Predicting Defects in Code

Finding ways to improve software development in a telecommunications system using software metrics

JOHAN KANNALA

Master’s Degree Project
Stockholm, Sweden 2005

TRITA-NA-E05082
Predicting Defects in Code

Finding ways to improve software development in a telecommunications system using software metrics

JOHAN KANNALA

TRITA-NA-E05082

Master’s Thesis in Computer Science (20 credits)
at the School of Engineering Physics,
Royal Institute of Technology year 2005
Supervisor at Nada was Linda Kann
Examiner was Stefan Arnborg
Predicting defects in code

Finding ways to improve software development in a telecommunications system using software metrics

Abstract

This report describes the findings and experiences of a master’s project investigating the relation between fault-proneness and cyclomatic complexity of code in a large telecommunications platform.

The ability to estimate the fault-proneness of code from code complexity would be valuable, among other things, for guiding test activities and for preventing defects through design rules.

The cyclomatic complexity and the number of trouble reports in a number of load modules (executable units of code) have been measured, and different theories regarding the effect of complexity on fault-proneness have been investigated. Unfortunately, the results proved to be inconclusive.

The inability to arrive at an acceptable conclusion was deemed primarily to be because of limitations in the defect data. Because of this, numerous recommendations are given on how to improve the trouble reporting system and how to draw better conclusions from defect data in general.
Att förutsäga fel i kod

Sätt att förbättra utvecklingen av ett telekommunikationssystem med hjälp av mjukvarumetriker

Sammanfattning

Den här rapporten beskriver ett examensarbete som har undersökt ett eventuellt samband mellan cyklomatisk komplexitet och felbenägenhet i källkoden till ett telekommunikationssystem.

Förmågan att kunna uppskatta felbenägenheten hos kod från dess komplexitet skulle vara värdefull, bland annat för att styra testningsaktiviteter och för att begränsa antalet fel genom designregler.

Den cyklomatiska komplexiteten och antalet felrapporter hos ett antal laddmoduler (exekverbara kodenheter) har studerats, och olika teorier om sambandet mellan komplexitet och felbenägenhet har prövats. Tyvärr gick det inte att komma till någon definitiv slutsats om användbarheten av cyklomatisk komplexitet.

Svårigheterna med att dra slutsatser om kodkomplexitet bedömdes huvudsakli- gen bero på brister i tillgängliga feldata. Därför ges också ett antal rekommendationer för hur felrapporteringen skulle kunna förbättras och hur man allmänt ska kunna dra bättre slutsatser om produktkvalité från insamlade feldata.
Credits and acknowledgements

During the course of this project, I have met with and interviewed a large number of people in different positions and with different responsibilities. A little to my surprise, I must say, the vast majority of them really took their time to listen and engagedly discuss with me about my inquiries. This has provided me with many useful ideas and a much better understanding of the problems at hand. Since I do not want to add several more pages to this already voluminous report and risk forgetting to mention someone, I will just say: You know who you are, and I’m very grateful.

There are, however, a number of people I’d still like to single out and mention in particular.

First of all, I want to thank my supervisor, Yong Hui Jin, for being supportive and helpful beyond both duty and kindness. I have also been fortunate enough to have an encouraging supervisor at KTH, Linda Kann, who always seems to be on the lookout on behalf of my best interest.

I would like to thank Carl Engblom and Ulf Lindström who, together with Yong Hui and me, made up the “governing council” of this project. Our brainstorming sessions every Monday morning during the execution of the project provided me with many stimulating ideas and guidance through an otherwise treacherous domain. I want to thank Sigrid Eldh for her experienced advice in our interesting discussions on software quality. I am also grateful for the great interest and support in my project from Tommy Sundgren and Sven-Erik Berggren. They have painstakingly read my report and provided me with valuable feedback.

I would like to thank Henrik Thane for the permission to use the diagram in figure 3.1 from his presentation Computerized Control Systems: Complexity and Risk.

Finally, I want to thank Jan Enger, Senior Lecturer at the KTH department of statistics, for patiently receiving and discussing with me on several occasions, despite not being related to this project in any other way than once having been my lecturer in statistics.
Contents

1 Introduction ................................. 1
  1.1 The background of the master’s project ............... 1
  1.2 Overview of the document .......................... 2

2 Acronyms, glossary and definition of terms .......... 3
  2.1 Acronyms ..................................... 3
  2.2 Glossary and definitions .......................... 4

3 An overview of software metrics .................. 7
  3.1 What is a metric? .............................. 7
  3.2 Utility of metrics ............................... 7
    3.2.1 Evaluation ................................ 8
    3.2.2 Prediction ................................ 9
    3.2.3 Control ................................... 9
  3.3 Types of metrics ............................ 9
    3.3.1 Size metrics ............................. 10
    3.3.2 Complexity metrics ....................... 11
    3.3.3 Quality metrics ........................... 12
  3.4 What to measure ............................... 13
  3.5 Studying defects .............................. 14
    3.5.1 Software problem terminlogy ................. 15
    3.5.2 The relation between faults and failures .... 16
    3.5.3 How to reduce the number of faults .......... 17
    3.5.4 Defect detection techniques ................ 18
    3.5.5 Collecting defect data ..................... 19
    3.5.6 Difficulties with defect data ............... 23
  3.6 McCabe’s cyclomatic complexity metric .......... 27
    3.6.1 Flow graphs ................................ 28
    3.6.2 Calculation of the metric .................. 28
    3.6.3 Usage of the metric ....................... 30
    3.6.4 Limitations and problems ................... 32

4 Description of the project ....................... 36
  4.1 Presentation of The System ...................... 36
4.1.1 Structure of The System ........................................ 36
4.1.2 The new process and product structure ...................... 39
4.2 The metrics tool: C and C++ Code Counter ..................... 42
  4.2.1 Capabilities of CCCC .................................. 43
  4.2.2 Limitations and problems with CCCC ...................... 44
  4.2.3 The implications of the problems for this project ......... 44
4.3 The TR-system .................................................. 45
  4.3.1 Capabilities of the TR-system ........................... 45
  4.3.2 Limitations of the TR-system ............................ 47
4.4 The quality measure ............................................ 48
  4.4.1 Defect density ......................................... 48
  4.4.2 Modification rate ....................................... 49
  4.4.3 Maintenance difficulty ................................... 49
4.5 The complexity measure ......................................... 50
  4.5.1 Cyclomatic complexity vs. interaction complexity ....... 50
  4.5.2 Complexity across functions ............................ 51
  4.5.3 Complexity over time .................................... 51
4.6 Measures to be used in the investigation ....................... 51
  4.6.1 TR counts as a measure of module quality ............... 51
  4.6.2 Cyclomatic complexity as a measure of module complexity ... 52
  4.6.3 Lines of function code as a measure of module size ...... 52
4.7 The load modules chosen for analysis .......................... 53
4.8 Performing the measurements .................................... 54
  4.8.1 Measuring complexity .................................... 54
  4.8.2 Measuring defects ....................................... 54
  4.8.3 Assembling the data ..................................... 55

5 Results of measurements ........................................... 56
  5.1 The defect counts .......................................... 56
5.2 The complexity values ......................................... 58
  5.2.1 The complexity distribution ............................. 58
  5.2.2 The complexity density distribution ..................... 59
  5.2.3 What about the really high complexity values? ......... 61
5.3 The defect and complexity relation ............................ 63
  5.3.1 Defect density and complexity .......................... 63
  5.3.2 Defect density and complexity density .................. 68
  5.3.3 Defects and complexity mass ............................ 69
5.4 Other analyses ................................................ 73
  5.4.1 Defects and size ....................................... 73
  5.4.2 Complexity and size .................................... 77

6 Discussion of results ............................................. 79
  6.1 The relation between complexity and defects ............... 79
  6.1.1 Complexity ............................................ 79
6.1.2 Complexity density ............................................. 80
6.1.3 Complexity mass .............................................. 80
6.1.4 Conclusion ..................................................... 80
6.2 The analysis ........................................................ 81
6.2.1 Uncertainties in the analysis ................................. 81
6.2.2 On the usage of transforms of complexity .................. 82
6.3 The TR statistics .................................................. 83
6.3.1 Uncertainties in the TR statistics .............................. 84
6.4 The complexity measure ......................................... 87
6.4.1 Uncertainties in the complexity measure ................. 87
6.5 How to specify the location of a defect ....................... 90
6.5.1 The Load Module ............................................. 90
6.5.2 The Software Block .......................................... 91
6.5.3 The Software Unit ........................................... 91
6.5.4 The new Block ............................................... 92
6.6 What is really software quality? ............................... 94
6.6.1 Intrinsic quality or resulting quality ....................... 94

7 Recommendations .................................................. 96
7.1 Defect data ....................................................... 96
7.1.1 Tracking the faults .......................................... 96
7.1.2 Detecting the faults ......................................... 99
7.1.3 Classifying the faults ....................................... 100
7.1.4 Validity of defect data and conclusions ................. 101
7.2 Project data ..................................................... 103
7.3 Cyclomatic complexity ......................................... 105
7.4 Other suggestions for improvements ......................... 106
7.4.1 Establish a company baseline .............................. 106
7.4.2 Automate measurements .................................... 107
7.4.3 Distinguish between faults, failures and changes ...... 107
7.4.4 Extend defect tracking to other design products ....... 108
7.4.5 Software design quality ranks ............................. 109
7.5 How to make it work ............................................. 109
7.5.1 Who does what and when? ................................. 111
7.5.2 Implementing the changes .................................. 112
7.6 Lists of recommendations ..................................... 113
7.6.1 High priority list ............................................ 114
7.6.2 Medium priority list ....................................... 114
7.6.3 Low priority list ............................................ 115

8 Going from here .................................................... 116
8.1 Suggestions for future work ................................... 116
8.2 The vision ......................................................... 118
8.3 Final words ....................................................... 119
A On the measurement theoretic properties of cyclomatic complexity
Chapter 1

Introduction

This master’s project was undertaken at a developer of telecommunications technology, fall to spring 2003/2004. The system which has been under study is a platform for the next generation of data and telecommunication networks. This system will henceforth be referred to as “The System”.

The objective of the master’s project was to investigate the usefulness of software metrics, and particularly the McCabe’s cyclomatic complexity metric, in support of the development organisation.

1.1 The background of the master’s project

Maintenance and fault corrections consume vast amounts of resources for software developers, with as much as up to 75 % of the total development cost being spent on debugging, testing and verification activities [12]. Since the cost of fixing a fault increases dramatically the longer the fault is allowed to remain in the software, there is much to be saved by preventing or dealing with faults as early as possible.

Software metrics have the potential to be a useful instrument when trying to prevent or predict faults. As (predominantly) objective indicators of the state of products, process or code, software metrics could provide guidance for making decisions and allocating resources, especially much so in a situation with an ever growing, increasingly unmanageable code base where human resources are scarce.

The focus of this master’s project has been on code complexity metrics and their connection with fault-proneness. In particular, the cyclomatic complexity measure, as introduced by T. J. McCabe, has been under close scrutiny, but also defect data and its potential usefulness on its own account has been investigated.

A few isolated measurements of cyclomatic complexity had been done before within the organisation, but this study was, as far as I know, the first attempt to investigate the complexity of the code on a larger scale and try to relate it to fault-proneness.

The results of these measurements and, maybe more importantly, the experiences of performing them are documented in this report together with recommendations.
Enough background and theory on software measurement and error analysis is also provided for the report to serve as an introduction to the field.

1.2 Overview of the document

The document is divided into 8 chapters and 1 appendix. Not everyone will have to read all of them, and, depending on background and interest, some parts can be skipped or read when there is need for more background or detail.

Chapter 1 contains an introduction to the report and this overview of the document.

Chapter 2 is a glossary of acronyms and relevant terminology, intended to be used as a reference while reading.

Chapter 3 gives an introduction to the field of software metrics with a special emphasis on McCabe’s cyclomatic complexity and defect metrics.

Chapter 4 describes this project. It presents the platform that has been under investigation and the systems and tools which have been used. It also discusses the considerations related to measuring quality and complexity in the system, as well as the practical execution of the measurements.

Chapter 5 presents the results of the measurements with diagrams and comments.

Chapter 6 discusses the results, factors of uncertainty and conclusions on the hypotheses. It also discusses a few other topics related to measurements and quality.

Chapter 7 contains recommendations based on the experiences from this project. The suggestions are mainly for improvement of the trouble reporting process, but there are also recommendations for data gathering and quality related activities in general.

Chapter 8 is looking into the future: suggestions for future work, what the ultimate goal would be, and some closing remarks.

Appendix A has a brief discussion about the measurement theoretic properties of the cyclomatic complexity measure.
Chapter 2

Acronyms, glossary and definition of terms

The definitions below are mostly adopted from [16], [10], and [5] in the case of graph theoretic terminology.

2.1 Acronyms

ATM: Asynchronous Transfer Mode.
CCCC: C and C++ Code Counter.
CR: Configuration Record or Change Request.
DM: Delivery Module.
DP: Delivery Product.
FD: Functional Description.
GQM: Goal-Question-Metric.
I&V: Integration and Verification.
IF: Interface.
IFU: Interface Unit.
ISP: In Service Performance.
LM: Load Module.
LMC: Load Module Container.
LOC: Lines Of Code.
LOFC: Lines Of Function Code.
MTTF: Mean Time To Failure.
MVFS: Multi-Version File System.
SA: System Area.
SC: Software Configuration.
SPC: Statistical Process Control.
SQR: Software Design Quality Ranks.
SUBSYS: Subsystem.
SWB: Software Block.
SWU: Software Unit.
SYS: System.
TR: Trouble Report.
VR: Verification Record.
VS: Verification Specification.
WoW: Way of Working.

2.2 Glossary and definitions

\(v(G)\): Signifies the cyclomatic number of a graph. Is also used to mean the cyclomatic complexity of the flow graph of a program. In this report it will be used to signify the cyclomatic complexity.

Attribute: A characteristic of an item, e.g. the quality of a product or the number of typographic errors in a specification.

Black box testing: See “Functional testing”.

Block: Used as the smallest deliverable and manageable part of a subsystem. Also referred to as a new Software Block as opposed to an old Software Block.

Branch coverage: The percentage of decision outcomes exercised during branch testing of a computer program.

Branch testing: Testing designed to execute each outcome of each decision point in a computer program.
Circuit: A sequence of edges in a graph, such that the terminal end-point of a previous edge coincides with the initial end-point of the following edge, the initial point of the sequence is the same as the terminal point, and no edge appears twice in the sequence.

Code coverage: Collective name for techniques used to evaluate the quality of structural tests. E.g. “Statement coverage” and “Branch coverage”.

Configuration Record: A build-time record documenting the dependencies and files used in a build.

Cyclomatic complexity: A structural complexity measure suggested by McCabe [21]. Measures the number of linearly independent paths through a program module.

Cyclomatic number: The cardinality of a cycle basis of a graph.

Defect: Here I will use it to mean: A fault. Otherwise in the literature, the meaning of the word “defect” seems to vary.

Delivery Module: Previously a related set of delivered products making up a part of the system.

Subsequently synonymous with a “Software Configuration”.

Delivery Product: The deliverable and executable view of a system.

Error: The act of making a mistake.

Failure: A deviation from required behaviour. A manifestation of faults.

Fault: The encoding of an error: An incorrectness in code, documentation or another artifact.

Functional testing: Testing that verifies the behaviour of a system or component against the specified functional requirements.

Interface: A product entity which defines and provides access to functionality in a part of the system.

Interface Unit: The source implementation of an Interface.

Line of function code: A line of code which belongs to a function (obviously).

Load Module: A collection of data that is interpretable by the target platform.

Load Module Container: A collection of Load Modules providing a defined set of functionality as part of the system.

Mean Time To Failure: The expected amount of time before a system or component will fail.
Planar graph: A graph is said to be planar if it is possible to represent the graph on a plane in which the nodes are distinct and the edges do not cross one another.

Software Block: A logically (or otherwise) bounded part of a subsystem.

Software Configuration: A set of deliverable products which creates a specific configuration of the system. Also referred to as a “Delivery Module”.

Software metric: A method or algorithm that gives a measure of a certain attribute of a software product or process.

Software Unit: The smallest part of logically (or otherwise) bounded functionality in a Software Block.

Statement coverage: The percentage of statements exercised during statement testing of a computer program.

Statement testing: Testing designed to execute each statement of a computer program.

Structural testing: Testing techniques that take into account the internal mechanism of a system or component. E.g. “Branch testing” and “Statement testing”.

Subsystem: A logically bounded and manageable grouping of functionality in a system or system area.

System: A product with a specific technical design within a specific field of application, consisting of a complete set of functional and realisation products.

System Area: A major grouping of related functionality in a system.

Test coverage: See “Code coverage”.

White box testing: See “Structural testing”.
Chapter 3

An overview of software metrics

In this chapter I intend to give an overview of software metrics.

For readers unfamiliar with the subject, an explanation of metrics in general, of their use, and of different types of metrics is given in section 3.1, 3.2, and 3.3 respectively. The problem of deciding what is worth measuring is discussed in section 3.4. In section 3.5 and 3.6, defect metrics and McCabe’s cyclomatic complexity metric are discussed in more detail, since they were of particular interest to this project.

For a more complete introduction to software metrics than presented here, the SEI curriculum module “Software Metrics” [24] is a good start. For a rigorous treatment of the topic, the book “Software Metrics: A Rigorous & Practical Approach” [10] is recommended reading.

3.1 What is a metric?

The IEEE Standard Glossary of Software Engineering Terminology [16] defines a metric to be: “A quantitative measure of the degree to which a system, component or process possesses a given attribute”. This is in particular a different meaning of the word than the definition of a metric in the mathematical sense. In the software terminology here, in this report, the metric is the method, algorithm or procedure that describes how to obtain a measure of a certain attribute that an object has. With a measure, I am referring to the outcome or value of a particular attribute or metric. Sometimes the word measure is used both to signify the value produced, as well as the method used to produce the value, and is thus used interchangeably with the word metric.

3.2 Utility of metrics

As in other branches of engineering, different kinds of measurements and “metrics” are used to get information about the properties and attributes of products and processes, for example, capacity or reliability estimates for various products, or
productivity values for the production process. Software metrics aspire to be the equivalent for the software industry, providing information about the products or processes related to “software engineering”. This can include a wide variety of activities, such as time-reporting by employees to determine how resources are being spent, development cost estimates for the next release of a product, or completeness of the functional requirements. Insight and knowledge about the inner workings of the development process can help identify problematic areas, quantify and evaluate the impact of actions taken to rectify these problems, and generally improve the ability to control and “tune” the development process.

In general, one can say that metrics are useful in mainly three ways; three different “levels” of utility:

1. Evaluation.
2. Prediction.
3. Control.

3.2.1 Evaluation

First, metrics data from the products and processes can be used to evaluate and understand what is being done. It is, for example, interesting to know how the time and resources are being spent in a particular project, or the impact of certain practice on the quality of the product.

For example: A set of development practices are being introduced in the development process: SQR, The Software Design Quality Ranks. They consist of five levels of increasing rigour in different quality related activities during the design phase.

- Question: What is the most economical SQR level for a new component?

In order to answer this question, it is necessary to have the following information:

1. The cost of SQR activities when developing a number of new components of different SQR levels.
2. The cost of corrective activities and maintenance for these components.
3. The size of these components.

With this information it is possible to calculate the sum of the preventive and corrective costs per unit size of the components and determine which SQR level gives the lowest overall cost.

Note: This is of course an idealised example and a real evaluation of SQR would probably entail a number of other considerations.
3.2.2 Prediction

Secondly, the metrics data can be used to make predictions about the future. How much effort will it take to develop this component? How many defects do we expect to find? How reliable will the component be?

One may consider prediction to be “the next level” of utility after evaluation, since prediction, in most cases, is based on the evaluation of past projects and products. A database with historical data might, for instance, reveal that projects with certain characteristics will consume proportionally more resources than other projects, or that certain products have a tendency to be more fault-prone.

If we assume that the future behaves in a similar way as the past, this knowledge will allow us to draw conclusions about the future, based on metrics data collected earlier in the projects, before the majority of the resources have been committed.

3.2.3 Control

The third, and really the ultimate, purpose of metrics data, is to achieve control of the development process.

Using both information from evaluations and the predictions based on them, we may change our processes, methods and products, in order to achieve our goals.

As an example, if it turns out that more rigorous control of the early system design have resulted in systems with significantly lower maintenance costs, we then predict that if we use this more rigorous control in future projects, they will also have lower maintenance costs. Thus, we introduce this practice in our projects with the hope of achieving this benefit.

And this is really the situation we want to be in: A situation where practices, methods and tools can be continuously evaluated and modified to perform better in the future.

3.3 Types of metrics

Metrics can be classified in a number of different ways. It is common to distinguish between process metrics, measuring attributes of the development process, and product metrics, measuring attributes of the products produced.

One can also distinguish between metrics measuring an internal or external attributes [10]. An internal attribute of a product or process is an attribute that can be measured purely in terms of the product or process itself. An external attribute is an attribute that can be measured only with the product or process in relation to its environment. Of the examples above, “cost estimate of the next release of the product” would be a process metric measuring an external process attribute, while “completeness of the functional requirements” would be a product metric measuring an internal attribute of a product (the requirements).

One can further distinguish between objective and subjective metrics. “The number of defects discovered during component testing” would be an objective metric,
whereas “your judgement of the complexity of a component on a scale from 1 to 5” would be a subjective metric. It is preferable for a metric to be objective, since that will produce consistent and repeatable measurements.

In this project the focus has been on product metrics and, in particular, statically measurable code metrics. Defect metrics are also discussed at length in this report, but it is debatable whether they should be considered to be product or process metrics.

3.3.1 Size metrics

Size is a fundamental attribute of software products, both on its own account and as part in deriving other metrics. It can be of interest to know the size of the functionality of a component in order to estimate how long it will take to implement and test the component, or the size of an entire subsystem in order to decide whether a redesign is feasible within a specified time frame.

Lines of code

The most common aspect of size to measure is probably the number of lines of source code of a program. While this seems like a straightforward thing to determine, there exists a variety of definitions of what actually constitutes a line of code. For example:

1. Lines of code that are not blank, LOC.
2. Lines of code that are not blank or comments, NCLOC.
3. Non-commented source statements, NCSS.
4. Executable statements, ES.

The line count can vary a great deal between the different definitions, up to as much as a factor of 5 (!) [18]. This is something which is important to be aware of when making comparisons between different products or results. In practice, the most commonly used definition of a line of code seems to be “Lines of code that are not blank or comments”, NCLOC [24].

Function Points and Specification Weights

A different measure of size is Albrecht's Function Points [2] and DeMarco's Specification Weights metric\(^1\) [7]. Both metrics try to measure the size of the functionality of the software from specifications and design documents, through the use of different kinds of counting and weighting schemes.

Since the measures are based on design documents, it means that they are available at an earlier stage than code metrics, and that they are insensitive to the chosen implementation language.

\(^1\)The original name of DeMarco's metrics was the bung metrics.
A disadvantage of Function Points and Specification Weights, is that there is a certain amount of subjectivity in their calculation (in particular for Functions Points), and that they tend to underestimate the size of the system [10].

