes to and B reads from.

Figure 8:4 Direct, indirect and global information flow



Using these notions, we can describe two particular attributes of the information flow. The fan-in of a module M is the number of local flows that terminate at M, plus the number of data structures from which information is retrieved by M. Similarly, the fan-out of a module M is the number of local flows that emanate from M, plus the number of data structures that are updated by M.

Based on these concepts, Henry and Kafura (1981) measure information flow "complexity" as:

Information flow complexity(M) = length(M) x [fan-in(M) x fan-out(M)]^2

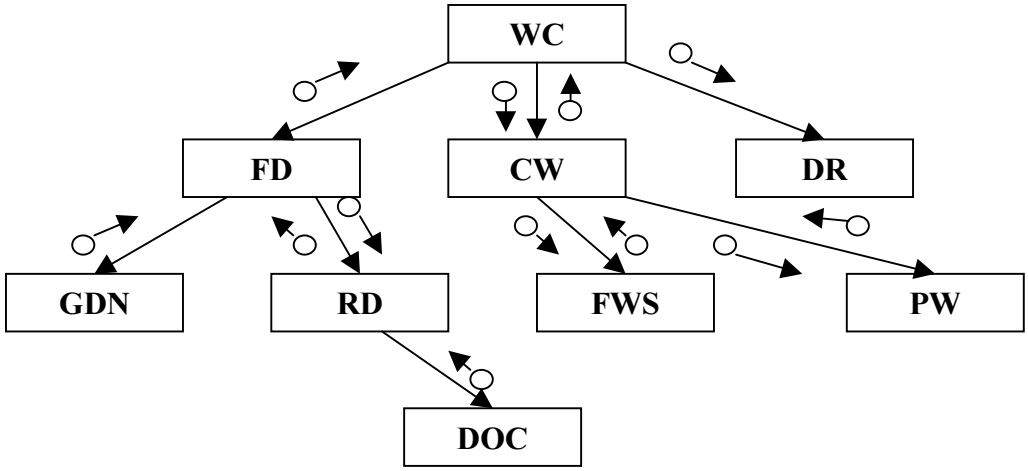
Figure 8:5 shows an example of how this measure is calculated from a design's modular structure.

The length factor in Henry and Kafura's equation can be measured in different ways, but usually the method of counting lines of source code is applied. The formula also includes a power component to model the non-linear nature of complexity. The assumption is that if something is more complex than something else, then it is much more complex rather than just a little bit more complex.

Some authors propose that as a guide, the 25 percent of modules with the highest scores for fan-in, fan-out and information flow complexity values should be investigated (Rapps & Weyuker, 1985). High information flow complexity values indicate highly coupled components. These modules need to be looked at in terms of fan-in and fan-out to see how to reduce the complexity level. Sometimes a 'traffic center' may be hit. This is a module where, for whatever reasons, there is a high information flow complexity value, but things cannot be improved. Switching components often exhibit this. Here there is a potential problem area, which, if it is

also a large component, may be very error-prone. If the complexity cannot be reduced, then one should make sure that the component is thoroughly tested.

Figure 8:5 Example of Henry and Kafura's information flow complexity measure.



Module	fan-in	fan-out	$[(\text{fan-in})(\text{fan-out})]^2$	length	'complexity'
WC	2	2	16	30	480
FD	2	2	16	11	176
CW	3	3	27	40	1080
DR	1	0	0	23	0
GDN	0	1	0	14	0
RD	2	1	4	28	112
FWS	1	1	1	46	46
PW	1	1	1	29	29
DOC	0	1	0	18	0

(Fenton & Pfleeger, 1996, p. 314)

8.3.2 Strengths and weaknesses of Henry and Kafura's measure

The measure of Henry and Kafura has received considerable attention. It was the subject of a major validation study using industrial software, in the sense that its prediction ability was investigated. Henry and Kafura established a correlation with their measure and maintenance change data for the UNIX operating system; those modules with excessively high values for information flow complexity were well-known by the maintenance staff as being the source of many serious problems (Henry & Kafura, 1981).

The merit of this measure is that it addresses a higher level of complexity than in the case of McCabe's metrics. To get an overall picture of complexity one need also to study the interactions between modules, not only the transactions within it. Modules with either a low fan-in or low fan-out are, in a sense, isolated from the system and hence have low complexity. This is recognized by Henry and Kafura's measure, and they justify the multiplication of fan-in and fan-out, by referring to this observation.

However from a measurement theory perspective, researchers have questioned the basis for this action. In particular, they are concerned about the lack of complexity for

many modules, as the multiplication leads to a value of zero complexity for any module in which either the fan-in or fan-out is zero. Many experts have also questioned the length factor, partly because it is a separate attribute and partly because it contradicts measurement theory. Consequently Martin Shepperd studied Henry and Kafura's measure in depth and identified a number of theoretical problems with it (Shepperd & Ince, 1990). He proposed a refinement to the measure, by excluding the length factor:

Shepperd complexity (M) = [fan-in(M) x fan-out(M)]^2

Moreover, he changed the definitions of some of the concepts used in the original model, like local and global flows, global data structures and which modules that should be counted. Shepperd's refinements attempt to capture a specific view of information flow structure, and are thus consistent with measurement theory. His empirical validation studies examined how closely the measure correlate with a specific process measure, namely development time. Interestingly, the relationship between development time and the Henry-Kafura measure was not significant for Shepperd's data, but Shepperd's pure-information flow-structure is significantly related. However, other studies say that both Henry and Kafura's and Shepperd's models are not very useful when trying to predict the error-proneness of a program (Ohlsson, 1996; Goodman, 1993). In these studies even Lines of Code correlate closely to attributes of error than these measures do.

8.4 Data structure

We have seen how information flow (or data flow) can be measured, so that we have some sense of how data interact with a system or module. However, once again we are turning to the intra-modular level of measurement, and focus on the attempts that have been made to define measures of actual data items and their structure. In this section, we examine one representative data-structure measure to see what it can tell us about system products.

The focus of data structure is of course data itself, i.e. the atomic components of the program (if we ignore the bit level). We want to capture the "amount of data" for a given system by trying to identify those constructs that make up the program and weigh them in a way that they accurately reflect the total complexity of the system. The idea is that when we use more constructs we are also creating a more complex program. This complexity increases even further if we use a large number of *different* constructs. Thus the overall complexity of a system cannot be depicted completely without measures of data structure; control-flow measures can fail to identify complexity when it is hidden in the data structure.

8.4.1 Halstead's measures of complexity

Maurice Halstead's measures of complexity (Halstead, 1977) were proposed shortly after McCabe presented his cyclomatic number in 1975. Halstead intended to make his measurement an alternative to the counting of Lines of Code as a measure of size and complexity, but it has, since the end of the 1970's, mainly been used as a predictive measure of the error-proneness of a program.

Halstead's software science attempted to capture attributes of a program that paralleled physical and psychological measurements in other disciplines. He began by defining a program P as a collection of tokens, classified as either operators or operands. The basic definitions for these tokens were:

n_1 = number of unique operators
n_2 = number of unique operands

N_1 = total occurrences of operators
N_2 = total occurrences of operands

The identification of operators and operands depends on the programming language used. We will illustrate the basic tokens with the code example we had in the section about McCabe's cyclomatic number (8.2.1, Figure 8:1). If we separate this procedure in operators and operands we get the following numbers:

Table 8:1 Example of counting operators and operands

Operator	**Number of occurrences**
Procedure	1
sort()	1
Var	2
:	3
Array	1
;	6
Integer	2
,	2
begin … end	2
for … do	2
if … then	1
:=	5
<	1
[]	6
n_1 = 14	**N_1 = 35**

Operand	**Number of occurrences**
X	7
n	2
i	6
j	5
save	3
2	1
1	1
n_2 = 7	**N_2 = 25**

The choice of counting a token as an operand or an operator is a matter of judgement, and the definition of the terms depends on the programming language used. The tools available today for implementing Halstead's measures are also adjusted to a specific language, like C, C++, Java, Pascal etc.

Based on these notions of operators and operands, Halstead defines a number of attributes of software. These are shown in Table 8:2.

Table 8:2 The measures of Halstead

Size of Vocabulary	$n = n_1 + n_2$
Program Length	$N = N_1 + N_2$
Program Volume	$V = N \text{ LOG2}(n)$
Programming Effort	$E = (n_1 N_2 N \text{ LOG2}(n)) / (2\ n_2)$
Programming Time (seconds)	$T = E/18$
Approximation for N	$N = n_1 \log(n_1) \times n_2 \log(n_2)$

(Zuse, 1991, pp. 142-143)

The measures "Size of Vocabulary" and "Program Length" is rather self-explanatory, but we will clarify the others somewhat. The volume of a program is akin to the number of mental comparisons needed to write a program of length N. It is supposed to correspond to the amount of computer storage necessary for a uniform binary encoding. Thus, Halstead has proposed reasonable measures of three internal program attributes that reflect different views of size.

The measure of effort is based on the relationships reported in the psychology literature (Ohlsson, 1996). The unit of measurement of E is elementary mental discriminations needed to understand the program, and Halstead referred to psychologists claiming that the human mind is capable of making a limited number of such elementary discriminations per second. These psychologists asserted that this number lies between 5 and 20. Halstead claimed that this number *is* 18, and hence the programming time T for a program of effort E is expressed as in the equation for Programming Time above.

8.4.2 Strengths and weaknesses of Halstead's measures

Of the measures presented in this chapter, Halstead's metrics is probably the most criticized. One of the problems associated with Halstead's model is how to define operands and operators. We have already touched upon this issue when we discussed the dependence on programming language. Halstead assert that all commandos in a program can be separated in operators and operands, but the problem is that even if we have succeeded in categorizing the components of one language, we have to do it all over again when we begin to use another language. Moreover the definition of operators and operands may differ between different users, since there is no standard, which will make comparisons between software developers difficult (Ohlsson, 1996).

Halstead's argumentation has also been accused of not giving a consensus of the meaning of attributes such as volume and effort. In other words, the relationship between the empirical relational system (that is, the real world) and the mathematical model is unclear. Further, Halstead gives no real indication of the relationships among different components of his theory. And we cannot tell if he is defining measures or prediction systems (Fenton & Pfleeger, 1996).

The psychological assumptions that Halstead makes in his measurement of effort and programming time have also been criticized. The opponents have claimed that the

postulation that the human mind is capable of 18 mental discriminations per second is too exact, and does not have enough empirical evidence to be accepted as scientific (Ohlsson, 1996).

Moreover, the model was originally developed for use in measuring small-scale systems developed during the 1970's, and experts have thus claimed that it is not as useful, when measuring the more extensive modern software systems. Davis and Leblanc (1988) made a study of a number of classical measures of software, among them Halstead and McCabe, and their results indicated that Halstead's size measures where the best predictors of error-proneness of a program. Their object of study was also smaller systems, which may explain why they reached this result.

Despite the shortcomings of the model, Halstead's measures are still widely used, as the other measures presented in this chapter. Above all, it is important to realize its importance of this work as the first major effort attempting to relate program characteristics with complexity. This work provided the impetus to others in the area of software metrics.

8.5 Conclusion of the structural complexity metrics

As we have seen from this survey, there is no such thing as the perfect measure of structural complexity, that encompasses all aspects of the concept. Rather, each measure has its benefits as well as shortcomings. Moreover, since the researchers do not agree on the predictive power of the measures with regard to error-proneness, it is not possible to give a distinct answer of which measure is the "best". We think that each company has to make its own study of the relationship between these measures, or a combination of them, and error-proneness, in order to find the measure/measures that suit their business.

Since each of these measures is addressing one aspect of structural complexity it may be advisable to combine two or more of these measures in order to get a more complete picture of the complexity, and perhaps also increase the predictive power of the measurements. Due to lack of time, we have not been able to carry out such an extensive study of the structural complexity measures. We will suggest this as an assignment for future master students. However, as we will see in Chapter 10, *Productivity, Quality and Performance*, we need to have a measure that can predict the quality of the software developed. This means that we have to suggest one of these measures to be used, together with other measures described in this report, at ERV.

We believe that McCabe's cyclomatic number can be preferred for three reasons. It is widely used among software companies, for predicting error-proneness and reliability. Moreover, it is almost always supported in the automatic measuring tools available at the market, and for many different languages. Finally, it can be used early in the software development process, in contrast to, for example, Halstead's measures.

8.6 Summary

In this chapter we have focused on measures for structural complexity, mainly for predicting error-proneness and the quality dimension of computer software. We divided structure in three different categories: control-flow structure, data-flow

structure and data structure. Control-flow structure is focusing on what we call the structural primitives of a program: sequence, selection and iteration. Program design or code can be depicted by flowgraphs, from which we can develop different measures of the executable nature of a program. McCabe's measure of cyclomatic complexity was investigated, and an example of how to count it was elaborated. The strengths identified with the measure are mainly its ease of use and understanding, together with the applicability in different stages of the software development process. The criticism against the model is mainly of a theoretical nature, meaning that it is not measuring the computational complexity, but rather the psychological complexity of the program. Additionally, it is not relating to other aspects of complexity that is captured by other measures.

Data-flow structure emphasizes the inter-modular aspect of complexity, i.e. the connections and relationships between the modules. Models and graphs of this structure can also be made, and from these pictures of the program we can develop measures of how the information flow through the system. The example of Henry and Kafura's measure shows how we can capture the complexity of this information flow, by defining the concepts of local and global flows and summarize them in the model of fan-in and fan-out of modules. The measure should be seen as a complement to the intra-modular models of complexity, since it focuses on a different level of program complexity. In subsequent studies evidence have been found that Henry and Kafura's measure is rather useful when trying to predict the error-proneness of a program. However, it has been criticized for not being mathematical correct according to measurement theory, and revisions of the formula have been made.

Then, we turned once again to the intra-modular level, and focused on measures of actual data items and their structure, to make the picture of structural complexity complete. An example of these measures is Halstead's original measures of size and complexity. He identifies two data components of programs, operators and operands, and divide each of these in two categories, which results in four parameters: number of unique operators, number of unique operands, total occurrences of operators and total occurrences of operands. Based on these concepts Halstead propose different measures of size, effort and time for estimating and evaluate different aspects of a program, such as error-proneness and development time. Many experts have recognized the problems with these measures. Above all the model is criticized for its lack of standard definitions of the concepts used and for the psychological assumptions made. Moreover, it has been dismissed as not applicable for larger systems. The studies that claim that the measure correlates strongly with error-proneness have also been made on small-scale systems, as the model originally was developed for.

Even if we recognized that there is no consensus of which measure correlates strongest with error-proneness, we suggested that McCabe's cyclomatic number should be used at ERV. This choice was based on its widespread use, its incorporation in most of the automatic measuring tools and its applicability early in the development process.

9 Algorithmic Complexity and Size Measures.

9.1 Software size and productivity

In Chapter 7, *Software Complexity*, we described software complexity with a model that was divided in two main tracks. One of the tracks bears upon error-proneness, which in the end leads to quality issues. The other track considers size and effort, which in its final step influence productivity. Our mission with this report is partly to look at productivity of software projects. As our model of software complexity shows the divided tracks of error-proneness and size/effort merges again, to some part, when it comes down to quality and productivity. In other words, the tracks can not be totally apart since they both have influences on each other. However, in Chapter 8, *Structural Measures of Software Complexity*, we had the discussion of quality issues and structural complexity. Now it is time to enter deeply in the field of algorithmic complexity and size metrics in software projects. As quality can be related to structural complexity, productivity is related to algorithmic complexity. In this chapter we discuss the issues of algorithmic complexity and how it influences software size. We also describe different functional metrics that measures software project. But first we need a brief introduction of productivity as an overall subject for later discussions.

When it comes to measuring software projects, productivity is one of the most interesting measurements for software management. We have already discussed this in Chapter 6, *Software Metrics*. Anyway, it is obvious that the ability of making accurate estimations of costs and time-efforts in the pre-face of projects, as well as getting a concluding confirmation of project efficiency, are desirable for management. So far we have learned that software development projects are influenced by various factors not always concrete or measurable. This makes it difficult to measure software projects: How efficient is the project? Are the estimated development costs accurate? Is the final application well-structured, maintainable, reliable etc.? One way to measure these kind of projects is by comparing similar projects to each other, and from this gaining a baseline of comparable measures. Due to the fact that productivity is influenced by size/effort, which in turn is influenced by various subjective factors such as the language used, tools, desired functionality etc., no complete productivity-formula has been developed. The most common way to measure productivity is still by using the equation:

$$productivity = \frac{size}{effort}$$

However, from the equation above one can see that software size is an essential parameter for measuring productivity. The traditional size counting metric is Lines of Code (LOC) which still is used by several software development corporations. This method has many disadvantages, as will be discussed later, and software developers have therefore tried to invent alternative methods. Size as a simple volume metric is maybe enough when producing physical products as clothes, furniture, cars etc. For software projects, though, a more accurate size metric should also consider the factors of complexity as we have discussed. Development of alternative metrics is still an ongoing process, since no method has gained total acceptance. One reason is because of the differences between software applications that make it hard to find general

methods that work on all kinds of projects. Another reason can be that parts of the software development community have not yet settled down with the fact that their work has to be measured.

The development of alternative methods to counting lines of code has lead to the origin of functional metrics. They are also size measures, but include some algorithmic complexity too. Instead of counting a static value like LOC they focus on user functionality and counts functional parameters as input, output, inquiries, internal and external logical data files, and in some cases also algorithms.

9.2 Algorithmic complexity

Algorithmic complexity reflects the complexity of the algorithm implemented to solve the problem. It also reflects the degree of algorithms within a specific application. Harder kinds of systems seem to require more algorithms than Management Information System (MIS) software and therefore have a higher algorithmic complexity.

In a broader sense algorithmic complexity deals with *spatial complexity*, i.e. the length and size of the software, and *algorithmic volumes*. Problems with high complexity seem to require longer algorithms and the basic concepts for this measure is therefore the length and structure of algorithms (Jones, 1996).

9.2.1 Algorithms

Before we get further into the field of algorithmic complexity it is appropriate to give the definition of an "algorithm". An algorithm is defined as the rules which must be completely expressed in order to solve a significant computational problem (Jones, 1996). To make it easier to understand what an algorithm really is we will give you some examples of typical algorithms:

- *Sorting*
  One of the earliest forms of algorithms created during the computer era.

