An Empirical Investigation into Metrics for Object-Oriented Software

Michelle Helen Cartwright

A thesis submitted in partial fulfillment of the requirements of Bournemouth University for the degree of Doctor of Philosophy

October 1998

Bournemouth University

Abstract

Object-Oriented methods have increased in popularity over the last decade, and are now the norm for software development in many application areas. Many claims were made for the superiority of object-oriented methods over more traditional methods, and these claims have largely been accepted, or at least not questioned by the software community. Such was the motivation for this thesis. One way of capturing information about software is the use of software metrics. However, if we are to have faith in the information, we must be satisfied that these metrics do indeed tell us what we need to know. This is not easy when the software characteristics we are interested in are intangible and unable to be precisely defined.

This thesis considers the attempts to measure software and to make predictions regarding maintainabilty and effort over the last three decades. It examines traditional software metrics and considers their failings in the light of the calls for better standards of validation in terms of measurement theory and empirical study. From this five lessons were derived. The relatively new area of metrics for object-oriented systems is examined to determine whether suggestions for improvement have been widely heeded.

The thesis uses an industrial case study and an experiment to examine one feature of object-orientation, inheritance, and its effect on aspects of maintainability, namely number of defects and time to implement a change. The case study is also used to demonstrate that it is possible to obtain early, simple and useful local prediction systems for important attributes such as system size and defects, using readily available measures rather than attempting predefined and possibly time consuming metrics which may suffer from poor definition, invalidity or inability to predict or capture anything of real use.

The thesis concludes that there is empirical evidence to suggest a hypothesis linking inheritance and increased incidence of defects and increased maintenance effort and that more empirical studies are needed in order to test the hypothesis. This suggests that we should treat claims regarding the benefits of object-orientation for maintenance with some caution. This thesis also concludes that with the ability to produce, with little effort, accurate local metrics, we have an acceptable substitute for the large predefined metrics suites with their attendant problems.

Acknowledgements

Thanks first and foremost to Martin Shepperd for his help and support, without which I would never have completed this thesis, and probably wouldn't even have started it.

Next thanks to Dan Simpson, for his help in getting me this far.

Thanks to DEC, particularly David Knight and Peter Hogarth, for having me here and for the financial support, and to Jacqui Holmes for making life much easier than it would otherwise have been.

Thanks to Claire Joyce, who supervised the experiment and collected the data.

Last but not least thanks to my fellow researchers, especially Chris Schofield for answering various questions and going through all this with me.

Finally a pox on Bill Gates and his organisation (yes, this thesis was produced in spite of Microsoft Word).

Author's Declaration

The following publications are based on this thesis:

(i) Cartwright, M. H. and M. J. Shepperd. "Maintenance the Future of Object-Orientation." In Durham 95 Ninth European Workshop on Software Maintenance in Durham. UK, 1995.

(ii) Cartwright, M and M. J. Shepperd. An Empirical Study of Object-Oriented Metrics. Bournemouth University, 1997. Technical Report

(iii) Cartwright, M. and M. J. Shepperd. "Building Predictive Models from Object-Oriented Metrics." In Proc 8th European Software Control and Metrics Conf. in Berlin, 1997.

(iv) Cartwright, M. H. "An Empirical View of Inheritance." Information and Software Technology (accepted for publication) (1998)

Chapter 1 Introduction

Synopsis

This chapter describes the research problem undertaken by this doctoral research, namely the need to redress the lack of empirical evidence regarding the application of object-oriented (OO) technology, with the emphasis on software maintenance. Next the chapter explains the need for such an investigation. Various OO concepts are defined and the scope of the research delineated. The research approach is outlined and the chapter concludes by summarising the structure of the remainder of the thesis.

1.1 The Research Problem

Although the original ideas behind OO technology derive from work on the programming language Simula in the 1960s, it was not until the 1980s when the work was popularised and its use became more widespread. Presently, C++ and Java are widely used and widely taught. The OO paradigm could be regarded as the orthodoxy of the late 1990s. One reason for its pre-eminence is that proponents of the OO paradigm makes a number of claims as to its benefits. Those pertaining to maintainabilty are considered here below.

The common thread running through many of the textbooks and papers on OO is that OO leads to a simpler solution to a problem, or at least more complex problems can be tackled than with more conventional methods, because object-oriented methods provide ways of abstracting out information leading to a system that is relatively easy to understand.

Wirfs-Brock, Wilkerson and Weiner (Wirfs-Brock, Wilkerson et al. 1990) feel that the use of OO methods leads to a software system which is more

maintainable and extensible. They claim that encapsulation and information hiding constrain communication between objects (lower coupling), enabling communication patterns between objects to be more easily understood and so making it easier for the maintainer to locate errors and to assess where side effects of changes could occur.

Rumbaugh *et al.* (Rumbaugh, Blaha et al. 1991) claim that the use of OO analysis and design methods will lead to "better understanding of requirements, cleaner designs, and more maintainable systems". Encapsulation is again highlighted as a concept which promotes maintainability by minimising interdependency between objects and thus the effects that changing one object will have on others in the same system.

Rao (Rao 1993) claims the OO results in "improved programmer productivity and ease of software maintenance" and considers information hiding to be a factor which effects maintainability, since it limits the effects of change.

Booch (Booch 1991) feels the use of object-oriented methods leads to smaller systems where code is reused and also systems that have a simpler structure. He feels that the process by which OO software is developed "reduces the risk of building complex software systems, because they are designed to evolve incrementally from smaller systems in which we already have confidence". In his earlier work, (Booch 1986) he concludes that since OO "captures a model of the real world" the resulting software will be more understandable and maintainable.

However, we have concerns that these claims are largely unverified. Unfortunately, we have comparatively little empirically based knowledge of the behaviour of systems that have been implemented

using OO technology, with a few notable exceptions such as (Wilde, Matthews et al. 1993; Cartwright and Shepperd 1997b; Harrison, Counsell et al. 1997; Hatton 1997). Thus as OOT (object-oriented technology), and particularly the use of C++, continues to be heavily invested in, research into better understanding, and prediction, of the behaviour of object-oriented software is a matter of some urgency.

Despite the need for empirical research into large scale OO systems, the majority of object-oriented metrics research has concentrated upon defining sets of structural metrics, (e.g. (Abreu and Carapuca 1994; Chidamber and Kemerer 1994)). The structural metrics are measures of a range of attributes, in the main pertaining to various architectural aspects of OO systems. Without empirical evidence it is not possible to say how useful these measures are, particularly in the sense of being inputs to prediction systems (e.g. of defects, reliability, cost etc.) that can yield sufficiently accurate results to aid in the process of developing software. It seems, therefore reasonable to conclude that there is a need to study OO systems including those drawn from industry without restricting ourselves to predefined sets of metrics, which may or may not be useful. We can then make best use of the available data, rather than discarding something potentially useful because it is not required or considered by a particular metrics set.

The aims of this research are:

i) To investigate the impact of key OO mechanisms, specifically inheritance, on software maintenance.

ii) To examine previous work in the area of traditional, complexity metrics development and identify any problems with this approach. These problems could be used to derive

"lessons to be learned", which would be considered when assessing the metrics proposed for object-oriented software.

iii) To consider the available OO metrics in the light of what was discovered from the above aims (examination of previous work and the impact of inheritance on maintainability) and ascertain which, if any, metrics fulfilled the criteria of being easy to obtain and useful.

iv) To develop simple local prediction systems for size and maintainabilty (in terms of defects) and assess their accuracy.

1.2 Scope and Definition

This research concentrates on the effects of object-oriented software (designed using an object-oriented analysis and design method (OOAD) and coded with an object-oriented programming language (OOPL)), upon software maintenance. Here it is worth clarifying what is meant by maintainability. In the literature dealing with OO and maintenance, the term is rarely clarified. From their context, it is sometimes taken to mean corrective maintenance or more often, perfective maintenance (Lientz and Swanson 1980). In terms of the case study, we are considering corrective maintenance, the fixing of errors or defects. In the experiment the maintenance changes are intended to be perfective (adding new requirements) although it was not impossible that some corrective maintenance might be required.

Figure 1.1 below shows a McCall inspired model (see van Vliet (1993), one of many possible secondary references, since the original McCall reference is somewhat obscure). This indicates the aspects of

maintainability with which this thesis is concerned. In the experiment described in chapter 5, time to implement a change is used as a measure of maintainability. The case study uses the number of defects (and defect density) as a measure of correctness. Both maintainability and correctness are affected by the understandability of the solution and familiarity with the problem domain. This thesis is concerned only with the former. The model shows three issues that impact upon understandability. These are size, which has been measured in the case study by LOC, STATES and EVENTS, amongst others; architectural and structural mechanisms, which can be further categorised, and documentation, which is outside the scope of this thesis. Inheritance, coupling and cohesion are architectural / structural mechanisms. This thesis is concerned with inheritance, measured by the Chidamber and Kemerer (Chidamber and Kemerer (1994)) metrics, DIT and NOC (see section 3.3.1 and chapter 5). The case study suggests then, a link between inheritance and correctness, but of course, the findings of a case study cannot be generalised (see section 1.3).

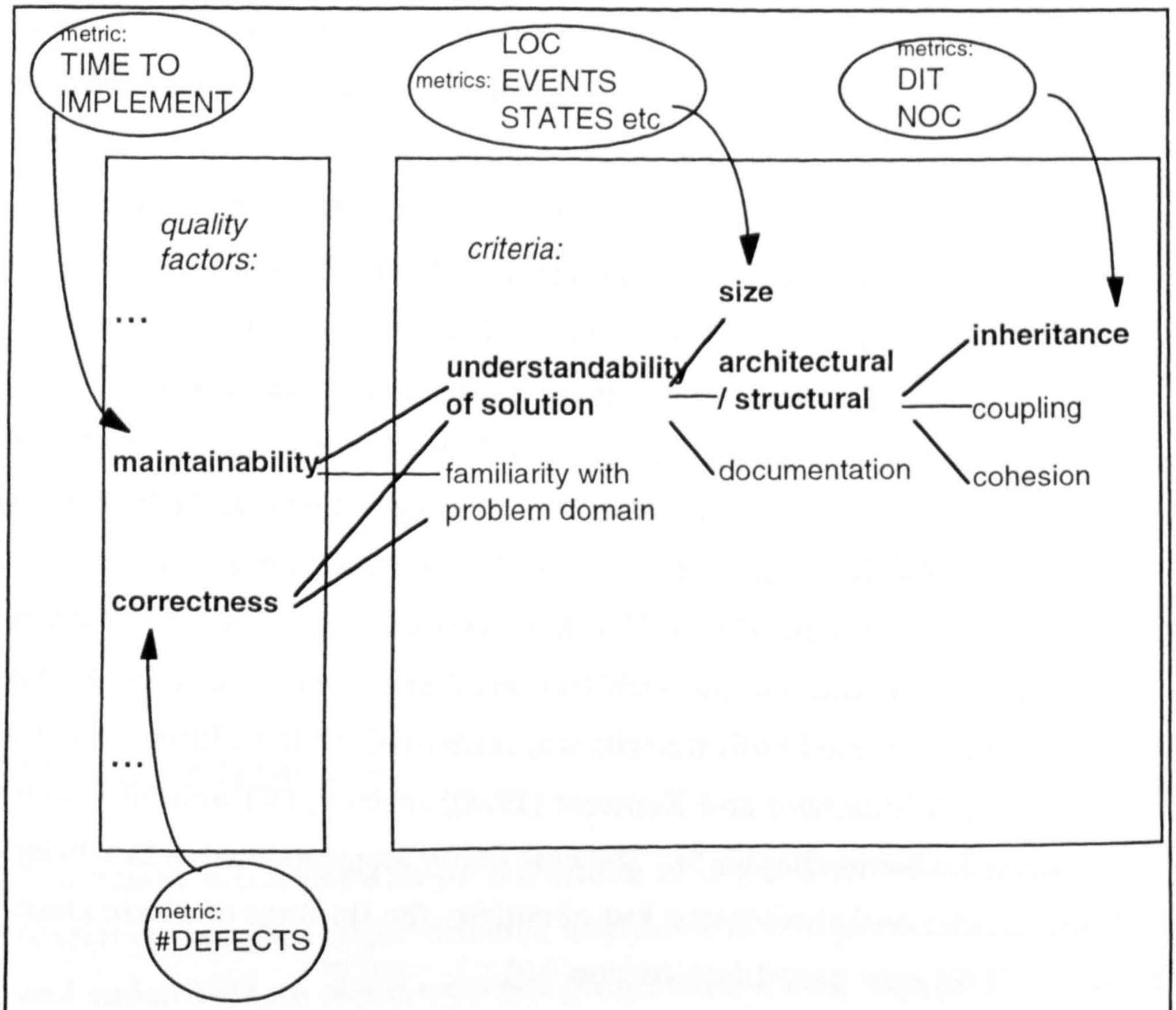

Figure 1.1: Partial Quality Model for Maintainability

A case study and an experiment were chosen as research methods. Both are based on C++ systems, since, at this time (of carrying out the research) C++ is the most successful[1] and widely used OO language for industrial systems.

The object-oriented paradigm exhibits three characteristics recognised as important to the development of good quality software, namely encapsulation or information hiding, abstraction and modularity

---

[1] In terms of industrial interest, usage, number of systems developed and variety of application areas. This was indicated by a small survey carried out by the author in 1992. There are journals either dedicated to or giving prominent coverage to C++ , including some aimed at a more mass market as opposed to special interest groups.

((Pressman 1992)). (Booch 1991) adds others to this list - hierarchy, typing, concurrency and persistence.

As mentioned in section 1.1, OO concepts were first included in the language Simula. This language never achieved the popularity of subsequent OOPLs such as Smalltalk, C++, and more recently, Java. Object-oriented analysis and design (OOAD) methods have developed from object-oriented programming languages and are therefore very different from traditional methods in terms of both the way that problems are decomposed and in the architecture of any system developed using an OO method. However, there is no universal agreement on what features an object-oriented language should include, what exactly each concept should entail or how it should be used. Thus, since what constitutes OO programming is interpreted differently among different practitioners (Rentsch 1982; Booch 1991), OOADs have developed different ideas and emphasise different aspects of OO, and OO terminology is not standardised across programming languages or development methods.

(Booch 1991) feels that the object-oriented approach is more than a software development method. In addition to programming languages and analysis and design methods, OO is applicable to user interfaces, databases, knowledge bases and computer architecture and as such offers a more integrated approach to system design.

Some of the terms used in this report are explained. Due to the lack of uniformity in the terms used to describe concepts, an effort will be made to give commonly used synonyms, which, throughout the remainder of the report, may be used interchangeably.

(i) Object

An object has identity (data) and behaviour (operations.) It may be concrete or conceptual. It may represent an entity in the real world, such as a device, something which exists only in the context of the system, a role played by a person and so on. Booch (Booch 1991) defines an object as "a tangible entity that exhibits some well defined behavior".

(ii) Class

A class is a collection of objects sharing a common data structure and behaviour. Each object is said to be an instance of the class. The data or attributes for all objects in a class are common but the specific values may differ.

(iii) Operation

This term is used interchangeably with method An operation is simply an action which an object may perform and is invoked using a *message*.

(iv) Abstraction

This is a way of dealing with complexity, where essentially, a simplified view of a problem is created, with unnecessary details suppressed. Booch (Booch 1991)defines abstraction in the context of OO thus: " An abstraction denotes the essential characteristics of an object that distinguishes it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the user". Abstraction allows the separation of an object's behaviour or what it does from its implementation or how it does something.

(v) Encapsulation

This is the grouping of data and operations which affect the data into an object. The information encapsulated may be hidden from external view. The external view of an object, or its public face consists of what it offers or can do, whereas the internal view or private face is concerned with the implementation or how things are done. This is known as information hiding.

(vi) Inheritance

This is a further means of classification. Objects may be grouped into classes, classes may be arranged into an inheritance hierarchy, whereby new classes may be created as refinements of existing classes. In other words inheritance provides a mechanism for the creation of a taxonomy of classes, where classes inherit behaviour from others and refine or add something more to form a unique class.
Inheritance is usually taken to mean that each class inherits from only one other, unless specifically defined as multiple inheritance.

(vii) Sub and Superclasses

These arise out of the use of inheritance. A subclass inherits properties from a superclass and refines them i.e. adds its own behaviour.

(viii) Polymorphism

This abstraction allows an object to send out a message without needing to specify which object should implement it. Two or more objects may be capable of responding to the message, depending upon the data or parameters supplied. For example, an object can send out a print

message, without concerning itself with deciding which object should carry out this task. There will be two or more objects (normally related by inheritance) which can implement the command, each in its own way, appropriate to a particular type of document. Thus when the client sends out a print message along with information that, for example, it is a graphics file, and its required destination (printer or screen), the appropriate object can respond and implement the method to print that document. Another example is the "+" operator, it will be implemented in different ways according to what is being added. This is also known as overloading. There are various classifications of polymorphism. The interested reader is referred to (Booch 1994).

It is also necessary to consider what is meant by the term metric. In this thesis it is used as a generic term for both measurement and prediction system, applied to software. It is used when it is not necessary to distinguish between these more specific terms, where the particular term intended is obvious from the context, or where the work of others is being discussed and the work in question does not distinguish between the terms measurement or prediction system. The metric may be derived from code, design or some other artefact or process arising from the activity of software development. A measure is taken to mean a measurement taken from some software artefact (e.g. the number of classes taken from design documentation) or some combination of measures used together, perhaps as a proxy for a less tangible concept.(e.g. the number of classes + the number of couplings between classes) being used as a proxy measure for the complexity of a design. A prediction system uses measurements in some way to provide a prediction about some attribute which at that time is unmeasurable (e.g. using the measure of the number of methods in a design to predict the number of lines of code in the finished system).

So, to summarise, this thesis is concerned with the investigation of fully OO systems, as defined above, and excludes what is sometimes referred to as object based systems such as Ada 83. In practice the focus is C++ which is currently the most widely adopted OO programming language.

1.3 Research Methodology

This section will briefly define and discuss the approach taken

1)case study

The more traditional area of application for this research method is in the field of social science research. It is, however used in object-oriented systems research, examples of which include (Booch 1986; Mancl and Havanas 1990; de Champeaux, Anderson et al. 1992; Wilde, Matthews et al. 1993; Capper, Colgate et al. 1994; Pomberger and Pree 1994).

The working definition used in this thesis is taken from Yin (Yin 1994):
A case study is an empirical enquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident. This suggests that contextual conditions may be relevant to the investigation. Case studies are also likely to encounter situations where there are more variables of interest than there are datapoints. Case studies may also utilise multiple sources of evidence (and if so data should triangulate).

As with any approach, case study has its strengths and weaknesses. It facilitates the study of a phenomenon in its real context, playing down the need for distinction between the boundaries of phenomenon and context, but sacrifices control. Pre-established theories and propositions can guide data collection and analysis, and it is flexible enough not to exclude the unexpected. However, a case study suffers in its potential for generalisation, in that the results from one case can not necessarily be generalised to another (similar) situation. Yin (Yin 1994) suggests that "case studies, like experiments, are generalizable to theoretical propositions and not to populations or universes." In this thesis, the case study will be used to formulate a hypothesis or hypotheses, which can be further tested by experimentation and analysis. Yin (Yin 1993) describes this approach as an exploratory case study. Ideally a number of case studies could be used to build up a body of evidence in support of a particular contention. In the absence of further industrial case studies during the time available, an experiment, also looking at object-oriented software maintenance was conducted.

2)experiment

An experiment answers the same type of research question, namely "how?" and "why?", as a case study (Yin 1994), and can thus be complementary. It allows the researcher to isolate the phenomena from its context and thus is able to filter out extraneous factors to some extent. This control is gained at the expense of reality. A laboratory setting will be artificial and what happens in the lab may not happen in the real life context. The most typical experiment, and the type used in this research is a factorial design where the independent variables are varied

systematically and the dependent variables are quantitative, objective measures (Adelman 1991).

Factor	Case study	Experiment
scale	Large phenomena	small well defined events
No. of cases	few	many
control	little	much
inference	local	generalizable if replicated
setting	In situ	laboratory

Table 1.1: Comparison of characteristics of empirical approaches

1.4 Structure of the Thesis

Chapter two provides a review of the development of software metrics since the earliest reported work in the 1950s. It traces the progress, and sometimes lack of it, over the subsequent four decades. It considers the two main approaches to the development and validation of metrics, the axiomatic approach and the empirical approach and also the application of measurement theory to the development and validation of metrics. From this history and critique, five major problems and thus lessons to be learned are derived.

Chapter three considers many of the metrics proposed for object-oriented systems. The metrics fall into two categories, traditional metrics, that is metrics already proposed for structured systems, and new metrics developed specifically for object-oriented systems. These are examined in the light of the five lessons to be learned (chapter 2), to give an assessment of the current state of object-oriented software metrics.

Chapter four describes the case study carried out on a large industrial object-oriented telecommunications system. The aims of the case study are given. From the analysis a hypothesis is drawn that classes that are part of an inheritance hierarchy contain a higher density of defects than classes that are not involved in an inheritance relationship. The case study is also used to demonstrate that simple local prediction systems can be derived easily, without the need to use predefined metrics which are sometimes complicated, difficult to collect at design time or part of a large suite of similar metrics. Measures collected are defined and the process by which the prediction systems are derived and tested is described and the results presented.

Chapter five describes the experiment carried out to investigate the impact of inheritance on maintenance effort. The experiment used is a partial replication of an experiment designed and implemented by John Daly for his doctoral thesis at the University of Strathclyde (Daly 1996). The hypotheses under test are presented and tested.

Chapter six is the final chapter with the summary and conclusions to be drawn from the research program. It also outlines the weaknesses of the research and suggests further research that could be carried out to build up a body of empirical evidence and to complement the work carried out so far.

1.5 Background Knowledge

The reader is expected to have some knowledge of object-oriented concepts, although practical experience is not necessary. Readers are referred to the following texts on object-oriented analysis and design: (Wirfs-Brock, Wilkerson et al. 1990; Shlaer and Mellor 1992). Shlaer and

Mellor describes the analysis and design method used in the industrial case study described in chapter 4. Wirfs-Brock *et al* provides a more "pure" introduction to object-orientation, having evolved from the experience of Smalltalk designers, as opposed to the former method, which developed from a structured approach. The reader is provided with the necessary aspects of measurement theory in sections 2.4.1.1 and 2.4.1.2. Chapters 4 and 5 require a basic knowledge of statistics for exploratory data analysis and hypothesis testing. One recommended book is Tukey's book on exploratory data analysis (Tukey 1977), but any book covering hypothesis testing and non parametric statistics should suffice, since the tests and plots employed are in common use.

1.6 Reports resulting from this research

The following reports have been produced:

(i) Cartwright, M. H. and M. J. Shepperd. "Maintenance the Future of Object-Orientation." In Durham 95 Ninth European Workshop on Software Maintenance in Durham. UK, 1995.

(ii) Cartwright, M and M. J. Shepperd. An Empirical Study of Object-Oriented Metrics. Bournemouth University, 1997. Technical Report

(iii) Cartwright, M. and M. J. Shepperd. "Building Predictive Models from Object-Oriented Metrics." In Proc 8th European Software Control and Metrics Conf. in Berlin, 1997.

(iv) Cartwright, M. H. "An Empirical View of Inheritance." Information and Software Technology (accepted for publication) (1998).

Also presentations at :

Durham '95 (i);

BMW '96 Bournemouth Metrics Workshop, 18th-19th April 1996;
ESCOM 97 (ii);
EASE '98 (iv).

1.7 Summary

Empirical research on the application of object-oriented development has lagged far behind the uptake of object-oriented methods and languages. This thesis aims to provide empirical evidence on the impact of object-orientation on software maintenance. Software metrics are an important resource for assessing attributes such as maintenance effort and related attributes such as software quality. Therefore in addition to the empirical evidence provided via the case study and the experiment, an extensive review of metrics for object-oriented systems is given. The metrics considered are maintenance related in that they purport to capture or predict attributes directly or indirectly related to software maintenance.

Chapter 2 The History of Software Metrics in 57 Pages.

Synopsis

We can all learn from the mistakes of the past. Software metrics proves no exception. This chapter considers the development of software metrics, how researchers have identified problems in metrics and the way in which metrics (both measures and prediction systems) were developed, and how research in other fields can be utilised to improve the development and validation process.

2.1 Introduction

The history of software metrics is a much covered topic almost any book or paper in this area will include some sort of description, critique or overview (see (Pressman 1992; Shepperd and Ince 1993), for examples). In this thesis it is included for the following reason. The major developments that have occurred in the field of software will undoubtedly have influenced the way in which software metrics applicable to the object-oriented paradigm have so far developed and will continue to be developed. Thus, background information on the past development of software metrics will provide a context in which current research, development and practice can be analysed and discussed. Therefore, this chapter does not contain an exhaustive list of software metrics over the last three decades, but takes the form of a critique of the major developments and themes of software metrics to date, thus providing a framework for the examination and critique of object-oriented software metrics which began to appear in the early 1990's.

The need for software engineering techniques, including measurement, in order to plan and control software projects, especially with regard to cost was recognised as far back as 1956, with the SAGE air defence system,

which was probably the first large (around 500, 000 LOC) system to be developed and was also probably one of the first to require teams of programmers as opposed to individuals (Benington 1956; Benington 1983). However, the techniques Benington describes were not taken up by his contemporaries; as he notes, there was no attempt by the computing industry to apply engineering management, despite its success in the SAGE project, possibly because as Benington notes, at the time programmers were regarded as "different" and unable to work under such control. Another notable point in this paper is the statement "the time and cost required to prepare a system program are comparable with the time and cost of building the computer itself." This seems to be ahead of its time - it was not until the late 1960s/early 1970s that software development costs surpassed hardware costs, making the planning, control and cost of software development into a major issue.

Section 2.2 considers the main themes and influential metrics in the 1970's and early 1980's. Section 2.3 examines approaches to validation. Section 2.4 discusses the emergence of the application of classical measurement theory to software metrics, in particular the work of Fenton and Kitchenham published in 1991. In section 2.5 we revisit the previous developments covered in 2.1 and discuss them in the light of measurement theory and prediction systems covered in section 2.4. The focus is on structural metrics - this mirrors the state of object-oriented metrics, which have also tended to be mainly structural/complexity measures. However, not all metrics fall completely into popular categories such as structural/complexity, or cost/effort. See (Fenton and Pfleeger 1996) for a breakdown of areas which go under the banner of software metrics. So although some of the metrics covered (notably Function Points and Bang) are usually considered as cost/effort metrics (prediction systems), they are included here because they require some

structural knowledge as input to the model and thus in a sense, can be considered structural metrics.

2.2 Structural Metrics

Structural metrics are those which measure some aspect of software, such as coupling between components, for example, but are more than simple counts, such as LOC. Structural metrics do not include measures of performance. Structural measures would then normally be used to make some prediction or inference regarding another attribute, such as quality or size. Traditional structural metrics can be categorised in a number of ways, although there will be an inevitable overlap in some cases. The division of this section reflects the concerns of a maturing development process. It can be seen the earlier metrics concentrate on code complexity, followed by design metrics a few years later, with metrics for database systems appearing relatively recently.

2.2.1 Complexity Metrics

As stated above, the need to plan and control software development was recognised as long ago as the 1950s by Benington (Benington 1956; Benington 1983). This opinion did not appear to be widely shared, since the first published attempt at measuring software (structural as opposed to cost estimation or performance) did not appear until 1972. Halstead's Software Physics (Halstead 1972), later Halstead's Software Science (Halstead 1977) was an early attempt to measure code complexity and generated a great deal of interest. It was the first attempt to draw together a variety of factors thought to affect code complexity in order to provide a framework for software measurement to predict other, more useful, measures such as effort and time. Halstead postulated the idea that, as in

other disciplines, such as physics, there were fundamental laws that would hold for software, whatever the environment or development process.

There is little doubt of Halstead's influence on the development of software metrics, since the metrics still continue to be cited and used, despite criticism of its underlying theory and the empirical evidence offered as support (Hamer and Frewin 1982; Shepperd and Ince 1993). Its value lies in the fact it was the first published attempt to provide a framework for measuring software, as opposed to a simple measure such as LOC and the first metric to attempt to measure or quantify characteristics "scientifically".[2] Its legacy is the drive to quantify complexity and thus create a generally applicable complexity metric.

Halstead's Software Science was not only welcomed, it also shaped subsequent software metrics research. For the rest of the decade, and indeed into the next, research effort concentrated entirely on code complexity metrics which were implicitly assumed to be language and environment independent. This was not the case[3], however, which contributed to the list of problems with code metrics. Complexity was perceived as the key to predicting such factors as maintainability, effort, development costs etc.

The simplest metric, LOC has tended to be rejected, partly because what constitutes a line of code can vary with programming style (see (Software

---

[2] Although Benington's original paper of 1956 predates Haltead's publication by almost twenty years, it was not widely read (see (Benington 1983)), plus it does not give sufficient details of how to use measurements to make a prediction to allow the reader to apply it in practice.

[3] An example of metrics not being language and environment independent is that both Software Science and Cyclomatic Complexity paid little heed to modulariztion of software. SS assumes systems to consist of one module (Hamer and Frewin 1982) and CC tends to increase with modularization (Shepperd 1988).

Metrics Definition Working Group 1991), which is just one of a number of publications), and partly because the metric is regarded as too simplistic to characterise something as *complex* as complexity. However, the concept of complexity remains at best, loosely defined. In keeping with the ad hoc approach to metrics development in the 1970's, measures were suggested without a clear idea of what was being measured, how the measures were to be used, what they were to predict or what to do with such predictions.

One reason for this rather casual approach is the rather hazy concept of complexity. At first glance its meaning may seem straight forward, but it becomes obvious that it is not. First there are different types of complexity and second there are many issues, many of which are human factors, which affect the perception of complexity. As we well know, where human factors become involved things become somewhat more *complicated*. It is thus debatable whether complexity is actually a property of the software itself or whether it is dependent upon the interaction of individuals, the task they are performing and the software product upon which they are performing it.

From the literature it can be seen that there have been attempts to categorise the different types of complexity such as *computational* (based on the difficulties in performing the various mathematical computations in an algorithm) and *psychological* or conceptual, which considers the difficulties in the interaction between programmer and software. However, these do little to clarify the situation, since intuitively, software complexity must be an interaction of the two, and the classification does little to determine to what extent either affects the development or maintenance of a program.

We are left with a situation as follows. Software complexity cannot easily be formally defined, if at all. It means different things to different people. A variety of factors are involved, many of them human factors. Even if we could agree upon the factors involved and upon an acceptable definition, such intangible characteristics would be difficult, if not impossible, to capture using measurement. Complexity in itself seems to be of little use. We would then need to know how complexity affects, for example, maintainability, and work out some prediction system.

Thus complexity metrics are attempting to capture an intangible concept based on various factors, many of which (particularly those involving cognition) cannot be satisfactorily measured. Without a definition of what we are trying to capture, we cannot validate a measure or assess its usefulness.

Notwithstanding the problems outlined above, the goal of software metrics research in the 1970's (and beyond) was to measure complexity. Another influential, widely cited and used code complexity metric was McCabe's Cyclomatic Complexity (McCabe 1976). Here programs are represented as directed graphs showing executable statements as nodes and control flow as the edges between them. The complexity works out to be the number of decisions plus one. The result is intended to provide an upper bound beyond which subdivision of the module should occur, and to indicate the amount of testing effort needed. Cyclomatic Complexity has subsequently been used to predict other complexity related features (see (Curtis, Sheppard et al. 1979; Shepperd 1988; Gill and Kemerer 1991)).

2.2.2 Design Metrics

The realisation that taking measurements earlier in the development process would be of more benefit than code metrics in the planning and control of a project came about in the very late 1970's / early 1980's. The first published attempt at system design metrics seems to be Yin and Winchester's complexity metric (Yin and Winchester 1978). This considers only the complexity of the interconnections between modules in the design by treating the module hierarchy chart as a graph and representing complexity as the extent to which it departs from a tree structure. Similar is Benyon-Tinker's graph-based metric (Benyon-Tinker 1979) which also represents modules as nodes and calls as edges. Here a module will be considered only once by the metric, however often it is invoked, therefore, the representation will always be a pure tree. Complexity is instead regarded as a function of the length and breadth of the tree.

Another attempt is Yau and Collofello's system stability metric (Yau and Collofello 1980), where design's "resistance to change" is assessed. In a poor design a simple change will ripple through the design whereas a good design will contain the change within the module. This approach was flawed, since it cannot be reliably calculated purely from design information. Thus, the publication of Henry and Kafura's paper on system design (Henry and Kafura 1981a), based on Henry's doctoral thesis, completed 1979, could be regarded as the first complete design metric in that it cònsidered both the internal complexity of a module as well as that of connections between modules.

Henry and Kafura's Information Flow metric is based on the idea that a good design should be modular with little coupling between modules, an idea adapted for software architecture by (Stevens, Myers et al. 1974) from the work of (Alexander 1964). The complexity of a module is calculated from the number of information flows entering (fanin) and leaving

(fanout) a module (multiplied together and then squared) multiplied by the internal complexity, measured in LOC. This is expressed as:

$length * (fi * fo)^2$.

An obvious drawback is that if applied at the design stage actual LOC will not be available[4]. The authors offer some results from an empirical validation of industrial software, using a collection of changes made to the UNIX system to determine whether the metric can predict which procedures are likely to need changes made to them. The authors present the results of a correlation between changes and high procedure (module) complexity as measured by the metric. The results show a high correlation between the metric and changes (r=0.94), which improves when the length parameter is removed (r=0.98). Various uses were suggested : identifying outliers (modules of unusually high complexity) and system "hot spots" where an unusual amount of "traffic" occurs. Increases in complexity could indicate a missing level of abstraction. Thus, the metric could be used to improve the design, with the subsequent benefits being, of course, the software would be easier to implement, test and maintain.

Post Henry and Kafura came further information flow metrics, IF4 (Shepperd 1989; Shepperd 1990; Shepperd and Ince 1993) and Card and Agresti's design complexity measures (Card and Agresti 1988). IF4 is based on Henry and Kafura's measure, but the definitions of information flow are modified in accordance with criticisms of Henry and Kafura's original metric (Kitchenham 1988; Ince and Shepperd 1989; Shepperd 1990a). Card and Agresti propose separate metrics for structural and internal module complexity. Fanin was discarded since it was not so

---

[4] In practice it is often omitted. Its contribution is questionable — see section 2.5.2.3 .

significant an indicator as fanout and also since counting just one way (i.e. fanin or fanout) ensures that a connection between two modules is not double counted. The significance of inter module (structural) metrics is supported by Troy and Zweben's study of 21 design measures (Troy and Zweben 1981), relating to five categories, including coupling. It was concluded that coupling had the most influence on error counts.

2.2.3 Specification Metrics

Albrecht (Albrecht and Gaffney 1983), proposed a metric, (referred to as Function Points or simply FP) which can be extracted from a specification. It is intended to be used to predict development effort, as an alternative to predicting LOC. The different functions described are identified and then weighted according to the complexity of the function type. The advantages are obvious - the earlier feedback is made, the better. The model has been well received and successfully used in industry (Behrens 1983; Kemerer 1987). It has been subject to modification (Symons 1988) for use with entity-relationship models and for use with real-time systems (Jones 1987).[5]

Another specification metric is DeMarco's Bang metric (DeMarco 1982) which utilises such specification notations as data flow diagrams, entity-relationship diagrams and state transition diagrams. The systems are classified as function strong (a count based on low level data flow diagram bubbles or processes) or data strong (derived from the number of entities in the entity-relationship diagram), or a hybrid. The metric can be used to produce a product size estimate. There is little published work validating the metric, although one by (Rask, Laamanen et al. 1993) suggests that Bang offers advantages over Albrecht's Function Points.

---

[5] This point seems to have been retracted (verbally during discussion at meetings).

2.2.4 Database Metrics

The metrics considered so far are largely functional metrics. In other words they consider function or processing rather than data. This reflects metrics research as a whole. The focus has been almost exclusively on functional systems, with few database metrics being proposed. The earliest seems to be (Geritsen, Morgan et al. 1977) which considers both network and relational databases. Function points (Albrecht and Gaffney 1983) consider both internal data and exported/shared data. Mk II function points (Symons 1991) uses entity-relationship models to derive counts. The Bang metric (DeMarco 1982) also uses entity-relationship models for "data strong" systems. Other work has been carried out by (MacDonnell 1992; MacDonnell 1993), who suggests a large number of measures which can be extracted automatically from the various models used to describe a database system. Later work with Shepperd (MacDonnell, Shepperd et al. 1997) concentrates on a smaller number of measures taken from the entity-relationship model and the functional model in order to make a size prediction. Another study (Gray, Carey et al. 1991) suggests database design metrics, emphasising the need to be able to automatically extract them using CASE tools. Among the metrics suggested is an extension to IF4 (Shepperd 1990b), called IF4+, and an entity-relationship or ER metric.

2.2.5 Summary

During the 1970's and early 1980's, many software metrics were proposed. The main area of interest was software complexity, because of its effects on activities such as maintainability and testing. There was, however no

general agreement on what constituted complexity, or how it could be measured.

The metrics examined in this section were all subject to attempts at "validation". Many papers have been published in support of, for example, Software Science, the authors claiming they have evidence to back this up. The high correlation found between Software Science and actual data have tended to be the result of misuse of, or incorrectly applied, statistics. For example, the study by Funami and Halstead (Funami and Halstead 1976), used data containing errors and the calculations of the parameters n_1 and n_2 are unreliable and thus so are estimates of effort (E) on which they depend (see (Hamer and Frewin 1982)). Ottenstein's validation (Ottenstein 1979), used the same data. Accurate calculations were produced (which varied considerably from those produced by Funami and Halstead), but since the estimates did not correspond well with the actual counts, it has not proved possible to reproduce the desired result without some sort of manipulation. Hamer and Frewin (Hamer and Frewin 1982)), reasonably state that "the claimed experimental support is largely illusory.". The results cannot be relied upon since even when calculations are performed on the same data, different results can occur, depending upon the interpretation. Additionally, a truly independent validation needs to use a different data set to that from which the metric is derived. Hamer and Frewin found that when a different data set was used for validation, the relationship between V (volume) and bugs was non-linear (a linear relationship was observed by Funami and Halstead) and so a high correlation coefficient does not necessarily mean that the number of bugs will rise in direct proportion to size (as represented by volume, V). However, this desire to validate the metric by whatever means should not be surprising. An intuitively pleasing metric, both easily collectable and accurate is something that developers and project managers would naturally crave.

However, metrics need to be rigorously examined and validated (with due regard to measurement theory, thus ensuring correct use of relevant statistics), and, if they do not stand up to this validation, discarded. This is the attitude which emerged in the mid to late 1980's, where the widely cited and used metrics of the 1970's/early 1980's were critically examined and, in many cases, debunked.

The lack of definition, the lack of rigour in deriving metrics and indeed the lack of guidance on how such measures should be used, were typical of an era of confusion. Those who proposed metrics realised the need for measurement in the software development process, but not of the need for definition and validation, so that practitioners could understand what characteristics were being measured, how this was to be done, and have a reasonable degree of certainty that the metric actually did what was claimed.

It can be seen that validity is central to software metrics. The next section (2.3) will consider the issue in some depth. Two approaches to validating metrics will be considered, as will their effectiveness when used in isolation and as complementary techniques. Section 2.4 will consider the application of measurement theory to software metrics and validation.

2.3 Validating Metrics

The mid to late 1980's saw moves to improve techniques for validating metrics. Two strands emerged, the axiomatic approach and the empirical approach. Although evidence had been offered in support of metrics before, the approach which emerged at this time was to more critically validate existing metrics using either a more formal technique, axiomatic validation, as explained in section 2.3.1, or using suitable statistical

techniques (see section 2.3.2), which addressed the usefulness of software metrics. In this context "critically validate" means to provide a demonstration that the metrics worked (i.e. captured what they purported to or gave accurate predictions). Purely speculative work was no longer sufficient. The critical eye cast over metrics that had been accepted almost without question was long overdue. However, metrics development, validation and usage still suffered from confusion, which was not addressed.

2.3.1 The Axiomatic Approach

The axiomatic approach was a more formal[6] way of validating a metric than the empirical approach. It does not deal with the usefulness of the metric in practice, but whether a metric behaves as required in theory. It is a deductive way of reasoning i.e. the metric must be formally defined, and then judged against a set of axioms which describe the desired properties of the metric, which allows us to "see" how the metric would behave in practice. This is the opposite to the empirical approach which is inductive (the results of applying the metric to data are analysed to deduce the effectiveness/validity of the metric). The axiomatic approach allows metrics to be validated without collecting data, which can be difficult and is invariably time consuming, thus it allows for a "quick and easy" validation of a proposed metric. However, the approach has drawbacks which detract from the intuitively appealing notion of proving formally and mathematically whether a metric is or is not valid. Firstly there is the problem of the definition of an axiom set. It is a task requiring rather more skill than proposing software metrics. Defining axioms requires some appreciation of measurement theory. Early work

---

[6] Here formal means that the metrics are tested using a set of rules (axioms) and their compliance or otherwise to these rules can be demonstrated mathematically.

in the area was carried out by Prather, (Prather 1984), who proposed a set of three axioms and applied them to complexity measures. Criticism of the work centres on the weakness of the axioms, since they "validate" measures which are not actually acceptable (Weyuker 1988; Shepperd 1992) as can be seen from that fact that McCabe's Cyclomatic measure satisfies all three axioms, despite its weaknesses (see (Prather 1984)). More extensively quoted, and used to validate software metrics, are Weyuker's axioms (Weyuker 1988), a set of nine axioms for the evaluation of complexity measures. These are reproduced in table 2.1 below, where M is a metric applied to a program, and c is complexity.

PROPERTY NUMBER:	
1	there are programs P and Q for which $M(P) \neq M(Q)$
2	if c is a non-negative number, then there are only finitely many distinct programs P for which M(P)=c
3	there are distinct programs P and Q for which M(P)=M(Q)
4	there are functionally equivalent programs P and Q for which $M(P) \neq M(Q)$
5	for any program bodies P and Q, we have $M(P) \leq M(P;Q)$ and $M(Q) \leq M(P;Q)$
6	there exist program bodies P, Q, and R such that M(P)=M(Q) and $M(P;R) \neq M(Q;R)$
7	there are program bodies P and Q such that Q is formed by permuting the order of the statements of P and $M(P) \neq M(Q)$
8	if P is a renaming of Q then M(P)=M(Q)
9	there exist program bodies P and Q such that $M(P)+M(Q)<M(P;Q)$

Table 2.1: Weyuker's Axioms

This work has also been criticised since whereas Prather's axioms accept weak metrics, Weyuker's axioms reject reasonable metrics (Shepperd 1992; Zuse 1992; Fenton and Pfleeger 1996). Additionally the axioms have been shown to be inconsistent. Zuse, for example has proved, using

representation theory[7], that properties 5 and 6 are contradictory. Property 5 implies that size is a major factor in complexity, since a program cannot (according to this) reduce complexity by adding code (i.e. must be monotonistic). It reflects the received wisdom that we can understand smaller programs better than large ones because we can see more of them at once, but ignores comprehensibility as a factor (i.e. the lower the comprehensibility the higher the complexity). However, property 6 states that two programs of equal complexity can each be concatenated to a third program, the resulting two programs having differing complexities. This axiom covers comprehensibility and not size. Thus the two contradictory views of complexity lead to a situation where no single measure can satisfy both, in other words the notions of low comprehensibility and size embodied in the properties cannot be captured in one measure since they contradict each other. Zuse also shows that the two properties require different measurement scales[8], ratio for property 5 but explicitly not for property 6. This has been disputed by Weyuker, amongst others (Morasca, Briand et al. 1997), but the weight of opinion tends to back Zuse's criticisms (Kitchenham, Pfleeger et al. 1997).

Aside from the issue of flaws in an axiom set, the problem is that the "desirable" properties of a metric are subjective. What is desirable is dependent on what the author of the set considers to be so, not upon universal truths[9]. Weyuker did not claim that the properties she proposed were sufficient as they stand. However, the likelihood of evolving a complete and generally acceptable axiom set for validating complexity metrics is no more likely than arriving upon a generally

---

[7] Representation theory is discussed in section 2.4.1.1.
[8] Measurement scales are considered in section 2.4.1.2.
[9] The notion of universal truths or laws has plagued software metrics development since Halstead. Software Science was based upon the notion that there were fundamental truths/laws about software that would hold, independent of environment, application area etc.

acceptable definition of software complexity itself. As discussed above, differing views on complexity can lead to contradictory statements.

Secondly the axiomatic approach cannot demonstrate the usefulness of a metric, only that the model meets certain criteria, thus is inadequate when used in isolation. The issue of usefulness is separate from that of validity, to be deduced from empirical analysis, which must therefore be considered at least as a complementary technique. The usefulness of a metric is as much an issue as its validity (in the sense that the metric represents what it claims to and does not violate mathematical rules). Usefulness can only be assessed via an empirical study since this is the only technique to try to apply metrics in practice. The formal/axiomatic approach can represent the "real world" mathematically and thus can model what should happen in predefined circumstances. It cannot, as empirical observation can, discover unforeseen phenomena nor assess the practicality and ease of applying the measures.

To summarise: the axiomatic approach introduces pleasing formality into software metrics validation, enabling us to ascertain the likely behaviour of a metric without having to apply it. It deals with general principles, unlike the empirical approach it cannot assess the usefulness of the said metric nor indicate whether or not it is applicable in practice (i.e. the axiomatic approach is deductive not inductive); in addition the axioms sets available (Prather's and Weyuker's) have drawbacks as discussed above, neither able to strike the right balance, either accepting weak measures (Prather) or rejecting reasonable metrics (Weyuker).

2.3.2 The Empirical Approach

The empirical approach requires data to be collected in order to validate a metric. Therefore it has the drawback of being a more time consuming approach to metrics validation than the use of axioms. However, it has a number of advantages. It allows the investigator to observe how things behave in reality, which of course can be rather different from how they "should" behave, software development being a cerebral, human centred activity, thus not conforming to mathematical or physical rules as do other activities, e.g. structural engineering. By collecting data and analysing it in order to discover the patterns within it, inferences may be drawn about what is occurring during the software development process, the effects on program structure on error rates and maintenance activities.[10]

The empirical approach to metrics validation can take the form of industrial data collection or of controlled experiments. Both of these have been used in order to validate metrics. Both suffer from some inadequacy, in the former, the difficulty in replication of results, and in the latter the difficulty of simulating "real" software development under experimental conditions. Further problems with empirical validation occur when the investigation is unfocused or ill defined. It is important to know what you are looking for, and to define what measurements are being taken and how (counting rules - so that measurements are taken consistently). Where such clarity is lacking, spurious correlations can be mistaken as meaningful. An example is Software Science, where measurements were taken to represent complexity, in order to then predict effort, but the metric was also used in other studies, without modification, to predict other complexity related characteristics such as quality and defects (e.g. (Fitzimmons 1978)). Such a lack of clarity and

---

[10] In others words it is an open systems approach, which admits the possibility of external, unknown events. The opposite, closed system is represented by the formal

focus leads to confusion regarding the actual result, where the direct measurement taken can be used to represent another characteristic, or whether is should be used as an input to an equation (see section 2.4 regarding the confusion between measures and predictions systems).

An empirical study is the best technique to assess the usefulness of a metric (axiomatic techniques can't and intuition can't be proved), but this does not happen automatically. The metrics need to be tested by replication, using appropriate statistical tests, with actual data, in order to show accuracy and statistical significance.

Further, there are problems of replication. This is a problem particularly apparent in industrial-based studies. The cost and effort involved in developing software and also in collecting metrics, mean that replicating the study is not an option. Thus although a study may show a metric to work in one situation, it will not necessarily work as well, or at all in another situation, where, for example, the environment is different. Therefore, the effects of such factors cannot be assessed empirically in an industrial setting, since companies will not, understandably, commit resources to, for example, developing a product in two different ways to assess the effect of design method or language upon prediction systems. This can be done in a laboratory based experiment, but still with practical difficulties - if the products were developed simultaneously, the teams would differ in terms of experience, if consecutively, the team would have gained in domain knowledge from the first project, and so on. This can be balanced by assessing the experience of team members and using this knowledge in team selection, but other factors cannot be overcome, such as the size and nature of the system under development (many

---

approach, since all events must be known and defined within the system in order for validation to take place.

industrial systems take years to complete, involve millions of lines of code etc.), this cannot be matched in a laboratory based experiment.

Other problems can occur with experimental design and the statistical techniques used to analyse results. Common examples include the use of statistical techniques inappropriate to the measurement scale (see section 2.4 on measurement theory) or to the data distribution, lack of focus to the experiment (trying to find out too many things in one go), insufficient cases, experimental bias, Hawthorne Effect and so on. However, contrasting results do not necessarily mean one or other study was poorly designed, an allowance must be made for external variation between different environments etc. Taking time to design experiments and to choose appropriate statistics, using the considerable amount of published material on both subjects will alleviate many problems, but some cannot be removed nor their effects be calculated in order to make allowances.

There is frequently a conflict been the need for control and the need for reality. Laboratory based experiments allow more control and the opportunity for replication of experiments in order to strengthen conclusions. However, they are contrived, and cannot compensate for the lack of reality. Empirical observation in the field means reality is in built but the trade off is surrender of control and the opportunity to replicate.

However imperfect, the empirical approach at least satisfied the desire for evidence with regard to metrics. The use of sound statistical techniques to analyse data and test hypotheses lends credence to the conclusions drawn from the results. Additionally, use of "good" experimental practice such as triangulation (i.e. where the study involves measuring the same attribute in more than one way) and replication (attempting to

replicate results using other datapoints/datasets) can help improve the standard of empirical studies. Statistics can be misused, of course. Statistical analysis of the data collected does not necessarily provide absolute proof of a theory. Take Halstead's measures, for example ((Halstead 1972; Halstead 1977; Halstead 1979)) — his results appeared to support the usefulness and accuracy of the measures he proposed, but his metrics are now widely discredited, after the findings of other empirical studies disagreed with the results and queried the experimental design and statistics used((Hamer and Frewin 1982; Shepperd and Ince 1993)). In short, when attempting to validate software metrics we will always fall short of the accepted desiderata for an empirical study [11], thus for any validation, criticisms may be made of the conclusions drawn, the data collected, the techniques used and the assumptions made during analysis. With limited resources we must be pragmatic and accept trade offs, such as giving up control of how the software development process is carried out in return for getting data from industrial software, or by accepting student developed software in return for greater control on the empirical study.

2.3.3 Summary

It is implicit in the above sections on axiomatic and empirical techniques, that while many favour one over the other, and presumably my own bias towards empirical methods shows, they can be used most effectively as complementary techniques. An axiomatic validation can help to focus an empirical one, by demonstrating what should happen, and thus what

---

[11] The idea of "accepted" desiderata has evolved from the "good" and "bad" examples published in software engineering particularly. Good examples take problems into account, bad ones get criticised by other empiricists. The perfect empirical validation would be large scale, in situ, with controls, with an explicit hypothesis. Results would need to be repeatable, i.e. in other studies, be able to take into account the affects of

to look for. An empirical study can demonstrate whether or not the mathematical models are applicable in practice.

2.3.4. Difficulties in Validation

As discussed above, both of the available approaches to metrics validation have limitations. This is essentially because of the nature of software and the processes by which it is developed. Although we use the terms "science" and "engineering" with regard to software, attempt to build formal models and to impose discipline on the process of development, software is intangible. In science and engineering, predictions can be made based on what is known about the physical properties of a substance, prototype models can be built and tested, experiments can be carried out and results replicated, allowing reliable and useful formal models or prediction systems to be built specifying what would happen under what conditions.

Software, however, has little in the way of physical properties. There are some measurements that can be taken, the amount of disk space it takes up, the time taken to execute, for example. But they can tell us little about how the program will behave, how error prone it will be, how easy to maintain. We need to assign numbers to intangible properties in order to provide the management information required in order to plan, allocate resources, assess the success or otherwise of a project. We cannot know for sure what all of the factors that have affected the end software product are, or how much each has contributed. Therefore assigning meaningful numbers to these factors and assessing the end result is impossible to do with 100% accuracy or certainty. We must be satisfied with probabilities, significance, feasible explanations.

---

various factors upon the results such as the environment, problem area, personnel and so

Moreover, we must accept that a valid metric (empirically or formally valid) does not mean it is useful, and that a metric that is useful in one situation will not necessarily be useful in another, because of the myriad of affecting factors that are either unknown or unquantifiable. The goal or purpose behind the metric affects its usefulness — it may be valid in that it accurately captures a particular attribute, but if the goal is really to measure or predict something different, then the metric is not useful for that purpose.

It has been shown, however, that improvements can be made to the development and validation of software metrics. A number of papers were published on the subject of measurement theory and its relevance to software engineering. Some aspects of classical measurement theory, namely those that have been applied to software engineering measurement, will be discussed in the following section 2.4.