**Halstead Length and Volume**

Halstead defined a whole suite of metrics for different aspects of software development in his book “Elements of Software Science” [13], such as *Program Length*, *Program Volume*, *Program Level*, *Difficulty* and *Effort*. Length and Volume are interesting alternatives for measuring the size of program code and they are insensitive to the formatting and the style of the code. The *Vocabulary*, \( n \), of a program is defined to be:

\[
n = n_1 + n_2
\]

where: \( n_1 \) is the number of unique *operators* in the program and 
\( n_2 \) is the number of unique *operands* in the program.

The *Program Length*, \( N \), is defined to be:

\[
N = N_1 + N_2
\]

where: \( N_1 \) is the total number of *operators* in the program and 
\( N_2 \) is the total number of *operands* in the program.

The *Program Volume*, \( V \) is defined to be:

\[
v = N \log_2 n
\]

and can be interpreted as the number of bits required for an optimal encoding of the program.

**3.3.2 Complexity metrics**

The book by Fenton and Pfleeger [10] considers complexity to be aspect of size, while I rather consider size to be an aspect of complexity. In either case, one can identify several aspects of complexity [10]:

- **Problem complexity** is the complexity of the underlying problem.
- **Algorithmic complexity** is the complexity of the algorithm implemented to solve the problem.
- **Structural complexity** is the complexity of the structure and design of the algorithm.
- **Cognitive complexity** is the difficulty in understanding the algorithm.

The metrics most commonly referred to as “complexity metrics” are usually structural metrics of some kind. The focus of this project has been on these structural complexity metrics and, in particular, *cyclomatic complexity*. The assumption is that structural complexity significantly contributes to the cognitive complexity, which in turn is expected to contribute to the fault-proneness of the code.
McCabe’s cyclomatic complexity

The cyclomatic complexity metric, \( v(G) \), was introduced by McCabe in his 1976 paper “A Complexity Measure” [21]. It is one of the most famous and widely used complexity metrics. The metric measures the number of linearly independent paths through the flow graph of a program module, and has been used both as a complexity measure of the decision structure, as well as a measure to guide testing effort. For an in-depth treatment of cyclomatic complexity, see section 3.6.

Information Flow

The Information Flow metric was suggested by Henry and Kafura 1981 in their paper “Software Structure Metrics Based on Information Flow” [15]. While \( v(G) \) is concerned with the internal structure of a module (an intra-modular structural metric), Information Flow is concerned with the external structure between modules (an inter-modular structural metric). Information Flow considers the data flows between different modules, and defines the fan-in of a module, \( M \), to be the number of data flows which terminate in \( M \), and the fan-out to be the number of data flows which emanate from \( M \). The Information Flow of \( M \) is then computed as:

\[
IF(M) = \text{length}(M) \cdot (\text{fan-in}(M) \cdot \text{fan-out}(M))^2
\]

3.3.3 Quality metrics

Software quality encompasses a variety of aspects: correctness, reliability, usability, maintainability, flexibility and many other -ilities. In safety critical systems, such as air traffic control, (nuclear?) power plant operation and medical systems, etc., the aspects of reliability and correctness are naturally of utmost importance, but they are also of enough importance in other applications, that quality sometimes is thought of in a narrower sense, simply as the absence of software defects.

Defect metrics

The definition of a “defect” seems to vary, from a fault in code or design documents, to a system failure or a modification in code. Whatever the precise definition of a defect, they are often used as a direct measure of quality. One of the most common defect metrics is probably the defect density, which is defined as:

\[
\text{Defect density} = \frac{\text{Number of defects}}{\text{Size of product}}
\]

This is a good example of a size metric being used together with a defect metric in order to derive another, more useful metric. The size metric can be any of the ones mentioned in section 3.3.1 above. As it is fairly insensitive to the size of the product, the defect density corresponds quite closely to what we would think of as the “quality” of the product, and is useful for comparing different products. A valid comparison does of course require that one is using the same definitions for defects
and for size. For a more in-depth discussion of defects, see section 3.5, “Studying defects”.

Reliability metrics

While the number of defects in a product certainly is an interesting thing to know, the utility of that information lies with the assumption that having more defects is worse than having few. But how much worse? Reliability metrics tries to answer the more practical question of how likely the product is to fail over time, by estimating things, such as the reliability function, \( R(t) \), or the Mean Time To Failure, MTTF.

A complication in software reliability compared with other reliability studies, is that a reliability growth is to be expected over time as faults are being corrected. A model is assumed, such as, for example, the Jelinski-Moranda model [10], and the parameters of the model are estimated through historical failure data.

3.4 What to measure

Well, we are certainly spoiled for choices when it comes to things one could measure, but if everything that might be of interest is measured, it is likely that these very measurements only would reveal that most of the time is spent documenting the documentation effort. So, how do we know what is really important to measure?

The Goal-Question-Metric paradigm

The GQM paradigm was introduced by V. R. Basili et al. [4], and is a very useful way of identifying which metrics are needed in order to achieve a certain goal in the organisation. The GQM is a three step procedure (possibly iterative) from top to bottom:

1. List the goals of the metrics program.

2. Derive from each goal the questions that must be answered in order to determine if the goals have been met.

3. Determine what it is necessary to measure (which metrics to use) in order to answer the questions.

The following example of using the GQM will serve to illustrate the concept, although it is certainly lacking in several aspects. It would probably need several refinements as the answer to some questions would yield newer and better ones.
3.5 Studying defects

The study of defects, faults, failures or other kinds of “unwanted behaviour” in a product is of great interest. Operational failures are an important aspect of product quality and tracking them down and fixing them consumes vast amounts of resources for many organisations. Many post-release failures will also strain relations with customers, abuse the reputation of the product and generally cause damage that is difficult to estimate.

Finding Defects Late is very Expensive and Time-Consuming

[Diagram showing cost of defects across development stages]

Figure 3.1. The cost of defects, [28]

Looking at figure 3.1, it is easy to see why one would be interested in dealing with the problems as early as possible. The cost of finding and fixing defects grows tremendously the further down the life cycle of the product they are discovered. The shape of the cost function in figure 3.1 may be difficult to modify, but the number of introduced (created) defects could possibly be lowered, and shifting the number of
discovered faults towards the earlier phases in the life cycle would lower the overall cost of repair.

Now, the fact that defects cost money, and that it would be better if we found them as early as possible or avoided them altogether, is hardly going to raise any eyebrows. Everyone involved in software development knows this perfectly well. What is less known is how to deal with these problems in the best way. Despite some “best practices” that have evolved during the years, the “best way” is likely going to be a different way for each organisation. What works for one, is not guaranteed to solve the problems for another. This makes it necessary for organisations to keep track of their own ills in order to effectively tackle their problems.

3.5.1 Software problem terminology

The terminology relating to software problems and bugs seems to vary quite a bit. In this report I’m going to use the terminology as described in the glossary in chapter 2, except where explicitly stated. This terminology is in turn inspired by the IEEE Standard Glossary of Software Engineering Terminology [16] and the book Software Metrics: A Rigorous & Practical Approach [10].

So, just about everyone in this modern age seems to be acquainted with the concept of a “software bug”. Most of us will surely also vouch for this being a less than pleasant experience. For the readers happily unaware of this problem, I warmly recommend the first chapters of Les Hatton’s book, Safer C [14], for a sometimes tragic, sometimes humorous account of software related problems. I will not use the term “bug” in this document when referring to a software flaw, but rather the term “fault”. (See the glossary in section 2.2.)

American writer Robert Orbin once said: “To err is human—and to blame it on a computer is even more so”. As with many other things, it all usually starts with a human error. It is almost impossible to completely avoid making mistakes, and when it happens, the act of making the mistake will be referred to as an error. (See figure 3.2.) This error which has been made, can give rise to a number of faults, the actual encodings of the error, which are introduced in code, documentation or other products. It is possible that no faults are introduced, but more likely there will be one or more. These faults lie hidden in the product until such a time when their influence gives rise to a deviation from the expected behaviour of the product and a failure occurs. Several failures can, and do, occur due to a single fault. Their symptoms could possibly be different as well, making their relation, as to belong to the same fault, difficult to establish until they have been thoroughly analysed. Hopefully the failures will be documented in a trouble report in some manner, although there are surely cases where the failure will not be documented in any way. Finally, a number of changes are usually made to the system in order to correct the faults.

So what is a defect then? The term “defect” seems to be used to mean different things: faults, failures, errors or code changes. In this report a defect will always be a fault, but I will instead use the word defect in those places where I feel there could be a more general interpretation or use.
1 Error leads to...

...2 faults being introduced.

1 in code which is duplicated...
...giving 3 faults in the system.

2 of the faults give rise to 3 failures...
...which result in 2 trouble reports.

1 of the trouble reports is cancelled.

The other results in 1 correction...
...for a total of 2 changes to the system

**Figure 3.2.** The problem with problems

### 3.5.2 The relation between faults and failures

In the end it is the failures of the system that cost money, and that is what we want to avoid. In theory, you can make as many errors as you please, as long as the faults never show up. In practice, they tend to come back to haunt you. Unfortunately, by their very nature, you are unaware of the faults until they show up, and you have very little choice as to whether the failure turn out to be innocent or catastrophic.

An investigation made by Adams and others at IBM 1975–1980 came up with some very interesting results [1]. For a number of systems from which they had usage information and problem statistics, they studied discoveries and rediscoveries of post-release design faults. Their result confirmed the feelings of some of the service personnel, that a few faults were very “virulent”, causing many problems, while many were relatively harmless. Their investigation showed that only about 2% of the faults had a mean time to problem occurrence of less than 5 years of
usage time, about 10% had a mean time until occurrence less than 50 years and 33% of the faults could be expected to appear about once every 5000 years. (See figure 3.3.)

![Figure 3.3. Mean time to problem occurrence for faults in the study by Adams, [1]](image)

Thus actually, only a small fraction of the faults seem to be critical when it comes to reducing failures, and simply fixing or preventing any randomly chosen fault is not guaranteed to improve the reliability significantly. Although continuing to fix faults in this manner is likely to do so eventually.

To be able to discover and distinguishing these “vital few” faults from the “trivial many” early on in development would be very valuable indeed.

### 3.5.3 How to reduce the number of faults

Since faults in the product are the underlying cause for failures, it is obviously preferable to have as few faults in the product as possible. In order to achieve this, we can take two different, or preferably complementary, approaches:

1. Prevent the faults from being introduced, i.e. preventive action.

2. Be sure to fix the faults, i.e. corrective action.

The first approach would be to avoid introducing faults into the product in the first place. This may include a variety of “good practices”, such as having stable specifications, competent developers, good communication in and between development teams, and generally taking responsibility for the product. A systematic way
of eliminating “bad practices” is \textit{Root Cause Analysis}. This is a process in which the cause of the error is thoroughly investigated, and the result is fed back into the development process in the form of an action plan to prevent this from happening in the future.

The second approach would be to make sure that you are aware of the important faults and can remove them before the system is delivered to the customer. This may include careful and effective testing, inspections and reviews of code and documents, the usage of different tools, etc. See section 3.5.4 for more on different defect detection techniques.

\subsection*{3.5.4 Defect detection techniques}
Defect detection techniques are of interest both for corrective and preventive activities. While it obviously is necessary to detect a fault in order to correct it, it is also important to have information on the faults in order to prevent them from being introduced in an effective way. I will here give a brief overview of a few different defect detection techniques. A more comprehensive overview can be found in [25].

Defect detection methods can roughly be divided into three classes:

1. Static analysis.
2. Dynamic analysis.
3. Formal methods.

\textbf{Static analysis}

Static analysis is analysis of the subject without executing it. Examples include the usage of different tools for analysis of the code, or inspections, in which a section of code, specification, design, or some other artifact, is examined by an appointed group and potential problems are identified.

Data published by Grady [11] indicate that inspections and code readings are quite effective as defect detection techniques. (See table 3.1.)

\begin{table}[h]
\begin{center}
\begin{tabular}{|l|c|}
\hline
Activity & Defects found per hour \\
\hline
Regular use & 0.210 \\
Black-box testing & 0.282 \\
White-box testing & 0.322 \\
Reading/Inspections & 1.057 \\
\hline
\end{tabular}
\end{center}
\caption{Defects found per hour for different activities [11]}
\end{table}

A drawback with code inspections is that they are most effective when applied to all elements of the code, something which tends to be time consuming [25], and in defence of testing in table 3.1, one could argue that testing should find defects more relevant to the intended usage of the system.
**Dynamic analysis**

Dynamic analysis is techniques which require the execution of the subject in order to observe its behaviour. The archetypal example of dynamic analysis would be *testing*. One can differentiate between:

1. Unit testing.
   Unit testing is performed on units, modules or subroutines in their isolation. Unit testing can consist of *structural testing* which takes into account the internal structure of the module, and *functional testing* which examines the external behaviour of the module.

2. Integration testing.
   Integration testing is conducted when several units or modules are integrated with each other.

3. System testing.
   System testing is executed in order to verify the complete system.

4. Acceptance testing.
   Acceptance testing is performed by the customer before the product is accepted.

For most of these test phases one can use several testing techniques, such as:

1. Stress testing.
2. Boundary value testing.
3. Performance testing.

**Formal methods**

Formal methods involve the usage of rigorous mathematical techniques. Examples of formal methods would be *proof of correctness* of algorithms or writing the specifications in a *formal specification language* for ease of verification.

**3.5.5 Collecting defect data**

In order to learn how to prevent or detect software defects effectively, it is important to collect information about them. There are several aspects related to the defects that are of interest:

1. The failure.
   Information related to the manifestation of the fault would be things such as: What has failed? When did it fail? How did we detect it? What were the symptoms?
2. The fault.

Information related to the fault/defect would be: Where was the fault? What kind of fault was it? Why was it introduced?

3. The resolution.

Information related to the resolution would be: Was the problem fixed? When was it done? How was it done? How difficult was the correction?

Here, the primary focus will be on the fault. The information related to the failure/discovery or resolution will be considered to be attributes of the fault, although it would be preferable to introduce a separation between fault, failure and resolution information. For more information on collection of defect data and defect classification, see [25] or [17].

When gathering information about the fault, one can ask the usual questions of where, when and why?:

- Where was the fault located?
- When was the fault introduced?
- Why was the fault introduced?
- When was the fault detected?
- How/why was the fault detected?

It may also be of interest to classify the fault according to some other attributes. For example, its type, severity, priority, difficulty of correction, etc.

The location of the fault

Knowing the location of the fault is of obvious interest. This is information which is necessary to have in order to actually fix a particular fault, but it may also be used to create statistics on where most of your faults reside to help locating problematic areas and improper practices. As any practitioner of medicine would know, it is vital to know where it hurts before attempting treatment.

The Pareto Principle makes it likely that one can benefit a lot from being able to allocate resources to the right place at the right time. The problem is just to determine the appropriate level of detail. For example, simply knowing that there are problems in the design organisation does not help to allocate resources within.

From a utility point of view, it would be best to store as detailed information as possible, keeping track of the precise lines of code that was changed. This is not very practical, however, as you need quite a bit of additional information in order to assemble this information on a level that is useful. You would, for instance, need the source code together with the hierarchical structure of the code and product in order to determine whether this particular fault belonged to a certain subsystem.
The Pareto Principle

The Pareto Principle is also called the “20–80 rule”. In general, it states that a smaller subset of a population (the 20 %) has the biggest share of a particular attribute (the 80 %). An good example of this would be Adams’ study [1], in which a minority of the faults was found to be responsible for the majority of the failures. (See section 3.5.2.)

The Pareto Principle was also tested by Fenton and Ohlsson in [9], where it was applied to the number of discovered defects in modules. The results of their investigation turned out to be consistent with the Pareto Principle, with 20 % of the modules being responsible for nearly 60 % of the defects discovered during testing. For defects discovered during operation, the case was even more convincing: in one data set, 10 % of the modules were responsible for 100 % of the operational defects, and in another, 10 % were responsible for 80 % of the defects.

This provides a strong case for the value of carefully tracking the location of the faults in order to discover problem areas and focus resources to where they are most needed. “To separate the vital few, from the trivial many”.

From a practical point of view, you want the locations of the faults to be specified on the level that is appropriate for the analysis at hand. In the case of this project, it would have been ideal to trace the defects to a specific version of a particular function. In another case, another level of “granularity” might have been more useful.

It is necessary to find a balance between potential usefulness and what is deemed practical. The level of detail when specifying the location of a fault should be on a sufficiently low level to allow interesting analyses to be made, while still being available on a level that is convenient to use.

When the fault was introduced

The time of introduction is not to be confused or substituted with the time of detection. The time of detection is a poor indicator on when the error was made, simply stating that the fault was introduced “sometime before this point in time”.

The time of introduction is important to know in order to correctly associate the fault with the time, phase and activity that was responsible for introducing the fault. Further information on this activity and the cause of the fault can be used to identify, for example, problematic practices. Together with the time of detection, the time of introduction can also be used to calculate the latency of the faults: how long they have been present but undetected in the product. This is useful information on the ability (or inability) of the testing organisation to quickly locate faults.
Interesting information to record is the release, phase and activity which introduced the fault.

Information on when the fault was introduced does of course entail speculation on events of the past, and can as such be subject to quite a bit of uncertainty.

**Why the fault was introduced**

When attempting to prevent errors from being made, knowing the cause of the fault is of obvious importance. This information can be useful for identifying and preventing fault-inducing practices and other problems. When coupled with information on the location and time of introduction, one can evaluate different projects, phases and methods against each other or the company baseline.

Typical causes could be: **misunderstanding of principle, clerical mistake, unclear specifications**, due to the fix of another fault, due to changes in other parts of the system, etc.

As with the time of introduction, the cause of the fault is subject to speculation on past events and can be somewhat uncertain.

**When the fault was detected**

Unlike the time of introduction of the fault, the time of the detection is straightforward to determine, as it does not require guessing the time of a past event. Coupled with other fault information, this can be used to determine what kind of faults are discovered when and by whom. Many interesting questions about testing and detection could be answered. For example:

- Are we catching the appropriate types of faults when we should?
- Are there many faults of a certain type that cause problems by being discovered too late?
- Are our efforts in trying to catch a certain type of fault successful?

Information that is interesting to record is the time, the version/release of the code, document or product, as well as the phase of development which detected the fault.

**How the fault was detected**

The manner of detection of a fault can provide insight into what kind of defect detection techniques or tests are effective in catching faults. It may be that some tests or activities are more effective than others, particularly for detecting certain kinds of faults.

Examples of activities could be: different kinds of inspections, code reading, unit testing, testing of fault scenarios, performance testing, etc. There is a certain dependency on the phase in which the fault was detected, as not all activities are used in all phases.
The type of the fault

The type of the fault is important in order to know what kind of faults are being made, found and corrected. Knowing the types of faults which are found, or knowing which are not found when they should be, can provide guidance for how to make the detection process more effective, or provide a basis for checklists of common faults to be used during inspections.

The type classification can include categories such as, for code: Logic problem: Missing condition test and Interface problem: Subroutine mismatch: Inconsistent subroutine arguments, for documentation: Documentation problem: Missing item, or generally: Performance problem.

The severity of the fault

The severity of the fault is the level of disruption the fault causes in the product and for the users. This is important to know in order to prioritise the faults correctly. A minor fault with major costs of correction may not be worthwhile to correct, at least not immediately, while a stopping fault has to be corrected regardless of the cost.

A sufficiently, but not unnecessarily, detailed scale of severity, capable of capturing the necessary distinctions should be used. Levels ranging from No problem: Testing error to Critical: Major feature not working or system crashes in 3-6 steps are usual.

3.5.6 Difficulties with defect data

Simply measuring and collecting lots of data will not do much good, unless you also can prove that it actually provides a correct view of the current situation. Comparisons that are made and conclusions that are drawn must be shown to be fair and well-founded in order to ensure that the improvement efforts based on them will be effective. This is actually something which is significantly more difficult than it may seem.

The relation between faults and failures

As discussed in section 3.5.2, the relation between number of faults and number of failures is not as simple as it often is assumed. Many faults seem to be reasonably benign in the sense that they rarely cause a failure, while a smaller part of the faults seems to be responsible for most of the failures.

This puts the approach of improving the quality of the product by preventing or correcting simply any fault on a slightly uncertain ground, since failure to correct the critical faults will not significantly improve the reliability. It also affects the credibility of the defect density metric as a measure of quality, since higher defect density does not necessarily mean lower quality.
In practice, however, this is not likely to be a big problem. While finding and correcting a solitary fault (or a few) might fail to remove one of the critically important ones, finding and correcting a larger number of faults will become ever more likely to do so. With a sufficiently large number of faults removed, we would expect the reliability to have increased proportionally to the percentage of corrected faults. Actually, it should do even better than that, since the important faults are the ones of frequent occurrence and thus the ones likely to be detected and corrected at an early stage.

It could also prove that an intimate knowledge of the faults will reveal a connection between certain characteristics of the faults and their frequency of occurrence or severity. In this case, preventive or corrective actions could be focused on this particularly important group of faults.

In any case, it is important to be aware of the fact that not all faults are equally detrimental to the product.

**Measuring the right thing**

If you want to try to control and improve something like “quality” that, even in the narrower sense as “absence of defects”, depends on many different factors, you must be sure to correctly measure one aspect and be certain that the others are accounted for.

An example to illustrate: Given that in module 1, $M_1$, 50 defects have been found, and in module 2, $M_2$, 40 defects have been found, one would say that $M_1$ is worse. Given additionally that $M_1$ happens to be twice as big as $M_2$, one would instead say that $M_2$ is worse. Given additionally again that $M_1$ has received half as much testing as $M_2$, one would again say that maybe $M_1$ is worse.

Thus, there is a need to make certain that you are measuring the correct thing. If you, for example, want to measure the defect density correctly, you want to make sure that:

1. The definition of a defect is the same.
2. The definition of size is the same.
3. The testing effort has been consistent.
4. The reporting has been consistent.

since these are factors other than the “quality” that will affect the defect density measure.

**An attribute of the product or the testing process?**

If you want to be sure to have a system with a defect count of zero, you can achieve that in two ways:

1. Be perfect. Never make a mistake.
2. Do not test the system at all. Be sure never to use it either. Both of these will guarantee that you find zero defects and thus a perfect “quality”.

Given the absurdity of the paragraph above, it is surprising how defects by default seem to be interpreted in terms of product quality instead of testing process quality.

**Effect of defect detection effort on defect statistics** The precise number of defects in a product is an internal product attribute, that at a given time has a real and fixed value. This real value is unfortunately unknown to us, and the best we can do is to try and find a good estimate of it, for example, by testing the product and counting the defects found.

This estimate is, however, also strongly dependent on the process that produced it, as it is of any underlying defects in the product, and their respective contributions are impossible to separate without additional information. For example, if you knew the exact number of defects in a module, you could easily evaluate different testing processes by comparing the number of defects they managed to find in the module.

To be able to draw conclusions about the present state of defects and “quality” of a module, information is needed on the effort invested in the different defect detection techniques that were used to produce the defect data. (See section 3.5.4.)

Some sort of time scale could be used, such as:

- The number of releases.
- Age in months of the module.

But these are quite uncertain and indirect measures of testing/usage, since testing or usage could vary greatly between different products, phases and releases. Especially much so if the time interval is short.