- *Searching*
  Binary searches, tree searches radix searches, and many others.

- *Step-rate calculation functions*
  E.g. calculations of income tax rates, where a certain level of taxable income is related to a certain tax rate.

- *Feedback loops*
  Very common in process control applications, hospital patient monitoring applications, and many forms of embedded software such as that for fuel injection, radar and navigation.

When an algorithm is defined the next step is to appoint the degree of difficulty for that specific algorithm, in other words to weigh the algorithm. There is still no taxonomy of how to do this, except for some guiding lines made by Capers Jones at the SPR (Software Productivity Research). The basis so far for how to determine algorithm weights is twofold: 1) the number of calculation steps or rules required by

the algorithm and 2) the number of factors or data elements required by the algorithm (Jones, 1996).

9.3 Size measures

Earlier we described the standard equation of productivity as size divided by effort. In this section we will describe the main categories for software size metrics. The first one, namely Lines of Code (LOC), can be classified as a natural deliverable of a software project. Other natural deliverables are the number of pages of paper documents and test cases. These are all strictly volume deliverables and the tangible outputs of many tasks.

The other category of software size metrics is functional size metrics and they are classified as synthetic deliverables of a software project. They are the volumes of abstract constructs and include Function Points (in all variations) and Feature Points. All these techniques will be further described. According to Capers Jones (1996) the synthetic functional deliverables are superior to the natural deliverables for economic studies, but it is useful to record the natural deliverables anyway.

In this report we categorize the metrics that we are about to describe as *plain size metrics* (Lines of Code) or *functional size metrics* (variants of Function Points and Feature Points). We have focused on four fundamental metrics, IFPUG's Function Points, SPR's Function Points, SPR's Feature Points, and a new technique called Full Function Point, which is under development at the University of Québec. We have also found some other techniques but they are only mentioned briefly. The original functional size metrics is Albrecht's Function Points, which has developed into IFPUG's Function Points, or also referenced to as Function Point Analysis (FPA). An overall introduction to Function Points will therefore be given before we describe the different techniques one by one. But first of all, let us have a brief look at some elements that can influence the results of size measurements before we enter more deeply into the size metrics.

Measurements can be effected by subjective factors due to human actions. People have a tendency to not like being measured, even if they understand that this is necessary. This is why every organization that measures productivity, or suchlike, must clarify the reason for measuring. Negative pressure on programmers or project leaders from doubtful measurements can make them add unnecessary code just to allocating resources or rewards since the productivity increases with the present productivity equation. This is actually one of the criticisms of the metric LOC.

9.3.1 Lines of Code

Counting lines of code is the most traditional method for measuring software size. It is a quick method that can be performed with automated tools. This is one reason why counting lines of code has been so widespread among software development companies, despite of the limitations of the measure. We have recently talked about algorithmic complexity and that it is the track of our complexity model that handles size and effort. Counting lines of code is however a pure quantitative measure and it does not consider algorithmic complexity at all. This method is so commonly known though, and is for this reason needed as a historical background for later discussion of alternative size measures.

Since the actual counting is made on code, the possibilities for estimating software size early in the project are often small. If management lack information about the size of the program, they have difficulties to estimate development costs. A desirable feature should be to know how much one line of code costs, and how many lines of code the specific program requires. In the economic sense lines of code is neither goods nor services. A customer does not know how many lines of code that exist in a software product and can therefore not buy lines of codes directly. Often they neither know, nor care, how much code was written or in what language, as long as they know the functionality and costs of the program.

Another restriction for using Lines of Code is the language dependency. To be able to compare software products with measures of lines of code they must be coded in the same language. When developing software in a homogenous environment that is not a problem. But today almost every software project uses a mixture of languages. Comparison between a mixed language product and another product coded strictly in COBOL for example is not accurate when counting lines of code. We also face the problem of size variations that are due to individual programming style. A minor controlled study carried out by IBM illustrated that code size varied due to the styles of the programmers and their interpretations of what the specifications asked for (Jones, 1996). Since there is no standard of how to measure lines of code the ability to compare industry data is limited. Every company may have their own internal rules of how to measure their products and projects. Questions could arise about if blank lines count or comments. Should data declaration be included, and what happens if a statement extends over several lines? Some guidelines have been published by organizations like the Software Engineering Institute, but they still do not represent any totally accepted standard.

Other serious deficiencies associated with Lines of Code (*Software Metrics – why bother?,* 1998):

1. The lack of a national or international standard for a line of code metric that encompasses all procedural languages.
2. Software can be produced by methods where entities such as lines of code are totally irrelevant. Example: program generators, spreadsheets, graphic icons, reusable modules of unknown size, and inheritance.
3. Lines of Code metrics paradoxically move backward as the level of the language gets higher, so that the most powerful and advanced languages appear to be less productive than the more primitive low-level languages. This is due to the equation of productivity (*Productivity = Size / Effort*) which generates a lower productivity if the size of the code decreases.

The ability to estimate size of software projects is of increasing interest for most companies. Lines of Code fails with this approach as it can only be counted after the application is coded. A final reason for not using LOC in studies of software production costs is that it does not include the other deliverables of the product. As we said in the introduction to counting lines of code, the metric do not consider algorithmic complexity.

9.3.2 Function Points in general

Function Point counting is one of the fastest growing software management techniques in the software industry today (Garmus & Herron, 1996). To some, the Function Point methodology is the cornerstone of their software development and management. To others, it is simply one of many software management tools that are used to successfully build systems.

Software metrics talks about quantitative and qualitative aspects to software measurement. Function Points are classified as one of the key quantitative measures. In contrast with counting lines of code which has the producers point of view of the project, the Function Point method is more focused on the users point of view. The intention of the Function Point technique is to give equivalent results regardless of the application or the technologies used.

Function Points do not reveal as much by itself as with the combination of other metrics. It is just one of a number of required measures for software development projects. This will be brought up only briefly in this section but more thoroughly discussed in Chapter 10, *Productivity, Quality and Performance*. The main focus in this section will be on describing development of Function Points and different function point techniques.

9.3.2.1 History of Function Points

It was in the mid-1970s that the Function Point methodology developed by Allan Albrecht at IBM. They needed to establish a more effective and predictable measure of system size, to better predict or estimate delivery of software. The ambition of Albrecht was therefore to create a function point metric that could meet five main goals. The final method he developed had the following characteristics (Jones, 1996):

1. It dealt with the external features of software.
2. It dealt with features that were important to users.
3. It could be applied early in a product's life cycle.
4. It could be linked to economic productivity.
5. It was independent of source code or language.

These goals were the fundamentals of Albrecht's Original 1979 Function Point Methodology and was discussed publicly for the first time at a joint Share/Guide/IBM conference. Continued statistical analysis was made from 1979 to 1984. The refinements that were made resulted in the Function Point method we use today, named IFPUG Function Points or Function Point Analysis (FPA).

In the beginning, Function Point activities were mostly limited to those selected organizations that had the insight to understand the usefulness of the Function Point method. High profile companies such as AT&T, Motorola, Hewlett-Packard, and Boeing were among the early users of the methodology. With their reputations as quality producers of software, Function Points began to gain exposure to a wider audience (Garmus & Herron, 1996).

At the beginning of the year 1986, several hundred companies had been using Function Points. A critical mass or function point users occurred and it was decided to form a nonprofit organization to handle questions and development of the method.

The organization was named the International Function Point User Group, but is often referenced just as the IFPUG (Garmus & Herron, 1996).

Function Point Analysis, authored by Brian Dreger, is considered to be the first practical layman's guide and instruction to Function Point analysis (Garmus & Herron, 1996). It was published in the fall of 1989 and was generally accepted as a standard for counting Function Points. Since the need to update Dreger's original work became obvious and the IFPUG organization had grown in size and prominence, it naturally took the role as being the keeper of the Function Point guidelines. Today there is a well-documented and maintained set of counting guidelines that are available to all IFPUG members. Since the inception of the Function Point methodology, the enhancements that are made have commonly been focused on improving the clarity of definitions, rules, and guidelines. The general counting rules, that were initially established, have hence never really changed.

9.3.2.2 Function Points measurement opportunities

Function Points can be used as a stand-alone metric to monitor the application domain at a high level, or in combination with other measures to create a variety of valuable software process performance and quality measures. Some of these measures are considered to be normalized and therefore allowed for comparison among industry segments. In the software measurement practice there are a commonly accepted set of core metrics used in combination with Function Points: level of effort, costs, defects, and duration. Together they result in commonly accepted industry measures as shown in Table 9:1.

Table 9:1 Common Industry Software Measure

Type	Metrics	Measure
Productivity	Function Points / Effort	Function Points per person month
Responsiveness	Function Points / Duration	Function Points per calendar month
Quality	Defect / Function Points	Defects per Function Points
Business	Costs / Function Points	Cost per Function Point

(Garmus & Herron, 1996, p. 26)

Increased cost consciousness among organization has affected their need to gain control over their application domain. By establishing a *portfolio baseline* of functional metrics they can get a better understanding of their software assets. A portfolio baseline can be describes as a history record. Results from previous projects are collected in the record, and with a baseline big enough conclusions from these projects can help to predict or estimate the results of new projects under development. Portfolio counts thus provide an organization with size data for comparing applications and potentially making comparisons to purchased software packages (Garmus & Herron, 1996).

9.3.2.3 Different counting groups

To actually perform a count or even get insight into the nature of Function Points and function point counting requires training. The great benefit is that it only takes a one-day class and a little on-the-job training by an experienced counter to learn the technique. There are no limitations of who is best suited to count the Function Points either. According to Bohem (1997), every category has their benefits and disadvantages:

- ***Technical people*** have traditionally been the main function point counters. They have great experience in the coding area and are therefore good at estimating Function Points. Unfortunately they also know how much effort a certain function may have demanded and do not accept a low value of the function when measured in Function Points. As long as they apply the counting rules correctly they are well skilled people to count Function Points.

- ***Senior level people*** are also suitable for counting Function Points. After counting, a senior technical person will have a good idea of whether the project is proceeded as planned. In addition to the number of function points, he will also know if the right functions are among them.

- Even ***users*** (clients or customers) can in some cases be suited for counting Function Points. Sometimes outsourcing agreements can place system development in the hands of people that have no knowledge of the well being of the firm. If they can be a part of the counting process they also can control that they are getting the required functionality in their system.

9.3.2.4 Number of people counting in an organization

Three possible constellations of counting groups can be established: everyone, a small group, or a single person. All project personnel should be familiar with Function Points but most of them will not be able to count them accurately. It is important to understand how they are being applied, but for getting an accurate result of the Function Points the personnel counting must count on a regular basis. If they only count every six moth, or so, they have forgotten the rules between these occasions. To have constant retraining for everybody in a large organization is not very cost effective. The ideal constellation in a large organization is therefore a small group involved with function point counting and other estimation activities (Bohem, 1997). If they also are project independent, less bias feedback will be provided regarding the projects. As a group, they will also be able to look at each other for assistance with difficult counting situations.

With light workload, allocating a single person to be the counting guru has the same advantages as allocating a small counting group in larger organizations. A relationship with an industry consultant is probably a must to be able to bounce off different interpretations of counting rules (Bohem, 1997).

9.3.3 IFPUG's Function Point method

The IFPUG's Function Point methodology, or simply Function Point Analysis (FPA) as we will also refer it to, is equivalent to the 1984 revision of Allan Albrecht's original method. The counting process involves the identification of five parameters

categorized into two function type groups: *data function types* and *transactional function types*. The two data function types are called Internal Logical File (ILF) and External Interface File (EIF). The transactional function types are External Input (EI), External Output (EO) and External Inquiry (EQ). All three types are identified as elementary processes that perform updates, retrievals, outputs, etc. Both data function types and transactional function types are defined as user identifiable groups of logically related data or control information. A full definition of these terms is given in Appendix B.

Before we describe the method more thoroughly, let us get an overview of the seven steps of the total process used to size Function Points.

1. *Determine the type of Function Point count*
2. *Identify the application boundary*
3. *Identify all data functions and their complexity*
4. *Identify all transactional functions and their complexity*
5. *Determine the Unadjusted Function Point count*
6. *Determine the Value Adjustment Factor – fourteen General System Characteristics*
7. *Calculate the final Adjusted Function Point Count*

The description of each of these steps that will follow has the purpose to give a notion of how Function Points are calculated. It will only be a brief explanation of the technique, though. Complete guidelines of how to calculate Function Points are distributed by the IFPUG with their Function Point Manual.

Determine the type of Function Point count

- Development Project Function Point counts
- Enhancement Project Function Point counts
- Application Function Point counts

Development Project Function Point counts and Application Function Point counts are basically the same. A development project of 1,000 function points will result in an application that has an application count of 1,000 function points. The Development Project Function Point Count measures the functionality provided to end-users with the first installation of the application. The Application Function Point count on the other hand, measure an installed application, and provides counts of the current functionality whether the application has been changed or not. The Enhancement Project Function Point Count measures modifications to an existing application. This includes the combined functionality provided to users by adding new functions, deleting old functions, and/or changing existing functions. After modifications are made, the Application Function Point Count must be revised to reflect the appropriate changes in the application's functionality. Even the Development Project Function Point Count must be updated as the development process proceeds. During development of new applications there can be modifications, such as added functionality that otherwise would not be captured in the count.

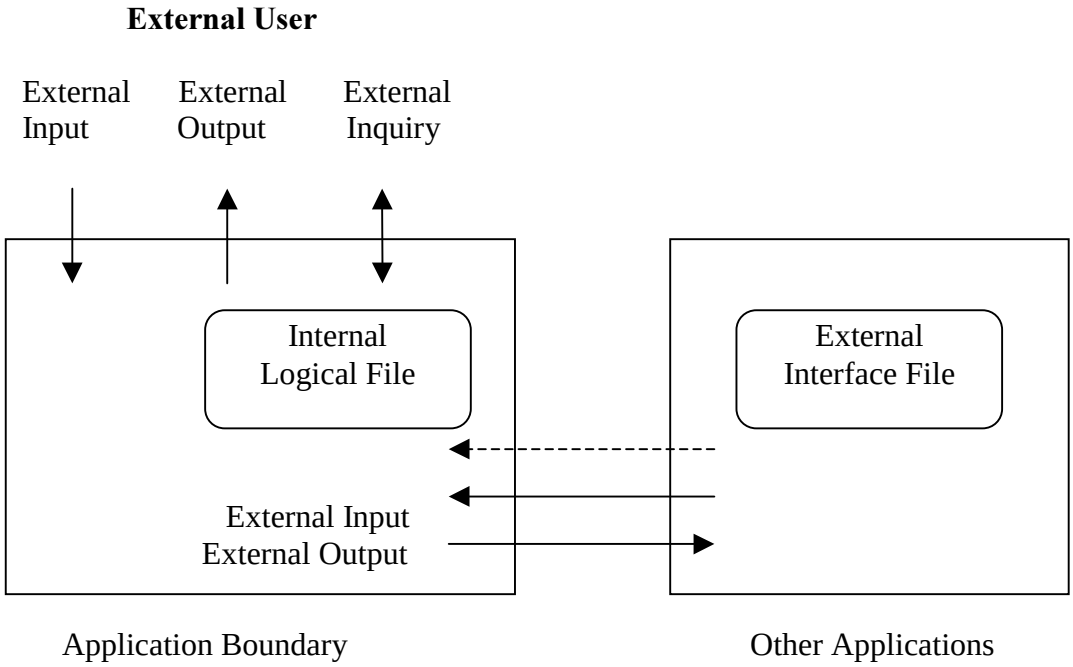
Identify the application boundary

Boundaries indicate the border between the project or application being measured and external applications or the user domain. Figure 9:1 illustrates the application boundaries. It also shows the connection to the data and transactional functions that are identified in the following two steps of the counting process.

Identify all data functions and their complexity

The data functions, internal logical files (ILFs) and external interface files (EIFs), relate to the logical data stored and available for update and retrieval. When they are identified each ILF and EIF is assigned a functional complexity based on the number of data element types and record element types. Using a complexity matrix the complexity level for each ILF and EIF is set from low, average to high as show in Table 9:2. The concepts of Record Element Types (RETs) and Data Entity Types (DETs), in the table, are defined in Appendix B.

Figure 9:1 Function Point Counting Components



(Garmus & Herron, 1996, p. 39)

Table 9:2 Complexity Matrix for Internal Logical or External Interface files

Record Element Types	**Data Entity Types**		
	1-19	**20-50**	**51+**
< 2	L	L	A
2 - 5	L	A	H
> 5	A	H	H

L=Low, A=Average, H=High

(Garmus & Herron, 1996, p. 51)

Identify all transactional functions and their complexity

The transactional functions, external inputs (EIs), external outputs (EOs), and external inquiries (EQs), perform the processes of updates, retrieval, outputs etc. They are each identified and counted and weighted in a complexity matrix similar to that of data functions. The difference is that Record Element Types are exchanged for the number of File Types Referenced (FTRs) instead (see Appendix B). The grades of the data fields also differ for each of the transactional functions.

Determine the Unadjusted Function Point Count

Now the number of data and transactional functions are counted and their contributed level of complexity is assigned to each of them. Each component is then weighted in the Unadjusted Function Point table. The summarized values results in Unadjusted Function Points. In table 9:3 the Unadjusted Function Point (UFP) table is combined with an example of how to count the UFP. The bold figures in the Function Level columns correspond to the number of components found. With this procedure the algorithmic complexity of the software is evaluated.

