ONLY CONNECT

An Investigation into
the Relationship between
Object-Oriented Design Metrics
and the Hacking Culture

Tobias G. Mayer
South Bank University, 1999

"It did not seem so difficult. She need trouble him with no gift of her own. She would only point out the salvation that was latent in his own soul, and in the soul of every man. Only connect! That was the whole of her sermon. Only connect the prose and the passion, and the two will be exalted, and human love will be seen at its highest. Live in fragments no longer. Only connect, and the beast and the monk, robbed of the isolation that life is to either, will die."

E. M. Forster, "Howard's End" (1910)

Only Connect

Author **Tobias Mayer**
tobias@sbu.ac.uk
tobias@risk.freeserve.co.uk

Supervisor **Darren Dalcher**
Forensic Systems Research Group, SCISM
dalched@sbu.ac.uk

Published by **School of Computing, Information Systems & Mathematics** (SCISM)
South Bank University
103 Borough Road
London SE1 0AA, UK

Project no: 198

Document Body text: Times New Roman, 11/14 pt
Titles: Verdana 18-48 pt, sub-titles: Arial 12pt
Created in Microsoft Word 97
Printed on a Canon BJ-30

Copyright © T. G. Mayer & SCISM, 1999

Preface

This piece of work is the end product of a twelve-month research project. The majority of the research was focused on the field of object-oriented software design metrics, with the last four-or-so months incorporating additional research into the 'hacking culture', with the aim of seeking relationships between the two areas and ways in which each might support the other.

The original aim of the research was to perform a critical analysis of current object-oriented design metrics. Following such an analysis, it was intended that a proposal be made for a new set of metrics which would draw on the best of the current work and eliminate many of the existing problems and inconsistencies. The need to critically analyse the existing work became clear during an earlier project I was involved in when it became necessary to identify, and measure, the use of certain OO-specific mechanisms, such as inheritance and encapsulation in a collection of Java programs. A cursory search for a set of widely used OO design metrics to carry out this measurement proved fruitless. The field was less than ten years old and little work of quality had been carried out in that time. Certainly no definitive set of OO design metrics existed. The metrics that did exist appeared to be ill formed and/or rooted in the function-oriented (structured) programming paradigm, and remained largely un-validated. The further this was investigated the more it proved true: the field of OO design metrics was in disarray. Enough work had been done in the field - enough *bad* work - to make the job of performing a critical analysis a long one, but at the same time a complex and fascinating one.

Yet, deep into the critical analysis of the current work the pointlessness of proposing yet another set of metrics dawned on me. Who would use them? Who would even *see* them? Why would they necessarily be any better than were those that existed already? My involvement in the field by this time required that I didn't just settle for this, my first solution but that I gave the issue a lot more thought. Something altogether bigger was required.

At this point I decided to swap my microscope for a wide-angle lens (so to speak). I began looking around at other fields within the software engineering discipline to see if anything happening outside of the field of metrics could generate fresh ideas or potential solutions that could then be applied within the metrics field. The field of software development life cycles held great appeal but rather than investigate the field in a general sense I decided to act instead on the advice of my supervisor and focus on just one life cycle (which is not really a life cycle at all): hacking. The anarchic quality of the hacking culture appealed to me, as did the curious juxtaposition of hacking and metrics. It would be a great challenge to discover relationships between those two fields.

Following my decision I searched for, and read anything to do with hacking. This led me to areas such as 'software creativity', the hacker ethic, 'free software' and Open Source, to name just a few. The more I read on this subject, a subject that was apparently a million miles from my work on OO metrics, the more scope I saw for making connections between the two. There was, indeed *is*, enormous scope for further research. It only remained now to decide how much could realistically be accomplished in the last three months I had left for this work.

The final document is in two parts, which represent the two phases of the research. Part One offers a thorough critical analysis of current object-orientated design metrics and Part Two explores some of the ways that OO metrics and the hacking culture can co-exist.

Background to the material

Some of the material included in this work has previously appeared in slightly different forms. Chapter Three contains most of the paper "A critical analysis of current OO design metrics", which is published in *Software Quality Management VII: Managing Quality* (pp.147-160), Edited by C. Hawkins, G. King, M. Ross, G. Staples, London, British Computer Society, 1999. Some of the ideas therein and further ideas on OO coupling were presented at the associated SQM'99 conference, Southampton Institute, 31 March 1999. Chapter Four contains the paper "Measuring OO systems: a critical analysis of the MOOD metrics", which was presented at IEEE TOOLS Europe'99, Nancy, France, 7-10 June 1999 and appears in the conference proceedings. Some of the ideas and criticisms contained in Chapters One and Two were first presented at the *United Kingdom Software Metrics Association (UKSMA) 10th Anniversary Conference*, London, Oct. 1998.

Readership

This work is intended for software engineers, or persons with an intermediate-level understanding of that field. The first part of the work is more narrowly focused on software metrics and should appeal to those concerned with software design quality and the object-oriented paradigm. In particular its use will be in providing valuable research material to anyone interested in furthering the development of object-oriented design metrics or the theory underlying that discipline. The second part of the work has a more general appeal. It has a wider focus and is more ideas-centered. It may provide useful insights to persons with an interest in studying the creative processes inherent in the activities of software design and development. The somewhat technical nature of the material means that the work will not appeal to a general readership.[†]

Conventions used

The masculine pronouns, *he*, *him*, *his* are used throughout the text when discussing a general third person. No sexism is intended. There are only two genders in the English language, male and female, and thus no neuter pronouns. The decision to use the masculine terms, instead of the feminine terms, or of alternating between the two, (the use of 'their' is simply incorrect) was based partly on the fact that the author is a male, but mostly on fluidity of reading. Unfortunately, it is still jarring to most readers to come across the words *she*, *her*, *hers* in a technical paper. The masculine alternatives read more naturally and the flow of ideas is less likely to be lost.

[†] No steps have been taken, such as the inclusion of a glossary, to accommodate the general reader. It is believed that even with such aids the work is still too overtly technical and discipline-specific to have wider appeal.

Acknowledgements

I would like to thank the following people for the part they played, knowingly or otherwise, in the creation of this work: **Darren Dalcher**, for the initial inspiration to investigate hacking and for his on-going supervision of the work. **Tracy Hall**, for her input on the metrics issues and for recognising my research abilities in the first place (when no one else would) and encouraging me to submit work to professional and academic conferences and journals. **Pete Chalk**, for having the confidence to use my metrics software on his first year Software Engineering Principles unit - more than once! **Peter Winbourne**, for his inspired mathematics teaching and for the confidence he instilled in me in my first term of study, without either of which I would never have understood half the papers I read. A very special thanks and a round of hugs to my wife, **Jennifer** and our two sons **Ty** and **Finn**, for their love, support and tolerance through all of this. And lastly, **Mike Parker** for accepting me onto the BSc Computing Studies course in the first place, despite the fact I applied late and had none of the pre-requisites.

Contents

PART ONE: A Critical Analysis of Object-Oriented Design Metrics

PART TWO: OO Metrics & the Hacking Culture

Part One

A Critical Analysis of Object-Oriented Design Metrics

Introduction to Part One

Since the emergence of object-orientation as a prominent approach to software development it has become widely accepted that conventional software metrics are not adequate to measure object-oriented systems. This can be illustrated by looking at the attribute 'complexity' as measured by McCabe's Cyclomatic Complexity metric [MCCA77]. Although OO systems are no less complex than their structured counterparts the complexity is dictated by data and procedural abstraction and by the mechanisms of polymorphism and encapsulation [TEGA92, TEGA93]; McCabe's measure (the criticisms aside, e.g. [SHEP88]) is clearly inappropriate to measure complexity represented in this way.

The challenge then, is to develop new metrics that are suited to the OO paradigm. Many have already risen to this challenge, most notably Chidamber and Kemerer [CHID91, CHID94], Abreau (the MOOD project) [ABRE93, ABRE94, ABRE95, ABRE96, MOOD98], and Lorenz and Kidd [LORE94, LORE98]. Well over one hundred different metrics have been proposed over the past ten years and new ones are appearing frequently.

This paper represents a deliberate step back from much of the current work in the area of object-orientated design metrics. It is the author's contention that this field of research has moved forward too quickly and too haphazardly for its products, the metrics themselves, to be of much practical use to software engineers and IT managers - the people who should be using them.

Some basic fundamental questions regarding, for instance, the nature of an OO system, the need for paradigm-specific metrics, &c. are not being asked. Furthermore, there is little communication between the different proponents of these metrics, resulting in disagreements, not so much between what should be measured but how it should be measured, and inconsistencies in the terminology used. There is also the sense that the same metrics, with different names, are being defined repeatedly by different proponents. This latter situation is somewhat ironic given the OO paradigm's emphasis on reuse.

It is important to acknowledge that some interesting and important work *has* occurred in the field of object-orientated design measurement, but at the same time it is necessary to recognise that much of the work done so far suffers from three major drawbacks:

i) There is a serious lack of cohesion between the different proponents, resulting in:
   - the absence of generally agreed acceptance criteria
   - inconsistency in the terminology used to describe the measurable attributes
   - different, incompatible sets of metrics

ii) Many of the proposed metrics lack a solid measurement theoretical foundation which means that the resulting values lack consistency and are therefore unreliable.

iii) Much of the work to date is rooted in function-orientated ideas and methodology.

A great deal of time and money has undoubtedly been spent on the development and empirical validation of the various metrics with little in the way of conclusive evidence to demonstrate either the usefulness of any particular suite or of one suite's superiority over another. Much of the validation has been carried out by the proponents themselves, indicating a high degree of subjectivity in the results. It will be shown that such empirical validation, in any case, is often premature due to the majority of the proposed metrics being fundamentally flawed, along the lines described above. With few exceptions these metrics variously fail to meet basic measurement theory requirements, are founded on an inaccurate view of the OO paradigm, lack fineness of granularity or lack clarity of definition making accurate implementation impossible.

There is also evidence to suggest that some of the work is driven by a commercial interest. This of course adds to the lack of consistency and agreement.

This work does not propose any new metrics; instead it offers an analysis and critique of the current state of object-orientated design metrics. It begins by outlining the properties, or mechanisms, that characterize the OO paradigm. A set of basic measurement criteria is then introduced and various observations are made as to how the current OO design metrics fail to meet these criteria. The main goal of this part of the work is to illustrate some of the discrepancies, inconsistencies and misunderstandings that currently exist in the field of object-oriented metrics. In conclusion a recommendation is made for a fresh approach to this subject. It is suggested that current and future proponents of OO design metrics work in a more cohesive and communicative way, building on the existing foundations and sharing new ideas, in order to develop an accurate and industry-wide set of metrics.

This, the first part of this work, contains five chapters. Chapter One provides an overview and history of the field of object-orientated design metrics; Chapter Two introduces a set of validation criteria and offers some general criticisms of the existing work. Chapters Three and Four go on to offer a more in-depth analysis of the Chidamber & Kemerer metrics and the MOOD metrics, respectively. Chapter Five summarizes the material covered in the previous chapters and looks forward.

1 An Overview of OO Design Metrics

This chapter provides an overview of the current state of object-orientated design metrics and offers some general criticisms of the existing work. It is structured in two sections; Section One provides the background and history of the various object-oriented metrics that have been proposed since 1990. Section Two introduces the key design mechanisms of the OO paradigm and highlights some of the ways in which the proponents of OO metrics have misunderstood these mechanisms.