A more direct measure of “how many chances the module has had to fail” would be to have actual testing/usage information of the module. This could be things such as:

- Hours of use of the system.
- Hours spent on inspections of the code.
- Number of test cases executed.

But even such detailed information is not guaranteed to describe the defect detection effort perfectly, since different types of usage will exercise the code differently and uncover different defects.

To make matters even worse, a single test or usage scenario will only exercise a fixed amount of code and may only uncover defects therein. This implies that in order to find a certain percentage of the defects in a bigger, more complex module, you also need to test it more than a smaller, simpler module.

While this may seem like an exceedingly difficult problem, it is unfortunately one that it is necessary to deal with in order to be able to draw good conclusion about quality from defect statistics.
Effect of previous defect detection activities on defect statistics  In order to answer questions about previous states of quality, such as the original defect density of the module, you also need to know how the code has been exercised in previous phases and releases of the code.

Since newly produced code only contains a certain number of defects, earlier defect detection activities will have uncovered and removed a number of them. Without knowledge of the history of the code, a code which has had no previous testing will appear much worse than a code which at the beginning actually was worse, but has undergone rigorous testing since then.

Using figure 3.4 to illustrate the problem, one can see that, since the testing effort and number of defects found are documented in testing phase 2, it is possible to correctly determine that the quality of module 1 is lower, both before and after these tests. However, assuming that the testing in the undocumented phase 1 was approximately the same, one would incorrectly conclude that module 1 also was of lower quality from the beginning, when it was not!

![Diagram](image-url)

**Figure 3.4.** Testing effort in different phases

Fenton and Ohlsson give in their paper [9] example of pre-release and post-release defect data which seem to be negatively correlated. That is, most pre-release defects imply least post-release defects and vice versa. This result would seem to be a good
example of how previous testing affects subsequent defect counts. This is also the explanation put forward by Fenton and Ohlsson in their paper.

Obviously, the testing in previous phases and releases will very much affect the defect statistics in the following phases, and if two products are to be compared, it is important to know their history to ensure a fair comparison.

Consistent and reliable reporting

While it is necessary to have controlled and documented defect detection activities, no chain is stronger than its weakest link, and it is also necessary to have consistent and reliable reporting of defects if the defect data is to be reliable. It must specifically not be the case, that some defects are randomly reported while others are simply fixed. It must be clearly defined and documented when and how defects are to be reported.

Some parts of the data-gathering activities are naturally subject to more uncertainty than others. The location of a fault in the code may, for example, not be straightforward to determine, and in the case of a misunderstanding, it is rarely the case that only one part is to blame. The time and cause of a fault is also subject to a certain amount of speculation and guesswork of past events.

A different contribution to uncertainty in the data, can be the reluctance of individuals or entire organisations to report data that could make them look bad. This could be such things as “forgetting” to report a fault that was discovered in their own code/document and just fixing it silently instead, or reporting the cause of the fault as “unclear specifications” instead of the more embarrassing “careless mistake”. It is understandably not a very pleasant thing to have to report incriminating facts on your own work, and the fear that this information may be used against you can introduce a significant bias in the data, hide important trends and “pass the blame around”. For suggestions on how to deal with this problem, see section 7.5, on “How to make it work”.

3.6 McCabe’s cyclomatic complexity metric

The by now quite famous cyclomatic complexity metric was introduced by Thomas McCabe 1976 in his paper “A Complexity Measure” [21]. It has been widely cited, refined and discussed about, but the question of its utility is still somewhat controversial. The general opinion seems to be that it relays useful information, although conclusive empirical evidence of its usefulness is difficult to find.

Cyclomatic complexity borrows its mathematical formulation, as well as the designation, \( v(G) \), from formal graph theory. It is defined on the control flow graph of a program, module, component or function, with a single point of entry and exit.
3.6.1 Flow graphs

The cyclomatic complexity, \( v(G) \), is one in a family of hierarchical metrics that is definable on the flow graph of a program [10]. A flow graph of a program is a directed graph, in which each node represents a program statement, and each edge represents the flow of control from one statement to the other. (See figure 3.5.)

![Flow Graph](image)

**Figure 3.5.** A Flow Graph

The cyclomatic complexity, \( v(G) \), of a flow graph, \( G \), is equal to the number of linearly independent paths through \( G \). What this means in general terms, is that any possible path can be generated through a linear combination of these linearly independent paths, that is, the set of linearly independent paths constitutes a basis set for all paths.

3.6.2 Calculation of the metric

The symbol \( v(G) \) for cyclomatic complexity is borrowed from graph theory where the cyclomatic number of a graph, \( G \), is designated by \( v(G) \). For a rigorous, if somewhat daunting, treatment of graphs, cyclomatic number and the theorems mentioned here, see [5].

**Definition of cyclomatic complexity**

A theorem of graph theory gives, that for a strongly connected directed graph, the number of independent circuits is equal to the cyclomatic number of the graph [5]. Another theorem gives that the cyclomatic number of a connected (but not
necessarily strongly connected) graph, $G$, with $n$ nodes and $e$ edges is: $v(G) = e - n + 1$.

The flow graph of a program (figure 3.5) is not strongly connected, however, but the addition of an extra edge from the exit node to the start node of the flow graph makes it strongly connected. (See figure 3.6.) This is represented by the addition of 1 to the cyclomatic number of the original flow graph, and the definition of the cyclomatic complexity is thus:

**Definition of cyclomatic complexity**

$$v(G) = e - n + 2$$  \hspace{1cm} (3.1)

**Counting faces**

For a planar graph with $e$ edges, $n$ nodes and $f$ faces, the following relation, *Euler’s formula*, holds: $n - e + f = 2$. Rearranging Euler’s formula above and using the definition of cyclomatic complexity (equation 3.1) we get:

$$v(G) = e - n + 2 = f$$

Thus, if the flow graph is planar, the cyclomatic complexity can easily be determined by counting the number of faces (regions) in the graph. (See figure 3.7.)
Counting predicates

A straight and unbranched flow graph has got exactly one out-edge for every node except for the end node, which has none. The cyclomatic complexity of such a graph is then equal to one: \( v(G) = e - n + 2 = (n - 1) - n + 2 = 1 \). A decision node with two out-edges increases the number of edges by one, while the rest of the nodes retain the same property as before, resulting in an increase of 1 to the cyclomatic complexity. Thus, if the flow graph consists of only binary decision nodes (with two out-edges), the cyclomatic complexity can be calculated as:

\[
    v(G) = d + 1
\]

where \( d \) is the number of binary decision nodes in the flow graph. In the case of multi-decision nodes with more than two out-edges, a similar procedure of counting decision nodes and incrementing the cyclomatic complexity by their respective “values” is possible. This can be implemented by counting the appropriate decision predicate directly from source code and thus avoiding to generate the flow graph. This is also the way cyclomatic complexity is measured by the metrics tool, CCCC, which was used in this project. For more information on CCCC, see section 4.2.

3.6.3 Usage of the metric

Cyclomatic complexity has been interpreted in primarily two ways:

1. As a measure of complexity.
2. As a way to guide testing.

A complexity measure

As suggested by the title of McCabe’s 1976 paper, cyclomatic complexity is fore-
mostly thought of as a measure of program complexity. It is a size independent measure that correlates well with our intuitive sense of complexity. A number of good examples of complex and non-complex code are given in [21].

A module with a higher cyclomatic complexity contains a bigger and likely more complicated control structure, which should make the module more difficult to write, maintain, and understand in general. Cyclomatic complexity should, as such, be an indicator of the cognitive complexity of the code. This has been subject to much criticism and many attempted refinements. (See, for example, [26].)

A suggestion put forward by McCabe, is to limit cyclomatic complexity to 10 or 15 in an attempt to keep complexity in a module on a manageable level. The idea is that complexity beyond a certain threshold defeats the capabilities of the human mind, and the likelihood to make errors would then increase significantly. The rationale for this is the psychological finding that most people are capable of mental manipulation of only 7 ± 2 objects simultaneously [23].

Structural testing according to McCabe

Another application for cyclomatic complexity, as suggested by McCabe in his 1976 paper, was the use of cyclomatic complexity to quantify testability and to guide testing effort. This has since then been further expanded on, and is described in detail in the report "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric" [29].

McCabe refers to this testing method as structured testing, while the IEEE Glossary [16] defines structural testing (observe the subtle difference), also known as white-box testing, to be the more general group of testing techniques which take into account the internal structure of the component. In order to avoid the almost inevitable confusion between the two, I will use the term “structural testing according to McCabe” to refer to McCabe’s “structured testing”.

As implied, structural testing according to McCabe is a structural testing technique, similar to statement testing or branch testing. Remember that cyclomatic complexity is the number of paths in a basis set of paths for the flow graph of the module. The structural testing criterion according to McCabe now states that you should test exactly this basis set of paths. The idea behind this testing technique is that the paths in the basis set are linearly independent and, since they constitute a basis, any other path can be expressed as a combination of the basis paths.

---

2 Well, almost size independent.

3 Also known as "perplexity".
Structural testing according to McCabe is a more rigorous testing technique than statement or branch testing. It achieves both complete statement coverage as well as branch coverage. A higher number of tests is usually also required in order to reach complete structural testing coverage according to McCabe. The more complete description of the control structure of the module produces a test set proportionate to the complexity of the module, and the independence between the paths does not allow for interactions between branches in the code to hide faults, something which could be the case when testing statements or branches. For an example, see [29, pg.33].

Note that the cyclomatic complexity in itself does not in any way produce a basis set that satisfies structural testing according to McCabe, it only gives the number of basis paths necessary. Given a method to generate linearly independent paths, though, the cyclomatic complexity provides a criteria for when enough paths have been generated. Also, no set of fewer than \( v(G) \) tests can fulfill the structural testing criterion according to McCabe. For details on “the baseline method” for generating a basis set of paths, see [29].

### 3.6.4 Limitations and problems

The most common critique of cyclomatic complexity is towards its interpretation as a measure of general complexity.

**Does not account for inter-modular complexity**

Cyclomatic complexity is defined on a per-module basis, and produces a measure of the amount of control flow logic in a module. In his paper [21], McCabe gives a more general formula for the cyclomatic complexity of \( p \) distinct, connected flow graphs:

\[
v(G) = e - n + 2p
\]

but this can easily be shown to be the sum of the cyclomatic complexity of the \( p \) flow graphs:

\[
e - n + 2p = \sum_{i=1}^{p} e_i - \sum_{i=1}^{p} n_i + 2p = \sum_{i=1}^{p} (e_i - n_i + 2) = \sum_{i=1}^{p} v(G_i)
\]

As Shepperd points out in his critique of cyclomatic complexity [26], the usage of formula (3.2) actually penalises a modular design. Thus, cyclomatic complexity is by itself not well suited to reflect the complexity of a collection of interconnected modules.

An issue to be aware of in relation to this, is that focusing too much on a particular aspect may cause an unchecked complementary aspect to get quite out of control. In order to solve a certain problem, a certain amount of decision structure is necessary. In an attempt to produce as low cyclomatic complexity as possible, factoring out functionality and structure into subprocedures is an effective way to lower the cyclomatic complexity of each procedure. However, an excessive division of
the functionality into too many small procedures will create an interaction problem with many procedures passing and sharing data. The inter-modular complexity will increase as a result of the decrease in intra-modular complexity. The complicated interactions will obscure the functionality and are also likely to make the code more prone to defects. Somewhere in between the two extremes, one can expect to find a minimum of cognitive complexity, where functionality and responsibility is judiciously assigned to appropriate functions and objects that are easy to understand. Clearly, simply lowering cyclomatic complexity does not in every case guarantee a less complex code.

Does not account for data complexity

Another aspect of complexity not accounted for in cyclomatic complexity is “data complexity”. The significance of this aspect was highlighted during one interview with a senior designer. The designer remarked that the first thing he always looks at when faced with new source code is the data structures. A complex data structure usually means a complex object. Objects which do not contain data, state, or any “memory” of previous usage, retain their behaviour between different invocations and are thus much more predictable and easy to understand.

Insensitive to the context of control logic

Criticism has also been directed towards the uniform treatment of decision predicates in cyclomatic complexity. Every binary decision simply contributes with +1 towards the cyclomatic complexity, even though some decisions obviously are more difficult to understand than others. This means that, for example, a straight sequence of ten consecutive if-statements has the same cyclomatic complexity as a mixture of ten while-loops and if-statements in a four-level deep nested structure. This shortcoming is particularly obvious in the case of the switch-construct in C/C++/Java. A switch-structure is in itself shallow, regular and quite easy to understand, but gives a significant contribution to the cyclomatic complexity. This is also acknowledged in McCabe’s original paper and different modifications to the metric for this construct has been suggested, for instance, to treat cyclomatic switch complexity logarithmically.

Criticism of the metric of this kind often takes the form of considering two pieces of code, A and B, of which \( v(G_A) \leq v(G_B) \), but where it is “obvious” that A is more complex than B. Modifications to the counting rules for cyclomatic complexity are then suggested, so that it will correspond more closely to the intuitive notion of complexity, such as in the example with the switch-constructs above.

A problem with these kinds of modifications is that they sever the original connection of the metric to graph theory and testability.
Is testing a path basis good enough?

Testing every conceivable path in a module is impossible when a loop structure exists, since the loop can be executed anything from zero to infinitely many times. Contrary to that, Watson and McCabe claim in their NIST report [29] that the basis paths have the following property:

“Testing beyond $v(G)$ independent paths is redundantly exercising linear combinations of basis paths.”

However, there is a definitive difference between testing the basis paths which make up the linear combination of a path, and testing the resultant path, as can be seen from the following example. (See figure 3.8.)

$$257\, p_1 - 256\, p_0 = p_{257}$$

![Diagram](image)

**Figure 3.8.** Constructing a path algebraically
Suppose we have a simple flow graph with only one loop construct. The cyclomatic complexity of the flow graph is 2, and a path basis for this flow graph would consist of one path exercising the loop once, \( p_1 \), and another path which does not, \( p_0 \). Suppose this loop was thought to be able to execute up to an integer (32 bit) number of times, but the fault was in implementing the counter as a byte (8 bit) number. This fault in the boundary condition of the loop would not be detected with fewer than 257 iterations of the loop, and it is obvious that simply testing the two basis paths and saying that we also have tested 257 executions of the loop, since that path, \( p_{257} \), can be expressed as a linear combination of \( p_1 \) and \( p_0 \): \( 257 \cdot p_1 - 256 \cdot p_0 = p_{257} \), is not the same as actually executing \( p_{257} \) itself.

Thus, structural testing according to McCabe does not subsume “all paths testing”, as the quote above would suggest, even though all paths happen to be “algebraically expressible” in terms of the basis paths.

On criticism of cyclomatic complexity

Some of the criticism put forward against the cyclomatic complexity is definitely motivated, and it is important to be aware of the limitations and problems associated with the metric. However, in some cases (for example in [26]) I find that the criticism is a bit similar to criticising height measurements in humans on the basis that it does not correlate well with, or explain, the physical health of a human being. Height is only one measure of a physical attribute of a human being, which by itself does not give the complete picture of his or her state of health. But when combined with a weight measurement, the combined height/weight ratio gives an indication of the corpulence of the person, something which is likely to correlate quite well with the physical health.

It is important to keep in mind that cyclomatic complexity reflects one aspect of complexity, and the inability of the metric to explain external attributes, such as quality or reliability, by itself, does not invalidate the utility of the metric in general.
Chapter 4

Description of the project

In this chapter, the systems, tools, methods and other considerations of this project will be presented.

First, in section 4.1, the system which has been studied will be introduced. Following that, the metrics tool CCCC and the trouble reporting system will be presented in section 4.2 and 4.3 respectively. The considerations of measuring defects/quality are discussed in section 4.4 and those of measuring complexity in section 4.5. The measures which ultimately were chosen to be used in this project are presented in section 4.6. A presentation of the modules chosen for investigation can be found in section 4.7, and, finally, a description of the actual procedure of performing the measurements in section 4.8.

4.1 Presentation of The System

The system which has been studied in this project will simply be referred to as “The System”.

The System is a platform product on which different kinds of packet switching network applications for the next generation of telecom and datacom networks can be built. (See figure 4.1.)

The System has been designed with scalability and flexibility in mind. It consists of both the hardware node, customisable and extensible with different device boards and additional subracks/subnodes, as well as the software operating on this hardware. The software consists of a multi processor, distributed, real-time control system, a flexible element management system and an ATM-based packet transport system.

4.1.1 Structure of The System

Many times in this report I am going to refer to different kinds of system entities, such as LMs, SWBs and SWUs. Because of that, I am going to give an introduction to the different system entities and their functions here. The reader may find it helpful to come back and review this section again, later on while reading the report.
The functionality in a large and complex system such as this one has to be broken down into different levels of abstraction and stepwise refinement. It is necessary to have appropriate entities to which responsibility and functionality can be assigned during system design, or which can be packaged and loaded on the target run-time platform. The entities that exist, their relations and properties, are defined and described in the product structure and the product type manual.

The terminology regarding the structures of the system tends to be a bit confusing, and I will therefore explain what I intend to use them to mean here in this report.

**The product structure** is collectively referring to the different structures used to design, document, implement, package and deliver the system/product.

**The implementation structure** is referring to the particular structure and the elements which are used to functionally break down the system and ultimately to implement it in code.

**Implementation structure**

The System is first, and most commonly, divided into a number of system areas, SAs, representing major groupings of functionality. (See figure 4.2.) There are, however, cases where the SA level isn’t being used.

Within the system or system area, the functionality is further divided into subsystems, which provide the system with a more specific area of functionality.
Each subsystem is again divided into Software Blocks, SWBs, which represent bigger areas of functionality within a subsystem. SWBs can be of various sizes, anything from 1000 to 50000 lines of code, depending on their content.

SWBs are, yet again, divided into Software Units, SWUs, which typically represent an object on the design level and are on the scale of a few thousand lines of code. The SWU is usually the level where the actual source code exists.

On the different levels in the system, there also exists various Interface Units, IFUs, which represent interfaces to functionality in different parts of the system.

**Delivery structure**

The deliverable version of the system is represented by the Delivery Product, or DP, at the top of the delivery structure. The DP consists of a number of system external interfaces for use by application development and a number of Delivery Modules.

A Delivery Module, or DM, is a collection of Load Modules being delivered together. A DM typically consists of all the Load Modules from a particular subsystem.

A Load Module, or LM, is an executable unit derived from the code base in the implementation structure. Typically, this is a compiled, binary executable that runs
on a processor in the system. The LMs are essentially the building blocks of the executable and deliverable system.

An Interface, or IF, is the deliverable version of one or several IFUs in the implementations structure. As with the IFUs, there exists IFs on different levels in the delivery structure. The IFs found under the DP are system external and for use by external application development, while the ones under the DM are for use only by internal development activities. This is also why there is an “internal delivery structure” demarcated in the delivery structure (figure 4.2).

4.1.2 The new process and product structure

At around the time of writing, a new development process, build support and product structure is being introduced in the design organisation.

The new Way of Working

The new development process is intended to be used as from release 4 of The System. It is thus referred to as the Rel4 Way of Working. The idea behind the new WoW is to build the system in small and easily verifiable steps. The Rel4 WoW includes, among other things:

- The system is built, integrated and verified regularly and frequently.
- Development is done in small steps (Work Packages) to build the final product.
- Verification of Work Packages is done before integration to allow fast and problem-free integration.
- Each integrated version will be a complete, up-and-running system.

Improved build support

Together with the Rel4 WoW, an improved build support will also be introduced. The new build support will take care of a number of tasks which earlier used to be performed manually.

- An automated publish and release mechanism.
- Automated and consistent labelling of code and deliverables.
- Centrally administrated config specs.

This will hopefully bring some order to the confusing array of naming and labelling schemes which used to exist earlier, as well as reducing the problems of having incorrect builds due to faulty config specs.
The new product structure

The change that probably is of most interest for this particular project, is the creation of a new product and implementation structure. (See figure 4.3.) At the time of writing, this is still work in progress and the structure presented here may not agree completely with the final version.

![Diagram of implementation structure and delivery structure]

**Figure 4.3. The new implementation and delivery structure**

The new product structure, as well as the improved build support, is being created with the intent of supporting the Rel4 WoW goal of frequent and incremental system integration. The new product structure is also intended to be simpler and less error-prone than the old one.

A number of changes are introduced in the new implementation and delivery structure. Some of them are highlighted below:

**Introduction of a new Block** The biggest of the changes is probably the introduction of a new Block. (See figure 4.3.) The SWB and the SWU in the old product structure are replaced by the Block, which will be the smallest manageable and deliverable part of a subsystem, suitably maintained by a design team of 2–5 people. The new Block will act as a sort of “container” for a deliverable entity, in
the sense that it encapsulates a LM or a library together with its associated source code. (See figure 4.4.) The Block will also be revision handled together with its associated deliverable.

Filesystem Structure

```
<table>
<thead>
<tr>
<th>SUBSYS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block_Name1</td>
</tr>
<tr>
<td>Block_Name2</td>
</tr>
</tbody>
</table>
```

The new Block

Development library

Load Module

```
<table>
<thead>
<tr>
<th>src/</th>
</tr>
</thead>
<tbody>
<tr>
<td>swu/*.c</td>
</tr>
<tr>
<td>ifu/*.c</td>
</tr>
<tr>
<td>test/</td>
</tr>
<tr>
<td>tb1/</td>
</tr>
<tr>
<td>tb2/</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Lib_Name1</th>
</tr>
</thead>
<tbody>
<tr>
<td>inc/*.h</td>
</tr>
<tr>
<td>.sig</td>
</tr>
<tr>
<td>.con</td>
</tr>
<tr>
<td>lib/*.a</td>
</tr>
<tr>
<td>obj/*.o</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>LM_Name1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ppc/im.ppc</td>
</tr>
<tr>
<td>Sparc/im.sparc</td>
</tr>
</tbody>
</table>
```

![Figure 4.4. The new Block](image)

As a result, many of the old SWBs, which previously served as a code base for a number of deliverables, will be divided into smaller and more numerous Blocks, each associated with only one deliverable. Code bearing entities which previously existed on other levels in the system, such as IFUs and SWUs, will be reassigned to a new Block under a particular subsystem. The result will be a wider and more shallow implementation structure, where pretty much any implementation object will be represented by a Block.

**No IFUs from subsystem level and above** There will no longer be any code bearing IFUs on any other level than the Block level. This means that IFUs which previously was external to the subsystem, system area, or system now is reassigned to a Block in a particular subsystem.

One consequence of this, is that it is no longer possible to determine the visibility and role of a particular IFU from its position in the product structure.
**LMs moved from delivery structure to implementation structure**  One of the objectives of the Block was to tie the deliverable product closer to its source code by aggregating them into a Block. If such a one-to-one relation between a Block and a LM was possible, it would allow for much easier mapping between code and deliverable.

Unfortunately, this is not always the case. There are significant amounts of source code that is shared and reused between different LMs. This will likely result in a situation where many of the Blocks deliver libraries and object code which is linked together into a LM in a separate Block. This will then result in build dependencies between the Blocks and the same many-to-many relations that was to be avoided.