Table 9:3 IFPUG Unadjusted Function Points

Components	Function Levels			
	Low	Average	High	**Sum**
Internal Logical File (ILF)	**3**X7	**2**X10	**0**X15	41
External Interface File (EIF)	**2**X5	**1**X7	**1**X10	27
External Input (EI)	**5**X3	**2**X4	**0**X6	23
External Output (EO)	**3**X4	**1**X5	**0**X7	17
External Inquiry (EQ)	**2**X3	**1**X4	**0**X6	10
Unadjusted Function Points				**118**

Data Function Types
Transactional Function Types

(Garmus & Herron, 1996, p. 53 (modified))

Determine the Value Adjustment Factor

User/owner functions are also characteristics that affect the performance of an application. The General System Characteristics lines fourteen different factors to be considered to identify the Value Adjustment Factor (VAF). This is a multiplier that is used on the Unadjusted Function Point Count in order to calculate the Final Adjusted Function Point Count. Each General System Characteristic (GSC) must be evaluated independently of its Degree of Influence (DI), which is a scale of zero to five:

0. Not present, or no influence
1. Incidental influence
2. Moderate influence
3. Average influence
4. Significant influence
5. Strong influence throughout

The scores of the assigned values are summed to calculate a Total Degree of Influence (TDI). A real-time, telecommunication, or process control system should expect a total score between 30 and 60. Then, the TDI will be used in a separate calculation to determine the Value Adjustment Factor (VAF).
The equation to produce the Value Adjustment Factor:

$$VAF = (TDI * 0.01) + 0.65$$

The General System Characteristics examined:

1) Data Communications
2) Distributed Data Processing
3) Performance
4) Heavily Used Configuration
5) Transaction Rate
6) On-Line Data Entry
7) End-User Efficiency
8) On-Line Update
9) Complex Processing
10) Reusability
11) Installation Ease
12) Operational Ease
13) Multiple Sites
14) Facilitate Change

Further reading of what each GSC stands for can be found in *Measuring the Software Process,* by David Garmus and David Herron (1996).

Calculate the final Adjusted Function Point Count

This is the last step of the Function Point Count. Depending on which type of Function Point Count that has been performed, there are different equations to calculate the final Adjusted Function Point Count. The three types are Development, Enhancement or Application Project Function Point Count.

Now when we have described the method it can be interesting to know how long it takes to count function points with this method? Different suggestions are made of course. The difference is from 100 to 4000 function points per day. Including

preparation and possible presentation of complete project view, the 100 function points per day counting rate seems reasonable.

9.3.3.1 Criticism of Function Points

It is not to be forgotten that Function Points were developed in, and for, an MIS world and do not consider all the elements of complexity that are inherent in other types of software. A lot of critics to the methodology are thereby tended to originate from people in the scientific, telecommunications, and real-time/embedded software communities. However, this technique is still superior to the measures of counting lines of code. New methodologies as Feature Points, Mark II Method, and 3D Function Points have arisen to compensate the lack of complexity issues in the Function Point method. As some of them are based on the Function Point method, and in most cases are at the experimental stage, it can be a good approach to start explore the Function Point method and then continue with a complimentary method.

9.3.4 SPR's Function Point and Feature Point method

Software Productivity Research (SPR) introduced a new way to calculate function points in October 1985. The SPR function point variation simplified the way complexity was dealt with and reduced the human effort associated with counting function points. As described earlier the IFPUG technique assesses complexity by weighting 14 influential factors and evaluating the numbers of field and file references. This has been a target of criticism since it is a subjective way of counting and needs human effort. The aim of the SPR function point methodology is to get around these factors but still result in a reliable count. In fact, the method yields function point totals that are essentially the same as the current IFPUG function point method, with an average within 1.5 percent of the IFPUG method. The methodology has three additional goals to the original five goals Albrecht intended to meet (se 9.3.2.1). The additional SPR goals, according to Jones (1996) are:

6. To create function point totals easily and rapidly and to be able to create function points prior to the availability of all of the IFPUG factors in a normal project life cycle.
7. To predict source code size for any known language
8. To retrofit function points to existing software

The primary difference between the IFPUG and SPR function point methodology is, as said before, the way they deal with complexity. Compared to the IFPUG method, the SPR method does not use complexity matrixes based on RETs, DETs and FTRs. Instead it tries to separate the overall topic "complexity" into three distinct questions:

1. How complex are the problems or algorithms facing the team?
2. How complex are the code structure and control flow of the application?
3. How complex is the data structure of the application?

Each question has a numeric value range from one to five and is based on the level of complexity. The questions are described below (Jones, 1996):

Problem complexity?

1. Simple algorithms and simple calculation
2. Majority of simple algorithms and calculation
3. Algorithms and calculations of average complexity
4. Some difficult or complex calculations
5. Many difficult algorithms and complex calculations

Code complexity?
1. Nonprocedural (generated, spreadsheet, query, etc.)
2. Well structured with reusable modules
3. Well structured (small modules and simple paths)
4. Fair structure but with some complex modules and paths
5. Poor structure with large modules and complex paths

Data complexity?
1. Simple data with few variables and low complexity
2. Numerous variables but simple data relationships
3. Multiple files, fields, and data interaction
4. Complex file structures and data interactions
5. Very complex file structures and data interactions

The complexity sum of the problem and data complexity matches an adjustment multiplier as showed in Table 9:3. The code complexity question is actually not used by the SPR method for normal function point calculation. Code complexity is used when retrofitting Function Points to existing software as we will be describe later. Anyway, the adjustment multiplier that we have gained from the complexity sum is used in the final step of the SPR Function Point Method to calculate the Adjusted function point total.

Counting example:
Let's say that we have collected the number of data and transactional functions for a simple application, and the complexity sum of problem and data complexity is found to be 2 (1+1). From Table 9:3 we than can get the adjustment multiplier which is 0.6. The unadjusted total function point count that is summed up to be 24 is then multiplied with 0.6 to yield the Adjusted function point total of 14.4. The calculations are showed further in Table 9:4.

Table 9:3 The SPR Complexity Adjustment Factors

Sum of logical (problem) and data complexity	Adjustment multiplier
1	0.5
2	0.6
3	0.7
4	0.8
5	0.9
6	1.0
7	1.1
8	1.2
9	1.3
10	1.4
11	1.5

(Jones, 1996, p. 76)

Table 9:4 The 1985 SPR Function Point Method

Significant parameter	Raw data	Empirical weight	Total
Number of inputs?	1	X 4 =	4
Number of outputs?	2	X 5 =	10
Number of inquiries?	0	X 4 =	0
Number of data files?	1	X 10 =	10
Number of interfaces?	0	X 7 =	0
Unadjusted total			24
Complexity adjustment			0.6
Adjusted function point total			14.4
Integer value of adjusted total			14.0

(Jones, 1996, p. 76 (modified))

9.3.4.1 Establishing a baseline with the SPR technique

Earlier we talked about the benefits of establishing a business portfolio baseline, preferably in function points. Collecting such project data in function points can take quite a long time, especially if it must be done by hand. Previous projects may have been developed in other languages than those used at present. This complicates the reason for establishing a baseline, namely to compare function points between different projects. By using a programming language table (see Appendix D) these limitations can actually be evaded. To understand this technique a short description of what a language level is may be preferred.

9.3.4.2 Language level

The level of a language was at first defined as the number of basic assembly language statements that it would take to produce the functionality of one statement in the target language. Thus COBOL is defined as a level 3 language as it takes about three assembly language statements to create the functionality of one COBOL statement. After the publication of the function point metric in the late 1970s the definition enhanced to approach function point counts. The current definition it thus: “The number of source code statements required to encode one function point” (Jones, 1996, p. 78). SPR merged the new definition of “level” with the old and created a list of 300 common languages that showed both the traditional “assemble-level” and the average number of source code statements per function point. As language levels go up, fewer statements to code one Function Point are required. For example, COBOL is as we described earlier a level 3 language and requires about 105 statements per Function Point. The fact that various languages have a numeric value makes it possible to convert size from one language to another. Even to estimate function point size for applications written in mixed languages is possible by using the language level table.

9.3.4.3 Retrofitting existing software to Function Points

The table of language levels can be applied to terminated software projects to make a rough estimation of the degree of function points in each project. The language level table makes the count independent of the language environment used. An application coded in COBOL consisting of 91,000 lines of code can be estimated with the language table to be equivalent to a system of 1,000 function points. With this

technique, called backfiring, it should be possible to guess how many function points are implemented based on the number of lines of code. This makes it possible to establish a business portfolio baseline in function points (Bohem, 1997).

However, even though languages of the same level empirically require the same number of source code statements to implement one function point, the complexity of the application and its code have a strong impact. The relationship between function points and source code size often fluctuates widely partially due to individual programming styles and variations in the dialects of many languages. Therefore the programming language table displays the average source statements per function point as well as the maximum and minimum ranges.

9.3.4.4 Counting with SPR's 1985 Backfire Method

The SPR function point method supports retrofitting of Function Points to existing software and also takes the issue of different complexity into consideration. That makes the SPR algorithms bi-directional. If function points, code complexity, and the source language are the inputs, then the algorithms will predict source code size. If the inputs are source code size, code complexity, and source language, the algorithms will predict function points (Jones, 1996).

When backfiring is performed to an old application and the total function points have been calculated, it is time to take complexity into consideration too. Highly complex code tends to require more source statements per function point than extremely simple code. The complexity sum (code complexity, problem complexity and data complexity) of the SPR function point method that were described earlier now becomes useful again. This time the complexity sum relates to a code size adjustment multiplier as shown in the table below. The initial function point total is divided by the code size adjustment factor to calculate the final function point total.

Table 9:5 Adjustment Factors for Source Code Size Prediction

Sum of problem, code an data complexity	Code size adjustment multiplier
3	0.70
4	0.75
5	0.80
6	0.85
7	0.90
8	0.95
9	1.00
10	1.05
11	1.10
12	1.15
13	1.20
14	1.25
15	1.30

(Jones, 1996, p. 94)

This technique can be used for normal forward calculations as well with good results. It should be noted, though, that overall impact of complexity on source code size is not yet an exact science.

9.3.4.5 Efforts with the SPR Function Point count

With this technique it is not necessary to count the number of data element types, file types, or record types as it is with the current IFPUG method. Neither it is necessary to assign a low, average, or high value to each specific input, output, inquiry, data file, or interface or to value the 14 influential as defined by the IFPUG method. Since the SPR method deals with a reduced number of complexity considerations, the method can be applied somewhat earlier than the IFPUG method. The benefits of this method are above all that there are less parameters to count and simple calculations. According to Jones (1996), users who are generally familiar with function point principles and the application, can easily generate function point totals in less than a minute.

9.3.4.6 Development of Feature Points

Allan Albrecht's aim, when he developed the Function Point metric, was to be able to measure and compare different software projects independent of language and computer environment. In other words, he wanted to create a general-purpose metric for all kinds of software. The method was, however, first applied to Management Information Systems (MIS) and this lead to the misconception that the metric was only suitable for such systems. Even though this was not true, from the beginning anyway, the main focus has been to solve the measurement of classical MIS. This has lead to a lot of criticism from the scientific community. They state that the Function Point method may not bee optimal for real-time software (Jones, 1996). Capers Jones, who developed the SPR Function Point Count, also developed an experimental method for applying Function Point methods to software systems including operating systems and telecom systems. The method is known as Feature Points.

9.3.4.7 Feature Points vs. Function Points

The most visible difference between Function Points and Feature Points is that Feature Points make use of an additional component, algorithms, adding the set of the five Function Point components: inputs, outputs, inquiries, external interface files, and internal logical files. Harder systems seem to have a higher algorithmic complexity than MIS software, but less inputs and outputs. Therefore it can be a bit misleading to count these kinds of systems with Function Points. The Feature Point method is a good alternative for this matter as it compensates these problems by taking algorithms into consideration. Each algorithm is also assigned weight value, as with the rest of the components. In addition, the values assigned for logical data files are reduced, as they are less significant to harder kinds of systems.

Table 9:6 Ratios of Feature Points to Function Points for Selected Application Types

Application	Function points	Feature points
Batch MIS project	1	0.80
On-line MIS projects	1	1.00
On-line database project	1	1.00
Switching systems projects	1	1.20
Embedded real-time projects	1	1.35
Factory automation projects	1	1.50
Diagnostic and prediction projects	1	1.75

(Jones, 1996, p. 106)

In applications where the number of algorithms and logical data files is the same, Function Points and Feature Points generates practically the same numerical total. But when there are many more algorithms than files, Feature Points get a higher value than Function Points. Conversely, if there are several files and only a few algorithms, which is common with most information systems, the Feature Point total will be less than the Function Point total. The conclusion is to use the Feature Point Method when the algorithmic complexity is more prominent than the data files. Feature Points are still considered as an experimental method even though it has been successfully applied in several organizations, including Perker Elmer, Instrumentation Labs, and Motorola (Garmus & Herron, 1996).

9.3.4.8 Counting and weighting algorithms

The definition of an algorithm in standard software engineering texts is, as we have mentioned, "the set of rules which must be described and encoded to solve a computational problem" (Jones, 1996, p. 98). For feature point counting purposes, an algorithm can be defined in the following terms: "An algorithm is a bounded computational problem which is included within a specific computer problem" (Jones, 1996, p. 99). As a helping guideline for determining an algorithm and its complexity, Capers Jones at the SPR, developed a list of supplemental rules that can be followed. These rules have not gone past generalized definitions so the determination is still very subjective. However, they are being tuned and evaluated and research is underway to develop a more rigorous taxonomy and weighting scale for algorithms. When assigning the value of complexity, the SPR treatment of algorithms assumes a range of 10 to 1 for algorithmic complexity. In reality the range could be 1000 to 1, but the range must be limited or else it will be too difficult for humans to categorize the grade of complexity for each algorithm.

9.3.4.9 Shortcomings of Feature Points

Although algorithms are described and discussed in more than 50 books, there is no available taxonomy for classifying algorithms. Since there are no standard of how to determine and weighing an algorithm the method will continue to be treated as experimental. The method will not gain full acceptance until a consistent definition is achieved.

9.3.5 Full Function Points

During our research of finding methods of counting functional size we have continuously run in to new methods. Our latest finding is the Full Function Point Method, which is an extension of Function Points for real-time software. It is based on work done by the Software Engineering Management Research Laboratory at the Université du Québec à Montréal and Software Engineering Laboratory in Applied Metrics (SELAM) and was presented in 1997.

Full Function Points (FFP) is based on the observation that real-time software has several characteristics not taken into account in Function Point Analysis (FPA). Real-time software often contains more control data and sub-processes that MIS software. Since FFP is an extension of the standard FPA technique, all IFPUG rules are included in this new measurement technique, the small number of subsets of IFPUG rules dealing with control concepts having been expanded considerably. FFP solves this by introduces additional data and transactional function types (St-Pierre, Maya, Abran and Desharnais, 1997b).

9.3.5.1 Characteristics of real-time software

Both FPA and FFP separates the characteristics of an applications into transactional and data characteristics. The real-time software characteristics can be described as follows (St-Pierre et al., 1997a, 1997b):

Transactional characteristics: The number of sub-processes of a real-time process varies substantially. By contrast, the processes in the MIS domain display a more stable number of sub-processes. To measure the characteristics of a variable number of sub-processes adequately, it is necessary to consider not only processes as defined in FPA (elementary processes), but sub-processes as well.

Data characteristics: There are two kinds of control data structure: multiple occurrence groups of data and single occurrence groups of data. Multiple occurrence groups of data can have more than one instance of the same type of record. Single occurrence groups of data have one and only one instance of the record. There are usually a large number of single-occurrence control variables in a real-time software product.

The reader should also recall the discussion about real-time systems in Chapter 5, *The prerequisites at Ericsson Mobile Data Design*, for a more complete description.

9.3.5.2 FFP Function Types

The new function types introduced in FFP are only used to measure control data and control processes. Management data and other processes are still counted using the standard FPA technique. All FFP function types are listed in Table 9:7

Table 9:7 Full Function Point Functional Types

FFP Management Function Types:	
Internal Logical File (ILF)	exists in FPA, unchanged in FFP
External Interface File (EIF)	exists in FPA, unchanged in FFP
External Input (EI)	exists in FPA, unchanged in FFP
External Output (EO)	exists in FPA, unchanged in FFP
External Inquiry (EQ)	exists in FPA, unchanged in FFP
FFP Control Function Types:	
Updated Control Group (UCG)	new function type, similar to ILF
Read-only Control Group (RCG)	new function type, similar to EIF
External Control Entry (ECE)	new function type, similar to a subset of EI
External Control Exit (ECX)	new function type, similar to a subset of EO/EQ
Internal Control Read (ICR)	new function type, similar to a subset of EI/EO/EQ
Internal Control Write (ICW)	new function type, similar to a subset of EI

(St-Pierre et al., 1997a, p. 12)

The unadjusted count of an application using the proposed extension (FFP) can be expressed as follows (St-Pierre et al., 1997a):

FFP = Management FP + Control FP = (FPA – Control information) + Control FP

9.3.5.3 Point assignment

When the function types are identified each type will be assigned different points. FFP differ between point assignment for Management Function Types and Control Function Types. For the Management Function Types the procedure is the same as with FPA. For Control Function Types the point assignment is quite different. First of all there is a difference in how to assign points between Control Data Function Types and Control Transactional Function Types (St-Pierre et al., 1997a)

- Control Data Function Types
  These functions are divided in two groups, multiple occurrence group of data or single occurrence group of data.

  a) Multiple occurrence groups of data
     This group has the same structure as ILFs and EIFs in FPA. Consequently, they are counted in the same way by using the number of Data Element Types (DETs) and Record Element Types (RETs) and the corresponding complexity matrix.

  b) Single occurrence groups of data
     The number of points assigned to this group only depends on the number of DETs. For UCGs the number of points is calculated as ((number of DETs / 5) + 5). The formula for RCGs is (number of DETs /5)

- Control Transactional Function Types
  The number of points assigned to these types depends on the number of DETs. With the help from a translation table, the number of DETs is translated into a determined number of function points.

9.3.5.4 Counting procedure and rules

All definitions, point assignment table etc. are given in Appendix C, which can be used as a guideline for counting FFPs. In Figure 9:3, however, a comprehensive graphical description of the procedures of the FFP method is given.