1.1 OO Metrics Background and History

This section begins by reviewing some of the early attempts to measure OO systems. It goes on to look at both the individual metrics and the frameworks that have been specifically developed to measure OO systems. In concludes by summarizing the most recent work that has occurred in this area.

1.1.1 Early work

The earliest set of OO metrics cited in the literature is that developed by Morris in 1989 [MORR89]. The metrics are largely based on the structured design concepts of coupling, cohesion, cyclomatic complexity and 'fan-in'. They are all defined at a system/application level. None were empirically tested. Subsequent work has all but ignored this contribution.

In his pioneering book, "Object-Oriented Design (with Applications)", first published in 1991 Booch suggests five 'meaningful metrics' to test if a class is well designed [BOOC94]:

- ? Coupling
- ? Cohesion
- ? Sufficiency
- ? Completeness
- ? Primitiveness

No kind of formal definitions or formulations are given for these 'metrics', rather they are the aspects of a class which Booch suggests need to be measured. The former two, coupling and cohesion are derived from the concepts used when measuring structured programs, but Booch acknowledges that the terms need *"liberal interpretation"*. In particular he points out the tension between the concepts of inheritance and coupling in that inheritance is considered a useful and desirable aspect of a design while coupling is considered undesirable.

Both sufficiency and completeness are extremely difficult to capture automatically and require human understanding. Sufficiency implies that a class has the minimal interface possible to be meaningful, while completeness implies that the class represents all possible attributes and behaviours of the entity it represents.

Primitiveness concerns the kinds of methods a class has. A primitive method is one that needs knowledge of the underlying representation while a non-primitive method is one that is (or could be) implemented on top of other, primitive, methods. Convenience methods are an example of the latter kind.

Booch does not suggest ways to measure other aspects such as encapsulation or polymorphism, nor does he address the concept of system-wide measures.

A number of other researchers, including Lieberherr *et al* [LIEB88], have made attempts to define good OO design rules based on the concepts of coupling and cohesion, but fall short of defining actual metrics. Some have concentrated their efforts on single OO-specific attributes, e.g. Bieman's work on object/class reuse, [BIEM91], while others have proposed metrics that are applicable only to a single language, e.g. Lake and Cook's complexity metric for C++ [LAKE92].

It is likely that many practitioners resorted to conventional measures (e.g. Lines of Code) to assess their systems but it was not until 1991 that conventional metrics were formally tested on OO systems. Tegarden *et al* used a sample of conventional metrics (namely Halstead's Software Science metrics, McCabe's Cyclomatic Complexity and Lines of Code) to measure the procedural complexity of an OO system [TEGA91]. They conclude that the conventional measures are useful, but in a limited way, and call for additional metrics to fully measure all aspects of an OO system.

1.1.2 Major contributions

In 1991 Chidamber and Kemerer outlined some initial proposals for language-independent metrics in [CHID91]. This work was developed in [CHID94] and a total of six class-level metrics were defined and tested on systems developed in C++ and Smalltalk? . The 1994 paper is one of the first serious attempts to define and empirically validate a set of language-independent, object-orientated design measures. Chidamber & Kemerer acknowledge that the suite may be incomplete and call on other researchers to improve and extend it.

The MOOD project, another important contribution to the field of OO design metrics, began work in 1993 with the introduction of a number of candidate metrics at the method, class and

system levels. Metrics relating to the categories of design, size, complexity, reuse, productivity and quality were suggested at each of the three levels.

Over the following four years the work of the MOOD project progressed with the proposal of eight system-wide design metrics together with a set of evaluation criteria. Each metric is intended to quantify the presence or absence of a certain property and thus has a range of 0 (total absence) to 1 (maximum presence). The MOOD metrics are built on a theoretical/ mathematical foundation and clear language bindings are provided.

In 1994 Lorenz and Kidd published their book 'Object Orientated Software Metrics' [LORE94] in which they define thirty-one design metrics (plus a dozen project metrics). Lorenz and Kidd's work draws on their own project experience using Smalltalk? and C++, but is not validated in any mathematical or theoretical way. Many of the listed metrics (e.g. coupling, cohesion) are not defined, just proposed as candidates. Most others are counts of classes, methods and data fields or averages of these counts across a system or sub-system. The more sophisticated metrics include 'Specialization Index' (SIX), which rates a class as to how well it specializes its superclass, and 'Method Complexity' (MCX), which uses a system of applying weights to different kinds of message-sends to determine the complexity of a method. Lorenz and Kidd's set of metrics are implemented in the Hatteras Inc.? tool, 'OOMetric'.

McCabe and Associates have also defined a set of OO metrics and implemented them in a commercial tool, alongside the traditional McCabe complexity metrics [MCCA94, MCCA98]. However, the definitions of the measures are ambiguous, no formulations are given, and there is no indication as to their usefulness. To confuse the issue still further, two of the McCabe measures have the same names as measures in the Chidamber & Kemerer suite, but different definitions, while another has a similar definition but a different name.

Both McCabe (McCabe & Associates) and Lorenz (Hatteras Software?) have a commercial interest, which may explain the lack of formulations for their respective proposals: one may assume that they do not want to give too much away.

1.1.3 Other contributions

Other contributions to this field include a software complexity model designed by Tegarden, Sheetz and Monarchi, [TEGA93], a proposal for a standard terminology and a data model of the OO architecture suggested by Churcher and Shepperd, [CHUR94] and a framework to deal with OO system coupling by Hitz and Montazeri, [HITZ95].

The Tegarden model is based on the notion of software complexity for structured systems and defines measures of coupling and cohesion at four different design levels: system, object[1], method and variable. It addresses the problem of complexity as dealt with by the mechanisms of inheritance, polymorphism and encapsulation. Although many measures are proposed - and the model incorporates other, existing OO metrics - none are validated in any empirical way.

The framework proposed by Hitz and Montazeri differentiates between object level and class level coupling (interestingly, one of very few works that make this important distinction). A basic - ordinal - metric is proposed which assigns 'weights' to different forms of dependencies between classes (or objects). In some ways this work is a response to the Chidamber & Kemerer coupling measure (CBO) which the authors criticise [HITZ96] for failing to distinguish

[1] The authors use the word 'object' as a synonym for class, i.e. they do not refer to individual instantiations but to the design. This is an example of the lack of differentiation between classes and objects, discussed in Section 2.2.1 (c4); many authors appear to consider them the same thing, when clearly they are not.

between different coupling strengths. The authors also propose a (greatly) improved formulation for the Chidamber & Kemerer LCOM measure, which is finer-grained and closer matches an intuitive understanding of class cohesion.

The Churcher and Shepperd work is based on their argument that it is premature to proceed with the development of specific metrics until an implementation-independent terminology has been established and a satisfactory framework for the validation of such metrics is available. They provide the beginnings of such a terminology and propose a framework in the form of a relational data model. The data model contains information about the entities and relationships that exist in an OO system, the idea being that metrics can be added to the data model in the same way that the other items are, thus ensuring consistency between the metrics and what it is they are measuring.

1.1.4 Recent work

In 1997 Bansiya and Davis defined a large set of OO design metrics [BANS97] which were implemented in their own measurement tool, 'QMOOD'. The tool is C++ specific and is not able to access the implementation of a class, only the interface. The metrics, thirty-one in total are similar to those developed by McCabe and Lorenz & Kidd. More recently the same proponents, together with Etzkorn and Li, proposed a single class cohesion metric, CAMC, [BANS99] which is intended for measuring the cohesion of a class based on the parameter types of the class's methods. It is thus suited for use at the design stage, as it does not depend on the class's implementation.

Some interesting work is being developed by Binkley & Schach [BINK96, BINK97]. Their focus is on producing a paradigm-independent coupling measure. They claim to have proven that there is no significant difference between OO systems and their procedural counterparts, and thus both types of system can be measured with the same tools. Theirs, however, is currently a lone voice.

No other significant new metrics have been proposed in the last two or three years, although existing metrics are being further developed and tested. The Chidamber & Kemerer suite has undergone further empirical tests, e.g. [HARR96, BASI96] and has been analyzed from a theoretical viewpoint [MAYE99, ZUSE94]. The empirical tests are inconclusive whilst the theoretical analyses show the metrics to have a weak theoretical foundation. The MOOD project is also continuing with the definition of a new OO language to standardise code prior to processing it in the MOOD metrics tool. A single complexity metric is also in development based on six of the original eight metrics. The MOOD metrics were independently evaluated in 1998 by Harrison *et al* [HARR98]. Again the results are inconclusive. Lorenz recently updated the original set of metrics published in [LORE94]. The MCX metric has been dropped and replaced by the McCabe Complexity metric. Other additions to the set include three of the Chidamber & Kemerer metrics and a number of new metrics, many of which are specific to Smalltalk? .

No evidence can be found of independent validation of the Lorenz metrics or the McCabe metrics, although metrics from both sets are finding their way into commercial tools developed by independent organisations.

1.2 OO design mechanisms

There are a number of widely agreed properties, or design mechanisms, that characterize the OO paradigm [BOOC94], [BUDD??], [LORE94]. In particular there are five mechanisms that a language would be expected to support if it is to be classed as a true 'object-orientated' language, namely

- ? Inheritance
- ? Message passing (Collaboration)
- ? Encapsulation
- ? Polymorphism
- ? Abstraction

It is worth noting that the third mechanism, encapsulation, is often considered to be synonymous with information-hiding. It will be shown that it is important to recognise information hiding as just one aspect of encapsulation, the other important aspect being that of class unity.

Although all five of the above mechanisms are characteristic of an OO design it is pointed out in [BINK97] that only the first mechanism, inheritance, is unique to the OO paradigm. The other four can all be implemented, to varying degrees, in a non-OO language. While this is true, and certainly worth acknowledging, it is important not to simply dismiss them as 'general programming mechanisms'. Non-OO languages in general do not have specific, i.e. built-in, support for these mechanisms whereas OO languages do. Furthermore it can be argued that all five are essential to an OO design's quality, as to ignore the benefits that the use of these mechanisms supplies is to produce a less than optimal design.

1.2.1 Definition of OO attributes

The majority of current OO metrics are designed to provide measures of the use of these mechanisms or of the attributes that underlie them (e.g. data-hiding, unity and cohesion may be considered as attributes of encapsulation). This is laudable but for one important fact: the mechanisms are poorly understood and the attributes are either inaccurately defined, are defined differently by different proponents or are left completely undefined with an assumption made that their definition is somehow implicit. The following two examples will illustrate this point.

Information hiding and encapsulation

As stated above, information hiding (or data hiding) and encapsulation are not synonymous. Encapsulation is concerned with the packaging of data and behaviour that together represent a single entity. To properly assess the quality of encapsulation requires human understanding, there are however two important aspects of encapsulation that can be assessed automatically:

Class Unity

Data Visibility (information hiding)

Class unity is a measure of whether the class's instance fields are all linked through method access.

Data visibility captures the attribute of data hiding. It measures the extent to which the class's data is hidden from other classes.

A class containing only private data members is not automatically well encapsulated. Likewise, a unified class may be considered un-encapsulated if all or most of its data attributes are public. In order to gauge the encapsulation level of a class it is necessary to assess both attributes. Many so-called encapsulation metrics fail to recognise this and it is usually just the level of data privacy that is considered.