2.4 The 1990's: Measurement Theory and Prediction Systems

The 1980's saw empirical validation of software metrics and some unsatisfactory attempts to formally validate them. Measurement theory has been promoted as a way of formally validating metrics to complement the empirical approach which assesses the usefulness and applicability of metrics. Although measurement theory is not new, its application to the problem of software metrics validation started at the end of the 1980's ((Zuse and Bollmann 1989)), gaining momentum in the early 1990's which saw a number of publications on the subject, primarily by Fenton and Kitchenham, also Pfleeger and Zuse. This helped to clarify the terms in use, since the term metric has and continues to be applied to

---

on.

both a measure of some attribute and to a model which uses measures in order to generate a prediction about the software in question. The term "prediction system" was introduced to enable a distinction to be drawn between a metric as a measure of an attribute and metric being used as a predictor of an, as yet, unobtainable attribute.

The confusion between measures and prediction systems can be illustrated with the much maligned measure, LOC. It is a valid measure, which captures what it purports to, i.e., the number of lines of code in a program. Counting rules can be specified, such as not counting blank lines, but that aside, it properly represents the attribute code length. However, as a measure of complexity, for which it was often used, it is clearly unsatisfactory as it stands. It might to be used as an input into a prediction system which can then, for example, give an estimate of error rates. Of course, even if the prediction system into which it is input is not useful, LOC is still a valid *measure*.

Additionally a distinction between "valid" and "useful" is drawn (Fenton and Kitchenham 1991). Broadly speaking measurement theory can determine the validity of a measure and empirical studies can determine its usefulness. Fenton and Kitchenham (Fenton and Kitchenham 1991) give an informal definition of validity. A measure is considered valid if it "accurately characterises the proposed attribute" and a prediction system is valid "if it makes accurate predictions".

Further work by Kitchenham, Pleeger and Fenton (Kitchenham, Pleeger et al., 1995), breaks down measures (terms used are direct measures, indirect measures and predictive measures) and the measurement process into more elementary components, and discusses them with regard to their properties, in order to define a measurement structure model as part of a framework for validating software measurement. They

present a list of guidelines which they suggest should be applied in order to avoid major problems. The authors emphasise that attempts at software measurement should not stop until researchers can be certain the are correctly validate their measures. The suggestions made by the authors are ideals to aim for. The theoretical suggestions, such as satisfying the representation condition and using scale types correctly can be met by paying due attention to measurement theory. Suggestions involving empirical corroboration ("validation" by use of empirical techniques) are harder to comply with due to practical issues, such as availability of data.

Clearly empirical validation is necessary to effectively validate a measure in the fullest sense. An empirical validation will not give proof in the formal, mathematical sense, but will give evidence so that a hypothesis can confidently be confirmed or rejected. Additionally, since the validity of prediction systems is based upon their accuracy, the question of an acceptable margin of error is raised (how accurate must something be) as is the number of cases used in the hypothesis testing (how many are needed before the hypothesis can be accepted/rejected with confidence). There seems to be no one answer to either question. Acceptable accuracy will depend on the person using the system. The minimum number of cases will depend on a number of factors: on the test being carried out; on the quality of the data (e.g. student programmers as opposed to "real" programmers); on the plausibility of the hypothesis being tested. Conventionally confidence limit are set at alpha=0.05, i.e. we can be 95% confident of the correct outcome. Setting alpha too low means although we are unlikely to wrongly reject an the null hypothesis, we are less likely to correctly accept the alternative hypothesis. There will also be a personal view regarding accuracy and confidence, thus the validity of a prediction will always be open to question - it is as much a question of persuasion as "proof". The validation of a measure (metric for

assessment) can be carried out according to representation theory of measurement (see (Fenton 1991) or (Fenton and Pfleeger 1996)), by ensuring that the mathematical representation (measure or metric) of the attribute corresponds to the empirical world.

There has been a failure to recognise that a measure does not have to be valid (in the measurement theory sense) to be useful. Valid and useful are not one and the same thing. An example of this is Function Points (Albrecht 1979; Albrecht and Gaffney 1983; Albrecht 1984) and Mark II Function Points (Symons 1991). Both are very widely used metrics. These measures are not considered valid, one reason being the way in which the metrics are constructed, another the instability of model upon which the metrics are based (Kitchenham, Pfleeger et al. 1995). This means that both Function Point metrics could behave unpredictably, as demonstrated by a number of conflicting empirical studies (Low and Jeffery 1990; Kemerer 1993). However, the metrics enjoy widespread use and support among practitioners. Conversely there are any number of perfectly valid measures which can be taken directly from software, which are of little or no use, either because they are not available until late in the development process, because they do not tell us anything of interest nor are used in a useful prediction system, or because they are closely related to (i.e. tell us the same thing as) a more readily available measure. Pfleeger has published extensively on metrics validation and the application of measurement theory to software, but admits "a measure can be useful as a predictor without being valid in the sense of measurement theory" (Pfleeger, Jeffery et al. 1997).

Appearing to be useful does not necessarily mean that a metric really is useful. This is particularly true of many of the early complexity metrics, such as Software Science (Halstead 1977) or Cyclomatic Complexity (McCabe 1976). Both of which were the subject of many independent

empirical studies which supported the claims made to the metrics' usefulness with respect to a number of features (see (Shepperd and Ince 1993) for a discussion and review of work concerning both of these metrics). Such results are now regarded largely as erroneous [12], either because of flaws in the analysis, or because it is unclear as to what is being measured or tested.

2.4.1 Measurement Theory and its Application to Software Metrics

This section considers some of the concepts of measurement theory and examines how it has been applied to the measurement of software. The aim of the section is to provide a basis for any further discussion or application of aspects of measurement theory in this or subsequent chapters. It will provide a context within with metrics can be examined and criticised.

The essence of measurement theory is to provide rules and definitions for the process of measurement and the measures themselves. Measurement can be defined as the process of assigning numbers (or sometimes other mathematical entities or symbols) to some attribute of an entity in order to describe that attribute (Pfanzagl 1968; Krantz, Luce et al. 1971; Roberts 1979). The two main strands of measurement theory applicable to software metrics are representation theory and scales.

2.4.1.1 Representation Theory

This aspect of measurement theory is concerned with the mapping between the real, empirical world and the mathematical, theoretical world. Its purpose is to ensure that the "real" relationships between

---

[12] It seems that McCabe's v(G) may be able to predict branch coverage testing effort.

"real" artefacts and the rules which govern these relationships are represented accurately and are preserved in the mathematical model.

The formal explanation is as follows (from (Finkelstein and Leaning 1984)):

an empirical relation system is $Q = \langle Q, R \rangle$;

Q is the set of observations;

R is the set of relations on Qsuch that $R = \{R_1, R_2, ..., R_n\}$;

a numerical system is $N = \langle N, P \rangle$;

N is the set of real numbers;

$P$is the set of relations on N such that $P = \{P_1, P_2, ..., P_n\}$;

the measurement function M maps Q to $N, M : Q \rightarrow N$.

The mapping must be done so that the observed relationships between empirical entities hold for the numbers representing them. This is described by the following Representation Theorem:

P_i is a relation on N which corresponds to the relation R_i on Q;

if $q, r, ... \in Q$,

$$R_i(q, r, ...) \Leftrightarrow P_i[M(q), M(r), ...] \qquad (1)$$

for all $ii = 1, n$

The conditions under which this holds are representation conditions, thus if M satisfies the representation conditions, there exist other mappings, M' which satisfy these conditions and are related to via a transformation f, such that $M' = f(M)$. It follows from (1) that:

$$R_i(q,r,...) \Leftrightarrow P_i[[f[M(q)],f[M(r)],...]] \quad (2)$$

All f belong to the set of admissible transformations, a.k.a. the uniqueness condition. The scale of measurement, can be denoted as $S = \langle Q, N, M \rangle$ (see section 2.4.1.2 below for a discussion of measurement scales).

2.4.1.2 Scales

This process is governed by rules which determine the appropriate measurement scale and thus the legitimate operations or transformations which can be applied to the measures. Here we are primarily concerned with direct measures, such as length (i.e. those which do not depend on the measurement of any other attribute), as opposed to indirect measures (those depending on the measurement of one or more other attributes, such as density).

Stevens (Stevens 1946), introduced a classification of scales of measurement, on which the table below (table 2.2) is based.

Scale	*Basic Empirical Operations*	*Example*
Nominal	equality (~)	labelling
Ordinal	equality, greater or less (~, <, >)	Beaufort Scale
Interval	equality, greater or less, equality of intervals or differences (~, <, >, (X,Y~V,W))	women's dress sizes
Ratio	equality, greater or less, equality of intervals, equality of ratios (~, <, >, (X,Y~V,W), (X/Y~V/W))	length

Table 2.2: Measurement scales, legitimate operations and examples

The nominal scale allows numbers to be assigned as labels, (where words letters or symbols could also be used), the only rule being "do not assign the same numeral to different classes or different numerals in the same class" (Stevens 1946). The Nominal scale is a binary system, so that an entity either has some property or does not. Using the notation in section 2.4.2.1:

the empirical relational system is $\langle Q,\sim\rangle$, where ~ is a binary equivalence relation;

for $q,r,...,\in Q$, the representation theorem is

$$q \sim r \Leftrightarrow M(q) = M(r) \qquad (3)$$

The ordinal scale is an extension to the nominal scale in that it allows rank ordering of classes or categories which must be preserved. In the example given above, the Beaufort Scale assigns a number (which represents an estimate of wind speed) according to observed effects on land, e.g. smoke rising vertically is assigned 0, a hurricane is 12 on the scale. This is also a binary system. In addition to the relation given for the Nominal scale above (3), in the Ordinal scale the relation set, R also contains $\succ$, thus:

$$q \succ r \Leftrightarrow M(q) > M(r) \qquad (4)$$

The interval scale builds on the ordinal scale in that it preserves ordering, as in the ordinal scale, and also captures information about the size of the interval between classes. The classic example being temperature measured on the Fahrenheit or Celsius scales, a different example is women's dress sizes. If R contains the relation $\succeq$ where

$(q,r) \succeq (s,t)$ means the interval (q,r) between q and r is greater than or equal to the interval (s,t), then

1. $(q,r) \succeq (s,t) \Leftrightarrow M(q,r) \geq M(s,t)$ and
2. $M(q,s) = M(q,r) + M(r,s)$ (5)

As its name suggests, the ratio scale preserves the ratio between entities allowing us to say, for example, that one piece of string is twice as long as another. The starting point is an absolute zero (the total lack of attribute), increasing at regular intervals. We can therefore meaningfully add two measurements to create a third. So

1. $q \sim r \Leftrightarrow M(q) = M(r)$ and
2. $q \circ r \sim s \Leftrightarrow M(q) + M(r) = M(s)$ (6)

Another is the absolute scale, the most restrictive scale, where the actual count (of the number of occurrences of x in y) is the only possible measurement. An example of this might be the number of entities in an entity-relationship diagram see (Fenton and Pfleeger 1996).

The classification of scales allows us to determine which statements regarding measurements are meaningful, and which statistical or mathematical operations can be legitimately performed. Thus measurement theory applied to software metrics allows us to determine whether or not the measure is valid and meaningful. It does not however, ensure that a measure is useful. This needs to be demonstrated empirically, to test if a valid, measure or prediction system gives us useful, meaningful information or predictions about software. There are occasions where useful measurements are not meaningful according to Stevens' scales, for example the mean of a set of exam marks is

commonly used, despite the argument that marks can be assumed to be ordinal and thus mean would not be considered legitimate or meaningful (Finkelstein and Leaning 1984).

There are opposing views. Stevens' work on scales has been criticised by some statisticians and psychologists. (Baker, Hardyck et al. 1966) cite "statistically minded" psychologists and a statistician who argue that statistics apply to numbers not objects and thus statistical operations need not be limited to what is consistent with the scale properties of what is observed. Baker et al carry out a study using *t* values, as an example of a robust and commonly used statistic. The data was transformed to simulate typical situations (in psychological analysis) where inconsistencies with measurement scales might occur. The authors conclude that with minor reservations, the probabilities estimated from the *t* distribution vary little according to the measurement scale used.

Doubts regarding the issue of measurement scale are raised by Briand et al (Briand, El Emam et al. 1996), who cite a number of statisticians/data analysts holding the dissenting view that a pragmatic approach to measurement and analysis is necessary since an inflexible application of measurement theory has a detrimental effect on the volume of results. Briand et al consider this a to be a serious problem in software engineering, since as a relatively immature discipline, results of software measurement are scarce. The paper presents the view that since parametric tests have more power than non-parametric tests, then these should be used where possible, to avoid the possibility of not rejecting the null hypothesis because of a lack of statistical power, noting that researchers are more likely to conclude that the metric is not valid than to conclude that the weakness of non-parametric tests is the problem. However there is of course the opposing view that researchers are more likely to reject the null hypothesis anyway, since they are not necessarily

strictly impartial if their research has been geared towards supporting the alternative hypothesis.

2.4.1.3 Summary

Valid measures must represent a defined attribute of a defined entity. Measurement theory demands a rigour not previously associated with software engineering metrics, ensuring that we clearly state what we will measure and how, facilitating a common understanding of what we are trying to capture. Therefore, measurement theory leads us to question whether attributes such as complexity, the target of so many metrics, can ever successfully be captured. It is evident that complexity cannot be defined to the satisfaction of all and since it cannot be defined, for any measures claiming to capture complexity , it will be hard to demonstrate that they satisfy the representation theorem.

Measurement theory has clarified measures and prediction systems. Prior to the movement to apply measurement theory to software engineering, the distinction between a basic measure, capturing a definable attribute (e.g. lines of code) and a prediction of some as yet unknown characteristic (e.g. errors) was not made. With hindsight, it is inconceivable that this confusion went on for so many years and that few researchers/practitioners recognised the problem. Undoubtedly some recognised the problem but could not provide a solution, but the whole research area at the time was characterised by confusion and lack of clarity of purpose.

The contribution made by the application of measurement theory can be summarised as follows:

• it defines measurement, allowing us to differentiate between a measurement and prediction system, and thus appropriate tests can be applied;

• it can demonstrate validity of a proposed metric, via representation theory;

• it emphasises meaningfulness, since the use of appropriate statistical tests allow us to draw meaningful conclusions.

2.4.2 Prediction Systems

In order to classify a "metric" as the more specific "prediction system", it must be of the following form:

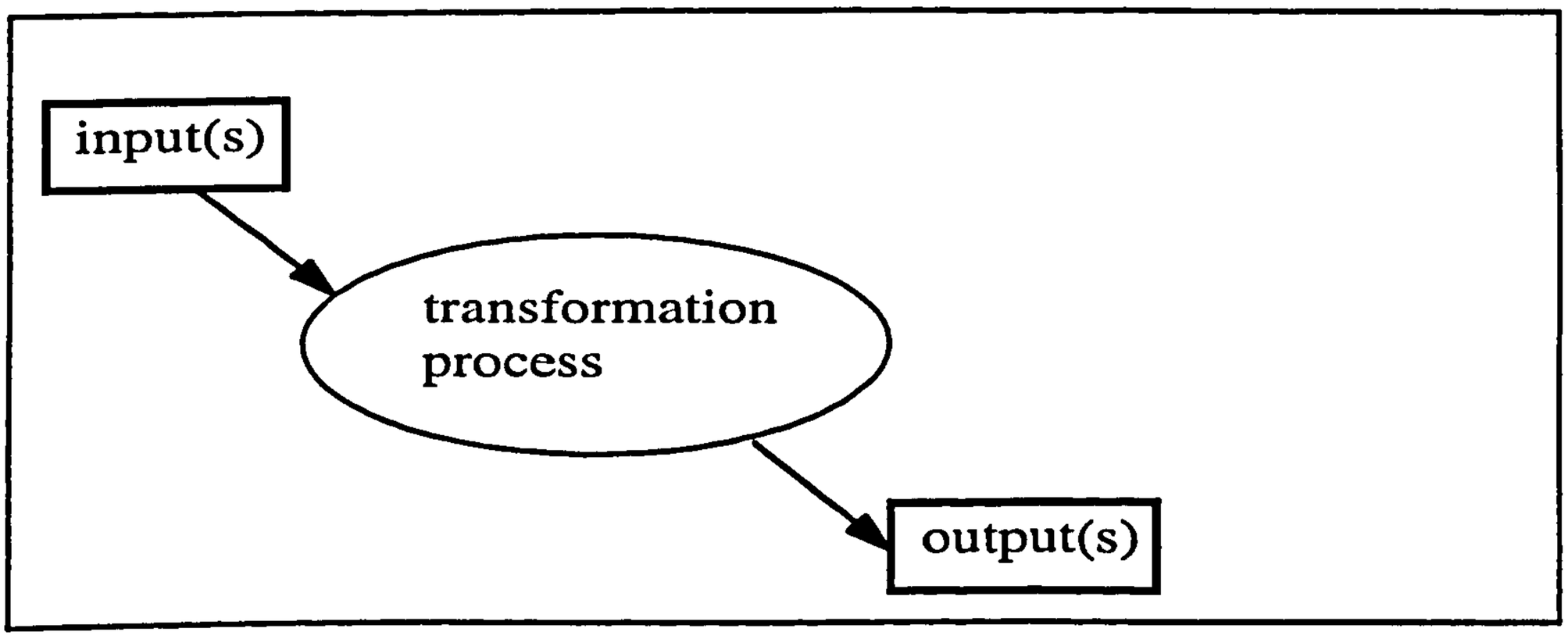

Figure 2.1: A prediction system

A prediction system must have at least one input and can have one or more outputs. The inputs undergo a transformation process, such as being fed into a predefined equation (or a series of equations), the output of which is the predicted software attribute. All parts of the prediction system must be clearly and unambiguously defined. This applies to the measurement unit used for the input and output, counting rules for collecting inputs, what steps are to be taken during the transformation process, using which inputs and producing which outputs.

Validation for a prediction system is different from that of a straightforward measurement (assuming inputs are validated separately). Prediction systems are probabilistic, and thus whether a prediction system is valid or invalid is of lesser importance than whether it is useful or not useful. However, the terms can still be used, especially validation process, but we are interested more in the accuracy of predictions than whether stringent measurement theory standards are met throughout. Therefore the emphasis must be on empirical validation, since this is the way to assess usefulness. The problems do not end there, since prediction systems can utilise widely different measurements as inputs in order to produce different types of output. Thus the techniques used will vary according to the type of data available and what is being predicted. We must be pragmatic in our selection, using the best techniques we can apply, but not rejecting validations because the techniques which can be used in one particular study are less sophisticated or sensitive than those used in another. An example of this is the validation of cost estimating prediction systems compared with validation of defect prediction systems. Because cost estimation involves data which is continuous, likely to contain a large number of data points, with high values, a large number of fairly sophisticated techniques can be applied to test the accuracy of its predictions. However, for a defect prediction system, the data is discrete, sparse and actual values tend to be low. Also the data tends to be skewed. Therefore the same range of techniques are not available. In this situation, although, for example MMRE (mean magnitude relative error) could be applied, it would not be useful because of the nature of the data — predicting one defect where there are actually two is a fairly close estimate, but in terms of MMRE it would be 100% wrong. Thus, for prediction systems such as defects, adjusted R^2 is often used. It is a test in which we can have some confidence, since it gives us the amount of variation in the dependent

variable which can be explained in terms of the independent variable. A problem which often occurs with empirical studies is that they present no explicit prediction system and rely on the correlation coefficient alone to indicate a relationship. This cannot be considered to have demonstrated the usefulness of a particular measure since it can be seen from measurement theory (and common sense) that without knowing exactly what is being assessed and how, we cannot adequately test the hypothesis nor repeat experiments for confirmation.

2.4.3 Lessons to be Drawn from Measurement Theory

Consideration of both classical measurement theory and the application of measurement to traditional science and engineering disciplines has highlighted failings in software engineering measurement and in the validation of software engineering metrics.

failing:
1 failure to distinguish between a measurement and a prediction system
2 attempts to capture poorly defined attributes
3 inadequacy, and often absence of validation for metrics
4 lack of goal or motive for using metrics or prediction systems
5 failure to establish validity and/or usefulness (since these are not one and the same)

Table 2.3: Summary of failings in software engineering measurement

1. First and most important is the *failure to distinguish between a measurement and a prediction system*, which is closely related to the fourth point, the tendency to overlook the need for goal or motive for using metrics. Where measurements are unavailable, e.g. number of errors per KLOC at design time, we require a prediction system. In the past, metrics have tended to stop short of this. For example many

metrics have attempted to capture system complexity in order to predict effort, defects etc. However, although various uses for complexity measures have been mooted, few have shown how they think complexity should be used in order to predict such attributes.[13] Thus, notwithstanding the issues regarding complexity measures, since in this case complexity is merely used as an example, these cannot be regarded as prediction systems.

2. The *attempts to capture poorly defined attributes*, or those attributes which are intangible and thus cannot satisfactorily be defined (quality and complexity being two such attributes). It follows that without a clear definition, the attribute cannot successfully be captured, at least with any consensus. Such "complexity" and "quality" metrics have captured other, more specific or definable attributes felt to have an effect on the complexity/quality of software. These measurements have then been used as a proxy for the desired attribute, the implication being that there is a high correlation between the two attributes, for example, the number of decisions plus one representing complexity according to McCabe's Cyclomatic Complexity metric. In order to be validated, one must be clear whether a measure is a direct one, or indirect (and thus needs to undergo some sort of transformation). Lack of definition can be extended to counting rules, where how or what to collect is often not explicitly defined, leading to a situation where different results can be obtained according to the practitioner's interpretation.

3. The *inadequacy, and often absence, of validation for metrics.* Measurement theory can assess the validity of a metric in how it captures the defined attribute, appropriateness of the chosen scale, etc. A further flaw in metrics development has been unsatisfactory (or entirely absent)

---

[13] Of course one might not always require a prediction, one may wish to compare two systems.

validation. There were attempts to address this problem during the 1980's, using axiomatic validation (Prather 1984; Weyuker 1988) (see section 2.3.1) and empirical validation (see section 2.3.2). Neither approach turned out to be satisfactory. First both approaches had flaws, and second they did not share the same criteria for validation and tended to be used in isolation, not as complementary techniques.

Both Prather's and Weyuker's axioms concentrate on complexity metrics. The rationale is that a valid metric will conform to the axiom set. Initially this seems a pleasing and formal way to validate a metric, but in practice, both axioms sets fail to give a satisfactory validation, although for very different reasons.

Prather's set of three axioms have limited applicability, being appropriate only for structured programs and control flow metrics. They are also weak, for example, despite known weaknesses in McCabe's Cyclomatic Complexity, the metric is valid according to Prather's axioms.

Conversely Weyuker's axioms are highly restrictive, none of the metrics used to demonstrate the set satisfy all of the axioms. See (Shepperd and Ince 1993) for a detailed criticism.

Both axioms sets are artificial, favouring mathematical formality over reality, avoiding real world problems. Thus the empirical approach is favoured by many. It has the benefit of being able to assess the usefulness of a metric and since it is a practical approach, observes how metrics behave in reality. Criticisms can and have been levelled at this approach, such as the lack of rigour in some empirical studies and the conflict between reality and control (the industrial vs. the laboratory setting), see section 2.3.2. Also, inappropriate use of statistical tests may result in

misleading conclusions being drawn, since results are invested with a significance they do not possess (see section 2.5.2.3, as an example).

Clearly neither approach can demonstrate the validity of a metric, Prather's axioms are too weak, Weyuker's too restrictive, and the empirical approach can be poorly designed or poorly conducted in practice. Measurement theory improves the situation, in that it offers a formal mathematical approach which need not be highly restrictive. It can become impractical and impenetrable, such as some of the work of Zuse (Zuse 1991), and has not been universally accepted, many statisticians feeling that the interpretation of measurement theory is overly restrictive and not pragmatic. See (Briand, El Emam et al. 1996) for a criticism of measurement theory when applied strictly.

The implication is that any approach loses value when taken to extremes. It seems sensible to apply measurement theory to ensure a well defined attribute and measurement and the use of appropriate statistical techniques. However, the real world vs. mathematical world debate is still an issue, and so pragmatism (as embodied by the empirical approach to validation) is also called for. The two are complementary and exert a modifying influence over each other, with the empirical approach introducing an element of practicality and the concept of usefulness, and measurement theory ensuring that the empirical approach conforms to acceptable levels of mathematical rigour. It is after all important to ensure that the metric are actually measuring what they are intended to measure. We need to assess that the mathematical model captures what we see in the real or empirical world.

The two approaches have been exploited to suit metrics research. Statistical tests and data have been chosen to give support to the desired outcome (as with Henry and Kafura, above, where inappropriate

methods of analysis have been used) or have been ignored in favour of an entirely formal validation (often using Weyuker's axioms) where lack of data, time or resources means an empirical validation cannot be carried out. Since the two approaches are complementary, ideally a combination of the two would be used to lead to a validation in which we can have confidence: use measurement theory to ensure that appropriate statistical tests are used, and thus meaningful conclusions can be drawn; use empirical validation to determine how the metric behaves in reality, its usefulness[14] and practicality (such as ease of collection and ease of calculation). This thesis concentrates on the more time consuming empirical approach at the expense of the formal, measurement theory approach, simply because that is my bias. It should be reiterated that I am aware of the principles of measurement theory and thus have attempted not to violate them, even though this thesis does not include a measurement theoretic validation of the metrics presented.

4. The lack of a *goal or motive for using metrics or prediction systems*. The need for a goal, besides being common sense, has been highlighted in the GQM (goal question metric) method (Basili and Rombach 1988; Rombach and Basili 1990). Where metrics are chosen according to the goals or aims of measurement (identification of goals is the first step in the process). Unfortunately, all too often the primary focus has been on the metric alone, with apparently little or no thought given to the model on which the metric or prediction system is based or as to why measurement is required. Metrics are only useful as part of a larger activity, assessing software quality or making predictions for project management, resource allocation and so on. It is not uncommon to read a metrics text book or paper (those aimed at practitioners) and see

---

[14] Pfleeger et al., admit that measures can be useful "without being valid in the sense of measurement theory" (Pfleeger, Jeffery et al. 1997)

lists of metrics to collect, with little or no explanation as to how and why they are to be collected, nor how the results are to be used (recent examples being those by (Lorenz and Kidd 1994; Henderson-Sellers 1996)). It would therefore be reasonable to assume that there is no clear goal or motive, nor explicit definitions, and thus no validation. After all, if the authors had done such work, then surely they would publish whatever they could in order to bolster their claims for the metrics they propose. There seems to be a tendency to propose lists of metrics covering every aspect of, for example, code, rather than considering what information is important, how it can be obtained, and how to use the results and to what purpose. Presumably this is because it is easier to suggest measures/prediction systems, even seemingly complicated ones, than it is to define its purpose and to validate the metric against this i.e. does it satisfactorily fulfil its purpose.

5. Finally, *failure to establish validity and/or usefulness (since these are not one and the same)* A measure may be valid, but convey little useful information. For example LOC has been criticised as a poor metric. However it is valid, since it captures precisely the length of a program in terms of the number of lines of code. It's usefulness in conveying other information (on its own or as part of a prediction system) is a different issue, to be assessed by empirical observation.

2.5 Metrics Revisited

We can now re-examine metrics development using the insights provided by the application of measurement theory as discussed in 2.4 above. This section will examine the main themes and most influential metrics (as outlined in section 2.2 above) in the light of what has been learnt from the application of measurement theory to software

engineering. The first section, 2.5.1 is a general criticism of the hazy concepts which software measures and prediction systems attempt to capture. The second section, 2.5.2 applies these issues (detailed in 2.4.3) to the metrics. Section 2.6 summarises the approach and emphasis metrics development and validation should take if it is to move from an undisciplined, arbitrary field, to one where proposed measures and prediction systems can be accepted with some confidence as to their validity and usefulness.

2.5.1 Describing the Indescribable

Complexity, quality, and so on are much used terms by researchers and practitioners. Both groups are guilty of devoting time, attention and resources to considering and measuring these attributes without having first satisfactorily defining them e.g. (Jones 1978; Henry and Kafura 1981; Weyuker 1988; McCabe and Butler 1989; Henry and Selig 1990; Maus 1992; Lee, Liang et al. 1993; Lorenz and Kidd 1994; Constantine 1997; de Champeaux 1997). This is, of course extremely difficult. The concepts are subjective. Although a group may have some general agreement on, for example, whether one program is more complex than another, they would be hard pressed to quantify this, and could not produce a set of desiderata which would be applicable in all situations and satisfy all those involved. Even for tangible objects, the concept of "acceptable quality" varies from person to person. For example, the US Food and Drug administration, sets "acceptable" levels of contamination of food stuffs (i.e. below which the food is considered to be of "unacceptable quality"), such as: tomato paste, 30 fly eggs or 15 fly eggs plus one lava in a 100g sample; peanut butter average of 30 or more insect fragments, or one or more rodent hairs per 100g sample; canned mushrooms, up to 20 maggots per 100g of drained mushrooms. It is questionable that anyone

would happily eat a mushroom pizza if they were informed that it contained up to 30 fly eggs and 20 maggots!

Thus for software, something that is intangible, the situation is even less clear. We can model the software according to "good" design practice and test the end product in an attempt to remove faults, but we cannot produce universally acceptable criteria by which to measure concepts such as complexity or quality because we cannot define it.[15] Unfortunately much of the work in the field of software metrics has concentrated on the pursuit of software quality or software complexity metric. These are the metrics which have been taken up by industry, in preference to more simple measures or prediction systems.

2.5.2 How metrics "measure up" -- no pun intended

The metrics considered in sections 2.1 and 2.2 purport to capture complexity or a related characteristic. The definitions and factors thought to contribute to software complexity vary. We will now re-examine some of the more popular metrics. Recurring themes/criticisms will run throughout the following sections, since each has something in common with others, such as poor definition or poor empirical evidence, for example. In particular, the inadequacies summarised in 2.4.3 will be considered as applicable. These are listed below, refer to 2.4.3 for more detail:

- failure to distinguish between a measure and a prediction system;

---

[15] Gilb suggests that operational definitions, which can be measured, should be produced (Gilb 1988). However, we still face the problem of an agreed definition, subjectivity etc., and so would need to accept that there can be no single, universal commodity. A workable definition may still fail to capture important aspects of the attribute.

- poor definitions;
- inadequate validation;
- absence of clear goal or purpose;
- failure to establish validity and/or usefulness (since these are not one and the same).

2.5.2.1 Halstead's Software Science

Halstead intended Software Science as a prediction system, the assumption that complexity was the goal for all of the Software Science metrics is an error or misinterpretation which must be blamed on others, since although the language level is related to complexity, many subsequent researchers have treated Software Science as a whole as a set of complexity measures. Although the distinction between measurement and prediction system was not current at the time, it was clear, that however flawed its assumptions, Software Science was intended primarily to predict effort. Confusion may have arisen from its emphasis on complexity, with some practitioners and researchers (including Halstead himself) assuming that complexity could be used as a proxy for other attributes, such as bugs and quality.

However, the allegation of having poorly defined attributes, can indeed be levelled at Software Science. Halstead made a comprehensive attempt to draw together various factors (Halstead 1972; Halstead 1977; Halstead 1979) affecting software. He believed that, as with the physical sciences, a set of fundamental laws, which would remain true whatever the development environment, could be produced for software development. Halstead did not study the code in isolation, but also included the idea of cognitive complexity. This was an attempt to

include the programmer's perspective on the code. Halstead felt that in addition to the structure of the code, the cognitive effort of the programmer needed to be included before the complexity, and thus, effort could be predicted. This is a valid point. Complexity is subjective; what one programmer regards as complex may not be a problem for another. Halstead claimed to use the results of cognitive psychology studies in his prediction system. However, subsequent examination of his theories suggests that the theories have been incorrectly and selectively applied (and are themselves questionable). Halstead proposed that the number of parameters in a module should be six (five input and one output), based upon his interpretation of Miller's research into short term memory and sensory stimulation (see (Coulter 1983)). However, this is not the same sort of activity as programming. Additionally it cannot be assumed that parameters necessarily indicate the complexity or type of function. Cognitive research indicates that the amount of information which can be held in short term memory when carrying out a task depends on the task itself. With an easy task all of the brain's resources can be used to maintain the necessary information in short term memory, but a more complex task means more resources are needed for processing and thus cannot maintain as many items in short term memory.

Another misinterpretation seems to concern the "Stroud Number". Stroud conducted research into sensory memory processing. He used the term "psychological time" defined as "the time in which we are aware of things happening" with regard to sensory input and operations upon the input. He stated the ratio was somewhere between 5 and 20 moments of psychological time to each second of physical time, suggesting a figure of 10 as most likely. Halstead again generalised the research and applied it to programming, suggesting that between 5 and 20 mental discriminations with regard to programming were possible, suggesting

an average figure of 18 from an empirical study. Clearly programming is not an activity based in sensory memory. A further problem is that Halstead assumed that human memory used a binary search which is not supported by cognitive psychology research.

Halstead did posit a model for Software Science and saw the importance of having a model, presumably from his studies of other sciences, where the notion of defining theory and a model, from which a hypothesis can be derived and tested has long been the norm. Indeed the process followed by Halstead is laudable, considering that software metrics research was still very much in its infancy, and even more so when compared with other metrics development around the same period.

However, as is now generally known and accepted, Halstead's metric is invalid. Supporting empirical evidence has been shown to be dubious (Hamer and Frewin 1982; Coulter 1983; Shepperd and Ince 1993). Hamer and Frewin, for example, criticise the standard of experimental design, and conclude that the experiments to test the hypotheses "are virtually incapable of rejecting the hypotheses — they simply do not have the power to identify false hypotheses." They also report errors in the test data used in (Funami and Halstead 1976), which was based on (Akiyama 1971).

Thus although Halstead's model was inspired by cognitive research, it was ill founded. Further criticism of Halstead's metric can be found in (Hamer and Frewin 1982). It's value lies in the way it was presented — an explicitly defined model of program complexity. Although both the theory behind Software Science and the validity of this prediction system has been debunked, Software Science is noteworthy in that it was based upon a theory and model, and attempted to draw together the various factors which contribute to software complexity. Its most negative effect

was that it laid down the challenge to other researchers to discover the ideal complexity metric[16], which could be used to predict all complexity related software attributes, such as maintainability and effort.

2.5.2.2 McCabe's Cyclomatic Complexity

McCabe's Cyclomatic Complexity Metric (McCabe 1976) is a code complexity metric which has less to commend it than Software Science, in terms of definition and theory. How to apply it as a predictor of errors, development effort, and so on, is not defined, but that has not prevented its use as a prediction system for almost any attribute thought to be related to complexity, e.g. "programmer performance", defined as the time taken to locate and fix bugs (Curtis, Sheppard et al. 1979), maintenance effort (Gill and Kemerer 1991). Research, reflecting on its application as a prediction system, suggests Cyclomatic Complexity is little more than a size indicator, since it correlates strongly with LOC (Shepperd 1988), though of course, since it is a code metric, LOC would be readily available anyway. Basili and Perricone, found it a poor indicator of error density, in fact error density decreased with increasing Cyclomatic complexity (Basili and Perricone 1984). In general, the independent empirical validations carried out are unsupportive. An exception is Henry et al, (Henry and Kafura 1981), whose study shows a strong correlation between Cyclomatic Complexity and error rates by module. However, error free modules were not included in the study, thus making the results questionable.

Essentially the Cyclomatic Complexity is equal to the number of decisions in a procedure or module plus one. McCabe suggested an upper limit to

---

[16] Given Halstead metric had no real goal nor use, the need for a metrics or prediction system to have a goal and to be demonstrably useful in some situation, seemed not to be picked up on by the next wave of metrics researchers.

program complexity (10 in most situations). Shepperd criticises the metric on a number of points, including that it is inconsistent with the accepted thinking on modular software — Cyclomatic Complexity increases with modularisation and other accepted notions of good programming style for improving program structure (Shepperd 1988), which is borne out by other studies (Baker and Zweben 1980; Prather 1984). This confirms that the underlying model, such as it is, is flawed, since relying on the metric for guidance regarding program structure would be misleading.

Cyclomatic Complexity also suffers from being ill defined. McCabe is suggesting that most software properties can be derived from, and thus are linked in some way to the number of decisions in the program code and does not adequately describe what a decision can be. For example, IF statements are counted but not ELSE statements. Without well defined counting rules, the metric may not be applied as intended, and further empirical work may be based upon incorrect data. Indeed no attempt was made to explicitly define complexity, or even some of the factors which might contribute to code complexity (a criticism which cannot be levelled at Software Science), it was simply assumed that readers had a shared understanding of the issue, illustrated by McCabe's statement that his complexity measure "is designed to conform to our intuitive notion of complexity". He seems to have missed the point of a complexity metric; if everyone were able to rely upon their intuition and shared understanding of complexity, there would be no need to quantify it.

In conclusion, Cyclomatic Complexity could be regarded as a measure which counts the number of decisions in a program, which could go some way to indicating complexity, but it adds 1 to the count, causing problems (loses additivity). However many factors are ignored, and as discussed previously (see 2.2.1), complexity itself is both difficult to define

and capture, thus Cyclomatic Complexity cannot be considered as a prediction system. This means its usefulness is, to say the least, limited[17]. Its potential usefulness and its validity are further compromised by the absence of a clear and explicitly defined goal. Without a goal, any empirical validations carried out are somewhat speculative, depending on what the investigators think the goal to be, else the investigation is carried out to see to what use the metric can be put.[18] This is apparent to the wide variety of attributes that empirical studies have attempted to link to Cyclomatic Complexity.

2.5.2.3 Henry and Kafura's Information Flow Metric

The search for a code complexity metric influenced subsequent metrics research. It was recognised that metrics applied earlier in the development cycle would be of more use, but researchers into design metrics still aimed to develop complexity metrics, the idea being that a complex design is indicative of complex code. The classic system design metric, Henry and Kafura's information flow measure (Henry and Kafura 1981) considers both the internal complexity of a module, and the external complexity in terms of the information flows between it and other modules in the system. It is meant then, as a prediction system, to predict module code complexity at design time. However, the purpose of the prediction remains unclear, what can be indicated from the module complexity? In the papers concerning the metric (Henry and Kafura 1981; Henry and Kafura 1984) and in Henry's doctoral thesis, a number of motives are given, such as controlling complexity, high development costs, high maintenance costs, providing guidelines for software design

---

[17] It seems that the only conceivable use would be to predict effort for branch coverage testing.

and improving software reliability, but how the module complexity measure should be used to predict such attributes remains undefined.

Internal complexity is measured in LOC. An obvious drawback is that actual LOC is not available at design time, limiting the usefulness of the information flow metric, as originally defined, as a prediction system. Another drawback is the usefulness of LOC as a complexity measure, although, some studies have shown LOC to outperform complexity measures such as Cyclomatic Complexity, as a predictor of some quality measures (Kitchenham 1981). Studies on the effect (of using LOC as a measure of internal complexity) on the performance of the information flow measure are varied, some finding that using LOC improves the performance of the metric (Kafura and Reddy 1987; Rombach 1987) and others finding that its use detracts from the metric's performance (Henry and Kafura 1981; Shepperd and Ince 1991). External complexity is based on the desiderata for good design of minimising coupling and maximising cohesion (Stevens, Myers et al. 1974) and is calculated by counting the flows[19] into and out of a module and squaring the total In addition to attempting to capture a concept so difficult to define as complexity, the metric suffers from further ambiguity since the counting rules are open to interpretation, due to hazy and apparently conflicting definitions of indirect flows. In other words it is not clear the inputs to the metric are to be calculated. Thus we can not be sure of what we are attempting to capture indirectly, nor whether the means by which we are attempting to do so is correct.

Unlike Cyclomatic complexity, the model associated with Henry and Kafura's information flow measure considers modularity and system

---

[18] Although it makes far more sense to define a goal and then attempt to reach it, it is common to find that a metric is first devised and then attempts made to see what use it can be put to.

architecture by incorporating both internal complexity (of a module) and external complexity (connections to other modules and data structures within the same system. A number of flaws can be seen when the metric is applied.

It is not clear what it is trying to measure/predict and for what purpose (see the introductory paragraph in this section). A second criticism is that internal complexity is defined as LOC, which has been widely criticised as a complexity metric and is unavailable at design time (see the second paragraph in this section), LOC for the internal complexity component would therefore need to be estimated. Alternatives to LOC, suggested by Henry and Kafura, Software Science and Cyclomatic Complexity, are also code metrics, thus suffering from the same problem of availability, (Henry and Kafura were presumably unaware of the flaws in these metrics, see sections 2.5.2.1 and 2.5.2.2 above]. Much more attention has been given to the external complexity component of the model, but the definitions of the inputs to this part of the model are hazy and conflict, particularly regarding indirect flows. The terminology used is inadequately defined, as with other metrics, a shared understanding is relied on. Shepperd (Shepperd and Ince 1993) points out that some cases of indirect flows will be detected only on analysis of a module's code, i.e. they will not show up at design time. In addition, Henry and Kafura make the decision that a flow should be followed (and counted) over no more than two levels of the system. There is no reason given for this. Questions have also been raised as to what indirect flows correspond to in the real world (Ince and Shepperd 1989). Other flaws become apparent when attempting to apply the model. One is that it penalises reuse where the reused module has information flowing in or out of it, since the flows between the module and all that use it are counted, making it

---

[19] Unfortunately, control flow and information flow are not distinguished.

appear more complicated than it is, since the more it is used, the higher its complexity becomes. (Benyon-Tinker's metric (Benyon-Tinker 1979), counts a module only once, however many times it is called, but this distorts the picture, since the system will always appear as a tree structure). Conversely, modules communicating via a "global data structure", are not calculated using flows, but instead a simple count of accesses is used, potentially leading to a lower figure (and thus seemingly encouraging the used of global data over local data). Any module with a zero fan-in or fan-out will have a complexity measure of zero because of the formulation of the model equation, where clearly a module can have functionality (and thus complexity) even if information flow is one way. Further, no consideration seems to have been given to the type of information flow. All flows are assumed equal, when this will clearly never be the case in reality.

The poor definition of the Information Flow metric makes validation difficult, since studies are not necessarily assessing the same metric (its value will depend on the definitions and counting rules used). Henry and Kafura's own empirical validation of the information flow metric (Henry and Kafura 1981; Henry and Kafura 1981), assumes a normal distribution of data and thus uses parametric tests when in fact the data is skewed, necessitating the use of non parametric tests and also remove two observations (from a total of eight) as outliers. When reinstated, the correlation coefficient drops considerably (to the extent it can no longer be considered significant). A further point, connected with poor definition of attributes, is the lack of clarity regarding the hypothesis under test. There is also the possibility of manipulating statistics and data (removing data points, for example), in order to increase the statistical significance of results. Shepperd (Shepperd and Ince 1993) suggests at the very least, an empirical validation must posit an

unambiguous hypothesis which it is possible to reject, and use appropriate statistical techniques.

2.5.2.4 Function Points

Beside the functional design/code complexity metrics considered above, other metrics are open to the same criticisms.

One such metric is Function Points (Albrecht 1979), as introduced in section 2.2.3. Function points has been much debated, since despite the lack of supporting validation, it is widely used in industry, and considered a useful metric, with a user's group dedicated to its application and improvement.

Problems exist both with the "model" and the validation of the metric. It is not possible to interpret function points as a measure since it has inputs, suggesting that it is intended as a prediction system. As a prediction system it lacks clarity of purpose and of definition, both in its collecting procedure and in its counting rules. As such there will be differences between practitioners' calculations. Function point users would counter such criticism by drawing attention to the official user groups (e.g. the International Function Point User Group) who publish counting practices and the fact that function point counters can take an exam and be certified as proficient in function point counting.

Shepperd (Shepperd 1994) summarises the results of eight empirical studies which report widely varying results for function points as a predictor of effort, with R^2 values from 0.9 to 0.18, although typically the R^2 value is low, thus indicating it is of dubious value. Where MMRE is performed, these tend to be disturbingly high (103 % for Kemerer

(Kemerer 1987) and 99% for Shepperd and Turner (Shepperd and Turner 1993)). Studies also report variation in variation in the range of counts (i.e. different counts for the same system) (Low and Jeffery 1990; Kemerer and Porter 1992), since some aspects are subjective.

Function points certainly would not survive a formal validation since the model, such as it is, is poorly thought out. When described in simple terms it can be seen for what it is — playing with numbers rather than a well thought out process. The following summary is taken from (Kitchenham, Pfleeger et al. 1995) "Albrecht's model involves classifying each input using an ordinal scale (simple, average, complex) according to the number of data elements and logical files involved, mapping those values to numbers and summing the numbers."

Measurement theory (scale types - see section 2.4.1.2) tells us (and it is common sense) that we cannot sum ordinal measures, thus this is meaningless. (Kitchenham, Pfleeger et al. 1995) point out a further violation, since the smallest value a non-null system can take is 3, thus function point values are discontinuous (moves from 0 to 3, 4, 5 etc.) and is without a unit value, thus comparisons such as *system x* is twice as complex as *system y* cannot hold.

Further criticisms of the model/theory upon which it is based can be made. Function points are intended to be adjusted to suit the local environment. In the empirical studies summarised in (Shepperd 1994), unadjusted function points perform at least as well as the adjusted function points. This would indicate that the adjustments have no beneficial effect on the predictive power of the prediction system. Additionally, studies (Jeffery and Stathis 1993; Kitchenham and Kansala 1993) have found dependencies between the inputs (function types) which form the unadjusted function point, indicating instability, since if

such inputs are strongly correlated, they effectively capture the same phenomenon, and thus its impact will be increased, as demonstrated by the fact that the two studies found different correlations between function point elements. Further, the values for the weights supplied were based upon project data from IBM in the 1970s, which is not necessarily applicable to other environments. Albrecht describes the weights as reflecting the value of the function from the customer's point of view, but in reality this has little bearing on effort or cost. Another criticism of the model and validation upon it, is that although it is claimed to be language independent this is not borne out in Symons' evaluation (Symons 1988), which indicates that it is, in fact, dependent, a point reiterated by (Verner, Tate et al. 1989).

Symons introduced Mark II Function Points (Symons 1991) to improve upon the inadequacies of Albrecht's function points, but some remain, particularly confusion over what is being measured and the complexity of the system making calibration difficult (Shepperd 1994). (Kitchenham, Pfleeger et al. 1995) identify three size attributes present in Mark II function points, which are then individually weighted to represent development effort, and then summed. The authors consider this process acceptable if Mark II function points are used as an effort model, otherwise the authors' measurement validation framework (based on measurement theory) is violated. The reason being that there is no theoretical model of the relationship needed in order to convert function points into a unit of size or functionality. There are also inconsistencies in the weights given, an example being input and output size are measured in the same unit but are given different weights.

Thus for both types of function points there is confusion about what is actually being measured, and until this is established it cannot be successfully validated. Such criticisms appear to carry little weight with

practitioners — as illustrated by a recent article (Furey 1997) claiming that function points were technology independent, repeatable and consistent and could provide valid estimates and the basis for valid comparisons. These statements appear to be based wholly on personal observation and possibly anecdotal evidence. Perhaps Pfleeger's statement, "a measure can be useful as a predictor without being valid in the sense of measurement theory" (Pfleeger, Jeffery et al. 1997), could be applicable here.

2.5.2.5 Summary of Common Product Metrics

The metrics criticised above are merely well known examples of the many complexity metrics proposed. All complexity metrics can be criticised for attempting to capture an inadequately defined attribute. Some, such as Halstead's Software Science have attempted to define and include the various facets of complexity with regard to software, but it is an impossible task. It is hard to believe that so much effort has been expended on such an ill defined goal, particularly since software complexity itself is not what we wish to predict. It is of interest because of the perceived relationship between complexity and other attributes, such as maintainability, errors, and effort.

2.5.3 The State We're In / Predicting the Future

As mentioned previously, there has been a lack of clarity in the derivation and use of metrics. Firstly the term metric has been used to cover both direct measurement (e.g. LOC) and prediction systems, (e.g. Haltead's E metric). This has been especially true with complexity metrics. Metrics requiring hard to capture or calculate inputs or even estimations of inputs (consider Henry and Kafura's information flow

metric, or function points), have been proposed in order to assess or predict complexity. However, a measure of complexity is of little use in itself, since the information required is a prediction of something more tangible, the number of errors, development time in person-days and cost to develop, for example. However, researchers (and the practitioners who embraced the metrics) seem to have been content to leave this matter unresolved for many years, since it was not until the work by the likes of Fenton, Kitchenham and Pfleeger published in the early 1990's that the distinction was drawn and definitions offered. An excellent example is Fenton and Kitchenham's paper (Fenton and Kitchenham 1991) which considers the use of measurement theory in validation. They emphasise the necessity for clear and unambiguous definitions of what is being measured, how and for what purpose before a satisfactory validation can take place. It is noted by (Fenton and Kitchenham 1991) that in some cases the distinction between an indirect measurement and a prediction system are not always obvious and so validation in both senses should be performed.

2.6 Summary

To date, many metrics have been proposed and debunked. Measurement theory has identified areas which metrics validation and construction must address. Empirical evidence is vital in order to assess the usefulness of prediction systems, since mathematical validity alone cannot tell us how accurate a prediction system is. The following lessons learned from the successes and failures of over twenty five years of software metrics research are summarised as follows (see section 2.4.3 for a fuller explanation):

- failure to distinguish between a measure and a prediction system;
- poor definitions;
- inadequate validation;
- absence of clear goal or purpose;
- failure to distinguish between validity and usefulness.

These lessons must be applied to future developments in order to prevent the same mistakes being made. However, it is obvious that even today, these mistakes are continue to be made and warnings must be repeated (Pfleeger, Jeffery et al. 1997).

The process of metrics development and the application of measurement theory to this process holds true whatever the software paradigm used. Thus although different characteristics need to be assessed, and different inputs will be used to make different predictions, the way in which we derive these measures and predictions systems and validate them need not change.

The analysis of software metrics development and validation presented in this chapter emphasises the need for validity in three respects.

First, measures should be valid, ideally, whether they are simple measures of assessment or inputs to prediction systems. This can be achieved by the application of measurement theory, to ensure that the metric satisfies the representation conditions. Axiomatic validation can be used to ensure it does not violate the belief of how it should behave[20].

---

[20] As previously discussed, axioms themselves can be proved/validated since they come from beliefs of how the world should behave, although, as in the case of Weyuker (Weyuker 1988), sets can be shown to be inconsistent or contradictory. Thus the axioms should be carefully chosen - since they can only demonstrate the metric's consistency with the axiom set, we need to be satisfied with the validity of the axiom set itself.

Second we need to ensure that the goal and process model of the measurement or prediction system is clearly stated. If not, testing its validity or usefulness will prove difficult, because of the likelihood of differing interpretations.

Third, we must ensure that prediction systems are useful, in that they make accurate predictions and predict something useful. This can be tested by empirical studies, though we are limited to determining the probability that a prediction system is accurate. Additionally the scope of the prediction system, that is, in what situation and environment can it be applied with confidence, must be made clear. This indicates the need to move away from the traditional quest for a metric applicable in all situations and environments.

Thus, metrics proposed for the object-oriented paradigm, examined in the next chapter, need to conform to the same standards of validity and usefulness as traditional metrics. These metrics must be clear whether they are prediction systems or measures of assessment. The attributes to be captured (for assessment or as inputs to a prediction system) must be clearly defined, as must the rules by which they are derived and the purpose of the measure or prediction system must be made clear. Metrics must be associated with a model, to provide meaning and a means by which to validate the mathematical representation of the real-world entity or attribute, which together with the empirical validation will allow us to make a reliable assessment of the metric's accuracy in capturing the intended attribute (in the case of a measure) or the accuracy of its predictions (in the case of a prediction system). The following chapter (chapter 3) will bear these points in mind when examining the metrics proposed for object-oriented systems.

Theme/Lesson	*Software Science*	*Cyclomatic Complexity*	*Henry and Kafura's Information Flow*	*Function Points*
distinction between measurement/prediction system	not explicit,	no	no	no
definition: attribute/counting rules	poor/poor	poor/OK	poor/contradictory	poor/contradictory
validation (supporting)	empirical unsatisfactory	empirical unsatisfactory, not compelling	empirical unsatisfactory, not compelling	empirical unsatisfactory
clearly defined goal	effort	no	no (give many and varied possible) applications	no
valid/useful	no/no	no/in one situation	no/could be useful for maintainability (Rombach 1987)	no/possibly in some situations

Table 2.4: Comparison of Traditional Metrics

Blank

In

Original

Chapter 3 Metrics for Object-Oriented Software

Synopsis

The widespread uptake of object technology has ensured that a significant number of metrics researchers and practitioners have turned their attention to measuring object-oriented software. Despite calls advocating the use of empirical evaluation and validation, many metrics continue to be proposed with little empirical evidence to support them. In fact, it can be seen that despite there being lessons to learn from the earlier years of metrics development, the message seems not to have reached many of those working in the area. This chapter will consider some of the object-oriented metrics proposed so far, paying particular attention to design complexity and quality metrics. The metrics will be described and assessed according to the points raised in 2.4.3.