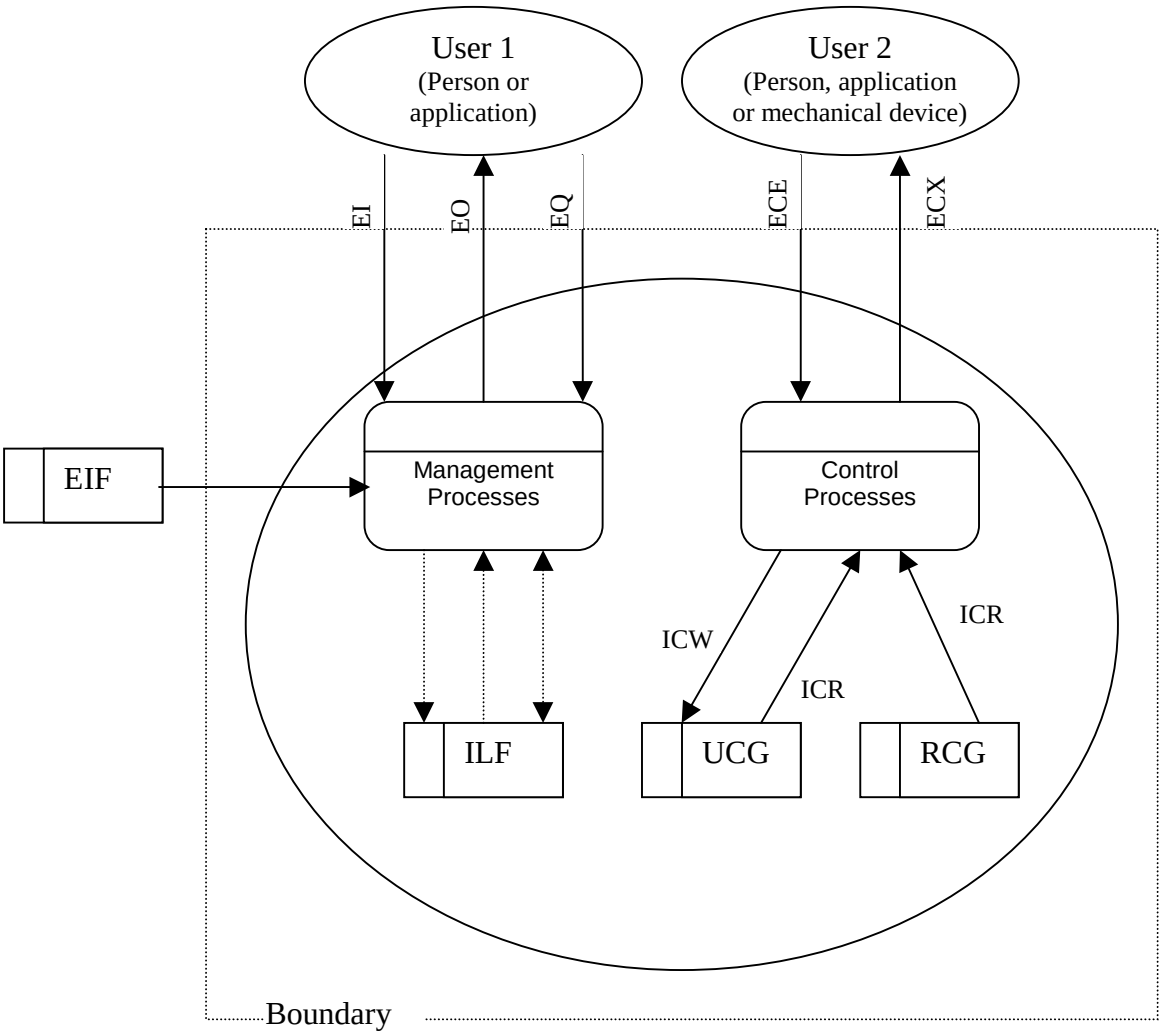
9.3.6 Mark II Method

Charles Symons developed this method, published in 1988, because of the shortcomings he thought Function Points contained. The two methods both use the same basic parameters in their calculations, but the Mark II method has fewer total parameters and is therefore thought to be conceptually simpler to use. Performance based on effort and size correlation is also considered equal using the two methods, but with greater variations expected on small projects when using the Mark II method. The acceptance of the Mark II method has been limited, though, due to the lack of wide-scale usage. It is mainly companies centered in the U.K that uses this method (Garmus & Herron, 1996).

9.3.7 3D Function Points

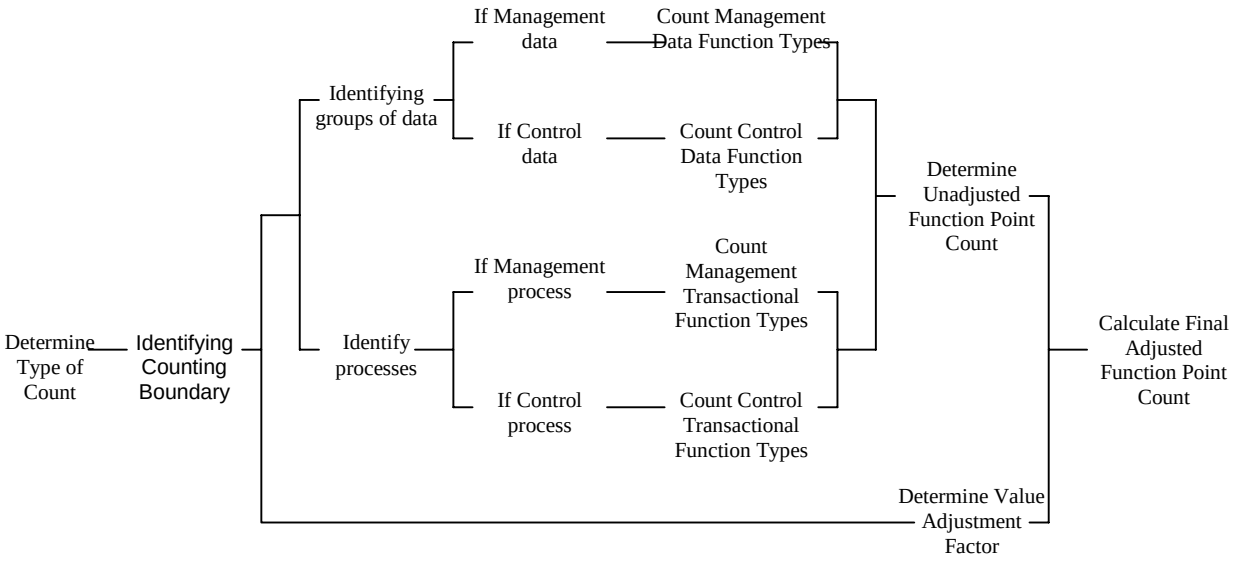
The 3D Function Point technique was developed between 1989 to 1992 by the Boeing Company to address two classical problems with the Albrecht approach. First the Function Point technique was considered difficult to use. Secondly the method was not a metric that could serve the scientific and real-time community very well. The 3D method has three dimensions: data, function, and control. These are considered as the problem types than an application can be expressed in. Each dimension represents some of the complexity that an application exists of. Sometimes one dimension dominates, but all of them must be analyzed to get an accurate measure. This technique is still an experimental metric and not widely used at this time (Garmus & Herron, 1996).

Figure 9:2 Diagram of FFP Function Types



(St-Pierre et al., 1997a)

Figure 9:3 FFP Counting Procedure Diagram



(St-Pierre, et al., 1997a, p. 15)

9.4 Conclusion of the size metrics

Even though we have stated in this chapter that Function Points do not fit very well for real-time systems the method should not be neglected at once. According to Garmus and Herron (1996) the basic Function Point Analysis (FPA) actually works very well for embedded and real-time software. Even practitioners agree that the method is flexible enough to be adaptable to other software environments than MIS software. Some of the definitions of inputs and outputs are however necessary to expand to fit better with real-time systems. Successful tests with Function Points have actually been applied to both public and private telephone switching systems and suchlike.

As the function point metric develops, the need for specialized variations such as feature point metrics will probably be reduced. However, the overall IFPUG literature needs to be revised and expanded to facilitate the use of Function Points for other applications than MIS systems. The conclusion is thus that a standardized method, like FPA, will have much greater potential of gaining acceptance. As the IFPUG also tries to adjust the method to other systems it will remain at a strong position. So far, the method is as we know not adjusted well enough for real-time systems. Full Function Points (FFP) on the other hand should be very well suited for these kind of systems. We have already described the character of real-time systems so you should be familiar with that they contain a large amount of sub-processes. This is considered in FFP with their several functional types. The algorithmic complexity is thus considered in FFP, both with concern to algorithmic volume (size) and difficulty (functionality). As FFP is an extension of FPA it is also developed with the intention to be comparable to those systems counted with FPA.

The facts we have stated here are the basis of our choices for the continued work. Mark II Method and 3D Function Points is not considered at all, as they do not have a wider acceptance in the software community. Feature Points also falls under this criteria. Our first intention was actually to continue with all three of the methods; FPA, Feature Points and FFP. Due to lack of time we had to withdraw one of them. As we just explained there is a natural bond between FPA and FFP. Therefore we find it reasonable to continue with these methods and test both of them in a pilot project to learn more of how they work in reality. This way we may also give well-grounded reasons for our later suggestions to ERV.

A clarification may also be made of why we considered functional size metrics at all. One general characteristic for all of the functional size metrics, which confirms our choice of them over Lines of Code, is that they consider the problem complexity we discussed in Chapter 7, *Software Complexity*. If the solution is adding more complexity this has no effect on the functionality of the application. It does not generate more points, but only decreases the productivity. This implies that functional size metrics are well suited to productivity measures.

9.5 Summary

Algorithmic complexity deals with spatial complexity and algorithmic volumes. It also reflects the complexity of the algorithm implemented to solve the problem and the degree of algorithms within a specific application. Algorithmic complexity is thus a measure of size that also comprehends the difficulty of the problem. As a size measure it is suitable for productivity measures. The traditional formula of productivity is size divided by effort. The most common way of measuring productivity of software projects has been by using number of lines of code as the size metric and man-hours as the effort metric. Counting Lines of Code is very easy as it can be executed by automated tools. It is a simple but insufficient metric and alternative metrics has therefore been demanded. In 1979, Allan Albrecht developed a method called Function Points. It was refined to what we today called the IFPUG Function Points Method or Function Point Analysis (FPA). IFPUG is an abbreviation of the International Function Point User Group, which is an unprofitable organization that supervises standardization and development suggestions for the method. Anyway, the method has a users perspective and measures size with user functionality. The more functionality and difficult degree of the functionality, the higher function point counts. The Function Point count has been criticized to not be applicable to real-time and other scientific systems. It was originally developed in and for the MIS environment, so the criticism is justified. Alternative methods like Feature Points and Full Function Points have therefore been developed to measure these harder kind of systems. The IFPUG is continuously improving their method, though, so that it can be adjusted to fit better with such systems. A common method for all types of systems would be best for the whole software community.

Based on extensive use and standardization work performed by IFPUG we chose FPA as one of our candidates for further test. The other metric we chose was FFP, since it is adjusted to real-time systems, is an extension of FPA and therefore comparable to systems counted with FPA.

10 Productivity, Quality and Performance

We have now moved one step further down in our model of software complexity, and have reached the concepts of productivity and quality. In the following chapter we will try to give a description of how the measures of complexity in connection with error-proneness and size can be used for describing such attributes of the software development processes and products as quality and productivity. Our main focus is on how quality and productivity respectively can be defined and measured. In the end of the chapter we will try to depict how these two attributes can be combined into one measure. We have chosen to use the concept "performance" to describe this fusion between the productivity and quality dimension.

10.1 Productivity

In 1975, Brooks observed that if you throw more people on to a late software project, you will make it later (Brooks, 1995). Many who have worked on large projects agree. Brooks' wisdom reflects our frequent emphasis on productivity. Because we do not always understand how to get the most and best from ourselves and others, we sometimes treat software production as if it were much like other production. By speeding up the assembly line or adding more personnel, we somehow expect to finish on time. We think of productivity as the production of a set of components over a certain period of time, and we want to maximize the number of components built for a given duration. But in fact, individual programmer productivity usually decreases as we add people to our projects. The added personnel can influence the quality and not always positively.

We begin our investigation of productivity in a rigorous way. That is, we ask ourselves what attribute we are measuring, and what intuitive properties it has that must be preserved by any measurement we make. Next, we ask which entities have the attribute. Only then can we address specific measures of productivity.

10.1.1 A definition of productivity

In economics, productivity is defined in a straightforward way (The New Encyclopaedia Brittanica, 1991, p. 719):

> "a measure of productive efficiency calculated as the ratio of what is produced to what is required to produce it. The inputs taken as the denominator of the ratio are usually the traditional factors of production – land, labour, and capital – taken singly or in the aggregate"

The idea of comparing input with output is useful for software developers, as well. Intuitively, the notion of productivity involves the contrast between what goes into a process and what comes out. Somehow, increasing the inputs should increase the outputs, or improving the process for the same inputs should increase the outputs. However, measuring the productivity of a software engineer, or of a software-engineering process, is not at all intuitive. In other words, we do not believe that it its entirely simple to apply this way of thinking on the software development process. The relationship between input and output is not that straightforward, when the input is the experience, knowledge and time of the system developer and when the output is computer software. This finding originates from the fact that one has difficulties to

understand what constitutes the set of inputs, and to know how process changes influence the relationship of input to output. What we have found instead is that software engineers define productivity in terms of a measure, rather than considering carefully what attribute is being captured. The measure most commonly used is one of size over effort (Jones, 1996). That is, the size of the output is compared with the amount of effort input to yield a **productivity equation**:

$$productivity = \frac{size}{effort}$$

Size is usually measured as Lines of Code (but can be any size measure, including those described in Chapter 9, *Algorithmic Complexity and Size Measures*), and effort is measured in person days or person months (Fenton & Pfleeger, 1996). Thus, software developers often calculate productivity as:

$$productivity = \frac{Lines\ of\ Code}{person\ months}$$

The equation's simplicity hides the difficulty of measuring effort. Effort is not something that can be recorded as easily as reading the speed of a car or the temperature. Indeed, many projects that have reached completion cannot report actual expended effort. When a person spends a day working on a project, the "day" measure does not reflect whether the day consisted of 8, 12 or 16 hours.

More importantly, as some of us are more effective on some days than on others, one eight-hour day of work can mean eight hours of productive effort for one person but only two hours for another. Similarly, we often count two half-time workers to be equivalent of one full-time worker. But the contributions of a half-time worker may be very different from 50% of a full-timer. Other effort considerations involve the contributions of support staff whose hours are charged directly to the project, and of corporate staff (such as researchers) whose time is paid from non-project funds but who nevertheless make a significant contribution to the project's output (Goodman, 1993).

The numerator, size, presents problems too. The productivity equation view output only in terms of size, so that the value of the output is ignored. We must consider productivity in light of the benefit we deliver, not just the size of the code. It does no good to be very effective at building useless products. We will come back to how we can take account of quality factors later in this chapter.

There are other concerns about the productivity equation and its underlying assumptions. By considering output solely in terms of the number of components produced, this view of productivity treats software development to be much like any traditional manufacturing process. For example, consider the automobile production. Designing a car is clearly similar to designing a complex software system, and both design processes involve prototyping and choosing among several alternatives. However, for automobile manufacture, the primary focus of the implementation process is replication: building many copies of the same item, perhaps with mild variations (such as color, trim, or radio type). Here, it is customary and reasonable to

measure productivity in terms of the number of items produced per person month. Each item has a clearly understood utility in terms of a known market price, and this measure of productivity corresponds well with our intuition about the effectiveness of the manufacturing process.

By contrast, there is usually little or no replication in software development. And even when replication is required, the cost of each copy is generally minimal in comparison with the cost of building the original. This small replication cost is a key feature that distinguishes software "manufacture" from more traditional manufacturing. Another aspect is the role of the creative process. Just as it would be silly to measure the productivity of poets by computing the number of words or lines they produce and dividing by the number of hours they spend writing, it may be equally inappropriate to measure software productivity using the productivity equation (Fenton & Pfleeger, 1996).

Instead, we need a measure of productivity that has a more direct relationship to quality. Similarly, we need to measure input by more than effort or time, as other resources (such as tools and training) are "consumed" during the production process. The combination of the productivity and quality dimensions will be discussed later in this chapter. However, we feel that, for reasons explained in Chapter 5, *The Situation at Ericsson Mobile Data Design*, man-hours is a simple but useful estimate of effort used in an ERV software project. If we choose to take other factors into consideration when determining effort, we may create a model that is too complex to be useful.

10.1.2 Productivity of what?

The discussion above shows us that we must take care to distinguish the productivity of a process from the productivity of the resources. In many instances, we are more concerned about the latter. We want to know how productive the developers are, so that we can take steps to improve their productivity. This need to measure and understand personnel productivity has its drawbacks though. When people feel they are being monitored and measured, they may become suspicious and resentful, and they may supply poor-quality data as a result. By relying on a goal-driven approach to measurement, and by making clear the likely benefits to all concerned, a measurement program can avoid this problem.

Of course, there are other resources whose productivity we can measure. For example, we may evaluate the productivity of a compiler. Even in this relatively simple example, people may confuse product or resource attributes with process attributes. Although the productivity of a compiler (a product or a resource, depending on whether we are building it or using it) must take into account the process of compilation (that is, running it on some source code), it is incorrect to talk of the productivity of the compilation process.

In the discussion that follows, we do not differentiate clearly between product, process, and resource productivity, as they are in fact intertwined. We can meaningfully refer to personnel productivity during a given process while working on a particular product, and it is in this sense that productivity is an external resource attribute. That is, we need a context for measuring and assessing productivity. For example, consider the productivity of an individual programmer during the coding of a program. Any measure of the programmer's productivity must reference the coding

activity, as opposed to design or testing. Table 10:1 shows some examples of typical processes and products we should consider when measuring the productivity of certain typical resources. We have also added a column for relevant resources used, since they may also be important.

Table 10:1 Productivity of resources

Resource (whose productivity we wish to assess)	**Process**	**Product**	**Examples of resources used**
Programmer	Coding	Program	Compiler
Programming team	Coding	Program	Compiler
Programmer	Testing	Program	Debugger
Programmer	Coding and documenting	Program, documentation, and user manual	Compiler and word-processor
Designer	Developing	Set of detailed module designs	CASE tool
Specifier	Constructing	Formal specification	-
Programmer	Maintaining	Set of programs plus documentation	Program
Compiler	Compiling	Source-code program	Microprocessor

(Fenton & Pfleeger, 1996, p. 337 (modified))

Table 10:1 highlights several possible limitations of the productivity equation. Notice how the formula addresses only the first two examples; that is, it can be used only to measure the productivity of programmers during coding. It is irrelevant for assessing the productivity of other software personnel performing other development tasks. In fact, whether the formula actually measures programmer productivity during software development in the intuitive sense discussed earlier is also highly questionable. We are not suggesting that the productivity equation should never be used. Rather, we suggest that the equation should not be defined and used as the only measure of (personnel) productivity, as it captures only a narrow sense of what we intuitively mean about productivity (Fenton & Pfleeger, 1996).