Inheritance and polymorphism

In the definition of the MOOD metrics (see Chapter Four) the view is taken that if a method is overridden then the original method is not inherited. This is not necessarily true. Inheritance and polymorphism are not mutually exclusive. It is certainly true that only the overriding method may be invoked for an object of the class but it is also possible, indeed likely, that the behaviour defined in the original (overridden) method is invoked from within the body of the overriding method; e.g. by a call to 'super'. Thus the behaviour defined in the overridden method is indeed 'inherited'.

In terms of polymorphism the MOOD team make no distinction between methods which specialize (i.e. extend the services of) the overridden method and those which completely change it. The former is considered good OO design while the latter is considered an anomaly to be detected and queried [LORE94]. Simply counting the number of methods that have the same name and parameter list is meaningless.

Another type of polymorphism is 'overloading': defining a method with the same name and return type as an inherited method but with a different parameter set. The question needs to be addressed as to whether or not overloaded methods should be counted as new methods. This is a particularly difficult question in terms of operator overloading.

The concept of polymorphism with its many different variations is cause for much confusion and misunderstanding in the development of OO metrics. The MOOD team members (again!) are confused to the point where they define a metric to count the number of attributes overridden by a class. Such a metric is meaningless. The concept of overriding concerns the behaviour defined in a method. An attribute does not have behaviour, and thus cannot be overridden. While this is an extreme case it illustrates the type of problem that can arise through the misunderstanding or misinterpretation of a fundamental term.

It is impossible to define a metric if no one knows what that metric is supposed to be measuring. To that old adage, "You can't manage what you can't measure" can be added a new one: "You can't measure what you can't manage to define".

---

2 Correctness & Usability

This chapter introduces a set of criteria for validating the correctness and usability of OO design metrics and goes on to describe how these criteria fail to be met by many of the current OO design metrics; examples are given from the most commonly known metrics suites.

2.1 Criteria for validating OO design metrics

The author has defined a set of criteria, drawn from the work of Fenton and Pfleeger [FENT96] and from previous research [MAYE98, 99a, 99b], which is used as a basis for the exploration and analysis of the existing metrics. The criteria are expressed as a set of eight requirements, which are divided into two categories: Correctness and Usability. The requirements state that to be valid an object-orientated design metric should:

Correctness Requirements

c1. be language-independent

c2. be faithful to the OO paradigm

c3. be unambiguously defined

c4. be collectable by static analysis

c5. be firmly rooted in measurement theory

Usability Requirements

u1. be automatable

u2. have intuitive appeal to practitioners[2]

u3. be genuinely useful to practitioners

2.2 Observations and criticisms

In this section each of the requirements introduced above is described in greater detail and a number of critical observations are made on current OO design metrics.

2.2.1 Correctness requirements

c1. Language-independent

A metric that is defined for only a single language will have limited use as comparisons between the effectiveness of different OO languages will not be possible and this in turn will make the usefulness of the metric difficult to evaluate. Within the Hatteras? Inc. suite of metrics there are a number of metrics that are defined specifically for the Smalltalk? language. Clearly, these metrics have a limited appeal. The only time such a metric might be useful is to measure an anomalous attribute, such as the use of 'friend' functions in C++, that is not available in other languages and should not be used in a true OO model.

It is important to define the set of languages that a metric applies to. Obviously, class-based metrics are no use for object-based languages that do not use a class structure.[3] Within the set of applicable languages there will be great variety of syntax, grammar rules and style. Where such differences exist between languages the definition of a metric should clearly state how it is to be applied in the different cases. The only metrics suite that acknowledges this issue is the MOOD set. The MOOD team provides 'language bindings' for Eiffel and C++ with further bindings planned. This greatly improves the likelihood of accurate implementation of the metrics by persons other than the proponents themselves.

c2. Faithful to the OO paradigm

A metric should be concerned with a single attribute, which is clearly defined and specific to the OO paradigm. Structured programming terms should be avoided, or at least seriously considered prior to their adoption.

An important criticism of many of the OO metrics that have been proposed, is that of the terminology used and the underlying thinking that accompanies it. More than one proponent has defined a metric based on the McCabe Cyclomatic Complexity metric[4]. Such a concept of complexity is clearly inappropriate for object-orientated measures due to the complexity often being distributed amongst polymorphic methods, rather than contained in a multi-way case- or if-statement. The same is true of other concepts/terms used in the language of traditional software measurement. Two such terms that play an important role in conventional software design and measurement are coupling and cohesion.

The large majority of current object-orientated metrics are concerned with the attributes of coupling and cohesion. Booch puts them at the top of his list of five "meaningful metrics" for measuring the quality of a class design [BOOC94, p136]. The Chidamber & Kemerer metrics,

[2] I.e. software engineers, project managers and any other person who actually uses the metric.
[3] See [WEGN90] or [BOOC94] for a discussion of the different types of object-based/object-orientated languages.
[4] For example, C&K suggest that developers approach the task of writing a method as they would a traditional program and, that being the case, methods can be measured using conventional complexity measures [CHID94, p.482]

CBO and LCOM are designed specifically to measure coupling and cohesion respectively. Lorenz and Kidd suggest candidate metrics for both attributes [LORE94]. Hitz and Montazeri have produced a substantial piece of work relating to the different forms of coupling in OO systems [HITZ95]. The complexity model for OO systems developed by Tegarden and colleagues [TEGA93] is centred on the measurements of coupling and cohesion at four different levels of a system: system, object, method and variable.

Coupling and cohesion are, in a sense, artificial attributes, devised for structured programming due to the absence of clear relationships between discrete components on the one hand, and the absence of rules relating to the connection of parts contained in a single component on the other hand. OOP has a number of design fundamentals, or mechanisms not present in structured programming that renders these attributes redundant.

However, because of their importance in structured design academics and practitioners interested in devising new metrics for the OO paradigm have attempted to translate the meanings of these concepts to suit the new metrics. The author believes this to be a mistake and one that has caused OO measures to be imprecise, overly complicated and inadequate. On one hand it indicates that many proponents are not recognizing the unique aspects of OO design but are assuming it to be simply an extension of structured programming techniques which can then be measured using conventional metrics such as LOC or Halstead's Software Science. On the other hand it shows lack of understanding of an important principle of measurement: that we should start with an entity and identify the attribute of interest, not start with an attribute and try to make it apply to the entity.

c3. Unambiguously defined

The metrics proposed by Chidamber & Kemerer and MOOD are defined using both natural language and mathematical expressions. The majority of the others are defined in natural language only. The definitions vary as to their level of precision but in almost all cases there is room for ambiguity. The McCabe set is an extreme example of poorly defined metrics. No formulations for the McCabe measures can be found in the literature, and no explanation as to their use in assessing a design. The measure 'Number of Non-Overloaded Calls' is particularly ambiguous. Does it concern an individual class, a hierarchy, or the system as a whole? Should it be measured statically (i.e. potential calls) or dynamically (i.e. actual calls)? And how should the resulting value be used to assess the design? Is a high, or a low, value preferable?[5] No doubt McCabe and Associates have settled most of these questions to their own satisfaction in order to implement the metric in an automated tool but it seems that without buying the tool there is no way of knowing. Certainly the work done by McCabe and Associates is not contributing in any helpful way to the field in general. Such vagueness and/or secrecy, more than just being unhelpful, are potentially damaging to the reputation of software measurement as a science.

There are many cases when the actual entities being measured, largely classes, methods and data-attributes, are ambiguously categorized. An important case in point is that of the categorisation of a method in terms of its belonging to a particular class. Churcher and Shepperd comment on the variety of interpretations open to the idea of counting methods in a class [CHUR95], but, surprisingly, do not consider the issue of constructors. Indeed, the important difference between methods and constructors is scarcely mentioned in any of the literature.

[5] Attempts to contact McCabe & Associates to seek answers to these questions have met with no response.

A constructor is a special kind of method. It is only called when an object of the class is created. A class may define multiple constructors but only one can be called per object and it will be executed only once. A method, on the other hand, can be executed any number of times, and all the methods in a class can potentially be called by a single object of the class. This clearly has significance in terms of testing the class; i.e. classes with multiple constructors are likely to need at least as many objects created in the test harness. It is also significant for all metrics that rely on method counts either directly or indirectly. The decision of whether to include or exclude constructors can make a considerable difference to the resulting values. At the very least, the definition of such a measure should explicitly state whether or not constructors are to be included.

c4. Collectable by static analysis

A design metric should not be concerned with the number of instances of an object in the run-time system, or how often a particular method is called. This is the domain of dynamic analysis and concerns the behaviour of the system rather than the design.

> *"The concepts of a class and an object are tightly interwoven (…) However, there are important differences between these two terms. Whereas an object is a concrete entity that exists in time and space, a class represents only an abstraction, the 'essence' of an object, as it were." (Grady Booch, 1991)*

It seems extraordinary that so few proponents of OO design measures have failed to consider the important differences between measuring classes and measuring objects. In the generally used language of OO design a class is a template from which objects (instances) of the class type can be created [6]. A single class may have any number of instances existing at one time. The class specifies a behaviour - in the form of methods - that is common to all objects of the class, and it defines a data structure - in the form of instance fields that each object fills in with its own values [WEGN90]. Thus, the 'state' of each object will be different from other objects of the same class type, and it is likely to change during the run-time of the system.

A metric should either measure classes or objects, not both. Objects only exist in the run-time system - they are dynamic entities that get constructed and destroyed. This being the case then static analysis is not capable of measuring objects *per se*, only the classes (templates) that underlie the objects.

Metrics such as those that count the number of accesses to a field, or the number of messages sent/received are essentially object metrics. The McCabe PUBDATA metric indicates the number of accesses to a class's public and protected data. At a class level only an approximate number of accesses can be calculated - actual accesses, particularly those defined in conditional blocks or in overridden/overriding methods, may or may not occur, and the number of accesses from within a non-deterministic loop cannot be known.

The Chidamber & Kemerer RFC metric [3.4.5] is concerned with method 'call-back', which, although a potential value can be calculated from the classes in a system, is more interesting, and arguably more useful, when observed from an object-interaction, i.e. dynamic, viewpoint. This is particularly true if the dependencies between classes are measured in some other way.

[6] See [BERA96] for other definitions of a class.

c5. Rooted in measurement theory

A great deal of excellent work has been done in the area of measurement theory: on how it should be applied to software metrics, e.g. [FENT96, FENT92, BAKE90, RUSS91]. There can be little, if any justification for not validating new metrics against these principles. Whether it is through lack of knowledge of this work, lack of interest or lack of understanding is not clear but many of the recent proponents of OO design metrics are ignoring this work. The result is metrics, such as the Chidamber & Kemerer "Weighted Methods per Class" (WMC) metric [3.4.1], the Lorenz & Kidd "Specialization Index" (SIX) metric [2.2.1, c5] and the MOOD Inheritance Factor (MIF and AIF) metrics [4.3.1] that fail to stand up to even the most basic of these principles.