3.1 Introduction

Although applying measurement to a new paradigm, we still need to consider why we are measuring and how this is to be done. Despite prior experience in software metrics to look back on, the answers to these questions remain, in the majority of work, as hazy as they ever were for the metrics developed, applied and validated prior to the advent of object-orientation.

This chapter will consider a number of metrics for object-oriented software. Firstly we will examine the attempts to apply traditional metrics to object-oriented systems (section 3.2). Then we will consider the exclusively object-oriented metrics (section 3.3). The previous chapter (2) highlighted a number of points to consider, or lessons to be learned, based on past mistakes and the application of measurement theory to software metrics, namely:

(i) The lack of a clearly defined goal;
(ii) The failure to distinguish between measures and prediction systems (metrics can be taken to mean either);
(iii) Poor definition of attributes to be captured and the counting rules for doing so;
(iv) Poor validation;
(v) Failure to establish validity and/or usefulness.

These will be considered throughout the examination of object-oriented software metrics.

3.2 Recycling Metrics (the application of traditional complexity metrics to object-oriented systems)

A minority of the research into measurement of object-oriented systems has attempted to apply traditional metrics to object-oriented software. Unsurprisingly, activity in this area does not seem to have been sustained. These early attempts were quite possibly influenced by tool support. Inevitably there will be a lag between the introduction of a technology and the tools necessary to support it. Until there is some call for object-oriented metrics to be incorporated into a tool, the developers of such tools are unlikely to include "new" metrics and are of course limited to those metrics already proposed, since they tend not to be in the business of developing and validating metrics themselves. Thus metrics practitioners working on an object-oriented project, probably for the first time, would need to make do with what was available and the suggestion that familiar metrics would work for object-oriented software measurement would be very appealing. The first published analysis and design method, Shlaer-Mellor (Shlaer and Mellor 1988; Shlaer and Mellor 1992), deliberately used familiar notations and models, specifically an entity-relationship model (known as an information model), state

models, using a familiar state-transition notation and a dataflow diagram, again familiar to those involved in structured design. This could give the impression that object-oriented design was not so very different and that traditional metrics would still be applicable.

The idea that traditional metrics (i.e. for structure design and code) could be successfully applied to object-oriented software, is intuitively implausible[21]. Two of the selling points of object-oriented technology have been as follows. First, that it is a new and better way of developing software, that it is a different way of modelling the real world (by focusing upon objects as the "building blocks" and encapsulating the associated data and processes within them, rather than creating an artificial division between data and process). Second, it is a different way of executing a program (using mechanisms such as dynamic binding and polymorphism). Thus metrics based on a structured, top-down approach to design and coding seem unlikely to be useful for a technology which works by interaction or co-operation between objects rather than requiring a module to be controlled by those above it in the calling hierarchy. These doubts would seem to be confirmed by the speculative nature of such proposals, offering no validation for the claims made.

3.2.1 Software Science

One traditional metric suggested for measuring object-oriented software is Halstead's Software Science (Halstead 1977), which is surprising given the amount of criticism this metric has received and the studies refuting its validity (see chapter 2, section 2.5.2.1). However, (Coppick and

---

[21] LOC could be considered an exception. Many have question its usefulness for structured systems, and doubtless the same arguments can and will be raised regarding its application to OO systems. It remains undeniably popular, however, and no satisfactory replacement has been found.

Cheatham 1992; Tegarden, Sheetz et al. 1992; Lee, Liang et al. 1993) have all proposed the use of Software Science for measuring OO software. (Coppick and Cheatham 1992), for example gloss over the many studies debunking Software Science, citing only one study, which is positive. They mention briefly the lack of agreement regarding complexity, yet continue to use the term liberally throughout the paper without offering a definition for the reader. The justification for applying a traditional metric seems to be in the parallels drawn between object-oriented and structured software. That the complexity of an object (or module) is dependent on the number of operations (functions) it has, and just as structured modules are decomposed into several more cohesive modules, so are objects, using the inheritance mechanism. This suggests a rather hazy or certainly limited understanding of OO. Operations or responsibilities can be shared amongst classes that are not related via inheritance. Inheritance does not necessarily indicate that there is some shared responsibility for carrying out some function - a class can inherit data or methods and use them as appropriate to carry out a completely different task. Inheritance is a mechanism for reuse, avoiding the repetition of code, not a mechanism for decomposition in the conventional sense. Additionally the authors state that OO design is data centred[22], not function centred, but do not consider data complexity, preferring to attempt to apply a prediction system which was meant for traditional functional design. Software Science is applied to a small LISP Flavors graphics editor demonstration program, using a tool (presumably developed by the authors, which collects a number of undefined measures and uses them as inputs to Software Science, producing

---

[22] This statement is not true for all methods. It is true of, for example Shlaer and Mellor (Shlaer and Mellor 1988; Shlaer and Mellor 1992), who are evidently influenced by traditional data analysis/design, but not of the CRC approach (Wirfs-Brock, Wilkerson et al. 1990) which can be considered a responsibility based approach, which is quite unlike any traditional methods.

"reasonable" outputs. The estimates produced are not compared with actual totals, thus this claim cannot be substantiated.

Coppick et al's work can be criticised on many points. The "results" are no more than the product of Halstead's prediction system, yet claims regarding the applicability of this metric to OO are made without any attempt to compare the estimates with actual figures, or even to get a subjective assessment from experts (developers) of how "reasonable" the figures are. The authors have taken a (discredited) metric and applied it without making allowances for the difference between the object-oriented and structured paradigms. Software Science is not universal model, applicable to all modes of development. It was based upon specific inputs and subsequently shown not to be a useful indicator or effort, size or the many other attributes it was used to predict. The authors have not considered the nature of object-oriented systems that they work by passing messages between objects to initiate the operations, which fulfil some task. They have considered only internal complexity, ignoring completely the communications between objects, which is where much of the complexity in an object-oriented system lies. Thus the model for measuring or predicting attributes of an object-oriented system needs to incorporate some measurement representation of the mechanisms specific to object-orientation, since these are what makes a system object-oriented.

Tegarden, Sheetz and Monarchi (Tegarden, Sheetz et al. 1992), give a number of reasons why they consider traditional metrics are applicable to OO software: that they are unaware of any empirical evidence rejecting the contention that they are applicable; that they already exist and are understood by researchers and practitioners; and that there is supporting empirical evidence regarding their use with structured systems. These statements can be countered easily. First, lack of supporting empirical

evidence refuting the usefulness of traditional metrics for OO systems (particularly when due to the fact that such studies have not been carried out!) does not automatically mean that they are applicable. Second, the authors ignore that fact that the weight of empirical evidence regarding Software Science (and Cyclomatic Complexity) indicates that it is not a useful metric for structured systems, and thus unlikely to be useful when applied to object-oriented systems. The fact that traditional metrics exist and are understood does not mean that they are valid or useful[23]. Again only the internal complexity (referred to as procedural complexity) is considered, ignoring object communication. The authors also seem to be confusing two issues, the validity of traditional metrics as indicators (of complexity[24] — see chapter 2 for a discussion on the problems associated with complexity metrics) and the use of metrics to assess the relative complexity of four different implementations of the same problem. The implementations use inheritance and polymorphism, either individually, together, or not at all (where inheritance is not used, operations (methods) and operands (variables) are duplicated). The authors take some simple measures and calculate the Software Science volume metric. The volume for each implementation is compared. They note that volume is highest where neither inheritance nor polymorphism is used and lowest where both mechanisms are used. They claim this supports their contention that polymorphism and inheritance both reduce complexity. However, what they have obtained is an indirect measure of size, since that is all that volume is. It would be easier to count LOC.

Given the problems with defining complexity, the empirical evidence against Software Science as a valid predictor of anything useful, and the

---

[23] Neither can anyone be certain that many traditional metrics are "understood", that is, that there is any consensus as to how they are applied, for what reason, etc., given the ambiguities in definition, lack of validation, etc. See chapter 2.

lack of compelling evidence of size as a complexity measure, the authors' claims for the usefulness of Software Science or for the beneficial effects of inheritance and polymorphism cannot be upheld. Their so called evidence is meaningless, since Software Science is invalid. They have merely repeated the mistakes of early metrics researchers - of attempting to capture an indefinable attribute, of not really understanding what they are capturing and of having no actual model (they are attempting to fit another prediction system developed for a different situation).

Another paper suggesting the use of metrics based on Software Science is that of (Lee, Liang et al. 1993), although at least they do consider object communication (coupling between objects) and suggest the additional use of information flow metrics. However, as with the other papers advocating the use of Software Science for object-oriented systems, the authors ignore, or are unaware, that the majority of empirical evidence is against Software Science. Misleadingly, the authors claim that "Empirical tests have shown Software Science (is) highly correlated with the number of bugs in a program, programming time, and the quality of a program", but give no references or supporting evidence.

The authors use the Software Science length metric to estimate the complexity of a number of entities, starting with methods, despite admitting that methods tend to be small (the reason given for rejecting Cyclomatic Complexity to measure method complexity). Two definitions are given for class complexity. First that class complexity is equal to the sum of the complexity of its methods (including inherited methods)[25].The second treats the class as an entity and calculates its complexity as the product of the length of the class (sum of length of the

---

[24] Which would then be used in order to try to predict a number of attributes.

methods) and the coupling of the class (the interactions of the methods defined within the class). Hierarchies are variously defined. First as entities (where complexity is the product of the length of the hierarchy and the coupling within the hierarchy). Second as a collection of methods (complexity being the sum of the complexity of all methods contained within it). Third as a collection of classes (complexity equals the sum of class complexities within the hierarchy). The final entity is a program, which is defined as the sum of the complexity of the main program plus the complexity of class hierarchies in the system. No empirical evidence is given in support of the proposed metrics. Weyuker's axioms (Weyuker 1988) are used to give a formal validation (see 2.3.1 for a critique of Weyuker's axioms). The authors do not suggest any guidelines for use of the metrics. As with early metrics researchers, they are seeking to calculate an attribute which cannot be defined and has no clear purpose.

3.2.2 Cyclomatic Complexity

(Tegarden, Sheetz et al. 1992) consider cyclomatic complexity to be a suitable indicator of the complexity of object-oriented systems. They suggest that low cyclomatic complexity indicates either a system that is not complex, from the point of view of the metric, or that "decisions normally measured in a structured module are deferred through message passing to other objects", indicating that few methods would have a high cyclomatic complexity. Firstly one cannot say something is "not complex from the view of this metric". A metric either accurately captures the intended attribute or it does not. The second explanation for low Cyclomatic Complexity indicates that the metric (even assuming it

---

[25] This seems a very simplistic view of complexity. An object, or indeed a module, is usually taken to mean more than the sum of its constituent methods or functionality, though the implications for complexity cannot be quantified.

was valid) is not suitable for object-oriented software because the structure is different and thus the attribute cannot be captured in the same way. (Coppick and Cheatham 1992) also propose the use of Cyclomatic Complexity. A complexity limit of 100 per object is suggested (based on a complexity limit of 10, multiplied by a maximum of 10 methods) as "intuitively reasonable", but offer no other explanation. This limit indicates a limit of 90 decisions per object, which seems an arbitrary figure. The metric was calculated for the same small package as the authors' study of Software Science (see section 3.2.1). As discussed previously in chapter 2, we cannot rely on a shared understanding of a concept. Authors need to explicitly define what attribute is being captured, how and for what purpose, in order to allow consistency and validation. Again, McCabe's cyclomatic complexity has been largely discredited as a traditional metric[26], thus cannot be said to apply for object-oriented systems, since we cannot be certain of what it is trying to achieve, or to test its effectiveness.

3.2.3 Information Flow

(Lee, Liang et al. 1993) consider complexity caused by communication between objects, and adapt Henry and Kafura's Information Flow metric to measure coupling between methods, which are summed to find the class complexity, hierarchy complexity and program complexity. Again Weyuker's axioms are employed (Weyuker 1988), by using additive operators instead of the multiplicative operators defined in the original. Weyuker's axioms been criticised by (Shepperd and Ince 1993) and (Fenton and Pfleeger 1996), amongst others (see chapter 2, section 2.3.1. The application of Weyuker's axioms does not demonstrate validity, and

---

[26] It could be said to have some merit when applied to the issue of test coverage, see section 2.1.1

the usefulness of the metrics cannot be assessed without empirical evidence.

3.2.4 Function Points

There have recently been attempts to apply function points to object-oriented developments, with one such attempt being to relate "use cases", a description of a business function to be implemented in an OO system (Armour, Catherwood et al. 1996) to functions points as a predictor of size. The authors compared the function point count of the use cases with the function point size of the implementation (in Smalltalk), and identified an average 433% growth in the four projects studied. However, the criticisms of function points as a predictor of size still stand (see section 2.5.2.4). It must also be noted that they have merely identified an approximately fourfold increase in the number of function points captured from the projects — no more meaning can be attached to it than that. This study has been extended (Catherwood, Sood et al. 1997), where object data (number of objects, number of methods) was captured in order to study the relationship between objects and function points. Information from three more projects was added to that presented in (Armour, Catherwood et al. 1996), and the average growth recalculated as 381%.

The results of the function point/object data study are as in table 3.1 below (derived from (Catherwood, Sood et al. 1997)). Although explicitly defined, it seems that "object" could be replaced by "class".

	# objects per function point based on Use Cases	*# objects per function point based on Impl. System*	*# methods per function point based on Use Cases*	*#methods per function point based on Impl. System*
mean	1.16	0.38	18.18	4.96
standard deviation	0.05	0.12	8.44	0.9

Table 3.1: Some summary statistics for functions point/object data study (derived from (Catherwood, Sood et al. 1997))

The authors found that by removing a particular project (with a different implementation language, PowerBuilder), the standard deviation could be reduced. From this it can be deduced that the prediction system is implementation dependent. Additionally, although a "tight" correlation is reported, neither the result, significance, nor the type of correlation used is stated. A repeat analysis of the figures supplied reveals that the result varies with the correlation test used. It can also be seen from the new analysis that the scatterplots of function points against number of objects show that one outlier has a great effect on the regression line. Figure 3.1 shows the regression plot and correlations for all datapoints, and 3.2 repeats these with the outlier removed

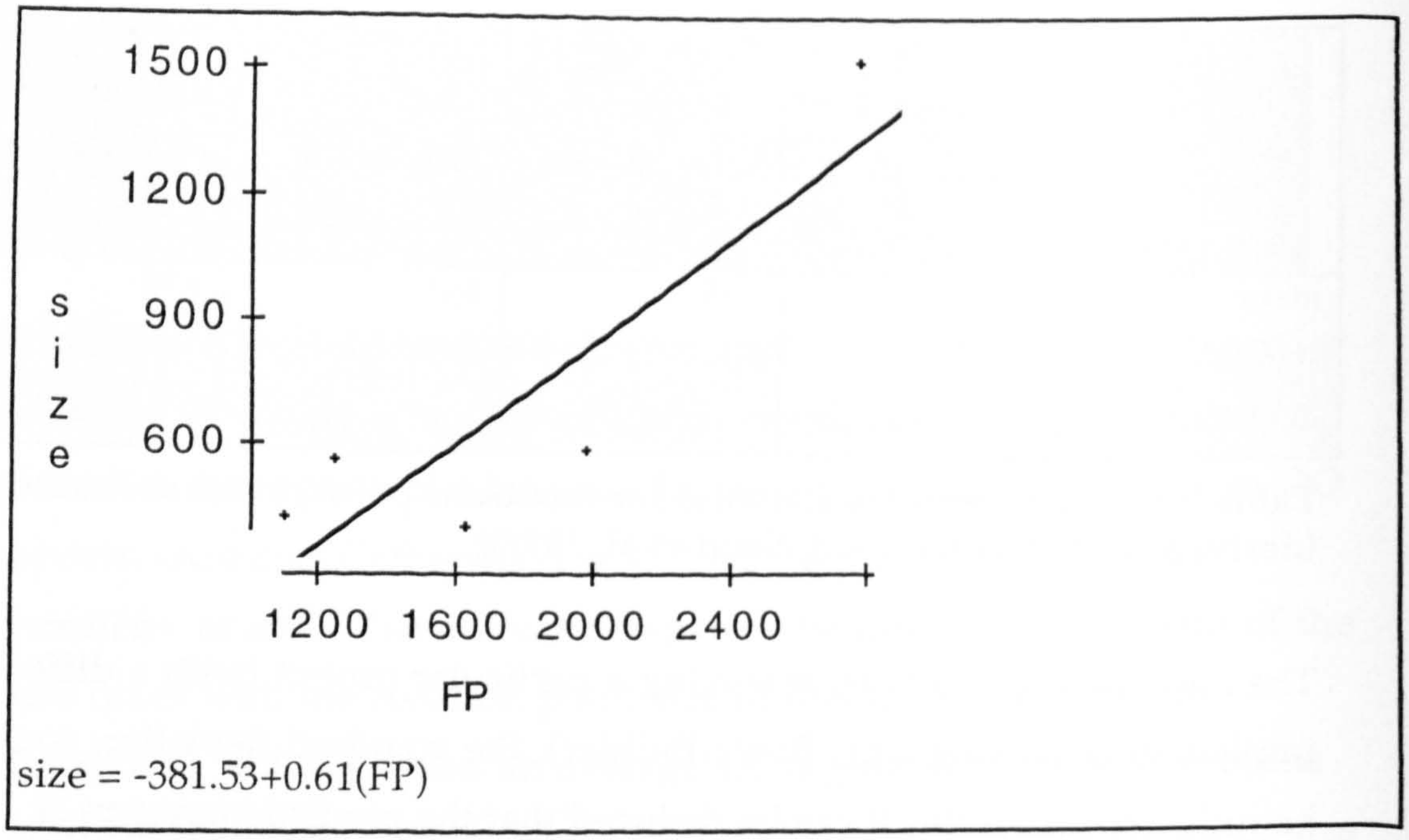

Figure 3.1: Regression plot of Function Points against Size (measured as no. of objects)

Pearson Product Moment Correlation Coefficient 0.883.

Spearman Rank Correlation Coefficient 0.700.

The Pearson correlation is significant at the 5% confidence level, but the Spearman is not significant at 10%.

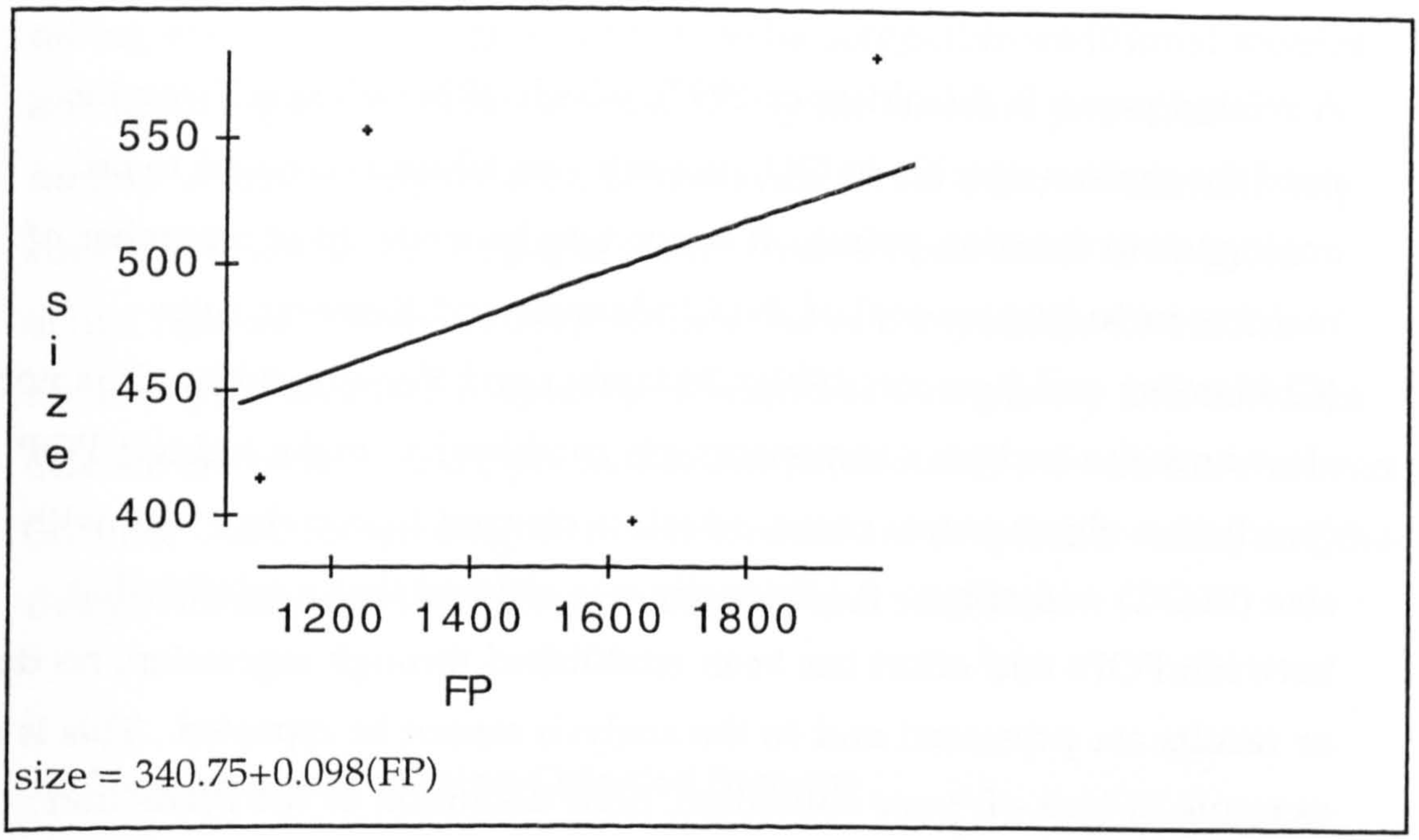

Figure 3.2: Regression plot of Function Points against Size (measured as no. of objects) with outlier removed

Pearson Product Moment Correlation 0.412.

Spearman Rank Correlation Coefficient 0.400.

Neither are significant (Pearson at 5% and Spearman at 10% confidence levels).

However, five datapoints are not really enough to draw conclusions from. Further, the datasets (in particular the number of projects included) vary according to the calculations — correlations between function point counts and object/method counts are based on five projects whereas function point growth is based on seven. The reason for this is not explained, thus in addition to the criticisms above, there is inconsistency in the dataset according to the tests performed. It is unfortunate, given the general shortage of completed object-oriented projects from which such data can be extracted, that the authors are focusing on one particular metric, especially since it is aimed at traditional systems development and is of debatable validity.

A related paper is (Minkiewicz 1997), which, although specifying the need for metrics specific to OO, presents one which is claimed to be analogous to function points. It appears to be made up of a number of metrics, including several of the Chidamber and Kemerer suite (Chidamber and Kemerer 1991; Chidamber and Kemerer 1994). It is not clear how the various components are combined to make a single POP (predictive object point) count, which is claimed to correlate well with size (SLOC) and effort. Additionally it is claimed that a relationship between POPs and effort has been established through regression, no data or results are presented and so the analysis cannot be repeated. This is an example of lack of/poor validation, poor definition of the prediction system and of how the measures are taken and the lack of a clear purpose. Although linear regression is used to "establish" a relationship between POPs and effort, we have no idea of how POPs themselves are derived or the reasoning behind the process.

3.2.5 Summary on the Application of Traditional Metrics

It is fairly obvious both from the limited number of papers suggesting specific traditional metrics as applicable to object-oriented systems, and from the speculative nature of the work, that this is not a branch of OO metrics research that has proved fruitful. There may well be some traditional measures that could be used for OO software, but there does not appear to be any published work that demonstrates this (or not). The papers cited above are not credible for the following reasons: they advocate the use of traditional complexity metrics, yet ignore the empirical evidence that has led to these metrics being discredited. They are not really looking at them afresh by examining the model and seeing if it could be applied more successfully to OO software, but are merely

taking what they (wrongly) perceive to be acceptable traditional metrics and applying them to OO software. Only the function point based metrics actually compare their results with actual data, to ascertain any correlation, and only one paper (Catherwood, Sood et al. 1997) gives actual figures. All are looking at the issue of complexity, which has obvious problems (see chapter 2). Cyclomatic Complexity and Software Science are code metrics, yet traditional (structured) metrics have moved on since then, recognising that code metrics are of limited value, and that design metrics provide earlier predictions and feedback.

3.3 New Metrics for Object-Oriented Systems

Most researchers behind the contention that object technology needs metrics developed specifically to take into account the features unique to the paradigm. A number have been proposed, but are still largely speculative or with little support in terms of validation or an assessment of their usefulness. This is partly due to the precedent set by traditional metrics, that it is enough to speculate, particularly if the metric can be accompanied by mathematics and be described as "intuitively reasonable", or at best be "validated" according to Weyuker's axioms. However, some of the proposed metrics have had some attempts at empirical validation. This is limited, however, by the lack of available data, due to the difficulties in obtaining "real", mature [27] object-oriented systems to study. This section considers what can be termed the "state of the art" of object-oriented metrics. The amount of consideration given to each will depend upon the material available and the interest they have generated.

---

[27] Mature in the sense of being tested, delivered and maintained, since these phases are necessary for those studies considering defects, maintainability and so on.

3.3.1 Chidamber and Kemerer's Metric Suite

Chidamber and Kemerer were the first to publish metrics for OO design (Chidamber and Kemerer 1991). They have become the de facto standard, and have been incorporated into code analysis tools (e.g. Logiscope). One reason for their popularity is undoubtedly because they were the first and consequently the most studied metrics (as was the case with Software Science). Also they were formally defined (using sets), and it was made clear in the original publication that work was ongoing and empirical validation would be forthcoming (published in (Chidamber and Kemerer 1994)). This is not to say that the metrics are without flaws. Ambiguity in the definition of counting rules was highlighted by (Churcher and Shepperd 1995). Research contributing to this thesis found difficulties applying most of the metrics at design time (Cartwright and Shepperd 1997b). They are claimed to be design *complexity* metrics, and thus have the problems of definition and purpose associated with traditional complexity metrics (see chapter 2).

The Chidamber and Kemerer (CK) metrics will now be examined in more detail. This will include independent empirical validations as well as that published by Chidamber and Kemerer. First a description of the metrics suite is given. The suite is then discussed in the light of the "lessons learned" from the application of measurement theory to software metrics, as described in section 2.4.3. The discussion is based upon examination of the metrics, experience in using them and the findings of other authors.

3.3.1.1 The Metrics Suite

Chidamber and Kemerer published an early paper on object-oriented software metrics (Chidamber and Kemerer 1991), proposing a suite of six

design metrics, with the intention of capturing the different architectural features of object-oriented systems in order to assess design quality and to predict "managerial metrics" such as effort. Further work in the form of empirical studies and some refinements to the metrics as first proposed followed (Chidamber and Kemerer 1994; Chidamber, Darcy et al. 1997). The metrics are based on classes within an OO system. The terms class and object are used interchangeably.[28]

WMC (Weighted Methods Per Class). This metric is intended to measure the complexity of a class, assuming that a class with more methods than another is also likely to be more complex. Weightings are not fully specified, thus the general approach is to assume all methods are equally complex and thus calculate WMC as the count of the number of methods in a class. The alternative is to decide upon some other method for calculating the internal complexity of a class. Comments in (Chidamber and Kemerer 1995) indicate that only methods specified in a class are included, that is, any methods inherited from a parent are excluded.

DIT (Depth of Inheritance Tree). It is assumed that a class deeper in the inheritance hierarchy is more complex because of the number of definitions, methods, etc., inherited from ancestors. It is defined as the maximum depth of the inheritance graph of each class, thus allowing for multiple inheritance. The base class is DIT=zero, its children DIT=1 and so on.

NOC (Number Of Children). This metric is calculated as the number of direct descendants for a class. It is assumed that a class with more children can be regarded as more complex since it directly affects more classes.

---

[28] C&K intend these as design metrics. Most methods use the term class during design (an object being an instantiation of a class).

CBO (Coupling Between Objects). This represents the number of other classes to which a class is coupled (here object = class since at design time we don't know anything about actual instantiations, i.e. objects). The definition of coupling between classes is that one class uses the methods/variables of another. In this sense, objects are instantiations of classes.

RFC (Response For A Class). This is a count of the number of methods that could potentially be executed in response to a message received. This assumes that the higher the count, the more complex the class.

LCOM (Lack Of Cohesion Of Methods). This indicates the number of pairs of methods without shared instance variables minus the number of pairs which do have shared instance variables. When the result is negative, the metric is set to 0.

3.3.1.2 Goal

Chidamber and Kemerer suggest that complexity can be used in cost estimation, evaluating productivity, estimating maintenance requirements and improving software quality (Chidamber and Kemerer 1991). How these measures can be used in this way is not specified, that is, no prediction systems are defined Or validated. Additionally, whilst they are clearly complexity metrics, it is implied that size is also assessed, which is not clear from the nature of the measures defined. However, in a later technical report (Chidamber, Darcy et al. 1997), it is suggested that the metrics are "measurements of design complexity" and thus may be used to assess/predict variations in productivity, rework effort and design effort. Stepwise regression is used to produce an equation for each

of these dependant variables in terms of four of the CK metrics (excluding DIT and NOC). The measures have been developed with some consideration of measurement theory[29], in that they are formally defined (using sets) and based upon a model of OO design. It could be argued that the attribute of interest is not clearly enough defined and that there is some ambiguity in the definition of counting rules (Churcher and Shepperd 1995). The concept of objects employed in this research is based upon Bunge's ontology (Bunge 1977), since this deals with the definition of representation of the (real) world. This is consistent with the approach of object-oriented design, which aims to model the real world in a more natural way, (i.e. independent of implementation), than the functional approach, where artificial separations are made between data and processes. Graham (Graham 1995), questions the suitability of Bunge's ontology as a basis for object-oriented metrics, since it implies an object is defined by its properties, which is not necessarily the case in an object-oriented system.

The metrics are presented as design metrics, thus implicitly implementation independent. Henderson-Sellers suggests that since (Henderson-Sellers 1996) the WMC metric does not consider the possibility of method type, there is potentially a drawback in applying the method to a system where the intended implementation language is C++, since this language does use different types of method[30]. We cannot be sure that method type is or is not an issue, and Henderson-Sellers does not demonstrate that there is a problem with ignoring method type. It is

---

[29] The authors certainly consider measurement theory, but doubts have been raised as to how vigorously it has been applied (Hitz and Montazeri 1996).

[30] It could be argued that the weighting applied could reflect the different method types. However, it is still clear that we do not really know whether weightings are useful or whether the metric should be a simple count of methods, and if so what use is a count of methods? Obviously more empirical work is needed, but until counting rules, implementation issues and the actual purpose(s) are agreed on, studies will not necessarily apply the metrics consistently.

not clear from Chidamber and Kemerer's work whether this is due to lack of consideration of method type, or whether they considered it a non-issue..

Further criticism of the metrics suite with respect to its lack of implementation independence is offered in (Henderson-Sellers 1996), where DIT is considered to be more suited to a Smalltalk application, with a single base class, than C++ where a single base class is not required. However, there is no evidence that this is necessarily an issue where DIT is interpreted at a class level (each class having its own value of DIT). If DIT is interpreted as a system metric, with a single value for the whole system, this issue would have an impact. However, such a definition, although not precluded, is not what is intended by Chidamber and Kemerer, who present the suite as class metrics, and present empirical evidence which makes it clear that the metric is calculated at class level (Chidamber and Kemerer 1994).

A further criticism can be levelled at the lack of clear purpose and guidance for use of the CBO metric. It is suggested by Chidamber and Kemerer that a highly coupled system is not desirable, as is already generally agreed within the software engineering community. However, some degree of coupling is not only unavoidable, but is necessary. Henderson-Sellers *et al.* (Henderson-Sellers 1996) point out that inheritance based coupling is an unavoidable consequence of using inheritance, and consider that it should be counted separately from other coupling (see section 3.3.1.4).

Suggestions are necessary as to what values are acceptable, or how to ascertain such bounds. This is a criticism which is applicable to all of the metrics in the suite, and is not confined to Chidamber and Kemerer. Certainly the lack of guidance given in this case may well be a

consequence of the general lack of guidance and sense of purpose among complexity metrics in general. The previous chapter criticised the tendency to adopt complexity as a goal, since complexity in itself told us little. What was, and is still of interest, is the relationship between complexity and other more useful attributes, and thus information is needed on how the assessed complexity would affect these other attributes, for example how could we use CBO to assess the relative quality of designs.

3.3.1.3 Validation

Validation in the original paper (Chidamber and Kemerer 1991) is based on Weyuker's axioms (Weyuker 1988) (see chapter 2) the appropriateness of which have been questioned by (Kitchenham, Pfleeger et al. 1995) amongst others. However, a subsequent paper (Chidamber and Kemerer 1994) provides preliminary results of an empirical investigation. The authors suggest that the metrics be used to aid:

(i) reuse

(ii) identifying design flaws (an example being excessive declaration of subclasses)

(iii) in allocation of testing resources (for classes with high values for the CBO and RFC metrics)

(iv) gain an insight into trade-offs made between maximising reuse (by inheritance) and ease of understanding and testing (designing a shallower inheritance hierarchy)[31].

Interviews with developers at the two data sites involved are used to give an (informed) subjective evaluation of the results. Although the implied use of these metrics is as predictors (i.e. inputs into prediction

---

[31] Chidamber and Kemerer found that inheritance hierarchies tend to be shallow

systems), it should be noted that these are complexity measures, i.e. assessment metrics rather than prediction systems. This is because the empirical study does not compare predicted against actual values such as defect data, testing effort etc. Thus the implication that they are useful predictors of defects, testing requirements and so forth has not been empirically assessed.

In the later technical report (Chidamber, Darcy et al. 1997) the authors find that WMC, CBO and RFC to be highly correlated. In the stepwise regression analysis, CBO and LCOM are statistically significant predictors for design effort, rework effort and productivity.

In (Hitz and Montazeri 1996), some problems with the metrics are identified. The authors make it clear that they are considering the metrics as measures rather than as predictors. The authors suggest that improvements could be made to the metrics by a more rigorous application of measurement theory principles. They summarise the stages necessary for developing and validating metrics and suggest that Chidamber and Kemerer do not list satisfactory empirical relation systems for the metrics, concentrating instead on the effects of the metrics on other attributes. Hitz *et al.* emphasise the need for a "sufficient" set of empirical relations to be defined in order to account for the possible situations which may occur. A metric must successfully map the empirical relation system to a numerical system — if the empirical relation system is incomplete or poorly defined then one cannot be certain that the representation condition holds and thus cannot satisfactorily demonstrate that the metric is valid.

(particularly in C++ applications).

The Chidamber and Kemerer metrics have been used in independent empirical studies to assess maintainability (Li and Henry 1993a; Li and Henry 1993b), evaluation of object-oriented analysis and design methods (Sharble and Cohen 1993), probability of faults in a class (Basili, Briand et al. 1995), size (de Champeaux 1997), quality (Binkley and Schach 1996) and number of defects (Cartwright and Shepperd 1997b).

Li and Henry (Li and Henry 1993a), consider five of Chidamber and Kemerer's metrics (rejecting CBO), as well as five of their own, in an empirical study based on two commercial systems, developed in Classic Ada. The additional metrics will be defined in section 3.3.2.1. The authors collect maintenance effort data from the systems over three years. Unlike Basili *et al.* (Basili, Briand et al. 1995), Li and Henry define the WMC metric as the "summation of McCabe's cyclomatic complexity of all local measures", i.e. $v(G)$ for the class, which can be more simply expressed as $m+n$, where m is the number of decisions in the class and n is the number of methods. Why two apparently different commodities should be added together is unclear and the validity of doing so, as well as the usefulness of this procedure in an object-oriented system, where typically methods are small (Wilde, Matthews et al. 1993), is questionable[32]. Moreover, the "object-orientedness" of Ada, might be better described as object based, since although it shares certain common features with more obviously object-oriented languages, it lacks others (Wegner 1990). Although the variant used in (Li and Henry 1993a), Classic-Ada, is described as an object-oriented language, where object-oriented constructs such as class and superclass have been added to the standard Ada constructs.

---

[32] Where methods are typically small and preferably fairly atomic, then the number of decisions is likely to equal one, thus this measure is likely to be the same as counting the number of methods.

Regression analysis is used to assess the suitability of the metrics as predictors of maintenance effort by using them as independent variables to predict the dependent variable, maintenance effort, and compare this with the results of using regression analysis using size measures to predict maintenance effort. "The number of lines changed per class" is the definition of maintenance effort. No information is given on time taken to implement changes, so maintenance is simply a count of modified lines of code. The authors conclude that the metrics are good predictors of maintenance effort. However, the various measures are not independently tested, all of them being included in a multiple regression equation, thus the validity of the Chidamber and Kemerer metrics alone is not satisfactorily established in this study. See 3.3.2 for a criticism of the empirical study.

Basili *et al.* (Basili, Briand et al. 1995) use a modified version of the metric suite to suit C++, since indications are that it is not language independent and does not reflect many mechanisms peculiar to C++ (Chidamber and Kemerer 1994; Chidamber and Kemerer 1995; Churcher and Shepperd 1995). Their study is based upon an experiment with student programmers, with some (unspecified) experience with C++ but not necessarily with OO methods, or with the libraries provided. A C++ programmer familiar with the libraries was available for consultation. Code data and defects discovered and fixed were collected. The Chidamber and Kemerer metrics were extracted from the code delivered at the end of implementation, error data during testing and fix data during the repair stage. The amount of modification made to a class was categorised as none, small or large, classes being allocated according to the developer's estimate of the percentage of code modified. The authors found all but LCOM to be "adequate" predictors of fault prone classes (in terms of predicting whether a class would contain one or more faults or none), and that they performed better than traditional code metrics. It

should be noted that although Basili *et al.* state that the metrics are available earlier in the lifecycle than traditional code metrics, they extracted them from the code, rather than design documentation. Thus they fail to demonstrate that they can be considered design metrics, a claim challenged by the empirical study carried out for this thesis (see chapter 5).

Sharble and Cohen (Sharble and Cohen 1993), use the Chidamber and Kemerer metrics suite to compare two approaches to object-oriented analysis and design, data-driven and responsibility-driven. The CK metrics are supplemented by three others. The metrics are not validated, but are applied to both designs and the results compared to determine which approach lead to the least complex design, the conclusion being that the responsibility-driven design is less complex than the data-driven design.

Cartwright (Cartwright and Shepperd 1997b) found a significant +ve correlation between DIT and error density. However, the indications were that involvement in an inheritance structure at whatever level was more relevant to error proneness than the actual depth. NOC was not found to have strong correlations with defects or size. The other metrics could not be analysed since they proved impossible to collect from the design documentation. A more detailed discussion of this study can be found in chapter 5.

In (de Champeaux 1997), Chidamber and Kemerer's metrics are included in a large number of metrics applied to a case study developed by students. He notes a "disturbing" significant correlation between his adaptation of WMC and CBO, and between WMC and RFC. It is suggested that RFC and CBO are both measuring size rather than different aspects of quality.

Four of the metrics, CBO, RFC, DIT and NOC are assessed in (Binkley and Schach 1996). This study uses expert opinion to compare the quality of alternative design solutions to a particular problem and rank them. The authors then apply 16 metrics; to find which confirmed the experts' consensus based opinion. However, the results are unclear since metrics are grouped according to type and the results (success rate at match experts' opinion) is given per category. They find that simple coupling metrics (which includes CBO) have a success rate of just 17%, inheritance based metrics (including DIT and NOC) 28% and RFC (given its own category), 33%. It is noted that fourteen of the sixteen metrics tested, including the Chidamber and Kemerer metrics, score lower than a random. The criticism levelled by the authors is that the metrics are defined at too high a level of abstraction to measure design details. Examined more closely the authors seem to mean that the metrics are too crude and do not distinguish between different types of coupling and inheritance. The authors conclude that the accuracy of coupling measures is the determining factor in their success as predictors of quality and that more abstract measures cannot give an accurate measure of complexity and thus will not give an accurate prediction. Although this argument is plausible, the study does not demonstrate this satisfactorily -- the criteria for judging designs is not given, nor the scores of the individual metrics. Neither is it explained how they were used as predictors. Calculating, CBO, for example, does not predict the quality of a design. In the absence of guidance on thresholds etc., from Chidamber and Kemerer it must be assumed that the authors supplied their own, but what they are and how they were arrived at is not published.

Wilkie and Hylands (Wilkie and Hylands 1998) present the results of applying the metrics suite to a 25 KLOC 114 class "industrial grade" C++

system. They apply two versions of the WMC metric[33], WMC_{ss} which uses Software Science (Halstead 1977; Halstead 1979) and WMC_{cc}, which uses Cyclomatic Complexity (McCabe 1976; McCabe and Butler 1989) to measure method complexity (see sections 3.2.1 and 3.2.2 for criticisms of the application of these traditional and controversial metrics to object-oriented software). The authors suggest that WMC_{ss} and DIT alone "provide a significant contribution to the fault predicting capabilities of the C&K suite". This is the result of regression analysis involving the full set of classes and a subset of the classes, in which all of the classes included had some associated fault fixing effort. The adjusted R^2 values for both sets are low (0.3 and 0.44 respectively) which raises some doubts as to their usefulness in practice. The same study suggests a relationship between RFC and product enhancement but does not give supporting figures. The analysis is repeated to take account of the effects of inheritance (a class which inherits will have the complexities of its ancestors added to its own complexity). The results differ, so that DIT and WMC_{ss} are no longer significant indicators of fault fixing effort and that CBO emerges as being more significant (although once more the R^2 is very low (0.09)).

3.3.1.4 Definition

The second paper (Chidamber and Kemerer 1994) makes some changes to the original definitions. Unfortunately there is still some ambiguity as suggested in (Churcher and Shepperd 1995), where an example is given showing the possible differing interpretations and results for the WMC metric. The particular problem illustrated therein was subsequently clarified (Chidamber and Kemerer 1995), but there is still the possibility of the continuing use of the earlier definitions.

---

[33] Ignoring inherited methods, i.e. using methods declared only in that class

Further, the drawbacks with the original metrics have not all been cleared up, an example being LCOM, where any negative values are set to 0. This seems to make the metric insensitive for highly cohesive classes and obviously cannot discriminate between lower values since all negatives are set to 0. In the absence of explicit guidelines on how to use the result (how do we determine if a score is acceptable or not?), this metric is limited in its use as, for example, an indicator of quality, since all classes reaching a certain level of cohesion score the same value (i.e. 0). Henderson-Sellers, Constantine and Graham consider this in (Henderson-Sellers, Constantine et al. 1996), and give an example where classes have the same LCOM score but, upon examination of the designs, appear to have different levels of cohesion. They point out that although high LCOM values can indicate low cohesion the converse is not necessarily true. From the examples presented, the authors conclude that it is possible for a value of LCOM = 0 to indicate a highly cohesive class, a not very cohesive class as well as a class with no cohesion. Henderson-Sellers *et al.* suggest a new definition for LCOM to overcome this problem. Although the cohesiveness of a class, or indeed any unit, can be somewhat subjective, since it is easy enough to distinguish between a highly cohesive class and a non cohesive class, but harder to rank classes which seem similar, the argument is persuasive. It shows how very differently structured and sized classes can have the same score. Common sense tells us that it would be possible, although possibly somewhat contrived, to construct a class in such a way that a low score was obtained, but the class would not necessarily be cohesive. For example, methods could share variables without necessarily making any sensible use out of them, merely to improve the LCOM score. This would be consistent with the argument of Henderson-Sellers *et al.*, that a low score did not necessarily indicate high cohesion. Thus the definition

of this metric does not fulfil its stated goal of assessing cohesion, which means that it fails to satisfy the representation condition.

Henderson-Sellers *et al.* also note the change in definition of CBO, for which the 1991 definition implied only bi-directional coupling was counted. The authors feel that Chidamber and Kemerer probably did mean this, but it was a possible ambiguity arising from the poor definition. In 1994, the definition was changed making it clear that one-way coupling was not precluded. However, in the redefinition, inheritance based coupling, i.e. between parent and child classes, is counted along with non inheritance coupling, where classes are collaborating by message passing to fulfil a task, whereas the 1991 definition distinguished between them. Although there is no evidence to suggest that the type is important, we cannot know that without analysis, and if the types are not distinguished, this cannot be done. Li and Henry overcame this problem by replacing CBO with two coupling metrics one of which counts the types of coupling (Li and Henry 1993a).

3.3.1.5 Summary

Despite the relative maturity of the Chidamber and Kemerer metrics, they still lack sufficient independent empirical validation to judge the usefulness of the individual metrics. This problem is by no means unique to these metrics and is largely due to the immaturity of the paradigm; having relatively few suitable[34] OO systems to study; the variation in design methods (and indeed lack of any design methods in many of the mature systems); the variation in languages used, and an apparent reluctance by developers to supply data for analysis.

---

[34] That is, real-world, reasonably large, mature, stable systems.

A further problem with these metrics is, as mentioned above, despite being described as design metrics, it is not always possible to collect them from design documents. Indeed, Basili *et al.* (Basili, Briand et al. 1995), collect the metrics from code, as apparently do Li and Henry (Li and Henry 1993a). This is an obvious drawback if the metrics are intended to give feedback into the design process. Some of the problems in definition and validity (i.e. as defined the metrics do not satisfy the stated goal) may occur from a limited understanding of object-orientation as practice. A connected issue is that Smalltalk, upon which much academic work, including Chidamber and Kemerer's, is based, is a "pure" object-oriented language, whereas C++, which is much more popular in industry, is regarded as a hybrid. As a hybrid, C++ employs some mechanisms which are peculiar to itself and are not standard OO. This makes it difficult to suggest generally applicable metrics for object-oriented systems, since the implementation of the system could well affect its complexity and also the way it is designed. Many very different analysis and design methods are available, which will also make it difficult to specify design metrics since the same models are not necessarily employed in each. Any metric which is to be considered generally applicable must therefore confine itself to measuring aspects which are common across methods and languages, such as the use of classes and inheritance message passing. However, the necessity of such an abstract view may well make such a metric of less use than one that is environment specific.

3.3.2. Other Object-Oriented Metrics

A number of other metrics, relating to both size and complexity, have been proposed. So far the proposals are largely speculative, few having convincing empirical evidence to back them up. This section will

consider a number of such metrics, some will be discussed in some detail, and others summarised in a table, for reasons of brevity. The metrics chosen for more detailed discussion are included on the basis of the amount of published material available, the detail entered into.

3.3.2.1 Li and Henry

Li and Henry (Li and Henry 1993a) supplement Chidamber and Kemerer's metrics suite with a further five metrics, three of which they conclude are accurate predictors of maintenance effort. The authors reject Chidamber and Kemerer's CBO metric, replacing it with two other coupling measures, MPC, message-passing coupling and DAC, coupling through abstract data types (ADT's).

MPC is defined as the number of send statements defined in a class, and DAC as the number of ADT's defined in a class. In the absence of a precise textual or a formal definition, this metric is open to interpretation. The actual definition given is "DAC = number of ADTs defined in a class" where a class is defined as an implementation of an ADT. The textual definition given seems to imply that one class implements one ADT, thus giving a value of DAC = 1.

However, reading what seems to be the motivation for defining DAC, the potential for coupling through ADTs, the following definition seems reasonable. The coupling referred to is the coupling between classes where a variable defined in one class is of an ADT type of another. This seems confirmed by the concern of the authors that such a couple means that the class in which the variable (of another ADT type) is defined may access the properties of the other class (i.e. the class implementing the ADT of which the defined variable is a type), and possibly violating

encapsulation if direct access to the private data of the ADT class is not prevented. DAC would benefit greatly from a clearer definition, either formal or textual. As it is, it is hard to determine whether we are capturing what is intended and if the metric collected is a reasonable representation of the indirect attribute of interest (presumably complexity which itself would be used to predict another indirect attribute maintainability).

In addition they collect the number of local methods in a class (NOM), as a complexity metric and two size metrics, SIZE1, the number of semicolons in a class and SIZE2, the number of attributes plus the number of local methods. The reasoning behind the latter, or what it may offer over the more traditional size measure, (SIZE1 is essentially LOC), is not discussed. It is hard to see why adding two counts of different attributes might be of use.

The goal behind the proposed metrics is to predict maintainability. Maintainability or maintenance effort, another term used, is defined by Li and Henry as the number of lines changed per class in its maintenance history and is referred to as change. It is explicitly stated that the metrics are intended as predictors of maintenance effort, but how this is to be done is not demonstrated.

The authors use data from two industrial Ada systems of 39 and 71 classes respectively. The empirical validation carried out can be questioned on a number of issues. One of these is the suitability of Ada as a representative object-oriented language, although as stated in 3.3.1.3, a non-standard variant, Classic-Ada, is used. A further point is the use of multiple regression as opposed to either single-variable regression or stepwise multiple regression. Additionally, the R^2 and adjusted R^2 figures from the "full" multiple regression equation (where size

measures are included) are virtually the same as those from the "refined" multiple regression test (with size measures removed beforehand). From the test these figures are:

	R^2 (sys A/sys B)	adj. R^2 (sys A/sys B)
Full regression model	0.9096/0.8737	0.8773/0.8550
Refined regression model	0.9030/0.8680	0.8771/0.8533

Table 3.2: Regression models using full metrics set and refined metrics set. Dependent variable is change.

Regression is a reasonable technique, but using multiple regression (as opposed to stepwise multiple regression) does not allow insignificant independent variables to be rejected, thus the effect of each variable cannot be independently ascertained. Additionally using 10 or even 8 variables makes for an overcomplicated model with greater potential for collinearity, where it is unclear what effect on the model each variable has. Further including so many variables may lead to overfitting, making the model suitable for that particular dataset but no other. The advantage of using stepwise regression is that we can see the effect as each variable is added to the equation, plus those which are not significant can be rejected. Instead, Li and Henry use VIF (Variation Inflation Factor) to determine which if any variables should be removed from the "full" model. The criterion is that any variable with a VIF higher than 50 should be rejected. However, they do not follow this criterion for all variables, considering "other factors" such as a strong correlation between a size metric and McCabe's complexity metric as justification for retaining a variable with an excessive VIF and rejecting one with a VIF below the rejection threshold.

Additionally, since a cross correlation for all variables is not carried out (or at least reported), it is hard to determine how closely related the

independent variables might be. For example, it is reasonable from the definitions given to assume there might be some relationship between NOM (number of methods) and WMC (weighted methods per class) which might effect the regression equation. The usefulness of each of the individual metrics has not been ascertained, and thus neither has the true accuracy of the prediction system (i.e. using all of the metrics added together). The usefulness of the prediction, even if it is accurate, is limited since the data is collected from source code. The conclusions reached by Li and Henry (Li and Henry 1993a) are over generalised, e.g. "There is a strong relationship between metrics and maintenance effort in object-oriented systems.". Knowing which measurements to collect and how to use them to predict maintenance effort or changes is more useful and interesting.

3.3.2.2. Lorenz and Kidd

In, apparently the first book dedicated to object-oriented metrics (Lorenz and Kidd 1994), Lorenz and Kidd suggest a large number of object-oriented metrics, based on analysis of C++ and Smalltalk projects. These are divided into two categories, project and design metrics. The design metrics consist of 27 measures, divided into 7 categories, looking at methods, classes, inheritance and other "external" measures. Related metrics are also suggested for each of the metrics in the main list. Explanations for the metrics are vague, both in the sense of the indirect attributes they are attempting to capture and in the counting rules necessary to capture them. Also some of the titles of metrics do not match those listed under the various categories. Although it is stated that the metrics are based on "actual project experiences", little in the way of actual statistics or results appears, apart from some bar charts. We are presented with suggested thresholds for each measure, again with little

or no explanation as to why or how the figures are derived. When thresholds are breached, there are "Suggested actions" to be taken. The overall impression is the metrics, thresholds, suggestions and so on are anecdotal, and not based on data analysis.

The authors then make a recommendation of which of the listed metrics should be used (bringing down the total of design metrics to a mere 24), and for what purpose, which can be one of four categories, model quality, class quality, method quality and management. How to use the metrics is not specified, although it is stated that they are "meaningful metrics that will help you foster better designs, develop more reusable code, and prepare better estimates."

A practitioner using the metrics would need to ascertain for themselves how to use them, how useful they are, which thresholds to use in which circumstances and what action to take. Using all of the metrics suggested seems potentially risky in terms of the trade off between cost to collect and any benefits that may be gained from their use. It is unfortunate that the authors have not shown how or why these metrics were derived, since as they are based on actual projects, presumably there is data available to analyse. We are unable to check for collinearity since no data is provided and nor do the authors indicate that this has been considered. No clear goal for the metrics suite is given, the authors instead making statements such as "The metrics should be used to support the desired motivations.".