**LMC replaces old DM**  The old concept of a DM is in the new product structure replaced by a *Load Module Container*, or LMC. As implied by its name, a LMC aggregates a number of LMs into a manageable entity similar to the old DM. But while the DM mostly was a convenient grouping of LMs for delivery from a subsystem, the LMC is a suitable set of LMs for purposes of deployment.

A LMC is intended to serve as a monolithic part of the run-time system that performs a well defined function within the system. The LMCs of the system can be added, removed or replaced to provide different configurations of the system.

**Introduction of the SC**  In an unfortunate name collision, the Delivery Module also appears in the new delivery structure, but with a different purpose. (See figure 4.3.) It is alternatively known as a *Software Configuration*, which, besides avoiding confusion, also better describes its purpose.

A *Software Configuration*, or SC, is a set of LMCs that make up a fully functional configuration of the system. By choosing a different set of LMCs and configuration data, another SC can be created for a different purpose. The idea is to provide a small number of *Reference Configurations* of the system which most applications can use, instead of supporting a potentially vast number of different configurations.

**External system interfaces collapsed into an IFC**  The external system interfaces are in the new delivery structure replaced by a single *Interface Container*, or IFC.

Since there are no interfaces on the system level in the implementation structure, the IFC will be made up of the appropriate IFUs from their respective Blocks. (See figure 4.3.)

### 4.2 The metrics tool: C and C++ Code Counter

The metrics tool used to extract complexity and size measures in this project is CCCC, the C and C++ Code Counter. CCCC is an open source metrics tool developed by Tim Littlefair as part of his Ph.D. in evaluating the use of software
code metrics in software development [19]. There are certainly other tools on the market that are more complete and reliable, but CCCC is readily available, simple to use, and the source code is available for modifications, should that become necessary. As such, it seemed to be a good choice as the metric tool.

4.2.1 Capabilities of CCCC

CCCC operates on a single file or a list of files at the time. It produces a set of XML and HTML files in the form of a report as output for the analysed code. It does not care about resolving dependencies in the code, and can thus be run on a single file or function in isolation. It also treats most declarations and preprocessor directives as blank lines.

CCCC seems to be more suited for C++ code than it is for C code, but most of its facilities are still useful for C. The metrics are divided into three categories:

1. Procedural metrics.

   Metrics defined on a per-function basis.

   • Lines of code, LOC
   • Lines of comments, COM
   • McCabe’s cyclomatic complexity, MVG
   • Lines of code per line of comment, L_C
   • Cyclomatic complexity per line of comment, M_C

2. Object oriented design.

   A number of metrics not very useful for C code.

   • Weighted methods per class, WMC
   • Depth of inheritance tree, DIT
   • Number of children, NOC
   • Coupling between objects, CBO

3. Structural metrics.

   A couple of metrics related to the structure inbetween modules.

   • Fan-in, FI
   • Fan-out, FO
   • Information Flow measure, IF4

   The most interesting metric for this project was, of course, cyclomatic complexity, but lines of code was also a very useful measure to have. The Fan-in, Fan-out and Information Flow might have been of interest, but turned out to be unusable for C code.
4.2.2 Limitations and problems with CCCC

CCC is unfortunately not a problem-free product, and a number of issues were discovered during the course of this project.

Crashes on certain constructs

First, it seems that the parser in CCCC is incapable of coping with certain legal C constructs. Very common `typedef struct data sdata` constructs will cause parse errors, but are in the end simply skipped by the parser and do not affect the operation of the tool. A much worse condition arises if the same kind of `typedef` construct is followed by a bitfield declaration, in which case the tool completely fails and aborts the processing of the current source code file with no more error output than the usual and frequent syntax error. Even though a very unusual occurrence, this introduces an uncertainty as to whether the tool actually measured all that was intended, or if a few functions just happened to be left out.

Miscalculates cyclomatic complexity

Another less serious, but unsightly, flaw in CCCC, is that it miscalculates \( v(G) \) by \(-1\) in certain cases. This was obvious from a significant number of functions with \( v(G) = 0 \), something which is impossible. This fault is due to incorrect counting of `return` statements.

Information flow measure unusable for C code

As briefly mentioned before, the Fan-in/Fan-out measure as defined by CCCC is not useful for C code. CCCC defines a Fan-in or a Fan-out as a data flow in or out of a “module”. A “module” in CCCC is a C++ class, and functions which do not belong to any class are all considered to belong to the module “anonymous”. Thus, in the case of C code, all functions belong to the same module and there is no Fan-in or Fan-out whatsoever.

No Halstead metrics

Other than these issues mentioned, one could possibly wish for CCCC to be able to calculate some of Halstead’s metrics which seem to be widely used, such as Program length, Program volume and Effort.

4.2.3 The implications of the problems for this project

The problem of CCCC crashing on some C constructs is really the only one which could have had serious consequences for this investigation. An unnoticed failure when parsing a source file might leave a number of functions out of the complexity data and affect the complexity distribution. These constructs are fortunately sufficiently rare not to make this a serious problem.
The miscalculation of cyclomatic complexity is of little concern, since an "off-by-one" error in $v(G)$ only would affect the hypothetical fault-proneness of the functions by the smallest amount, and the effect on the load modules in total would be the same for all of them.

The problem of not being able to use information flow as a measure of interaction complexity between functions, was a serious problem for the plans of creating an improved complexity measure for a load module as a whole. It was probably the most critical factor as to why this was not done.

4.3 The TR-system

The design organisation is using a change request management system to keep track of product feedback, such as requests for change or product defects (trouble reports). This TR-system is used in most of the design organisation and in the application development that is based on the platform. The number of users is estimated to a couple of thousands.

In short, one can say that the TR-system is a system designed to formalise and monitor the progress of issues requiring certain actions, such as failures that need to be corrected or changes/features that need to be implemented. This is a role it seems to be serving quite well too, although it is somewhat lacking when it comes to recording fault data.

4.3.1 Capabilities of the TR-system

The TR-system appears to be a very flexible and customisable system. It can be modified to record different kinds of information and to follow different procedures. It is possible to distribute the system across different sites and still use it as if it was a single installation. It also provides for integration with a version control system or software configuration management system. Because of this customisability, the features of different configurations may vary significantly, and I want to make clear that the system which is discussed below is the adaptation of the TR-system for this particular design organisation.

As mentioned, the task of the system is to record, monitor and manage Trouble Reports, TRs, and Change Requests, CRs. Although the system has a wider scope and use, my interest in the system is primarily for its abilities to record and manage TRs filed as result of a system failure.

A TR that is submitted follows a certain life cycle. (See figure 4.5.) Every TR starts in state NEW, and is then moved between different intermediate states until it reaches an end state, such as CLOSED or CANCELLED. Some of the state transitions are only allowed by a person assigned to a certain role in the TR-system, such as an area responsible. It is usually possible, or necessary, to submit certain information related to the life cycle phase of the TR when changing states. Relevant notifications of the state change of the TR are also sent by email to the individuals concerned.
TRs are organised into classes and projects. Different classes may have different process models and hold different kinds of information, and projects are then assigned to the appropriate classes. Projects, in turn, contain Products/Areas on which a TR is reported. At certain state transitions, certain information can, or must, be supplied. This information is divided into different categories:

**Initial information** contains information, such as a slogan, which product/area and version the defect was found in, the severity of the failure, and some information on the instance and activity which uncovered the defect.

**Analysis & decision information** holds comments from the change control board decision, estimated time for fixing the defect, etc.

**Delivery information** supplies information on the planned and actual release of the correction.
Verification information is information on the instance and person which performed the verification of the correction.

Attachments. There are usually a number of free-text attachments at the end of the TR. Here it is possible to elaborate on the problem, analysis, solution and verification. The information content here ranges from being non-existent to very detailed.

4.3.2 Limitations of the TR-system

The focus of the TR-system is, as mentioned above, on the administration of changes and problems, rather than the documentation of information about faults. It is not surprising that it is difficult for a system to be effective in both roles at the same time, and many of the “limitations” mentioned below stem from this fact.

TR-process sometimes unnecessarily cumbersome

There are situations where the life cycle of a TR is unnecessarily bulky and cumbersome. These are, in particular, situations where the fault is discovered, analysed and corrected almost immediately, such as during inspections, or early in development in component testing. In these cases, it is only of interest to document information about the discovery and removal of the fault, and some of the information and states in the TR-process are unneeded.

Location of fault poorly specified

The information in the TR-system is primarily for tracking the resolution of an issue or a problem. The Product/Area on which every TR is reported found and corrected, corresponds to, for example, a load module, a subsystem, or a piece of hardware. This is information on the appropriate level for the submitter who reports the failing component, or the tester who needs to verify the corrected component, but it is not on the level of detail that is useful for evaluating the design organisation. This lack of detail is in my opinion the most serious problem with the TR-system. See section 6.5 for more on this topic.

TRs not necessarily the result of a fault

Not all TRs are necessarily the result of an actual fault in the system. In trying to do many things, the TR-system allows for a wide variety in “types” of TRs. Some can be due to quite subtle kinds of unwanted behaviour, such as debugging output, error codes that are too verbose or too terse, or more like feature requests instead of an actual failure.
Lack of information about the fault and failure

In the cases where the TR is the result of an actual fault and failure, there is still a lack of useful information on the character and cause of the fault and failure. There are attachments to the TR which should be used to document the problem, the analysis, and the solution, but they are not always filled in and do not follow any specified format. This makes the information virtually useless for anything but a properly knowledgeable human reader. This lack of information on the fault and failure limits the usefulness of the defect data, and may conceal interesting and important trends.

Cloning of TRs

Another problem when collecting defect data, is the concept of cloning. When development is branched into separately supported products, a fault that is discovered in common code will result in the original TR being duplicated to the other development branches, this to ensure that the changes are being made to all instances of the code. The fault is the same, but there are two corrections necessary due to duplication of the code. With a recent release of the TR-system, it should now be possible to exclude duplicated TRs from the statistics, but many of the older defects are likely be counted twice or more in the case of cloning.

4.4 The quality measure

The idea behind this project was that complex code is more difficult to write and to understand, and as such, should be more prone to defects. There are, however, a number of different quality measures that might be of interest to study.

4.4.1 Defect density

The defect density was the quantity which first and foremost came to mind as a suitable quality measure. It is calculated as:

\[
\text{Defect density} = \frac{\text{Number of defects}}{\text{Size of code}}
\]

The defect density is a measure which corresponds well with our intuitive sense of the “quality” of the code. A certain “quality” of development should correspond to a certain frequency of introduction of defects in the code, and more densely distributed defects should make an executing thread more likely to encounter a defect. It is also a measure that is fairly insensitive to the size of the system. The defect data should furthermore be readily available using TRs from the TR-system as “defects”. See section 4.6.1 for more on TRs as “defects”.

Unfortunately, it turned out that the defect data available from the TR-system only specified the location of a defect on a load module, subsystem or system, all which are of a very big size. The only reliable way to trace a defect to a level closer.
to the code, was through manual inspection of the trouble report and version history, something which was an unreasonable work effort for the number of TRs in question. This proved to be a serious problem, since the hope had been to relate defects to a particular function, for which a single value of cyclomatic complexity could be calculated. Instead I was forced to consider, at the very least, LMs which could consist of hundreds of functions. This, more than anything, was a strong incentive to consider other measures of “quality”.

4.4.2 Modification rate

Instead of tracing a reported defect down to code, the possibility of studying the code itself was considered. Every corrected problem, whether reported or not, gives rise to a change in the source code and version history. A code with many and frequent changes is likely to have been problematic with many faults. By comparing consecutive versions of the code in the Multi-Version File System (MVFS), it should be possible to see how many times, and by how much, the code has changed.

This approach has the potential of detecting every corrected fault from all different phases of development, circumventing any uncertainties in the reporting of faults and the TR-system. It may also be possible to use the size of the corrections as a natural weighting scheme for the seriousness of the fault.

Unfortunately, there are other serious problems with this method. First, not all changes are due to poor implementation and fault corrections. Some parts of the code may be subject to extensive re-design due to active development and demands of additional functionality. It would be very difficult to distinguish the necessary changes due to new functionality from the “unnecessary” due to poor code. Secondly, the practice of coding and checking in is likely to vary between different programmers. Some may be of the habit of making fewer, but more extensive modifications at a time, possibly re-designing, fixing many faults and implementing new features at the same time, before submitting the changes to the code base. In this case, it will be very difficult to say which changes are due to corrections, and which are due to new functionality. Others will make smaller changes which are submitted more often. Finally, in giving up on the TRs, additional information on the fault is lost, such as severity or phase of detection.

4.4.3 Maintenance difficulty

There are a number of things which make studying maintenance difficulties in their relation to code complexity to seem appealing:

1. The quantity under consideration would be TR turn-around time, which is easy to extract.

2. One would suspect maintenance effort to be strongly related to complexity.

3. It makes maximum use of the available defect data.
Most will probably agree that it is more difficult to write complex code in a correct manner, but we are not very likely to be confused by our own work at the time of implementation. On the other hand, to understand, modify or correct the work of someone else, or even your own after six months have passed, is a completely different matter. A complex and confusing implementation is in this case sure to make modifications much more time consuming and risky, in some cases maybe creating more problems than they solve.

Another advantage with studying maintenance difficulties, is that it makes better use of the available defect data. Every single correction in code with a certain complexity corresponds to a specific turn-around time and will produce one measurement, whereas defect counts and densities require you to collect a sufficient number of defects in order to produce a single measurement.

The TR turn-around time is unfortunately not necessarily an accurate indicator of the actual effort invested in correcting the fault. A substantial amount of time can often be spent trying to recreate the problem and locating the fault through the use of what trace facilities and debugging output exist. The effort in this kind of debugging activity is not likely to be strongly dependent on code complexity. Moreover, it is possible that the TR has been lying open for a substantial amount of time, without being actively worked on, because other activities took precedence.

Another problem is that it will still be necessary to locate the precise function in which the correction was made, in order to measure the cyclomatic complexity.

4.5 The complexity measure

To measure cyclomatic complexity is in most cases straightforward and there are many automated tools capable of this. The tool used in this project to measure cyclomatic complexity was CCCC. (See section 4.2.)

McCabe’s cyclomatic complexity is defined on a function with one point of entry and one point of exit. (See section 3.6 for details on cyclomatic complexity.) Because of that, complications arise when considering the faults and quality of a larger component, such as a load module, which contains a greater measure of complexity than a single function. This stems both from having a larger number of functions, each with their own complexity, as well as another kind of interaction complexity from the interactions and dependencies between the different functions in the module.

4.5.1 Cyclomatic complexity vs. interaction complexity

There is, to a certain extent, a tradeoff between cyclomatic complexity and interaction complexity. A given problem has in itself a minimum amount of complexity that is needed in order to solve the problem. The actual implementation of the solution has in turn at least that much complexity distributed and assigned to different smaller subtasks. A single function which performs all that is necessary for the solution is one of the extremes, which has a maximum of cyclomatic complexity and a minimum of interaction complexity. A large number of small functions, each
performing a very simple and elementary task, is the other extreme, which has got a very low cyclomatic complexity for each and every function, but in which the many intricate interactions and dependencies inbetween the functions are very complex.

4.5.2 Complexity across functions

Disregarding this “interaction complexity” for a moment, there is still the problem of determining the “complexity” of a larger number of functions taken together. Each of the functions in the LM yields one value for cyclomatic complexity, and plotting the frequency of these values in a histogram produces an image of how the complexity values are distributed across the module as a whole. From the distribution, it is possible to extract different attributes, such as measures of location (mean, median, quartiles), or measures of spread (variance, standard deviation). These measures will, of course, incur a certain loss of the information carried in the distribution.

4.5.3 Complexity over time

Apart from the problems with the “spatial dimension” of deciding the location of faults and deciding which complexity measure to use, there is also the “temporal dimension” to consider. Even if it is possible to determine the complexity of the precise location where the fault was found, it is not exactly that complexity which is interesting. The complexity which we really would like to have is the complexity at the time the fault was introduced, since, by hypothesis, that would be what caused the programmer to make the error. This can be quite difficult to determine and a simplification would be to introduce a “time window”, during which the majority of the faults present in the code should have been found, and the complexity of the code can be considered to be the same. These two objectives come at the expense of each other and finding a good balance is likely to be a difficult judgement call.

4.6 Measures to be used in the investigation

In the end, the most important factor in the decision on which of the measures to choose, was simplicity. The time and resources that would have been needed for other, possibly better alternatives, were simply not reasonable for this project. It is also a sensible thing to try the simple ideas first.

Thus, on the basis of its simplicity and obvious feasibility, the chosen approach was to study a number of load modules with their associated number of trouble reports, and the unaltered cyclomatic complexity of their functions.

4.6.1 TR counts as a measure of module quality

For the quality measure, the simple TR counts were the only reasonable choice. Modification rate would have been difficult to measure accurately, if at all, and maintenance difficulty would still have demanded TRs to be traced to a precise location in code, something which would have required enormous amounts of time.
To collect the number of TRs for a LM was fairly straightforward using the tools accompanying the TR-system, and the number of TRs would hopefully be an acceptable measure of the relative number of faults among the chosen LMs. This requires the following to be at least somewhat true:

1. A TR is the result of a single, actual fault.
2. Any TR related to a fault already known or solved is cancelled.
3. There has been no cloning of TRs.
4. The practice of reporting failures has been the same over time and between different development/testing teams.
5. The testing has been comparable between different products and throughout all phases of development.

The only TRs included in the count were those that had been “resolved”, that is, TRs which were in states IMPLEMENTED, VERIFIED, or CLOSED. These are TRs which should have resulted in a change or action of some kind, and will hopefully correspond to an actual corrected fault.

There also existed a number of TRs which were reported on the subsystem or system instead of the load module. (More common in older releases of the system.) The hope was, however, that this practice would be somewhat equal throughout the design organisation, and that the relative number of TRs among the LMs still would be representative.

### 4.6.2 Cyclomatic complexity as a measure of module complexity

Since the chosen quality measure now was defined on a LM, it would have been preferable to have a complexity measure more suited for aggregate objects such as LMs than cyclomatic complexity is.

Unfortunately, to investigate and propose a new complexity measure for LMs is something which in itself would have made a suitable master’s project. Additionally, even if a really good measure could be found, there would still have to be tool support in order to measure it. The metrics tool used in this project, CCCC, did not have any measure of “inter-functional structural complexity”, and for that, it would have been necessary to acquire a new tool with better capabilities.

Thus, because of its obvious feasibility, the cyclomatic complexity of all the functions in a LM was measured, with the hope that this information would be able to capture enough of the complexity of a load module to make it useful.

### 4.6.3 Lines of function code as a measure of module size

Size measurements are interesting both on their own account and as part of another measure, such as in the defect density or for normalising the complexity distribution. (See section 5.2.1.)
As a measure of module size in this project, the number of lines of function code, LOFC, has been used. That is, any non-blank, non-comment line of code which belongs to a function. This size measure does, in particular, not include any kind of declarations, definitions or comments outside of functions.

The size of a module is computed as the sum of the sizes of all the functions in the module as reported by the metrics tool.

The reason for choosing this size measure, is that cyclomatic complexity only is defined for a function, and when computing complexity distributions, among other things, it is necessary to ascribe a certain complexity to the lines of code. A drawback is that the defect density measure is likely to seem a bit high when compared with other measurements that are using another definition of size. However, since this measure is used for all size measurements in this project, the relative size of the modules should remain the same and would not affect the results in a negative way.

4.7 The load modules chosen for analysis

For the first initial measurements and “proof of concept”, three load modules of varying size and “quality” were chosen. The results from these three LMs were fairly promising and led to further measurements of an additional eight.

The LMs were selected from four different subsystems, based on recommendations from interviews with individuals familiar with the LMs. The different load modules and some characteristics are summarised in table 4.1.

<table>
<thead>
<tr>
<th>Load module</th>
<th>Average size</th>
<th>Perceived complexity</th>
<th>Introduced in release</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM1</td>
<td>2100</td>
<td>Simple</td>
<td>REL 1.2.1</td>
</tr>
<tr>
<td>LM2</td>
<td>5800</td>
<td>Quite simple</td>
<td>REL 1.6</td>
</tr>
<tr>
<td>LM3</td>
<td>3800</td>
<td>Medium</td>
<td>REL 1.6</td>
</tr>
<tr>
<td>LM4</td>
<td>11000</td>
<td>Medium</td>
<td>REL 1.4</td>
</tr>
<tr>
<td>LM5</td>
<td>14000</td>
<td>Medium</td>
<td>REL 3.2</td>
</tr>
<tr>
<td>LM6</td>
<td>7000</td>
<td>Quite difficult</td>
<td>REL 1.6.1</td>
</tr>
<tr>
<td>LM7</td>
<td>7200</td>
<td>Quite difficult</td>
<td>REL 1.2.1</td>
</tr>
<tr>
<td>LM8</td>
<td>22000</td>
<td>Quite difficult</td>
<td>REL 1.2.1</td>
</tr>
<tr>
<td>LM9</td>
<td>5400</td>
<td>Difficult</td>
<td>REL 1.2.1</td>
</tr>
<tr>
<td>LM10</td>
<td>28000</td>
<td>Difficult</td>
<td>REL 3.2</td>
</tr>
<tr>
<td>LM11</td>
<td>27000</td>
<td>Very difficult</td>
<td>REL 1.6.1</td>
</tr>
</tbody>
</table>

Table 4.1. Summary of load modules chosen for investigation

The size in table 4.1 is the approximate average size of the LM (measured in LOFC, see section 4.6.3) over time, and may be very different from the size in the latest release. Some of the LMs have increased their size more than ten times during their lifetime!
The perceived complexity of the LMs in table 4.1 is not based on any measurements, but on the subjective opinions from the interviews. It is quite possible that someone else would disagree with the ordering presented here, and these values should only be taken as a very rough estimate of the complexity.

4.8 Performing the measurements

The purpose of this section is to give a not-too-detailed account of how the measurement process was implemented, for the use of someone with the intent of doing similar measurements.

The majority of the operations in these measurements are performed using ordinary UNIX tools, such as BASH scripts and AWK and SED programs. At the heart are a couple of BASH scripts which, with the use of an XML-like configuration file for each LM, first measures complexity, then extracts defect data and finally assembles the data into raw data files of complexity and defects.

In the configuration file for each LM, a number of selected releases are defined. These are releases which are considered to be “big” enough to have been put through a significant amount of testing and that are somewhat evenly spaced in time.

4.8.1 Measuring complexity

The most complicated part of the scripts is that of measuring the complexity.

For every LM, a number of releases are defined in its configuration file together with the location of the binary file of the LM. From each binary release of the LM, a Configuration Record, or CR, is extracted and processed by an AWK script in order to produce a list of source code files to be used for complexity measurements. This list is then fed into the metrics tool, CCCC, which performs the measurements, leaving, besides the near-to-read HTML reports, the raw datafile in the internal format of CCCC. This cccc.db file is finally processed by another AWK script to produce the data file containing the raw complexity and size data for all the functions in the LM.

4.8.2 Measuring defects

The defect data is relatively simple to gather using command line tools which accompany the TR-system.

Again, using the configuration file of the LM, a number of releases/baselines from which to gather TR-data are defined for every chosen binary release of the LM.