The WMC metric breaks a fundamental rule of measurement theory: that a measure should be concerned with a single attribute [FENT91]. If all method complexities equal one then WMC supplies a count of the methods. Once weighting in the form of a complexity value is added this count is lost. A class with one method that has a complexity value of ten would give the same WMC value as another class with ten methods, each having a complexity value of one. The viewpoints for WMC are largely concerned with method count, so the addition of a complexity value seems an unnecessary complication. It is worth noting that the McCabe measure of the same name is defined as a count of all methods defined in a class (in which case the word 'weighted' is superfluous).

The SIX, MIF and AIF metrics do not satisfy the 'representation condition' of measurement theory, which requires that the relationships between the entities being measured map directly to the numbers produced by measuring those entities. For example, if entity A is twice as big as entity B and the metric for 'size of A' is 6 then the metric for 'size of B' must be 3; if it is not, then the metric scale does not accurately capture the attribute in question, in this case size.

An extended description of the SIX metric, with examples, will illustrate why the 'representation condition' is so important.

The Lorenz & Kidd " Specialization Index" (SIX) metric

SIX is defined for a single class as:

> **(*nesting level * method overrides*) / *total methods***

where the terms 'nesting level', 'method overrides' and 'total methods' are defined as:

nesting level: the depth of the class in its hierarchy

method overrides:

> *"The number of methods defined in a class that are also defined in one or more superclasses. Methods that invoke the superclass' method or override template methods are not included."* [LORE94]

total methods: all methods defined in the class itself (i.e. excludes inherited methods).

The justification for this measure is given as:

"The farther down in the class hierarchy you go, the more specialized the (sub)class should be. Therefore, overridden methods bear more weight toward reaching an anomaly threshold than for a class higher in the inheritance hierarchy." [LORE94]

The anomaly threshold is given as 15% which is based on the maximum number of methods defined by a class being 20 and the maximum number of overridden methods in a class being three, both at a nesting level of one, i.e. (1*3)/20 = 0.15

This seems reasonable and logical but closer examination of some 'real' situations shows this measure to be so coarsely grained and inconsistent as to be useless.

Situation 1

Class A is a root level (nesting level=0) class which defines 3 non-abstract methods, i(), j() and k().

Class B is derived from A. B overrides i(), j() and k(), without invoking the respective parent methods, and defines a further 20 methods.

Clearly B is in no way a specialization of A as all inherited methods have had their behaviour changed by B, without invoking the superclass' methods. The SIX value for B is 1*3/23 = 0.1304 (or 13%) which is below the anomaly threshold, implying that B is a specialization of A.

Situation 2

Class C is derived from class X and is at level 5 in the hierarchy. C inherits some 30+ methods from its superclass/es and defines 4 additional methods, all of which override inherited methods. 3 of these methods invoke the superclass' method. Such overrides do not count in the 'method override' measure (see above). The fourth override defines a new behaviour for the method and so does count as an overridden method. Intuitively, it would appear that C is indeed a specialization of X: C extends the capabilities of the superclass/es and changes only a single behaviour. However, the SIX value for C is (5*1)/4 = 1.25 (or 125%), so far from the anomaly threshold as to be outside the accepted 'percentage' range, implying that C is nowhere close to being a specialization of X. The large SIX value indicates that many changes need to be made to C in order to make it a specialization of X. Again we find this to be counter-intuative; by removing the single overriding method or by changing it to include a call to the superclass' method the SIX value becomes (5*0)/3 [or (5*0)/4] = 0, in other words, a perfect specialization.

The above situations illustrate the importance of validating a measure using measurement theory, in particular testing the representation condition. In the case of the measure SIX the empirical relation system does not map correctly to the formal (numerical) relation system. By our intuitive, common sense understanding class B (above) is less of a specialization than is class C. Lack of quality is what is being measured here so this relationship can be stated formally as B > C (B has a greater lack of quality than C does). When this is mapped to the numerical relation system the semantics of the empirical relationship should be preserved, however, we find that (μ)B = 13% and (μ)C = 125%; in other words, (μ)B < (μ)C.

(Note: If this was always the case it could be said that the empirical system maps inversely to the numerical system, but in the case of SIX cases can certainly be found where this does not hold.)

If it is an agreed design principle that overriding a method (for convenience) is more incorrect for classes deeper in the hierarchy (and no other evidence exists to support this view) then it would be enough for the SIX measure to simply multiply the number of overrides by the 'nesting level' value. Lorenz & Kidd state that a maximum of three overrides is acceptable at nesting level =1, with less overrides the deeper you go. If the SIX anomaly value is set at >3 then this new formula would allow 1-2 overrides at level 2, 1 override at level 3 and zero (or <1) overrides after that. Dividing by the number of methods adds nothing to this measure; in fact, it greatly reduces its accuracy.

Validation using measurement theory must now be a minimum requirement for any new metric in order to avoid a continuation of the haphazard approach that has tainted software measurement in the past.

2.2.2 Usability requirements

u1. Automatable

There are many potentially useful attributes that can be measured, but unless the measurement is automatable the data will not be collected. Manual data collection is tedious and prone to inaccuracies. It must be possible to collect the data required for a measure in an automated way, otherwise it is unlikely that the measure will be of practical use.

It is equally important that the metric's definition is not compromised as a result of automation. The Chidamber & Kemerer "Response For a Class" (RFC) metric [3.4.5] aims to count (for a class) all methods that can potentially be executed in response to a message received by an object of that class. Due to the practical considerations of automating this metric to collect the required data Chidamber & Kemerer recommend only counting the methods declared in the class itself plus the methods called directly from within the class. The original definition for the metric then becomes meaningless.

u2. Intuitive appeal

An important usability issue is that of 'user appeal'. A measure must have a degree of intuitive appeal; i.e. it must be meaningful and understandable to those who will use it. If the metric's definition is overly complicated to the point where only a mathematician or a philosopher can understand it the chances of it being put to wide use are slim. The MOOD metrics, with their overly-complex mathematical formulations, verge on the unintuitive, which may explain why they have not stirred up much interest in the industry.

u3. Genuinely useful

A measure should aid designers, developers and project managers and provide positive benefits to a company. It may not always be possible to determine this until the measure has been tested, perhaps alongside other measures that complement or support it, but the reason for using a measure must always be known beforehand. Don't measure just for the sake of measuring.

The usefulness of some of the existing OO measures is dubious. McCabe defines a polymorphism measure called PCTCALL, which is defined as "the number of non-overloaded calls in a system". No explanation is given as to why this measure is useful or how the value derived from this measure can be used. There are any number of things that can be measured in an OO design, and many are very easy to collect the data for, but unless the results have some meaning to practitioners they are useless.

This chapter has introduced a set of requirements for ensuring the correctness and usability of OO metrics. Examples were given from some of the better-known metrics suites to illustrate how these requirements are, in general, not being met. Although not applied in such a systematic way, the same set of requirements underlie the more in-depth evaluations performed on the Chidamber & Kemerer metrics and the MOOD metrics in the following two chapters.

3 Analysis and Critique of the Chidamber & Kemerer Metrics

3.1 Introduction

Probably the best known work in the field of OO design metrics is that done by Chidamber and Kemerer (C&K) at MIT. Some initial proposals for language-independent OO design metrics were outlined in [CHID91]. The ideas were expanded in [CHID94] and a suite of six metrics was presented. The C&K work represents one of the most thorough treatments of the subject at the current time. However, the authors explicitly state,

> *"…there is no reason to believe that the proposed metrics will be found to be comprehensive, and further work could result in additions, changes and possible deletions from this suite".*

C&K attempted to establish a firm foundation for their metrics by adopting the ontology of Bunge [BUNG77, BUNG79] as a theoretical basis, by evaluating each of the metrics using a set of axioms proposed by E. Weyuker [WEYU88] and by empirically testing the metrics on professional systems developed in C++ and Smalltalk? . Despite this attempt to produce theoretically sound metrics the suite has come under some criticism from the academic community. Zuse [ZUSE94] criticises the metrics on the grounds that they have no additive property related to the concatenation operation (i.e. the combination of two classes), and thus cannot be founded on a ratio scale. Other important criticisms expose the lack of a clear terminology and language binding [CHUR95] and a number of inadequacies in the meaningfulness of the metrics and in the method of evaluation used [HITZ96].

Despite the criticisms, and with little further empirical or theoretical evaluation, the C&K metrics have been incorporated into a number of software measurement tools and look set to become industry standards. Like McCabe's Cyclomatic Complexity metric [MCCA76] (also criticised for being based on a poor theoretical foundation [SHEP88]) the C&K metrics appear to have an intuitive appeal to software engineers.

Such intuitive appeal is a necessary criterion, but is by no means sufficient to warrant the use of a metric. As suggested in the previous chapter, it is only one of eight basic requirements that an OO metric is expected to meet. The following analysis indicates that the C&K metrics do not satisfy all eight of these requirements; their use as quality indicators is thus in doubt.

As well as providing a critique of the C&K metrics this chapter also serves a more constructive purpose: to make other researchers and practitioners aware of some of the problems that may arise from using these measures. As a by-product, the axioms of E. Weyuker [WEYU88] come under scrutiny in terms of their applicability to object-orientated metrics.

The remainder of this chapter is structured as follows: Part Two briefly outlines the foundations on which the metrics suite was built; Part Three addresses issues relating to the use of Weyuker's axioms as a validation criterion, in particular the issues of complexity and class combination are considered. Part Four is a detailed analysis of each of the six metrics; Part Five offers an example of inaccurate data interpretation and Part Six offers some concluding remarks.

3.2 Foundations

3.2.1 Theoretical Foundation

The concept of objects used by C&K is founded on the ontological principles proposed by M. Bunge in his “Treatise on Basic Philosophy” [BUNG77, BUNG79]. This follows from the work of other researchers, including Y. Wand, e.g. [WAND89] and R. Weber, e.g. [WEBE91], who have found Bunge's generalised principles to be ideally suited to the OO paradigm.

3.2.2 Theoretical Validation

Each of the metrics is validated against the following sub-set of Weyuker’s axioms for software complexity measures, lexically adapted to apply to the OO paradigm.

Noncoarseness: ? *P* and ? *Q*, such that ? (*P*) ? ? (*Q*)

Not every class can have the same metric.

Nonuniqueness: ? P and ? Q, such that ? (P) = ? (Q)

Two classes can be equally complex.

Design Details are Important: (functionality of P) = (functionality of Q) /? ? (P) = ? (Q)

Two classes providing the same functionality will not necessarily have the same metric.

Monotonicity: ? (P and Q) (?(P) ? ?(P+Q) and (?(Q) ? ?(P+Q)

The metric for two classes in combination can never be less than the metric for either of the component classes.

Nonequivalence of Interaction: ? P, ? Q, ? R, such that ? (P) = ? (Q) /? ?(P+R) = ?(Q+R)

If two classes have the same metric it does not imply that they will have the same metric when each is combined with a third class.

Interaction Increases Complexity: ? P and ? Q, such that ? (P) + ? (Q) < ?(P+Q)

> When two classes are combined the joint metric can be greater than the sum of the two individual metrics.

The authors acknowledge that this rule set has received a number of criticisms, e.g. [FENT91, CHER91] but they chose to use it because it *"is a widely known formal analytical approach"*.

3.2.3 Empirical Validation

To ensure the usefulness of the metrics C&K incorporate the experiences of professional software developers, which are expressed as 'viewpoints' for each metric. These viewpoints are considered when interpreting the results produced by the empirical tests.