One interpretation of this statement is that the metrics can be put to any use. Lorenz and Kidd do not distinguish between measurement and prediction systems. Many of the metrics are used to form prediction systems, but without empirical evidence or any clear way of checking predictions (the counting rules and attributes are often unclear, and the

data input into the predictions systems is not presented), they cannot be acceptable. In the absence of actual predictions, it could be assumed that they are to be treated as measures from which an expert can make some inference, but again this is not clear. The metrics suggested are derived from practical experience, with no reference to theoretical underpinnings. Certainly many of the metrics are poorly defined, both in terms of what is being measured and how, and purpose. Further, some of the definitions given could be misleading, for example a "comment line" defined as a physical line of code that contains a comment — it is possible for a physical line to contain both source code and a comment. Such definitions need to be more clearly thought out lest misleading inferences be made.

Although it is not unreasonable to emphasise usefulness (to be demonstrated empirically) rather than validity in the sense of measurement theory, neither approach is followed. Metrics should be either valid (in the sense of measurement theory) or useful, or preferably both. These metrics demonstrate neither property. For example, method complexity is calculated as the total number of complexities (presumably for a class) divided by the total number of methods. The total number of complexities is calculated by counting the number of methods of each of a given type and multiplied by the suggested weighting associated with that type. These are then added to give a total for the class. Adding weighted counts violates scales in the same way Function Points do, by assuming that the weights will ensure that the different categories of method (as defined by (Lorenz and Kidd) are equivalent. Additionally one can question how representative is the data used to derive the weightings. However, the Function Point method is arguably sometimes useful as a predictor of size, whereas there is no empirical evidence to suggest that Lorenz and Kidd's method complexity metric is at all useful.

3.3.2.3. Henderson-Sellers

In his book (Henderson-Sellers 1996), Henderson-Sellers reviews a number of object-oriented metrics. Various perspectives of an object-oriented system are considered and metrics considered suitable for each of these are listed (this is an elaboration on (Henderson-Sellers 1994)). The book offers little in the way of validation of the metrics suggested and is lacking in clarity — definitions and guidance on collecting the metrics are missing, as is the purpose of collecting the measures. Apart from being told they are complexity metrics, the reader is given little information on how to use them. Given the large numbers of measures suggested, this sort of information would be highly desirable before embarking on what could be a costly metrics collection programme. A count of suggested measures gives:

per class — 20;
system level — 22;
reuse — 4.

The book also refers to earlier work in the area of object-oriented metrics by Henderson-Sellers. In (Henderson-Sellers 1991), metrics are suggested for size and reuse, with the emphasis being on providing early estimates. The suggested metrics include "appropriate weights" and are not backed up with formal validation or any empirical evidence.

3.3.5 de Champeaux

After publishing early papers on object-oriented development, and methods (de Champeaux, Anderson et al. 1992; de Champeaux and Faure

1992; de Champeaux, Lea et al. 1992), de Champeaux extended this work to include metrics. The metrics are applied to an object-oriented development by student programmers. These cover both effort metrics (both effort measures and effort estimation) and product or artefact metrics. Well over 20 artefact metrics are defined, including Chidamber and Kemerer's class metrics. The metrics consider classes and above (i.e. subsystem, use cases, class relationships). The purpose is unclear — what each can be used to do (except where summed to produce other measures) is undefined. An example of the lack of purpose in measurement being "A straightforward way of measuring a class is to count its attributes" — the purpose of doing so is not explained.

de Champeaux does not explicitly differentiate between measures and prediction systems. However, from the metrics defined, it seems that the ultimate aim is to predict such attributes as development effort.

Where the metrics are applied to the development, often the "results" (i.e. a list of numbers) are given with little or no interpretation as to their meaning. The few analyses provided offer little useful information, for example, for the vocabulary[35] metric, described as:

$$\mu\text{vocabulary(VE)} = \#\text{attributes in VE} + \#\text{states in VE} + \#\text{operations in VE}$$

Where VE is a vocabulary entry (with each entry corresponding to a class description).

---

[35] The purpose is vague, but the vocabulary itself is described as "a set of all templated narratives of all classes, relationships, ensemble classes, and their instances that are expected to play a role in the formal model to be constructed. The identification of the entries for the vocabulary is a matter of good taste, experience, gut-level intuition, unjustified braveness, etc."

This suggests that it is some sort of size metric. Presumably we are to assume that the larger the value, the more effort will be required to implement the class, although this is not explicit. The total number of artefacts, maximum metrics value, minimum metric value, median metric value and average metric value is given. A histogram showing the frequency of the metrics values for classes in the system is presented, showing a positively skewed distribution. The author suggests that attention should be focused on artefacts with the highest scores and it is to these that the most capable members of the team should be assigned.

A major drawback for what is intended as an empirical study of effort estimation metrics is that there are serious omissions in data collection, illustrated with the following quotation.
"... The construction of these class descriptions is the lion's share of the analysis effort. Unfortunately, we did not track the development effort per class - nor for that matter the finer granularity of the summary diagram, static diagram, and dynamic diagram. Our notes indicate that the summary diagrams together took 3 hours. We don't have figures for the other diagrams. A best estimate is that the static diagrams took 2 hours and the dynamic diagrams took an additional 8 hours."

In the absence of information as to how these "best estimates" were derived, it is reasonable to assume they are "guestimates". In addition to being generally unsatisfactory (lack of analysis and interpretation of results), the study is obviously of limited use since vital actual project data is missing.

3.3.2.4. Rajaraman and Lyu

In (Rajaraman and Lyu 1992a; Rajaraman and Lyu 1992b), complexity metrics are presented. The metrics are intended primarily to measure coupling in C++ systems, but the authors feel they could be used for other object-oriented languages, although how is not explained. As with Chidamber and Kemerer, the premise is that highly coupled classes increase the complexity of the system, in that any changes to the class are more likely to affect other classes and thus make maintenance more difficult and testing more demanding. Rajaraman and Lyu, however, consider the necessary trade-off between coupling and inheritance, noting that inheritance involves coupling between parent and child classes and "is crucial to achieving reusability and extendibility ... but it has adverse affects on code understandability." Chidamber and Kemerer, on the other hand, imply that coupling inhibits reuse. These differing views reflect different ways of implementing reuse — reuse within an application, versus reuse between applications.

The authors represent coupling as a directed multigraph, with nodes corresponding to classes and arcs to interaction (i.e. coupling) between classes, such as referencing variables or using a function or method defined in another class. The four metrics are:

CIC (Class Inheritance-related Coupling)
A count of the number of accesses to variables and/or uses of functions defined in an ancestor class.

CNIC (Class Non-inheritance-related Coupling)
A count of the uses of functions or accesses of variables defined in a class which are not defined in that class or in any of its ancestor classes. This included the use of "friend" functions and global variables/functions.

CC (Class Coupling)

This is a summation of inheritance and non inheritance related coupling of each class, and can more easily be calculated as CIC + CNIC, which equals the number of outward arcs from the corresponding node in the graph.

AMC (Average Method Coupling)

For a class this is the ratio of class coupling to number of functions, i.e. $AMC = CC/n$, where n is the number of member functions in the class.

Validation of the metrics involved ranking five C++ systems in order of perceived difficulty of maintenance, calculating the CC and AMC metrics as well as LOC, Software Science and Cyclomatic Complexity scores. Rank correlations were computed between the metrics and the difficulty of maintenance ranks, the result being that the CC and AMC metrics had higher correlation coefficients than the other measures, although McCabe came close for all but one of the systems studied. The authors conclude that this was because the developers of this system "are more knowledgeable in C++ and object-oriented programming and hence have exploited its language constructs more fully" and suggest the developers of the other systems may have programmed the systems in a functional rather than object-oriented way.

Criticisms can be made of the authors' approach to validation. It appears that not only were the developers of the systems asked to rank them in order of perceived difficulty, but the majority of the developers may not have developed the software in an object-oriented manner . They " ... may have programmed in C++ as they would in a language based on functional decomposition like COBOL". This is indeed possible in C++, since it is a hybrid, and is a factor to consider in developing metrics for C++, but in this experiment means that we may not (since the authors

themselves seem unsure) be comparing like with like. The metrics were collected from each system and the results for each class in the system correlated against the subjective ranking for that class. This would allow for the "definite" object-oriented projects to be considered separately for the "doubtful" projects. However, project data is not supplied, just one set comparing two measures against LOC, Halstead's Software Science and McCabe's Cyclomatic Complexity for one project, so it is not possible to re-evaluate the data.

The goal or aim of the metrics are not met by the metrics as they stand. The stated aim is to use a measure of coupling as an indicator of maintenance difficulty, which seems reasonable enough. However, the measures defined are not used in prediction systems to obtain any predictions, nor is maintenance difficulty explicitly defined, instead subjective assessment is used as a proxy for some collectable measure. It seems reasonable to conclude that the authors mean either maintenance effort or reliability, since they state that data such as mean time to failure and mean time to repair would have improved the validation. The authors have thus not as yet demonstrated the validity or usefulness of their measures, but on a positive note, the definitions appear clear enough for the measures to be applied in other studies.

3.3.2.5. Abreu

Abreu presented a number of object-oriented metrics in 1993 (Abreu 1993). Although stating that traditional metrics such as those of McCabe and Halstead are aimed at procedural languages and such metrics do not address object-oriented concepts, Cyclomatic Complexity, Halstead's volume metric and Henry and Kafura's information flow metrics are included in the proposed framework for OO metrics (TAPROOT).

Subsequent work (Abreu and Carapuca 1994) does not pursue this and concentrates on new metrics for object-oriented systems. The metrics are intended to evaluate object-oriented mechanisms considered to be important by the authors, such as inheritance, encapsulation, polymorphism, in relation to quality, productivity and reuse.

In (Abreu and Carapuca 1994), the authors list seven criteria which the metrics they derive should meet. The last criterion is that metrics should be language independent. This immediately makes the task of deriving valid and useful metrics more difficult, since there are considerable differences between languages, particularly, for example, "pure" languages such as Smalltalk and hybrids such as C++, which seem likely to have some influence on the validity or usefulness of a metric. The authors meet this criteria by avoiding specific language constructs, and using a more abstract approach. In all some 25 metrics are defined, most as inputs into the main metrics or factors. However, textual definitions are vague, and some of the inputs are undefined. The authors state that they wish to avoid the "YAM" (yet another metric) trap, but have themselves presented a long list of what must, in the absence of formal or empirical evidence, be regarded as speculative metrics. Little explanation is given as to why the measures and factors are trying to capture the specified attributes, save that results can be used "to compute design heuristics" — again, how to do so and with what limits are not specified. The authors state in (Abreu and Carapuca 1994) that a "study of correlation between MOOD metrics and quality attributes ... will be one of the next steps", confirming the immature nature of the work. They also suggest that some of the metrics "can be combined to obtain a generic OO software system complexity metric". The first stage in such a process is the evaluation of the metrics against Weyuker's axioms (Weyuker 1988). The inadequacies and contradictions of these axioms has been discussed in section 2.2.1.

However, the MOOD metrics have since undergone some refinement and empirical validation (Abreu, Goulao et al. 1995; Abreu and Melo 1996). In (Abreu, Goulao et al. 1995) six metrics are defined and applied to a number of class libraries, ranging in size from 4884 LOC and 35 classes to 74895 LOC and 128 classes. In a further study, the metrics are applied to small student projects in a controlled experiment (Abreu and Melo 1996). The definitions given in this section will use the refined MOOD metrics. Notably the number of metrics defined is reduced. The equations have undergone some "fine tuning" and, for both the written and mathematical definitions, the situation with regard to inheritance is clarified, where appropriate. In other words the metrics have been refined in the light of the comments about OO metrics in general, namely that there has been ambiguity as to whether or not inherited methods, attributes etc. have been included.

The MOOD metrics are categorised as follows: AHF and MHF are measures of information hiding; MIF and AIF are measures of inheritance; COF is a measure of coupling (not including inheritance coupling) and PF measures polymorphism. The metrics are defined and explained as follows using the definitions in (Abreu, Goulao et al. 1995):

(i) Method Hiding Factor (MHF)

$$MHF = \frac{\sum_{i=1}^{TC} Mh(Ci)}{\sum_{i=1}^{TC} Md(Ci)}$$

TC = total number of classes in the system

$Md(Ci) = Mv(Ci) + Mh(Ci)$ = methods defined in Ci

$Mv(Ci)$ = visible methods in class Ci

$Mh(Ci)$ = hidden methods in class Ci

(ii) Attribute Hiding Factor (AHF)

$$AHF = \frac{\sum_{i=1}^{TC} Ah(Ci)}{\sum_{i=1}^{TC} Ad(Ci)}$$

$Ad(Ci) = Av(Ci) + Ah(Ci)$ = attributes defined in Ci

$Av(Ci)$ = visible attributes in class Ci

$Ah(Ci)$ = hidden attributes in class Ci

Metrics (i) and (ii) are based on the assumption that information hiding (encapsulation) has a positive effect on quality, by reducing the effects of complexity, and thus its use should be promoted. For attributes, this would be reflected in high values for AHF. For MHF, there is a trade off between abstraction and method hiding (corresponding to a high value of MHF) and class functionality (measured by the number of visible methods) indicated by a low value for MHF. Thus the developer might use these metrics with an upper and lower limit of acceptable values.

(iii) Method Inheritance Factor (MIF)

$$MIF = \frac{\sum_{i=1}^{TC} Mi(Ci)}{\sum_{i=1}^{TC} Ma(Ci)}$$

$Ma(Ci) = Md(Ci) + Mi(Ci)$ = available methods in Ci

$Md(Ci) = Mn(Ci) + Mo(Ci)$ = methods defined in Ci

$Mn(Ci)$ = new methods in Ci

$Mo(Ci)$ = overriding methods in Ci

$Mi(Ci)$ = methods inherited in Ci

(iv) Attribute Inheritance Factor (AIF)

$$AIF = \frac{\sum_{i=1}^{TC} Ai(Ci)}{\sum_{i=1}^{TC} Aa(Ci)}$$

$A_a(C_i) = A_d(C_i) + A_i(C_i)$ = attributes available in C_i

$A_d(C_i) = A_n(C_i) + A_o(C_i)$ = attributes defined in C_i

$A_n(C_i)$ = new attributes in class C_i

$A_o(C_i)$ = overriding attributes in class C_i

$A_i(C_i)$ = attributes inherited in class C_i

Metrics (iii) and (iv) are measures of inheritance. Both anecdotal and empirical evidence exists (Cartwright and Shepperd 1997a; Chidamber, Darcy et al. 1997) which suggests that use of inheritance can have adverse effects such as increasing the likelihood of defects (probably linked to the relative difficulties in understanding and testing classes in an inheritance tree). This evidence raises questions regarding the useful effects of inheritance, so these metrics should be used this knowledge in mind.

(v) Polymorphism Factor (PF)

$$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]}$$

$M_o(C_i)$ = overriding methods in C_i

$M_n(C_i)$ = new methods in C_i

$DC(C_i)$ = number of descendants of class C_i (derived classes)

Polymorphism can be affected via inheritance, and like inheritance, there is a trade-off between its useful and complicating effects. Whereas polymorphism allows for flexibility in refining classes without affecting clients, it also complicates tracing control flow, making it harder to understand and debug code

(vi) Coupling Factor (COF)

$$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client(Ci, Cj)\right]}{TC^2 - TC - 2 \times \sum_{i=1}^{TC} DC(Ci)}$$

$TC^2 - TC$ = maximum number of coupling in a system with TC classes

$2 \times \sum_{i=1}^{TC} DC(Ci)$ = maximum number of couplings due to inheritance

$$is - client(Cc, Cs) = \begin{cases} 1 & iff\ Cc \Rightarrow Cs \wedge Cc \neq Cs \wedge \neg(Cc \rightarrow Cs) \\ 0 & otherwise \end{cases}$$

where $(Cc \Rightarrow Cs)$ means client class Cc contains at least one reference to a method or attribute of supplier class Cs and $(Cc \rightarrow Cs)$ indicates an inheritance relation.

It has long been accepted that whilst some coupling is unavoidable, this should be minimised. The premise behind OO designs is that classes will co-operate with each other to perform some task rather than repeating code or loading all the functionality necessary into one object. Thus coupling between communicating objects must be accepted as necessary and offset against the benefits of an object-oriented design. Further coupling will occur when inheritance is used, although it is made clear that this metric is not intended to capture inheritance based-coupling. This metric should, therefore, also be used with an upper and lower bound.

In (Abreu, Goulao et al. 1995) the authors conclude that the sample to which they applied the metrics is too small to conduct a meaningful empirical study. The tests carried out on the sample indicate that AHF and MIF are size-dependent and that AHF has a strong negative correlation with MIF, and PF has a strong positive correlation with COF.

The authors conclude that the size dependence and correlation between the AHF and MIF metrics is coincidental, brought about by the small sample size, although this is not demonstrated. The PF/COF correlation is felt to be due to an outlier value of COF for one system. This conclusion is reached after setting the apparently anomalous value equal to the average COF for the other four systems and recalculating the correlation. With such a small sample size it is, as the authors admit, hard to draw any meaningful conclusions. Additionally, the method adopted for dealing with an outlying value is highly questionable. Standard procedure is to remove an outlying value completely or re-expressing values after applying a transformation to all. It is understandable that the authors feel that reducing the sample size further is undesirable. However, by setting the anomalous value equal to that of the average for the other values, they are merely causing a "flatter" line to be drawn among values with little correlation by adding one more favourable value. This conclusion can be drawn since the authors argue that the outlier is forcing a correlation that doesn't really exist and thus the other values are likely to be much lower with little correlation.

In (Abreu and Melo 1996) the MOOD metrics are evaluated against eight small information management systems developed from identical requirements under controlled conditions. The MHF and AHF metrics are refined slightly:

$$MHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{mi}))}{\sum_{i=1}^{TC} M_d(C_i)}$$

where

$$V(Mmi) = \frac{\sum_{j=1}^{TC} is\_visible(Mmi, Cj)}{TC - 1}$$

and

$$is\_visible(Mmi, Cj) = \begin{cases} 0 & iff \begin{cases} j \neq 1 \\ Cj \quad may\_call \ \ Mmi \end{cases} \\ 1 & otherwise \end{cases}$$

$$AHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{Ad(Ci)} (1 - V(Ami))}{\sum_{i=1}^{TC} Ad(Ci)}$$

$$V(Ami) = \frac{\sum_{j=1}^{TC} is\_visible(Ami, Cj)}{TC - 1}$$

$$is\_visible(Ami, Cj) = \begin{cases} 0 & iff \begin{cases} j \neq 1 \\ Cj \quad may\_call \ \ Ami \end{cases} \\ 1 & otherwise \end{cases}$$

The stated aim of the paper is to evaluate the impact of object-oriented design on the quality characteristics defect density and rework. The metrics are collected from the source code by the MOODKIT tool developed to support the MOOD metrics. Defects were detected during white box testing and failures during black box testing. Rework effort is expressed as manhours taken to correct the discovered defects. These quality characteristics were correlated (Pearson) against the MOOD metrics extracted from the source code. The highest correlations were a negative correlation between MIF and defect density (and thus rework) and a positive correlation between COF and defect density, failure density and rework. The first of these, the negative correlation between MIF (Method Inheritance Factor) and defect density/rework runs counter to

other studies on the effect of inheritance on maintenance activities. Such studies include the case study and experiment described in chapters four and five of this thesis, also to be found (Cartwright and Shepperd 1997b; Harrison and Counsell 1997; Cartwright 1998). Abreu and Melo suggest that that the results show inheritance to be a technique to reduce the defect density in code when used "sparingly" but not at higher levels where they feel the beneficial effects will reverse, but with no evidence to support this claim. The authors do not, unfortunately attempt to explain this phenomena. In Daly's study (Daly 1996), a reduction was found in maintenance effort (in terms of adding functionality) in systems with up to three levels of inheritance (or DIT = 0 in terms of the CK metrics (Chidamber and Kemerer 1994), when compared with a flat structure. This effect was reversed when systems with five levels of inheritance were maintained. Perhaps the failure of MIF to take depth into consideration might help explain the claims made. The high positive correlation between COF (Coupling Factor) with the density and rework measures supports the generally accepted contention that coupling increases complexity and thus increases the potential for defects and maintenance effort by decreasing understandability. However, further analysis by Abreu and Melo, which includes putting the measures into a multiple regression equation suggests that the two inheritance metrics MIF and AIF, contribute comparatively little to the regression model. The adjusted R square for the model is high, particularly for defect density and rework, although one would expect these two to correlate since rework is dependent on the number of defects to a large extent.

To conclude, further empirical study is needed to establish the usefulness of the metrics, the coupling metric, COF looks to be the most reasonable and promising at present. The observed effect of the inheritance metric, MIF, is interesting and warrants further study to see if it negatively correlates with defect density in other systems. Additionally a further

study to test the expected result of a high MIF, that is a positive correlation with defect density, would be interesting.

3.3.2.6 Hopkins

Hopkins (Hopkins 1994), proposed complexity metrics to assess design quality, concentrating on class interface. The author states that measuring class interface complexity will "provide important information on the ease of understanding of the class interface (which is needed for reuse, maintenance, rework and redevelopment) ... (and) it might be reasonable to assume that the complexity of the interfaces will give some idea of how difficult the class will be to design (correctly) and implement."

Metrics are defined on a method and class level. Method interface complexity is defined as the number of different classes/types possibly returned from the method plus the sum of the number of classes/types for each of the arguments in the method. This is denoted as

$$IC_{meth} = N_{return-classes} + \sum_{i=1}^{N_{args}} N_{arg-classes}(i)$$

This measure is somewhat confusing. Presumably $N_{arg-classes}$ refers to the number of classes which could possibly supply a particular argument, although this is not obvious from the text. The reasoning behind the measure is that a method is more complex if it has a large number of arguments, or if the arguments can be of different types or the result returned can be of different types.

Class interface complexity is considered for both the public and private interfaces. For the public interface this is achieved by the summation of the method complexities for each method in the public interface.

$$IC_{pub} = \sum_{i=1}^{Npub} IC_{meth-pub}(i)$$

The aim is that the value calculated for this metric is independent of the nature of the interface.

The complexity of the private interface is given as:

$$IC_{prot} = \sum_{i=1}^{Nprot} IC_{meth-prot}(i) = \sum_{j=1}^{Iprot} IC_{iv}(j)$$

In this case the complexity of instance variables is included with the complexity of the methods in the private interface. Complexity of instance variables is given as $IC_{iv} = 2 \times N_{classes}$ for instance variables with read/write access and as $IC_{iv} = N_{classes}$ for those which are read only, where $N_{classes}$ is the number of different classes/types to which an instance variable can refer. It is suggested that the two class metrics can be combined as follows.

$$IC_{class} = IC_{pub} + (weight \times IC_{prot})$$

A weighting is added to take account of the subclass interface which the author feels is more complicated to test and thus requires weighting, the suggested, and admittedly arbitrary, weighting is 5.

These metrics are not validated empirically or formally, nor is the way in which they can be used to predict reuse, maintenance etc., which seems to be the implied aim. Although mathematically defined, the textual explanation is complicated and it is impossible to be sure exactly what is being measured, still less why. With empirical validation, it is

impossible to ascertain whether the measures capture what they purport to, and whether once captured, this measure is useful in any way.

3.3.2.7 Graham's SOMA Metrics

These metrics (Graham 1995) are proposed for use within the SOMA method, which is described as a full lifecycle method. Here the ideas of development phases, felt to be a hangover from structured methods, are dispensed with, and development is seen as an activity network with no predefined sequence, but with necessary dependencies.

Some of the metrics are based upon those of Chidamber and Kemerer (Chidamber and Kemerer 1994), Lorenz and Kidd (Lorenz and Kidd 1994) and Henderson-Sellers and Edwards' MOSES method (Henderson-Sellers and Edwards 1994), the aim being to form a single metrics suite.

The suite is divide into two sets, applicable to the two models in the SOMA method, the Business Object Model (BOM) and the Task Object Model (TOM). A total of twelve metrics are given, plus it is suggested that the Chidamber and Kemerer metrics (with the exception of LCOM) could be collected from the BOM. The metrics described are in the main, counts, such as number of classes, number of tasks and so on. Some of these counts are method specific, although from the definitions give, they could be generalised to some other object-oriented methods. A complexity metric is defined for each model, class complexity is taken from the BOM and task complexity from the TOM. They are defined as follows:

BM1. Weighted complexity of each class (WCC)

$$WCC = WA * A + WM * LM * M + WR * NR * R$$

A = number of attributes and associations

M = number of operations/methods

R = number of rulesets

NR = number of rules per ruleset*average number of antecedent clauses per rule

LM = proportional excess of SLOCs per method over an agreed, language dependent, standard

WA, WM, WR are empirically discovered weights;

TM1. Weighted complexity (WCT) of each task, T

$$WCT = WI * I + WA * A + WE * E + WR * NR * R$$

I = number of objects per task (here object is meant in the grammatical sense, so usually corresponds to the number of noun phrases)

A = number of associated tasks (if any)

E = number of exceptions or side scripts

R = number of rulesets

NR = number of rules per ruleset*average number of antecedent clauses per rule

WI, WA, WE, WR are empirically discovered weights.

One of the TOM metrics, TM7 or number of task points (defined as the number of atomic tasks, i.e. those with no component tasks, shown as the leaf nodes of the tree), is said by Graham to be "the most important and most novel SOMA metric". It can be collected at requirements capture, and is considered by Graham to be a possible replacement for function points.

Graham suggests metrics to be collected later, from the physical design and code, LCOM* (a modified version of Chidamber and Kemerer's LCOM, as suggested in (Henderson-Sellers 1996) and cyclomatic complexity of methods. Also suggested is an effort estimation prediction $E = a + pTk$ ($E = a + pT^k$ in the earlier paper) where E is effort in man-hours, T is the task point count, p is the inverse of productivity in task points per man-hour (to be determined empirically) and k and a are constants, a being start-up and overhead costs, whereas k remains undefined. Productivity is considered to be a function of the level of reuse and may depend on other factors.

Graham seems to appreciate the need for metrics to be both formally and empirically sound, when he criticises Lorenz and Kidd's comment on Chidamber and Kemerer's metrics (Lorenz and Kidd dismissed the CK metrics as being too theoretical), saying "a sound approach to metrics must be grounded in both theory and practice". This statement concurs with what is one of the major themes of this thesis, namely that we need to take heed of measurement theory to ensure our metrics are mathematically valid but must also be pragmatic and ensure that they are empirically validated as extensively as is possible. Additionally the metrics are defined as part of a particular method and are not claimed to be generally applicable, which draws near to the argument for locally applicable metrics, which runs through this thesis. However, despite these statements it is unclear as to the level of empirical validation the metrics have undergone. Certainly the definitions of some of the metrics could be clearer, and counting rules need to be defined. There are omissions — no help is given with the "empirically derived" weights in the complexity metrics or the constants in the effort prediction system,

are these weights to be derived locally, if so, how?[36] It is also suggested that the weights for TM2, weighted complexity if each task, could be "zero if empirical study shows that a factor such as *E* has no effect." Since this implies that a total score of zero for the metric is possible, the metric obviously need further thought — it seems reasonable to expect the author to discover if such factors have an effect or not before proposing a metric which includes them. In the absence of empirical evidence, the metrics must be regarded as speculative and indeed comments such as "Some combination of these metrics ought to correlate with cost of build ... " reinforce this. Given that they are method specific (and consider implementation language to be an affecting factor) and based to some extent upon practical experience, there is reason to be hopeful that some of the metrics may, in the future, be shown to be valid or at least useful.[37]

A code metric to replace LOC and token counting for C, C++ and Java is presented. It is presumably intended as a prediction system for size. Counting rules are given for each of the three languages to take into account the differences between them. The metric is intended as a style independent measure. However, no supporting evidence is given, such as correlations between the metric and actual size measures, although the author suggests that "preliminary research suggests ... program size measures similar to those given by professional developers." indicating that any validation is assessed against expert opinion, rather than actual values. Neither is the prediction system(s) itself (themselves) explicitly defined, thus it cannot be calculated independently. The work must therefore be considered speculative, since it lacks both empirical and

---

[36] It can be assumed that Graham used locally derived weights when trying out the metrics at Swiss Bank, since references to data collection from a number of project across several large organisations are all in future tense.

[37] However, to date, I have been unable to discover subsequent publications regarding the practical applications and empirical experience of the metrics. A paper entitled (Graham 1996) covers the same ground and is extremely similar to the 1995 version.

formal validation. The lack of an explicitly defined prediction system means that it cannot be independently validated.

3.3.2.8 Harrison

Some results from a preliminary investigation into quality metrics for object-oriented systems are presented in (Harrison, Samaraweera et al. 1996). A number of measures were taken from code in order to try and predict certain attributes considered to be indicators of quality. The empirical study of specially developed C++ programs indicated correlations between some of the measures taken and the number of modification requests made, and some correlation between other measures and errors found during testing. More recent work (Harrison, Counsell et al. 1997) has investigated metrics based on those proposed by Lorenz and Kidd (Lorenz and Kidd 1994). Twelve metrics, the majority purporting to measure product size, were applied to three C++ systems, of 1.4 KLOC, 8.9 KLOC and 24 KLOC respectively, the largest being a commercial retail system.

It is interesting to note that unlike many studies on object-oriented metrics, the authors consider both criteria for theoretical validation of metrics alongside the empirical evaluation which is the main concern of the paper. In other words, although concentrating on empirical methods to assess the metrics, theoretical considerations are not ignored. In this study the authors present two hypotheses, firstly that the metrics are (indirect) measures of software complexity and secondly that all, bar one, of the metrics can be said to be measures of system size and thus be used to estimate system size.

The metrics are correlated [38]with subjective complexity for a class, SC, (ranked on an ordinal scale of 1-5 where 1 is trivial and 5 is very complex) and with non-comment source lines, NCSL, as a measure of size. The results for the smallest system indicate a strong negative correlation between SC and the number of public methods in a class (-0.67) and the number of methods in a class (-0.63). The authors suggest this runs contrary to the expected outcome, that a class with more methods would be more complex. This may reflect a design decision where classes are kept small, with fewer methods, and thus interact with other classes to perform a task. The L&K metric seems to suggest this makes the system more complex and is thus, according to the accepted notions regarding complexity and coupling, not a good thing. Such a design, is however, typically object-oriented. Is the premise behind an object-oriented design the problem, or should the traditional view of coupling be adapted to suit object-oriented designs? The authors conclude that a simplistic view of this metric, leading to a decision regarding optimum method size could be counter-productive. For the 8.9 KLOC system, the correlation between SC and PM, and between SC and NM is positive (0.50 and 0.88 respectively), more in keeping with the generally accepted idea that as a system increases in size, its complexity is likely to increase also

For NCSL the smaller size shows a positive correlation with both NM and NCR (0.48 and 0.55 respectively)[39]. The relationship between the number of methods and the number of lines of code needs no further explanation, and is confirmed by the results for the 8.9 KLOC system

---

[38] Pearson's, Kendall's and Spearman's correlation coefficient were all used. Comments will concentrate on the Spearman's Rho correlation since this is a non parametric test more suitable for ranked and discrete data as used in these tests, than the Pearson's Product Moment.

[39] Although at 0.48, the correlation between size and the number of methods is not especially strong, indicating that the methods and classes are small, again a feature of a typically object-oriented design.

(0.86). More interesting is the relationship between size and reuse, where larger classes are reused more than smaller classes in this system - a typical OO design tends towards small methods and OO is said to promote reuse. No such relationship is found in the 8.9 KLOC system.

The authors also emphasise the problem with the ambiguous definitions and objective for the metrics (thus bearing out the criticisms in section 3.3.2.2.).

3.3.2.9 Other metrics for object-oriented systems

A number of other metrics have been proposed. The purpose or attributes to be captured vary, as do the approaches to validation (if any). The following table lists, by author, proposed metrics. Independent validations of other's metrics are included.

theme /lesson:	goal	measurement or prediction system	definition: attribute /counting rules/metric	validation	valid/useful
Abbot et al (Abbott, Korson et al.)	Cognitive complexity measure from design to predict expert preference among design alternatives	Prediction system	Textual definition seems OK.	Against own criteria. Uses expert opinion rather than actual data.	Predicted expert preference between two alternatives in 16/20 comparisons.
Balasubramanian (Balasubramanian 1996)	Not explicit - desire to improve some of CK metrics & to add to metrics that are available	Measures since no guidance on what to do with them is given	Those based on CK metrics are given as formulae, but the new metrics has only a vague textual definition.	No attempt given	Applied to 2 student programs and results compared with CK metrics - results are not analysed. No evidence from which to draw conclusion.
Barnard (Barnard 1998)	Metrics to predict reusability	Selected measures are combined to	Variable - many are simple counts	Simple counts OK. Measures are applied to a	If assumptions upon which the measures are

		form a reusability score (P.S.)	so seem OK, but possibly potential for ambiguity. Others are CK metrics, others subjective scores	number of reuse libraries (thus components assumed to be reusable), where a common trend is available, metric is assumed to indicate reusability	based (that the libraries contain classes that are good examples of reuse) then the selected metrics seem to capture attributes indicative of reusability. Derived prediction system is untested.
Bieman & Zhao (Bieman and Zhao 1995)	Metrics to characterise use of inheritance	measures	Not explicit - my interpretation of data & comment is that inheritance structure starts at level 1, with 0 assigned to classes not involved.	Simple counts & descriptive stats not trying to make predictions or assess in terms of other characteristics at this stage	Useful breakdown of use of inheritance by application type - may lend weight to proposition that some problems better suited to OO approach than others.
Burbeck(Burbeck 1996)	Code metrics to measure complexity of Smalltalk methods to provide feedback during development /mainten-ance	measures	yes - seem well enough explained to apply	all simple counts on ratio scale	Gives suggested thresholds of acceptability. Emphasises advisory nature of metrics. In principle seems a good idea, (provided the assumptions made regarding complexity & advice given is correct. Would benefit from empirical study of how the metrics affect the development process.
Chen & Lu (Chen and Lu 1993)	Complexity metrics to indicate candidates for redesign	Complexity measures, coupling & cohesion measures, & a reuse indicator	The 3 complexity metrics use supplied tables of seemingly arbitrary values, which allow a possible range of values to be supplied, but no indication of	Not considered. Values from tables appear ordinal yet are summed Coupling metrics - summation of simple counts, cohesions, seems OK, class hierarchy is adding together	Experiment uses expert judgement to evaluate metrics. Not enough to convince.

			how to decide. Others metrics seem clear enough.	an number of counts of rather different things so is questionable.	
Ebert (Ebert and Morschel 1997)	Enhance quality/ maintain-ability	Complexity metrics to predict quality /maintainability	Simple definitions but counting rules could be open to interpretation	Mainly simple counts.	Low R^2 for model of maintainability (from subjective ratings) and no comparison of predictions with actual values.
Hudli *et al* (Hudli, Hoskins et al. 1994)	Evaluate design of class/program. Metrics said to relate to quality factors such as maintainability	Measures - each said to indicate something, but not how to use it.	Textual, many ambiguities.	None	No support offered. Seem entirely speculative and untried.
Moser et al (Moser and Nierstrasz 1996)	Effort prediction via size/ complexity.	Measures as inputs into a prediction system.	Textual and formal, but confusing - not all components adequately defined.	Yes but only scatterplots given.	Seems to be valid. Is out performed by FP
Sneed (Sneed 1995)	Effort prediction via size	"measure of volume" then used to predict effort.	Metrics defined but counting rules unclear.	None given.	Not valid - violation of scales/measurement theory - many of same problems as FP.
Wilkie & Hylands (Wilkie and Hylands 1998)	Coupling complexity metrics as extension to CK suite	Measure - how this would affect faults, maintenance effort is unspecified.	Metrics defined and simple example given.	None given - research is "on-going:	No empirical evidence as yet.

Table 3.2: More object-oriented metrics

3.3.3 Conclusions

It is evident from the literature pertaining to metrics for object-oriented systems that, with few exceptions, little heed has been taken of the need for more rigour in developing and validating measures or prediction. The presentation of metrics for object-oriented systems cover a spectrum of which the extreme can be characterised as follows:

• An informal approach lacking rigour in empirical validation, and often clarity of purpose and definition, but emphasising pragmatism as typified by Lorenz and Kidd, for example;
• The other very formal, presenting mathematical formulae and proofs using a closed systems approach, which by its nature means that the metrics are dissociated from the "real", pragmatic, empirical world to which measures and prediction systems must ultimately be applied, via some representational model. This extreme is represented by (Schmidt and Zimmermann 1994).

Between the extremes, publications which are not entirely speculative tend to favour either formal or empirical validations. Many suffer from lack of clarity either in the purpose of the measures or prediction systems proposed, and/or in the definition of counting rules and attributes.

Such speculation, immaturity of definition and of evaluation is acceptable in a very new field. After all, a starting point is needed, and the products of software development do not have direct parallels with the products of other disciplines. There is however, a legacy of what not to do in terms of the development of software measures and prediction systems (see chapter 2), as well as some examples of how to apply a degree of rigour to their development and validation (see the work of Kitchenham, Fenton and Pfleeger in particular).

Admittedly, a demand for absolute proof of a metrics worth and validity would be counterproductive, and indeed not possible to the satisfaction of all. It would preclude metrics which appear useful but are not valid according to measurement theory, such as Function Points. However, a certain amount of discipline, along the lines of the five problems highlighted in this thesis (see the introduction, 3.1 for a recap) would be helpful. It would show that the metrics had been developed with some

aim in mind, that thought had been given to their practical application, to whether they were capturing and/or predicting numerically what they purported to and whether they were ultimately of any use, at least in certain situations.

The following chapter (4) will assess the current state of software metrics, concentrating on the application of metrics for object-oriented systems.

Blank

In

Original

Chapter 4 An Empirical Study of An Object-Oriented System

Synopsis

The aim of this chapter is to add to existing empirical knowledge of object-oriented software by analysing a large, industrial object-oriented system using data collected from design and maintenance phases of the same system. The analysis uncovers patterns in the data which indicate relationships between some attributes, notably inheritance and defects. This chapter also considers the problems which were encountered in trying to apply predefined metrics and demonstrates the ease with which an accurate locally applicable prediction system can be derived.

4.1 Introduction

Chapter 2 concluded with a list of lessons to be learnt from past metrics development and validation. In chapter 3, research into metrics for object-oriented software was examined with regard to these themes or lessons and a number of metrics were critically evaluated according to the list. To recap, the lessons/themes are summarised below:

1. Lack of clear goal or aim;
2. Confusion between measurement and prediction systems;
3. Poor definition of attributes and counting rules;
4. Lack of or poor validation (formal and in terms of empirical evaluation);
5. Failure to determine validity and/or usefulness of metrics.

The desiderata can be derived:

Desideratum	Explanation
1. A clear goal	The goal of the measurement or prediction system must be clearly stated
2. Distinction between measurement and prediction system	It should be made clear whether the "metric" is a measure (direct) or a prediction system (using direct measurements of attributes in order to predict other, indirect attributes)
3. Clear definition of attributes and counting rules	The attributes of interest and the counting rules for obtaining them must be explicitly and unambiguously defined either by means of a formula or by a clear textual definition
4. Acceptable standard of formal validation and empirical evaluation	Formal validations should use measurement theory to establish that a measure is a proper numerical representation of the empirically observed attribute and to ensure calculations do not violate scales, or if the authors feel there is a good reason for doing so, explain why. Empirical studies should use techniques and tests appropriate to the data that has been collected, demonstrating usefulness by comparing predictions against actual results.
5. Demonstration of validity and/or usefulness.	The validity can be assessed formally and usefulness empirically. N.B. a metric may be useful without being valid, and vice versa.

Table 4.1: Ideal standards for metrics development

It was therefore felt important to incorporate the above into the empirical investigation, as far as circumstances allowed. It must be reiterated however, that the above are ideals. Certainly, it is reasonable to expect that the first two should be attained. The third is also important. To some extent the level of ambiguity depends upon the person reading/applying the metric, but the developer of the metric must look for possible ambiguity in the definition. The final two are more difficult to achieve.

The original aim of the empirical study was to assess the Chidamber and Kemerer (CK) metrics by seeing if they could be used to predict maintainability, measured by defects. However, due to difficulties in collecting the majority of the metrics from the available data, a

subsequent aim was introduced. The problems experienced with collecting the CK metrics will be described in section 4.4.

The new aim was to investigate the process of developing locally applicable metrics using available data and tool support. To clarify, the aim was not to propose new metrics for OO systems, but to demonstrate that reasonable, locally applicable prediction systems could easily be developed *in situ*. If the process of developing accurate metrics locally could be shown to be straightforward, this would allow project or quality managers to develop their own metrics rather than depend upon predefined metrics. Such predefined metrics may be based upon experience of applications, environments or projects not relevant to other organisations, and as such may not work as well for other organisations and may also require significant investment in tools such as analysers in order to collect the metrics. This study will be described in section 4.7.

4.2 System Background

The system to be analysed was a large subsystem (132+ KLOC, 32 classes) of a much larger telecommunications product. The subsystem was designed using Shlaer and Mellor's Object-Oriented Analysis (Shlaer and Mellor 1992) and coded in C++. Design documentation, incident reports and maintenance data (where a particular defect had been identified and corrected) were made available. The system had been delivered and the data supplied referred to defects identified from integration testing (when the software was placed under change control) onwards. This included 12 months post delivery usage. The organisation supplying the data was a large (around 20 000 employees), company well established in the industry and experienced in the development of telecommunications

systems. The company is ISO 9000 accredited and places a high emphasis on reliability. Extensive effort and resources are put into testing. Testing does not rely on simulation alone, model rooms containing "mock ups" of the system under test are also used. The system under study was the first OO development for a team of experienced C developers. All had undergone thorough training in C++, object-oriented concepts and the Shlaer-Mellor method.

4.3 Provisos for the Empirical Study

The goal of the empirical study was to use empirical techniques to discover simple size and defect prediction metrics for use within the co-operating section of Company X, with the following provisos:

The metrics must be easy and cheap to collect, utilising existing mechanisms, such as code analysers, CASE tools, fault logs, change requests etc.;

The measures taken should be available early in the lifecycle in order to predict attributes of interest not available until later in the life cycle, namely size and defects.

The following section, 4.4, is concerned with the attempt to apply the Chidamber and Kemerer metrics suite. It describes the problems encountered with collecting the metrics.

4.4 Application of the Chidamber and Kemerer Metrics Suite

The metrics suite consists of six metrics, namely

WMC (Weighted Methods Per Class)

DIT (Depth of Inheritance Tree)

NOC (Number Of Children)

CBO (Coupling Between Objects)

RFC (Response For A Class)

LCOM (Lack Of Cohesion Of Methods)

For a more detailed discussion refer to section 3.3.1, and to the original papers (Chidamber and Kemerer 1991; Chidamber and Kemerer 1994).

It was found that only two of the metrics, DIT and NOC could be collected from the available analysis/design documentation (both were collected from the Shlaer/Mellor Information model). The other metrics could only be collected by analysing code. It was decided to abandon the attempt to collect the remaining four metrics for the following reasons:

(i) the CK metrics are described as design metrics, any value they may have been proved to have had would be reduced if they could not be collected until the coding stage;
(ii) the static code analyser used at Company X did not collect the CK metrics, nor could the measures collected be used or adapted to suit;
(iii) there was little merit in recommending metrics which would cost the company in effort (in collecting) and/or financially (investing in a new analyser) if, as the experience indicated, so few could be collected by the analysis/design stage. It is well known and widely accepted that metrics are more valuable at earlier stages allowing designs to be revisited before coding, or allowing informed decisions regarding allocation of coding and testing resources.

It was not possible, then, to make an empirical "validation" of the CK metrics – they are presented as a suite of metrics and the individual role

of each is not clear. Those collected, DIT and NOC, could be analysed, along with the other independent measures taken. The results of the analysis are presented in section 4.5.

4.4.1 Initial Conclusions on the Usefulness of DIT and NOC

Little use was made of inheritance in the system, so mean values were very low and median values were nil. Such low levels limit the feasibility of such measures as predictors (they were not intended merely to indicate the presence or absence of a feature in the system, but to quantify the feature, in this case inheritance). On reflection, the value of NOC for a class and even a system overall is always likely to be low. Where low levels of inheritance are used, few classes will have children, and even where inheritance is used, it seems that more classes will be children than parents. In the context of this case study, however, higher levels of DIT have indicated a higher incidence of defects per KLOC.

4.5 The Effects of Inheritance on Defects

The CK metrics successfully collected, DIT and NOC, are both measures of inheritance. DIT measure the depth of the inheritance hierarchy and NOC measures the number of child classes belonging to a parent class. Both of these measures were collected from the Shlaer/Mellor Information Model, which could be described as an extended entity-relationship model, with entities becoming classes (Shlaer and Mellor use the term object rather than class). The relationships between the classes are described, indicating how a class uses another in some way and also where a class inherits from another.

Also collected (per class) were the number of defects and LOC (lines of code, counted as the number of end of line markers, ";"). LOC was chosen as a measure of size because it is a valid measure and was already in use within the organisation.

The DIT and NOC for each class were compared with the number of defects and LOC for each class. To allow for the affects of size, the number of defects was size normalised to give defects per KLOC per class.

There were just two inheritance trees or structures in the system (figures 4.1 and 4.2), one of two levels consisting of seven classes and the other of one level, consisting of five classes. There are two possible explanations for this. Firstly that there is little in the problem area that naturally lends itself to inheritance. This is probably true of many problem areas outside of the examples in OO texts, which often feature simple examples with naturally extensive specialisation, e.g. GUIs, simple drawing packages, classification structures etc. A paper by Bieman and Zhao (Bieman and Zhao 1995) examines 19 C++ systems totalling 2744 classes and concludes that the use of inheritance tends to be greater in GUI applications than the others in the study, with the mean depth of inheritance for GUIs being more than twice that for other systems. Secondly the analysis and design method used, Shlaer/Mellor, does not provide explicit support for inheritance – it is not discouraged, but there is no guidance in how to look for possible inheritance hierarchies as in some other OO methods.

Furthermore, the highest defect densities calculated were for classes at the lowest level of their respective inheritance hierarchies. Compare defects/KLOC in figures 4.1 and 4.2 below with a median defects/KLOC of 0 for the non-inheritance classes in the system. This suggests that classes which utilise inheritance should be thoroughly tested since they are

more likely to contain defects. This in turn indicates that the developers were right to be cautious about using inheritance for this project. One question raised by this is whether the developers were using inheritance "properly". Since there is little guidance available on what constitutes "proper" use of inheritance, the question can be answered thus; the developers were experienced (although this was their first object-oriented development) and "stuck rigidly" to the Shlaer/Mellor method. As has been mentioned this method does not explicitly support or encourage inheritance, though neither does it prevent or discourage its use. It seems likely that the caution of the team, lack of support by the method and the lack of obvious candidates for inheritance in the problem area were all factors in what would seem to be low levels of inheritance in the system.

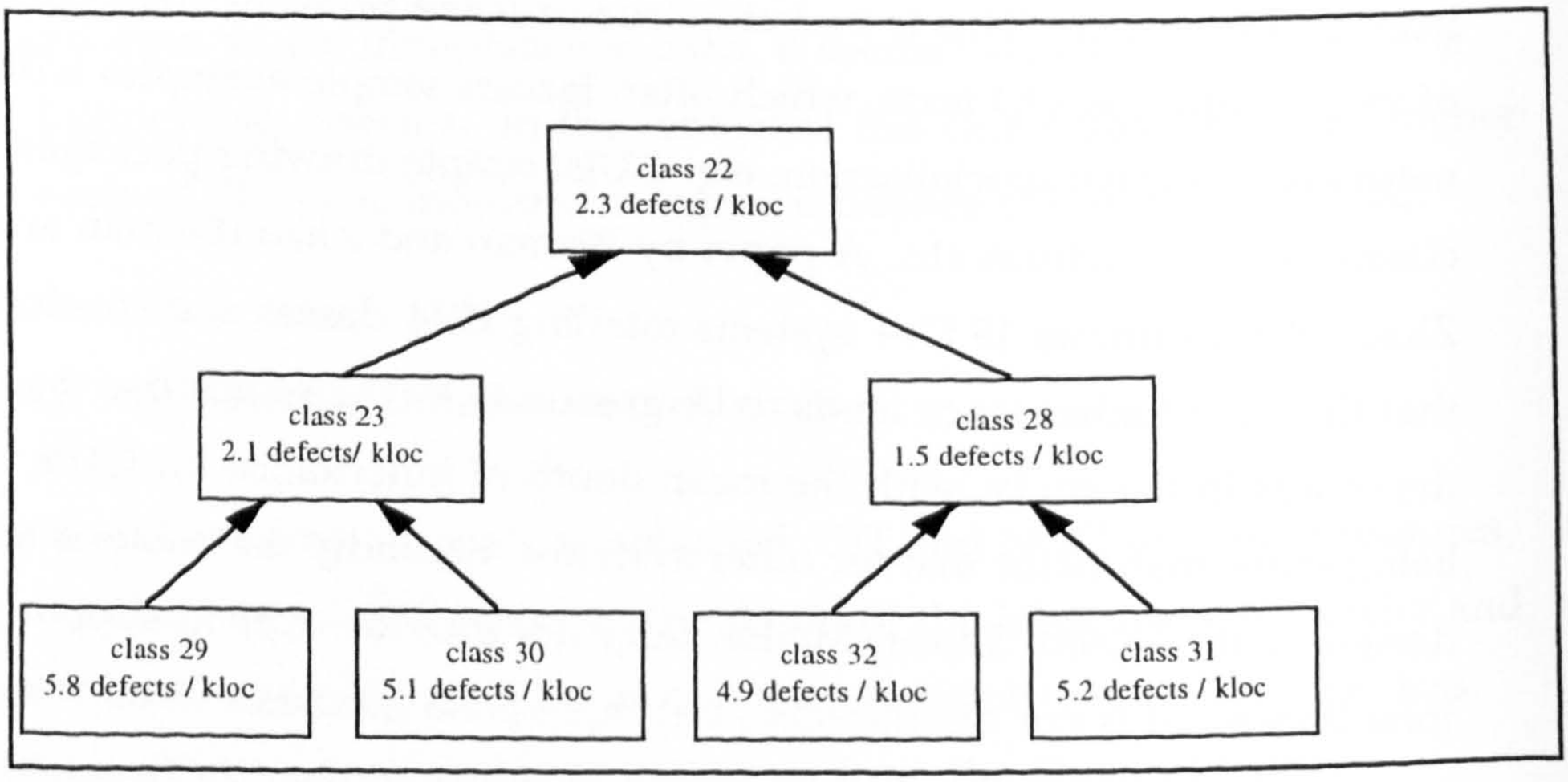

Figure 4.1: Larger inheritance hierarchy giving defects/KLOC

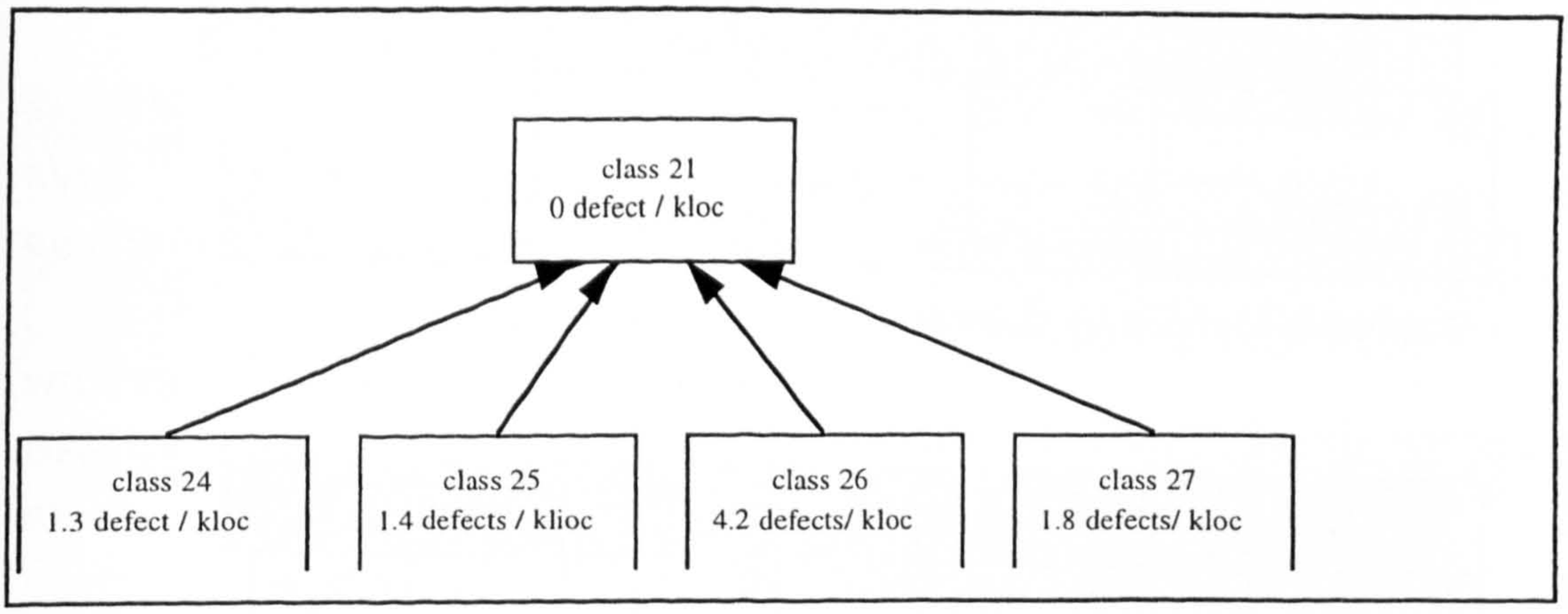

Figure 4.2: Smaller inheritance hierarchy giving defects/KLOC

Other summary statistics, with inheritance classes separated out from non-inheritance classes are shown in tables 4.2 and 4.3 below. It was suspected that defect density would be higher for classes within an inheritance structure than for those outside, since prior to size normalisation, obviously higher levels of defects tended to be associated with classes belonging to inheritance hierarchies. The raw data indicated means of 2.97 defects per KLOC for classes in an inheritance hierarchy and 0.5 defects per KLOC for those not involved in inheritance, a threefold increase in the incidence of defects/KLOC in inheritance classes compared with non inheritance classes (table 4.3). In order to test the hypothesis that the classes involved in inheritance structures were truly from a distinct sub-population, or whether the apparent increase in defects in inheritance classes occurred by chance, a two tailed unpaired t-test was applied. The result confirmed that they were indeed from a distinct sub-population, the F-value being calculated at 6.33, compared with a tabled value of 4.17 and with less than 1:1000 chance of this occurring by chance (2-tail prob. 0.00).