The corresponding releases/baselines are fed into the TR-system tool which produces a list of the relevant TRs. Only TRs belonging to any of the states IMPLEMENTED, VERIFIED and CLOSED were selected.

All the defects discovered in a particular release are considered to “belong” to the closest previous selected release, and are added to its defect count. The defect data is then written to a raw data file for further processing.
4.8.3 Assembling the data

The raw data files which contain the complexity and defect data may then be processed again by different scripts, in order to produce more neatly formatted data suitable for different analyses and diagrams. In this case, AWK scripts were used to produce the complexity distribution data for the LMs.

The tools used to produce the results and diagrams were OpenOffice Calc and Microsoft Excel, both capable of importing raw data from text files.
Chapter 5

Results of measurements

This chapter will present the analysis and the results of the measurements together with some brief commentaries.

The defect data is discussed in section 5.1, and section 5.2 deals with the complexity data. The potential relation between cyclomatic complexity and defects is finally investigated in section 5.3, while section 5.4 takes a look at the relation between size and defects, as well as between size and complexity.

*Note:* In this chapter, simply “complexity” will often be used to refer to “McCabe’s cyclomatic complexity”.

*Note2:* For the reader concerned with the measurement theoretic properties of cyclomatic complexity, I refer to the discussion in appendix A.

5.1 The defect counts

In these measurements, the number of resolved TRs was used as the defect count. (See section 4.6.1.) A few typical distributions of defects over the different releases can be seen in figure 5.1.

The appearance of the defect distributions was very surprising, since the defects had been expected to be much more spread out over time. Instead, about half of all the reported defects of LM6 and LM9 is reported on release 1.6.1 while the other half is spread out across the other releases. Most other LMs show the same pattern: “the lions share” of the defects are reported on a few releases, with the other releases having none to very few defects.

The implications of these results were simple enough: No time interval other than the entire life of the LM would be sufficient in order to “even out” the greatly varying “quality”. Choosing such a large time span would, however, create problems with the LM changing significantly over time, both in complexity and size.

In order to calculate a representative defect density, $d_p$, for the LM, the sum of all defects was divided by the average size of the LM in the chosen releases. The size of the LM was calculated as the sum of the sizes of all the functions in the LM. (See section 4.6.3.)
Figure 5.1. Defect distributions
5.2 The complexity values

A LM typically consists of a few hundred functions, each of them producing one value of cyclomatic complexity for every chosen release of the LM. We are confronted with the problem of finding a joint representation for the complexity of hundreds of functions, as well as the problem of a complexity that varies over time.

5.2.1 The complexity distribution

The hypothesis of the investigation was that higher complexity would lead to code that is more prone to faults. The foremost thing which comes to mind is probably to assume that a certain complexity corresponds to a certain defect density. This either implies that a certain amount of complexity will make the entire function equally difficult to understand, regardless of size, or that the size is more or less constant given the complexity.\(^1\) We need some kind of complexity measure that is independent of the size of the LM.

For every chosen release of a particular LM, there was a set of complexity values belonging to the different functions. The complexity was divided into intervals of 5 units and a frequency diagram was plotted, showing the number of functions which had a complexity in a certain interval. (Figure 5.2.)

![Figure 5.2. Number of functions vs. complexity for release 32](image)

\(^1\)They are strongly correlated. See section 5.4.2.
For all functions in one category, the sizes of the functions were summed to get the total number of lines of code which could be said to have a complexity in this interval. (Figure 5.3.)

![Graph](image)

**Figure 5.3.** Number of lines of code vs. complexity for release 3.2

This was then normalised by dividing each category with the total size of all functions, resulting in a size-independent complexity distribution for the LM. (Figure 5.4.)

To get a single complexity distribution from the different releases, the different distributions were simply averaged.

### 5.2.2 The complexity density distribution

The assumption in the previous section, that a certain complexity corresponds to a specific defect density, gives rise to a somewhat unexpected result when you allow the size to vary independently of the complexity.

\[
\frac{d}{s} = f(v) + \beta \iff d = sf(v) + \beta s
\]  

(5.1)

Suppose the relation between complexity, \(v\), and defect density, \(d/s\), is of the form in equation 5.1. If we multiply by size, \(s\), we get that the complexity dependent term in the expression also grows linearly with size, when the size contribution to the defects already has been accounted for in the \(\beta s\) term. In other words: A function twice as big will always have twice as many defects, even though the complexity is the same.
Figure 5.4. Complexity distribution for release 3.2

This is not really what we would expect. A function that is twice as big should probably have a few more defects due to the increased size, but the contribution of defects from complexity should remain about the same. To be precise: we would rather expect a certain complexity to correspond to a certain number of defects instead of a defect density. Something like the expression in equation 5.2.

\[ d = f(v) + \beta s \]  
\[ (5.2) \]

Since we do not want to see the contribution of the size term, \( \beta s \), in our defect count, we would prefer to still use the defect density, \( d_p = d/s \), where the size contribution is constant (equation 5.3).

\[ d_p = \frac{f(v)}{s} + \beta \]  
\[ (5.3) \]

This will unfortunately introduce a size dependency in the complexity term again. However, if the complexity dependent function, \( f(v) \), is a linear mapping, \( f(v) = \alpha v \), we may consider the defect density to be a function of a new quantity, the complexity density, \( v_p = v/s \), as shown in equation 5.4.

\[ d_p = \alpha \frac{v}{s} + \beta = \alpha v_p + \beta \]  
\[ (5.4) \]

In this case, even though the functions may be of various sizes and complexities, a certain density of complexity, \( v_p \), will correspond to a specific density of defects, \( d_p \).
The assumption that complexity is linearly related to the number of defects, is, of course, something which can be questioned. One of the consequences of the linearity is that “complexity does not interact with complexity”. The defect contribution of a certain amount of complexity is the same regardless of any other complexity in the same function. While this is not the only possible scenario, it is one that is intuitively plausible, and it is, again, a good idea to try simple things first.

The complexity density was thus calculated as “cyclomatic complexity per line of code” for each of the functions in the LMs. As with the complexity distribution above, the different densities of the functions were plotted in a frequency diagram, weighted by their size, and then normalised by dividing with the total size of the functions in the LM. This resulted in a complexity density distribution for every release of the LM. The final distribution was calculated as the average of the different releases. The distribution of functions, lines of code and percentage of code over the complexity density for a few LMs can be seen in figure 5.5, figure 5.6, and figure 5.7.

![Figure 5.5. Number of functions vs. complexity density for release 3.2](image)

### 5.2.3 What about the really high complexity values?

The cyclomatic complexity is the number of “linearly independent” paths through the flow graph of a function. In the complexity data it is possible to find a few functions with a very high complexity. One with \( v(G) = 123 \) and even one with \( v(G) = 169 \), which certainly is very much!
Figure 5.6. Number of lines of code vs. complexity density for release 3.2

Figure 5.7. Complexity density distribution for release 3.2
What do these “massively complex functions” look like? Well, most of the very high complexity functions seem to consist predominantly of switch constructs, possibly nested with if ... else ... or other switch constructs, but mostly in a well structured manner. Some of the high complexity functions were part of a state machine, switching first on input and then on internal state. Others were involved in diagnostic printing functionality and similar non-central functionality. Very few of the high complexity functions under closer examination seemed to have a really complex and “messy” structure, but were rather unfairly penalised by the way cyclomatic complexity treats switch constructs.

5.3 The defect and complexity relation

The objective of the investigation was to determine whether load modules that are “more complex” in some aspect also are more fault-prone (i.e. have a higher defect density), and, if this is the case, determine what this relation looks like.

A number of different approaches were attempted. The quantities investigated were:

- Pure complexity (section 5.3.1).
- Complexity density (section 5.3.2).
- Complexity mass (section 5.3.3).

5.3.1 Defect density and complexity

The complexity distribution provides a view of how the complexity in the LM is distributed as a whole. From this distribution, the idea was to extract some characteristics that could be related to the defect density.

The complexity distribution can also be thought of as the probability function, \( p(V) \), of a stochastic variable, \( V \), where the outcome, \( v \), is the complexity of a randomly chosen line of code from the load module.

A few different measures were extracted from the distribution, in order to see how they related to the fault-proneness of the load modules.

**Expectation value of complexity**

A simple measure that is possible to derive from the complexity distribution is the expectation value of the distribution (equation 5.5).

\[
E(V) = \sum \limits_i v_i p_v(v_i) \tag{5.5}
\]

This expectation value was computed for the 11 LMs, resulting in the graph in figure 5.8.

One could hope to discern the underlying relation between complexity and defect density “through the average” if:
1. The mapping \( v(G) \mapsto d_{\rho} \) is linear, so that \( E(d_{\rho}(V)) = d_{\rho}(E(V)) \).

2. The complexity distribution always is located narrowly around the expectation value.

![Graph showing defect density vs. E(V)](image)

**Figure 5.8. Defect density vs. expectation value of complexity**

Perhaps not very surprisingly, it is not possible to discern any particular relation in the graph. Most of the modules are clustered around a small area of the graph almost randomly. A few modules distinguish themselves from the majority.

- LM5 is fairly complex, but has a very low defect density.
- LM6 has a low complexity with a surprisingly high defect density.
- LM7, in a similar way to LM5, seems fairly complex but has a low defect density.
- LM9 is the most defect ridden module by far and it is also more complex than most.
- LM10 is the most complex module by far, but is not much more fault-prone than, for example, LM6.

**Expectation value of modified complexity**

While I would like to find a relation between complexity and defect density, computing the expectation value of the complexity itself, \( E(V) \), may not be the best
approach. In computing $E(V)$ much information about the distribution is lost, and if the relation between complexity and defect density, $g : v(G) \mapsto d_\rho$, is more complicated than a linear function, the expectation value of the complexity may relate poorly to the expectation value of the defect density.

A better result could be hoped for by making a clever guess of the relationship between complexity and defect density: a qualitative mapping $g : v(G) \mapsto d_\rho$, of which an expectation value could be computed (equation 5.6) that could relate better to the experimental defect density.

$$E(g(V)) = \sum_i g(v_i) p_Y(v_i)$$

(5.6)

In particular, if I magically were to guess precisely the correct mapping, $g : v(G) \mapsto d_\rho$, and the experimental measurements were accurate, the plots would yield an almost perfectly straight line.

First, one would expect that in code with a relatively low complexity, say less than 10 or 15, the complexity is still within manageable limits and does not contribute strongly to fault-proneness, whereas the quality of code with a complexity higher than 15 or 20 would start to deteriorate quickly.

Secondly, since many of the very high complexity functions predominantly seem to consist of relatively simple switch constructs, one would maybe not expect them to contribute in proportion to their complexity, and their influence should be reduced. The complexity from an interval in the “middle range” should be the most dangerous.

**Using a Gauss function** One simple model I tried was a Gauss function: $g(x) = e^{-(x-\alpha)^2/\beta}$. It has the property of both an exponential rise and fall-off centred around a value. The function I used to get the “best plot” was centred around $v(G) = 45$ and fell off to $1/e$ in about 28 units. It can be seen in figure 5.9(a). The plot of the defect density against the expectation value of this function, $E(g(V))$, can be seen in figure 5.9(b).

The LMs in the middle of figure 5.9(b) are a little bit more spread out than in the plot of $E(V)$ (figure 5.8), and there seems to be more of a linear relation between the two quantities. Modules which deviate from this linear trend are, as with $E(V)$, LM5, LM6, LM7 and LM9.

**Using a $\chi^2$ function** A problem with the Gauss function is that it is completely symmetrical and will fall off very quickly for higher complexities. Because of that, another function was tried which does not nullify the contribution of higher complexities so completely: $h(x) = x^\alpha e^{-x/\beta}$. This function is somewhat similar to (and certainly inspired by) a $\chi^2$ distribution. The function which was used consists of a quadratic growth factor and an exponential decay factor. It can be seen in figure 5.10(a). The plot of defect density plotted versus the expectation value of this function, $E(h(V))$, can be seen in figure 5.10(b).
\[(a) \ g(v(G)) = e^{-((v(G) - \mu)^2)/2\sigma^2}\]

(b) Defect density vs. expectation value of \(g(V)\)

**Figure 5.9.** A Gaussian weighting function and the resulting plot
Figure 5.10. A $\chi^2$ weighting function and the resulting plot
The appearance of figure 5.10(b) is similar to that of figure 5.9(b) with a somewhat linear trend, although LM5, LM6, LM7 and LM9 still deviates noticeably.

5.3.2 Defect density and complexity density

As discussed in section 5.2.2 above, one could argue that a complexity density measure, \( v_\rho = v/s \), should be related to the defect density. A complexity density distribution was then computed for each load module, and, as with the complexity distribution, a couple of different measures from the distribution were investigated.

Expectation value of complexity density

In an analysis similar to that of the complexity, the expectation value of the complexity density was investigated. As with the pure complexity measure, this should be an adequate measure, provided that the fault-proneness is not too far from linearly related to the complexity density, or if almost all of the distribution is centred around the expectation value. The resulting plot of \( E(V_\rho) \) can be seen in figure 5.11.

![Figure 5.11. Defect density vs. expectation value of complexity density](image)

It is difficult to imagine there to be some kind of relation between the measurements. A few things are interesting to make note of:

- LM9 and LM10 are both extreme outliers, but for different reasons. LM9 has the most faults by far, whereas LM10 is the most complex by far.
• LM6 appeared to be very simple in the complexity plots, but is one of the more complex ones when complexity density is considered.

• A number of modules, such as, LM1, LM3, LM5 and LM7 are remarkably free of faults while still being somewhat complex according to this measure.

**Expectation value of modified complexity density**

Again, as with the complexity, the expectation value of a couple of different non-linear functions of the complexity density was computed, to see if it was possible to better distinguish between the load modules.

**Using a power function** One hypothesis is that the lower complexity densities are not particularly dangerous, while higher complexity densities results in a quickly growing defect density. To investigate this theory, I used a convex function with a strictly positive second derivative, such as the power function: \( g(v_p) = (v_p + 1)^6 \). This function can be seen in figure 5.12(a). The expectation value of this function, \( E(g(V_p)) \), was then computed, and the defect density plotted against this value. This plot can be seen in figure 5.12(b).

The plot of \( E(g(V_p)) \) in figure 5.12(b) is very similar to the one of \( E(V_p) \) in figure 5.11, and not any better. LM9 and LM10 are the archetypal outliers, while LM5 and a few others are just slightly irregular. One would be hard pressed to imagine there to be some kind of relation from these points.

**Using an arctan function** Another thing which comes naturally, is to try the “opposite” of a convex exponential or power function, such as a concave logarithmic or fractional power function. This would reflect the idea that fault-proneness of code is growing quickly during a critical density interval in the beginning, but will level off for higher densities. Kind of: “It is already so bad it really cannot get any worse”. The function used here was \( \text{arctan}(x) \), which levels off asymptotically at \( \pi/2 \) and has a nice curvature. It can be seen in figure 5.13(a). The expectation value of this function, \( E(h(V_p)) \), was computed and plotted in figure 5.13(b).

Yet again, the plot of \( E(h(V_p)) \) is not very different from either \( E(V_p) \) or \( E(g(V_p)) \). Maybe a little more evenly distributed, but otherwise mostly the same.

**5.3.3 Defects and complexity mass**

The term complexity mass deserves a shorter explanation: Up until now, we have been considering some kind of representation of the “average complexity” and relating it to what must be thought of as an “average fault-proneness”. Another way would be to consider the total amount of complexity in a module:

\[
V_{\text{tot}} = \sum_i v(G_i) \quad v(G_i) \quad \text{Cyclomatic complexity of function } i
\]

I will refer to the sum of the complexity of all functions in a module as the *complexity mass* of the module.
(a) $g(v_\rho) = (v_\rho + 1)^\alpha$

(b) Defect density vs. expectation value of $g(V_\rho)$

**Figure 5.12.** A power weighting function and the resulting plot
(a) $h(r_{\rho}) = \arctan(15r_{\rho})$

(b) Defect density vs. expectation value of $h(V_{\rho})$

Figure 5.13. An arctan weighting function and the resulting plot

71
Defect counts and complexity mass

Despite my primary interest in defect density, one should probably expect complexity to be related to defect counts rather than defect density. (See section 5.2.2.) Thus, the thought occurs that one may consider the total complexity (the complexity mass) of a LM, in relation to the total number of defects in the LM. A perhaps non-obvious implication of such an analysis, is that the relation, \(d(v)\), between complexity and defects (again) would have to be linear, in order for the sum of the complexities to uniquely determine a certain number of defects. That is:

\[
\sum_i d(v_i) = d(\sum_i v_i)
\]

The two quantities were straightforward to plot, and can be seen in figure 5.14.

![Graph showing number of defects vs. complexity mass](image)

**Figure 5.14.** Number of defects vs. complexity mass

The reader of keen eyesight might have noticed that LM10 is missing in figure 5.14. This is due to the fact that LM10 has a more than twice as large complexity mass as any other LM and would make most of the plot difficult to read. LM10 is included in figure 5.15 below, showing the complexity mass of functions with \(v(G) \geq 20\). The assumption here is that only the complexity above a certain threshold, say 20, is responsible for the majority of the defects.

In both of these figures there seem to be an obvious relation between defects and complexity mass. When looking at figure 5.14, one would guess the relation to be linear, and the relation in figure 5.15 appears to be almost logarithmic.
Figure 5.15. Number of defects vs. complexity mass, $v(G) \geq 20$

If these plots seem surprisingly good, one should consider that complexity is strongly correlated with size, and we are thus also seeing the size dependency of defects in these plots. It is interesting to compare these plots with the plots of defect counts vs. size in section 5.4.1.

**Defect density and complexity mass**

Having plotted defect counts versus complexity mass, it might be interesting to also plot defect density versus complexity mass. This was done for the total complexity mass in figure 5.16, and for the complexity mass of functions with $v(G) \geq 20$ in figure 5.17. Again, LM10 is missing in figure 5.16 but present in figure 5.17.

The plots do not indicate any kind of relation between complexity mass and defect density in a LM.

### 5.4 Other analyses

#### 5.4.1 Defects and size

It is a fact that more code is likely to contain more defects. All other things being equal, one would expect twice as much code to contain about twice as many defects. For any metric that tries to explain or predict defects, the goal is to perform better than simple size metrics such as lines of code. For this reason, the relation between
Figure 5.16. Defect density vs. complexity mass

Figure 5.17. Defect density vs. complexity mass, \( v(G) \geq 20 \)
defects and size was investigated for the load modules in this project. The size measure used was *lines of function code*, LOFC, as explained in section 4.6.3.

**Defect counts and size**

Pretty much in the same way as with complexity mass, one would expect a bigger module to contain more defects. The number of defects in a load module was plotted against its average size in figure 5.18.

![Graph showing number of defects vs. number of lines of function code](image)

*Figure 5.18. Number of defects vs. number of lines of function code*

Figure 5.18 is especially interesting to compare with figure 5.14, “Number of defects vs. complexity mass”. In fact, they seem to be more similar than dissimilar. This serves as a good illustration of the strong connection between complexity and size, and the need for investigations that are able to differentiate between the two quantities.

A notable difference between the plots of complexity mass (figure 5.16 and 5.17) and lines of function code (figure 5.18) is LM10, which contains much more complexity than any other LM, but is not much larger than, for example, LM11. It is no great surprise that LM10 turned out to be the LM with the greatest complexity density by far.

**Defect density and size**

Given that I in the beginning of section 5.4.1 wrote that, “All other things being equal, one would expect twice as much code to contain about twice as many defects”,

75
it would be a quite pointless exercise to try to relate a “size independent” quality measure to the very thing it is supposed to independent of. But is the defect density really independent of size?

As is discussed by Fenton and Neil in [8], there are pretty much three different models of how defect density could be related to module size.

1. Defect density decreases as module size increases.

2. There exists an optimal module size where the defect density is at the lowest. Smaller or bigger modules are more fault-prone.¹

3. Defect density and module size are really unrelated.

Obviously, this question is very much dependent of what you define to be a module. Is it a function, an object (consisting of data and associated functions), or an executable package of objects as with the load modules investigated here? The answer could very well be a different one in each of these cases.

Here, however, I will take a quick look at whether there seems to be a relation between the defect density and size of a load module. The plot in figure 5.19 is interesting to compare with the plot of defect density vs. complexity mass in figure 5.16.

![Figure 5.19. Defect density vs. number of lines of function code](image)

As with the plot of defect counts vs. size, the defect density vs. size is strikingly similar to its complexity mass counterpart. Maybe not very surprising. There does

¹ Also known as “The Goldilock Conjecture”.

76
not seem to be any particular relation between defect density and the size of the load modules.

5.4.2 Complexity and size

With a grand total of 4246 paired measurements of $v(G)$ and size, it was pretty much unthinkable not to plot them against each other and see how they are correlated. Naturally then, this was also done, and the result can be seen in all its splendour in figure 5.20.

![Cyclomatic complexity vs. number of lines of code](image)

Figure 5.20. Cyclomatic complexity vs. number of lines of code

The reader of keen eyesight might have noticed that there are only 4244 data points in the scatter plot. The two missing functions had an extreme cyclomatic complexity of 169 and 267 respectively, and made the plot more difficult to read without adding significantly to the trend in the plot. All functions were included in the calculation of the regression line in figure 5.20 and the correlation coefficient below.

Although the vast majority of the data points are gathered in the lower, left part of the plot area, and not so easy to make out, there is an obvious relation between complexity and size, forming some kind of fan shaped area in the plot.

Something which is also typically done, is to calculate the correlation coefficient, which indicates to what extent two stochastic variables vary similarly around their
expectation values. It is calculated as:

\[
\rho(V, S) = \frac{\text{Cov}(V, S)}{\sqrt{\text{Var}(V) \text{Var}(S)}}
\]

Covariance of \(V\) and \(S\)

\(\text{Var}(V)\) Variance of \(V\)

The correlation coefficient for this data set was:

\[\rho(V, S) = 0.84\]
Chapter 6

Discussion of results

For the reader who skimmed through the previous chapter, it is probably obvious that it will not be possible to determine the relation between complexity and defects with any kind of confidence. Because of this, the discussion here will not be so much about what conclusions can be drawn from the results, as much as why it was not possible to draw any conclusions.

Section 6.1 discusses the hypothesised relation between complexity and defects, based on the results in the previous chapter. Section 6.2 deals with questions regarding the analysis of the data. Section 6.3 discusses uncertainties and considerations regarding the TR statistics, and section 6.4 deals with the complexity measure. In section 6.5, I will discuss the problem of how to specify the location of a discovered defect, and in section 6.6, I will talk a little about software quality.

6.1 The relation between complexity and defects

The original objective of this project was to investigate the relation between cyclomatic complexity and fault-proneness of code, and then determine suitable limits and guidelines to be used in the development process. The different results which were presented in the previous chapter will be summarised below and followed by a conclusion.

6.1.1 Complexity

When looking at the plots of defect density vs. the expectation value of the complexity (figure 5.8), a Gaussian function of complexity (figure 5.9(b)), and a $\chi^2$ function of complexity (figure 5.10(b)), it is possible to imagine some kind of linear relationship in the case of the Gaussian and the $\chi^2$ function. There are, unfortunately, also a number of outliers which significantly deviate from this trend, and too few data points to start fitting a curve to them.
6.1.2 Complexity density

The quantity I felt should be the most interesting, complexity density, turned out to be quite disappointing. The three plots of defect density vs. the expectation value of the complexity density (figure 5.11), a power function of complexity density (figure 5.12(b)), and an arctan function of complexity density (figure 5.13(b)), were more similar than different, despite radically different weighting functions in the expectation value. No particular trend could be discerned from these plots.