3.3 The Use of Weyuker's Axioms as a Validation Criterion

3.3.1 Measuring Complexity

There is a widespread misconception in the software measurement field that any design metric is a complexity metric. By validating their metrics against Weyuker's axioms - which are designed specifically for evaluating software complexity measures - C&K are implicitly stating that all the metrics in the suite are complexity metrics. However, C&K do not define complexity and it appears to have a different definition for each metric.

Intuitively, complexity (by any definition) is thought of as undesirable in a program. In measuring the complexity of an algorithm a high complexity value is clearly undesirable and it would be a goal of the designer or programmer to simplify the algorithm or split it into smaller units, each with a lower complexity value. In keeping with this intuitive view C&K organise their metrics such that a low value implies low complexity, and *visa-versa*. But, a low 'complexity' value for some of these metrics is not necessarily desirable, for instance it would not be preferable for all classes - or even most classes - in a system to have a DIT (depth of inheritance) value of 0 (i.e. be root classes). It can be argued that classes with a high DIT value, whether it is desirable or not, are indeed more 'complex' as it is harder to understand and maintain a class that inherits data and behaviour from other classes. This same argument however does not hold for the NOC (number of children) metric. It is again undesirable for all classes in a system to have a NOC value of 0, but regardless of the number of children a class has its 'complexity' remains constant. There is no way of knowing how many children a class has by examining the class itself. The NOC value will change as more child classes are added, but there is no internal change to the class itself. Furthermore, in different systems the same (re-used) class will have different values for NOC. It is illogical and unintuitive that the complexity of a thing can change if the thing itself remains the same.

A design that makes good use of the inheritance mechanism, although perhaps appearing more complex on the surface (due to the high metric values), would probably be less complex than a design where the majority of classes were root classes that had few or no children. In the latter case it is likely that there would be code duplication, a greater need for multi-way case statements and generally larger, non-unified classes.

3.3.2 Combination of Classes

The fourth, fifth and sixth of Weyuker's properties are concerned with the combination of two classes. To evaluate the DIT metric against property four, 'Monotonicity' (i.e. 'The metric for two classes in combination can never be less than the metric for either of the component classes') C&K state three possible relationships between two classes, P and Q.

i) P and Q are siblings,

ii) Q is a descendant of P

iii) P and Q are neither siblings nor descendants of each other.

In case ii) the DIT metric fails to satisfy property four. However the concern here is not whether or not the property is satisfied, rather it is with the whole concept of 'class combination'.

The second relationship (above) can be further classified as:

a) Q is an *immediate* descendant of P

b) Q is a *non-immediate* descendant of P

C&K do not give adequate consideration to this latter relationship, which may explain why the problems of combination described below were overlooked.

Combining two classes in a non-descendant, non-sibling relationship

In order to combine the two classes P and Q in case iii) C&K change the hierarchical structure from a single inheritance structure to a multiple inheritance structure. Putting aside the fact that many OO languages do not support multiple inheritance, this re-structuring seems unacceptable as it alters the entire design pattern and adds to the overall complexity of the system. In order to avoid it, other combinations need to be made to the structure (see figures 3a-3c, below).

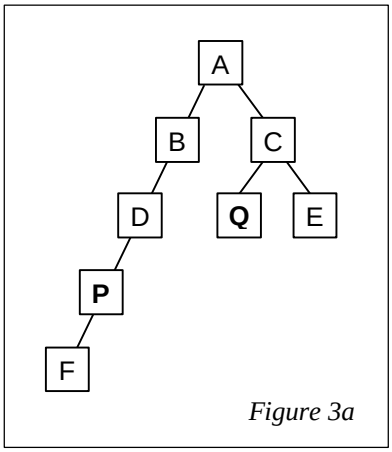


Figure 3a

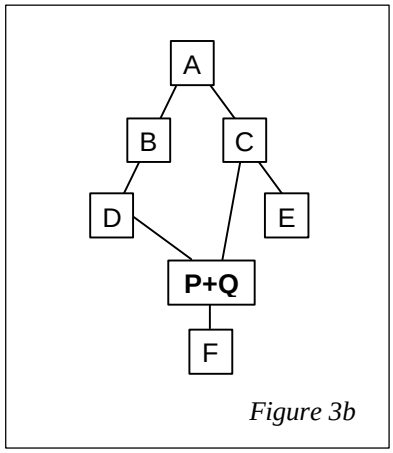


Figure 3b

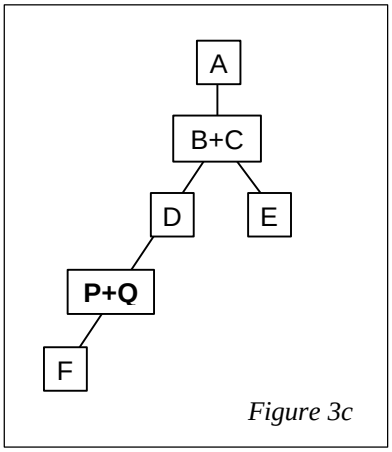


Figure 3c

Figure 3a shows a hierarchy with classes P and Q in a non-sibling/non-descendant relationship.

Figure 3b shows the C&K method of combining P and Q by changing the structure to one of multiple inheritance. The new class (P+Q) now has two parent classes: D and C.

Figure 3c shows the combination of P and Q whilst retaining the single inheritance structure. Note that classes B and C also need to be combined to achieve this.

Combining Two Classes in a Non-immediate Descendant Relationship

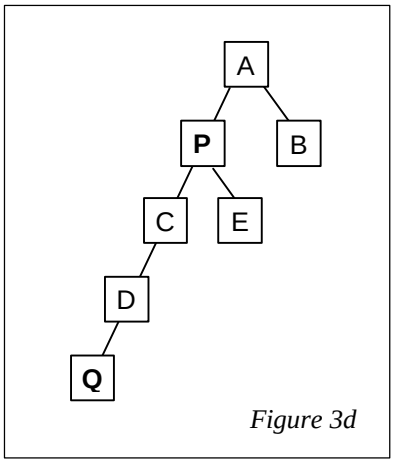


Figure 3d

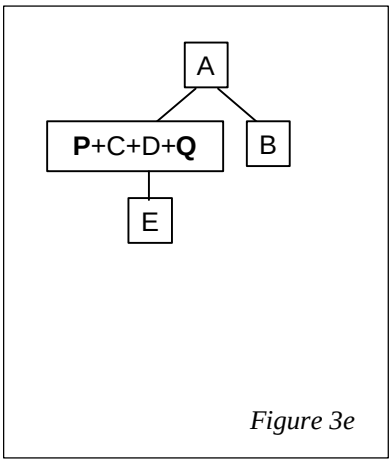


Figure 3e

Figure 3d shows P and Q in a (non-immediate) 'descendant-of' relationship.

Figure 3e shows the only way of combining P and Q without either class (or any other class in the hierarchy) losing its inherited properties. Moving Q up to P's position would result in Q losing the properties inherited from C and D. Moving P down to Q's position would result in C and D losing the properties inherited from P. (C&K offer no alternative way of combining two classes in this kind of relationship.)

Conditions for the Combination of Two Classes in a Hierarchy

Two conditions for the combination of two classes then emerge.

1. No class in the hierarchy should lose its inherited properties as a result of the combination.
2. A single inheritance structure should remain as such after the combination.

The above examples illustrate how combination in terms of classes has very different implications than it does for traditional (i.e. function-orientated design) measures. It may be that combination is not a valid operation between some classes, e.g. in the examples given above where it is necessary to combine more than the two required classes into one class or to make additional combinations within the hierarchy. If so, this renders Weyuker's fourth, fifth and sixth properties invalid and raises doubts as to the suitability of using properties designed to evaluate function-orientated metrics on class/object-orientated metrics.

3.3.3 Invalid Validation

The use of Weyuker's properties as a method of validation produces further inconsistencies. Weyuker's sixth property, 'Interaction Increases Complexity states that when two components, *A* and *B*, are combined the resulting complexity of the new component *AB* can be greater than the sum of the two separate components. The intention of this property is to ensure that dividing the problem space into a set of smaller, more manageable problem spaces can reduce the overall complexity of the system. This view is supported by the fourth property, 'Monotonicity', which states that the metric for two classes in combination can never be *less* than the metric for either of the component classes. It is cause for concern that none of the C&K metrics satisfies property six, and two of them, DIT and LCOM, fail to satisfy property four. The implication of this is that class designs become *less* complex the more you combine the classes, resulting -

ultimately - in a design with just one class. In terms of OO design this is clearly ludicrous (although in their summary of the analytical results C&K present an argument against this, but fail to take the implication to its logical conclusion).

Rather than concluding that the metrics themselves are inadequate it seems more likely that either the rule set is inappropriate for validating object-orientated metrics (the conclusion reached in section 3.2) or that the C&K metrics are not measuring complexity - by the definition(s) intended by Weyuker.

3.3.4 A Note of Caution

Until the term 'complexity' is more precisely defined it seems sensible to avoid it as a 'catch-all' attribute to measure, and define each individual measure in terms of what it is *actually* measuring. For NOC, (as an example) this would be 'influence on system' where a value of 0 would imply 'no influence' and higher values would imply increasingly more influence. As for Weyuker's axioms, until the criticisms (referenced above) are addressed - and possibly even after that - they are probably best left in the realm for which they were devised, namely function-orientated design, and even then only used as a supplement to the validation principles of measurement theory.

3.4 The Six Metrics

3.4.1 WMC Weighted Methods per Class

$$WMC = \sum_{i=1}^{n} c_i$$

where $c_1 \ldots c_n$ is the complexity of methods $M_1 \ldots M_n$ of a given class. If all method complexities are equal to 1 then WMC = n, the number of methods.

WMC breaks a fundamental rule of measurement theory: that a measure should be concerned with a single attribute [FENT91]. If all method complexities equal one then WMC supplies a count of the methods. Once weighting, in the form of a complexity value, is added this count is lost. A class with one method that has a complexity value of ten would give the same WMC value as another class with ten methods, each having a complexity value of one. The viewpoints for WMC are largely concerned with method count, so the addition of a complexity value seems an unnecessary complication. It is worth noting that the McCabe measure of the same name is defined as a count of all methods defined in a class, in which case the word 'weighted' is superfluous.

3.4.2 DIT Depth of Inheritance Tree

DIT = ?C?-1
where C = set of all classes in the linear branch from *c* to the root.
In the case of multiple inheritance DIT = max(?C1?…?Cn?) -1.
In a single inheritance structure DIT for a class *c*, is equivalent to the count of all classes *c* inherits from.

Two of the viewpoints for this metric refer to increased complexity due to more methods being inherited and more classes being involved, the third is concerned with potential reuse of

inherited methods. DIT is a measure of how many ancestor classes affect the measured class so it creates an inconsistency when the definition of this metric in a multiple inheritance structure is given as DIT = $max(?C_1? \ldots ?C_n?)$ -1, in other words the longest path from class *c* to the root class. This would not indicate the number of classes involved or the number of methods inherited, e.g. a single class may have a DIT value of two but have half a dozen (or more) ancestor classes. Excessive use of multiple inheritance is generally discouraged in OO design so an inheritance measure should be able to detect this anomaly.

3.4.3 NOC Number of Children

NOC = ?*C*?
where *C* = set of all classes which immediately extend a class *c*.