We have mentioned code reuse in earlier chapters, and obviously there is a difficulty in measuring the size of reused code. From the perspective of productivity, we must also consider the impact of reuse. The inclusion of previously constructed code will certainly increase the value of the productivity equation, but unless the code is executed, we have no real increase in productivity (Möller & Paulish, 1993). Thus, productivity as a measure defined by the productivity equation fails to satisfy, what mathematical theorists call, the representation condition for measurement. We can demonstrate this failure with an example.

When we measure programmer productivity, our entities are individuals producing specific programs. Suppose that P is the entity “Fred producing program A”, that it takes Fred 100 days to complete A, and that A is 5000 Lines of Code (LOC). According to the productivity equation, the productivity of P is then 50 LOC per day. Does this proposed measure capture our intuition about productivity in the sense of satisfying the representation condition? To see that it does not, suppose that Fred adds another copy of program A to the original one, in such a way that the second copy is never executed. This new program, A’, has 10000 LOC, but is functionally equivalent to the old one. Moreover, since the original version of A was already tested, the

incremental development time is nil, so the total time required to create A' is essentially equal to the time required to create A.

Intuitively, we know that Fred's productivity has not changed. However, let P' be the entity "Fred producing program A'". The productivity equation tells us that the productivity of P' is then 100 LOC per day. This significant increase in productivity (according to the measure) is a clear violation of the representation condition, telling us that the productivity equation does not define a true measure, in the sense of measurement theory. You may protest, saying that we can still use the productivity equation if we do not count reused code. However, consider the situation in which a programmer decides to remove a block of code from his program. Has his productivity suddenly decreased? We think not.

10.1.3 Proposed measures of productivity

Despite its theoretical problems, the productivity equation will continue to be used for many years. Its appeal derives from its simplicity and ease of automatic calculation. Moreover, because productivity (as defined by the productivity equation) is a ratio scale measure, we can perform all reasonable statistical analyses on it. In particular, we can meaningfully compute arithmetic means, yielding information about average team productivity across people in a given process or average programmer productivity across a number of different projects. We can also have meaningful discussions about proportional increases, decreases or comparisons in productivity.

In Chapter 9, *Algorithmic Complexity and Size Measures*, we pointed at some problems associated with measuring the size of software with Lines of Code. These problems stem from using Lines of Code not as a measure of length but as a measure of effort, utility, or functionality. If Lines of Code truly measured effort, utility, or functionality, then the measure ought to be indifferent to variations in counting convention and expressive power. In fact, it was precisely this problem, which lead Albrecht to formulate Function Points as a measure of functionality. As a result, some researchers (Fenton & Pfleeger, 1996; Möller & Paulish, 1996; Jones, 1996) have proposed that we measure programmer productivity as:

$$productivity = \frac{Function\ Points\ implemented}{person\ months}$$

If Function Points really do capture functionality, then this measure is attractive for several reasons:

- The function-points-based measure reflects more accurately the value of output.
- It can be used to assess the productivity of software-development staff at any stage in the life cycle, from requirements capture onwards. For example, to measure the productivity of the architectural designers, we only need to compute the number of Function Points for those parts of the system for which architectural design has been completed.
- We can measure progress by comparing completed Function Points with incomplete ones.

Many managers refuse to use Function Points because they find Lines of Code to be more tangible. That is, they understand what a line of code means, but they have at

best an intuitive grasp of the meaning of Function Points. Such managers prefer to "translate" function-point counts to line of code counts (Fenton & Pfleeger, 1996). Both Albrecht and Gaffney (1983) and Behrens (1983) have investigated the empirical relationship between Function Points and Lines of Code, and they have published tables showing the correspondence (Appendix D). Some cost-estimation tools calculate Function Points and translate them to Lines of Code automatically, for use as input to a Lines-of-Code-based cost model. This was also discussed in section 9.3.4.2--9.3.4.4.

A drawback of Function Points is their computational difficulty, compared with Lines of Code. For this reason, many practitioners prefer an alternative measure of functionality that can be extracted automatically by certain CASE tools. However, many of these automatic extractions depend on the use of a particular notion, and there does not exist any usable tool that can calculate Functions Points directly from source code (Jones, 1996).

Measures of length do not capture information about the quality and utility of software; neither do measures of functionality. Ideally, we want to relate our measures of productivity to the quality of the products, wherever possible and relevant. To do so, the measure of performance may be helpful, and we will return to it in section 10.3. For example, if test or operational failure data are available, then we can compute the operational reliability. This measure is relevant to a particular programmer's productivity only if the failures counted are attributable to errors made by an individual programmer. If we are measuring the productivity of the whole software-development team, as in the case of ERV, our data need not have such a fine granularity.

When it is not possible to compute external quality attributes directly, we can turn to the internal attributes described in Chapter 8, *Structural Measures of Software Complexity*. In particular, we are forced to make this compromise when we investigate the productivity of specifiers and designers. A designer may rapidly transform specifications into detailed designs. If the specifications are thereby poorly structured, we can relate the designer's productivity to quality for a more complete picture of design effectiveness.

10.2 Quality

A principal objective of software engineering is to improve the quality of software products. But quality, like beauty, is very much in the eyes of the beholder. In the philosophical debate about the meaning of software quality, proposed definitions include (Fenton & Pfleeger, 1996):

- fitness for purpose
- conformance to specification
- degree of excellence
- timeliness

However, from a measurement perspective, we must be able to define quality in terms of specific software product attributes of interest to the user. That is, we want to know how to measure the extent to which these attributes are present in our software

products. This knowledge will enable us to specify (and set targets for) quality attributes in measurable form.

External product attributes can be defined as those that can be measured only with respect to how the product relates to its environment. For example, if the product is software code, then its reliability (defined in terms of the probability of failure-free operation) is an external attribute; it is dependent on both the machine environment and the user. Whenever we think of software code as our product and we investigate an external attribute that is dependent on the user, we inevitably are dealing with an attribute synonymous with a particular view of quality (that is, a quality attribute). Thus, it is no coincidence that the attributes considered in this section relate to some popular views of software quality (Pfleeger, 1991).

In Chapter 8, *Structural Measures of Software Complexity*, we considered a range of internal attributes believed to influence quality in some way. Many practitioners and researchers measure and analyze internal attributes because they may be predictors of external attributes. There are two major advantages to doing so. First, the internal attributes are often available for measurement early in the life cycle, whereas external attributes are measurable only when the product is complete (or nearly so). Second, internal attributes are often easier to measure than external ones.

The objective of this chapter is to focus on defining which external attributes that are especially important, and consider how they can be measured. We begin by examining one of many standard quality models. However it is proposed as the basis for an international standard for software quality measurement, the IEEE (Institute of Electrical and Electronics Engineers) Std 1061 (IEEE, 1993). We use this model to identify key external attributes of interest, including *reliability*, represented by the number of known defects. Given the increasing use of software in systems that are crucial to our life and health, software reliability is particularly important.

10.2.1 IEEE Standard for a Software Quality Metrics Methodology

For many years, the user community sought a single model for depicting and expressing quality. The advantage of a universal model is clear: it makes the comparison easier between one product and another. In 1992 IEEE proposed the standard 1061 called “Standard for a Software Quality Metrics Methodology”, that builds on and further develops the ISO 9126 model (“Software Product Evaluation: Quality Characteristics and Guidelines for their Use”) (ISO, 1991). In the standard, software quality is defined to be “the degree to which software possesses a desired combination of attributes … required for that system” (IEEE, 1993, p. 4). Then quality is decomposed into six factors (some of these are defined differently in our model of software complexity):

- *Functionality*, i.e. the existence of certain properties and functions that satisfy stated or implied needs of users.
- *Reliability*, i.e. the capability of software to maintain its level of performance under stated conditions for a stated period of time.
- *Efficiency*, i.e. the relationship of the level of performance to the amount of resources used under stated conditions.

- evaluation of results) and the individual assessment of such use by users.
- *Maintainability*, i.e. the effort needed for specific modifications.
- *Portability*, i.e. the ability of software to be transferred from one environment to another.

The standard claims that these six are comprehensive; that is, any component of software quality can be described in terms of some aspect of one or more of the six factors. In turn, each of the six can be refined through multiple levels of subcharacteristics. The standard defines a process for evaluating software quality. Numerous companies use the IEEE Std 1061 model and framework to support their quality evaluations (Fenton, Iizuka & Whitty, 1995). Thus, despite criticisms, the standard is an important milestone in the development of software quality measurement.

10.2.2 Defect density as a measure of reliability

Software quality measurement using the decomposition approach clearly requires careful planning and data collection. Proper implementation even for a small number of quality attributes uses extra resources that managers are often reluctant to commit. In many situations, we need only a rough measure of overall software quality based on existing data and requiring few resources. For this reason, many software engineers think of software quality in a much narrower sense, where quality is considered only to be a lack of defects. Here, "defect" is interpreted to mean a known error, fault, or failure as discussed in section 7.3.1.

A *de facto* standard measure of software quality is *defect density*. For a given product (anything from a small program function to a complete system), we can consider the defects to be of two types: the known defects that have been discovered through testing, inspection, and other techniques, and the latent defects that may be present in the system but of which we are as yet unaware (Ohlsson, 1996). Then we can define the defect density as:

$$defect\ density = \frac{number\ of\ known\ defects}{product\ size}$$

Defect density is certainly an acceptable measure to apply to projects, and it provides useful information. However, before we use it, either for our own internal quality assurance purposes or to compare our performance with others, we must remember the following (Ohlsson, 1996; Fenton & Pfleeger, 1996; Grady, 1992):

- There is no general consensus on what constitutes a defect. A defect can either be a fault discovered during review and testing, or a failure that has been observed during software operation. The terminology differs widely among those who measure defects; fault rate, fault density and failure rate are used almost interchangeably. Thus, to use defect density as a comparative measure, we must be sure that all parties are counting the same things in the same ways.
- There is no consensus about how to measure software size in a consistent and comparable way. Unless defect densities are consistently calculated using the same definition of size, the results across projects or studies are incomparable.

- Although defect density is a product measure in our sense, it is derived from the process of finding defects. Thus, defect density may tell us more about the quality of the defect finding and reporting process than about the quality of the product itself.
- Even if we were able to know exactly the number of residual faults in our system, we would have to be extremely careful about making definitive statements about how the system will operate in practice. Our caution is based on two key findings. Firstly, it is difficult to determine in advance the seriousness of a fault. Secondly, there is great variability in the way systems are used by different users, and users do not always use the system in the ways that were expected or intended. Thus, it is difficult to predict which faults are likely to lead to failures, or to predict which failures will occur often.
- Studies show that it is quite possible to have products with a very large number of defects failing very rarely, if at all (Ohlsson, 1996). Such products are certainly of high quality, but their quality is not reflected in a measure based on defect counts. Hence, a very accurate residual fault density prediction may be a very poor predictor of operational reliability.

Despite these problems with using defect density, we understand the need for such a measure and the reason it has become a *de facto* standard in industry. Commercial organizations argue that they avoid many of the problems with the measure by having formal definitions that are understood and applied consistently in their own environment. But what works for a particular organization may not transfer to other organizations, so cross-organizational comparisons are dangerous. Nevertheless, organizations are hungry both for benchmarking data and for predictive models of defect density.

To get a comprehensive picture of software quality it is obvious that we need to take all factors IEEE Std 1061 into consideration. Depending on which sub-factor of quality that is most important for the system in question, we can choose to focus on measuring one or a few sub-factors. Researchers have also developed methods and measures to evaluate primarily the maintainability and usability factors. Measuring maintainability inevitably involves monitoring the maintenance process, capturing process measures such as the time to locate and fix faults. Some internal attribute measures, notably the structural measures described in Chapter 8, *Structural Measures of Software Complexity*, may also be used as indicators of likely maintainability.

Usability must involve assessing people who use the software, and an associated concept with usability is user-friendliness. To get a useful measure of this factor we have to quantify the effort required for learning and operating a system. We will not develop this discussion of maintainability and usability measures any further. Instead, we are referring to Chapter 5, *The Situation at Ericsson Mobile Data Design*, especially section 5.1.1, where we explained why reliability is the most important quality attribute for ERV. As we have seen in this chapter, defect density is a widely used and acknowledged approximation of reliability. Therefore we would like to suggest defect density (or "inverted defect density" as we will see in Chapter 12, *Conclusions and Recommendations*), and internal structural attributes as equivalents of software quality. Further, if our model of complexity is correct, the structure

complexity measures could be used as estimators of other quality factors, as well, such as usability and maintainability.

10.3 Performance

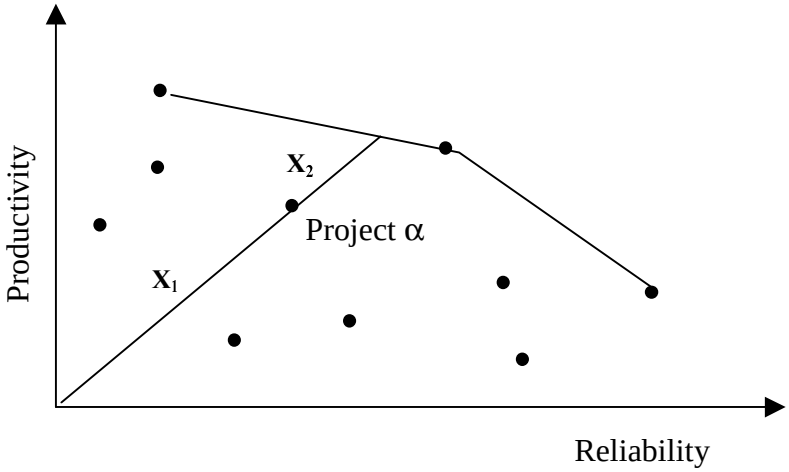
As we mentioned in our discussion of productivity the measures of productivity and quality is more useful if we can combine them and weigh them in a way that helps us to understand our overall achievement. If we focus to hard on productivity, we may improve our efficiency and lower our project costs, but this gain is worthless if we are not building quality systems that meet the demands from our customers. Similarly, if we are aiming to create the perfect system, we may loose control of the costs. Moreover, the time it takes to improve and correct the system may cause a late delivery of the product. In the software and computer business this often means that the product is delivered at a time when it is obsolete.

The approach of combining the quality and productivity dimensions of software is also useful for companies from another perspective. As we have seen in Chapter 5, *The Situation at Ericsson Mobile Data Design*, the situation at ERV is such, that for some projects the level of reliability is crucial for the decision if the system should be delivered or not. The quality is not allowed to fall below a certain level, because of internal or external demands, and if it does, the company must be prepared to bring more resources into the project in order to improve the quality. In other projects, the timeliness of releasing the product is the most important success factor. If our surveys tell us that the market will be ripe for our product in, let us say 6 months, it is important that our project will be finished before this juncture. If we fail to accomplish the product in 6 months, our competitors will most likely manage to do it, and we will loose market shares.

Thus, of these reasons it is important to find a way to integrate the two main aspects of software: productivity and quality. We will propose one way of doing this, and we call this a *performance measure*. The method behind the measure builds on something called *Data Envelopment Analysis* (DEA), which is in its turn a development of the mathematical technique known as linear programming (Goodman, 1993). The idea is that we can identify two characteristics of our process or products that are common across the organization. In our example we will consider productivity and quality (or more theoretically correct reliability). The next step is to define measures associated with these characteristics, and collect data from a number of projects.

One thing to consider in this case is to present a graph with productivity as the y-axis and quality (or reliability) as the x-axis (see Figure 10:1). The performance of each project can then be described in terms of two-dimensional vector coordinates. If the managers would rather want the information in non-graphical form, to make comparisons against “best in class” we can identify the boundary cases and use these to describe an envelope around the set of observations. For any particular project, drawing a vector from the origin, through the project point to the boundary can then represent its performance. Performance is then expressed as a ratio of the distance from the origin to the project point compared to the total length of the vector to the boundary.

Figure 10:1 Graphical presentation of the performance measure



$$Performance\ of\ \ project\ \alpha = \frac{x_1}{x_1 + x_2}$$

(Goodman, 1993, p. 167 (modified))

The advantage of DEA is that the technique can be applied to more dimensions than two, in fact any number, subject to the computation capabilities available. We will develop this measure further in Chapter 12, *Conclusions and Recommendations*, when we will discuss which specific measures of productivity and quality (reliability) ERV should use and at what time during the software development process the performance measure should be implemented. However, we do not recognize the need to develop the theoretical foundations of DEA further in this report. Those can be found in Goodman (1993).

10.4 Summary

Productivity was defined in this chapter by using the common productivity equation, where productivity equals the quotient between size and effort. Historically the lines-of-code-measure has been used as an estimation of software size, but the Function Points measure is gaining more use as an alternative size measure. The productivity equation still has some shortcomings that also have been discussed in this chapter.

Commonly, people are the only resource that is thought of as an object for productivity measurements. However, we also have to recognize that different types of personnel can be measured, and different measures and techniques must then be applied. Programmers, programming teams, designers and specifiers can all be the subjects of productivity measurements. Even a compiler is applicable for this purpose.

Software quality can be defined in many different ways, depending on who is making the definition. We are using the definition in IEEE (Institute of Electrical and Electronics Engineers) Standard for a Software Quality Metrics Methodology, where IEEE suggests that quality consist of six factors: functionality, reliability, efficiency, usability, maintainability and portability. For reasons discussed in Chapter 5, *The*

Situation at Ericsson Mobile Data Design, we will concentrate on reliability as our main quality concern. A *de facto* standard measure of software quality in general, and reliability in particular, has been developed in the industry, usually called defect density. Even if there are some cautions to be made when applying this measure, we would like to suggest it as a useful and relevant estimation of software quality.