Group	*Count*	*Mean*	*Median*	*Min*	*Max*
No inheritance	20	3.05	0	0	14
Inheritance	12	16.50	17	0	47

Table 4.2: Defects by classes

Group	*Count*	*Mean*	*Median*	*Min*	*Max*
No inheritance	20	0.90	0	0	2.70
Inheritance	12	3.00	2.20	0	5.85

Table 4.3: Defect densities by classes

This analysis of the effects of inheritance was followed up with a small scale student experiment on the effects of inheritance on maintenance. This study was carried out using an experiment conducted by Daly (Daly, Brooks et al. 1996; Daly 1996). The experiment and results will be discussed in Chapter 5.

4.6 An Examination of the Data Distribution

A number of variables were extracted from the analysis/design models, incident reports and change control data. A factor in deciding what to collect was the ease of collection from the available documentation — the effort required to collect metrics is as much part of their "usefulness" as their accuracy in capturing or predicting information about the system. All of the variables were either automatically extracted from the TEAMWORK[40] model, or taken from incident report/change log data

---

[40] Casetool, CADRE Technologies Inc.

Mnemonic	*Variable*	*Explanation*
ATTRIB	Attributes	Count of attributes per class from the information model.
STATES	States	Count of states per class in the state model
EVNT	Events	Count of events per class in the state model
READS	Reads	Count of all read accesses by a class contained in the CASE tool.
WRITES	Writes	Ditto writes
DELS	Deletes	Ditto deletes
RWD	Read/write/deletes	Count of synchronous accesses (i.e. the sum of READS, WRITES and DELS) per class from the CASE tool.
DIT	Depth Inheritance Tree	Depth of a class in the inheritance tree where the root class is zero.
NOC	Number of Children	Number of child classes.
LOC	Lines of code	C++ lines of code per class.
LOC_B	Lines of code (body)	C++ body file lines of code per class.
LOC_H	Lines of code (header)	C++ header file lines of code per class.
DEFECT	Defects	Count of defects identified per class.

Table 4.4: Variables collected

Table 4.4 lists the 13 variables collected, including the two CK metrics, DIT and NOC discussed in section 4.4. The first nine variables characterise the OO system architecture or structure and may be collected at analysis or design time. Duplicates are eliminated from the counts of events and synchronous accesses. The remaining four variables can be regarded as management variables since they represent the size and defect proneness of the system.

The following table shows summary statistics (mean, median, minimum, maximum) of the variables collected. (Raw data is in appendix A)

Variable	*Mean*	*Median*	*Min*	*Max*	*Skew*
ATTRIB	8.66	4.5	1	32	1.27
STATES	18.03	13	0	114	2.56
EVNT	20.53	10.5	0	122	2.13
READS	16.25	11.5	0	83	1.93
WRITES	14.22	8.5	0	56	1.09
DELS	1.50	1	0	5	0.95
RWD	31.97	22	0	131	1.3
DIT	0.44	0	0	2	1.29
NOC	0.25	0	0	4	3.45
LOC	4178.50	3524.5	603	20165	2.26
DEFECT	8.09	2	0	47	1.63

Table 4.5: Summary statistics of variables collected

It is apparent that since the median value is in all cases lower than the mean, all variables are exhibiting some tendency to skew. This is confirmed by the skewness figure in the final column, revealing a positive skew, confirmed by the skewness coefficients. This is the consequence of a few very large classes. Even excluding these few very large classes, it is clear that classes have an unexpectedly high KLOC values, typically in excess of 3.5 KLOC. It also confirms the observation in section 4.5, that inheritance is not widely used, since median DIT and NOC values are zero. Lastly, the median number of defects is 2 although

there is wide variation with the maximum value of 29 defects in a single class.

Data skew can also be illustrated using boxplots examples of which are shown in figures 4.3 and 4.4 below.

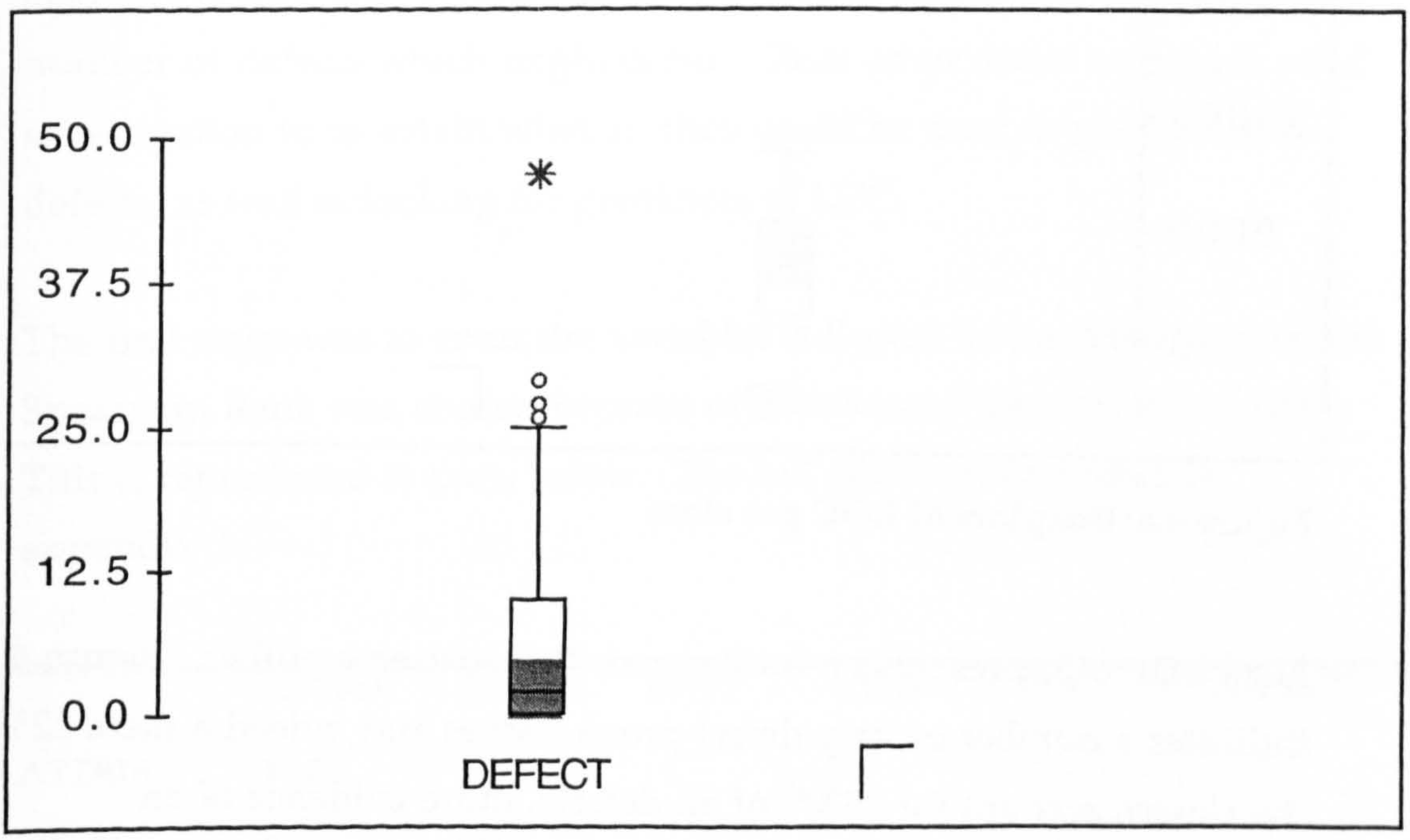

Figure 4.3: Boxplots of defects per class

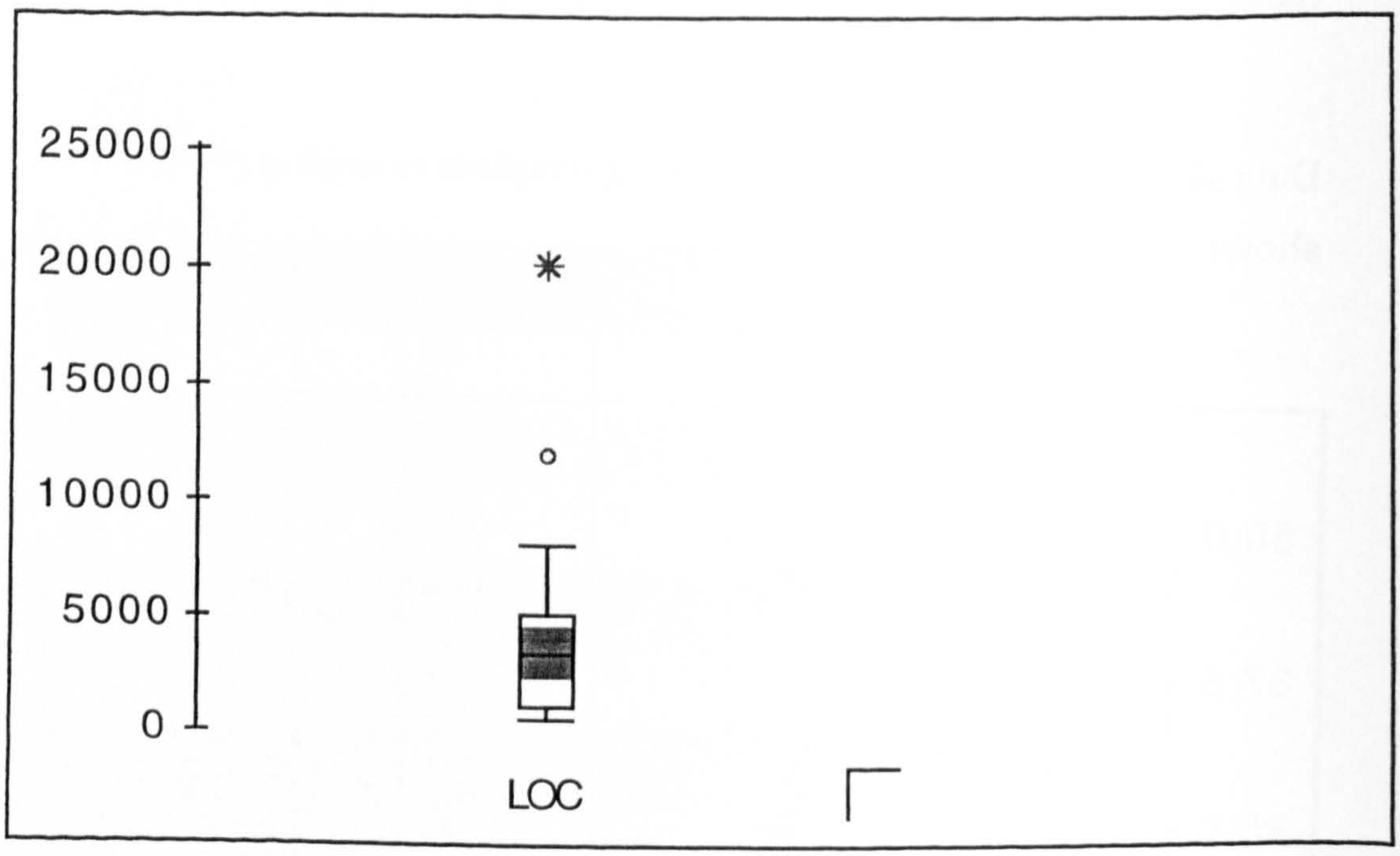

Figure 4.4: Boxplots of LOC per class

Note that 'o' represents an outlier and '*' an extreme outlier. Figure 4.3 indicates a number of very defect prone classes and indeed a mere 22% of the classes account for 75%[41] of all defects, more evidence of an approximate 20:80 "rule". Figure 4.4 indicates several unusually large classes, one in excess of 20000 LOC. The extreme outlier for DEFECT and LOC is the same class (class 22, see figure 4.1), thus emphasising the need to size normalise the defect data.

Class 22 is by far the largest class in the system (114 possible states, total LOC 20165, compared with the next largest, class 23, 60 possible states, total LOC 12101 and with average class size of 18.03 possible states and

---

[41] 22% of 32 classes is 7.04. This was rounded up to the nearest integer, making 8 classes. When rounded down to the top 7 classes the figure is 73% of defects.

total LOC of 1178.5). This makes it obvious that almost all of the measures taken were size driven.

4.7 Correlating Variables

Preliminary analysis indicates relationships between size and defects and also inheritance and defects. However, DIT is not useful to estimate the number of defects which might occur. Thus other direct measures need investigation to ascertain whether they could be used as predictors of defects, as well as looking for predictors of LOC.

The first stage was to enter the variables collected into a cross correlation. Spearman Rank was chosen because of the skewed distribution of data. This is reproduced in part, below. The full table can be found in appendix A.

	ATTRIB	STATES	EVNT	RWD	LOC	DEFECT
ATTRIB	1.000					
STATES	0.562	1.000				
EVNT	0.318	0.898	1.000			
RWD	0.508	0.858	0.859	1.000		
LOC	0.563	0.968	0.910	0.848	1.000	
DEFECT	0.166	0.751	0.838	0.769	0.759	1.000

Table 4.6: Results of Spearman Rank Correlation

Although correlation coefficients outside the range 0.296 to -0.296 are significant, it was decided only to consider those 0.75 and above/-0.75 and below since these could be considered strong correlations. Interestingly all of these variables correlate significantly with LOC (and for the most

part, with each other). This, together with the other inter item correlation suggests that as the size of a class increases, so does the number of states, events, synchronous accesses and the number of defects. This again underlines the need to use size normalised data to uncover effects which may be dominated by size (note the correlation coefficient of 0.759 for DEFECT/LOC).

4.8 Building Prediction Systems

The next stage was to build simple prediction systems for defects proneness (DEFECT) and size (LOC). The independent variables were chosen using the information from the cross correlation and fed into a stepwise multiple regression equation[42]. The R^2 and adjusted R^2 for any equation must be high (this indicates that the model fits the data well). Simple equations (using only one independent variable) were chosen over more complex equations using more variables. The reasons behind this were: adding in a second or third variable did not greatly increase the R^2 value; given that so many of the variables were quite highly correlated, collinearity may have been a problem; simple equations are preferable since the less effort needed to collect data and calculate equations, the more likely it is that data collection will be timely and calculation successful. Additionally, the variables selected for input into the equation were all available at the analysis and design stage, allowing for earlier (and thus potentially more valuable) predictions. Below is the resulting equation for predicting DEFECT

---

[42] This is the approach advocated by (Kitchenham, Pleeger et al. 1995)to ensure that only aspects that contribute to the model are included.

Dependent variable is: **DEFECT**
No Selector
R squared = 87.6% R squared (adjusted) = 87.2%
s = 4.240 with 32 - 2 = 30 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	3821.50	1	3821.50	213
Residual	539.221	30	17.9740	

Variable	Coefficient	s.e. of Coeff	t-ratio	prob
Constant	-0.575487	0.9566	-0.602	0.5520
EVNT	0.422246	0.0290	14.6	≤ 0.0001

DEFECT = -0.58 + 0.42(EVNT).

Table 4.7: Regression equation and R2/adjusted R2 for DEFECT

The high R^2 shows that the equation can be considered a good predictor of defects. Since the constant and multiplier are both less than zero they are rounded to two decimal places at this stage, since it is obvious that rounding to the nearest integer would have a serious distorting affect. Consequently rounding to integer figures will be performed upon the predictions themselves.

The standard error of coefficient for the constant is high, indicating a potentially large spread of values around the intercept, borne out by the probability that indicates that the intercept given is not significantly different from zero. This confirms a lack of confidence in a negative intercept, offering some support for the rounding up of negative defect predictions (produced using this equation) to zero, which occurs in section 4.9.1.1.

The standard error for EVNT is low, as is the probability that it is significant, so we may have confidence in this figure.

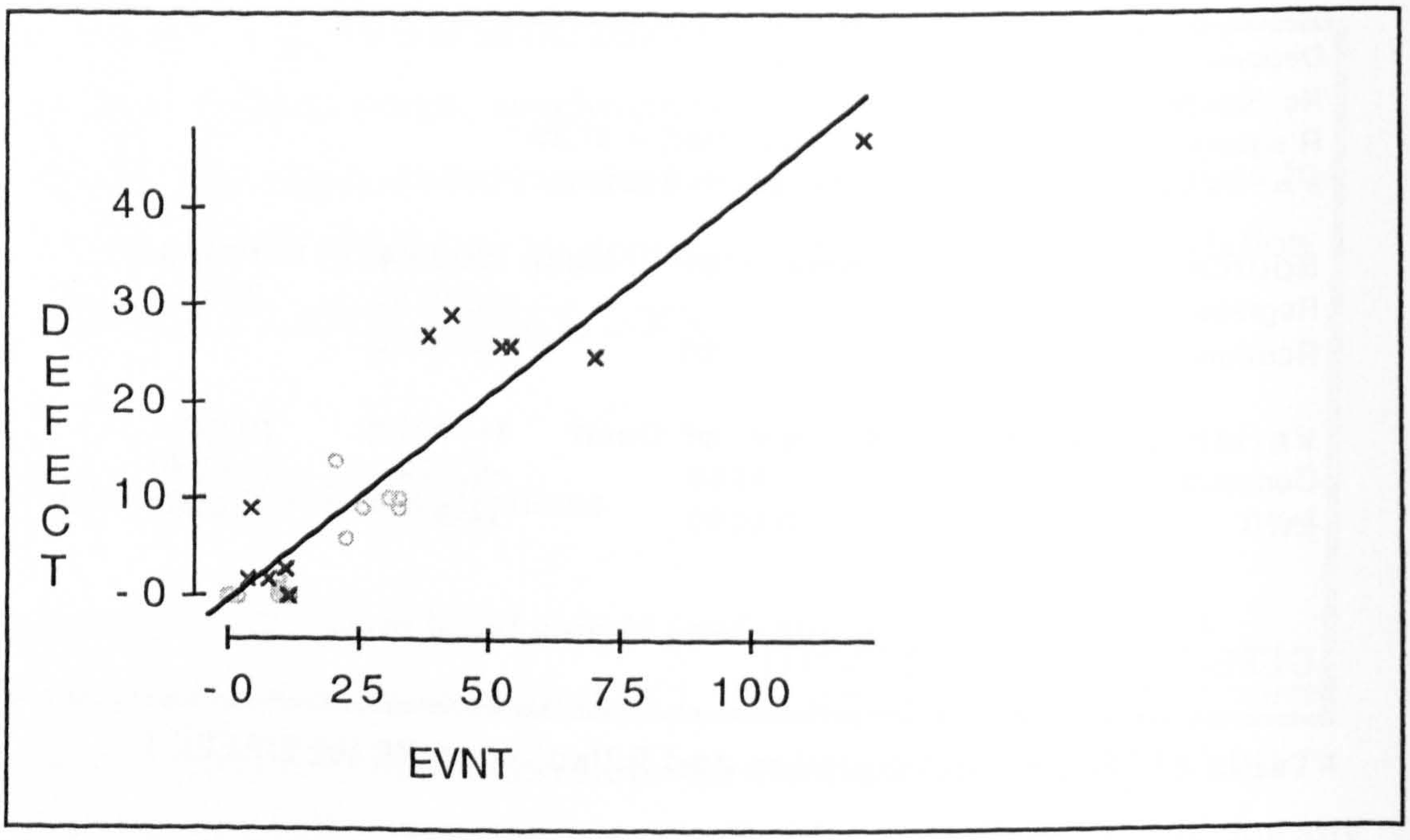

Figure 4.5: Scatterplot with regression line for DEFECT against EVNT (x=inheritance, o=no inheritance)

Out of interest the addition of DIT into the equation improves the adjusted R^2 value. See Table 4.8 below. However, the larger negative intercept is less pleasing, as is the fact that it is significantly different from zero (probability = 0.028).

Dependent variable is: **DEFECT**
No Selector
R squared = 93.8% R squared (adjusted) = 93.4%
s = 3.052 with 32 - 3 = 29 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	4090.66	2	2045.33	220
Residual	270.062	29	9.31250	

Variable	Coefficient	s.e. of Coeff	t-ratio	prob
Constant	-1.65497	0.7173	-2.31	0.0284
DIT	4.38205	0.8151	5.38	≤ 0.0001
EVNT	0.381447	0.0222	17.2	≤ 0.0001

DEFECT = -1.6 + 4.38(DIT) + 0.38(EVNT)

Table 4.8: Adding DIT to the regression equation and R2/adjusted R2 for DEFECT

This seems to confirm that inheritance classes in this system seem to be more defect prone, shown by the positive coefficient for DIT. Again from the dataset used, it was seen that the highest densities of faults came from classes at the lowest levels of their inheritance hierarchies. It may thus be fair to say that the fact that classes inherit (i.e. those for which DIT=1 or above) are the most interesting. Thus the inheritance as measured by DIT may also be of use in indicating the presence of a higher incidence of defects, although not in predicting how many.

Dependent variable is: **LOC**
No Selector
R squared = 96.7% R squared (adjusted) = 96.6%
s = 737.0 with 32 - 2 = 30 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	475082696	1	475082696	875
Residual	16296130	30	543204	

Variable	Coefficient	s.e. of Coeff	t-ratio	prob
Constant	1101.01	166.7	6.60	≤ 0.0001
STATES	170.676	5.771	29.6	≤ 0.0001

LOC = 1101 + 170(STATES)

Table 4.9: Regression equation and R2/adjusted R2 for LOC

Note that figures in the equation for LOC are rounded to the nearest integer. This must be done at some point, since we must have integer values (0.6 of a line of code would be nonsensical). In this case large integers are involved, the loss of the decimal places is unlikely to have any untoward effect. Note also the large positive intercept, which is significant. This indicates that any class will have a certain amount of code associated with it before any functionality is added. In C++ classes are divided into header and body files, with the header containing class declarations (e.g. data, template, function) and some definitions (e.g. type, constant, but **not** data or ordinary functions). Stroustrup (Stoustrup 1997) lists 14 types of information which may be included in a header file, which accounts for a overhead in terms of class header code, before functionality is added (in the body file).

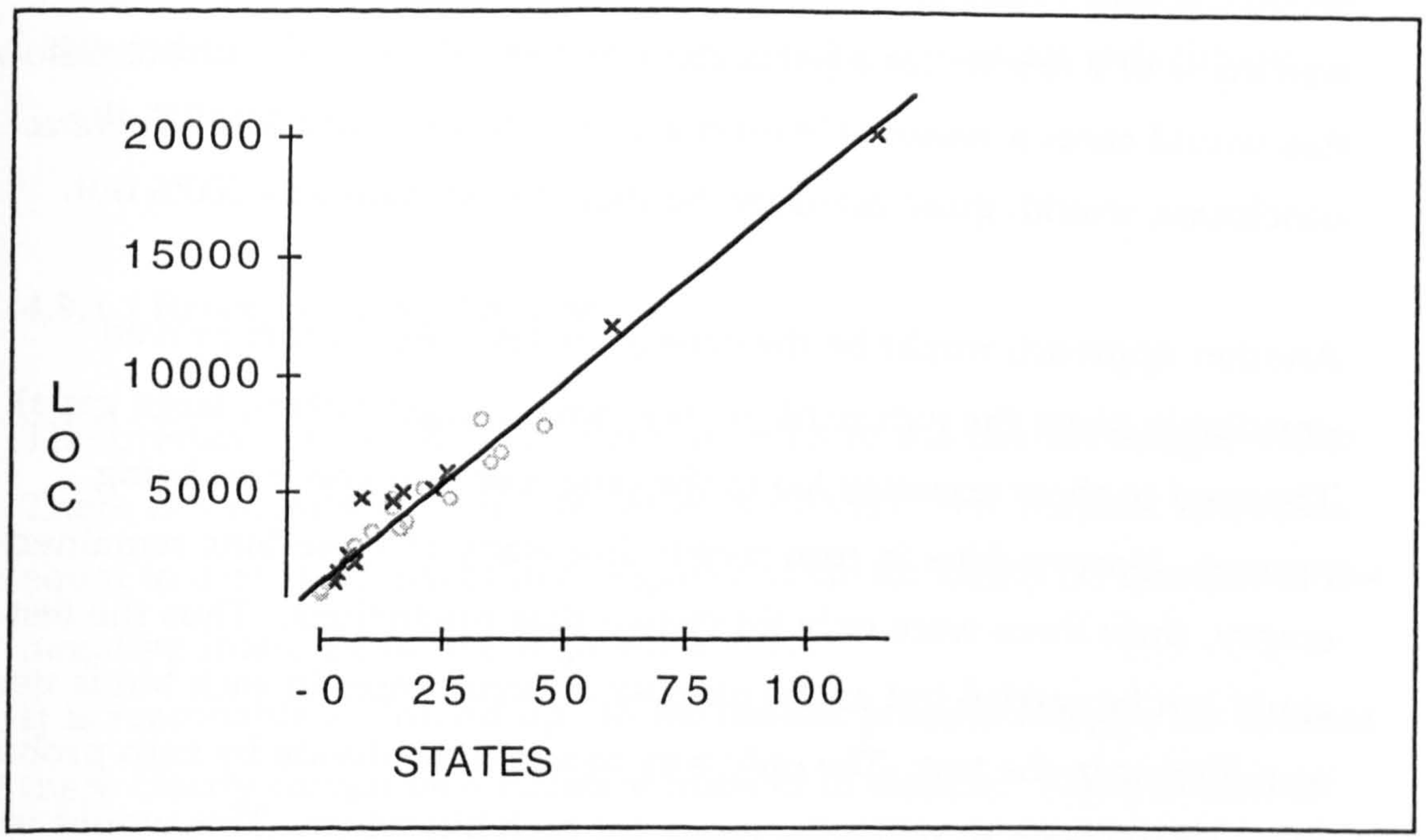

Figure 4.6: Scatterplot with regression line for LOC against STATES (x=inheritance, o=no inheritance)

Figure 4.6 shows the distribution of values along the regression line. The is obviously little scatter, indicating a strong relationship between the number of states and size in LOC.

4.9 Testing Prediction Systems

Since historical project data was available, the prediction systems (regression equations) could be applied to and tested against actual data from the same project i.e. the number of states for a class could be entered into the equations and the result compared against the actual LOC for that class, likewise for the other prediction system.

A means of testing the overall accuracy of prediction for the dataset was needed. A common test is MMRE (mean magnitude of relative error). However, the nature of much of the data (particularly for defects), being

small, discrete values made it unsuitable. For example, if a prediction was made of 2 defects for a particular class, and the actual number was 1, this would seem a reasonable estimate. However, using MMRE, the conclusion would, quite correctly, be that the estimate was 100% out.

Another approach might be the chi-square test. Again this proved unsuitable given the nature of the raw data (integer values, large gaps). The need to show accuracy led to the ranges or bins specified being narrow. Narrow bins in turn meant that many of these bins remained empty, since there were only thirty-two data predictions. Thus the test could not be carried out as the number of occurrences in each bin is used as a divisor in the test. The only way to avoid the divide by zero problem would have been to make the range for each bin wider. This would in turn, mean that the accuracy of each prediction was less apparent. To illustrate, for the number of defects, if the size of a bin had to be extended to the range 0-19, for example, this tells us less about the accuracy of predictions than smaller bins of say, 0-4, 5-9, etc.

One avenue would be to re-express the data in such a way as to preserve the order and size of the ratio between values. The following section discusses data re-expression.

4.9.1 Data Re-expression

Re-expression of data is an accepted technique for data analysis. The purpose being to make data analysis easier, by making the data more symmetrical (i.e. less skewed). Certain patterns which may not be obvious in raw data, for example, may become more apparent after data is re-expressed. However, not all manipulation is acceptable. It is important, for example, to preserve order. Tukey (Tukey 1977), gives

categories of data, such as ranks, counts, counted fractions, amongst others, and suggests appropriate re-expressions in order to facilitate analysis.

4.9.1.1 Re-expressions Applied

The predictions for DEFECT were rounded to the nearest integer value. There is a possibility that low values for EVNT (i.e. number of events equal to 0 or 1)will result in a negative value for DEFECT, because of the negative intercept on the regression line.
It is reasonable to "round up" to the nearest positive integer, i.e. 0, since there clearly cannot be a negative number of defects. There is further justification for this since there is little confidence in the negative intercept. The premise upon which the prediction system is based is that the greater the number of events generated by a class, the greater the number of defects is likely to be. If the number of events is 0 or 1, then clearly a low, but non-negative prediction would be a reasonable outcome. Thus the decision was made that any negative predictions should be considered to be predictions of zero. This is born out by a comparison – predicted negative defect values correspond to an actual defect value of zero.

Re-expressing the data (actual and predicted) using logs is a simple technique which will make a more normal distribution whilst preserving order. For LOC and LOC prediction, logs of the data will be taken (0 values are not possible using the regression equation derived), whereas with DEFECT and the predictions for DEFECT, the data will be "started" before re-expression by adding 1. This is because 0 values are possible and using lower values to "start" the data would lead to non

integer values.[43] Tukey suggests this procedure where small counts are involved.

Re-expression using square root is regarded as being halfway between raw data and log re-expression. This will also be shown in the following section for comparison purposes.

4.9.2 Comparing Transformed Actual and Predicted Values

Some exploratory analysis comparing the transformed actual and predicted data was carried out.

It was necessary to allocate unique names to each variable to reflect the transformation or change that took place. The following table may be used to trace the relationships between variables.

---

[43] The reader is referred to (Tukey 1977) chapter 3 on easy re-expression and chapter seven on choice of expression

Mnemonic	*Description*
LOC	lines of code per class (actual value)
PREDLOC	predicted lines of code per class
LLOC	log of LOC
LPRC	log of PREDLOC
√LOC	square root of LOC
√PRC	square root of PREDLOC
DEFECT	no. of defects per class
PDFCTRND	predicted no. of defects per class where -ve values are set to 0 and other non integer values rounded to nearest integer
DEFECT+	DEFECT+1 (to allow log transformation)
PDFCT+	PDFCTRND+1 (ditto)
LD+	log of DEFECT+
LP+	log of PDFCT+
√DFCT	square root of DEFECT
√PRD	square root of PDFCTRND
LDFCT	log defect

Table 4.10: Definitions of variables

4.9.2.1 Comparing LOC and PREDLOC

The correlation tests in table 4.11 all reveal a high correlation between values of LOC and PREDLOC. Concentrating on Spearman as a robust test for a skewed distribution (we know LOC is skewed and can see the same skew for PREDLOC in Figure 4.7), we see a correlation of 0.968. Since we know already that there is strong evidence of a relationship between the two, from the high adjusted R^2 value (96.6) shown in table 4.9, we can say that this high correlation indicates an accurate prediction system for this data. This is quite striking when represented as a scatterplot with regression line in figure 4.8. Most points are on or touching the regression line, the rest are very close.

Correlation Test:	*Pearson*	*Spearman*	*Kendall*
LOC/PREDLOC	0.983	0.968	0.887

Table 4.11: Correlations for LOC/PREDLOC

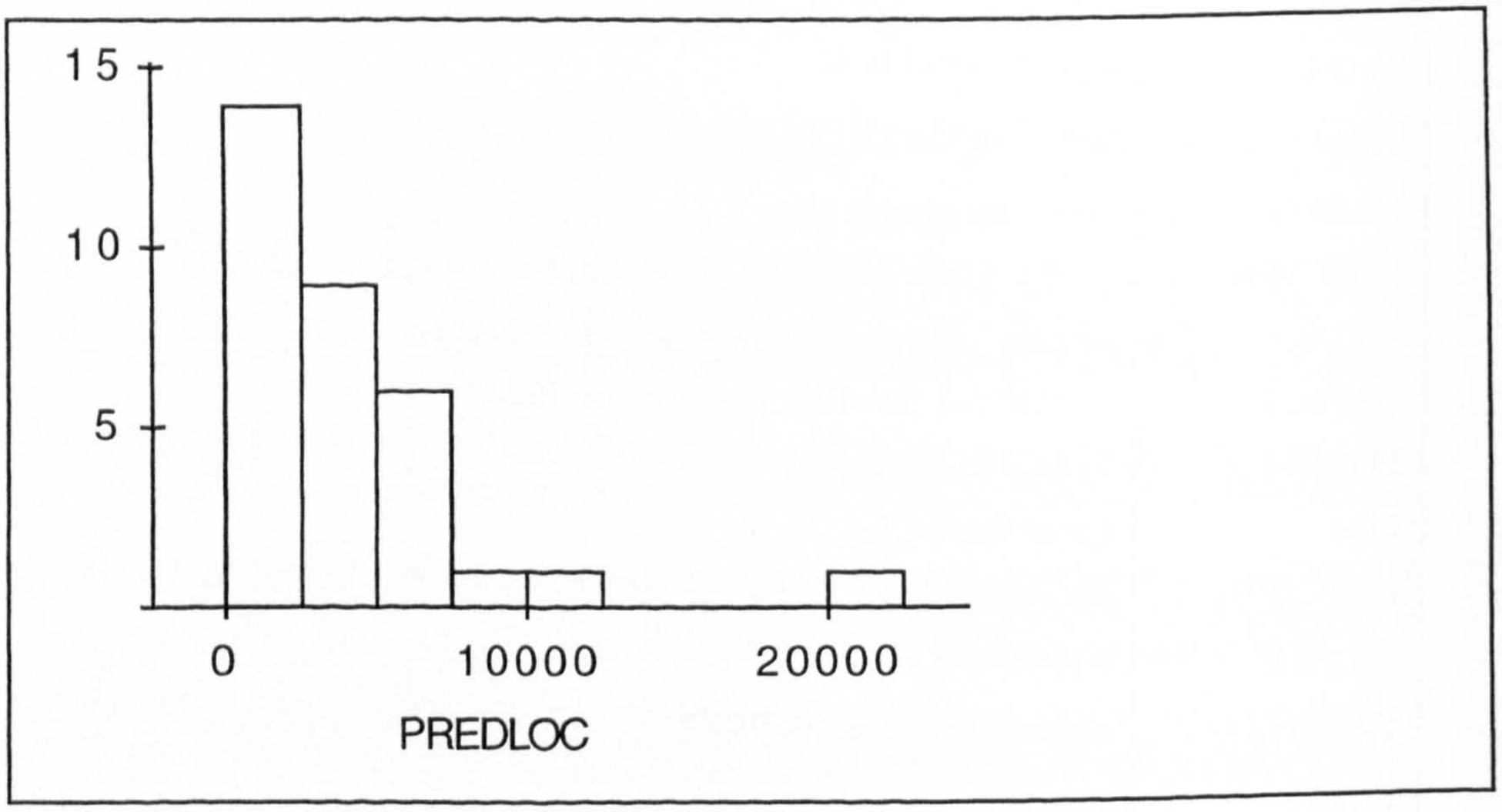

Figure 4.7: Histogram of PREDLOC

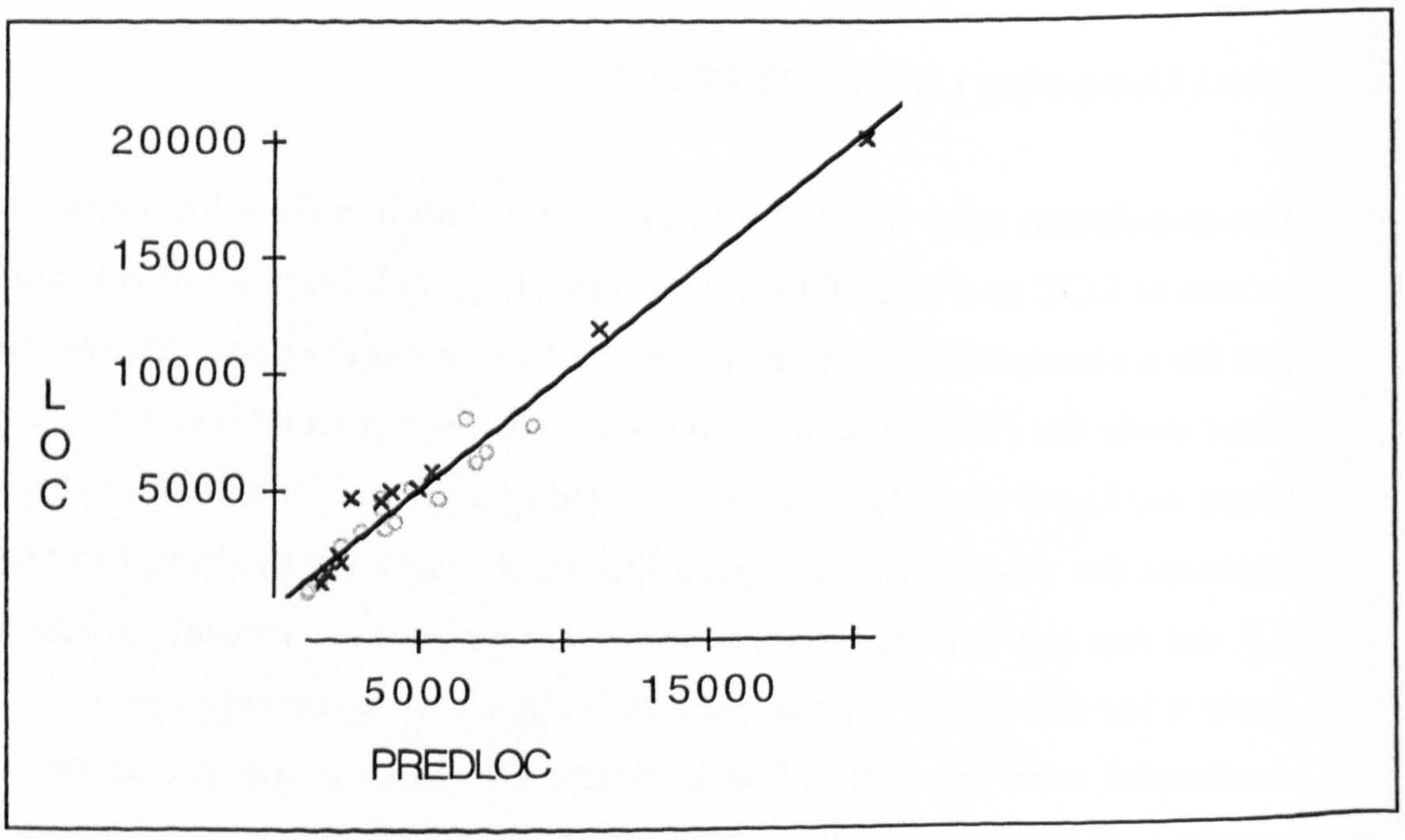

Figure 4.8: Scatterplot LOC/PREDLOC

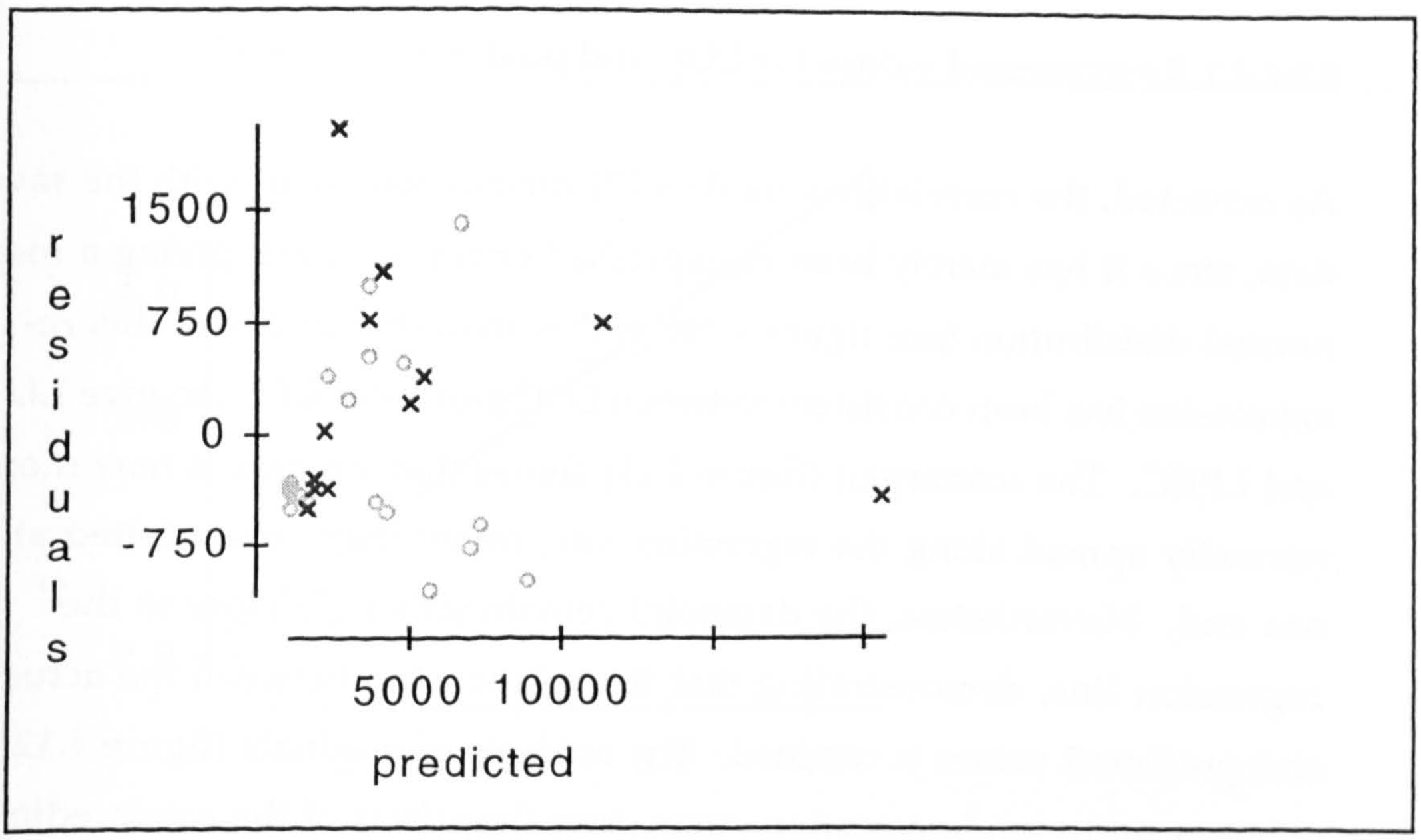

Figure 4.9: Residuals for LOC/PredLOC regression

Figure 4.9 shows that the model tends to perform less well for smaller classes. An explanation for this is the high value of the intercept, meaning that the prediction will never be less than 1101 LOC. There are a number of actual LOC values in the 700 LOC to 800 LOC range, these show as the clump of values at the -300/400 mark on the y-axis. A reasonable conclusion to be drawn is that classes with few or no states will also have smaller header files, thus the model will tend to over predict for such classes.

4.9.2.2 Comparing Re-expressed Values

Although the raw data provided sufficient indication of a relationship in itself, it is also interesting to look at the transformed data, since it will be used for hypothesis testing because the raw data is not amenable to a chi-square test (as explained at the start of 4.9)

4.9.2.2.1 Re-expressed values for LOC and predictions of LOC

As expected, the correlations (table 4.12) remain consistent with the raw data, since it has merely been re-expressed or transformed, giving a more normal distribution (see figure 4.10), rather than changed, and this re-expression has been consistent between LOC and PREDLOC, to give LLOC and LPRC. The scatterplot (figure 4.11) shows that the data is now more normally spread along the regression line, rather than concentrated at one end. Nevertheless, the datapoint remain strikingly close to the regression line, demonstrating that the relationship between the actual and predicted values is retained. The analysis of residuals (figure 4.12) shows a more random scatter, suggesting the effects of the overprediction for low values has less effect for the transformed values.

Correlation Test:	*Pearson*	*Spearman*	*Kendall*
LLOC/LPRC	0.973	0.968	0.887

Table 4.12: Correlations for LLOC/LPRC

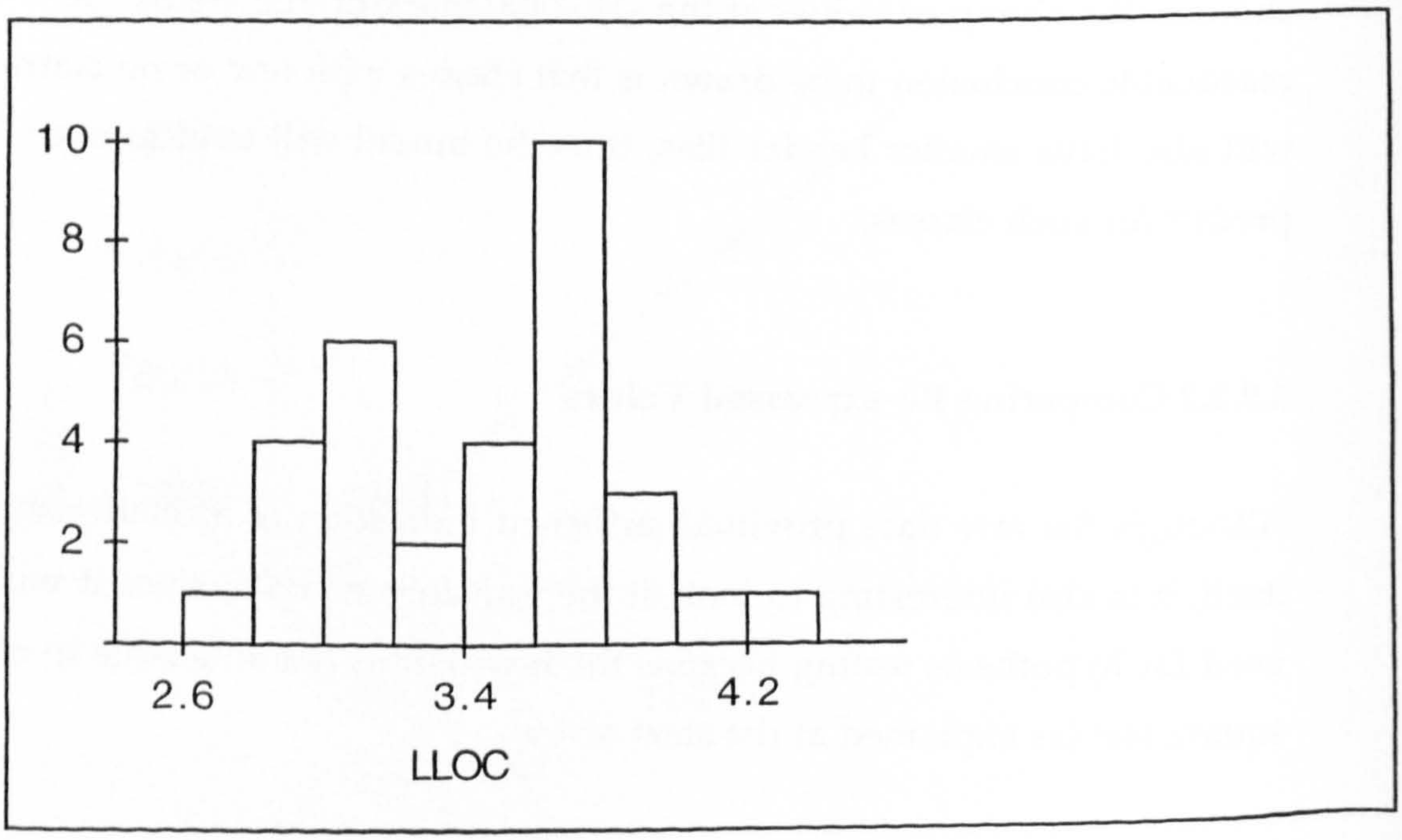

Figure 4.9: Histogram of LLOC

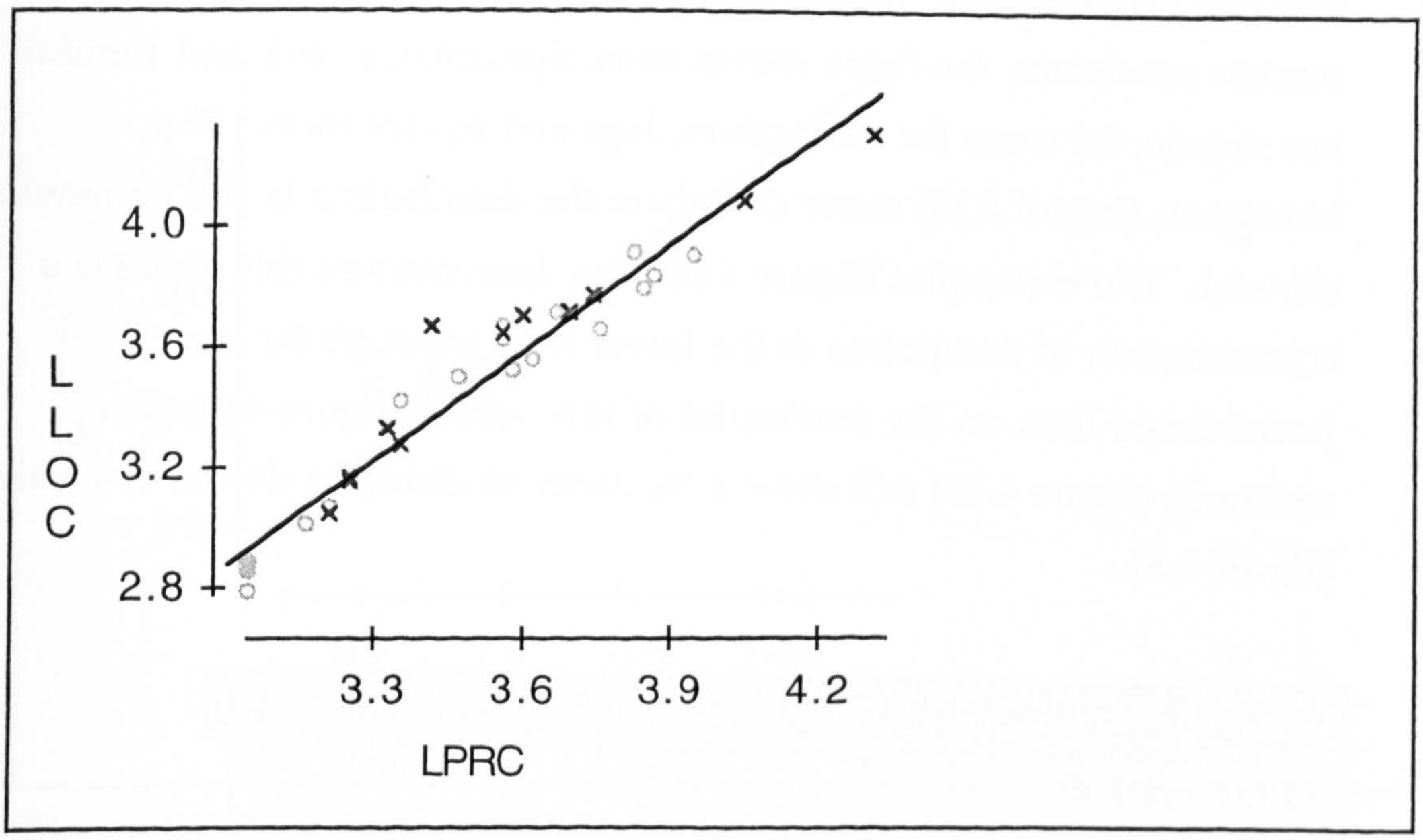

Figure 4.10: Scatterplot LLOC/LPRC

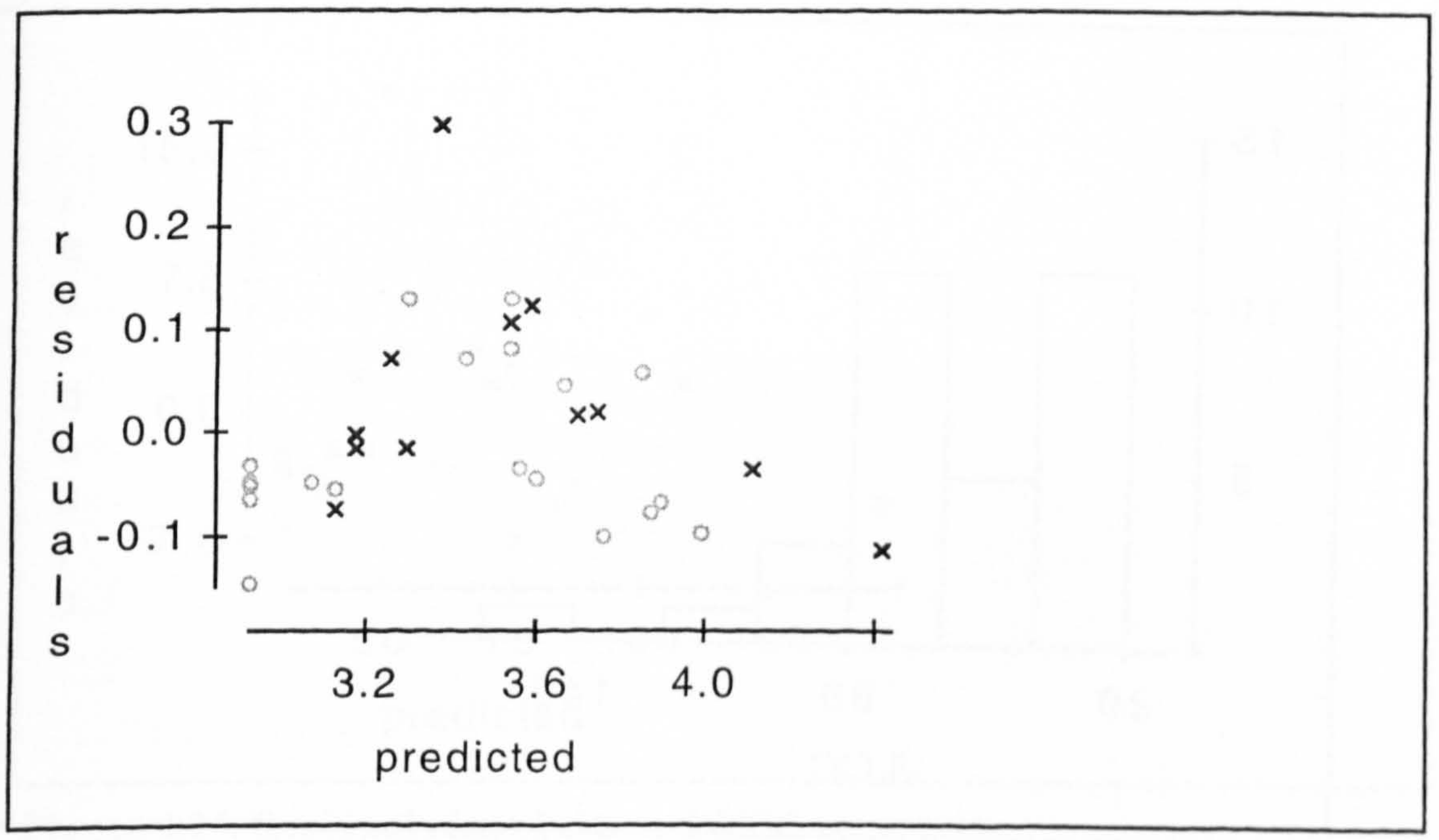

Figure 4.11: Residuals for LLOC/LPRC regression

It is also interesting to consider the square-root transformation, which Tukey described as a re-expression fitting halfway between raw values

and logs (Tukey 1977). Again, as expected, the correlations (table 4.13) remain consistent, the more robust tests, Spearman's rank and Kendall's tau remain the same for raw values, logs and square roots. The histogram (figure 4.13) show that there the distribution is still somewhat skewed. The scatterplot (figure 4.14) also demonstrates this; there is a concentration of datapoints at the lower end, although far less pronounced than on the scatterplot of raw values (figure 4.8). The residuals (figure 4.15) still show a tendency to clump as do the raw values (figure 4.9).