The NOC metric is concerned with potential reuse through inheritance but in the same way that DIT fails to give the full picture of number of ancestor classes the NOC metric fails to count all the descendants of a class. Only counting immediate subclasses may give a distorted view of the system, as the following example illustrates:

Example 3.4.3a

A hierarchy is structured thus: Class A is the root class, class B extends A, class C extends B and classes D, E, F, G and H extend C. Both A and B have a NOC value of one, but a total of seven classes inherit A's properties and a total of six inherit B's properties. As the hierarchy grows bigger so, potentially, does the discrepancy.

The definition of NOC does not satisfy viewpoint 1 (reuse) or viewpoint 3 (influence on the design) [CHID94, p.485], both of which support the idea of counting all of a class's descendants.

Both DIT and NOC are useful measures but are not sufficient to fully assess the quality of a hierarchy or to satisfy the respective viewpoints. Additional measures are required to count all of a class's ancestors (i.e. in a multiple inheritance system) and all of its descendants.

3.4.4 CBO Coupling Between Object Classes

CBO = ?*C*? where *C* = set of all classes to which a class *c* is coupled.
where 'coupled' is defined as: two classes are considered coupled when a method of one references a method or field of the other. CBO is a count of all such references for a single class.

C&K state that

> *"…two objects are coupled when methods declared in one class use methods or instance variables of the other class."*

Coupling is therefore a property of pairs of classes, but in terms of CBO it is defined for a single class as

> *"CBO for a class is a count of the other classes to which it is coupled".*

This does not clearly indicate the direction of the coupling, whether suppliers or clients (or both) should be counted. The viewpoints [CHID94, p.486] however, indicate that CBO is a count of suppliers. C&K further state that their definition of coupling also applies to

> *"coupling due to inheritance",*

but do not make it clear if all ancestors are automatically coupled to the measured class, simply because they are ancestors, or if the measured class has to explicitly access a field or method in

an ancestor class for it to count. In general, the definition for this metric is ambiguous, which makes its application impossible for anyone other than the original proponents.

Hitz and Montazeri criticise the CBO metric in [HITZ96] for failing to distinguish between different coupling strengths, claiming that the empirical relation system established by C&K for the attribute of coupling is insufficient[7]. A similar criticism is given by Binkley and Schach in [BINK96]. Both papers suggest that a single couple should contribute more or less to an overall coupling metric, depending on circumstances such as the number of parameters passed, what is being accessed (field or method) and whether the class is an ancestor or not.

As mentioned in chapter 2, the term 'coupling' is perhaps not an appropriate one to use in describing an OO design, where the relationships between components (which are classes) are more clearly defined than in structured programming.

3.4.5 RFC Response For a Class

RFC = ?RS? where RS = {M} ? all i {R i}
and where {M} = set of all methods of a class and {R i} = set of all methods recursively called by method i.

RFC aims to count (for a class) all methods that can potentially be executed in response to a message received by an object of that class. Due to the practical considerations of collecting this data C&K recommend that only one level of nesting should be considered i.e. should only count the methods declared in a class plus those methods called from within the class's methods. This gives a distorted picture as a single method call often has deeply nested 'call-back'. If C&K intend the metric to be a measure of the methods in a class plus the methods called then the definition should be re-defined to reflect this.

As all methods must be defined within a class (RFC ignores functions such as 'printf' in C++) a count of all supplier classes (i.e. direct and indirect suppliers) would perform a similar function and would satisfy the viewpoints [CHID94, p.487] for this metric. It would also be an easier measure to collect data for.

3.4.6 LCOM Lack of Cohesion in Methods

if ?P?>?Q? LCOM = ?P???Q?,
else LCOM = 0
P = set of all method pairs of class c with zero similarity,
Q = set of all method pairs of class c with non-zero similarity,
where similarity of a pair of methods is defined as the number of instance variables that are referenced by both methods.

LCOM is defined 'in reverse' to be in keeping with C&K's concept of a low value being less complex and *visa-versa*. Hitz & Montazeri comprehensively cover the weaknesses of this metric in [HITZ95, HITZ96], a critique with which this author fully concurs, and propose a new formulation for LCOM, restated here in non-mathematical terms.

LCOM for a single class is a count of 'method clusters', where any method in a cluster must access at least one instance variable (directly, or via an access method) that is also accessed by at least one other method in the cluster. The number of clusters, c, in a class will be 1<=c<=m, where m = the number of methods in a class.

[7] Hitz and Montazeri's own framework for class and object coupling is described in [HITZ95].

The name (which is confusing anyway) is now a misnomer; what the Hitz and Montazeri version of LCOM is actually measuring is an important aspect of encapsulation, namely the *unity* of a class. The importance of focusing on OO-specific attributes, rather than those attributes that have been found to be useful for function-orientated systems was covered in Chapter 1.

3.5 Interpretation of Data

In the interpretation of the data taken for the NOC metric C&K comment on the high number of classes (73% at one site and 68% at the other) having NOC=0 and suggest this indicates poor use of inheritance. What does not seem to be taken into consideration is that this percentage will almost always be large for any design that does make use of inheritance. For instance, a structure where every class (except the leaf-node classes) has two children will result in >50% of classes where NOC=0 and a structure where every class (except the leaf-node classes) has three children will result in >66% of classes having no children. This percentage grows the more children the childbearing classes have. In general, the formula for calculating % of classes where NOC=0 for regular inheritance trees is:

$$c^n \; / \; \sum_{i=0}^{n} c^i$$

where c = number of children per class and n = depth of nesting; e.g. for a hierarchy with a maximum depth of 4 where all classes have 2 children:

$2^4 / (2^0 + 2^1 + 2^2 + 2^3 + 2^4) = 16/31 = 0.516$ or 51.6%

The times a design will have less than 50% of classes with NOC=0 will be in cases such as figure 6 which does not indicate particularly effective use of inheritance.

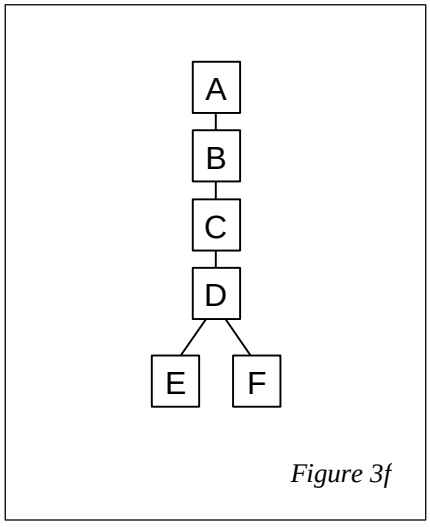


Figure 3f

This example illustrates the caution that needs to be observed when interpreting the data produced by a metric, particularly with a new (i.e. object-orientated) metric that we are not familiar with. Inaccurate interpretation can result in the wrong actions being taken to 'fix' the

problem. This in turn results in wasted person-hours and, of course, costs for the organisation involved.

3.6 Concluding Remarks

In general the C&K metrics suffer from unclear definitions and a failure to capture OO-specific attributes. The attributes of data-hiding, polymorphism and abstraction are not measured at all and the attributes of inheritance and encapsulation are only partially measured. The focus on coupling and cohesion (discussed in Chapter 2) and on 'complexity' restricts the usefulness of these measures and the use of Weyuker's axioms limits their validity.

As a by-product of this analysis Weyuker's axioms are shown to be inadequate to validate object-orientated metrics.

In the following chapter an equally detailed analysis is carried out on the metrics proposed by the MOOD project [ABRE93, ABRE94, ABRE95, ABRE96].

4 Analysis and Critique of the MOOD Metrics

4.1 Introduction

Arguably, the most thorough treatment of the subject of OO design metrics is that of the MOOD team, under the leadership of Abreau [ABRE94, ABRE95, ABRE96, MOOD98].

The MOOD metrics set was first proposed in 1994 by the MOOD project team, headed by Abreau [ARBR94]. The metrics are designed to meet a particular set of criteria, also proposed by the team. The criteria are listed here:

1. Metrics determination should be formally defined.
2. Non-size metrics should be system size independent.
3. Metrics should be dimensionless or expressed in some consistent unit system.
4. Metrics should be obtainable early in the life-cycle.
5. Metrics should be down-scaleable.
6. Metrics should be easily computable.
7. Metrics should be language independent.

The 1994 paper criticises the metrics of Chidamber & Kemerer [CHID 94], Bieman [BIEM92] and others for failing to fulfill these criteria, in particular 1, 2, 3 and 7, but offer no evidence to support the criticism.

The MOOD metrics are all system-wide measurements. Each metric is expressed as a quotient where the numerator is the actual use of a particular mechanism in the system being measured and the nominator is the maximum possible use of the same mechanism. The value for each metric will therefore be in the range 0-1. The definition for each metric is given as a mathematical expression. The original set [ABRE94] deals with the attributes of inheritance, visibility, coupling, clustering, polymorphism and reuse.

Since its inception the set has undergone a number of changes: the clustering and reuse metrics have been dropped and the definitions of others have been revised [ABRE95, ABRE96]. Language mappings (referred to as 'bindings') to C++ and Eiffel have been specified, with further bindings being planned. A great deal of time and effort has been spent in empirically validating these metrics, for the most part by the MOOD team themselves; there is evidence of only one independent validation [HARR98].

The MOOD metrics are evaluated here on a theoretical level and it is shown that any empirical validation is premature due to the majority of the MOOD metrics being fundamentally flawed. The metrics either fail to meet the MOOD team's own criteria or are founded on an imprecise, and in certain cases inaccurate, view of the OO paradigm.

Solutions to some of these anomalies are proposed and some important aspects of the OO paradigm are clarified, in particular those aspects that may cause difficulties when attempting to define accurate and meaningful design metrics. The suggestions made are not limited to the MOOD metrics but are intended to have a wider applicability in the field of OO metrics.

In conclusion the observation is made that the usefulness of OO metrics depends to a large extent on the quality of the tools that are built to collect them. Such tools cannot be accurately built until the metrics themselves have precise, unambiguous and meaningful definitions. It is recommended that further work be put into improving and testing the MOOD metrics before attempting to apply them to real life systems.

The remainder of this chapter is structured thus: Section Two describes the class-level metrics used by the MOOD team to build the six system-level metrics that comprise the set. A closer look at these class-level metrics serves to introduce the MOOD team's incomplete and (on occasion) inaccurate understanding of the OO paradigm. This is explored further in terms of the attributes measured, and then, in Section Three, in terms of the metrics themselves. Section Four offers some concluding remarks.

4.2 The OO paradigm

This section outlines the class-level metrics on which the MOOD set was built. It is then shown that the definitions provided indicate an overly simplistic view of the OO paradigm and include a basic misunderstanding of some important OO principles.

4.2.1 Class level metrics

Prior to defining the main set of metrics, which are all system-wide measurements, a number of basic class-level metrics are first defined in [ARBR94]. These are given as functions where the argument is the class being measured (C_i) and the return value is the value for the corresponding property. In order to clearly discuss and analyse the MOOD metrics it is necessary to repeat these class-level metrics here.