A possible explanation for the poor result could be that the relation between complexity and defects is significantly non-linear, contrary to the assumption made in section 5.2.2. If this is the case, it is possible for modules with the same complexity density to have very different defect densities, resulting in a great variance within the same group of LMs.

6.1.3 Complexity mass

More promising at first sight then, are the plots of the number of defects vs. the complexity mass. In figure 5.14 there appears to be a linear trend, although with significant variance. In figure 5.15, the growth of defects appears to be almost logarithmic, but this impression is mainly due to LM10, and could very well be coincidental considering the great variance of the other LMs.

If one feels that these plots show great promise, indicating that more complexity is associated with more defects, one has to remember that cyclomatic complexity is strongly correlated with size (section 5.4.2), and thus, to some extent, we are also plotting a measure of the size of the LM in these plots. A comparison of the plots of defects vs. size (figure 5.18) and defects vs. complexity mass (figure 5.14), will indeed reveal that they are very similar, and it is not really possible to say which one seems to be the better predictor.

As suspected, the plots of defect density vs. complexity mass or size (figure 5.16, figure 5.17, and figure 5.19), do not indicate any kind of relation between the two quantities.

6.1.4 Conclusion

It is, of course, quite optimistic hoping to find a good relation between complexity and fault-proneness with only 11 data points, and doing so will require measurements of very good quality. The quality of the measurements can in this case be suspected to be quite bad, as many factors are unaccounted for and may contribute to variations. It is possible that a larger number of measurements would have stabilised the situation a bit through the “law of great numbers”, but going from a meager 11 LMs to a modest 50 LMs would likely not have helped the situation much. Even if a seemingly good relation had been found, it would have been difficult to prove the validity of this relation.
The best conclusion one can draw from these plots, is that there may be some relation between defect density and complexity, and that complexity mass is comparable with size (LOC) as a predictor of defect counts.

If the defect data had been better, for instance, by allowing faults to be traced closer to code, or by using a subset/category of defects which could be more closely related to \( v(G) \), it might have been possible to reach a better conclusion. It does seem plausible that a relation between complicated/complex code and fault-proneness should exist, but it does not have to be very strong, nor easy to investigate.

6.2 The analysis

Good or bad results may equally well be the result of poor analysis as of poor data. The analysis of the data here could probably have been improved upon in some way, but with the growing suspicion that the quality of the data was poor, the time and resources were thought to be better spent on finding ways to improve the data for future investigations instead of performing a very detailed analysis of the present data. No matter how advanced the analysis, it will not be possible to find that which does not exist in the data.

6.2.1 Uncertainties in the analysis

LMs inappropriate objects of measurement

A LM may contain several hundreds of functions, each with its own cyclomatic complexity. From these it is necessary to calculate a single representative value which can be related to the fault-proneness. This value should characterise the complexity of the LM in the best possible way, in the sense that it should be as closely related to the fault-proneness as possible.

If we assume that a certain complexity (density?) in a function will cause a certain defect density, we would expect the entire LM to have the average defect density of all its functions, weighted by their respective sizes.

Unfortunately, simply computing the average complexity of the functions in the LM will not necessarily give us information about the average defect density of the LM, since the mapping from complexity to defect density may be non-trivial and we are losing valuable information in the average. (See section 6.2.2.)

A couple of attempts were made to find a good representation of the fault-proneness by computing the expectation value of a function of the complexity or the complexity density, \( E(g(V)) \). If this function happened to be precisely the correct mapping from complexity to defect density, it would produce a perfectly linear relation (disregarding any experimental uncertainty). However, if this representation is badly chosen, the relation to measured fault-proneness will be much more uncertain.

Suggested solution Avoid having to calculate averages by using objects of measurements that only produce one value of complexity, such as a single function.
LMs change significantly during measurement

The second serious problem in the analysis was the fact that the LMs changed significantly over time.

In order to even out the greatly varying defect data and to ensure that the testing and usage had been thorough enough, defects from the entire life of the LM were used as a quality measure. The problem with this approach was that the LMs might have changed significantly during such a long time. As an example, LM8 and LM7 increased their size more than ten times!

Thus, in addition to calculating a single representative value from a multitude of complexity values, it was also necessary to calculate a single representation of the LM from a multitude of versions. This was simply done by calculating the average complexity distribution of a number of chosen releases.

While the releases chosen to represent the LM were considered to be major releases, and hopefully sufficiently evenly spaced in time and usage to give a fair contribution to the average, this measure is still heavily dependent on the particular releases chosen. If some important releases were left out, or a few too similar ones were chosen, the average, consisting of less than 10 terms, would change significantly.

Apart from being sensitive to the subjective choice of releases, using an average of the complexity over such a long time will result in a significant loss of information about the LM, as well as obscure differences in complexity and quality which may have existed.

This uncertainty in the averages over time also affects the defect density measure, since the average size of the LM over time is used in the calculation.

Suggested solution  Make sure to have good quality and complexity data from shorter periods of time, during which the code can be considered to be mostly the same.

Averages poor way to limit complexity

Another problem with averages of complexity and fault-proneness is that even if suitable limits could have been found, these limits would also have been for averages, which is a rather poor way of limiting complexity and creating design rules.

Suggested solution  Yet again, make sure to use objects of measurement where averages can be avoided. That is, objects that are small enough to yield primitive values of complexity.

6.2.2  On the usage of transforms of complexity

In the analysis of the data in chapter 5, I'm using a method where I compute the modified (or transformed) value of the complexity, $g(v)$, and plot the expectation value of this transformed value, $E(g(V))$, instead of the expectation value of the
complexity itself. If \( g \) is a non-linear function, this will extract different and potentially more useful information from the complexity distribution by enhancing or reducing the influence of different characteristics. This could be a means of separating points that otherwise would be clustered too close together and a way to circumvent the problem of having to deal with averages.

Thus, by making a clever guess of the general shape of the complexity and defect relation, the computed expectation value, \( E(g(V)) \), will hopefully better match the experimental defect density value. As mentioned above, if this function of complexity is “the right one”, one would expect to see a straight line, \( y(x) = x \), since the plot is essentially defect density versus defect density (albeit one theoretical and one experimental). Variations would in this case only come from factors that are not accounted for, or due to the natural randomness in the experimental measurements of fault-proneness.

In order to find the “best” transform, several functions were investigated around a plausible general shape, and their parameters were adjusted to produce the strongest linear trend and the best separation of the data points. Now, when searching for a relation in this way, there is one thing one should be aware of: if the number of points is relatively small and the number of intervals (or “features”) in the distribution is of the same size, or larger, it may be possible to solve this problem of “finding the function which will put the data points on a straight line” as a system of linear equations. If this is the case, any function which is a solution to the system **will always put the points exactly on a straight line**. It will, in particular, be possible to find such a function even though the investigated relation is completely random, although the function is in that case likely to be quite bizarre.

This possibility of finding spurious (and likely bizarre) relations between two quantities can hopefully be avoided by having many data points, in which case the system of equations will be very overdetermined\(^1\), or by postulating the general shape of the function (as done here) instead of deriving it exclusively from the data.

### 6.3 The TR statistics

In trying to relate complexity to quality, it is of equal importance to correctly measure the quality as the complexity itself. Failure to correctly measure that which we purport to control, means that we could be precisely right without ever knowing it.

In this investigation the number of trouble reports was used as the measure of quality. One can, of course, question whether this is a good measure of the quality of a product, which consists of many different aspects (section 3.3.3), but since the frequently occurring faults are of primary interest (section 3.5.2), and since the cost of defects increases tremendously the later they are discovered (section 3.5), the number of TRs as discovered in the later stages of development and operation should be an acceptable measure of the quality of the system.

\(^1\) ... and only approximately solvable in, for example, the least squares sense.
Another question is whether the number of TRs is a measure that is noticeably
dependent on complexity. Here, there are actually two underlying assumptions:

1. The number of faults in the code is significantly dependent on the complexity
   of the code.

2. The number of TRs is a good estimate of the number of faults.

The second assumption will be discussed in this section, while the first will be
dealt with in section 6.4.

6.3.1 Uncertainties in the TR statistics

The first thing to cast uncertainty over the TR statistics was the plots of the dis-
tribution of TRs over different releases, such as those in figure 5.1 in the previous
chapter.

The expectations had been for a more even distribution of defects. There should
probably be peaks in the number of discovered defects just after redesign and mod-
ification, followed by a decline as defects are being corrected and the code matures.
Something along the lines of the hypothetical distribution in figure 6.1.

![Figure 6.1](image.png)

**Figure 6.1.** Hypothetical defect distribution over different releases

In reality, however, the defects turned out to be very unevenly distributed, with
a few releases having very many defects while the rest had none at all. There can
be a number of reasons for this, some of which are discussed below.

The primary consequence of the uneven TR distributions was the decision to
consider the entire TR history of the LM as a measure of quality. This was a
decision which certainly came with its own share of problems. (See section 6.2.1.)
However, even when considering the total number of TRs from all releases of the LM, it is still possible for the TR count to be a poor estimate of the number of faults in the LM. A number of reasons for this are also discussed below.

**Trouble reports unreliable indicator of number of faults**

As explained in section 4.6.1, the number of “resolved TRs” was used as a defect count in this investigation. The hope was that this would be an acceptable approximation of the number of faults, but that is not necessarily the case. The relation between faults and trouble reports can be rather complicated, as is illustrated in figure 3.2.

There are certainly faults which have escaped detection up until now, and if the detected faults have not been analysed correctly, or the TR cancelled correctly, it could also be the case that one fault is responsible for more than one TR. Furthermore, it is possible that a single TR is the result of several distinct faults, or that the TR corresponds to no real fault at all.

**Suggested solution** The deceptively simple answer is to not count TRs but to count faults instead. The problem is that faults are not so easily counted.

The first thing one can do, is to make sure only real faults are counted. Once. For this, it is going to be very helpful to be able to distinguish between the concepts of faults, failures and changes, as suggested in section 7.4.3.

The second thing to do, is to try and improve the estimate of the total number of faults in the module, by considering defect detection effort, reporting practices and population estimation methods, as suggested in the beginning of section 7.1.4.

**Different releases poor indicators of usage**

As mentioned in section 3.5.6, different releases are not a particularly good measure of “how many chances the system has had to fail”. Some releases may have been tested and used extensively, while others have hardly been tested and used at all. This could possibly explain the seemingly uneven defect statistics.

**Suggested solution** Using a time line which represents real time instead of the number of releases would probably make things look a bit better, but the preferred thing would be to have a scale based on actual usage/testing information.

**More a function of testing than of quality**

If one wants to draw conclusions about quality from defect statistics, it is necessary to untangle the two concepts of testing effort and quality, or one may end up measuring the wrong thing.

Resources are always limited, and this is certainly also the case when testing. Referring to the discussion in section 3.5.6 again, it would be necessary to test a bigger, more complex module more than a smaller one, in order to achieve an equally good estimate of the quality of the modules. But if resources are scarce, the testing
effort is likely to be more dependent on the amount of resources available, than on properties of the module in question.

If it is the case that the testing has been insufficient (in relation to the size or complexity of the module) and mostly the same over time, one would expect the tests to fail in the first release, but once they have been corrected, the component should pass through without problems. Modifications in the following releases are probably small enough not to break any of these relatively few tests. Then, at a later time, the testing is revised with new tests which will fail and cause a new spike in the TR statistics. The TR statistics are in this case more an indicator of when, and how, the testing has been performed, than of any underlying quality and defect density.

It is not necessarily the case that the testing must have been “bad” in order to produce unreliable TR statistics, it is enough if it has been different over time and between design teams. Previous defect detection activities may have affected the defect counts, and since defect data is not collected through all of development, it is difficult to say how much of the defect data is missing in the statistics.

**Suggested solution** Document the effort spent on testing and other defect detection activities. Use this information to help interpret defect statistics and to make sure the testing effort is appropriate in relation to the size and complexity of the component.

**More a function of reporting practice than of quality**

As already mentioned above, it is not necessarily the testing that has been limited. The defect statistics are equally much a function of reporting practice as it is of testing effort. There are surely activities which uncover defects that are simply fixed without ever showing up in the statistics. This is especially likely in the earlier design phases, where tester and developer are working intimately enough to make the overhead of the TR-system unnecessary. The defects that do show up in the statistics may just be the odd few which make it to the later stages of testing and they would be a poor estimate of the total amount of defects which has been found.

Now, one could argue that these “late discovered defects” are the costly and important ones, and the rest are not of much interest. It is, however, difficult (or impossible?) to characterise what makes a particular defect “dangerous”, and it could very well be that the only way to prevent them, is to prevent all defects in general.

**Suggested solution** Make sure it is clear what kind of defects should be reported and in which phases. In phases where defects are not reported, the effort invested in defect detection activities must still be documented.
Load modules are of too diverse quality

Since most load modules are of relatively big size, they contain code of many different kinds, serving different purposes, produced at different times by different people, and which is likely to be of very different quality. However, considered as a single entity, the average quality of most LMs tend to be pretty much the same. Looking at the diagrams in chapter 5, one can see that the defect density values of the LMs are, with a few exceptions, mostly the same.

When considering such large quantities of code as a single entity, the individual differences in the code are evened out over the entire LM, and much of the discriminative power is lost.

Suggested solution  Measure quality on a more detailed level, where greater individual differences can be observed.

6.4 The complexity measure

The first of the two assumptions mentioned in section 6.3 was that complexity causes defects. To be precise are there, yet again, two underlying assumptions:

1. The number of faults in the code is significantly dependent on code complexity.

2. Cyclomatic complexity is an acceptable measure of code complexity.

It is possible that one of these, or both, could be false, and different reasons for this are discussed below.

6.4.1 Uncertainties in the complexity measure

Many defects not dependent on complexity

While it seems intuitively plausible that a higher complexity should result in a code more prone to defects, it may be the case that complexity is not the dominant cause for defects. One can imagine a number of other factors contributing to defects that are not related to code complexity. For example:

- Carelessness.
  No matter how simple the code, there is always a chance of making a mistake anyway, such as forgetting to check for a certain condition, or overlooking a necessary change when copying and pasting code.

- Poor design documentation.
  Specifications and other design documents that are faulty or unclear (not up to date?) may cause the designer to produce faulty code even though he or she is following the specifications carefully.
• Communication problems.
  Poor communication between team members or between design teams may be the cause for misunderstandings and faulty code.

• Poor understanding of principles.
  It may be the case that a certain functionality, mechanism or algorithm is poorly understood and is thus implemented erroneously.

• Stress.
  Insufficient resources, tools and personnel to carry out a certain task will undoubtedly contribute to more mistakes being made.

• Inappropriate design.
  It may be that the design of a component is difficult to maintain and to modify, and that changes made to many parts of the system will have far reaching consequences that are difficult to foresee. The design may have been appropriate at one time, but has since then become obsolete and in need of redesign according to the newer requirements.

It could be that these other factors are far more influential than complexity, and unless they are taken into consideration in the experiments, we will only observe the seemingly random variations in these factors instead of the complexity.

**Suggested solution** One could try to either:

1. Account for the other factors in the measure of fault-proneness.

2. Focus on a subset of the faults that are more dependent on complexity.

The second option, to focus on a subset of the faults, is probably the most viable approach.

**Cyclomatic complexity poor measure of “real complexity”**

Real cognitive complexity is, of course, dependent of many different factors besides control structure. One can easily think of a number of other aspects which contribute to the “real complexity” of the code. For example:

• The choice and complexity of data structures.

• Interactions and interfaces between different parts of the code.

• Readability of the code, including such things as:
  - Naming of identifiers.
  - Placement of functionality.
- Quantity and quality of comments.
- Style of formatting code.

- Appropriateness in the choice of design for the problem.

**Suggested solution**  Difficult problem. If I had a good solution for this one, I’d already be famous. It is possible that one could improve the complexity measure by including a few other factors in the measure, but to do that correctly would be a very difficult task.

**Cyclomatic complexity unsuitable for LMs**

Really a part of the above fact, that cyclomatic complexity fails to consider all aspects of code complexity, is the possibility that cyclomatic complexity may be unsuitable as a complexity measure for load modules.

As discussed in section 4.5, there are additional complications when considering the complexity of bigger entities such as a LM. A LM which consists of many smaller entities, such as functions and data structures, introduces several new aspects of complexity. It may be the case, that cyclomatic complexity fails to differentiate between complex and non-complex LMs.

Cyclomatic complexity does, for example, not take into account the interactions between different functions and data objects inside the LM, and can thus not reasonably be expected to explain faults due to improper interactions. This group of “interface faults” is also a quite common one. Hatton reports some 25% of all statically detectable faults in a large population of C code to be “interface faults” [14].

**Suggested solution**  Also a difficult one. Perhaps try to include a contribution to the complexity due to interactions in the LM, such as Fan-in/Fan-out?

**Load modules are of too diverse complexity**

Another problem with LMs is that, due to their great size, they contain code of all kinds of complexities, from the most simple to the most complex. When such large amounts of code are considered as a single entity, the individual differences between different parts of the code are evened out. On average, most LMs tend to look pretty much the same.

**Suggested solution**  It should be easier to observe differences in the code if the complexity is measured on a lower level. Preferably on the level of individual functions.
6.5 How to specify the location of a defect

The thing which probably had the greatest impact on the project and the results, was the fact that all defects (the trouble reports) in the TR-system were attributed to either a load module, a subsystem or a system, all of which are of great size. A load module may consist of tens of thousands of lines of code and several hundred functions, while a subsystem encompasses a whole score of LMs, and a system is just unbelievably huge.

Since cyclomatic complexity is measured on functions, it would have been preferable to also have the defect data on individual functions. Admittedly, it is almost always possible, at a later time, to deduce precisely what changes were made due to the correction of a specific fault through manual inspection of the version history and source code. That is, however, a substantial amount of work if it is to be done for more than a handful of faults, and something which normally only is motivated by serious problems. A better traceability would be a very valuable thing to have for more continuous evaluations.

6.5.1 The Load Module

The most detailed information on the location of a fault that was readily available from the TR-system, was then the Load Module. (See section 4.1.1.)

The major concern with using a LM for specifying the location of a fault, is that a LM is not code: it is a binary object derived from code. Nevertheless, since virtually all code from an SWB is used in one or more LMs somewhere, one might be tempted to say that the LMs and the SWBs both represent the code, albeit on different conceptual levels (see figure 6.2). This may have its justification, but in

![Figure 6.2. Relation between SWBs and LMs](image)

using a derived object for referring to code, one is obviously adding an extra level of indirection which gives rise to a number of difficulties:
• A LM may contain code from various origins.
  It is unclear which code is actually responsible and a particular design organ-
  isation may get the blame for defects which aren’t their fault.

• Several LMs may be based on the same code.
  In this case, several LMs will suffer the same problems, even though they were
  maybe only discovered and attributed to one of them.

• A “new” release of a LM may not contain much (if any) new code.
  Apart from these “problems of representation”, the LMs also proved to be prob-
  lematic from a measurement point of view. The large amounts of code bundled
  together in a LM is likely to obscure differences that exist in size, complexity and
  quality between different parts of the code. They also improperly dictate the objects
  of measurement and severely limit the number of measurements from which to draw
  conclusions.
  It is, of course, still very interesting to know precisely which executable object
  have failed when reporting a failure, but when trying to specify where in the code
  the location of a fault was, the executable object is a poor choice.
  What would be needed is something which actually specifies the code, or the
  area of code, that contained the fault, instead of the executable object based on this
  code that happened to fail.

6.5.2 The Software Block

An entity which does designate an area of code, is the Software Block or SWB.  
(See section 4.1.1.) There would be some advantages to specifying faults on SWBs
instead of LMs:

• An SWB explicitly designates an area of code.

• Specifying the SWB would be an improvement in traceability for faults which
  earlier used to be reported on the subsystem or system level.

  Most of the SWBs are, unfortunately, on about the same level of size as the
  LMs, and will in most cases not give much of an improvement in the traceability
  of the faults. For example, having fault data on SWBs would not have helped this
  particular investigation much. Furthermore, the SWBs as of old are at the moment
  of writing being phased out and replaced by the new Blocks. (See section 4.1.2 and
  6.5.4.)

6.5.3 The Software Unit

The Software Unit, or SWU, is another object which also represents actual code,
but it is smaller than an SWB. (See section 4.1.1.) An SWU usually corresponds to
an object (or a few) on the design level, and is in the order of a few thousand lines
of code in size.
If the locational data is to be really useful to the individual design section or team, I believe it is necessary to have the locational data on a level equivalent to SWUs or better. Knowing to which SWU the faults belong would provide much better detail when analysing different design practices, coding conventions, code attributes (such as complexity), and for locating troublesome areas within an SWB. Being able to draw conclusions on a level that is useful to the designers is important, since it is mainly their effort of collecting data which makes the analysis possible in the first place.

One can, on the other hand, argue that such detailed information is not really necessary, since the designers are already familiar with their own work, its intricacies and problems. There can also be a certain comfort for the individual designer not to have faults traced to code with such detail. (See section 7.5.)

The main problem with SWUs, however, is that they are no longer recognised in the new product structure. The smallest manageable entity in the new product structure will instead be the Block. (See section 4.1.2 and 6.5.4.) This means that in order to report faults on SWUs in the future, they would have to be re-introduced into the new product structure. One also needs to consider whether the SWUs are entities of sufficient permanence and stability to warrant formal recognition, or if they must be expected to be created, deleted, merged or divided too often to be of use.

6.5.4 The new Block

With the introduction of the new way of working and the new product structure, the concept of a new Block is also introduced. (See section 4.1.2.) These Blocks will replace the old structure with SWBs and SWUs under each subsystem, serving as a sort of “container” for a deliverable product, typically a LM, and its associated code.

Advantages of the Block

The obvious advantage of the Block is that it is the smallest manageable and deliverable entity in the new product structure, and will, in all likelihood, be the way the source code is organised in the future.

Unlike a LM, but similar to an SWB or an SWU, a Block is an implementation entity by which the code is organised. Since all code in the new product structure has to be organised into Blocks, the collection of all code bearing Blocks constitutes a partition of the code base in the system. That is, every piece of code belongs to exactly one Block which uniquely represents the code.

Formal recognition in the product structure with unique product numbers will, furthermore, make the Blocks easy to identify and should allow for straightforward implementation in the TR-system.

The Blocks will also, in most cases, be a bit smaller in size than an SWB or a LM, and will thus allow for slightly better traceability.
Disadvantages of the Block

Using Blocks as objects for reporting faults is also associated with a few complications, mostly due to the variability of what constitutes a Block. A Block, unlike an SWU or a LM, is neither code nor derived object, but rather a container for both.

The mapping between code and deliverables used to be rather messy earlier, with complicated many-to-many relationships between code bearing SWUs and derived objects such as LMs and IFs. One of the objectives of the Block, was to establish a closer relationship between the underlying code base and its derived build result, allowing for easy one-to-one mapping between code and build result.