Correlation Test:	*Pearson*	*Spearman*	*Kendall*
√LOC/√PRC	0.976	0.968	0.887

Table 4.13: Correlations √LOC/√PRC

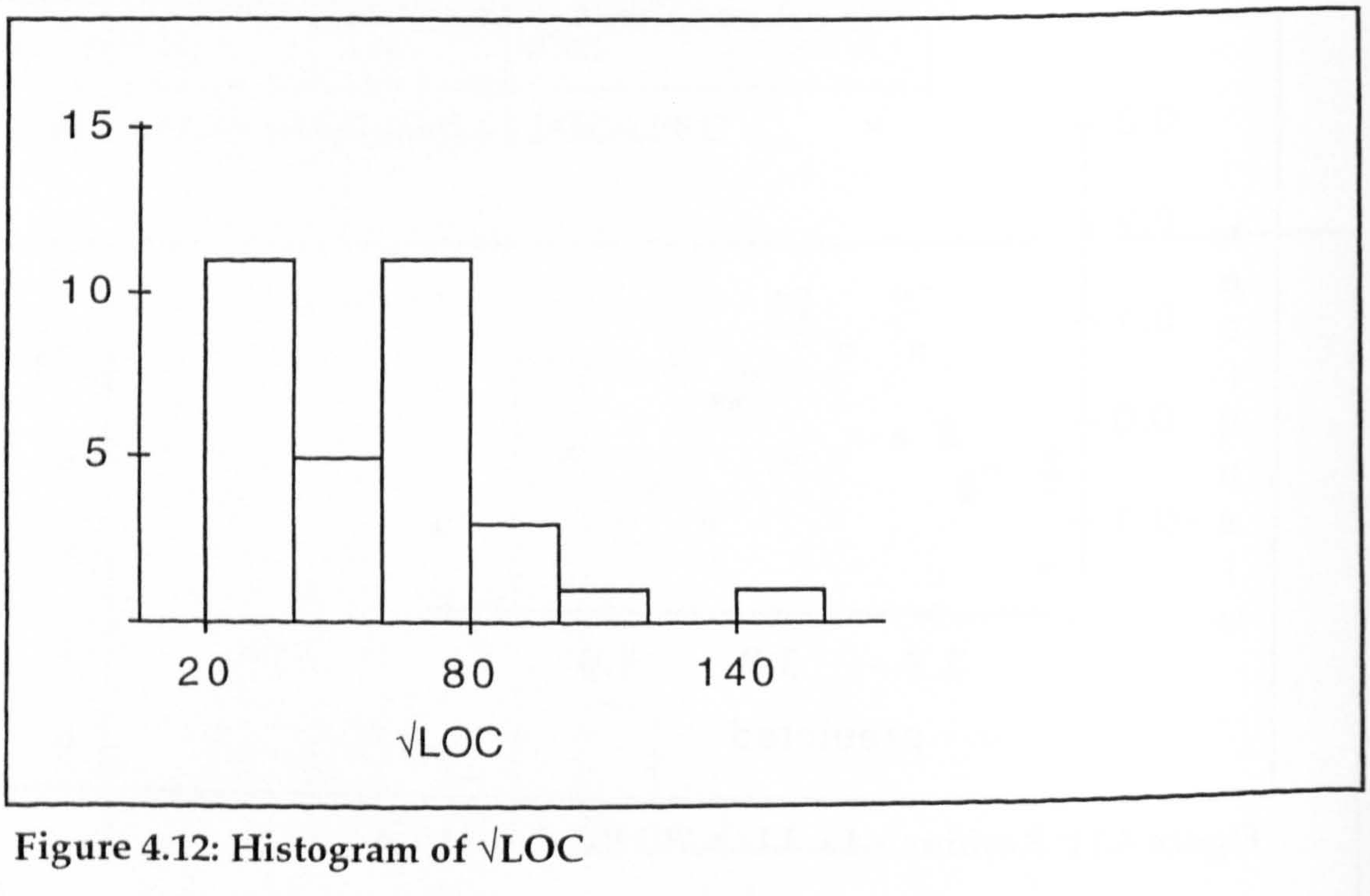

Figure 4.12: Histogram of √LOC

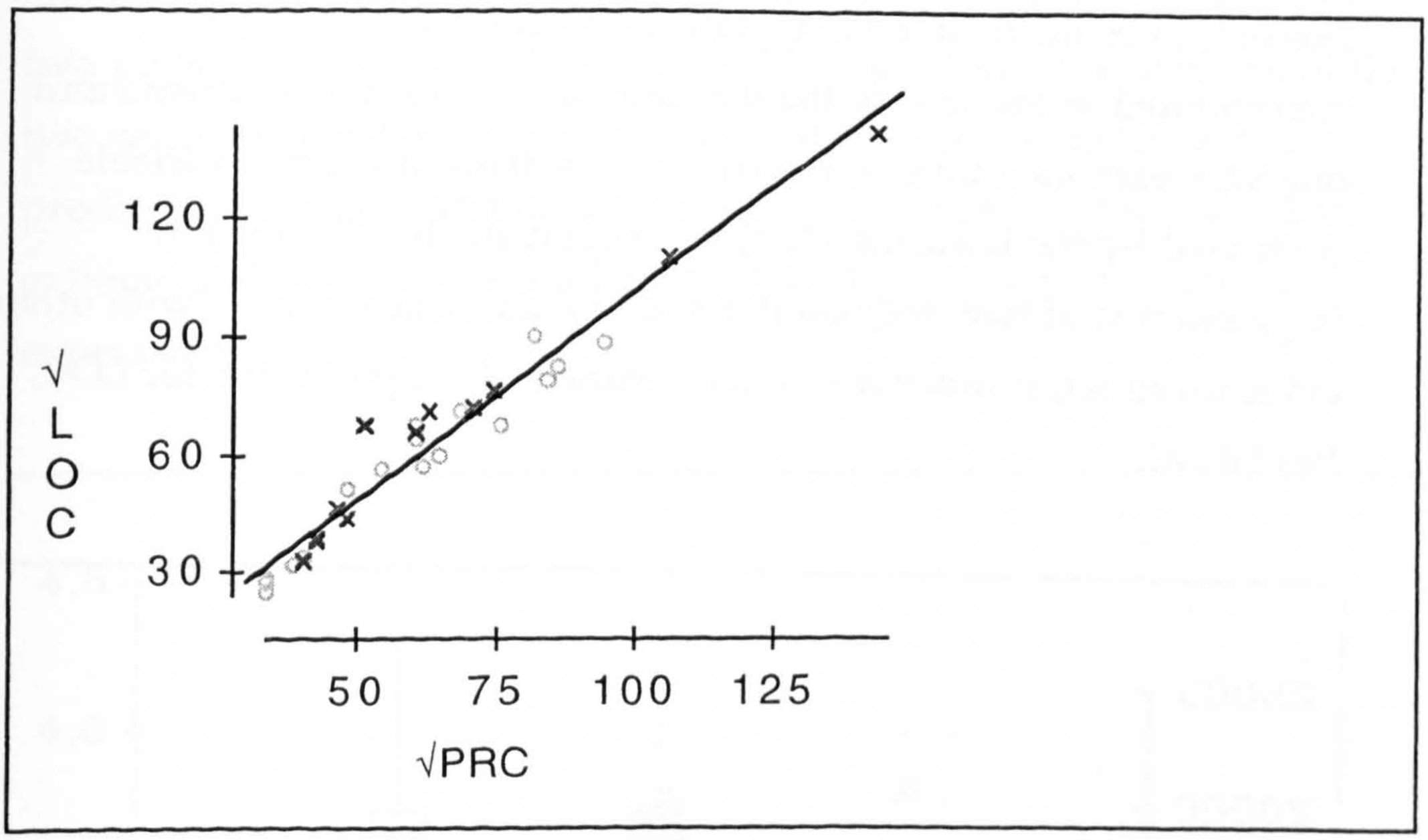

Figure 4.13: Scatterplot √LOC and√PRC

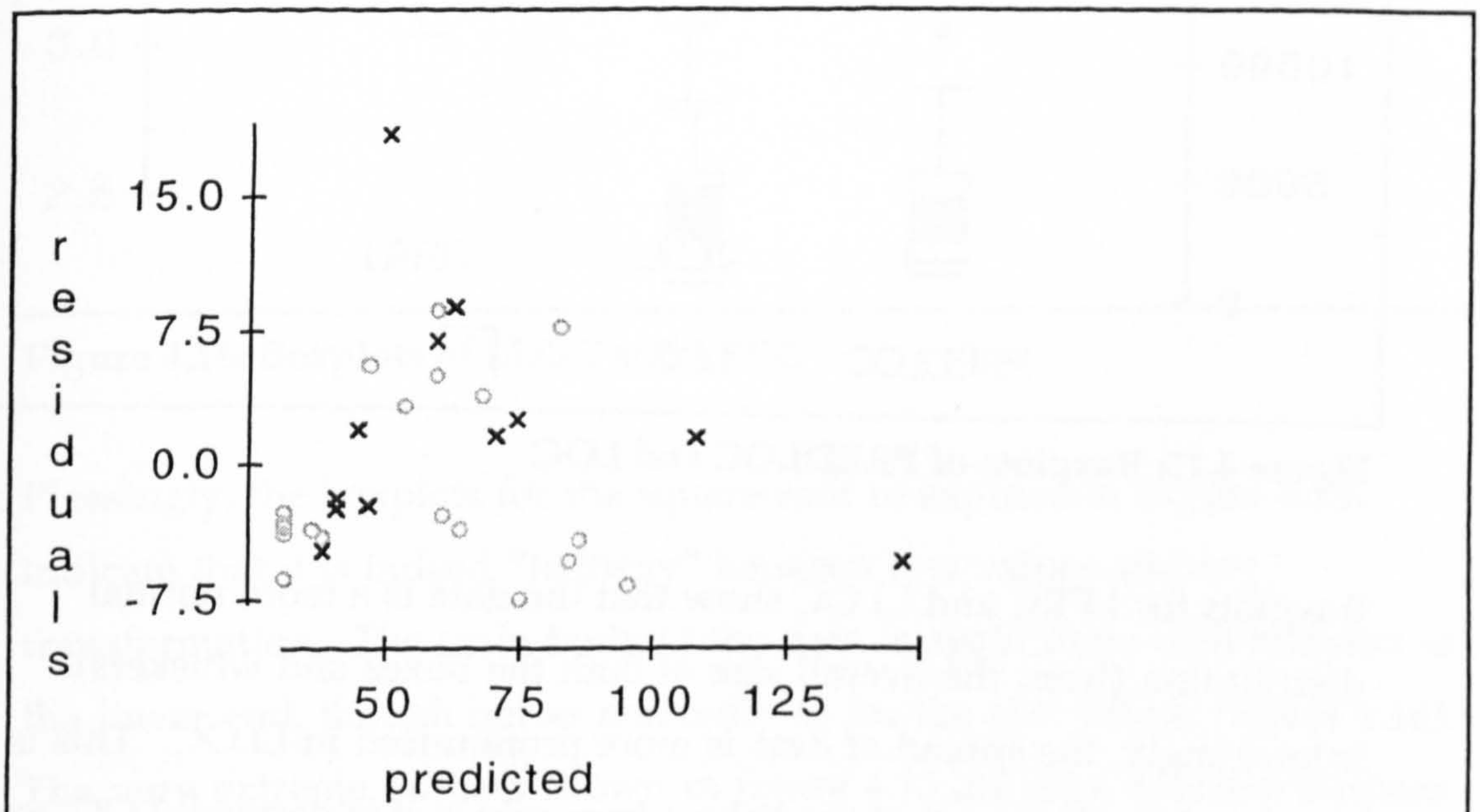

Figure 4.14: Residuals for √LOC and√PRC regression

Further evidence of the value of re-expression can be seen from the following boxplots. Figure 4.16 shows the distribution of the raw values, LOC and PREDLOC. It is obvious for both variables, that the data is

skewed, since the main body of data, represented by the box, is concentrated at one end of the distribution. There is one extreme and one very extreme outlier for each. The medians of the two variable (indicated by the horizontal bar) seem comparable. The overall impression is of two very similar sets of data, which in turn, with other evidence so far, is indicative of the accuracy of the prediction for LOC, PREDLOC.

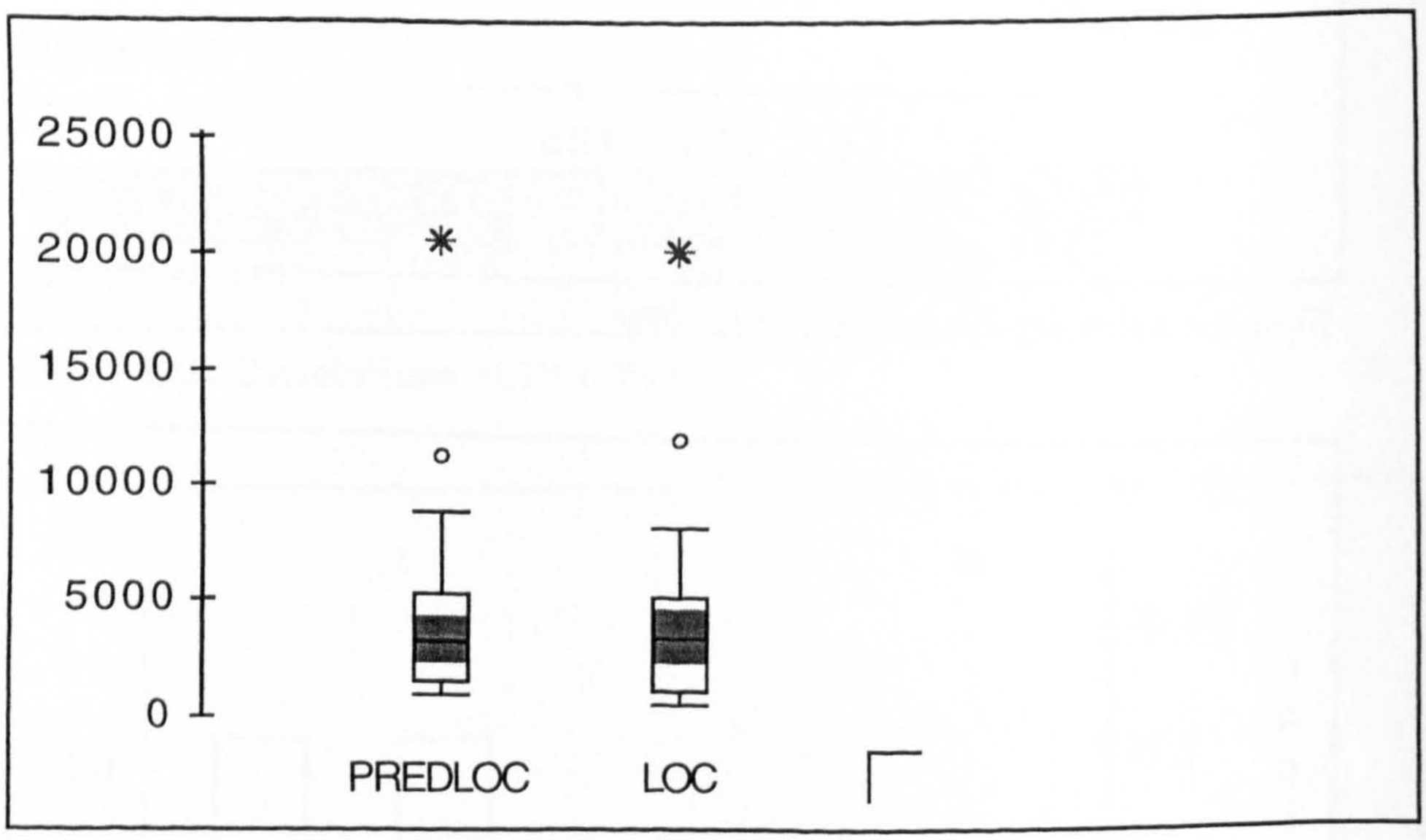

Figure 4.15: Boxplots of PREDLOC and LOC

Boxplots for LPRC and LLOC show that the data is a more normal distribution (from the overall size of both the boxes and whiskers). Interestingly, the spread of data is more pronounced in LLOC. This is explained by the fact that the values from which it is derived, LOC, are not constrained by a formula, unlike LPRC, derived from PREDLOC. The predicted values will never be smaller than the constant, due to the non zero intercept (actually 1101), this is shown on the plot by the short whisker on the boxplot for LPRC. This shows that the lower values are all higher than for LLOC. The actual values have no such constraints,

and so there is a potentially a wider distribution in values. Indeed the two plots are very similar from the middle up. Again the medians of the two seem comparable, which could indicate that the accuracy of the predictions is higher around the central values, than for outliers. The extreme and very extreme outliers are no longer apparent in this re-expression.

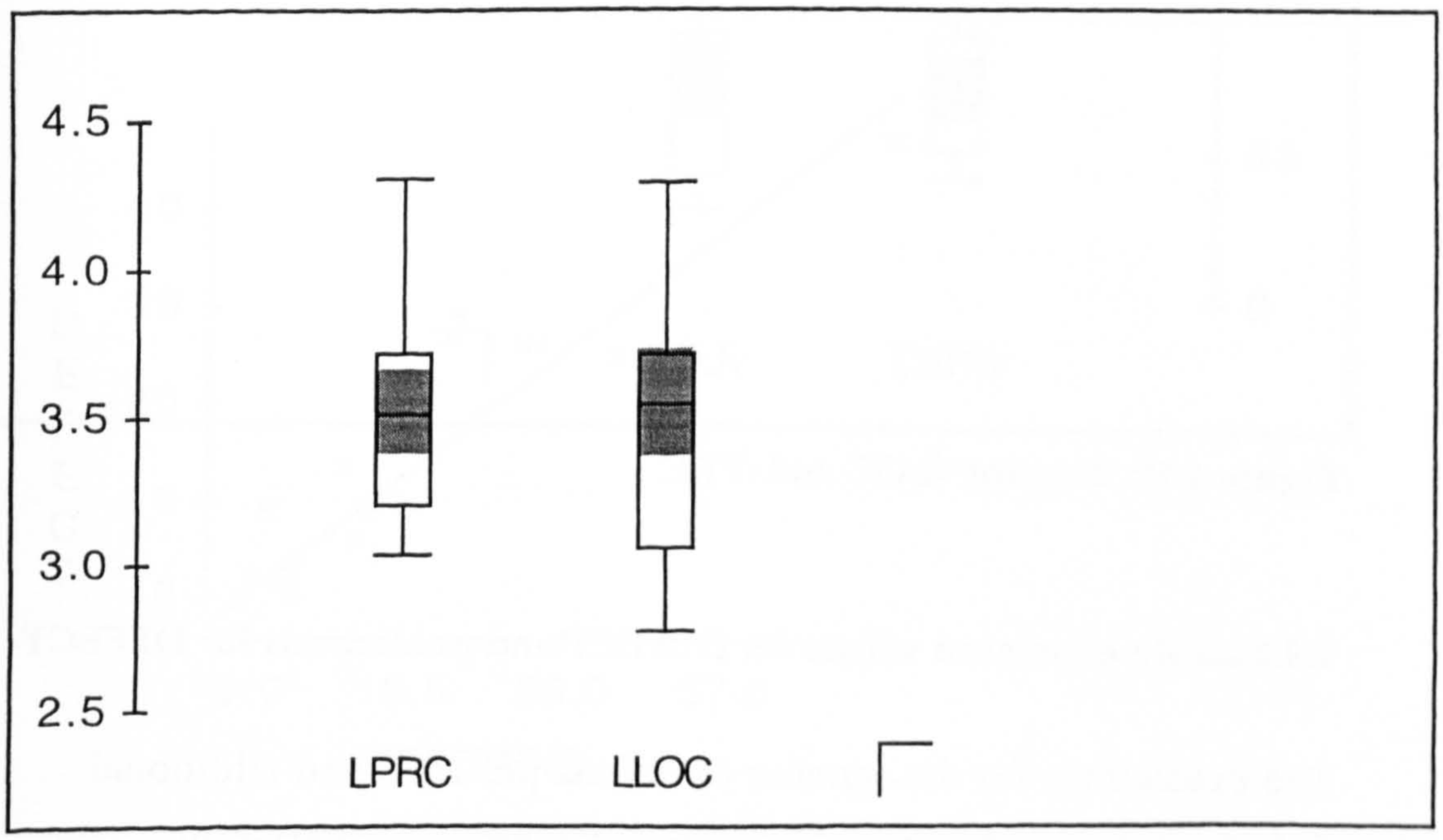

Figure 4.16: Boxplots of LLOC and LPRC

Pleasingly, the boxplots for the square-root re-expression (figure 4.18) indicate that it is indeed "halfway" between raw values and log transformation. The main body of the data is again more concentrated at the lower end, though not as markedly as for the raw values (figure 4.16). The very extreme outliers shown in figure 4.13 are now extreme outliers (which as stated above, are not outliers in the log re-expression shown in figure 4.17). Again, the position of the medians of each variable is comparable, and the naturally occurring variable (√LOC) shows a wider distribution in values)

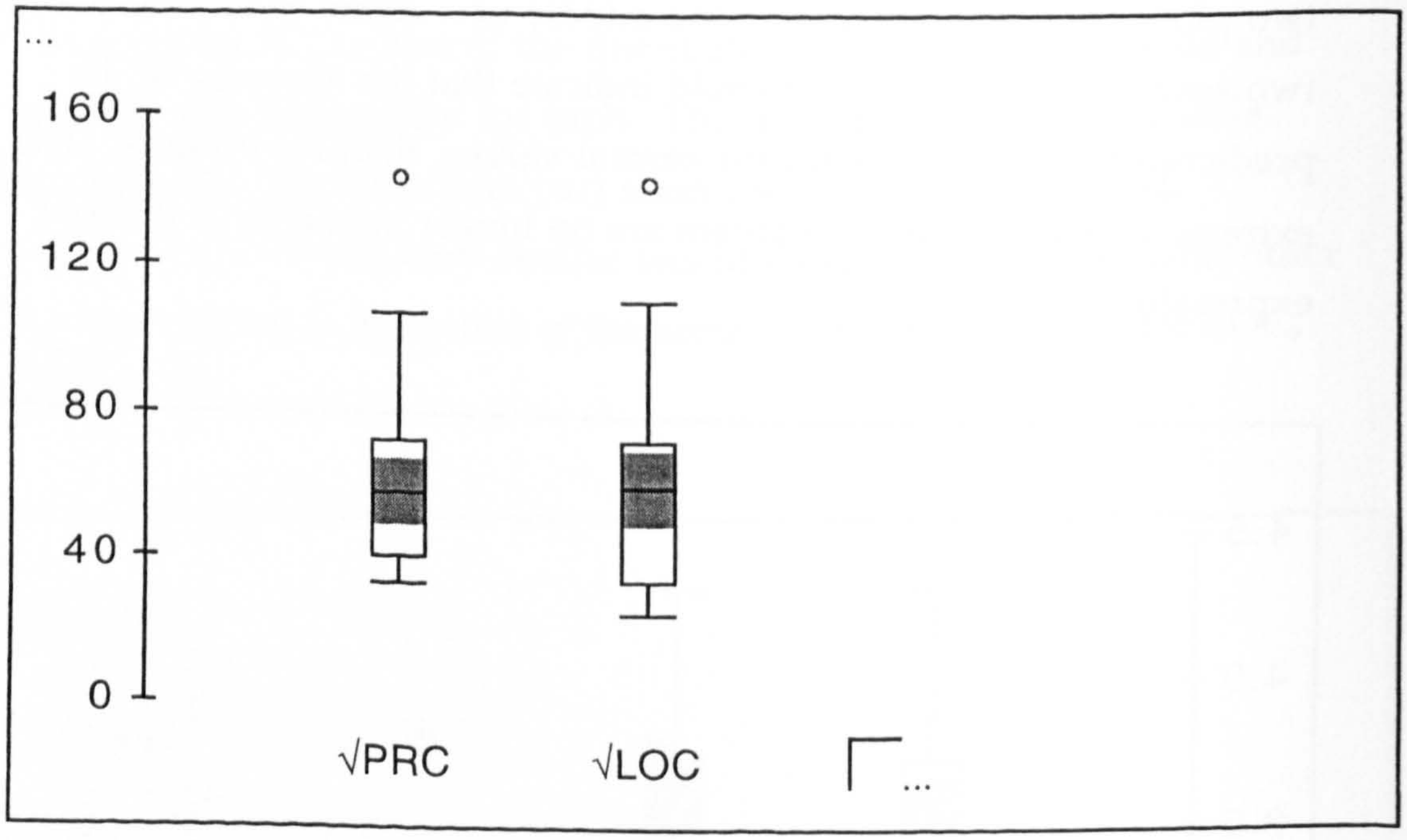

Figure 4.17: Boxplot √LOC and√PRC

4.9.2.2.2 Re-expressed values for DEFECT and predictions for DEFECT

The predictions for the number of defects per class had additional transformations prior to re-expression by log and square-root, since values had to be rounded to the nearest integer, negative values re-expressed as zero and the data was "started" ready for the log transformation by adding 1 (to avoid taking log of 0, this being undefined).

Therefore raw values of DEFECT were compared with rounded and non-negative values for the prediction of number of defects, terms PDFCTRND. Although less striking than correlations for LOC and related values in the previous section, the correlations are still significant, so that when taken with the high adjusted R^2 value shown in table 4.8, there is an indication that the two are related. This is also shown in the scatterplot, figure 4.19. The residuals for the raw data

(figure 4.20) show a reasonable degree of scatter, although there is a slight tendency to be more crowded around the lower end.

Correlation Test:	*Pearson*	*Spearman*	*Kendall*
DEFECT/PDFCTRND	0.938	0.860	0.742

Table 4.14: Correlations DEFECT/PDFCTRND

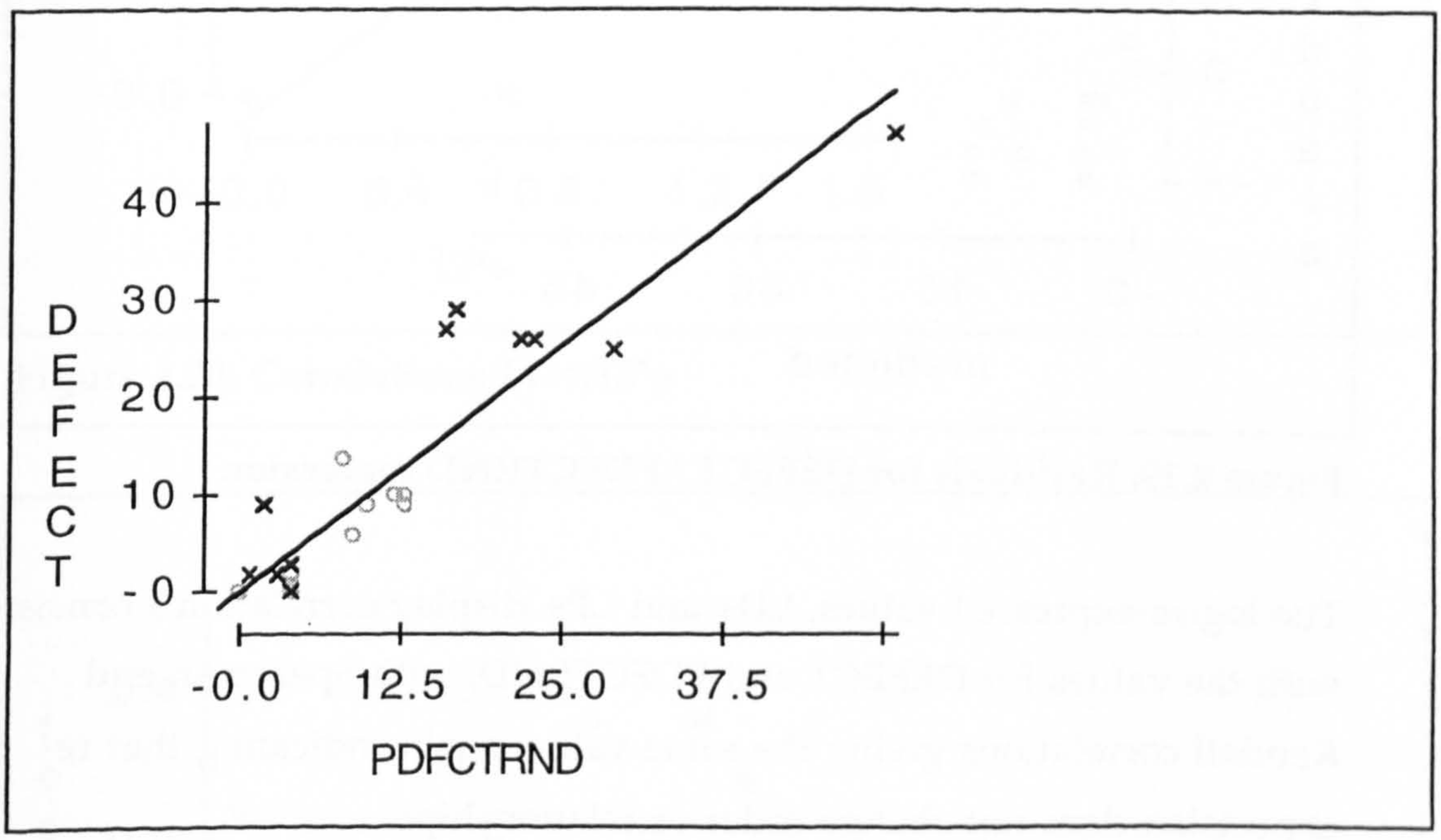

Figure 4.18: Scatterplot of DEFECT and PDFCTRND

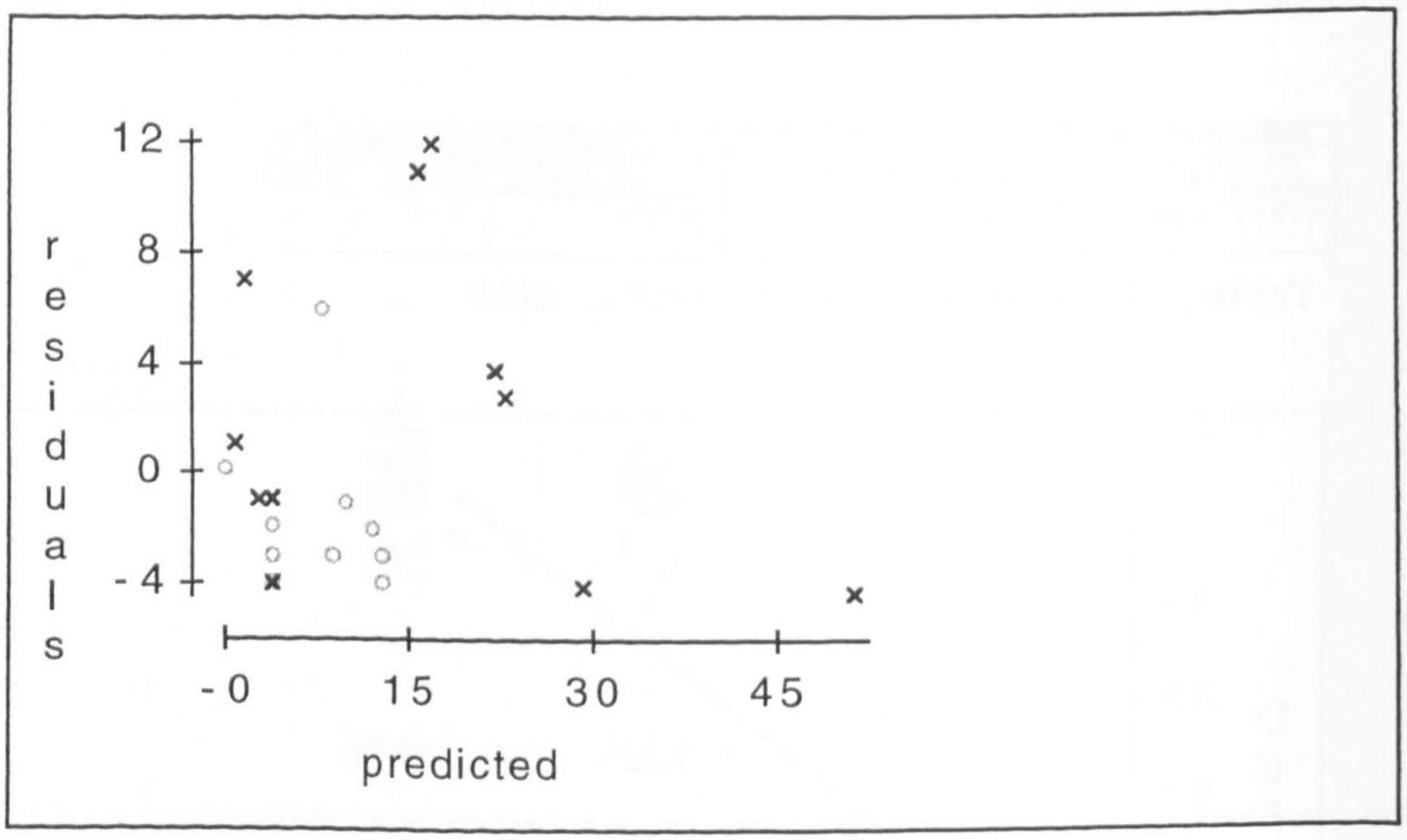

Figure 4.19: Residuals for DEFECT / PDFCTRND regression

The log-re-expressed values, LD+ and LP+ display correlation's consistent with the values for DEFECT and PDFCTRND, with Spearman and Kendall correlations giving the same value, again indicating that re-expression does not change order or relationships.

Correlation Test:	*Pearson*	*Spearman*	*Kendall*
LD+/LP+	0.878	0.860	0.742

Table 4.15: Correlations LD+/LP+

The scatterplot shows a more normal distribution with a less obvious regression line. This plot also demonstrates the higher incidence of defects in inheritance classes (as indicated by the "x") since most fall above the regression line. The residuals (figure 4.22) are far more randomly scattered.

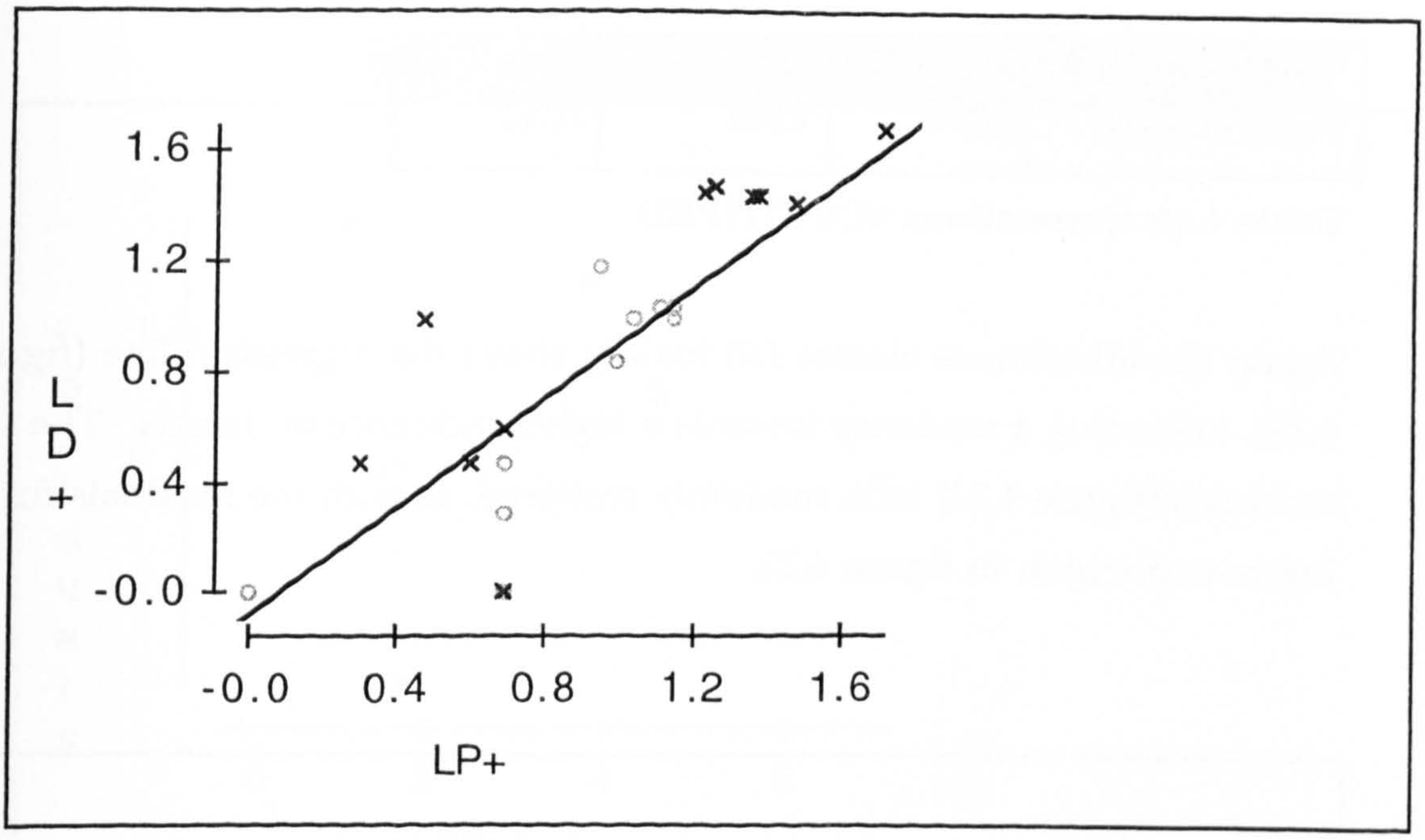

Figure 4.20: Correlations LD+/LP+

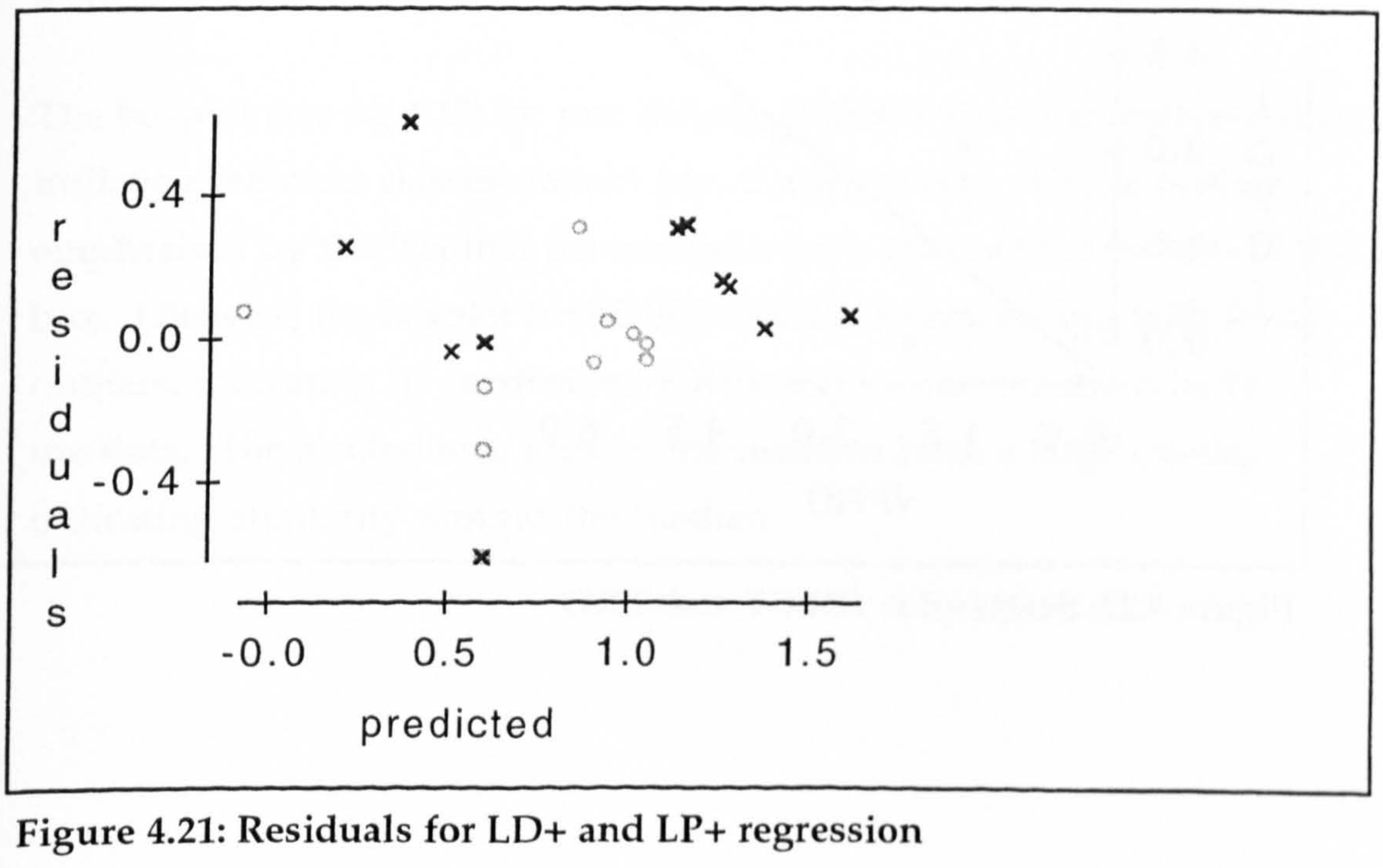

Figure 4.21: Residuals for LD+ and LP+ regression

Again for square-root re-expression the correlation remains consistent with those for the log re-expressed values (see table 4.15).

Correlation Test:	*Pearson*	*Spearman*	*Kendall*
√DFCT/√PRD	0.878	0.860	0.742

Table 4.16: Correlations √DFCT/√PRD

Again the inheritance classes fall mainly above the regression line (figure 4.23), indicating a tendency towards a higher incidence of defects. The residuals (figure 4.24) look randomly scattered, as with the residuals for log-re-expression in figure 4.22.

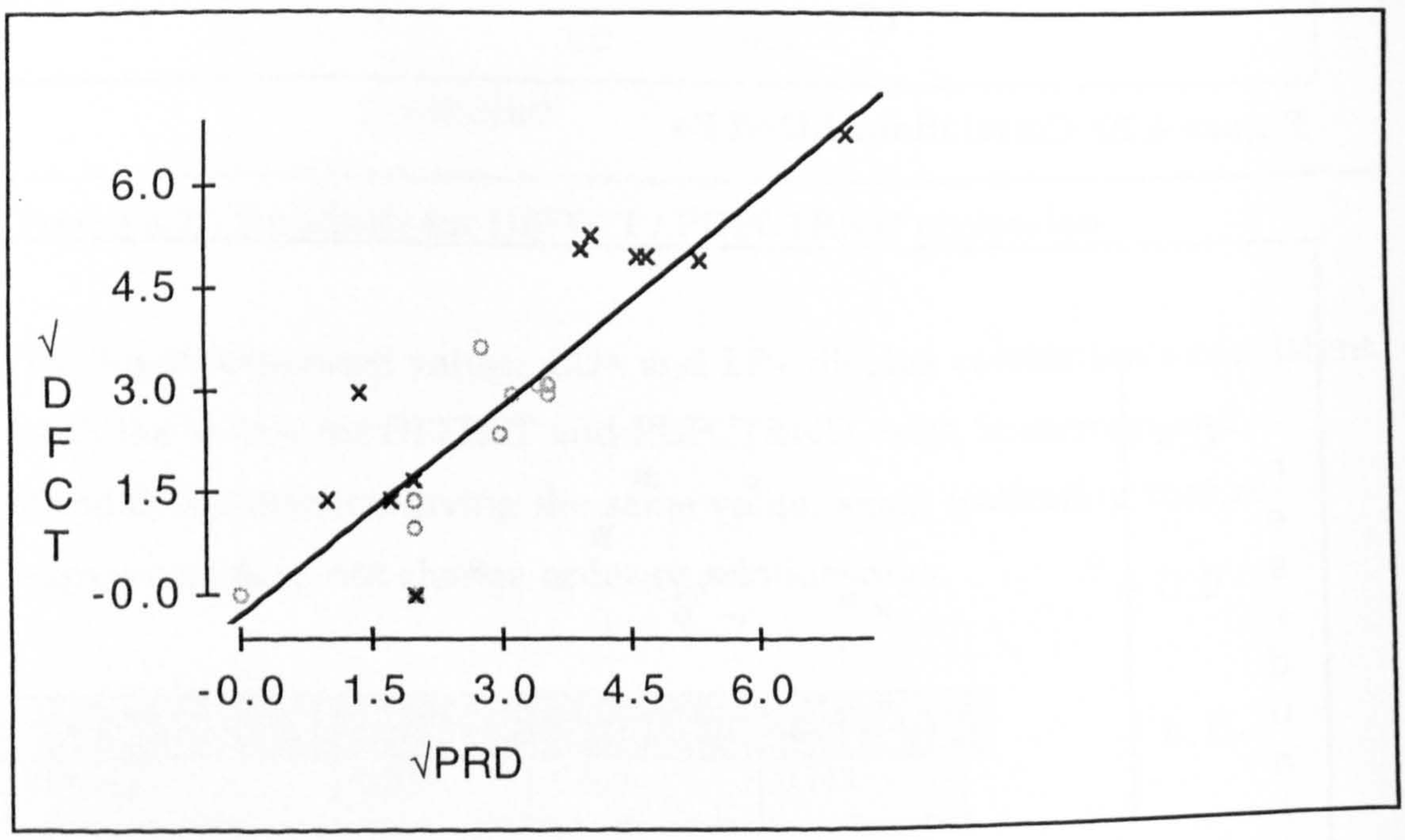

Figure 4.22: Scatterplot √DFCT and√PRD

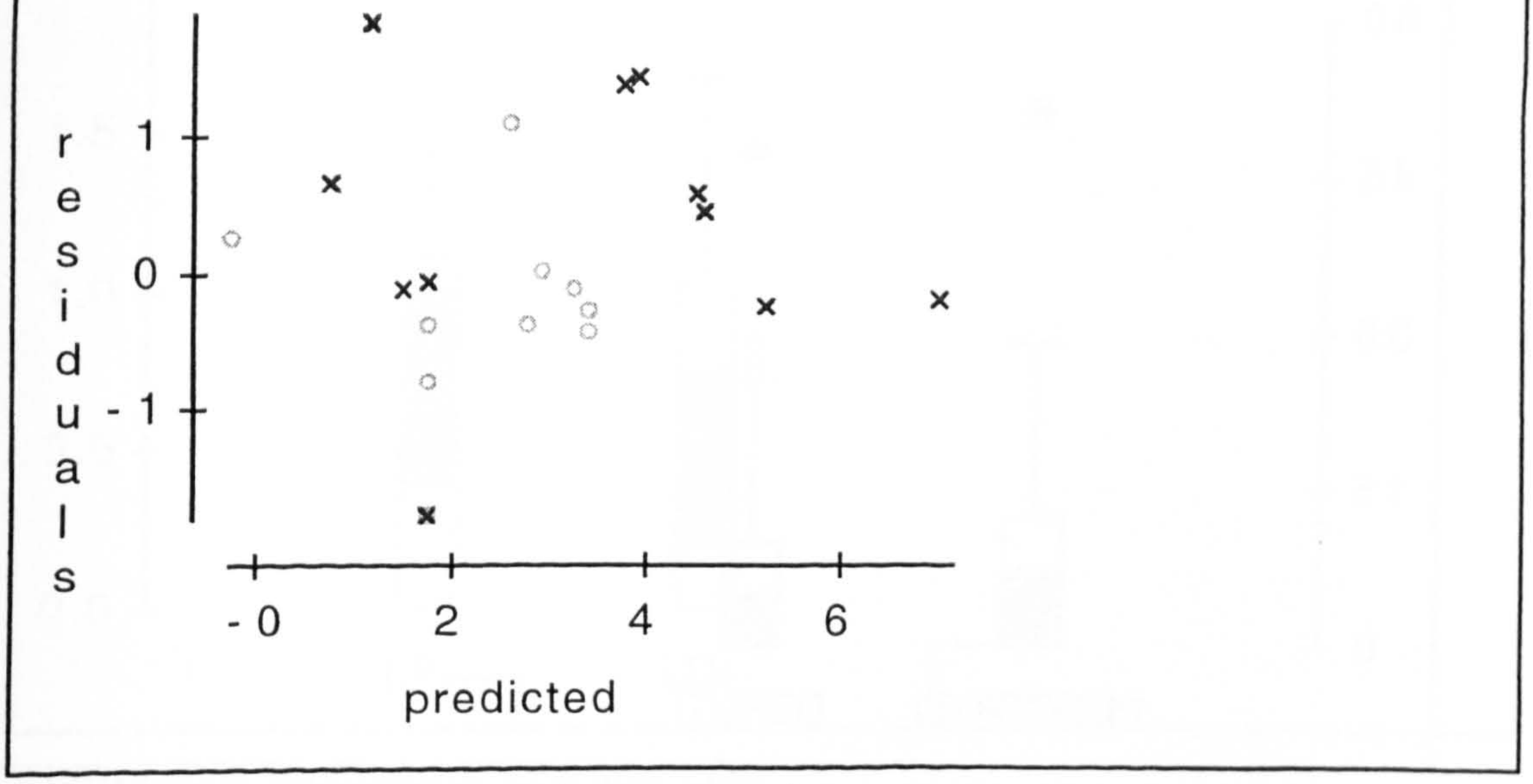

Figure 4.23: Residuals for √DFCT and√PRD regression

The boxplot (see fig 4.25) for raw defects, DEFECT shows an extreme skew, indicting the most classes contain few, if any, defects. This is further emphasised by the fact that the median occurs towards the bottom of the box. Likewise, the boxplot for PDEFECTRND is skewed, but with fewer outliers, indicating its predictions fall mainly within the main body of the data. The shaded area around the medians have a large overlap, indicating similarity around the median.