The last section of the chapter is devoted to a combined measure of project performance, where the two dimensions of project productivity and software quality is connected to get an overall picture of the project and what it has produced. The measure builds on a technique called Data Envelopment Analysis (DEA), and can be expressed both graphically and in an equation.

11 Field Tests at ERV

During the latter part of our work at ERV we were focusing on the evaluation of the chosen functional measures of software, Function Point Analysis (FPA) and Full Function Points (FFP), in relation to the ERV's organization and business. We applied both these methods to a part of a project that was concluded about a year ago (June 1998). It can shortly be described as a mobile data system based on the Japanese standard for mobile data communication (PDC). It lasted for about one year and involved about 50 persons.

We divided the application into modules, according to objective standards spelled out in the system documentation. Each one of these modules was logically coherent and related to the other modules, which meant that they together formed a working unit. In FPA and FFP terms we looked at each of these modules as separate applications. A wide range of module sizes was represented in this application. We wanted this distribution of size, since comparisons between the modules were then easier to make. The reader may want to recall the method we used during these field tests. If this is the case, we refer to section 4.4, for a more extensive explanation of the course of action.

Our hypothesis before the tests began was that the FFP would be more useful than FPA for the ERV organization. If we could falsify this hypothesis, then FPA would be better for ERV. In the following sections we will try to specify how the tests were performed combined with our continuous results and the final results we received concerning the applicability of the methods.

11.1 The results

As mentioned we separated the project of study in modules, all to all twelve pieces. For reasons of simplicity we will call them A, B, C, D, E, F, G, H, I, J, K and L. In the IWD document module A, B, C and D were treated as one unit. Thus, during the first phase of the tests, we did not count A, B, C and D as separate modules, since that was not possible. However, this does not mean that the sum of FPA and FFP count for these modules in the second round of the counting is comparable to the figure we received after the first phase for A, B, C and D as a unit. This is due to the fact that when applying FPA and FFP some parts of these modules are counted several times, above all the communication internally and externally with other modules.

The results that came out of testing the methods on this project are summarized in Table 11:1. Some important conclusions can be drawn when looking at this table. First of all the modules H, I, J, and perhaps also modules A and K could be regarded as I/O-heavy modules, i.e. modules with a large degree of communication with other parts of the system, but not so much algorithmic complexity. This can be concluded by looking at the quota between the figures for FPA and FFP. If this quota is relatively high the FPA method has a greater impact, and thus it is probable that these modules contains much input and output, processes that are counted high with the FPA method. The modules B, C and D, on the other hand, could be regarded as modules with many sub-processes and complex algorithms, since the FFP method has a greater impact relatively to FPA. Finally, the figures in the two columns for the second round of counting are higher than in the columns representing the first round.

This means that when we increase the level of detail, regarding the documents and code analyzed, we usually find more functionality to take into account. The exceptions are modules H and I. The reason for this is that they are only concerned with input and output, i.e. the functionality is included in the IWD document.

Table 11:1 Results of counting FPA and FFP

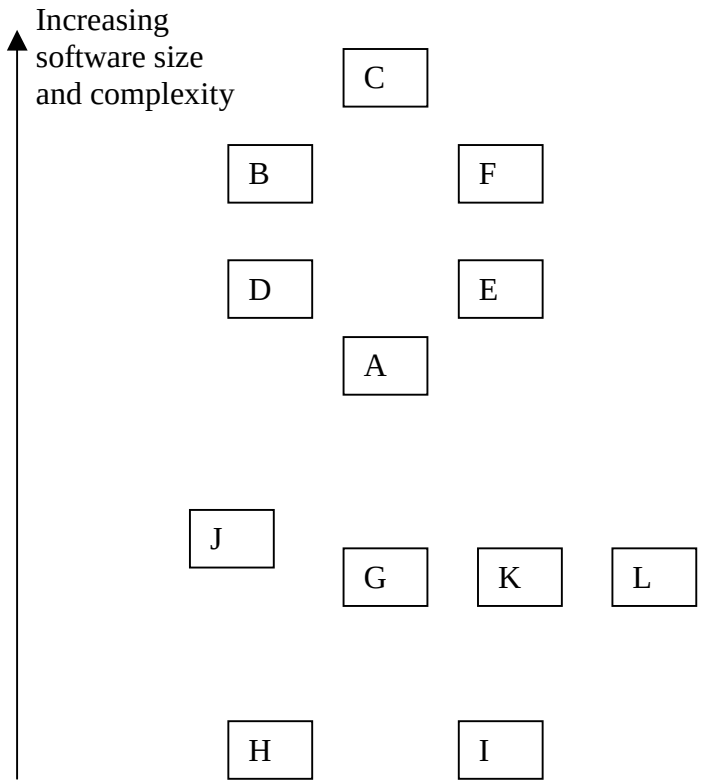
	Counting based on IWD's		**Counting based on IS's and source code**	
Module	**FPA**	**FFP**	**FPA**	**FFP**
A+B+C+D	231	42		
A			455	116
B			296	134.4
C			276	183
D			268	81
E	200	35.6	283	89.6
F	184	54.2	252	79.2
G	153	42.2	201	62.4
H	117	22	121	17
I	91	21	86	16
J	89	18	170	38.4
K	65	13	137	32.8
L	65	13	113	37.2

11.2 Validation of our results

Our main concern was to validate the results we got from our tests from an objective viewpoint. However, a fully objective view of this specific project at ERV was hard to acquire. The approach we chose was therefore to compare our results with the jointed opinion of the system developers involved in the project. When we had concluded our tests, we asked three of the system developers involved in the project to place the modules in order of precedence. We explained that we wanted them to order the modules according to size *and* complexity, since FPA and FFP combine these factors. The results of our tests were not shown to them until their order of precedence was done.

Naturally their knowledge of the modules varied. Some of the modules they had developed themselves, others were examined by them, but there were always one or a few modules that they had very little knowledge of. Thus, we compiled the opinions from these three persons. In this compilation, when a person had greater knowledge of a module than the others, his opinion took precedence over the others regarding this specific module. However, the opinions were very similar, especially regarding which modules that were the most and the least complex. The order of precedence that became the result of the "opinion compilation" can be seen in Figure 11:1.

Figure 11:1 Module complexity according to the system developers



The first thing to notice is that we can distinguish three groups of modules. The most complex group (C, B, F, D, E and A) consists of modules with many lines of code and many algorithms. The middle group (J, G, K and L) is made up of modules that contains a certain amount of input and output, but also fragments with algorithms and complex functionality. The last group (H and I) of modules are units with a pure I/O-functionality.

If we compare our tests of the FPA and FFP methods on these modules with the opinion of the system developers, we find that they are rather congruent. The column in Table 11:1 that best agrees with Figure 11:1 is without doubt, the counting of FFP on IS's and source code (4th column). The order of precedence there is C, B, A, E, D, F, G, J, L, K, H and I. The explanation for the exceptions (A and G is counted higher with FFP than according to the system developers, and F is counted lower) can be found in the system developer's perception of the problem. We suspect that they interpreted the order of precedence only as a software complexity issue and disregarded the size factor. If we look at A and G we also find that they are made up of relatively many lines of code, and F is rather complex but is also a low-volume module.

Thus, the outcome of this simple but rather straightforward validation of our results speaks in favor of a detailed counting with the FFP method. As we predicted, the FPA method fails to take into account the complexity factors that are inherent in real-time systems, such as the number of algorithms and the number of sub-processes (see 5.1.2, *Real-time systems*). Moreover, we need a detailed and comprehensive documentation in order to make accurate use of the FPA and FFP methods. To be able

to count all functionality included in a system, we need low-level design documents, and perhaps even source code to find the parameters needed.

11.3 Limitations

There are a number of possible sources of error that have to be recognized regarding our test results. Firstly, we may have misunderstood the functionality of the modules studied. Despite the fact that we spent almost four weeks on reading documentation on the system, we may have missed out important parts. Tele- and data communication systems are technically very complicated, and since we do not have a thorough knowledge of electronics and telecommunications it is hard to understand all parts of such a system.

We may also have misinterpreted the rules for FPA and FFP counting. In the case of FPA, the method is, as we have mentioned before, adjusted for MIS systems (Management Information Systems). Thus, when counting a real-time system, the definitions of the elements of the FPA method have to be interpreted from case to case. Regarding FFP the situation is more favorable. However, since the method is rather new, some initial difficulties ("teething problems") may be identified, which means that rules are not as clear as one would like them to be. We have tried to minimize this source of error by writing e-mails to one of the researchers that developed the method, Jean-Marc Desharnais (1999). He answered our questions about the interpretation of FFP rules in detail and with great enthusiasm.

Finally, miscalculations may have occurred during the counting sessions. To err is human, and the human factor may have taken its toll, when the summation was done or the equations used. Despite all these problems we think that our tests show that the FFP method are superior when it comes to estimating the size and complexity of real-time systems. In other words, the hypothesis established in the beginning of these tests, could not be falsified.

11.4 Summary

To be able to decide if Function Point Analysis or Full Function Points should be used for counting the projects at ERV, both methods were tested on a concluded project (or more correctly: *part* of a project). The tests were made in two rounds. In the first round only general documents, overviews of the system, were considered, and in the second round more specific documents of the low-level design and the source code were included. These results were then compared to the opinions of some of the system developers that had been involved in the project. The comparison showed that the FFP method applied to detailed documentation and code was most successful in terms of agreement with the opinions of the developers. These results will be used as one of the pieces in the puzzle of recommendations and suggestions we are going to put together in the next chapter.

12 Conclusions and Recommendations

We have now reached the end of our journey through the problem of software complexity, and we have arrived at the terminal station: the conclusions of our research and our recommendations for Ericsson Mobile Data Design AB (ERV). This chapter is made up of three parts. In the first section we are going to sum up our findings in this master thesis, and the set of questions we had in the first chapter will be answered. In the second section we are spelling out the concrete recommendations for ERV we have arrived at during our research. As a final round off, we will give suggestions for further research in the area of software metrics at ERV.

12.1 Conclusions

During this report we have answered the set of questions that we established in Chapter 3, *Problem*. Some of them have been easier to answer than others, and in some cases we have not been able to give a complete and comprehensive solution to the problems that formed the basis for this master thesis. Detailed answers to our questions are given in Chapter 7-11. However, in this section we will give a summary of our conclusions. Each set of questions will in turn be presented and answered.

- ***How can software complexity be defined? Where does software complexity come from and what does it lead to? How is software complexity related to software productivity and quality?***

The first set of questions dealt with the concept of software complexity. The reply to the first question was a definition of software complexity, saying that it is the degree of difficulty in analyzing, maintaining, testing, designing and modifying software. Moreover we defined the concept by dividing software complexity in four different parts: problem complexity, algorithmic complexity, structural complexity and cognitive complexity. Since we related mainly *algorithmic and structural complexity* to productivity and quality, we focused on these concepts in the rest of the report. In the same chapter we also answered our second question about software complexity by claiming that there are two sources of software complexity: the *complexity of the problem* and the *complexity of the solution.* The last-mentioned concept can also be referred to as *complexity added* during the different development phases, mainly design and coding. We found two main effects of software complexity: ***error-proneness*** and ***size***. These main effects were related to productivity and quality in order to answer our third question. In our model of software complexity (Figure 4:1) we connected error-proneness to the *quality* dimension of software through the concepts of *usability*, *reliability* and *need of change*. Moreover, we related size to the concept of *productivity*, by determining the level of *maintainability*, *understandability* and *computer power* needed. However, error-proneness is also, through the need of changing a system, influencing project productivity in different ways, as well as size, through maintainability, influences the quality level of software. Thus, the relationship between software complexity, productivity and quality is not straightforward.

- ***How can software complexity be measured? Which methods and measurements are available for capturing different aspects of software complexity?***

Further on in the report we answered the second set of questions, regarding methods for measuring software complexity, by identifying different measures of the aspects of software complexity we chose to focus on: structural and algorithmic complexity. Thus, the first question was answered by stating that software complexity can be measured either as structural or algorithmic complexity. To be able to answer the second question, we focused on the most important and used methods and measures. The structural complexity measures we identified were *McCabe's cyclomatic number*, measuring control-flow structure, *Henry and Kafura's information flow measure*, measuring data flow, and *Halstead's measures*, measuring data structure. Regarding the measures of algorithmic complexity, two alternative groups of size measures were found: *plain size* metrics (Lines of Code) and *functional size* metrics (variants of Function Points). Since there are many shortcomings with using Lines of Code, we chose to focus on different types of functional size metrics, including Function Point Analysis (FPA), SPR Function Points and Feature Points, Full Function Points (FFP) and some minor measures as 3D Function Points and Mark II Function Points.

- ***Which method or measurement of software complexity should be chosen with regard to ERV's need of a comparative measure of project productivity and quality? How is this method or measure going to be implemented in ERV's system development process?***

The third set of questions dealt with suggestions to ERV of how to use the measures and methods that were found. The first question was answered in three steps. First, we selected McCabe's cyclomatic number to be used at ERV as a measure of structural complexity and thereby also an estimate of product quality, since it was widely used, supported by automatic tools and applicable early in the development process. To be able to choose a measure of algorithmic complexity and a functional size metric, field tests were performed at ERV of the two main candidates: FPA and FFP. These tests showed that FFP is the method that is best adapted to the type of systems produced at ERV, namely real-time systems. When discussing the measurement of software project attributes we claimed that a combined measure of productivity and quality, called *performance* measure, should be used in order to capture as many aspects of software complexity as possible. The second question is answered in the next section.

12.2 Suggestions

Since our work has been performed at ERV, the goal for our research has always been that the conclusions should be used for concrete suggestions to ERV. So far, we have dropped some hints of how we would like ERV to measure their projects, but in this section we will spell this out more clearly. We have chosen to break these suggestions up in two parts: short-term and long-term suggestions. The short-term suggestions are more concerned with what can be done immediately for testing software metrics and the measurements proposed in this report. The long-term suggestions could be implemented right now, but are more of a strategic nature, i.e. explain how a long-term measurement program could be arranged.

12.2.1 Short-term suggestions

In Chapter 10, *Productivity, Quality and Performance*, we discussed different ways of measuring productivity and quality and we reached the conclusion that weighing these dimensions together in a measure of performance is a preferable way of doing this. However, we did not relate these findings to the ERV organization and

business in any way. To specify which measurements should be performed, at which time and by whom, is our main focus in this section.

If we recall our discussion of performance in section 10.3 it was represented by a two-dimensional vector, with productivity and quality measurements as inputs. We would also like the reader to recall the ERV model of system development, Darwin, spelled out in section 5.3. The design of this model has guided us to a certain way of defining where in the process the measurements should be performed.

We would like to suggest two types of measurements: expected performance and real performance. The measurement of expected performance is recommended to take place in the middle of the system development process. The measure itself consists of one measure of productivity and one of quality:

$$(Expected)\ Productivity = \frac{Implemented\ Full\ Function\ Points}{Man\ Time}$$

$$(Expected)\ Reliability = \frac{Implemented\ Full\ Function\ Points}{McCabe's\ Cyclomatic\ Number}$$

These measurements are suggested to take place at MS6 in the Darwin model. The reason for choosing this point of time is that a preliminary version of the source code is available, as well as the low-level design with documents needed for carrying out Full Function Points count and the automatic measure McCabe's Cyclomatic Number.

The reasons for defining productivity this way has been explained in Chapter 10, *Productivity, Quality and Performance*, and we refer the reader to section 10.1.3 for a complete discussion of this subject. The definition of effort as "Man Time" is made intentionally loose, since we do not have the authority to decide in which unit ERV should count time spent on a certain project. However, an important thing to emphasize in this context is that there must be a consistency in the counting of "Man Time", i.e. it must be done in the same way and presented in the same unit (months, days, hours or some other preferable unit) at all times.

Full Function Points have been chosen as the eligible measure of software size at ERV, after analyzing the test results explained in Chapter 11, *Field Tests at ERV*. By "Implemented Full Function Points" we mean that only the functionality included and implemented in the system at the time of measurement, i.e. at MS6, should be taken into account. Similarly, "Man Time" is defined as the time put into the project up until MS6. The productivity measure then gives an indication of how productive the project team has been during the design phases of the project.

In Chapter 8, *Structural Measures of Software Complexity*, we described alternative measures of structural complexity and what aspect of structural complexity they are measuring. In the same chapter McCabe's measure was chosen because it is widely used among software developments, tools for automatic counting are available, and it can be implemented early in the software development process. However, we also recognized that a more extensive examination of the structural complexity measures has to be made, in order to decide which measure that best suites ERV's purpose.

The measure of expected quality can then be seen as an inverse measure of defect density, and McCabe's measure is therefore used as an estimation of the number of errors in the code.

The measuring of real performance is recommended to take place in the end of the system development process. The measurements are direct parallels to the ones used for expected performance:

$$Productivity = \frac{Implemented\ Full\ Function\ Points}{Man\ Time}$$

$$Reliability = \frac{Implemented\ Full\ Function\ Points}{Number\ of\ Known\ Defects}$$

The difference between these measures and the ones proposed for expected performance is that the actual number of known defects, found during the test phase, has replaced McCabe's Cyclomatic Number. In that way we get a measure of how the system in reality is performing with regard to the quality dimension.

Implemented Full Function Points is here defined as the number of Full Function Points implemented up until MS9. Between MS6 (when expected performance measurements are made) and MS9, the main part of the testing activities take place. Functionality may be added or withdrawn from the system during these sessions, and Full Function Point counts must therefore be made again.

Spent time between MS6 and MS9 should be calculated and added to the figure for "Man Time" used for calculating expected performance. In this way we get the total time invested in the project up until MS9. At MS9 there should also exist reports of errors localized during the integration and verification phases. We leave to ERV to decide exactly which figure to choose as representing the number of known defects. However, consistency is again very important.

When we have calculated these measures of productivity and quality we are then able to get an overall picture of expected and real performance by drawing the graph described in section 10.3 (see Figure 10:1), or calculate the performance ratio. The measure of expected performance can be used to evaluate if more resources are needed for testing the system, and estimate how long time the integration and verification sessions will take. If the reliability (quality) of the system is the principal goal, the project should end up in the right part of the graph. On the other hand, if timeliness of the product is most important, we are aiming at placing the project as high up as possible relatively to other projects. As more projects are developed and productivity and quality data is collected, we can create a database to compare our current projects with. In this way we are able to estimate how the project is developing after MS6 by looking at similar projects (in terms of calculated productivity and quality) that have been concluded.