This concept of treating code and deliverable as a single unit would not pose a problem for reporting faults on Blocks, if a LM or library always was accompanied by its source code. Unfortunately, there seems to exist a number of different types of Blocks, many of which do not carry their own code base, or only a very small amount of it. (See figure 6.3.)

![Diagram of different types of Blocks]

The problem with this is that it allows a choice in how to specify the location of a fault. Either by using the Block of the failing executable, or the Block of the responsible code, and they need not be the same! Essentially, this is the difference pointed out above between reporting faults on a LM or an SWB.

The division between “Blocks bearing code” and “Blocks bearing deliverables” is particularly prevalent in a few subsystems, where most of the old LMs seem to have been directly assigned to new Blocks. They contain little, if any, code of their own and draw virtually all of their code from a few larger Blocks (the former SWBs). In practice then, we end up with the same situation as before: complicated many-to-many relationships between the code and the deliverable products, with the added inconvenience of now having to differentiate between many different types of Blocks.

This variability in the type of the Blocks, is presumably also responsible for the great variability in the size of the Blocks that can be seen. There are Blocks ranging from a few hundred or thousand lines of source code, up to around 50000, or even as much as 170000 lines of code in one case. Clearly, knowing that there has been a particular number of faults somewhere in these 170000 lines of code is about as useless for diagnostic purposes as it has ever been.
In the end, the Block confronts us with almost the same problem as the LMs above: we are still not referring to actual code, but rather to an entity related to code, i.e. a container which may contain the source code of the deliverable. This problem can be avoided by demanding of the person correcting the fault, always to specify the correct “code bearing Block” in which the fault was found and corrected. However, I suspect it will be difficult to ensure that this is always done.

6.6 What is really software quality?

When discussing software quality back and forth, inside and out, like this, it is probably a good idea to ask oneself: What is really meant by software quality?

The question is by no means an easy or obvious one. As mentioned in section 3.3.3, software quality consists of many different aspects, and even if we restrict ourselves to the narrower view of quality as “absence of defects”, the question still remains somewhat difficult.

6.6.1 Intrinsic quality or resulting quality

Suppose there are two modules, M1 and M2, of the same size. (See figure 6.4.) M1 has got few defects to begin with, but gets very limited testing before being put into use. M2 has got many defects to begin with, but undergoes rigorous testing and correction, reducing the number of defects to less than that of M1. Which one of M1 and M2 can be said to be of the highest quality?

![Diagram showing intrinsic and resulting quality]

The answer to that question depends on who is asking. For the customer using the product, it is obvious that M2 is of higher quality; he only cares about the final
product delivered to him. For the development organisation, it is obvious that M1 was of higher quality, since M2 surely caused them a lot of grief while testing.

I am, of course, speaking of the quality at two different times: the quality right after design and implementation, and the quality at the time of release. I will use the terminology saying that M1 had a higher intrinsic quality, while M2 had a higher resulting quality.

The intrinsic quality is the quality the product will have straight out of design and implementation. It is only dependent on the quality of development and completely independent of any testing or defect detection activities. A better intrinsic quality obviously provides a better starting point for reaching the desired release level quality, with lower cost and less resources. Being able to predict the intrinsic quality of the product will also allow for better cost estimates and allocation of resources early in planning.

The resulting quality is dependent both on the initial intrinsic quality of the product as well as any effort spent on defect detection and correction activities. In theory, it is always possible to achieve a good resulting quality through rigorous defect detection and correction activities. In practice, it can be the case that the resources necessary to spend in order to do so are absolutely prohibitive.

To know that your product is of a high resulting quality, is both important and reassuring. You can feel confident that the customer will be happy and that you will have few problems maintaining the product. Some may even argue that this is the only important thing, but I consider this to be an oversimplification of the situation. Much of this report has been about ways to evaluate, predict and control the intrinsic quality. In the end, this will also affect the resulting quality, since better starting conditions are likely to produce better final conditions. Most people will surely agree that it is preferable to be able to produce something which will be of good quality from the start, without having to go through a “baptism of fire” in the testing phase. In short, one might say that working to achieve as good intrinsic quality as possible, is the same as “building in quality” instead of “testing in quality”.

95
Chapter 7

Recommendations

This chapter contains recommendations for changes and improvements based on the experiences from this project. Most of the suggestions will be aimed towards improving the TR-procedure.

Suggestions for the TR-procedure and for collecting defect data can be found in section 7.1. In section 7.2 there is a short discussion about the collection of project data. Section 7.3 has some recommendation on how to use cyclomatic complexity. Section 7.4 contains a number of other recommendations, while section 7.5 contains advice on how to make a software metrics initiative successful. Finally, section 7.6 provides a summary of the recommendations in this chapter, categorised as high, medium and low priority.

7.1 Defect data

In order to prevent and detect faults, we need to study them. In particular, we want to learn what we can about the ones that cause the most problems. How can we prevent them? How can we detect them? How can we correct them?

"Know your enemy and know yourself", it is said. Knowing and characterising the errors you usually make will help you avoid them. For example, performing tailor-made tests to catch the pesky bugs you know always elude you.

7.1.1 Tracking the faults

In trying to prevent faults, it is of obvious interest to document something about their origins. Where were they introduced? When were they introduced? Why were they introduced?

This kind of information is, of course, only available after the fault has been analysed and is unavoidably subject to a certain amount of speculation and guesswork on behalf of the analyst. The time and phase of introduction and the cause of the fault may be especially difficult to determine if a long time has passed since the implementation and if the person responsible for implementing the code no longer is available for consulting.
Where the fault was located

The location of a fault is information that is of fundamental importance. The ability to trace a fault to its actual location allows you to assign responsibility for the fault to the correct code, component or design team. The usefulness of virtually all other fault information, such as type, cause, severity, and time of introduction, is affected by how the location of a fault is specified. The level to which faults can be traced determines the level of detail on which analyses can be made and conclusions can be drawn based on the fault data.

Following the discussion in section 6.5, a LM is not a very good candidate for specifying the location of a fault. It is, of course, still necessary to identify the executable product that failed, but the more precise location of the fault in the code should also be documented. The entity on which to report faults should preferably be one which explicitly designates code and that is sufficiently small to allow interesting analyses to be made.

Recommendation The Block is the obvious choice for specifying the location of a corrected fault, since it will be the smallest manageable and identifiable entity in the new product structure. Despite being an aggregate entity which does not exclusively designate code, a Block is a better choice for reporting faults than a LM, in that it is an implementation entity which contains its own unique code. Thus, as long as the Block which contains the code in which the fault was found, always is specified correctly, there is no ambiguity in tracing the fault to code. I cannot say, however, how it is to be ensured that this always is done.

Something which greatly would reduce the risk of confusion, is if the actual code in a Block was productified. If the code in a Block can be uniquely identified as a separate entity, it is possible to demand that a TR must terminate in such an entity and there can be no confusion as to which entity is to be held responsible. If a Block does not have any code of its own, it will obviously not contain a code product, and this will signify the Block as being non-code bearing and prevent faults from being reported on it.

An even better solution would be if it was possible to identify the code in more detail than is allowed by a Block, for example, if entities such as the SWUs of old were introduced as subdivisions of the code in a Block. The better traceability of defects should give much better discriminative power and allow for more detailed analyses to be made. It may, furthermore, be of interest to, for example, separately identify the test code belonging to a Block.

Implementation There exists a field “Corrected in product” in the TR-system which at the present is mostly redundant, since it usually contains the same information as the “Found in product” field. The “Corrected in product” field, or a new field (appropriately named “Corrected in Block” or similarly), could be used to store information on the location of the fault in code. It should be a mandatory field and give a drop-down list of valid choices for Blocks or SWUs.
Detailed information on the location of the fault such as this, can not be supplied until the problem has been thoroughly analysed. When first reporting the failure, the best one can expect is for the individual reporting the fault to specify as precisely as possible which executable product has failed. At the time of correction of the fault, however, it is known1 precisely where the fault was, and the information could be supplied by the engineer responsible for the correction as he or she changes the state of the TR from *Planned* to *Implemented*.

**When the fault was introduced**

Another vital bit of information that is needed in order to draw good conclusions, is what time, in what phase, during which activity, the fault was introduced. This information is necessary to have in order to correctly attribute the faults to the project, release or activity which actually introduced them.

Information that is of interest to record would be:

1. The release in which the fault was introduced.
2. The phase of development which introduced the fault.
3. During which activity the fault was introduced.

**Implementation** The TR-system contains fields to specify the release in which the faults was *found*, but not for when it is believed to have been *introduced*. A number of fields would have to be introduced to collect the above information.

**Why the fault was introduced**

Documenting the reasons for the faults is obviously a sensible and important thing to do when one is interested in reducing the number of faults. Knowing the cause behind the fault makes it possible to take actions to avoid the problem in the future.

It would, for instance, be of great interest to know how many faults are introduced due to the (attempted?) correction of another fault, due to changes in other parts of the system, or due to flaws in external products, such as hardware, firmware or supporting libraries. If many faults are found to be because of coding mistakes and poor constructs, it may be worthwhile to introduce an enforceable coding standard2. If many faults are discovered to be because of incomplete specifications or requirements, it may be worthwhile to introduce more rigorous control of specifications, for example, a formal specification language which automatically can be checked for consistency.

**Implementation** At the present, there is no field in the TR-system intended to hold information about the cause of the fault, and thus a field “Cause of Fault” or

---

1 Or at least believed to be known.
2 I.e. does not compile unless compliant with standard
similar will have to be introduced. This should present a drop-down list of legal options. The options for cause would possibly have to be dependent on the type of the fault.

7.1.2 Detecting the faults

Information on the cause and introduction of the fault is aimed towards the prevention of faults. In order to help evaluate the detection and correction of faults, one should also gather information related to the manifestation of the fault.

When the fault was detected

Documenting when the fault was detected should be straightforward. This information is useful in order to determine in which phases of development faults are being found.

Information that is of interest to record would be:

1. The release in which the fault was found.

2. The phase of development in which the fault was found.

Implementation In the TR-system there exists a field “Found in baseline/delivery” that already is used for documenting the release in which the fault was found. It is clearly useful, but could be improved to better specify the precise delivery. This will require a naming/versioning convention to be adopted that is consistent and meaningful throughout the entire development organisation.

The “Origination” field, which contains the instance which reported the fault, provides some indication of the phase of development in which the fault was found, but a dedicated “Phase” field with clearly defined values would be preferred. Examples of phases could be, system design, coding, design test, subsystem integration, operation, etc.

Information on when the fault was detected is suitably provided by the person reporting the fault.

How the fault was detected

As with the reason for introducing the fault, the “reason for its discovery” should be unambiguously categorised after which kind of activity or test that uncovered the fault. This kind of information is useful for evaluating the testing organisation and its activities.

Implementation There already exists a field “Activity” in the TR-system which could be used for this purpose by carefully defining the different activities which may uncover faults. Examples of activities could be inspection, component testing.
compilation, static analysis tools\textsuperscript{3}, stress testing, etc. See section 3.5.4 for more information of detection techniques/activities.

There also exists a mostly unused field, “Test case”, which could be used to document more detailed information on the testing which uncovered the fault. The field could either be used to specify the exact test case which failed, or in a more general sense by defining categories of test cases that would be useful for evaluating different testing practices. Examples of categories could be \textit{normal traffic test}, \textit{maximum traffic load}, \textit{restart scenario}, \textit{normal configuration}, etc.

The information on activity or test case is also suitably provided by the person reporting the fault.

### 7.1.3 Classifying the faults

Besides the information on the introduction and discovery of the fault, there is also other information that is interesting to collect, for example, the \textit{type}, \textit{severity} and \textit{difficulty of correction}.

**The type of the fault**

The fault should be classified according to what kind of fault it is. This is interesting to know in order to determine if certain groups of faults are more or less prominent in some aspect. Perhaps there is a certain type of fault that is very common or particularly serious?

**Implementation** There is currently no field in the TR-system to hold fault type information, and such a field should be introduced. The \textit{“IEEE Standard Classification for Software Anomalies”} \cite{IEEE} contains an extensive classification scheme for fault types. This scheme is probably too detailed, but will be useful as a basis for deriving a more appropriate type classification scheme.

It is not possible to submit type information before the problem has been analysed, and the information is thus most appropriately supplied by the analyst.

**The severity of the fault**

The severity of the fault is the impact the fault has on the product and for the users. The severity is an important thing to know in order to determine how to resolve and prioritise the problem.

There are three levels of severity defined in the TR-system:

1. A – Stopping Fault
2. B – Major Fault
3. C – Minor Fault/Enhancement

\textsuperscript{3}Such as \texttt{flex} or \texttt{lINT}
This is a fairly coarse categorisation and one may consider introducing a finer one. The appearance of such unofficial classifications as “Hot Hot 3.2” in addition to the normal severity classification may be an indication that additional levels of severity should be introduced, or that a separate prioritisation scheme should be used in addition to the severity of the fault. Even a quite benign fault could be a high priority at times.

**Implementation** The implementation of additional severity or priority levels in the TR-system should be straightforward. The only difficulty would be in defining the different levels unambiguously.

Severity information is most appropriately supplied by the individual reporting the problem.

**The difficulty in correcting the fault**

As mentioned in section 4.4.3, the TR turn-around time may not be a very accurate measure of the difficulty in correcting the fault. A better solution may be to explicitly specify the difficulty or the approximate amount of time invested in correcting the fault.

It could be of interest to know the difficulty of, or the time needed to:

- Recreate the failure.
- Locate the fault.
- Correct the fault.
- Verify the correction.

**Implementation** The TR turn-around time is already available from the TR-system, but if more specific information on time or difficulty is to be gathered, it will be necessary to introduce a new category of information and a number of new fields.

Information on the various difficulties would have to be supplied by the individual responsible for the respective activities: the tester, product/area responsible or designer.

**7.1.4 Validity of defect data and conclusions**

Good conclusions are dependent on having good and reliable data. What it means for the data to be “good and reliable” does in turn depend on what kind of conclusions one is interested in.

For example, in order to determine the most common cause for faults during a particular project, it is necessary to have a sufficiently large number of faults to give a good statistical accuracy. This is a requirement that shouldn’t prove very difficult to fulfil. On the other hand, if the objective is to estimate the total number
of (residual) faults in a module, it is necessary either to take into account the defect detection effort, or to use techniques such as fault seeding or fault pooling, in order to obtain some kind of estimate of the “population size”. This is necessary even if the defect data only is to be used for comparing the quality between different modules, and it is a significantly more difficult problem than simply ensuring sufficient sample size.

Recommendations for how to have consistent reporting and how to estimate testing effort is given in the two following subsections. For a description of fault seeding or fault pooling, see [22].

Putting statistical considerations and unchecked influential factors aside for a moment, it is of course also important that the data is correct. Correctness of the data will be discussed in the third and last subsection below.

**Having consistent reporting**

It must not be the case that defects sometimes are reported and sometimes simply fixed. If the reporting ambition is different in different phases of development, one must make sure only to compare the data from phases in which the ambition has been the same, lest the reporting will skew the defect data.

This should not be particularly difficult to achieve as long as it is clearly specified in the development process, when and what kind of defects are to be reported. Any changes or deviations in the reporting practice should be documented clearly in the project data.

**Having consistent testing**

If the testing or usage has been different in some phases between the compared products, one must be aware of this and maybe try to adjust the results to make the comparison fair. This makes it necessary to document what defect detection activities have been used for which product and when, together with other project data. For a delivered product, some data on the amount of usage is needed.

To ensure that the testing has been comparable, some way to quantify or measure the testing effort or usage is needed. When structural testing is performed, such as in component test, different measures of so called “code coverage” can be used. For example, that 100 % statement coverage and 80 % branch coverage should be achieved in component test. For functional testing performed in the various integration and verification activities, another measure of the testing effort must be used. The Functional Description (FD), Verification Specification (VS) and Verification Record (VR) may, for example, be used in some kind of measure of how well the VS explores the FD, and how well the VS was executed in the VR. Other factors of interest may be:

- How many of the interfaces were tested?
- Has stress testing been performed?
• How much negative testing and fault scenarios have been exercised?

Perhaps it would be appropriate to introduce some kind of Verification Quality Rank in addition to the SQR? The important thing is that some kind of acceptable measure of how much and what kind of testing has been performed in relation to the size/functionality of the component.

As was mentioned in section 3.5.6, the testing effort in previous phases (and releases) can very significantly affect the test results in the following phases (and releases). While it may be unreasonable to expect detailed defect data to be collected through all of design, it is still necessary to document the defect detection activities that have been used and the effort invested in them, in order to be able to draw valid conclusions from the defect statistics.

Validating the data

The quality of the data is of great importance, and having faulty and misleading data may be even more dangerous than having no data at all. Lack of data is at least correct in that it signifies ignorance of the situation, whereas faulty data give a dangerous illusion of knowledge which may lead to wasted or even harmful actions. As an illustration of the problem, an investigation by Basili and Weiss [4], found that about half of the data collected was corrupted in some way.

When considering the importance of having correct data, it may be worthwhile to have some sort of validation step before the data is accepted. Some kind of central collection point where all data is reviewed for consistency and completeness, or at least someone who reviews the supplied information in a reasonably objective way.

7.2 Project data

There has been much talking in this report about defects, their attributes and the process of documenting them. Yet, all of this is still but one side of the problem. The defect data can be thought of as a measure of our success or failure. It is the observable outcome of our efforts, the output resulting from our input. (See figure 7.1.) But what about the input, then?

Suppose now that we have managed to do really well. We are standing there with a finished project and a product that has a documented good quality. Apart from the satisfaction of knowing that the product really has good quality, we would now be interested in knowing how to do it again and persistently in the future. So how did we do it?

The entire objective of documenting and evaluating the outcome, is to be able to relate it to methods, techniques, tools, and effort, to determine what works and what does not, and then use that knowledge to produce better software consistently. Thus, there is a need to document what is being done during the course of a project, so that it later can be related to the success or failure.

Information of interest could be:
Figure 7.1. Inputs and outputs of the software development process

- The things produced:
  - What products?
  - How much code? New? modified?
  - How many documents?
  - How many tests?

- The time or effort put into the project:
  - How much effort on prestudy/feasibility?
  - How much effort on design?
  - How much effort on implementation?
  - How much effort on testing?

- The tools:
  - Which tools were used?
  - On what? To what extent?

- The methods:
  - Which defect detection techniques were used?
    * Inspections and reviews? How much? On what?
    * Static analysis tools?
    * Formal specification language?
  - Which implementation language?
– What kind of design rules? Coding standards?

The importance of one kind of data, in particular, has been emphasised several times already in this document: *testing effort*, or more generally *defect detection effort*. It is important that the defect detection effort is documented during all phases of the project if the defect data is to be useful for evaluating quality and for making comparisons between products. Another kind of data that is of special interest when trying to evaluate product quality, is the things produced: How much code is produced? How much is new, modified, or unchanged?

Ideally, it should be possible to use the data collected and construct a “road map” of how the project was executed. What was done and when? The effort, the decisions, the tools and the methods. It should be noted, that the information must reflect how the project *actually was* executed, not how it *should have been* executed. The information could appropriately be assembled by the project manager or someone with similar insight into the details of the project. Designers or design teams report to the subprojects, whose manager assembles the data and reports upwards towards the topmost project.

It seems that, at the present, not much project information of this kind is being collected and used. A system for this is planned to be introduced sometime in the near future, and I recommend some careful thought on what kind data should be collected and how.

The focus of this report has not been on project data, but rather on product data. For more and better information on project data and management, I recommend a good book on the subject.

### 7.3 Cyclomatic complexity

Despite not finding any empirical evidence of the usefulness of cyclomatic complexity as a predictor of fault prone modules, the cyclomatic complexity is still a readily available metric with an intuitively valuable content: the amount of decision logic in a module. It is possible that a future investigation, hopefully made possible by some of the suggestions in this report, will reveal better guidelines for its use, but until then, and since SQR level 3 states that cyclomatic complexity should be measured in any case, I will here give some general advice on how cyclomatic complexity can be used.

**As a guide for the number of test cases needed**

The cyclomatic complexity is the minimum number of test cases needed to test all paths through the module independently. (See section 3.6.) As a measure of the amount of decision logic in a function, the cyclomatic complexity of a function is an indicator of the number of test cases needed to fulfill the structural testing criterion according to McCabe, as well as the simpler branch coverage criterion.
In their paper, Watson and McCabe presents a *Weak structured testing* methodology, in which simply $v(G)$ number of different paths have to be executed while simultaneously satisfying complete branch coverage [29]. One could imagine using an even weaker version\(^4\) of structural testing according to McCabe, in which only $v(G)$ different test cases have to be executed.

If used for nothing else, one can at least draw the conclusion that a module with a number of test cases much less than its cyclomatic complexity has not been well tested.

**As an alternative measure of size**

The sum of all cyclomatic complexity in a component can also be thought of as a measure of size: the total amount of decision logic in a module or component. While regular LOC metrics only take into account the “external size” of a component, the cyclomatic complexity also takes into account the internal structure. One would therefore expect the total “cyclomatic complexity mass”, $\sum v(G_i)$, to be an equally relevant measure in many situations where LOC metrics normally have been used.

**For surveying large volumes of code**

There is a huge amount of code being written all the time, and even the programmer responsible for writing the code may sometimes find it difficult to remember what he did a year or two ago. Even more so when code is being moved around between different design organisations, and a new designer finds himself confronted with a large amount of completely unfamiliar code\(^5\). Cyclomatic complexity may in such a situation be used as a guide on where to focus attention. The (usually) short list of functions with the highest cyclomatic complexity should be the ones most likely to contain important functionality and the most important to focus on.

**As a guide for redesign**

While a high cyclomatic complexity does not necessarily have to be a bad thing, it is still an indicator of where a messy control structure could be. As with the above, the list of functions with the highest cyclomatic complexity may serve as a starting point, or as additional input, when considering the functions which are the most appropriate for redesign.

### 7.4 Other suggestions for improvements

**7.4.1 Establish a company baseline**

One of the primary goals of routinely collecting data and assembling results, is for it to provide a historical background to which newer data can be compared. Historical

---

\(^4\) *Foolish structural testing*?

\(^5\) ...painstakingly commented in Hungarian.
data can be used for validation of current projects, to make improved estimates for future projects, and it opens up the possibility to use Statistical Process Control for monitoring the development process. (See [25, section 5.4].) For example, one soon learns what is the normal amount of defects found in a certain phase of testing. Deviations from “the baseline”, for better or for worse, are good candidates for further investigation.

In order to establish a company baseline, one needs to collect, analyse and update the chosen data regularly. Data can be extracted from already existing systems, such as the TR-system, fault and project databases, or specifically collected for this particular purpose. The data to be chosen for monitoring may vary depending on the objective, but something such as (pre-release and post-release?) defect counts/densities and their type and cause, coupled with the necessary project data to assure their validity, would be a good candidate.

When the amount of data is not so overwhelming, it may be sufficient to simply extract the relevant data from the product or project and update a spreadsheet, but if the ambition rises to monitor more data and more products, some kind of database backed system will certainly perform better.