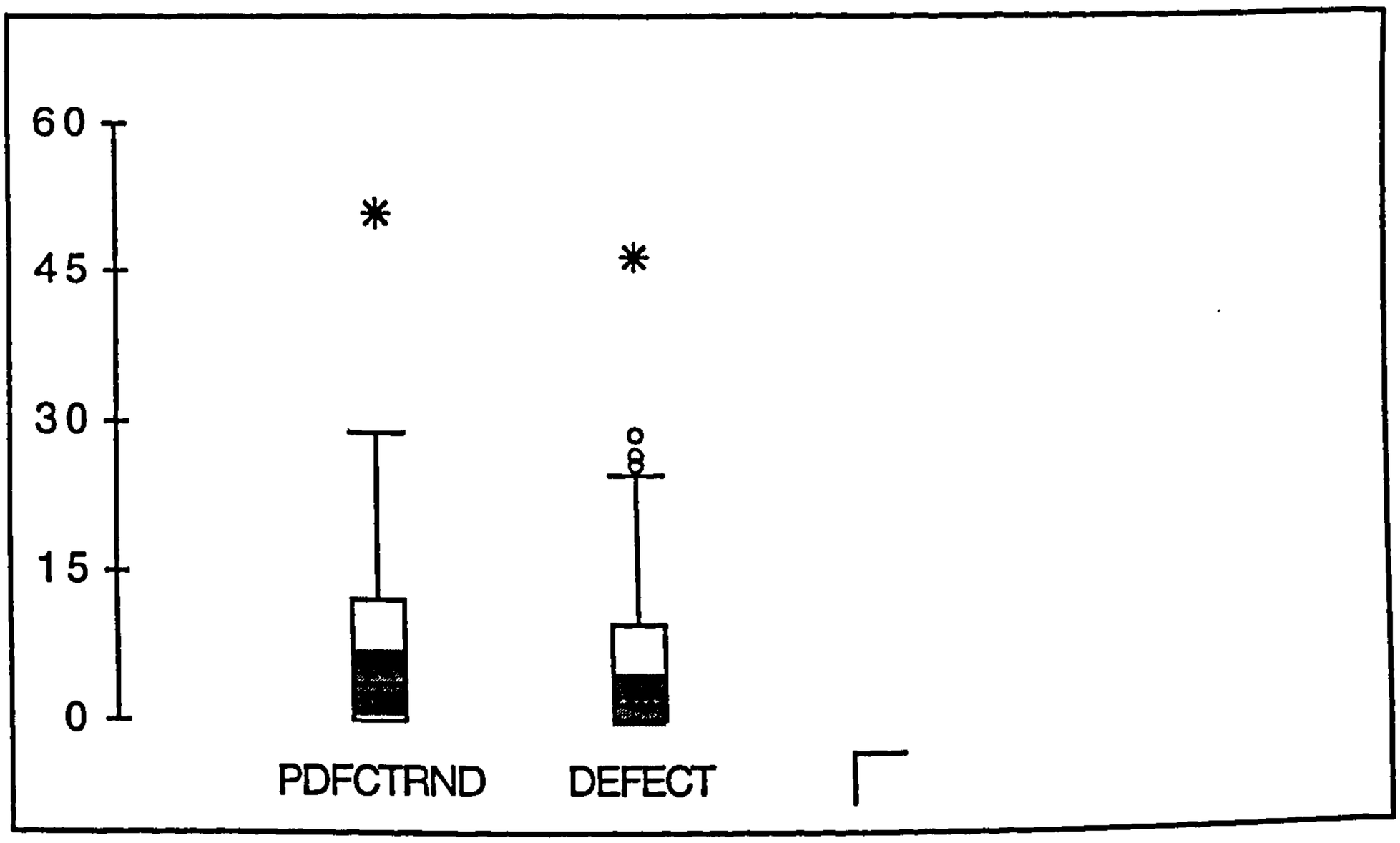

Figure 4.24: Boxplot PDEFECTRND and DEFECT

The boxplots for log re-expressed data (figure 4.26) and (figure 4.27) for square-root re-expressed data are more comparable. Firstly the distributions are more normal, with no outliers, and the boxes and whiskers of a similar size. Again the medians on each plot overlap, indicating a degree of similarity.

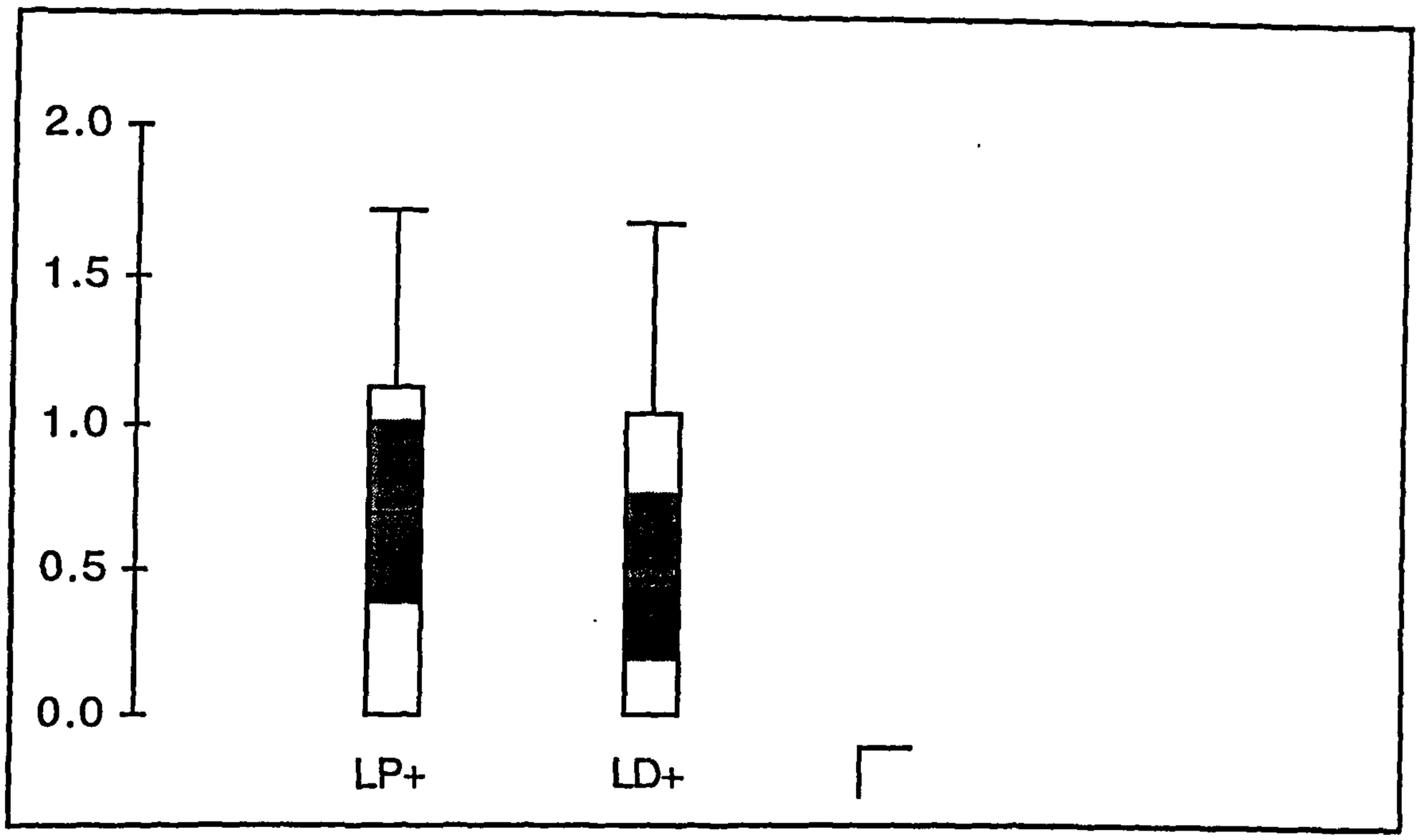

Figure 4.25: Boxplots of LP+ and LD+

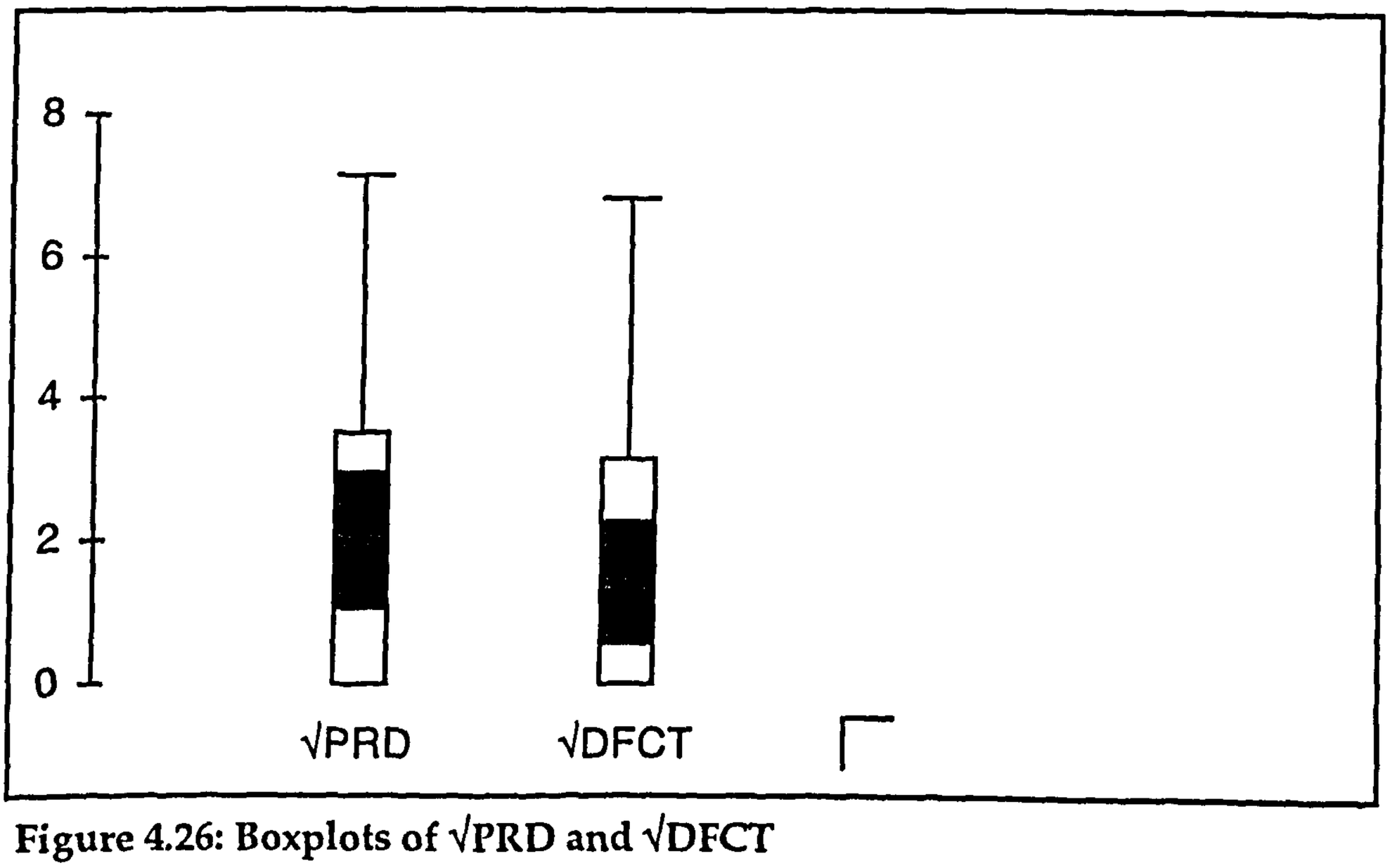

Figure 4.26: Boxplots of √PRD and √DFCT

4.9.2.2.3 General Conclusions on the Re-expression and Comparison of Actual and Predicted Data

The above plots and correlations provide sufficient evidence to suggest that the re-expressions have not affected relationships between data items whilst giving a more normal distribution. Thus hypothesis testing using the re-expressed values will be valid, since the order and relationships between data have not been changed.

4.9.3 Hypotheses and Hypothesis Testing

From the exploratory analysis and comparisons of the actual and predicted data there arose the following hypotheses:

The prediction system for DEFECTS gives an accurate prediction of the number of defects contained in a class;

The prediction system for LOC gives an accurate prediction of the number of lines of code contained in a class.

4.9.3.1 Chi -square

Although Chi-square proved unsuitable for use on the raw data it was possible to apply the test to the data when re-expressed as logarithms.

Chi-square for goodness of fit can be used to test whether the prediction (expected value) is a good predictor of the actual (observed) value by determining whether the two factors are independent of each other or are dependent.

For the hypotheses formulated above we wish the test to show that the two are dependent. The results from the Data Desk[44] contingency tables are based on a null hypothesis that the two factors are statistically independent and the alternative hypothesis of dependence. In both tables it can be seen that there is a probability <= 1 in 1000 of independence, i.e. in both cases, the factors are statistically dependent.

Each table shows the number of occurrences in each cell (from the data) and the expected value for that cell (if the null hypothesis were true). If the null hypothesis is true then the expected values will approximate the actual values, and conversely if the null hypothesis is false, the two will tend to differ. As can be seen from the tables (4.17 and 4.18) both show an overall tendency to differ.

The χ^2 value for LOC is 81.88. The probability of gaining a χ^2 value this high is less than 1:10000, so the null hypothesis can be rejected.

For defects the χ^2 value is 54.86. Again the probability of obtaining a value of this size is less than 1:10000, so the null hypothesis can be rejected.

It can be concluded that the models make reasonable predictions for both LOC and DEFECT.

---

[44] The stats package used.

Rows are levels of **actual LOC**
Columns are levels of **predicted LOC**
No Selector

	2.70-3.10	3.10-3.40	3.40-3.70	3.70-4.10	4.10-4.40	total
2.70-3.10	5	4	0	0	0	9
	1.40625	2.53125	2.53125	2.25000	0.281250	9
	3.03052	0.923167	-1.59099	-1.50000	-0.530330	0
3.10-3.40	0	4	0	0	0	4
	0.625000	1.12500	1.12500	1	0.125000	4
	-0.790569	2.71058	-1.06066	-1	-0.353553	0
3.40-3.70	0	1	7	1	0	9
	1.40625	2.53125	2.53125	2.25000	0.281250	9
	-1.18585	-0.962451	2.80879	-0.833333	-0.530330	0
3.70-4.10	0	0	2	7	0	9
	1.40625	2.53125	2.53125	2.25000	0.281250	9
	-1.18585	-1.59099	-0.333912	3.16667	-0.530330	0
4.10-4.40	0	0	0	0	1	1
	0.156250	0.281250	0.281250	0.250000	0.031250	1
	-0.395285	-0.530330	-0.530330	-0.500000	5.48008	0
total	5	9	9	8	1	32
	5	9	9	8	1	32
	0	0	0	0	0	0

table contents:
Count
Expected Values
Standardized Residuals

Chi-square = 81.88 with 16 df
$p \leq 0.0001$

Table 4.17: Chi-square test actual LOC and predicted LOC (re-expressed as logs)

Rows are levels of actualDefect Columns are levels of expected Defect No Selector					
	0-0.5	**0.5-1**	**1-1.5**	**1.5-2**	**total**
0-0.5	10	8	0	0	18
	6.18750	5.62500	5.62500	0.562500	18
	1.53268	1.00139	-2.37171	-0.750000	0
0.5-1	0	1	1	0	2
	0.687500	0.625000	0.625000	0.062500	2
	-0.829156	0.474342	0.474342	-0.250000	0
1-1.5	1	1	9	0	11
	3.78125	3.43750	3.43750	0.343750	11
	-1.43028	-1.31469	3.00019	-0.586302	0
1.5-2	0	0	0	1	1
	0.343750	0.312500	0.312500	0.031250	1
	-0.586302	-0.559017	-0.559017	5.48008	0
total	11	10	10	1	32
	11	10	10	1	32
	0	0	0	0	0

table contents:
Count
Expected Values
Standardized Residuals

Chi-square = 54.86 with 9 df
$p \leq 0.0001$

Table 4.18: Chi-square test actual defect and predicted defect (re-expressed as logs)

4.10Conclusions

This case study of a large industrial C++ system has demonstrated a number of points.

• First, that it not necessarily straightforward to apply pre-defined metrics. There seems still a tendency for metrics to be complex, vaguely defined and/or method independent. This is shown by the lack of success in applying the CK metrics. Although among the most mature and well defined of the metrics on offer, it still proved impossible to collect the majority of them from the design documentation available. A number of static code analysers now implement the CK metrics, but since they were intended as design metrics, this detracts from their potential usefulness.

• Second, that inheritance is less used than might be expected from the prominence given to the mechanism in OO textbooks. This was confirmed by examining the design documentation and from anecdotal evidence from the developers.

• Third, that inheritance has an effect on the number of defects in a class. Classes in an inheritance hierarchy had three times the defect density of classes not part of an inheritance hierarchy. Tests also confirmed the inheritance classes were a distinct sub-population of the dataset. Defects were also size driven, hence the use of defect densities. This seems to confirm anecdotal evidence that developers avoided the use of inheritance because they found it difficult to understand.

• Fourth, deriving locally applicable metrics from local data is not a difficult task, providing suitable tool support is available. This of course assumes that data, such as incident reports, change requests, developmental effort etc. is collected as a matter of course. In this case study, readily available design data was correlated against the dependent variables of interest, SIZE and DEFECT. The most promising of these were used to derive regression equations, of which the most simple with

high adjusted R^2 values were selected. In order to demonstrate the predictions made were reasonable, a chi-square test was carried out, which demonstrated that the values were dependent.

As discussed in chapter 1, the results of a single case study by itself are not generalizable. It has however, added to the empirical evidence on "real" object-oriented systems, particularly regarding the use of inheritance. The findings linking the use of inheritance has also lead to the formation of a further hypothesis, that the use of inheritance will effect the effort required to maintain software, such that software using inheritance will take more effort to maintain than software which does not use inheritance. This hypothesis will be tested in chapter 5.

Blank

In

Original

Chapter 5 An Experiment into the Effects of Inheritance on Maintenance Changes

Synopsis

An experiment investigating the effects of inheritance on software maintenance activities was carried out by a final year undergraduate[45] using material designed at the University of Strathclyde and fellow undergraduates as subjects. The aim of the experiment was twofold. Firstly it offered a means of testing the hypothesis formulated in chapter 4, that the use of inheritance increased the effort needed to maintain the software as compared with classes which did not use inheritance. Second it offered a partial replication to the experiments carried out at Strathclyde and added to the available empirical evidence on the maintenance of object-oriented software. This chapter describes the experiment and analysis of the data collected. It found that at three levels of inheritance, the effort needed to carry out the maintenance change was more than that required for the equivalent flat structure.

5.1 Reasons for the experiment

The empirical case study described in chapter 4 indicated that inheritance was associated with a higher level of defects than classes which were not involved in an inheritance relationship. Further, in the case study described, the classes with the highest densities were found at the bottom of their respective inheritance hierarchies, which would confirm the perceptions of the developers concerning the difficulties involved in using inheritance[46], as well as publications on maintenance of object-

---

[45] The analysis presented here was carried out by the author. The student, Claire Joyce, was, of course, expected to carry out an independent analysis as part of her project. Happily our conclusions concur.

[46] Such as difficulty in understanding classes using inheritance, which obviously has implications for testing and maintenance

oriented software and inheritance, such as (Lejter, Meyers et al. 1992; Wilde, Matthews et al. 1993; Dvorak 1994). It was therefore interesting to repeat an experiment by(Daly, Brooks et al. 1996; Daly 1996) concluding that software with an inheritance hierarchy three levels deep seemed to be more maintainable than equivalent software not using inheritance, and that it was not until five levels of inheritance were used that this phenomenon was no longer apparent. Since this was inconsistent with the results of the case study, the developers experience and indeed much of the anecdotal evidence available, it was decided to investigate further by replicating the experiment in part. The original work consisted of two systems, using three levels, each compared against a 'flat' (i.e. no use of inheritance) equivalent. This was followed by a further experiment using an extended version of one of the systems, using five levels of inheritance, again with an equivalent flat version. This experiment would use one of the three-level systems. A full replication was not feasible since the experiment would rely on volunteers and thus needed to minimise the time commitment made. The subjects of the Strathclyde experiment were under assessment, and so were compelled to take part.

5.2. Differences from the original experiment

It was decided to conduct the experiment following the original as far as practicable . The experiment was invigilated by and data collected by a final year undergraduate on fellow software engineering students. Although intended as an external replication of the work at Strathclyde, there were inevitably some differences, although these were minimised as far as was possible. The following differences between this experiment and the original remained:

sample size - this experiment had ten subjects, the original had 30;

experience - all had some training in OO, although this is unlikely to be precisely the same as the training given to the Strathclyde students;
material used - this experiment used only the three level version;
time - this experiment extended the time to two hours from one and three quarter hours.
experimental design - this experiments required each group to carry out one set of changes on one treatment (inheritance or flat), Strathclyde required groups to carry out both treatments.

5.3. Description of the experiment

There follows a brief description of the experimental procedure, materials, subjects and the actual task to be completed.

5.3.1 Procedure

Each subject received a sheet of instructions regarding the experiment, and two packs, one containing the maintenance change and the other the source code listing. Subjects worked independently

Ten minutes were allowed for reading and clarifying the instructions. The subjects then proceeded to open the first pack, containing details of the maintenance change required and were given a further ten minutes for reading and clarification of the requirements. Once this was complete, the subjects opened the remaining pack, containing the source code listing. On opening this pack, the subjects could begin the task and timing began. Completion of the task was dependent upon successful compilation and testing against required output (using supplied data), which was checked by the invigilator. If this was confirmed as correct by the invigilator, timing was stopped, else the subject was asked to

continue. Once their task was complete, the subjects were asked to complete a debriefing questionnaire, a copy of which is included in appendix B.

5.3.2 Materials

Each subject was provided with:

- instruction sheet;
- packs containing maintenance task requirements and source code listing;
- test data;
- HP work station, with Emacs editor and C++ compiler;
- source code.

5.3.3 Subjects' background

All ten subjects had a minimum of 6 months experience of object-orientation using C++, all belonged to the same degree course and thus received the same introduction to OO and C++. Variations occurred where subjects had outside experience, such as from work placements during their sandwich year.

Subjects were randomly assigned into two groups; group A were given the inheritance version (with three levels) and group B the flat version.

5.3.4 Maintenance task

The program to be modified was a simple library database system, allowing the creation, display, modification and deletion of records. Three categories of record were supported, book, conference and thesis.

The software was designed "in an object-oriented fashion" (Daly 1996) and coded in C++ using single inheritance. The flat version was derived from the inheritance version by removing inheritance links and then adding the data and functions that would have been inherited directly to the relevant classes (thus repeating code in each class). Any abstract classes (i.e. those existing for the sole purpose of allowing classes to inherit from them) were then removed from the flat version.

The flat version consisted of approximately 440 lines of code and four classes (each class made up of a header and implementation or body file), whilst the inheritance version consisted of approximately 390 lines of code and six classes.

The task given to the subjects was to add a new class to the library system. PhD-Thesis. This consisted of seven different fields and was intended as a specialisation of the Thesis class. Subjects were to create an instance of the new class with initial and default values, modify some values then display the results.

5.4 Data Collection

Data collection was automated using a shell script designed for the experiment. This was initiated at login and kept running throughout the experiment to record the process of modification. To recap; the data collected from the experiment was: time to complete, the final version of

the solution (to allow the number of LOC added to be calculated) and the completed debriefing questionnaire.

5.5 Preliminary Analysis

The table below summarises the time (TIME) taken to complete the task (in minutes), lines of code added (XTRALOC) and experience (EXP) in using C++ in years for each subject (SUBJ).

SUBJ	TREATMENT	TIME	XTRALOC	EXP
1	Inherit	63	42	2
2	Inherit	58	39	1
3	Inherit	63	57	1
4	Inherit	93	36	0.5
5	Inherit	109	35	0.5
6	Flat	31	77	1
7	Flat	40	76	1
8	Flat	61	76	1
9	Flat	32	79	1
10	Flat	69	79	2

Table 5.1: Quantitative data collected

The following tables, 5.2, 5.3, 5.4 give the summary statistics for the raw data in table 5.1.

variable	*count*	*mean*	*median*	*variance*	*SD*	*min*	*max*
TIME	10	61.90	62	620.32	24.91	31	109
XTRALOC	10	59.60	66.50	388.49	19.71	35	79
EXP	10	1.10	1	0.27	0.52	0.50	2

Table 5.2: Summary statistics for data collected

variable	*count*	*mean*	*median*	*variance*	*SD*	*min*	*max*
i-time	5	77.20	63	508.20	22.54	58	109
i-loc	5	41.80	39	79.70	8.93	35	57
i-exp	5	1	1	0.38	0.61	0.50	2

Table 5.3: Summary statistics for inheritance version

variable	*count*	*mean*	*median*	*variance*	*SD*	*min*	*max*
f-time	5	46.6	40	302.30	17.39	31	69
f-loc	5	77.4	77	2.30	1.52	76	79
f-exp	5	1.20	1	0.20	0.44	1	2

Table 5.4: Summary statistics for flat version

The data presented in tables 5.1-5.4 show a number of points of interest.

Firstly, the inheritance version has the most compact changes (in terms of lines of code added, XTRALOC), for subjects 4 and 5 less than half of the smallest change for the flat version. This is as expected, because the use of inheritance removes the need to repeat sections of code since the classes can use code (data or functions) declared in another class via the inheritance mechanism. In the flat structure, this mechanism is disabled, so the code must be repeated in any class that wishes to reuse that code. The size of changes made in the flat group is similar (min 76, max 79), whereas the inheritance group ranges from min 35 max 57. For the population as a whole, the data is not normally distributed.

Time taken tends to be less for the flat group than the inheritance group. It is interesting to note that those with the most experience in C++ did not finish more quickly. In the case of the flat group, the most experienced subject took the longest time. One explanation may be that the use of inheritance meant that the subjects found it harder to

comprehend the code (which would be de-localised, i.e. relevant information would be spread among ancestor classes, necessitating more effort to be spent on understanding the code and how to implement the changes than the flat group who did not have to consider the use of inheritance). For the population as a whole, the time is normally distributed.

Also interesting is that the least experienced programmers made the most compact changes (both were in the inheritance group). The reasons for this are unclear, possibly these subject relied more on their training in C++, following "good practice" more closely.

The boxplots in figure 5.1 below show a difference in the time taken by each group, with the inheritance group tending to take longer. There is some degree of overlap of the confidence interval around the median for each group (represented by the shaded area), but the medians themselves do not correspond, (as can be seen from tables 5.3 and 5.4. There is some overlap with the whiskers, indicating that the higher values for the flat treatment are comparable with the lower values for the inheritance treatment.

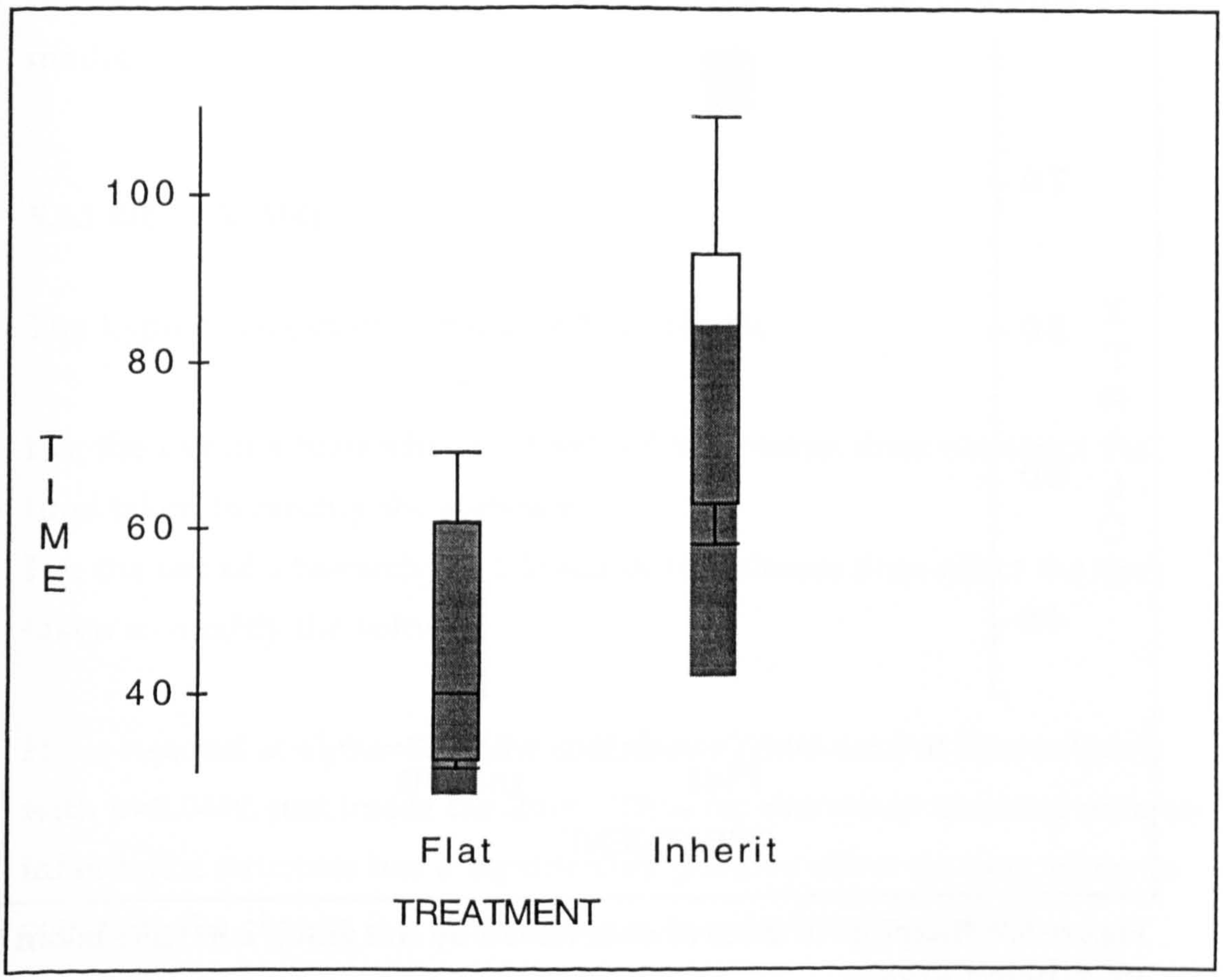

Figure 5.1: Boxplots of time taken by flat group and time taken by inheritance group

More striking are the boxplots of the lines of code added in order to make the change (figure 5.2). Firstly there is absolutely no overlap between the two, indicating a considerable difference in the values, with the flat group obviously adding more code. Ignoring the extreme outlier for the inheritance group (denoted by an "o") we can see a gap of around thirty lines of code between the groups. There is little variation or spread in the figures for the flat group, as evidenced by the absence of whiskers. For the inheritance group, there is a little more variation (again ignoring the outlier) but not a great deal. What figure 5.2 shows is that there is a very definite difference in the number of lines of code added to effect a solution according to whether inheritance was used or not.

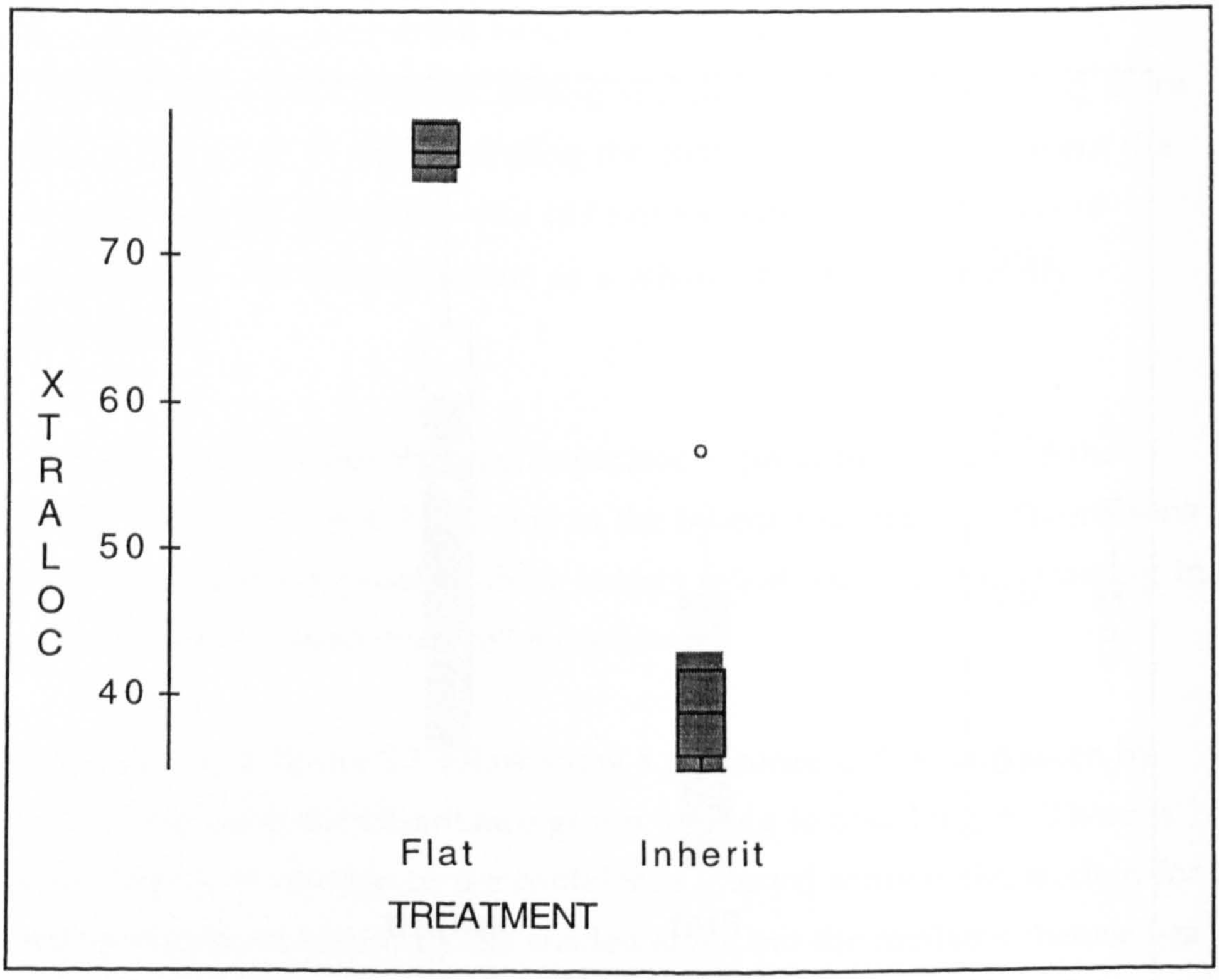

Figure 5.2: Boxplots of lines of code added by flat group and time taken by inheritance group

5.6 Hypothesis formulation and testing

From the raw data and summary statistics, there appears to be a difference between the inheritance group and the flat group, both in terms of size and time. The inheritance group tended to take longer to make the changes (median=63) than the flat group (median=40), and the changes made by the inheritance group were more compact (median=39) than the flat group (median=77). The time data is normally distributed so a two tailed t-test[47] was run to ascertain if there was a statistically significant difference between the two groups (by comparing the

---

[47] The variances are known not to be equal

medians). The data for lines of code added is not normal distributed so, a Mann-Whitney U test was run as a non parametric test to compare medians.

5.6.1 Effort (TIME)

The hypothesis can be summarised as follows:

Ho: the use of a hierarchy at 3 levels of inheritance does not affect the time taken to modify the software
Ha: the use of a hierarchy at 3 levels of inheritance does affect the time taken to modify the software.

Ho is rejected at alpha=0.05 (the confidence limits used at Strathclyde), with p=0.0449, just inside the limit. Thus for this study working with an inheritance structure has a significantly positive effect on time taken to complete the change (i.e. it took longer to complete the task).

2-Sample t-Test of μ1-μ2

No Selector

Individual Alpha Level 0.0500

Ho: μ1-μ2 = 0 Ha: μ1-μ2 ≠ 0

i-time - f-time:

Test Ho: μ(i-time)-μ(f-time) = 0 vs Ha: μ(i-time)-μ(f-time) ≠ 0
Difference Between Means = 30.600000 t-Statistic = 2.403 w/7 df
Reject Ho at Alpha = 0.0500
p = 0.0449

Table 5.5: Two-sample T-test time taken for inheritance version against time taken for flat version

5.6.2 Size (LOC)

Ho: the use of a hierarchy at 3 levels of inheritance does not affect the number of extra lines of code added to the software during the modification

Ha: the use of a hierarchy at 3 levels of inheritance does affect the number of extra lines of code added to the software during the modification

Ho is again rejected at alpha=0.05, with p=0.0086, thus for this study there is a very significant difference between the two samples. This indicates that for this study, using the inheritance structure leads to more compact changes.

To recap, the test indicates that changes to the inheritance version took longer to complete than the flat version but were more compact than those made to the flat version. This confirms what can be seen from the raw data in table 5.1. It contradicts the result at Strathclyde that three levels of inheritance made no difference to the maintainability of object-oriented programs.

Mann-Whitney U

No Selector

Individual Alpha Level 0.0500

Ho: Median1 = Median 2 Ha: Median1 ≠ Median2

Ties Omitted

Inherit:XTRALOC - Flat:XTRALOC :

Test Ho: Median(Inherit:XTRALOC) = Median(Flat:XTRALOC) vs Ha: Median(Inherit:XTRALOC) ≠ Median(Flat:XTRALOC)

	Rank Totals	Cases	Mean Rank
Inherit:XTRALOC	15	5	3
Flat:XTRALOC	40	5	8
Total	55	10	5.500
Ties Between Groups	•	0	•

U-Statistic: 0.000

U-prime: 25.000

Sets of ties between all included observations: 2

Variance: 22.917

Adjustment To Variance For Ties: -0.278

Expected Value: 12.500

z-Statistic: -2.627

p = 0.0086

Reject Ho at Alpha = 0.0500

Table 5.6: Mann-Whitney U test for inheritance version against size for flat version

5.7 Analysis of the Effects of Experience

One obvious potential problem with the data is that as a result of being randomly placed into groups, there was an imbalance in terms of experience. Whereas with a larger sample, any problems would probably be balanced out, with such a small sample, "unbalanced" groups are more obvious. In this experiment, the subjects' experience (of C++) could have been ascertained in advance, so in retrospect, it would have been prudent to use some blocking technique, such as defining two blocks "experienced" and "inexperienced" then allocating treatments randomly to each block. However, given the small number of subjects, it would not be possible to "randomly allocate" the versions among the inexperience block since it would contain just two subjects.

As noted previously, in the experiment the two least experienced subjects made the most compact changes as well as taking the longest time. Presuming compact changes are good (since this seems to be one of the "pros" of using inheritance), then we cannot conclude that relative inexperience means relatively lower ability[48].

To help discover any relationship between experience and performance, in terms of the time taken to complete and the size of change made, the data can be entered into scatterplots

5.7.1 Effects of Experience on Time Taken

In figure 5.3 below, the time taken is plotted against the experience. "x" represented inheritance treatment and "o" the flat treatment. From so few datapoints it is not possible to draw firm conclusions.

---

[48] It has been suggested that these subjects were "more thorough". Since their 6 months experience was gained at university, and presuming the training received was "best practice", it may be that they would implement a "good" change however long it took, where the more experienced subjects (with this additional experience gained in industry) would be more inclined to value speed over style. After all, productivity is still often measured in terms of LOC ...

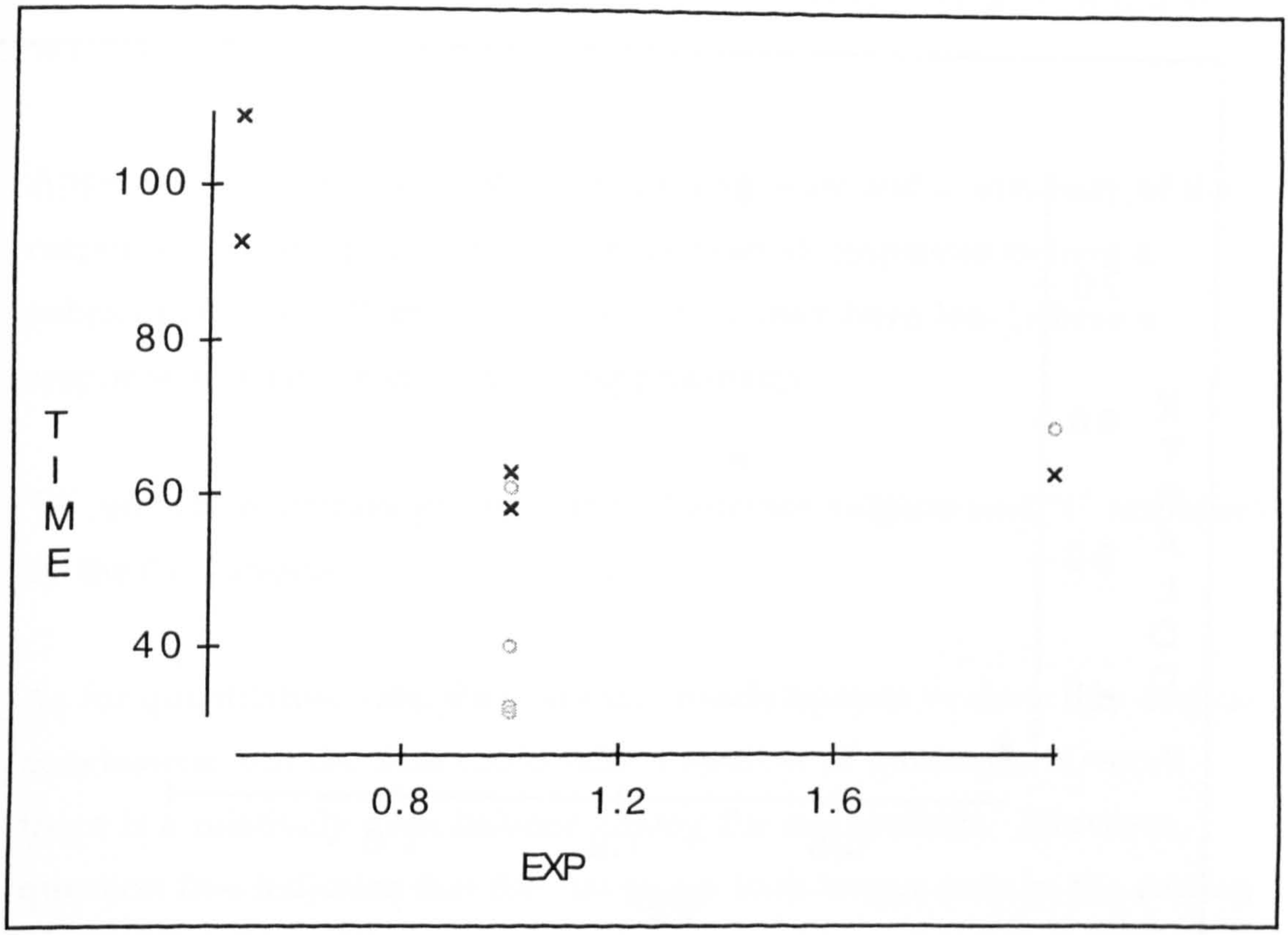

Figure 5.3: Scatterplot of TIME against EXP

5.7.2 Effects of experience on size

As in 5.7.1 above, the number of datapoints is too small to allow firm conclusions to be drawn. Again there is no obvious relationship shown in the scatterplot.

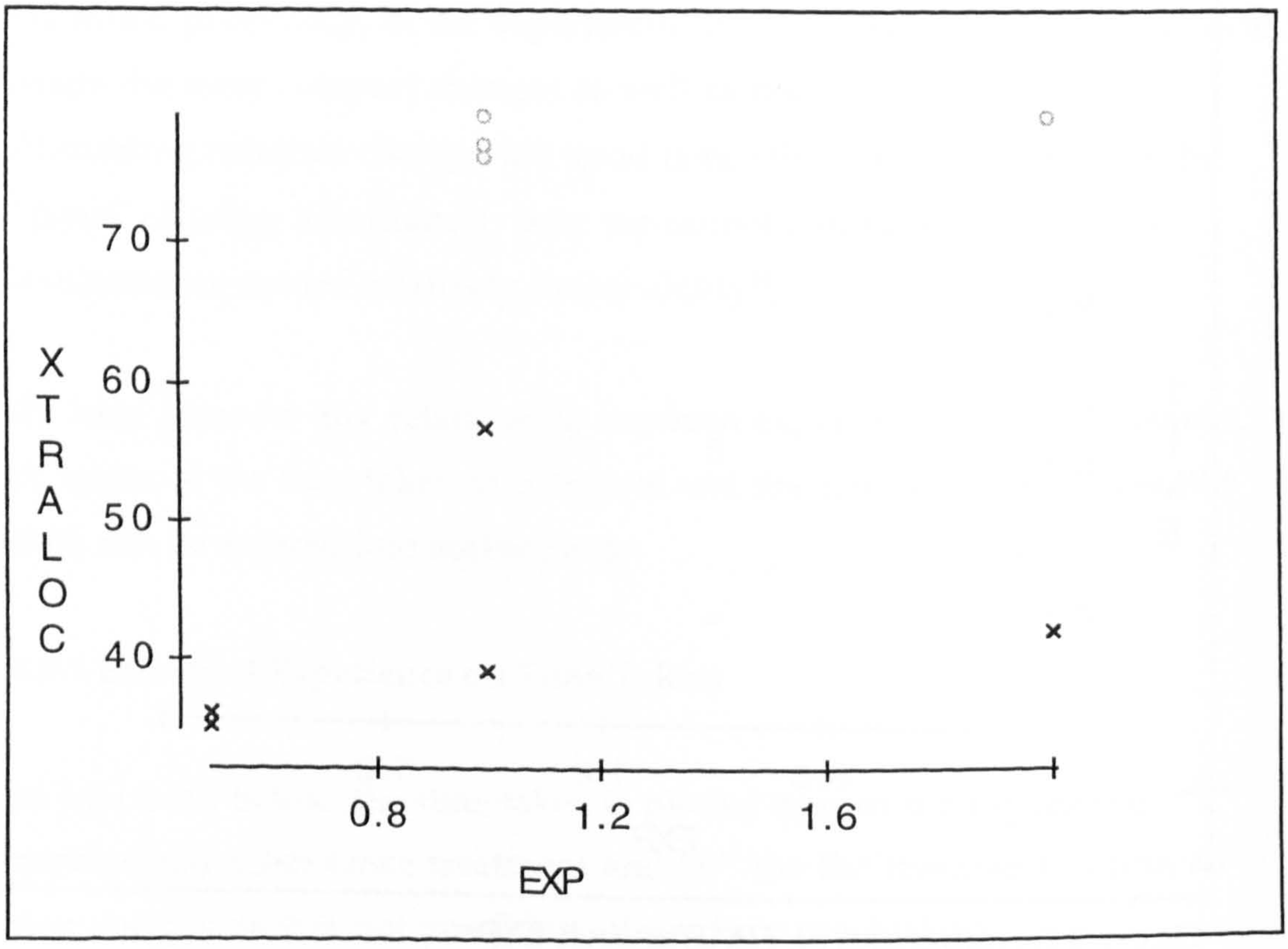

Figure 5.4: Scatterplot of XTRALOC against EXP

5.6.3 Conclusions on the relationship between experience and time taken and experience and LOC added

Regression of the data in figures 5.3 and 5.4 are not significant, i.e. they do not indicate a relationship between the variables. As stated in section 5.6 we cannot safely regard experience as a proxy for ability. Thus we cannot, from this study, draw firm conclusions on the relationship between experience and time taken or LOC added. Thus the imbalance of experience between treatments may not be too serious.

5.7 Debriefing Questionnaire

Each subject completed a debriefing questionnaire after their solution was signed off as successful.

Appendix B contains a copy of the questionnaire and a summary of the responses. Some questions have more than 10 responses (where a subject gave more than one answer), some may have less (where a response was not given or was inappropriate).

"i" refers to responses given by the inheritance subjects and "f" responses by the flat subjects

As for quantitative data, there are too few datapoints to draw any certain conclusions, but the data could raise a number of questions. Overall there is a relatively even balance among the respondents. However, question five indicates that the flat group took longer only in the editing of the code (this group were naturally expected to cut and paste code since inheritance was not used) than the inheritance treatment group who tended to have problems in understanding and debugging of code.

It is interesting to note that although the problem was a natural specialisation, suited to the use of inheritance, the inheritance group tended to have more trouble with understanding the code (q 5) and had less confidence in their understanding of the code (q9), which tends to support the contention that inheritance makes code harder to understand. Additional comments by participants in the inheritance treatment suggest that class hierarchy diagrams are important in understanding code which utilises inheritance.

5.8 Conclusion

As stated previously, the size of the dataset is too small for the results to be considered to counter the conclusions of Daly *et al* (Daly, Brooks et al. 1996; Daly 1996), but it can be viewed as an attempt at replication that raises questions regarding the conclusions of the original. In other words, the results of the original cannot be regarded as proof that inheritance of three levels does not adversely affect maintainability or that it is beneficial.

Obviously the nature of experimentation and particularly replication (as expounded at some length in Daly's thesis (Daly 1996)), means that even with an identical number of subjects, one cannot claim to carry out an exact replication (thus whether the findings agree with/rebut the original are always open to debate). However, attempts such as the experiment described in this chapter provide material which can be used as evidence towards a counter argument. Evidence for both sides needs to be built up before one can decide, on the weight of evidence, which side is more plausible. For this experiment it is interesting that even with the design of experiment being biased in favour of inheritance[49], this application suggested some adverse effects, which are more in keeping with anecdotal evidence from professional software developers.

It is interesting to note that when Daly repeated the experiment with an inheritance structure at five levels the mean time for changes by the inheritance group increased on average by 19.8 minutes per subject, compared with an increase of just 3.5 for the mean time to complete the change for the flat group.

---

[49] It is not possible to say whether or not this is intentional. However, factors that point to a bias are first, the nature of the task, which is a natural specialisation encouraging the use of inheritance to add a class at the bottom of an inheritance hierarchy, rather than a routine maintenance task , such as fixing a bug, or a task with the potential of "ripple through" effects such as changing data nearer the root of the class. Second the Strathclyde subjects were taught by those involved in the experiment, which could be considered as preparation and training for the purposes of the experiment.

The experimental findings, that three levels of experiment have a positive effect in the time taken to complete a maintenance task, offer some support to the contention that the use of inheritance makes the program structure harder for developers to understand. Developers of the system examined in chapter 4 confirmed that inheritance was little used because it made things more difficult. This was borne out by analysis showing classes which were part of an inheritance hierarchy to have a higher incidence of defects. It seems reasonable to conclude that if fewer extra lines of code were added (compared with the flat group), that the extra time taken was not spent in editing or typing, and thus must have gone into understanding the problem and developing a solution.

We can conclude that it would be well worth the effort to attempt to replicate the experiment again, or possibly to introduce a new, less trivial maintenance task.

Blank

In

Original

Chapter 6 Conclusions

Synopsis

From the literature review in chapter 2, a number of themes or lessons to be learned were derived. These were applied to the literature representing research into object-oriented software metrics in chapter 3. From this it could be concluded that OO metrics development seems to have been undergoing the same learning curve as traditional metrics, rather than benefiting from past experience and mistakes. Chapter 4 showed problems with the traditional approach of applying predefined metrics and suggested that it was relatively easy to derive measures and prediction systems to be applied locally as well as presenting the results of a empirical analysis of an object-oriented software system. Chapter 5 follows on from one of the interesting observations on inheritance resulting from the analysis in chapter 4, by describing an experiment looking into inheritance in object-oriented software. This final chapter will look at the conclusions of this work in more detail and suggest further work.