Real performance, on the other hand, can be used for evaluating the projects after conclusion. Did we reach our goals regarding reliability? How productive were we relatively speaking? Did the new methods or tools used in the project have any effect on the productivity rate? Such questions, and many more, could be answered

by looking at the real performance measure. The figure could also be used for comparisons with other Ericsson or external companies or as information to the customer. However, if the measure is going to be useful in this context, it is obvious that the same calculations have to be made at the companies in question.

By whom are these calculations going to be made? As we explained in Chapter 9, *Algorithmic Complexity and Size Measures*, there is no such thing as the ideal composition of a software measurement group. Since McCabe's cyclomatic number can be calculated automatically, and data about the number of defects and man time are collected already, the counting of Full Function Points (FFP) is the complicated part. ERV is a rather large organization, and educating everybody involved in system development on the FFP technique is therefore impracticable. Rather, we suggest that a few (two or three) employees, preferably people working with software engineering issues today, should be educated to become "experts" on the FFP method. However, bearing our own experience of counting FFP in mind, it is also necessary to include people with advanced knowledge of the software counted, i.e. the system developers, in the FFP counting group. If the expert knowledge of the system is combined with the expert knowledge of the FFP method, our opinion is that the counting session will be more effective and accurate. This implies that our suggestion is to include one or two experts on the FFP method, and a couple of system developers involved in the project counted, in the FFP counting group. The exact number of people depends on the scope of the project. After some practice, the calculation of FFP should not take more than a couple of days for an average project.

Some actions could be taken to make the measurement work easier. If the system developers are familiar with the definitions of the elements in the FFP method (i.e. UCG, RCG, ECE, ECX, ICR and ICW) they are able to make suggestions of where these can be found in the system. The experts on the FFP method could then decide where these parts fit in by interpreting the FFP counting rules. Moreover, a list of identified UCGs, RCGs, ECEs, ECXs, ICRs and ICWs, could be included in the design documentation for later use, when the system developers that worked in the project are not available any more.

The suggestions submitted in this section are things that can be carried out right away. They are intended to serve as a starting shot for some sort of a performance measurement program at ERV. Naturally, these actions must be performed in a context where a long-time strategy for the measurements have to be decided. Some suggestions of such a strategy are put forward in the next section.

12.2.2 Long-term suggestions

The long-term strategy for the measurements must be based on a well thought-out answer on the question: For what purpose are the measurements going to be used? Are they going to be used for comparisons with other Ericsson companies or perhaps benchmarking against competitors? Are they going to be used for estimating project development and to plan the consumption of resources and time? Are they going to be used for evaluating the effect of new methods and tools, by comparing internally with other ERV projects? There may be more uses that are not suggested here. However, what we would like to point out by asking these

questions is that the management have to make clear what the purpose of their measurements are, because it will decide what strategy the company need to pursue.

In the conditions for this master thesis, ERV established that they needed a measure to compare their projects to something. During our research we have come to the conclusion that this "something" is referring to that ERV, at an initial stage, wants a measure for comparing the productivity and quality of current projects with concluded dittos, for the purpose of evaluation. However, in the long term, ERV is interested in comparing their business with other Ericsson companies and competitors. If ways can be found for planning and estimating projects ERV admits that it is attractive, but it seems to us that the principal purpose is to use the measurements for comparisons.

If that is right we are suggesting that the following elements should be included in a long-term strategy:

- Create a framework for the collection of productivity and quality data. It is preferable that it is based on ERV's existing routines for collecting other types of project data.
- In an initial stage develop the FFP method internally by testing it on more projects and with system developing personnel included. It is important to carry through a training program, an advanced one for the two or three experts, and more basic education for the system developers involved in the counting sessions. This basic education has a two-fold purpose. On the one hand it is giving them the means for understanding how the measurements are carried out. On the other hand it motivates them to collect the data, when they know what it is used for.
- In the next stage, if comparisons with other Ericsson companies and competitors are demanded, begin to co-operate with the Software Engineering Management Research Laboratory (LRGL) and Software Engineering Laboratory in Applied Metrics (SELAM) at University of Quebec and Montreal in Canada, the developers of the FFP method, in order to receive guidance and new releases of the method. As we have mentioned, the method is quite new and improved versions of the FFP method are released continuously. There are also Ericsson companies in Canada that have tested the method and are partners of LRGL and SELAM projects. Using the experience of software measurements, and especially the FFP method, in these Ericsson companies, is naturally an important move to make when implementing a software metrics program at ERV.
- However, ERV should not focus too much on the FFP method, so that they loose the perspective on functional size measurements in general. Rather, we advice ERV to follow the development of the FFP and FPA methods closely, especially the work of IFPUG, the standardization organization for Function Points. According to the Function Point specialists we have been in contact with (P. Almén, personal communication, 16th March, 1999; M. Öhlin, personal communication, 16th March, 1999), it is developing an understanding in the Function Points' community that they have to move towards a common standard method for function point counting, that could be used irrespectively of which type of software that is developed. When this standard is agreed on, ERV need to have knowledge of it, to be able to compare their productivity with

competitors, since this standard probably will be used by most of the companies using the FPA method today.

These long-term suggestions are consciously presented as guiding principles without specifying concrete details. The reason for this is that the future is too uncertain, regarding the development of functional size measurements and ERV's business. The detailed strategic decisions have to be made by ERV themselves, but guidelines for these decisions have been proposed in this section. The short-term and long-term suggestions are also summarized in Appendix E, together with our suggestions of future research areas spelled out in the next section, to be used by the ERV managers as a reminder of our proposals.

12.3 Further research

During our research we have only had the time and resources to get an overview of the subject and a few solutions to ERV's software measurement problems. During this time our attention has also been drawn to the fact that there is still much work to do in this area. We have gathered these findings under this heading, were we would like to suggest areas for further research in general and at ERV.

Firstly, the possibilities to automate the FPA and FFP counting should be investigated. There are companies claiming that they have developed software tools that are able to count Function Points automatically on certain type of code or documentation. However, when examining these tools we found that they are neither accurate nor functional. The input has to be adapted to the tool to a great extent, and this is too time-consuming to be an alternative to manual counting. Thus, research could be made in this area, exploring which parts of the FPA or FFP counting that could be done automatically, exploring available software tools, and maybe developing an internal tool suited for the ERV organization.

Secondly, the quality dimension of software complexity could be developed further. Even if quality is a central concept in our master thesis, we have focused on productivity. This has meant that we have not had as much time as we had hoped to examine how software quality could be measured and controlled at ERV. There are existing automatic tools that can perform the measures of structural complexity presented in Chapter 8, *Structural Measures of Software Complexity*. These should be evaluated, together with different ways of predicting, estimating and controlling software quality.

Thirdly, ERV's organization should be studied in order to evaluate how a software measurement program should be implemented, with regard to efficiency and reliability of the measurements. This research should be aimed at exploring the prevailing methods and structures used for measuring software as well as attitudes amongst managers and system developers towards software measurements. This knowledge will be extremely helpful for an organization like ERV that are relying very much on their employees and models to get accurate data to a low cost.

There may exist more areas of research that have been overseen in this survey. Our hope is that the reader, during the study of this master thesis, has developed an understanding of what work there is to be done in the area of software metrics.

Naturally, we also hope that the reader has found something that has addressed his/her interests and maybe this report has stimulated someone to dig deeper into the subject of software metrics.

References

Books

Bache, R. & Bazzana, G. (1994). *Software Metrics for Product Assessment*. London: McGraw-Hill.

Basili, V.R. (1980*). Tutorial on Models and Metrics for Software Management and Engineering*. Los Alamitos: IEEE Computer Society Press.

Boehm, B.W. (1981). *Software Engineering Economics*. Englewood Cliffs, N.J.: Prentice Hall.

Boehm, B.W., Brown, J.R., & Kaspar, J.R. (1978). *Characteristics of Software Quality*. Amsterdam: TRW Series of Software Technology.

Brooks, F.P. (1995). *The Mythical Man-Month: Essays on Software Engineering* (2nd ed) (originally published in 1975). Reading: Addison Wesley.

Burr, A. & Owen, M. (1996). *Statistical Methods for Software Quality - Using Metrics for Process Improvement*. London: International Thomson Computer Press.

Conte, S.D., Shen, V.Y., & Dunsmore, H.E. (1986). *Software Engineering Metrics and Models*. Menlo Park: Benjamin Cummins Publishing

Fenton, N. E. & Pfleeger, S. L. (1996*). Software Metrics - A Rigorous & Practical Approach*. London: International Thomson Computer Press.

Fenton, N.E., Iizuka, Y., & Whitty, R.W. (eds) (1995). *Software Quality Assurance and Measurement*: A Worldwide Perspective. London: International Thomson Computer Press.

Garmus, D. & Herron, D. (1996). *Measuring The Software Process*. Upper Saddle River: Yourdon Press.

Gilb, T. (1988). *Principles of Software Engineering Management*. Reading: Addison-Wesley.

Goodman, P. (1993). *Practical implementation of software metrics*. London: McGraw-Hill.

Grady, R. B. (1992*). Practical software metrics for project management and process improvement*. New Jersey: Prentice Hall.

Halstead, M. (1977). *Elements of Software Science*. Amsterdam: Elsevier.

Jones, C. (1996). *Applied Software Measurement – Assuring Productivity and Quality*. New York: McGraw-Hill.

Möller, K.H. & Paulish, D.J. (1993*). Software Metrics - A Practitioner's Guide to Improved Product Development*. London: Chapman & Hall.

Ohlsson, N. (1996). *Software Quality Engineering by Early Identification of Fault-Prone Modules* (Linköping Studies in Science and Technology, Thesis No 575). Linköping: Linköping University, Department of Computer and Information Science.

Pfleeger, S.L. (1991). *Software Engineering: The Production of Quality Software* (2nd ed). New York: Macmillan.

Putnam, L. (1980). *Tutorial on Software Cost Estimating and Life Cycle Control: Getting the Software Numbers*. Los Alamitos: Compute Society Press.

Quirk, W.J. (1985). *Verification and Validation of Real-Time Software*. Berlin: Springer Verlag.

Symons, C.R. (1991). *Software Sizing and Estimating - MkII FPA* (Function Point Analysis). West Sussex: John Wiley & Sons Ltd.

Treble, S. & Douglas, N. (1995). *Sizing and Estimating Software in Practice - Making*

MkII Function Points work. Berkshire: McGraw-Hill.
Yourdon, E. & Constantine, L.L. (1979). *Structured Design*. Englewood Cliffs: Prentice Hall.
Zuse, H. (1991). *Software complexity – measures and methods*. Berlin: Walter de Gruyter & Co.

Periodicals, Reports & Encyclopaedias

Adams, E. (1984). Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1), 2-14.
Albrecht, A.J. & Gaffney, J. (1983). Software function, source lines of code and development effort prediction. *IEEE Transactions on Software Engineering*, SE-9(6), 639-648.
Basili, V.R. & Weiss, D.M. (1984). A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, SE-10 (november), No. 6, 728-738.
Behrens, C.A. (1983). Measuring the productivity of computer systems development activities with function points. *IEEE Transactions on Software Engineering*, SE-9(6), 648-652.
Coté, V., Bourque, P., Oligny, S., & Rivard, N. (1988). Software Metrics: An Overview of Recent Results. *The Journal of Systems and Software*, 8, 121-131.
Curtis, B., Sheppard, B., & Milliman, P. (1979). Third time charm: Stronger prediction of programmer performance by software complexity metrics. In *Proceedings of the 4th International Conference on Software Engineering* (pp. 356-360). Munich, Germany.
Davis, J.S. and LeBlanc, R. (1988). A study of the applicability of complexity measures. *IEEE Transaction on Software Engineering*, 14(9), 1366-1372.
Grover, P.S. & Gill, N.S. (1995). Composite Complexity Measures (CCM). In Lee, M., Barta, B.-Z., & Juliff, P. (Eds.), *Software Quality and Productivity - Theory, practice, education and training* (pp. 279-283). London: Chapman & Hall.
Halstead, M.H. (1979). Advances in Software Science. In *Advances in Computers, vol. 18*. New York: Academic Press.
Hausen, H.-L. (1989). Yet another model of software quality and productivity. In Littlewood, B. (ed.), *Measurement for Software Control and Assurance* (pp. 131-145). London: Elsevier.
Henry, S. & Kafura, D. (1981). Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5), 510-518.
Henry, S. & Kafura, D. (1984). The evaluation of software systems structure using quantitative software metrics. *Software Practice and Experience*, 14(6) (june), 561-573.
Henry, S., Kafura, D. & Harris, K. (1981). On the relationship among three software metrics. *SIGMETRICS Performance Evaluation Review*, 10 (spring), 81-88.
Institute of Electrical and Electronics Engineers (IEEE) (1993). IEEE Std 1061-1992 – Standard for a Software Quality Metrics Methodology. In *IEEE Standards Collection – Software engineering*. New York: The Institute of Electrical and Electronics Engineers, Inc.
International Standards Organisation (ISO) (1991). *Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their Use* (ISO/IEC IS 9126). Geneve: ISO/IEC.

McCabe, T. (1976). A software complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4), 308-320.

Möller, K.H. (1988). Increasing of Software Quality by Objectives and Residual Fault Prognosis. *First European Seminar on Software Quality*, Apr. 1988.

The New Encyclopaedia Brittanica. (1991). Chicago: Encyclopaedia Brittanica, Inc.

Pan, S. & Dromey, R.G. (1995). Using Strongest Postconditions To Improve Software Quality. In Lee, M., Barta, B.-Z., & Juliff, P. (Eds.). *Software Quality and Productivity - Theory, practice, education and training* (pp. 235-240). London: Chapman & Hall.

Rapps, S. & Weyuker, E.J. (1985). Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4), 367-375.

Shen, V.Y., Yu, T., Thebaut, S.M. & Paulsen, L.R. (1985). Identifying error-prone software – an empirical study. *IEEE Transactions on Software Engineering*, SE-11, No. 4, 317-323.

Shepperd, M.J. & Ince, D.C. (1990). The use of metrics in the early detection of design errors. *Proceedings of the European Software Engineering Conference '90* (pp. 67-85). Warwick, United Kingdom.

Tan, M. & Yap, C.Y. (1995). Impact of Organisational Maturity on Software Quality. In Lee, M., Barta, B.-Z., & Juliff, P. (Eds.), *Software Quality and Productivity - Theory, practice, education and training* (pp. 231-234). London: Chapman & Hall.

Tervonen, I. (1995). A Unifying Model for Software Quality Engineering. In Lee, M., Barta, B.-Z., & Juliff, P. (Eds.), *Software Quality and Productivity - Theory, practice, education and training* (pp. 200-205). London: Chapman & Hall.

Electronic documents

Abran, A., Desharnais, J.-M., Maya, M., St-Pierre, D., & Bourque, P. (1998). *Design of Functional Size Measurement for Real-Time Software.* Montréal, Université du Québec à Montréal [www document].. URL http://www.lrgl.uqam.ca/publications/pdf/407.pdf

Bohem, R. (1997). *Function Point FAQ*. Metuchen, USA, Software Composition Technologies, Inc [www document].. URL http://ourworld.compuserve.com/homepage/softcomp/

Desharnais, J.-M. & Morris, P. (1996). Validation Process in Software Engineering: an Example with Function Points. In *Forum on Software Engineering Standards (SES'96), Montreal* [www document].. URL http://www.lrgl.uqam.ca/publications/pdf/104.pdf

Ericsson Mobile Data Design AB (ERV) (1999a, April 13). *Information material about ERV* [www document]. URL http://www.erv.ericsson.se/frames/prod.html & http://www.erv.ericsson.se/frames/comp.html

Ericsson Mobile Data Design AB (ERV) (1999b, April 13*). Use cases for mobile data systems* [www document]. URL http://www.erv.ericsson.se/prod/users.html.

Introduction to Function Point Analysis (1998). *GIFPA*, Issue 2, summer [www document]. URL http://www.gifpa.co.uk/news/News2Web.pdf

Jones, C. (1999, April 27). *Programming Languages Table* [www document]. URL http://www.spr.com/library/0langtbl.htm

McCarty, W.B. (1999, March 26). An Empirical Study of Software Complexity Metrics for Prediction of Change-Prone Modules - A Dissertation Presented to the Faculty of The Claremont Graduate School [www document]. URL http://www.apu.edu/~bmccarty/dissertation/index.htm

Software Metrics - why bother?.(1998). *GIFPA*, Issue 1, spring [www document]. URL http://www.gifpa.co.uk/news/Issue1_ed2.pdf

St-Pierre, D., Maya, M., Abran, A., Desharnais, J.-M. & Oligny, S. (1997a). *Full Function Points: Counting Practices Manual*. Montréal, Université du Québec à Montréal [www document]. URL http://www.lrgl.uqam.ca/publications/pdf/267.pdf

St-Pierre, D., Maya, M., Abran, A., Desharnais, J.-M. & Oligny, S. (1997b). *Measuring the functional size of real-time software*. Montréal, Université du Québec à Montréal [www document]. URL http://www.lrgl.uqam.ca/publications/pdf/330.pdf

Unpublished electronic documents

Ericsson Mobile Data Design AB (ERV) (1999c). Välkomna till Ericsson Mobile Data Design AB (Presentation material on ERV for external use).

Ericsson Mobile Data Design AB (ERV) (1999d). Presentation of Darwin at ERV's intranet.

Personal communication

Almén, P. (1999, March, 16). Phone conversation.

Desharnais, J.-M. (desharnais.jean-marc@uqam.ca). (1999, March, 3; 1999, March, 15; 1999, March, 31; 1999, April, 23). Re: Tools for counting Full Function Points & Questions about FFP counting. E-mail correspondence. (claes.sandros@erv.ericsson.se).

Timmerås, M. (1999, March, 22). Oral information about Darwin and measurements at ERV.

Öhlin, M. (1999, March, 16). Phone conversation.