- Children Count: **CC(C_i)** - immediate sub-classes
- Descendants Count: **DC(C_i)** - all sub-classes

- Parents Count: **PC(C_i)** - immediate super-classes
- Ancestors Count: **AC(C_i)** - all super-classes
- M_d **(C_i)** - number of methods defined in C_i
- M_n **(C_i)** - number of new methods defined in C_i
- M_i **(C_i)** - number of methods inherited (and not overridden) by C_i
- M_o **(C_i)** - number of methods overridden by C_i (redefined inherited methods)
- M_a **(C_i)** - number of methods available in C_i

The following relationships between the method-count metrics are given:

$$M_d(C_i) = M_n(C_i) + M_o(C_i)$$

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$

These class-level metrics give the first indication that the MOOD team misunderstand some basic OO principles.

4.2.2 Attribute overriding?

The method-count metrics are also applied to attributes (notated by ***A*** in place of ***M***), with no consideration for the essential differences between methods and attributes.

The metric $A_o(C_i)$ is meaningless. The concept of overriding concerns the behaviour defined in a method. An attribute does not have behaviour, and thus cannot be overridden. It is certainly possible for a class, ***Q***, to define an attribute named ***x*** even if its parent class, ***P***, already has an attribute (of the same type) named ***x*** but the attribute ***Q.x*** does not *override* ***P.x***. Both attributes will form a part of any object created from class ***Q***. The methods of ***P*** that refer to ***x*** will use ***P.x*** and the methods of ***Q*** that refer to ***x*** will use ***Q.x***.[8]

If ***Q.x*** is counted as an 'overriding' attribute, rather than a new attribute when calculating $A_d(Q)$ then ***P.x*** would not be counted as an inherited attribute when calculating $A_i(Q)$. This would further result in an inaccurate value being returned for $A_a(Q)$.

4.2.3 Inheritance versus Polymorphism

The view is taken that if a method is overridden then the original method is not inherited. This is not necessarily true. Inheritance and polymorphism are not mutually exclusive. It is certainly true that only the overriding method may be invoked for an object of the class but it is also possible, indeed likely, that the overridden method is invoked from within the body of the overriding method, thus the behaviour defined in the overridden method is indeed 'inherited'. An example may help to make this clear:

```
class P {
   public void action() { //method body; }
}
class Q extends P {
   public void action() { super.action();  //rest of method body; }
}
```

[8] This technique is sometimes referred to as 'attribute shadowing' [FLAN97], but it is not commonly used due to it giving no particular programming advantage: it makes more sense to give the attribute ***x*** in class ***Q*** a different name.

The method `Q.action()` clearly inherits the behaviour defined in `P.action()`. The code does not have to be re-written; it is re-used. This observation affects both the 'Method Inheritance Factor (MIF) and the Polymorphism Factor (POF) metrics.

A system consisting of the two classes defined above would return a value of 0 for MIF, implying that there is no inheritance, which is misleading.

In terms of polymorphism the MOOD team make no distinction between methods which specialize (i.e. extend the services of) the overridden method and those which completely change it. The former is considered good OO design while the latter is considered an anomaly to be detected and queried (LORE94). These two styles of overriding will be discussed further, along with the POF metric, in section 3.3.

4.2.4 Information-hiding and encapsulation

The metrics, 'Method Hiding Factor' (MHF) and 'Attribute Hiding Factor' (AHF) are considered by the MOOD team to be measures of encapsulation [ARBR95, ARBR96]. This indicates poor understanding of the concept of encapsulation. Information hiding and encapsulation are not synonymous. Information hiding is only one part of the concept. Encapsulation can be thought of as being an aggregate of two different but related attributes, namely *privacy* and *unity*. The Chambers 20th Century Dictionary definition reflects this view:

> ***encapsulate*** *to enclose in a capsule: to capture the essence of, to describe succinctly but sufficiently.*

The *privacy* aspect satisfies only the first part of this definition. Privacy is concerned with the visibility of the fields and methods of a class to other classes in the system. Unity is described in Chambers as:

> ***unity*** *oneness: (…) the state or fact of being one or at one: that which has oneness: a single whole: the arrangement of all the parts to one purpose or effect.*

It is concerned with the degree to which a single class represents a single entity and satisfies the second part of the 'encapsulate' definition, above.

Encapsulation can be quantified by measuring both the unity and privacy (i.e. data privacy[9]) of a class. A class that has only private data members will not necessarily be unified. Equally, a fully unified class may contain only visible (public) data.

MHF and AHF are measures of the visibility of a class's properties. They are not measures of encapsulation. The AHF metric indirectly contributes to such a measure but it is doubtful if the MHF measure serves an equivalent purpose.

4.2.5 Private members

The view is taken that private members (methods and attributes) are not inherited by subclasses [ARBR95], thus they are not counted in the $\boldsymbol{M_i}$ **(**$\boldsymbol{C_i}$**)** and $\boldsymbol{A_i}$ **(**$\boldsymbol{C_i}$**)** metrics. While this may seem correct and logical in principle it causes some inconsistencies in practice. Consider a section of a class hierarchy: *Point* ✍ *ThreeDPoint* with the following definitions:

```
class Point {
     private int x, y;
     public void setX( int _x);
     public void setY( int _y);
```

[9] Privacy really concerns the data (fields or attributes) in a class. The visibility of methods is less important.

```
     public int getX();
     public int getY();
}
class ThreeDPoint extends Point {
     private int z;
     public void setZ( int _x);
     public int getZ();
}
```

The class *ThreeDPoint* inherits methods to get and set the *x* and *y* coordinates and yet does not (according to the MOOD definitions) inherit the coordinates themselves. When we consider that every instance of the class *ThreeDPoint* does in fact contain its own copy of each of the three coordinates it will be seen that the MOOD definition does not make sense. Our intuitive understanding of a *ThreeDPoint* object is an object with three coordinates. If the coordinates are not defined in the *ThreeDPoint* class itself they must be inherited. They should therefore be counted in the $\boldsymbol{A_i}$**(*ThreeDPoint*)** metric, and consequently in the $\boldsymbol{A_a}$**(*ThreeDPoint*)** metric.

It is perhaps more acceptable to ignore private methods in the $\boldsymbol{M_i}$ **(**$\boldsymbol{C_i}$**)** metric. Private methods cannot be invoked in association with any class other than the class in which they are defined and, unlike attributes, they do not hold a value that forms part of the class's representation. This is another example that illustrates the importance of treating methods and attributes separately and not assuming that they are always equivalent.

The important point to make here is that which ever decision is made regarding the inheritance of methods and attributes it is essential to provide sound reasoning for the decision and make it explicit in the metrics definitions. The MOOD team does neither. No reasons are provided for the decision and the decision itself is far from being explicit: it can only be found as a footnote in [ARBR95].

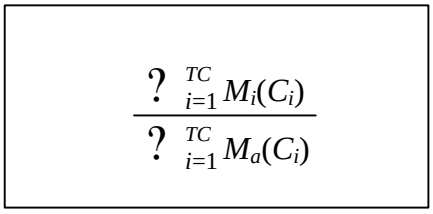
4.2.6 The OO paradigm

The preceding examples indicate that the MOOD team is working with an inaccurate and imprecise understanding of the OO paradigm. This raises doubts as to whether metrics built on such an unsteady foundation will prove correct and robust. In Section Two each metric is examined in more detail and further inaccuracies and inconsistencies in the metrics' definitions are reported. When appropriate, alternative definitions are proposed.

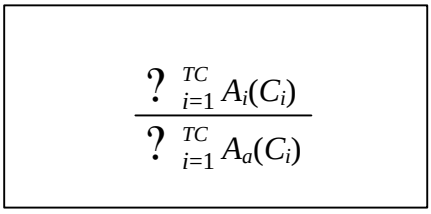
4.3 The MOOD metrics

Each of the MOOD metrics is expressed as a quotient where the numerator is the actual use of a particular mechanism (i.e. inheritance, information hiding, coupling and polymorphism) in the system being measured and the nominator is the maximum possible use of the same mechanism. The value for each metric will therefore be in the range 0-1, with 0 indicating no usage and 1 indicating maximum usage. The values for the metrics are expressed as percentages, 0-100%.

4.3.1 Inheritance Factors



$$\frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$



$$\frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

MIF : Method Inheritance Factor **AIF: Attribute Inheritance Factor**

The definitions for MIF and AIF are inconsistent with the 0-1 scale. The rest of this section explains this in terms of MIF; it applies equally to AIF.

The methods available in a leaf class, M_a(*Leaf_Class*) are not inheritable (within the system). Therefore the MIF denominator, by including leaf classes in the $M_a(C_i)$ sum does not represent the maximum possible inheritance, rather it represents a value *greater* than the maximum; hence the value for MIF, for any system, can never be 1.

Example:

A system consists of three classes, P, Q and R arranged in the following hierarchical structure: P ✍ Q ✍ R.

```
class P { public void x(); public void y();}
class Q extends P { //no methods defined}
class R extends Q { //no methods defined}
```
[10]

Q and R both inherit the two methods defined in class P and define no further methods. This is the maximum possible method inheritance in this system (i.e. all methods that can be inherited have been inherited, by all classes that are able to inherit them). Intuitively, it seems that the MIF value for this system should be 100%, but in fact it is 66.6% (the sum of all $M_i(C_i)$ = 4 and the sum of all $M_a(C_i)$ = 6, 4/6 = 0.666).

$M_i(P)$ = 0, $M_i(Q)$ = 2, $M_i(R)$ = 2, Total = 4

$M_a(P)$ = 2, $M_a(Q)$ = 2, M_a(R) = 2, Total = 6

This principle can be further illustrated. If class R in the above example had a new method added it should not change the MIF value for the system. This is consistent with our intuitive understanding of method inheritance. In fact we find that it does, with M_a(R) = 3 MIF becomes 4/7 = 0.57 (57%).

[10] This is admittedly an unlikely scenario, but nevertheless a possible one and as such it must be considered.

The definitions for MIF and AIF need to be amended to remove this inconsistency. However, even if this can be done the main problem with the MIF and AIF metrics is that they are not really telling us anything of interest, especially if it is accepted that all private methods and attributes are inherited (see section 2.5). It may be more interesting to develop a metric that measures inheritance at a class level, rather than separate metrics to capture method and attribute inheritance. After all, it is classes that are extended; methods and attributes just come as part of the package.

A Class Inheritance Factor (CIF) metric could be defined as the total count of all ancestors for all classes divided by the maximum possible inheritance for the system.

Inheritance is one-way. If class ***A*** extends class ***B*** then it is impossible for class ***B*** to also extend class ***A***. This means that the maximum inheritance level for a system with *n* classes will be 0 + 1 + ... + (*n*-1).

$$\frac{\sum_{i=1}^{TC} AC(C_i)}{TC*(TC-1) / 2}$$

where *TC* = Total Classes

Thus a more formal definition for a Class Inheritance Factor metric would be:

This value will be 100% when the classes are all arranged in a linear hierarchy. It will be 0% when there is no inheritance; i.e. all classes are base classes with zero descendants.

This metric would also be compatible with other inheritance metrics such as the QMOOD [BANS97] NOA (Number of Ancestors) metric or the McCabe [MCCA98] 'Fan-In' metric. The numerator will be the sum of each class's NOA or 'Fan-In' value.

4.3.2 Information Hiding Factors

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

MHF: Method Hiding Factor

$$\frac{\sum_{i=1}^{TC} \sum_{m=1}^{A_d(C_i)} (1-V(A_{mi}))}{\sum_{i=1}^{TC} A_d(C_i)}$$

AHF: Attribute Hiding Factor

where $V(M_{mi})$ | $V(A_{mi})$ = the visibility value of a member (method or attribute), i.e. a value between 0-1 where public members = 1, private members = 0 and semi-public (e.g. protected) members are calculated as the number of classes that can access the member / total classes in the system.