7.4.2 Automate measurements

As would be obvious by now, there are a great many things one could measure. Unfortunately, there is a price to pay in time needed to document these things, and the people who are performing this work must feel that their effort is worthwhile. Something which tends to make the effort feel less worthwhile, is if people have to type the same information into different systems several times, or fill out long, tedious paper forms. It is important that the collection of information is as effortless as possible.

As much as possible of the data gathering process should be automated to prevent unnecessary work. If data is entered in one system or database, it should automatically be replicated to other places where it is needed. If a separate fault database is kept, it should not be necessary to enter the same information in both the fault database and the TR-system; the TR-system should link (or at least replicate) the fault data. One can also imagine extracting locational data of the fault and updating the fault database automatically, when the source code changes are checked in. The relevant code metric measurements could also be included in the build support and automatically performed (and possibly reported?) when building the system.

7.4.3 Distinguish between faults, failures and changes

As explained in section 3.5.1, the relations between faults, failures and changes are not completely straightforward. It may be the case that the reason for the fault is yet another fault (say, in a design document), or that the correction of a fault will require changes in several places or products. It may also be of interest to keep track of which faults are responsible for the most frequent (or most severe) failures.
The main problem with TR-system, is that its primarily role is as a change management system (see section 4.3.2), and is, as such, not very well suited for use as a fault database.\footnote{Though it may very well be adapted to perform as one.}

A better solution would be to have a system (or several) which is capable of handling the separate concepts of faults, failures and changes and maintaining the relations between them. This would reduce overhead in situations where a more lightweight process is preferred, such as when faults are discovered in design or during inspections. In the case when several changes are needed to correct a fault, only the changes themselves would have to be registered in the change management system and linked to the fault they are supposed to correct.

The separate systems would only keep the data relevant for their domain, and the relations between them would be kept up by links: the changes to their fault, and the fault to its failures. It may also be possible to administrate the different system separately, and it might, for instance, be easier to achieve a better traceability of faults by having locally administrated fault databases.

7.4.4 Extend defect tracking to other design products

The source code is not the only thing produced in the process of building a system. Other artifacts, such as design documents or the testing environment, are produced in support of the system, and their performance or non-performance will also affect the quality of the system.

To include these artifacts in the defect tracking system would widen the scope of the quality improvement efforts. It would improve the ability to identify problem sources and enhance the predictive capabilities by including products from a wider range of activities and development phases.

For this to be possible, it is necessary for the other design products to be identified as products in the same way as code or load modules are. It should be possible to revision handle, deliver and report defects on, for example, design documents and the testing environment, pretty much in the same way as is done with code products.

**Design documents** Since virtually all systems have started out as a collection of requirements at one point in time, it should come as no great surprise that the design documents are important for the development of a system.

To some extent, the system can be said to be a realisation of the design documents, and any flaws in them are likely to make it into the code. Surely, there are many cases where the source code has been a correct implementation of incorrect design documents or beliefs. Brian Marick, for example, reports on a number of software fault surveys, in which pre-coding faults make up a significant portion of the total amount of faults [20].

An extension of the defect tracking to design documents would significantly improve the ability to trace problems to their origins.
Testing environment To create and maintain the test code needed to verify the system consumes a large amount of time and resources. It is not uncommon for the amount of test code to far exceed that of the product. This combined with the fact that the testing environment is closely related to the quality of the system (effectively being the mechanism whereby TRs are generated), make for a strong incentive to include the testing environment in the defect tracking system.

7.4.5 Software design quality ranks

The Software Design Quality Ranks (SQR) are a set of five ranking levels to be used when delivering software components from design to integration and verification activities. In order to fulfil the requirements of each level, there is an increasing demand on documentation, testing, static analysis, dynamic analysis, etc. of the component.

Utility The SQR rank of a delivered component can be used to help evaluate the defect data by ensuring that certain activities have been performed with a certain rigour. The demand in SQR for levels of “code coverage” should, for example, ensure that the test effort invested in the design phase has been somewhat comparable between different components with the same SQR level. This should in turn make carefully collected design defect data comparable, and, if used together with post-design testing effort, also post-design defect data.

Limitations One problem is that the SQR only considers activities prior to delivery from design to I&V. This should help ensuring a consistent quality out of design for components with the same SQR rank, but does not help with estimating post-delivery testing effort and usage.

Another problem is that, if the definition of the rankings themselves changes over time, comparisons between the “old” rank and the “new” rank will be invalid. A way to avoid this would be to make sure to document the actual activities along with the project data, instead of just the rank of the component.

7.5 How to make it work

Howard Rubin is quoted in Karl E. Wieger’s paper, Software Metrics: Ten Traps To Avoid [31], saying that up to 80 % of software metrics initiatives fail within two years. There are obviously many pitfalls to avoid and Wieger’s paper is highly recommended reading to any one who is involved in software metrics, and especially to any one who is to make decisions on this topic.

Must be cost-effective

The objective of our data gathering activities is to monitor our organisation and transform it into something more efficient. In the end, we want to lower the overall
costs, and the gains we can achieve in terms of improved quality and process control must outweigh the cost in time and resources invested in gathering the information. To start out, it is probably best to select only a few of the suggestions in this chapter, lest one fall into Wiegner’s trap number 2: Measuring too much, too soon.

For a summary of the most important suggestions, see the high priority list in section 7.6.

**Must have resources**

It is not possible to collect and analyse data without the expense of allocating sufficient resources. There must be personnel and time available to collect, assemble and analyse the data. It is also necessary to have the tools and facilities needed to measure and store the data, as well as training in how to use them.

**Must be put to use**

A closely related point to that of choosing what to collect, is that one must also make sure to actually use the data that is collected. What more, one should make sure it is clear to those who collect the data that it is being put to good use. Quick feedback of results to the organisation is important, so that those who spend time supplying the information feel their efforts are worthwhile and that it is an instrument to influence their situation.

**Must not be used to manage**

Another way in which the measurement initiative could become misguided, is if the data is used directly to manage and control the process. People will then quickly realize how to make the numbers “as management want them to be” instead of making the product the best they can. Experienced judgement and common sense should always weigh more heavily than what any metric has to say, since metrics, and especially a single metric, is more of a “hint” of the current state than the “absolute truth”.

**Must not be used to evaluate individuals**

Possibly the most important consideration here, is that one should be careful with what the collected data is used for. For example, as soon as someone suspects that the fault data reported is used by management to evaluate their performance, they will stop reporting completely or only report data which makes them look good. That is quite understandable, as no one will dig their own hole.

The data must NOT be used to evaluate, motivate, reward or reprimand individuals, but rather to evaluate changes to the process as a whole. To ensure a certain confidence, not all data should be available to everyone. Some data should be private to the individual and exclusively for their own benefit, while other data could
be private to the development team. Yet again: this is an important consideration that otherwise could render the measurements ineffective.

### 7.5.1 Who does what and when?

Here I intend to give a tentative example of how the data collection process could work in the typical case, when a code defect is found and reported in integration and verification. (See figure 7.2.)

<table>
<thead>
<tr>
<th>Tester</th>
<th>Product Responsible</th>
<th>Engineer</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Tester" /></td>
<td><img src="image2" alt="Product Responsible" /></td>
<td><img src="image3" alt="Engineer" /></td>
</tr>
</tbody>
</table>

**Figure 7.2. Who does what and when?**

1. A defect is detected in I&V. The tester who uncovered the defect creates the TR and supplies the detection/failure information, such as:
   - The product which failed.
   - A description of the failure.
   - The severity of the failure.
   - The version of the product.
   - The phase and activity which uncovered the defect.
   - The test case or kind of testing activity which uncovered the defect.

2. The TR is sent to the product responsible. He/she assigns the TR to the appropriate engineer.

3. The engineer analyses the TR and decides on a solution. This may possibly result in more TRs being written on other parts of the system that need to be corrected. Analysis information regarding the fault is added to the TR:
• The location of the fault in code.
• The type of fault.
• The cause of the fault.
• The time, phase and activity in which the fault was introduced.

4. The correction of the TR is implemented by the engineer and included in an updated release. Information related to the delivery is added:

• The effort of correcting the fault.
• The product(s) which have been affected.
• The version(s) which include the correction.

5. The solution to the problem and the information contained in the TR is reviewed for correctness and consistency by another person (possibly the product responsible).

6. The corrected product(s) are verified by the tester and the TR is closed.

7.5.2 Implementing the changes

Coming up with the suggestions in this report is probably only half the work involved in improving the data gathering process. The remaining part of the job will be to implement and realise these changes in the organisation. One can think of several ways to do this, and here I intend to give my thoughts on this matter.

Modifying the present TR-system

The present TR-system appears to be a very flexible and capable application. At the present, it is not really capable of serving as a fault database, but this can surely be improved upon.

For example, in addition to the information categories of Initial information, Analysis & decision information, Delivery information, etc. (see section 4.3.1), it would be appropriate to include a category of Fault information, where information pertaining to the underlying fault of the TR can be documented. Some existing categories would probably also have to be complemented with additional information, such as the Initial information which holds information related to the failure.

It may also be possible to modify the TR-system further to allow it to deal with failure, fault and change information separately, as suggested in section 7.4.3. The TR-system permits records which belong to different classes to have different life cycles and to hold different information. (See section 4.3.1.) By creating different classes of records for failures, faults and change requests, it should be possible to encompass the different concepts within the same system. This will obviously require much greater changes to the system than simply adding fault information to the TR as suggested above.
Introduction of a new TR-system

The TR-system presently in use is not a new system at all, and a replacement has been waiting behind the scenes for some time already. The reason for the delay seems to have been some technical difficulties, but the change from the old TR-system to the new one is likely to come eventually.

With the introduction of a new TR-system, a golden opportunity to introduce changes to the TR handling (as suggested here and otherwise) presents itself, since modifications and new procedures are going to be needed anyway.

Changes from the top down

Since the TR-system is a centrally administrated system which is used on an organisation wide scale, any changes to the system would have to be introduced in the entire organisation. It is my opinion that it would be very difficult to find mandate for such changes.

On the plus side, if it is possible to realise these changes, there will be plenty of resources available for the implementation of a good data gathering scheme.

Changes from the bottom up

Another way to introduce the changes would be through small projects. These projects would serve as test beds and “proof of concept” for the new data gathering schemes and quality improvement activities. They should try to take full advantage of the new way of working, tools and data.

The advantage of such an approach, is that it will not require commitment of the entire organisation, or very large amounts of resources, until the new scheme has proved to be successful. Strengthened with the proven concept, the scheme can then be expanded to include more of the organisation.

A problem with this is that these projects would have to be allowed freedom to explore new concepts, while somehow still remaining connected to the existing design organisation. It may also not be realistic to commit the relatively large amount of resources needed in order to acquire and introduce the new tools and methods for such small projects.

7.6 Lists of recommendations

These lists are a compilation of the earlier recommendations in this chapter. The high priority list contains things that are either considered very important, and/or easy to implement. The medium priority list contains suggestions that are less important, or maybe difficult to implement. The low priority list contains other things which may be of interest to consider.
7.6.1 High priority list

- Specify location of defects on Block level.

- Document type and cause of defects.

- Document effort spent on defect detection activities in all phases of development.

- Document what is being produced. New and modified code.

- Make sure there is someone (or a group) responsible for analysing the collected data and feeding the results back into the process.

Rationale

The high priority list consists of improvements in mainly three different fields:

1. The level of detail on which conclusions can be drawn.

2. Information for the prevention of defects.

3. Confidence in the conclusions on quality.

To improve the traceability of defects is a consideration that is of fundamental importance and will affect the usefulness of almost all other defect data that is collected.

The type and cause of the defects is information aimed towards identifying problematic areas, practices or whatever underlying causes for defects there may be. They should prove useful for guiding improvement efforts towards the prevention of defects in the future, something which I consider to be of primary importance.

Defect detection effort is the vital and missing piece of information that is needed in order to draw valid conclusions on the quality of a component from its defect data.

7.6.2 Medium priority list

- Specify location of defects on SWU (or equivalent) level.

- Document time and phase of introduction of defects.

- Document time, phase and means of discovery of defects.

- Document time and effort spent on the different activities in a project.

- Trace defects to design documents.
Rationale

It is important to be able to create defect statistics on a sufficiently detailed level, in order to be able to draw good and useful conclusions from the data. To have defect data on the SWU level (or equivalent) would certainly be interesting, but may prove to be difficult to realise.

The time and phase of when the defect was introduced is information aimed towards the prevention of defects. The reason why this is considered medium priority rather than high priority is because of the difficulty involved in correctly determining the time and phase of introduction.

The time, phase and means of discovery is information aimed towards evaluating and improving defect detection and removal techniques. This is certainly important, but is deemed secondary to defect prevention.

Having information on the time and effort spent on the different phases and activities in a project, would provide an important context to which the defect data and quality of the product can be related to.

Design documents are important for the development of the system, and the ability to monitor their quality should be useful.

7.6.3 Low priority list

- Document the difficulties of finding and correcting a defect.
- Introduce more detailed levels of severity.
- Introduce fault and change management systems capable of dealing with the different concepts of failures, faults and changes.
- Trace defects to testing environment and test code.

Rationale

Information on the difficulties when dealing with a defect, will give additional detail in the description of a defect, but is likely to require good supporting information if it is to be worth collecting.

The ability to differentiate between failures, faults and changes would certainly be valuable. It will, however, probably require a significant amount of effort of introducing new systems, or adapting old ones, to be capable of handling the data.

The creation and maintenance of test code and documents is a significant part of the responsibilities of the organisation. Nevertheless, I still consider it to be a lower priority than product source code and design documents, since its influence on the product quality is indirect rather than direct.
Chapter 8

Going from here

So, after all this talk of metrics, quality and defects, what do we do now? How do we proceed from here? We would of course like to reap as much benefit from this as we can by judiciously implementing the recommendations in chapter 7, but some of these will require a bit of additional consideration before being realised.

Section 8.1 contains suggestions for further investigations to answer questions that remain. In section 8.2, I intend to set the picture right by presenting a vision of what we could hope to achieve by doing this. Finally, section 8.3 contains some concluding remarks.

8.1 Suggestions for future work

As much as I would have liked to be concrete in my suggestions, they are necessarily a bit vague. It was not possible within the scope of this project to investigate precisely how the actual implementation of these suggestions should be realised. This, among other things, will have to be the work of future investigations.

How to perform the defect classifications

There are two aspects of defect classification that would have to be investigated:

1. Which attributes of defects are worthwhile to collect.

   There are many attributes of defects that one could measure and collect, but it is not likely that all of them are worth the effort. To start out, it is probably best to select a few attributes that are likely to be of use to the organisation.

2. What should the classifications for the attributes look like.

   To create a new classification scheme from scratch is a rather difficult task, and it is a good idea to seek inspiration from that which already has been tried and tested.

   The “IEEE Standard Classification for Software Anomalies” [17] would be a good starting point for how attributes such as type, cause, phase, etc. should
be classified. The scheme is, of course, not perfectly suited for this particular organisation, and will have to be tailored to give the appropriate amount of detail without being impractical to use.

A classification which probably will require particular care and thought is the cause category. The cause is of primary importance for identifying problems and preventing defects, and having a good classification with sufficient detail will greatly enhance its usefulness. The make up of the cause category is also dependent on how far it is possible to trace faults to their real causes, as well as the object it is applied to.

A good way to create a suitable classification scheme, would be to start with a tentative classification scheme, based on, for example, the aforementioned schemes and experienced judgement, and then attempt to classify a number of defects. This will undoubtedly reveal problems and necessary changes to the classification scheme.

**How to evaluate non-structural testing effort**

The importance of being able to measure testing effort in order to interpret defect statistics correctly, has already been mentioned several times in this report. One can think of different ways to evaluate testing effort, taking into account:

- Information from the FDs, VSs and VRs.
- Number of test cases.
- Number of interfaces tested.
- Size and complexity.

The best method will probably have to be the outcome of a separate investigation, taking into consideration testing theory and the actual information available.

**How to extend defect tracking to other areas**

While the focus of this investigation has been on defects in code, there could be even more to win by studying defects from all phases and products (section 7.4.4).

When a defect is found in code, it means that it already has incurred the cost of implementation and correction, when it ideally could have been detected and corrected already in the specifications. Really early fault prevention will deal with the problems as soon as they have been made, even before they come into the code.

Also mentioned in section 7.4.4, it would be interesting to consider tracking defects to the test code and testing documentation, since a significant amount of time and effort is invested in constructing and executing these tests.

An investigation of how to extend the collection of defects would include:

- How the new non-code, non-program objects should be handled.
• How the classification for the new kinds of defects should look.

• How to evaluate “defect detection effort” for non-code, non-program objects.

How to handle the distinction between faults, failures and changes

In this report, the distinction between faults and failures (section 3.5.1) has been somewhat blurred by the inclusion of failure/detection information with the fault. A better model of the situation would distinguish between the three different concepts of faults, failures, and changes.

It is of interest to keep these separate, since a fault can give rise to several, possibly different, failures, with their different nature and frequency of occurrence being of interest to document. A fault that has been discovered through a failure or another detection technique (section 3.5.4), will then likely be corrected through one or more changes to code or documents. Not every fault will necessarily result in a correction.

In order to differentiate between faults, failures and changes, a system, or several, is needed that can handle the different concepts and maintain the relations in between (section 7.4.3). How this distinction is to be realised and how such a system could be implemented would have to be the result of a separate investigation.

What kind of project data is necessary to collect

As mentioned in section 7.2, it is also important to collect data about what is being done, and how, in order to determine what works and what does not. The focus of this master's project has not been on project data, however, and I have thus been content with pointing out the importance of documenting defect detection effort.

Before routines for collecting project data are introduced, it is recommended that an investigation is undertaken to determine which data that is worth collecting. A good way of doing this is to use a top-down method, such as the GQM (section 3.4), starting with the goals or questions different parties in the organisation want achieved or answered, and then identifying the data needed to accomplish this.

8.2 The vision

What we would like to achieve eventually, is a more transparent development process, where the different activities, resources and results are accessible, monitored and controlled.

The resulting quality, or the effort required in order to achieve a certain quality, should be predictable from a relatively early time in the project. New methods, such as a new method of testing, or a new enforceable coding standard, could be tried out, and the resulting effect on the product or project could be measured and evaluated for cost-effectiveness. The most fault-prone practices, or the most troublesome areas of the system, could be identified in the statistics and investigations started to discover the cause of the problems.
Ultimately, the components that make up the system, and the methods that produce and integrate them with each other, should be as predictable as the properties of the bolts and girders in a bridge, and as well known as the physical laws that fuse them together into the finished construction. In the end, we should no longer have to run the train across the bridge to see whether it holds or not. The fact that it will hold should be clear from the beginning to the end.

8.3 Final words

In summary, I would like to say that the problem basically is made up of two elements:

1. To know what is being done and how.
2. To know if it is good or bad.

The usefulness of these two elements is intimately dependent on each other. Knowing what you did is of little use unless you know if it works or not, and knowing if it works or not is of relatively little interest unless you know what you did to achieve that in the first place.

This report has mainly been about one attempt to establish a connection between the two elements: if the code is complex or not, and if that is bad or not. The inability to prove or disprove this connection was deemed mostly to be because of difficulties in measuring the quality of the objects under study, i.e. the defect data.

The information about defects that I recommend to be collected is in most cases not new information, this is information which already exists. The information exists in the head of the engineer at the time of correction, and it sometimes exists in the free text analysis accompanying the TR. Unfortunately are neither of these of any use for statistical purposes due to their lack of structure. To take advantage of this information is mostly a question of defining its structure and making it available for analysis. It is about realising that past mistakes are a valuable and important resource, because they teach us how to behave in the future.

As much as there is a downward spiral of overwhelming quality problems, stealing time from new development and leading to even more problems, there is also an upward spiral of improvements. The recommendations in this report will lead to better insight into the workings of the development process, which in turn will give more opportunities to further investigate and understand the process. Also, when less time is spent on corrective activities, more time is available to improve the quality even further. These improvements in our ability to evaluate the effects of what we do, are the necessary kick-off in order to start the improvement cycle and begin the climb. This is not to be seen as much as the end of a beginning as it is the beginning of a beginning.
References


120


Appendix A

On the measurement theoretic properties of cyclomatic complexity

Note: This chapter is not intended to be read without some prior knowledge of measurement theory. For an introduction to the subject, see for example [10], [32] or [6].

Many seem to acknowledge the need for foundations of software metrics in measurement theory, Fenton [10] and Zuse [32], among others. Yet other people, like L. Briand et al. [6], hold a more pragmatic view of the constraints imposed by measurement theory.

The difficulties with the interpretations and statistical methods that is used with different measures, are due to uncertainties of the scale of the measurement in question. Different real scales allow for different manipulations and statistical methods to be used on the values. (See table A.1.)

Fenton and Pfleeger claim in their book [10, pg.63–64], that cyclomatic complexity falls to be even an ordinal scale as a measure of “general complexity”, while Zuse [32] manages to prove that a measure very close to cyclomatic complexity, \( u(G) = e - n + 1 \), is of the higher ratio scale.

These difficulties in determining the scale of the measure are, in turn, partly due to differences in the assumed properties of the underlying empirical relational system. Weyuker presented nine properties that would be reasonable and desirable for complexity measures to have [30], and thus, properties that the underlying empirical relational system would be assumed to have. Zuse, in proving that \( u(G) \) is on a ratio scale, contradicted some of Weyuker’s properties by assuming the empirical relational system to fulfill the axioms of an “extensive structure”, as well as the measure to be additive under concatenation of flow graphs.

Of course, the controversy lies in what the measure is supposed to measure. If the measure is supposed to reflect the “real” or “cognitive” complexity of code, one can, for example, not assume the measure to be commutative under concatenation of code blocks. On the other hand, if the measure is supposed to reflect the amount
<table>
<thead>
<tr>
<th>Scale Type</th>
<th>Defining relations</th>
<th>Examples of Appropriate Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>Equivalence</td>
<td>Mode Frequency</td>
</tr>
<tr>
<td>Ordinal</td>
<td>Equivalence</td>
<td>Median</td>
</tr>
<tr>
<td></td>
<td>Greater than</td>
<td>Percentile</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Spearman’s rho</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kendall’s tau</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kendall’s W</td>
</tr>
<tr>
<td>Interval</td>
<td>Equivalence</td>
<td>Mean</td>
</tr>
<tr>
<td></td>
<td>Greater than</td>
<td>Standard deviation</td>
</tr>
<tr>
<td></td>
<td>Known ratio of any intervals</td>
<td>Pearson product-moment correlation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Multiple product-moment correlation</td>
</tr>
<tr>
<td>Ratio</td>
<td>Equivalence</td>
<td>Geometric Mean</td>
</tr>
<tr>
<td></td>
<td>Greater than</td>
<td>Coefficient of Variation</td>
</tr>
<tr>
<td></td>
<td>Known ratio of any intervals</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Known ratio of any two scale values</td>
<td></td>
</tr>
</tbody>
</table>

Table A.1. Appropriate statistics for various scale types [27]

of decision logic in a code block, then it is reasonable for the measure to be commutative.

In particular, if cyclomatic complexity is considered to be a measure of the number (the count) of linearly independent paths, one “aspect” of complexity maybe, then intervals and ratios of the measure are meaningful and the measure is on a ratio scale. This is also the interpretation of cyclomatic complexity made here in this report.