Appendix

A. Wordlist

4GL	4th Generation Language
ATM	Automatic Teller Machine
CDPD	Cellular Digital Packet Data (American standard for digital mobile telephony)
CPU	Central Processing Unit
DET	Data Element Types
DI	Degree of Influence
ECE	External Control Entry
ECG	Electrocardiogram
ECX	External Control Exit
EEG	Electroencephalogram
EI	External Input
EIF	External Interface File
EO	External Output
EQ	External Inquiry
ERV	Ericsson Mobile Data Design AB
FFP	Full Function Points
FPA	Function Point Analysis
FTR	File Types Referenced
GPRS	General Packet Radio Service (2nd generation European standard for digital mobile telephony)
GSC	General System Characteristics
I/O	Input/Output
ICR	Internal Control Read
ICW	Internal Control Write
IEEE	Institute of Electrical and Electronics Engineers
IFPUG	International Function Points User Group
ILF	Internal Logical File
IS	Implementation Specification
ISO	International Standards Organisation
IWD	Interwork Description
LOC	Lines of Code
MIS	Management Information System
MS	Milestone
PDC	Personal Digital Cellular (Japanese standard for digital mobile telephony)
RCG	Read-only Control Group
RET	Record Element Types
SPR	Software Productivity Research
TDI	Total Degree of Influence
TG	Tollgate
UCG	Updated Control Group
UMTS	Universal Mobile Telephone System (3rd generation European standard for digital mobile telephony)
VAF	Value Adjustment Factor

B. Function Points – definitions

Data Function Types

Internal Logical Files:
An internal logical file (ILF) is a user identifiable group of logically related data or control information maintained through an elementary process of the application within the boundary of the application.

External Interface Files:
An external interface file (EIF) is a user identifiable group of logically related data or control information referenced by the application but maintained within the boundary of a different application.

Transactional Function Types

External Inputs:
An external input (EI) is an elementary process of the application, which processes data or control information that enters from outside the boundary of the application

External Outputs:
An external output (EO) is an elementary process of the application, which generates data or control information that exits the boundary of the application.

External Inquiries:
An external inquiry (EQ) is an elementary process of the application, which is made up of an input-output combination that results in data retrieval. The input side is the control information, which spells out the request, specifying what and/or how data is to be retrieved. The output side contains no derived data. No ILF is maintained during processing.

Other definitions

Data element types (DETs):
DETs are unique user recognizable, nonrecursive fields/attributes, including foreign key attributes, maintained on the ILF or EIF.

Record element types (RETs):
RETs are user recognizable subgroups (optional or mandatory) of data elements contained within an ILF or EIF. Subgroups are typically represented in an entity relationship diagram as entity subtypes or attributive entities, commonly called parent-child relationships. (The user has the option of using one or none of the optional subgroups during an elementary process that adds or creates an instance of the data; the user must use at least one of the mandatory subgroups.)

File types referenced (FTRs):
FTRs or more simply files referenced totals the number of internal logical files (ILFs) maintained, read, or referenced and the external interface files read or referenced by the EI transaction.

(Garmus and Herron, 1996)

C. Full Function Points Counting procedure and rules

In Chapter 9, *Algorithmic Complexity and Size Measures*, we have spelled out the general characteristics of the Full Function Point (FFP) method and the overall counting procedure. This procedure includes the following steps:

1. Determine the type of function point count
2. Identify the counting boundary
3. Determine the unadjusted function point count
4. Count data function types
5. Count transactional function types
6. Determine the value adjustment factor
7. Calculate the final adjusted function point calculation

For FFP, steps 1, 2, 4 and 5 are exactly the same as in Function Point Analysis (FPA), and they are described in Chapter 9, *Algorithmic Complexity and Size Measures*. Step 3 is divided into Management Function Types and Control Function Types, according to this scheme:

1. Identify groups of data
   - If Management data: Count Management Data Function Types, according to FPA
   - If Control Data: Count Control Data Function Types, according to FFP
2. Identify processes
   - If Management process: Count Management Transactional Function Types, according to FPA
   - If Control process: Count Control Transactional Function Types, according to FFP

I. Identify groups of data

This step consists of identifying the groups of data that could represent the functionality provided to the user by the application being measured. Once the groups of data are identified, the definitions and rules associated with these function types are applied to determine whether the identified groups of data are counted as FFP function types.

I:I Definitions
Group of data: Data identified and grouped together based on the functional perspective.
Management data: Data used by the application to support users in managing information, particularly business and administrative information.
Control data: Data used by the application to control, directly or indirectly, the behavior of an application or a mechanical device.

I:II Identification procedure
The procedure to identify group of data candidate is the following:

1. Look for groups of data identifiable from a functional perspective, i.e. a point of view of the functionality delivered by the application; it excludes technical and implementation considerations.
2. Determine if the group of data is a management group of data or a control group of data using the previous definitions. For management groups of data, existing FPA procedures and rules should be applied. For Control Groups of Data, the following procedures and rules should be applied.

II. Count control data function types

II:I Definitions
Updated Control Group (UCG): A UCG is a group of control data updated by the application being counted. It is identified from a functional perspective. The control data live for more than one transaction.
Read-only Control Group (RCG): An RCG is a group of control data used, but not updated, by the application being counted. It is identified from a functional perspective. The control data live for more than one transaction.
Functional perspective: Point of view of the functionality delivered by the application; it excludes technical and implementation considerations.
Transaction: All processing associated with an occurrence of an external trigger.

II:II Counting procedure
For each group of data identified in the previous step as a group of control data:
1. Determine if the group of control data is a UCG or an RCG using the definitions and rules.
2. Determine the UCG or RCG contribution (point assignment) to the unadjusted function point count.

II:III Identification rules
UCG identification rules:
The group is either a logically related group of data or a single occurrence group of data.
The group of data is **updated** within the application boundary.
The group of data lives for more than one transaction.
The group of data identified has not been counted as an RCG, ILF or EIF for the application.

All the previous counting rules must be applied from a functional perspective and they are all mandatory for the identification of a UCG.

RCG identification rules:
The group is either a logically related group of data or a single occurrence group of data.
The group of data is **not updated** within the application boundary.
The group of data is referenced by the application being counted.
The group of data lives for more than one transaction.
The group of data identified has not been counted as an UCG, ILF or EIF for the application.

All the previous counting rules must be applied from a functional perspective and they are all mandatory for the identification of a RCG.

II:IV Point assignment
The number of points assigned to UCGs and RCGs depends on the kind of control group of data (single or multiple occurrences). Since multiple occurrence groups of data have the same structure as ILFs and EIFs in FPA, they are counted in exactly the same way as these two FPA function types, that is, using their number of DETs and RETs and the corresponding complexity matrix.

For single occurrence groups of data, the number of points depends only on DETs. Once the number of DETs is determined using the same rules as for ILFs and EIFs, the number of points is calculated using the following formulas:

UCG: ((number of DETs/5) + 5)
RCG: (number of DETs/5)

These formulas are designed to keep the size of single occurrence groups of data as aligned as possible with the size of ILFs and EIFs of FPA.

A single occurrence UCG comprises all single control updated values (from a functional perspective) of the application being measured. Since it contains all single values of the application, there can be only one of them in an application. Consequently, an application can have more than one multiple occurrence UCG, but only one single occurrence UCG. The same goes for single occurrence RCGs.

In typical real-time applications, the number of such single values varies from a few up to hundreds. That is why a formula is used rather than a 3-level table like the standard FPA technique. It allows FFP to consider a large range of single occurrence groups of data.

III. Identify processes

Once the data function points (management and control) have been counted, the transactional function types are identified. Transactional function types represent the functionality provided to the user for the processing of data by an application. Therefore, to identify transactional function types we have to identify the processes of the application first.

III:I Definition
Control process: Process that controls, directly or indirectly, the behavior of an application or a mechanical device.

III:II Identification procedure
The procedure for identifying processes is the following:
1. Look for the different processes of the application from a functional perspective.
2. Determine if the process is a management process or a control process using the following definitions:
    - Management process: Process the purpose of which is to support the user in managing information, particularly business and administrative information.

- Control process: Process that controls, directly or indirectly, the behavior of an application or a mechanical device.

If the process is a control process, apply the definition and rules of the four new control transactional function types. If the process is a management process, apply the definition and rules of the current FPA transactional functions.

IV. Count control transactional function types

IV:I Definitions

External Control Entry (ECE): An ECE is a unique sub-process. It is identified from a functional perspective. An ECE processes control data coming from outside the application's boundary. It is the lowest level of decomposition of a process acting on one group of data. Consequently, if a process enters two groups of data, there are at least 2 ECEs. ECEs exclude the updating of data, a functionality that is covered by another Control Function Type (Internal Control Write).

External Control Exit (ECX): An ECX is a unique sub-process. It is identified from a functional perspective. An ECX processes control data going outside the application boundary. It is the lowest level of decomposition of a process acting on one group of data. Consequently, if a process exits two groups of data, there are at least 2 ECXs. ECXs exclude the reading of data, a functionality that is covered by another Control Function Type (Internal Control Read).

Internal Control Read (ICR): An ICR is a unique sub-process. It is identified from a functional perspective. An ICR reads control data. It is the lowest level of decomposition of a process acting on one group of data. Consequently, if a process reads two groups of data, there are at least 2 ICRs.

Internal Control Write (ICW): An ICW is a unique sub-process. It is identified from a functional perspective. An ICW writes control data. It is the lowest level of decomposition of a process acting on one group of data. Consequently, if a process writes on two groups of data, there are at least 2 ICWs.

User: Human beings, applications or mechanical devices, which interact with the application measured.

IV:II Counting procedure

Once a process has been identified as being a control process, the following steps must be performed:

1. Identify all functional (not technical) sub-processes of the control process.
2. Identify each sub-process as being an ECE, ECX, ICR or ICW according to the definition and rules.
3. Determine the ECE, ECX, ICR or ICW contribution (point assignment) to the unadjusted function point count.

Steps for identifying sub-processes:

a) According to the logical execution order of the sub-processes within the process, identify the first sub-process that receives, exits, reads or writes a group of control data.
b) Apply the relevant ECE, ECX, ICR or ICW set of rules.
c) Determine the ECE, ECX, ICR or ICW contribution (point assignment) to the unadjusted function point count.

d) Again according to the execution order, identify the next sub-process that enters, exits, reads or writes a group of control data. There might be more than one "next sub-process" (e.g. an "IF" statement with two options). In this case, all paths have to be explored if there is potential for new sub-processes to be identified.

Repeat steps 2 to 4 until all sub-processes of the processes are identified.
At the end of the cycle, remove all the duplicated sub-processes (same processing and same DETs).

Note: If the same sub-process is associated with different control processes, it can be counted more than once. Processing includes not only the entry, exit, reading or writing of data, but other types of processing as well (calculation, filtering, comparisons, etc.) associated with the identified sub-process.

IV:III Identification rules
ECE identification rules:
The sub-process receives a group of control data from outside the application boundary.
The sub-process receives only one group of data. If more than one different group of data is received, count one ECE for each group of data.
The sub-process does not exit, read or write data.
The sub-process is unique, that is, the processing and data elements identified are different from other ECEs associated with the same process.

Note 1: Clock triggers are considered external. Therefore, an event that takes place every 3 seconds is counted as an ECE with 1 DET, for example. However, the process that generates that event periodically is ignored.

Note 2: Unless a special process is necessary, reading internal time is not counted. For example, when a process writes a time stamp, no ICR is counted for obtaining the internal clock value.

All the previous counting rules must be applied from a functional perspective and they are all mandatory for the identification of an ECE.

ECX identification rules:
The sub-process sends control data external to the application's boundary.
The sub-process sends only one group of data. If more than one different group of data is sent outside the application's boundary, count one ECX for each group of data.
The sub-process does not receive, read or write data.
The sub-process is unique, that is, the processing and data elements identified are different from other ECXs associated with the same process.

Note: All messages without user data (e.g. confirmation and error) are counted as one ECX. The number of DETs is the number of different types of messages.

All the previous counting rules must be applied from a functional perspective and they are all mandatory for the identification of an ECX.

ICR identification rules:
The sub-process reads a group of control data.

The sub-process reads only one group of data. If more than one different group of data is read, count one ICR for each group of data.
The sub-process does not receive, exit or write data.
The sub-process is unique, that is, the processing and data elements identified are different from other ICRs associated with the same process.

All the previous counting rules must be applied from a functional perspective and they are all mandatory for the identification of an ICR.

ICW identification rules:
The sub-process writes a group of control data.
The sub-process writes only one group of data. If more than one different group of data is written, count one ICW for each group of data.
The sub-process does not receive, exit or read data.
The sub-process is unique, that is, the processing and data elements identified are different from other ICWs associated with the same process.

All the previous counting rules must be applied from a functional perspective and they are all mandatory for the identification of an ICW.

IV:IV Point assignment
The number of points assigned to control transactional functions (ECE, ECX, ICW and ICR) depends on the number of DETs. The following rules apply when counting DETs:

For an **ECE** and an **ECX**:
Count one DET for each unique user recognizable, nonrecursive field that crosses the boundary of the application.
For an **ICR**:
Count one DET for each unique user recognizable, nonrecursive field that is read from an ILF, EIF; UCG or RCG, including keys.
For an **ICW**:
Count one DET for each unique user recognizable, nonrecursive field that is written in an ILF or UCG, including keys.

The number of points assigned to Control Transactional Function Types (ECE, ECX, ICW and ICR) depends on the number of DETs. Once the number of DETs is determined, the following table is used to translate DETs into points:

DETs:	1 to 19 DETs	20 to 50 DETs	51+ DETs
Points:	1	2	3

These range boundaries (1 to 19, 20 to 50, 51+) were chosen in order to bring the size of Control Transactional Function Types in as close alignment as possible with FPA.

(St-Pierre, Maya, Abran, Desharnais, & Oligny, 1997a)

D. Sample from a Language Level Table

What is a language level?
As language levels go up, fewer statements to code one Function Point are required. For example, COBOL is a level 3 and requires about 105 statements per Function Point. The numeric levels of various languages provide a convenient shortcut for converting size from one language to another. For example, if an application requires 1000 non-commentary COBOL (level 3), then it would take only 500 statements in a level 6 language (such as C++) and only 250 statements in a level 12 language (such as OBJECTIVE C). As you can see, the average number of statements required is proportional to the levels of the various languages.

What is the basis for language levels?
The languages and levels in the table are gathered in four ways.
- Counting Function Points and Source Code
- Counting Source Code
- Inspecting Source Code
- Researching Languages

List of programming languages
As of 1996, there were more than 500 languages and major dialects of languages available to software practitioners. The table lists a sample of them.

Language	Level	Average source statements per Function Point
1st Generation default	1.00	320
2nd Generation default	3.00	107
3rd Generation default	4.00	80
4th Generation default	16.00	20
5th Generation default	70.00	5
Access	8.50	38
Ada 83	4.50	71
Ada 95	6.50	49
ANSI BASIC	5.00	64
ANSI SQL	25.00	13
Assembly (Basic)	1.00	320
Assembly (Macro)	1.50	213
BASIC	3.00	107
C	2.50	128
C++	6.00	53
COBOL	3.00	107
Crystal Reports	16.00	20
dBase IV	9.00	36
DELPHI	11.00	29
DOS Batch Files	2.50	128
EIFFEL	15.00	21
Erlang	8.00	40

EXCEL 5	57.00	6
FileMaker Pro	9.00	36
FORTRAN	3.00	107
HTML 3.0	22.00	15
JAVA	6.00	53
LISP	5.00	64
LOTUS 123 DOS	50.00	6
Machine language	0.50	640
Macro assembly	1.50	213
MAESTRO	20.00	16
Microsoft C	2.50	128
MS C ++ V. 7	6.00	53
MS Compiled BASIC	3.50	91
Natural language	0.10	3200
Non-procedural default	9.00	36
Object-Oriented default	11.00	29
ORACLE	8.00	40
PASCAL	3.50	91
PERL	15.00	21
Program Generator default	20.00	16
PROLOG	5.00	64
QUICK BASIC 3	5.50	58
RPG III	5.75	56
SMALLTALK/V	15.00	21
Spreadsheet default	50.00	6
SQL-Windows	27.00	12
SYBASE	8.00	40
Symantec C++	11.00	29
Turbo C	2.50	128
Turbo PASCAL >5	6.50	49
UNIX Shell Scripts	15.00	21
Visual Basic 5	11.00	29
Visual C++	9.50	34

On-going research of languages
The relationship between source code statements and Function Points has only been subject to research for a few years, so the margin of error in the table can be quite high. Even so, the method is useful enough so publication of a preliminary table may be helpful in filling in the gaps and correcting the errors.

A complete and reliable industry-wide study of languages and their levels is of necessity a large multi-year project. A reasonable sampling of applications and languages would require data from 5000 projects, assuming 10 projects in each language or dialect. The organizing principle used in this research is basically sound and the construction of a periodic table of languages is potentially as useful to software engineering as the periodic table of the elements has been to chemical engineering and to physics.

(Jones, 1999, April 27)

E. Suggestions to Ericsson Mobile Data Design

Short-term suggestions

- Implement two sets of measurements; one after concluding the code phase of the development (expected performance), and one after in the end of the test phase (real performance).
- Expected performance should be based on measures of implemented FFP, man time and McCabe's cyclomatic number. When real performance is measured McCabe's cyclomatic number is exchanged for the actual number of known defects. Otherwise the same measures, updated of course, are being used for calculating real performance as for expected performance.
- A measurement group should be established for each project. This group should consist of one or two experts on the FFP method, and a couple of experts on the software being counted, i.e. system developers from the project.
- The system documentation should be extended to include information about identified elements for the FFP method (i.e. UCGs, RCGs, ECEs, ECXs, ICRs and ICWs). The individual system developer should collect this information.

Long-term suggestions

- A framework for collecting productivity and quality data should be created.
- A training program for the "experts" and the system developers should be built up.
- ERV should establish a co-operation with the institutions developing and improving the FFP method.
- The progress for the FPA and FFP methods should be followed closely, especially the standardization work at IFPUG.

Further research

- The possibilities of automating the implementation of the FFP (and FPA) method(s).
- A more profound analysis of how software quality can be developed and controlled.
- ERV's organization could be analyzed for the purpose of finding the most efficient and reliable way to implement a software measurement program.