6.1 Summary of work done

The work carried out for this thesis involved an extensive literature survey of object-oriented and traditional software metrics and related subjects (including other object-orientated issues, measurement theory), in excess of 300 references were examined, of these at least 55 pertained to object-oriented software metrics.

From the survey some deficiencies with past approaches to metric development and validation were uncovered. A number of publications on object-oriented software metrics development and/or validation (taken here to be either formal validation or empirical assessment) were considered in the light of these deficiencies.

The case study was based on a large (132+ KLOC) industrial object-oriented software system, designed using Shlaer-Mellor OOA and coded in C++. This involved collecting and sorting though 39 metrics and reducing them to a manageable number without losing interesting information. Exploratory analysis was performed to find attributes of the design which could affect maintainability. From the data collected, prediction systems were developed and tested to demonstrate that it was possible to derive local prediction systems from simple measures.

As a consequence of some of the results from this analysis, which suggested a link between inheritance and errors in the system under study, an experiment was conducted into the impact of inheritance on object-oriented software maintainability.

6.2 Summary of problem area

The problem faced was twofold.

First, relatively little empirical evidence regarding the impact of object-orientation on software development and maintainability exists. Thus much of the literature dealing with object-orientation has tended to display a very positive attitude, giving rise to a widely held belief that the object-oriented paradigm offers advantages over conventional methods in all respects (Fichman and Kemerer 1993). A more specific claim is object-oriented systems are easier to maintain (Booch 1986). A body of empirical evidence, either case study or experimentation, is needed to provide information on how OO affects important software attributes, such as maintainability and quality.

Second, despite numerous metrics being put forward, many would be unsuitable for our purpose; of predicting errors and size in an object-oriented software system. There are four reasons why metrics/prediction systems might be unsuitable:

- metrics having been developed for structured software and are thus unlikely to be suitable for OO software;
- metrics being speculative or of dubious validity/usefulness;
- metrics being specific to a particular language, environment or method;
- metrics seemingly suitable and valid/useful but difficult to collect (lack of tool support or not available early enough in the development process).

These problems are not all necessarily easy to avoid. The first is simple enough – consider only those metrics which have been developed for object-oriented software. However, the immaturity of such metrics makes the second, third and fourth *pitfalls more likely to occur*. All metrics will go through a speculative phase, before some demonstration of their validity or usefulness can be made. As has been emphasised in previous chapters (2 and 3), validity and usefulness are not the same thing, so even if a metric can be shown to be valid, showing that it is useful can be difficult. In order to show usefulness, empirical studies need to be carried out. These would require suitable data. The more studies carried out, the more confidence we can have in the results.

The third reason, that of metrics being suited only to a particular language, method and so on, is also connected with the lack of data available to empirical studies. It is likely that developers of metrics will be familiar with and have data available for one particular language, method environment and so on, and thus any development and testing will be based upon that available data. Many metrics should not be

generalised, but frequently are. It is rare for metrics to be presented as metrics for C++ systems, but more commonly they are presented as metrics for OO systems. In some cases it is obvious that certain features are specific to, say, a particular programming language, but not always.

The fourth problem or reason for unsuitability with many proposed metrics is difficulty in collection. With speculative metrics, little thought tends to be given to collection (or even counting rules, in all too many cases). Even where some validation or empirical evaluation is carried out, it is not infrequent that a tool needs to be written specially to collect the metrics. This constitutes additional effort on the part of the user wishing to apply such metrics. Either they must write software to collect the metrics or wait for tool vendors to incorporate them into case tools. Furthermore, if, as in many cases, metrics are extracted from code (via static code analysers, for example), the value is reduced, since they cannot be used for early warnings or predictions.

It can be seen that predefined metric suites can suffer from a number of disadvantages. That is not to say that they are without value. Interesting things about the nature of the software systems under scrutiny can be more easily uncovered. They provide the basis for other, independent empirical studies (the Chidamber and Kemerer metrics and the Lorenz and Kidd metrics have been applied independently, for example). They may also provide ideas which can be adapted by users to suit the specific needs of their environment or project.

6.3 Summary of aims

The aims of the thesis (section 1.1) were

i) To investigate the impact of key OO mechanisms, specifically inheritance, on software maintenance.

This aim was motivated by several factors:

- inheritance can be closely associated with "object-orientedness" – looking at an object-oriented program one would expect to see some use of inheritance;

- Grady Booch's claim in 1986 that object-oriented software is easier to understand and maintain (Booch 1986). In this paper Booch tends not to concern himself much with inheritance and explains the claims for maintainabilty and understandabilty as being "due to the fact that objects and their related operations are localised" and so "reduce the scope of change upon the system". Whilst this applies to encapsulation and data hiding (which are not peculiar to OO), clearly the inheritance mechanism leads to some degree of delocalisation, where variables and operations may be declared in one class and used in another;

- Such concerns are voiced in a small number of papers, primarily concerned with tool support, and are the third factor to influence the aim. Papers by (Lejter, Meyers et al. 1992; Wilde and Huitt 1992; Wilde, Matthews et al. 1993; Li and Henry 1993a) consider the effects of inheritance (among other mechanisms) on maintainability and understandability.

The aim has been achieved by the analysis of error data in a C++ system, as described in chapter 4. The distribution of errors and error densities was analysed. The findings can be summarised as follows:

• the Chidamber and Kemerer metrics could not all be collected from the design documentation;

• there is some evidence in support of the "20:80" rule, since 22% of classes accounted for 75% of defects;

• it was found that classes involved in an inheritance hierarchy had three times the defect density of non inheritance classes, that the highest defect densities of all could be mapped to classes at the bottom of their respective hierarchy (i.e. classes with no descendants), and that little use was made of inheritance, with the median figure for DIT being 0;

• that size can be predicted very well from a simple measure available early in the analysis/design phase, in this case by counting the number of states, and this measure could be easily and automatically extracted from the case tool model;

• that defects can be predicted well from a simple measure available from the analysis/design phase, and this measure could be extracted automatically from the case tool model.

• the value of building local prediction systems has been demonstrated, since it was easier to collect simple, readily available measures and derive prediction systems from them than to use pre-defined metric suites, and these prediction systems could be shown to be accurate.

The significance of the findings will now be discussed. It is worth emphasising that the extent to which the results of a case study can be generalised continues to be a subject of debate. Yin's view, (Yin 1994), that a case study can be generalised into theory, but not to other populations, seems sensible.

Firstly it appears that not all of the Chidamber and Kemerer metrics can be regarded as design metrics. This makes no comment on their accuracy or validity; because they could not be collected, they could not be assessed. However, their usefulness can be disputed. It is accepted that the earlier a metric can be collected, the more use it can be since it can be used to make decisions regarding the remainder of the development process. If a metric is difficult to collect then its usefulness can be questioned in view of the amount of effort expended in order to collect it. The two metrics collected were of limited use. NOC (number of children) was not found to correlate with any attribute of interest. Although DIT (depth of inheritance tree) had some correlation with errors, it ignores the finding that simply being part of an inheritance structure had an effect on the number of errors. This is because the root of an inheritance hierarchy is counted at 0, and is thus allocated the same value as a class which is not involved in an inheritance hierarchy. Perhaps if the root were counted as 1, the metric would more accurately reflect the finding that inheritance impacts upon errors.

The 20:80 "rule" is often quoted to illustrate that a relatively small proportion of, say, modules are responsible for a large number of problems, such as maintenance effort. This case study confirms this ratio, albeit slightly adjusted at 22:75 for classes : errors. Thus in the context of object-oriented system, if the troublesome 20ish % of classes can be identified early on, then this can be fed into the decision making process with regard to testing effort, design reviews and so on.

The concentration of errors in inheritance hierarchies suggests that a higher proportion of resources e.g. testing effort should be devoted to classes in an inheritance hierarchy. As with the point above, if we are able to pinpoint the most problematic classes, then we can more

efficiently and effectively allocate resources to find/eradicate errors. This concentration of errors also raises questions about the use of inheritance. Bearing in mind that there is little use made of the mechanism in the case study, it may be that inheritance leads to designs and/or code which is harder to understand. This seems feasible, since there is inevitably some delocalisation, whereby data or methods declared in one class may be used in a descendant of that class. In order to understand what a class is doing and how, the delocalized data and functionality must be brought back together. This is not necessarily an easy task, it may involve several levels of inheritance and even multiple inheritance in some situations. Consequently, software that is harder to understand will be harder to test/maintain because of the ability to understand software requisite to those tasks.

The low levels of inheritance may be due to the difficulties that developers had in understanding inheritance (this is confirmed by anecdotal evidence), which in turn may in some part be due to their relative inexperience (this being their first OO development, although it must be stressed that they had received training), but is also likely to be influenced by the points made in the paragraph above, that the declocalization which occurs when inheritance is employed makes the software harder to comprehend. It is possible that the problem area to which OO was applied had an influence on the low levels of inheritance. It is possible that some areas do not "lend" themselves naturally to the use of inheritance, a possibility which may gain some credence from the fact that examples given in text books are fairly limited, with GUIs (where shapes and buttons etc. can be classified), chemical/ bottling plants (different types of sensor, and so on) being popular. It may be that many other real life applications require less classification and thus require less inheritance. This may mean that in some situations it is used as a mechanism for code reuse within the application, which may

in itself cause comprehension problems where classes which do not appear to be related conceptually are placed in a hierarchy purely to facilitate code sharing.[50]

The relationship between the position in the hierarchy and error density is also worth further consideration. A feasible explanation is that classes at the bottom of the hierarchy are "concrete", they are the classes that "do" things. Superclasses may on the other hand be abstract. In other words such classes were written to allow subclasses to inherit (and probably add to) the behaviour of the parent class behaviour, so these classes do not actually "do" anything. Indeed their methods may not be complete and may not be able to be implemented without redefinition.

In the absence of other datasets or case studies to work on[51], a small scale experiment was carried out, comparing an C++ program using inheritance with an equivalent flat version. Its findings were:

- it took longer to implement the changes to the inheritance version;
- that the changes made to the inheritance version were more compact.

Again we cannot generalise these results, particularly since student volunteers and small scale artefacts were used, so the population sample cannot be regarded as representative of the "real world".

The experiment considered another type of maintenance, namely perfective maintenance, where software is changed or augmented to meet a change in user requirements. Since it does not consider errors, the results cannot be considered as a confirmation (or repudiation) of the

---

[50] It must be emphasised that this is not so for this case study.
[51] By which I mean datasets I could use or case studies I could apply the same approach to rather than those carried out and published by others.

case study findings. They can be considered complementary to the case study. It can be inferred that these developers/maintainers find inheritance takes more effort to work with, compared with a flat structure. In the case study, defects density for the inheritance classes was three times the level for non-inheritance (i.e. "flat") classes. In the experiment, the subjects took longer to complete the changes to the inheritance version of the program, despite the change being a natural specialisation of an existing class, and thus not disruptive of the existing inheritance structure. It would not be unreasonable to conclude that the problem is, at least in part, one of comprehension. In the experiment the subjects working on the inheritance version took longer but produced many fewer additional lines of code than the "flat" subjects. In the case study the errors were concentrated on the inheritance classes. It is certainly not unreasonable to suggest the errors are more likely if a developer finds it harder to understand a class, and likewise it will be harder to test and trap errors.

The ability to predict attributes of interest before they can be measured is valued as a useful input into the project management function. The prediction systems were derived from readily available measures, both of which were available from the case tool analysis/design model. Both predictions systems can be considered accurate (from the high adjusted R^2 values of 96.6% for the prediction of LOC and 87.2% for the prediction of errors). These metrics cannot be said to be generalizable, but the approach could be, and the direct measurements taken could perhaps hold as indicators of size and errors for other OO systems.[52]

---

[52] This, of course, would need further empirical study with other datasets. The point is that the constant and multipliers are unlikely to remain the same but it is possible that the direct attributes may be suitable indicators of size and errors in other projects.

More importantly than the actual prediction systems presented, is that this case study has demonstrated that deriving simple, local prediction systems is possible. This is more significant than adding yet another set of metrics to the public domain, it shows that there are ways of deriving predictions without the need to have expert knowledge of measurement theory, of the application type, of the paradigm and so on. It also suggests that it is not always necessary to have an intermediate measure, such as complexity, which would then be used to indicate maintenance effort. Here we have taken a simple direct measure and used it to estimate the number of errors, which is one aspect of maintenance effort. There will be more discussion and a summary of the approach under aim (iv) below.

ii) To examine previous work in the area of complexity metrics development and identify any problems with the approach. These problems could be used to derive "lessons to be learned", which would be considered when assessing the metrics proposed for object-oriented software.

Chapter 2 considered some of the important traditional metrics and important themes in metrics development, such as measurement theory and empirical validation/evaluation. From this five problems with traditional metrics were identified, which could be translated as lessons to be learned from past mistakes or inadequacies. These were:

- the lack of a clearly defined goal;
- the failure to distinguish between measures and prediction systems (metrics can be taken to mean either);
- poor definition of attributes to be captured and the counting rules for doing so;

- poor validation;
- failure to establish validity and/or usefulness of metrics.

iii) To consider the available OO metrics in the light of what was discovered from the above aims (examination of previous work and the impact of inheritance on maintainability) and ascertain which, if any metrics fulfilled the criteria of easy to obtain, useful metrics.

A large number of metrics were considered, some in more detail than others. From this it could be seen that a majority of these metrics had inadequacies, many of which could have been addressed if past experience and problems had been heeded, for example the lack of a clear goal. It is true that for practical reasons it may be difficult to avoid all of the criticisms on the list above, particularly the latter two, to the satisfaction of all. Sadly, many of the proposed metrics/predictions systems were unclear on what was being measured and why. In order to be taken seriously a metric should at least have its purpose made clear. Secondly a number of the metrics proposed were simply traditional metrics applied to OO with little consideration for new features, such as inheritance. Again this is not acceptable, particularly since the traditional metrics suggested, such as McCabe, had already been largely discredited. To some extent, the desire to be first to publish will lead to the presentation of speculative metrics, with no empirical validation, and at best some formal validation according to measurement theory. To be taken seriously, such speculative metrics need to be followed up by empirical validation. This has happened with the more popular metrics, such as Chidamber and Kemerer and Lorenz and Kidd (a subset of measures were investigated in (Harrison and Counsell 1997)).

iv) To show how local prediction systems can be easily derived from local data, and the accuracy of these predictions ascertained.

Chapter 4 describes the process of selecting measures, deriving and testing a prediction system. It shows that simple measures can be used to predict attributes of interest without resorting to complex metrics with multiple inputs or trying to capture complex intermediate attributes such as complexity. A simple regression procedure was used to formulate the equation using the selected input and a set of historical data, all of which can be collected from existing electronic data sources, such as case tool, incident reports, change logs and so on. This meant that little effort was required to collect the input measures, since they already existed. Many metrics require more unusual measures to be collected, which must be done by hand or requires tool support incorporating the particular metric set to be available. With many, if not most, metrics, the difficulty lies not in the actual calculation but in the collection. By utilising measures which are readily available since they are already collected for other purpose, the effort of data collection is reduced. Data can be fed into a statistics package which will formulate a regression equation to be used as a prediction system.

6.4 Weaknesses/problems

(i) Neither the study nor the experiment can be considered definitive.

- Firstly there is just one small experiment and one case study. The experiment would need further replication before we could feel confident that the findings would hold for further studies, particularly since the results of the experiment differed from the original carried out at Strathclyde. Likewise using a number of case studies allows for more confidence in the case study results – the

more case studies for which a hypothesis holds, the more likely that it can be generalised.

- In mitigation, it is in the nature of empirical work that we cannot have a definitive answer to a question – we can build up a body of evidence which will support the probability of a particular hypothesis, but cannot claim to have proved it.

(ii) Additionally, it is always possible to criticise the use of student programmers and question the effect of experimental conditions on the subjects' behaviour. An experiment allows more control, but suffers in that it cannot be said to mirror "real" software. By necessity, the problems will be small scale, and more likely than not use student programmers as subjects. In this experiment we were further hampered by the need to rely on volunteers, since it was not possible to incorporate the experiment into a taught unit and compel students to take part. Subsequently numbers were smaller than had been hoped. This means that the study cannot be said to support or refute the claims made by the developers of the experiment at Strathclyde.

(iii) An additional problem lies with the experimental procedure, in the way in which the subjects were allocated. With the benefit of hindsight it would have been sensible to reduce the potential for problems with random allocation and small groups. Ideally a blocking mechanism would have been employed to ensure that the experience of the subjects would have been better balanced between the two groups, by sampling randomly from two blocks (one block of experienced subjects and one block of inexperienced subjects).

(iv) It is in the nature of case studies that they cannot be generalised to other populations. Firstly what can be collected is limited to what is

available or what the company will permit. Secondly the success of a case study can be greatly affected by the enthusiasm of the industrial contacts, which in turn can be affected by changes in personnel, workload etc.. The case study gives "real data" but at the loss of control. In this case, data from another project initially offered did not materialise following a reorganisation of the company. The same problems with obtaining further data were experienced when trying to obtain qualitative data to enhance the understanding of the case study. This case study was originally envisaged as a pilot study, allowing the examination of a larger system with less disruption to the company. Second time around the "right" or relevant questions to ask, measures to collect etc. would be known in advance, allowing most data to be collected at the start of the study when enthusiasm (that is the enthusiasm of the industrial collaborators) for the project would be highest. This approach would have been considerably more valuable than a single case study.

A further criticism is that the case study was based on the first OO project built by the team. This would almost certainly have some influence on the data. However, it must be emphasised that the team were experienced software developers and had all undergone training. Additionally many companies are still in the early stages of migration to OO, so the case study may be of particular interest to them.

However, the case study and experimental results do confirm anecdotal evidence that software designed using inheritance can be hard to understand and thus maintain. Both the study and the experiment suggest interesting avenues for further research, namely further empirical research into the effects of inheritance on defects, and into the effects of inheritance on maintenance effort.

6.5 Suggestions for further work

Further datasets/case studies are needed if results are to be generalizable. It would be interesting to include data from projects where the developers had experience of developing OO software. This would indicate if the concentration of errors in the inheritance hierarchy were influenced by the inexperience of the team or were entirely due to the use of the inheritance mechanism.

The experiment should be repeated using larger groups of subjects[53] and/ a blocking technique to ensure a more even distribution of experience (if the sample size demands this). Replication helps build a body of evidence and could help ascertain whether OO maintenance changes do take longer to complete and avoid the possibility that the effect is due to inexperience rather than the use of inheritance.

The case study and experiment have led to other ideas for related research. It would be interesting to see the effects of maintenance changes on the inheritance hierarchy. The changes to be made for the experiment were natural specialisations of a class at the deepest level of the existing hierarchy, allowing for a class to be added without disrupting or affecting the rest of the hierarchy. This would not always be the case. A maintenance change may well involve making changes to a class higher up in the hierarchy or possibly inserting a class between two existing classes. This is particularly true of "real" systems where non trivial changes occur. It would be interesting to study the effect on and disruption to the rest of the hierarchy, in terms of the stability of the structure and potential for "ripple through" effects to subclasses and collaborating classes.

---

[53] Also larger systems and more difficult changes, as recently carried out by Dr Rachel Harrison at Southampton (verbal communication).

It would also be interesting to study the effect of feedback on the development process. For example, if an inheritance hierarchy was predicted to have an increased likelihood of defects at say design time, how would this affect the development of the system (presuming action was taken such as increasing resources, testing effort etc.) when compared with the development of an identical system without the feedback.

It may also be of interest to separate the post delivery defects from those found by pre-delivery testing. This may indicate if defects related to inheritance are more likely to be missed during testing (possibly indicating inappropriate or inadequate testing strategies), or whether the high incidence of inheritance related defects found post delivery are due to the sheer number of defects to be found there.

6.6 Contribution to knowledge of the thesis

The thesis has contributed the following:

- a hypothesis and empirical evidence on the effects of inheritance on defects (via case study);

- empirical evidence, via an experiment, on the effects of inheritance on maintenance effort, which will add to limited existing empirical evidence since it is a replication of a previous experiment;

- a demonstration that it is possible to derive simple yet accurate locally *applicable prediction* systems from existing data, without recourse to complex pre-defined suites of metrics;

• a list of potential short comings of metrics (measures/predictions systems);

• an extensive review of object-oriented software metrics.

All contributions should be of interest to academia. In particular the case study has led to a firm hypothesis and supporting empirical evidence from a "real" system. This is a foundation for future research to deal with an important issue (maintenance and perhaps testing) applied to a paradigm which continues to dominate commercial software development. The review of object-oriented software metrics is extensive and covers not only the more popular metrics, but also relatively obscure contributions, providing a good starting point for anyone unfamiliar with the area of object-oriented software metrics. The list of potential short comings, derived from the review of traditional metrics development, provide a useful checklist for metrics development and validation.

The first three points may also be of interest and of practical use to industry. Firstly the link between inheritance and defects and inheritance and maintenance effort, may cause extra care to be taken in inspections, walkthroughs, reviews and testing of systems or parts of systems which utilise inheritance. This in turn may lead to more defects being spotted and fixed pre-delivery, and thus alleviate increased maintenance effort to some extent. The third contribution, the demonstration that it is possible to derive useful, local metrics without recourse to predefined metrics, requiring new tools should perhaps reassure quality and project managers that measurement and analysis can provide useful project information without the need for a great deal of effort or expense.

6.7 Final conclusions

The thesis has demonstrated that the aims outlined in chapter 1 have been met (section 6.3). It has led to a number of potential avenues for future research (section 6.5). It has resulted in papers and presentations (section 1.6) which have generated interest from both academia and industry.

References

Abbott, D. H., T. D. Korson, et al. (1994). A Proposed Design Complexity Metric for Object-Oriented Development. Dept of Computer Science, Clemson University.

Abreu, F. B. (1993). Metrics for Object-Oriented Development. Proc. 3rd International Conference on Software Quality, Lake Tahoe, Nevada, USA.

Abreu, F. B. and R. Carapuca (1994). "Candidate Metrics for Object-Oriented Software Within a Taxonomy Framework." Journal of Systems and Software 26(1): 87-96.

Abreu, F. B. and R. Carapuca (1994). Object-Oriented Software Engineering: Measuring and Controlling the Development Process. 4th International Conference on Software Quality, Washington DC,

Abreu, F. B., M. Goulao, et al. (1995). Toward the Design Quality Evaluation of Object-Oriented Software Systems. Proc. 5th International Conference on Software Quality, Austin, Texas, USA,

Abreu, F. B. and W. Melo (1996). Evaluating the Impact of Object-Oriented Design on Software Quality. Proc. 3rd International Software Metrics Symposium (METRICS '96), Berlin, Germany, IEEE.

Adelman, L. (1991). "Experiments, Quasi-Experiments and Case Studies: A Review of Empirical Methods for Evaluating Decision Support Systems." IEEE Transactions on Systems, Man, and Cybernetics 21(2): 293-301.

Akiyama, F. (1971). An Example of System Software Debugging. Proceedings. IFIP Congress: 353-8.

Albrecht, A. J. (1979). Measuring Application Development Productivity. SHARE-GUIDE Symposium, Monterey, CA, IBM.

Albrecht, A. J. (1984). AD/M Productivity Measurement and Estimate Validation. Report. IBM Corporate Information Systems and Administration.

Albrecht, A. J. and J. R. Gaffney (1983). "Software function, source lines of code, and development effort prediction: a *software science validation.*" IEEE Transactions on Software Engineering 9(6): 639-648.

Alexander, C. (1964). Notes on the Synthesis of Form. Cambridge MA, Harvard University Press.

Armour, F., B. Catherwood, et al. (1996). Experiences Measuring Object Oriented System Size with Use Cases. ESCOM, Wilslow, UK.

Baker, A. L. and S. H. Zweben (1980). "A comparison of measures of control flow complexity." IEEE Transactions on Software Engineering 6(6): 506-511.

Baker, B. O., C. D. Hardyck, et al. (1966). "Weak Measurements Vs. Strong Statistics: An Empirical Critique of S.S. Stevens' Proscriptions on Statistics." Educational and Psychological Measurement 26(2): 291-309.

Balasubramanian, N. V. (1996). Object-Oriented Metrics. Asia-Pacific Software Engineering Conference, IEEE.

Barnard, J. (1998). "A New Reusability Metrics for Object-Oriented Software." Software Quality Journal 7: 35-50.

Basili, V. R., L. Briand, et al. (1995). A Validation of Object-Oriented Design Metrics. Technical. University of Maryland.

Basili, V. R. and B. T. Perricone (1984). "Software errors and complexity: an empirical investigation." Communications of the ACM 27(1): 42-52.

Basili, V. R. and H. D. Rombach (1988). "The TAME project: Towards Improvement-oriented software environments." IEEE Transactions on Software Engineering 14(6): 758-771.

Behrens, C. A. (1983). "Measuring the Productivity of Computer Systems Development Activities with Function Points." IEEE Transactions on Software Engineering 9(6): 649-658.

Benington, H. D. (1956). Production of large computer programs. Symp. on Advanced Computer Programs for Digital Computers, Washington, D.C., Office of Naval Research.

Benington, H. D. (1983). "Production of Large Computer Systems." Annals of the History of Computing 5(4): 350-361.

Benyon-Tinker, G. (1979). Complexity Models in an Evolving Large System. Proceedings ACM Workshop on Quantitative Models.

Bieman, J. M. and J. X. Zhao (1995). Reuse Through Inheritance: A Quantitative Study of C++ Software. Proceedings of the Symposium on Software Reusability, Seattle, Washington.

Binkley, A. B. and S. R. Schach (1996). "A Comparison of Sixteen Quality Metrics for Object-Oriented Design." Information Processing Letters 58(6): 271-275.

Booch, G. (1986). "Object-Oriented Development." IEEE Transactions on Software Engineering 12(2): 211-221.

Booch, G. (1991). Object-Oriented Analysis and Design with Applications. Benjamin/Cummings.

Booch, G. (1994). Object-Oriented Analysis and Design With Applications. Redwood City, California, Benjamin/Cummings.

Briand, L., K. El Emam, et al. (1996). "On the Application of Measurement Theory in Software Engineering." Empirical Software Engineering 1(1): 61-88.

Bunge, M. (1977). Treatise on Basic Philosophy: Ontology I: The Furniture of the World. Boston, Riedel.

Burbeck, S. L. (1996). "Real-Time Complexity Metrics for Smalltalk Methods." IBM Systems Journal 35(2): 204-226.

Capper, N. P., R. J. Colgate, et al. (1994). "The Impact of Object-Oriented Technology on Software Quality: Three Case Histories." IBM Systems Journal 33(1): 131-157.

Card, D. N. and W. W. Agresti (1988). "Measuring software design complexity." J. of Systems & Software 8: 185-197.

Cartwright, M. and M. Shepperd (1997a). Building Predictive Models from Object-Oriented Metrics. Proc 8th European Software Control and Metrics Conf., Berlin.

Cartwright, M. and M. Shepperd (1997b). An Empirical Study of Object-Oriented Metrics. Technical Report, Dept. of Computing, Bournemouth University.

Cartwright, M. H. (1998). "An Empirical View of Inheritance." Information and Software Technology (accepted for publication).

Catherwood, B., M. Sood, et al. (1997). Continued Experiences Measuring Object-Oriented Systems. ESCOM 97, Berlin.

Chen, J. Y. and J. F. Lu (1993). "A New Metric for Object-Oriented Design." Information and Software Technology 35(4): 232-240.

Chidamber, S. and C. F. Kemerer (1995). "Reply To Comments on "A Metrics Suite for Object-Oriented Design"." IEEE Transactions on Software Engineering 21(3): 265.

Chidamber, S. R., D. P. Darcy, et al. (1997). Managerial use of object oriented software metrics. Working Paper. Katz Graduate School of Business, Univ. of Pittsburgh.

Chidamber, S. R. and C. F. Kemerer (1991). Towards a Metrics Suite for Object Oriented Design. OOPSLA '91, ACM.

Chidamber, S. R. and C. F. Kemerer (1994). "A Metrics Suite for Object-Oriented Design." IEEE Transactions on Software Engineering **20**(6): 476-93.

Churcher, N. I. and M. J. Shepperd (1995). "Comment on "A Metrics Suite for Object-Oriented Design"." IEEE Transactions on Software Engineering **21**(3): 263-265.

Constantine, L. (1997). "Efficient Objects." Object Magazine 7(7): 71-72,18.

Coppick, J. C. and T. J. Cheatham (1992). Software Metrics for Object-Oriented Systems. Proceedings - 1992 ACM Computer Science Conference Communications, Kansas City, Mo, USA, ACM.

Coulter, N. S. (1983). "Software Science and Cognitive Psychology." IEEE Transactions on Software Engineering 9(2): 166-171.

Curtis, B., S. Sheppard, et al. (1979). Third time charm: stronger prediction of programmer performance by software complexity metrics. 4th IEEE Intl. Conf. on Softw. Eng., IEEE.

Curtis, B., S. Sheppard, et al. (1979). "Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe Metrics." IEEE Transactions on Software Engineering 5(2): 96-104.

Daly, J., A. Brooks, et al. (1996). Evaluating the effect of inheritance on the maintainability of object-oriented software. Empirical Studies of Programmers, Washington, DC.

Daly, J. W. (1996). Replication and a Multi-method Approach to Empirical Software Engineering Research. PhD Thesis, Univ. of Strathclyde.

de Champeaux, D. (1997). Object-Oriented Development Process and Metrics. NJ, Prentice Hall.

de Champeaux, D., A. Anderson, et al. (1992). Case Study of Object-Oriented Software Development. OOPSLA '92, ACM.

de Champeaux, D. and P. Faure (1992). "A Comparative Study of Object-Oriented Analysis Methods." JOOP 5(1): 21-33.

de Champeaux, D., D. Lea, et al. (1992). The Process of Object-Oriented Design. OOPSLA '92.

DeMarco, T. (1982). Controlling Software Projects. Management, Measurement and Estimation. NY, Yourdon Press.

Dvorak, J. (1994). "Conceptual Entropy and Its Effect on Class Hierarchies." IEEE Computer 27(6): 59-63.

Ebert, C. and I. Morschel (1997). "Metrics for Quality Analysis and Improvement of Object-Oriented Software." Information and Software Technology 39(7): 497-509.

Fenton, N. and B. Kitchenham (1991). "Validating Software Measures." Journal of Software Testing, Verification and Reliability 1(2): 27-42.

Fenton, N. and S. Pfleeger (1996). Software Metrics : A Rigorous and Practical Approach. Second Edition. International Thomson Computer Press.

Fenton, N. E. (1991). Software Metrics: A Rigorous Approach. Chapman & Hall.

Fichman, R. G. and C. F. Kemerer (1993). "Adoption of software engineering process innovations: the case of object orientation." Sloan Management Review 34(2): 7-22.

Finkelstein, L. and M. S. Leaning (1984). "A review of the fundamental concepts of measurement." Measurement 2(1): 25-34.

Fitzimmons, A. (1978). Relating the Presence of Software Errors to the Theory of Software Science. Proceedings 11th Hawaii International Conference on Systems Science.

Funami, Y. and M. H. Halstead (1976). A Software Physics Analysis of Akiyama's Debugging Data. Proceedings of the Symposium on Computer Software Engineering.

Furey, S. (1997). "Why We Should Use Function Points." IEEE Software 14(2): 28, 30.

Geritsen, R., H. Morgan, et al. (1977). "On some metrics for databases or what is a very large database?" ACM SIGMOD Record (June): 50-74.

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

Gill, G. K. and C. F. Kemerer (1991). "Cyclomatic complexity density and software maintenance productivity." IEEE Transactions on Software Engineering 17(12).

Graham, I. (1995). Migrating to Object Technology. Addison-Wesley.

Graham, I. (1995). Progress With Object-Oriented Metrics. Object Expo Europe, London.

Graham, I. (1996). "Making Progress in Metrics. Task Point Analysis can be Performed at the Requirements Stage." Object Magazine 6(8): 68-73.

Gray, R. H. M., B. N. Carey, et al. (1991). "Design Metrics for Database Systems." BT Technology Journal 9(4): 69-79.

Halstead, M. H. (1972). "Natural Laws Controlling Algorithmic Structure." SIGPLAN Notices 7(2): 19-26.

Halstead, M. H. (1977). Elements of Software Science. New York, Elsevier North-Holland.

Halstead, M. H. (1979). Advances in software science. Advances in Computers. NY, Academic Press.

Hamer, P. G. and G. D. Frewin (1982). M.H. Halstead's Software Science - A Critical Examination. 6th Intl. Conf on Softw. Eng., Tokyo, IEEE.

Harrison, R. and S. Counsell (1997). An Assessment of the Impact of Inheritance on the Maintainability of Object-Oriented Systems.

International workshop on Empirical Studies of Software Maintenance, Bari, Italy.

Harrison, R., S. J. Counsell, et al. (1997). Empirical Assessment of Object-Oriented Design Metrics. EASE-97, Empirical Assessment in Software Engineering, Keele, UK.

Harrison, R., L. G. Samaraweera, et al. (1996). "An evaluation of Code Metrics for Object-Oriented Programs." Information and Software Technology 38(7): 443-450.

Hatton, L. (1997). "Software Failures: Follies and Fallacies." IEE Review 43(2): 49-52.

Henderson-Sellers, B. (1991). Some Metrics for Object-Oriented Software Engineering. Proc. TOOLS 6, Prentice Hall.

Henderson-Sellers, B. (1994). Identifying Internal and External Characteristics of Classes Likely to be Useful as Structural Complexity Metrics. OOIS '94. 1994 International Conference on Object-Oriented Information Systems 19-21 December 1994, London, UK, Springer-Verlag.

Henderson-Sellers, B. (1996). Object-Oriented Metrics: Measures of Complexity. New Jersey, Prentice Hall.

Henderson-Sellers, B., L. Constantine, et al. (1996). "Coupling and Cohesion (Towards a Valid Metrics Suite for Object-Oriented Analysis and Design)." Object-Oriented Systems 3(3): 143-158.

Henderson-Sellers, B. and J. M. Edwards (1994). BOOKTWO of Object-Oriented Knowledge: The Working Object. Prentice Hall.

Henry, S. and D. Kafura (1981). "Software quality metrics based on interconnectivity." J. of Systems & Software 2(2): 121-131.

Henry, S. and D. Kafura (1981). "Software structure metrics based on information flow." IEEE Transactions on Software Engineering 7(5): 510-518.

Henry, S. and D. Kafura (1984). "The evaluation of software systems' structure using quantitative software metrics." Software Practice & Experience 14(6): 561-573.

Henry, S. M. and C. Selig (1990). "Predicting code complexity from software designs." IEEE Software 7(2).

Hitz, M. and B. Montazeri (1996). "Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective." IEEE Transactions on Software Engineering 22(4): 267-271.

Hopkins, T. (1994). Complexity Metrics for Quality Assessment of Object-Oriented Design. 2nd International Conference on Software Quality Management, Edinburgh.

Hudli, R. V., C. L. Hoskins, et al. (1994). Software Metrics for Object-Oriented Design. Proceedings. IEEE International Conference on Computer Design, VLSI in Computers and Processors, Los Alamitos, CA, USA, IEEE.

Ince, D. C. and M. J. Shepperd (1989). An empirical and theoretical analysis of an information flow-based system design metric. 2nd European Softw. Eng. Conf., Springer-Verlag.

Ince, D. C. and M. J. Shepperd (1989). Quality control of software designs using cluster analysis. 1st European Software Quality Conf., Vienna, EOQ.

Jeffery, R. and J. Stathis (1993). Specification based software sizing: an empirical investigation of function metrics. NASA Goddard Software Engineering Workshop, Greenbelt, MD, USA.

Jones, C. (1987). A short history of function points and feature points. Technical Paper. Software Productivity Research Inc.

Jones, T. C. (1978). "Measuring programming quality and productivity." IBM Systems Journal 17(1).

Kafura, D. and G. R. Reddy (1987). "The use of software complexity metrics in software maintenance." IEEE Transactions on Software Engineering 13(3): 335-343.

Kemerer, C. F. (1987). "An empirical validation of software cost estimation models." Communications of the ACM 30(5): 416-429.

Kemerer, C. F. (1993). "Reliability of Function Point Measurements: A Field Experiment." Communications of the ACM 36(2).

Kemerer, C. F. and B. S. Porter (1992). "Improving the reliability of function point measurement: an empirical study." IEEE Transactions on Software Engineering 18(11): 1011-1024.

Kitchenham, B., S. L. Pfleeger, et al. (1995). "Towards a framework for software measurement validation." IEEE Transactions on Software Engineering 21(12): 929-944.

Kitchenham, B. A. (1981). "Measures of programming complexity." ICL Technical Journal 2(3): 298-316.

Kitchenham, B. A. (1988). An evaluation of software structure metrics. COMPSAC '88, Chicago, IL., IEEE.

Kitchenham, B. A. and K. Kansala (1993). Inter-item correlations among function points. 1st Intl. Symposium on Software Metrics, Baltimore, MD, IEEE Computer Society Press.

Kitchenham, B. A., S. L. Pfleeger, et al. (1997). "Reply to: Comments on "Towards a Framework for Software Measurement Validation"." IEEE Transactions on Software Engineering 23(3): 189.

Krantz, D. H., R. D. Luce, et al. (1971). Foundations of Measurement. London, Academic Press.

Lee, Y. S., B. S. Liang, et al. (1993). Some Complexity Metrics for Object-Oriented Programs Based on Information Flow. Proceedings, Computers in Design, Manufacturing and Production, Pris-Evry, France, IEEE Computer Society Press.

Lejter, M., S. Meyers, et al. (1992). "Support for Maintaining Object-Oriented Programs." IEEE Transactions on Software Engineering 18(12): 1045-1052.

Li, W. and S. Henry (1993a). "Object-Oriented Metrics that Predict Maintainability." Journal of Systems and Software 23: 111-22.

Li, W. and S. Henry (1993b). Maintenance metrics for the object oriented paradigm. 1st Intl. Software Metrics Symposium, Baltimore, IEEE Computer Society.

Lientz, B. and E. Swanson (1980). Software Maintenance Management. Reading, MA, Addison-Wesley.

Lorenz, M. and J. Kidd (1994). Object-Oriented Software Metrics. New Jersey, Prentice Hall.

Low, G. C. and D. R. Jeffery (1990). "Function points in the estimation and evaluation of the software process." IEEE Transactions on Software Engineering **16**(1): 64-71.

MacDonnell, S. G. (1992). Quantitative Functional Complexity Analysis of Commercial Software Systems. PhD Thesis, Dept. of Engineering, University of Cambridge.

MacDonnell, S. G. (1993). "Deriving relevant functional measures for automated development projects." Information & Software Technology **35**(9): 499-512.

MacDonnell, S. G., M. J. Shepperd, et al. (1997). Metrics for Database Systems: An Empirical Study. Proc. 4th IEEE Intl. Metrics Symp, Alberqueque.

Mancl, D. and W. Havanas *(1990)*. *A Study of the Impact of C++ on* Software Maintenance. Conference on Software Maintenance *1990, San* Diego, CA, USA, IEEE Computer Society Press.

Maus, A. (1992). Entropy as a Complexity Measure and the Optimal Module Size of Object-Oriented Programs. Algorithms, Software, Architecture, Information Processing 1992 (IFIP).

McCabe, T. J. (1976). "A Complexity Measure." IEEE Transactions on Software Engineering 2(4): 308-20.

McCabe, T. J. and C. W. Butler (1989). "Design complexity measurement and testing." Communications of the ACM 32(12): 1415-1425.

Minkiewicz, A. F. (1997). Predictive ObjectPoints, Measuring the Size of OO Applications. ESCOM, Berlin, Germany.

Morasca, S., L. C. Briand, et al. (1997). "Comments on "Towards a Framework for Software Measurement Validation"." IEEE Transactions on Software Engineering 23(3): 187-188.

Moser, S. and O. Nierstrasz (1996). "Measuring the Effects of Object-Oriented Frameworks on Developer Productivity." IEEE Computer 29(9): 45-51.

Ottenstein, L. M. (1979). "Quantitative Estimates of Debugging Requirements." IEEE Transactions on Software Engineering 5(5): 504-514.

Pfanzagl, J. (1968). Theory of Measurement. Wurzburg-Vienna, Physica-Verlag.

Pfleeger, S., R. Jeffery, et al. (1997). "Status Report on Software Measurement." IEEE Software 14(2): 33-43.

Pomberger, G. and W. Pree (1994). Quantitative and Qualitative Aspects of Object-Oriented Software Development. Proceedings ISOOMS '94, Object-Oriented Methodologies and Systems International Symposium, Palermo, Italy, Springer-Verlag, Berlin.

Prather, R. E. (1984). "An Axiomatic Theory of Software Complexity Measure." The Computer Journal 27(4): 340-347.

Pressman, R. (1992). Software Engineering. A Practitioners Approach. McGraw-Hill.

Rajaraman, C. and M. R. Lyu (1992a). Some Coupling Measures for C++ Programs. TOOLS USA '92 (Technology of Object-Oriented Languages and Systems).

Rajaraman, C. and M. R. Lyu (1992b). Reliability and Maintainability Related Software Coupling Metrics in C++ Programs. Third International Symposium on Software Reliability Engineering, IEEE.

Rao, B. (1993). C++ and the Object-Oriented Paradigm. McGraw-Hill.

Rask, R., P. Laamanen, et al. (1993). "Simulation and comparison of Albrecht's Function Point and DeMarco's Function Bang metrics in a CASE environment." IEEE Transactions on Software Engineering 19(7): 661-671.

Rentsch, T. (1982). "Object-Oriented Programming." SIGPLAN Notices 17(9): 51-57.

Roberts, F. S. (1979). Measurement Theory with Applications to Decision Making, Utility and the Social Sciences. Addison-Wesley.

Rombach, H. D. (1987). "A controlled experiment on the impact of software structure on maintainability." IEEE Transactions on Software Engineering 13(3): 344-354.

Rombach, H. D. and V. R. Basili (1990). Practical benefits of goal-oriented measurement. Annual Workshop of the Centre for Software Reliability: Reliability and Measurement, Garmisch-Partenkirchen, Germany, Elsevier.

Rumbaugh, J., M. Blaha, et al. (1991). Object-Oriented Modeling and Design. Prentice-Hall.

Schmidt, H. W. and W. Zimmermann (1994). Reasoning About the Complexity of Object-Oriented Programs. IFIP Transactions on Computer Science and Technology.

Sharble, R. C. and S. S. Cohen (1993). "The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods." Software Engineering Notes 18(2): 60-73.

Shepperd, M. and D. Ince (1993). Derivation and Validation of Software Metrics. Oxford, Oxford University Press.

Shepperd, M. J. (1988). "A Critique of Cyclomatic Complexity as a Software Metric." Software Engineering Journal 3(2): 1-8.

Shepperd, M. J. (1989). A metrics based tool for software design. 2nd Intl. Conf. on Softw. Eng. for Real Time Systems, The Royal Agriculture College, Cirencester, UK, IEE.

Shepperd, M. J. (1990a). "Early life cycle metrics and software quality models." Information & Software Technology 32(4): 311-316.

Shepperd, M. J. (1990b). "An empirical study of design measurement." Software Engineering Journal 5(1): 3-10.

Shepperd, M. J. (1992). Algebraic Models and Metric Validation. Formal Aspects of Measurement. London, Springer-Verlag. 157-175.

Shepperd, M. J. (1994). Some observations on Function Points. Annual. Conf. of the CSR, Invited Talk, Dublin,

Shepperd, M. J. and D. C. Ince (1991). "Design Metrics and Software Maintainability: An Experimental Investigation." Journal of Software Maintenance 3(4): 215-232.

Shepperd, M. J. and R. Turner (1993). Real-Time Function Points: an industrial validation. Proc. European Software Cost Modelling Meeting, Bristol. UK.

Shlaer, S. and S. J. Mellor (1988). Object-Oriented Systems Analysis: Modelling the World in Data. Prentice Hall.

Shlaer, S. and S. J. Mellor (1992). Object Lifecycles: Modelling the World in States. Prentice Hall.

Sneed, H. M. (1995). Estimating the Development Costs of Object-Oriented Software. Evolving Systems. Durham '95. 9th European Workshop on Software Maintenance, Durham, UK, DSM Ltd.

Software Metrics Definition Working Group (1991). Software size measurement with applications to source statement counting. Draft for Review. Software Engineering Institute, Carnegie Mellon.

Stevens, S. S. (1946). "On the theory of scales of measurement." Science **103**: 677-680.

Stevens, W. P., G. J. Myers, et al. (1974). "Structured design." IBM Systems Journal **13**(2): 115-139.

Stoustrup, B. (1997). The C++ Programming Language. Reading, Massachusetts, Addison Wesley.

Symons, C. R. (1988). "Function Point Analysis: Difficulties and Improvements." IEEE Transactions on Software Engineering **14**(1): 2-11.

Symons, C. R. (1991). Software sizing and estimating. Mk II FPA. Chichester, John Wiley.

Tegarden, D. P., S. D. Sheetz, et al. (1992). Effectiveness of Traditional Software Metrics for Object-Oriented Systems. Twenty-Fifth Hawaii International Conference on System Sciences, Hawaii, IEEE.

Troy, D. A. and S. H. Zweben (1981). "Measuring the quality of structured designs." J. of Systems & Software **2**(2): 113-120.

Tukey, J. W. (1977). Exploratory Data Analysis. Philipines, Addison-Wesley.

van Vliet, H. (1993). Software Engineering. Principles and Practice. Chichester, John Wiley.

Verner, J. M., G. Tate, et al. (1989). Technology dependence in function point behaviour. 11th Intl. Conf. on Softw. Eng., IEEE.

Wegner, P. (1990). "Concepts and Paradigms of Object-Oriented Programming." OOPS Messenger **1**(1): 7-87.

Weyuker, E. J. (1988). "Evaluating Software Complexity Measures." IEEE Transactions on Software Engineering **14**(9): 1357 - 1365.

Wilde, N. and R. Huitt (1992). "Maintenance Support for Object-Oriented Programs." IEEE Transactions on Software Engineering **18**(12): 1038-1044.

Wilde, N., P. Matthews, et al. (1993). "Maintaining Object-Oriented Software." IEEE Software **10**(Jan): 75-80.

Wilkie, F. G. and B. Hylands (1998). "Measuring Complexity in C++ Software." Software Practice and Experience 28(5): 513-546.

Wirfs-Brock, R., B. Wilkerson, et al. (1990). Designing Object-Oriented Software. Prentice Hall.

Yau, S. S. and J. S. Collofello (1980). "Some stability measures for software maintenance." IEEE Transactions on Software Engineering 6(6): 545–552.

Yin, B. H. and J. W. Winchester (1978). The establishment and use of measures to evaluate the quality of software designs. ACM Softw. Qual. Ass. Workshop.

Yin, R. K. (1993). Applications of Case Study Research. SAGE Publications.

Yin, R. K. (1994). Case Study Research: design and methods. SAGE Publications.

Zuse, H. (1991). Software Complexity: Measures and Methods. Berlin, Walter de Gruyter.

Zuse, H. (1992). "Properties of Software Measures." Software Quality Journal 1(4): 225-60.

Zuse, H. and P. Bollmann (1989). "Software metrics: using measurement theory to describe the properties and scales of static complexity metrics." ACM SIGPLAN Notices 24(8): 23-33.

Appendix A: Raw data for attributes collected

RWD	ATT -RIB	READS	WRIT -ES	DELS	LOC	DEF -ECT	STATES	NOC	DIT	EVNT	LOC _B	LOc _H
27	14	12	14	1	3213	0	11	0	0	2	2512	701
12	3	8	3	1	2699	0	7	0	0	12	2127	572
4	3	0	3	1	1041	0	2	0	0	0	729	312
6	5	0	5	1	1169	0	3	0	0	2	825	344
55	27	27	27	1	4675	2	15	0	0	10	3852	823
35	17	17	17	1	3655	1	18	0	0	10	2874	781
27	13	13	13	1	3394	0	16	0	0	11	2677	717
47	19	27	19	1	7946	10	46	0	0	31	6632	1314
6	5	0	5	1	1168	0	3	0	0	1	827	341
55	27	27	27	1	4198	0	15	0	0	10	3406	792
0	3	0	0	0	761	0	0	0	0	0	529	232
0	3	0	0	0	754	0	0	0	0	0	514	240
0	4	0	0	0	788	0	0	0	0	0	564	224
2	3	2	0	0	4701	0	9	4	0	12	3988	713
23	10	12	10	1	5181	14	21	0	0	21	4287	894
75	1	35	37	3	4445	26	15	0	2	55	3747	698
131	32	74	56	1	20165	47	114	2	0	122	17177	2988
74	5	32	39	3	5114	26	17	0	2	53	4287	827
114	24	83	31	0	12101	25	60	2	1	71	10320	1781
21	1	11	7	3	4630	6	27	0	0	23	3818	812
34	11	15	16	3	6299	9	35	0	0	33	5220	1079
16	2	9	5	2	1490	2	4	0	1	8	1119	371
17	2	10	5	2	1440	2	4	0	1	8	1058	382
11	3	3	6	2	2161	9	6	0	1	5	1652	509
8	2	1	5	2	1116	2	3	0	1	4	785	331
0	2	0	0	0	730	0	0	0	0	0	511	219
0	1	0	0	0	603	0	0	0	0	1	396	207
35	7	21	11	3	8155	9	33	0	0	26	6897	1258
38	16	16	19	3	6813	10	37	0	0	33	5604	1209
10	6	4	6	0	1940	3	7	2	1	11	1464	476
66	3	27	34	5	5239	27	23	0	2	39	4343	896
74	3	34	35	5	5928	29	26	0	2	43	4942	986

RWD	1.000												
ATTRIB	0.508	1.000											
READS	0.985	0.495	1.000										
WRITES	0.975	0.528	0.944	1.000									
DELS	0.563	-0.121	0.497	0.568	1.000								
LOC	0.848	0.563	0.864	0.800	0.477	1.000							
DEFECT	0.769	0.166	0.764	0.770	0.619	0.759	1.000						
STATES	0.858	0.562	0.866	0.821	0.496	0.968	0.751	1.000					
NOC	0.125	0.258	0.181	0.093	-0.407	0.314	0.182	0.251	1.000				
DIT	0.372	-0.338	0.362	0.383	0.461	0.136	0.580	0.103	0.089	1.000			
EVNT	0.859	0.318	0.876	0.813	0.534	0.910	0.838	0.898	0.341	0.382	1.000		
LOC_B	0.850	0.563	0.866	0.802	0.479	1.000	0.760	0.968	0.314	0.140	0.911	1.000	
LOC_H	0.865	0.620	0.874	0.823	0.476	0.988	0.741	0.982	0.258	0.102	0.882	0.987	1.000
	RWD	ATTRIB	READS	WRITE	DELS	LOC	DEFECT	STATES	NOC	DIT	EVNT	LOC_B	LOC_H

Appendix B: Copy of debriefing questionnaire

Questionnaire

Personal Details

Name:

Qualifications:

Programming Experience:

1. How long into the test did it take you to grasp what was required e.g. after reading instructions, examining the code etc.

2. How much trouble, if any, did you have with the C++ syntax?

3. On a scale of 1 - 10 how difficult would you say the modification was (1 = very easy, 10 = very difficult)

4. What caused you the most difficulty?

5. Overall what action would you say took you the most time to perform i.e. understanding the code, removing syntax errors, editing the changes etc.

6. What approach did you adopt to tackle the modification?

Understanding the code first, then tackling the task?

Tackle task immediately and attempt to understand the code as required.

Cutting and pasting the existing files to meet required specification

Other, please specify

7. Did you use inheritance or not? Explain why

8. If you answered yes to 7, which class did you use as the parent for the class director? Why did you use this class and how long did it take to make this decision?

9. How well did you understand the code?

10. What parts of the code, if any, did you not understand?

11. How would you judge the quality of the code you produced compared to the code you were given?

12. Having performed the modification, would you do anything different next time around? If yes, what?

13. Any other comments?

Appendix B (cont): Summary of Responses to Debriefing Questionnaire

1) Time taken to understand what was required
<=5 mins iiff
10 mins f
30 mins iiff

2) Trouble with syntax?
yes iiif
no iiffff

3) Difficulty of change (1=very easy, 10=very difficult)
1 iff
2 ifff
3 i
4
5 ii

4) What caused the most difficulty?
understanding syntax/language iiff
inheritance i
typos ff
other iif

5) What took up the most time?
editing iffff
debugging/typos iif
understanding code iii
coding f

6) Approach used?
understand first then tackle i
tackle first then understand as necessary iiif
cut and paste to meet spec iiffff

7) Inheritance used?
yes iiiii
no fffff

8) If inheritance used what was the parent class?
thesis iiiii

9) Understanding of code?
good/well ifffff
reasonably well iiii

10) Was any of the code hard to understand?
none iiifffff

some	i
most	i

11) Quality of code added compared with original?

same	iiiiffffff
worse	i

12) What would you do differently if given the chance?

nothing	iiif
use inheritance	ff
not use inheritance	i
understand requirements/code better	if
other	f