Method hiding

In [ABRE94] it is recommended that MHF should not be lower than a particular (as yet undefined) value but suggest that there is no upper limit, thus implying that it is 'good' for all methods in a class to be hidden (private). This is clearly non-sensical and indeed, in [ABRE95] this heuristic is amended to give both a lower and an upper limit. However, the number of private methods in a class does not tell us anything about the degree of information hiding in the class. It may tell us that a particular method (or methods) has been broken down into a number

of smaller methods to avoid duplication or for clarity of understanding. Such methods would only need to be visible to the containing class. But whether or not a method is broken down this way the containing class's implementation is still hidden. In the following example both classes have equal 'information-hiding' levels:

```
class P {
   private int x;
   public int m0() { do_action1; do_action2; do_action3; return x;}
}
class Q {
   private int x;
   public int m0() { m1(); m2(); m3(); return x;}
   private void m1(){ do_action1;};
   private void m2(){ do_action2;};
   private void m3(){ do_action3;};
}
```

In class *P* all of method *m0*'s behaviour is contained in the body of *m0*. In class *Q* the behaviour has been separated into three smaller methods which are called by *m0*. Both classes have identical interfaces and their respective implementations are equally well hidden from client classes. A count of the number of private methods in a class is not a particularly useful metric, and certainly does not contribute anything to our knowledge of a class's encapsulation level.

It is worth noting that the QMOOD metrics set [BANS97] contains an analogous metric, along with the recommendation that the resulting value for this metric should be high, i.e. most, or all methods in a class should be publicly visible. The Smalltalk? language supports this view: all Smalltalk? methods are public by default.

Attribute hiding

The AHF metric was covered to some extent in section 2.4. This is a clearly defined metric with no apparent inconsistencies. Its use is in determining the level of visibility of a class's data.

4.3.3 Polymorphism Factor

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

POF: Polymorphism Factor

The definition of POF means that it can only be applied to complete hierarchies. It therefore fails the MOOD team's fifth criterion:

> ? Metrics should be down-scaleable.

It is possible, indeed highly likely, that a sub-system will consist of a set of classes that extend a framework. This may be a set of library classes or a framework of low(er) level system classes. When measuring the sub-system it should be only the classes that belong to the sub-system that are measured; classes outside of its boundaries (which is where the framework or library classes

will lie) should not be considered. In such cases the denominator for the POF measure may be less than the numerator, resulting in a value greater than 1. An example will make this clear.

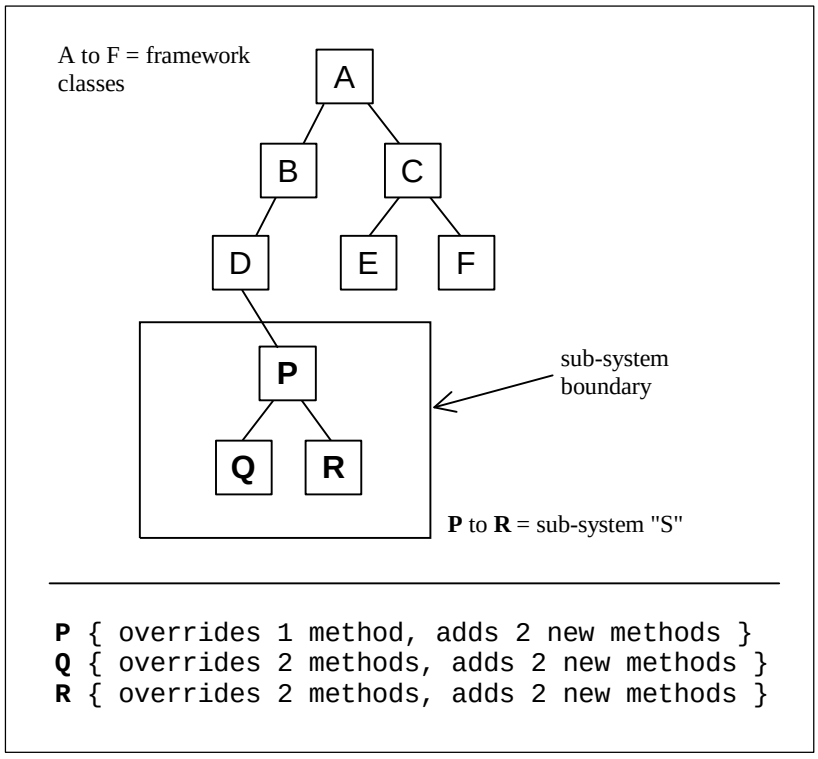


Figure 4a

In *figure 4a* sub-system "S" produces a value for POF which is outside the range 0-1.

M_o(**P**) = **1**, M_o(**Q**) = **2**, M_o(**R**) = **2**

(M_n(**P**)=2 * *DC*(**P**)=2) = **4**, (M_n(**Q**)=2 * *DC*(**Q**)=0) = **0**, (M_n(**R**)=2 * *DC*(**R**)=0) = **0**

therefore:

POF for sub-system "S" = (1+2+2) / (4+0+0) = 5/4 (and 5/4 > 1).

Taking this concept one step further, it is possible for an entire system to be built using an existing framework. This is especially likely in languages that are shipped with large class libraries, such as Java or Smalltalk. In such cases the whole system could produce POF>1.

Neither of the two languages considered by the MOOD team in the earlier development stages, namely Eiffel and C++ is as heavily dependent on a standard class library. This is possibly the reason why this eventuality was overlooked, and it illustrates the danger of defining a set of 'language-independent' metrics without considering all possible languages[11].

The POF metric can be redefined so as to remove this inconsistency.

$$\frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} M_{Ov}(C_i)}$$

New definition for POF

[11] i.e. within a given set of languages (which in this case is all class-based OO languages that support inheritance).

The numerator is unchanged. The denominator is the sum of all $M_{Ov}(C_i)$ where $M_{Ov}(C_i)$ is a count of all methods that can potentially be overridden by class C_i. This will consist of all methods in the parent class or classes excluding private methods and (in languages that support them) class-wide ('static') methods.[12]

This new definition for POF removes the anomaly described above and means that it is now down-scalable to sub-systems.

This definition still contains the discontinuity identified in [HARR98] concerning systems with no inheritance. The denominator will always be 0, giving an undefined value rather than a value of 0. However, as the numerator will also be 0 in every this is a philosophical, rather than a practical concern.

Specialization v Convenience

The POF metric, in both its original form and in the new form suggested here, does not distinguish between the two possible types of method overriding. An overriding method can either extend the behaviour defined in the overridden method or completely change it. The latter case is considered "*overriding for convenience*" by Lorenz [LORE94] and is considered undesirable: an anomaly to be detected. Lorenz defines a metric, Specialization Index' (SIX) which is intended to do just that, but it is not compatible with any of the MOOD metrics, not least because it is theoretically completely unsound [MAYE99b].

POF is only really a valuable metric if the organisation using it has strict guidelines regarding the use of polymorphism, e.g. an overriding method must either extend a template (pure virtual or abstract) method or invoke the superclass' method from within its body. Without clear guidelines the value produced by POF will have little meaning in terms of the quality of a system's design.

4.3.4 Coupling Factor

$$\frac{\sum_{i=1}^{TC}\left[\sum_{i=1}^{TC} is\_client(C_i, C_j)\right]}{TC^2 - TC}$$

where $is\_client(C_i, C_j)$
= **1** if C_i contains at least one non-inheritance reference to a method or attribute of class C_j
= **0** otherwise

COF: Coupling Factor

This metric is intended to count all client-supplier relationships in a system. It is not clear, however exactly what is meant by a 'non-inheritance reference'. The two main relationships in an OO system are 'is-a' and 'has-a' relationships. The former describes relationships based on inheritance and the latter describes client-supplier relationships, e.g. one class's use of another class as an instance variable[13].

The important point here is that the relationship between any two classes in a system is not constrained to just one or the other of these relationship types. As an example consider the two classes, *Component* and *Container*, from the Java *java.awt* library package. *Component* is the super class of all graphical components and *Container* is one of its subclasses. Thus the two

[12] The issue of class-wide methods and attributes is not dealt with in any of the MOOD papers.
[13] This is by no means a complete description of all OO relationships but it is sufficient for this example.

classes are in an inheritance relationship: *Container* **is-a** *Component*. However, each class also contains an attribute of the other class type, i.e. *Component* has an attribute of type *Container* and *Container* has an (array) attribute of type *Component*. *Container's* use of a set of *Components* has nothing to do with the fact that *Component* is its superclass, indeed if the hierarchy was redesigned to alter this fact it would not alter a container's need to maintain references[14] to all the components that reside in it.

The question that the MOOD team does not adequately answer is whether a client-supplier relationship under these conditions is counted. There is probably no 'correct' way of dealing with this situation in terms of the COF metric (there are good arguments to support both the counting and non-counting of such a relationship) but a decision needs to be made one way or the other and it needs to be explicit in the metric's definition.

One final point to be made concerns the name of the metric. Coupling has a clearly defined meaning in terms of conventional software measurement [HITZ96]; the same meaning is not being used here, therefore a different term is required. Other OO coupling metrics, e.g. Chidamber & Kemerer's CBO metric, have been criticized for not applying different weights to different types of 'coupling', e.g. [HITZ96]. This is a case of 'function-orientated-metrics-hangover'. The relationships in an OO system are very different to those in a function-orientated or procedural system and to try to apply the same metrics is cause for confusion and, arguably a major contributor to the lack of success of OO metrics. Such criticisms can be avoided simply by choosing a new term to replace 'coupling'. The term 'interconnectedness' although cumbersome is perhaps more accurate for describing what is essentially the set of (non-inheritance) edges in the graph that is the OO system design[15].

4.3.5 Other Metrics

Two additional metrics were proposed in [ABRE94] but have since been dropped from the set. Subsequent papers contain no explanation for this action. The reader is invited to draw his or her own conclusions.

CLF Clustering Factor

This was presumably dropped as it fails as a ratio measure in that the values for CLF can never be zero.

RUF Reuse Factor

In [ABRE94] this metric was singled out as being the only one that did not measure an aspect of the measured system's complexity. A composite metric was planned that incorporated the other seven metrics.

4.4 Conclusion

A detailed analysis of the MOOD metrics set has been carried out a number of fundamental flaws have been exposed. That the MOOD metrics are imperfect does not make them bad metrics. On the contrary, they are among the best of the many metrics currently available. The fact that they can be criticized in such depth indicates that much thought has gone into their planning and that their descriptions, while not always accurate, are certainly detailed and

[14] All Java attributes are references.

[15] The reader may notice that such a measurement would indeed include the type of client-supplier relationship described earlier.

thorough. In contrast, the metrics proposed by McCabe & Associates [MCCA94] have definitions so sparse as to make sensible theoretical analysis impossible.

Looking beyond the sometimes-bewildering mathematical expressions used to describe the MOOD metrics they are in essence simple and straightforward to compute. The MOOD metrics have the potential to mature into a decent, base-line set of metrics for OO systems. To reach this state it is recommended that the following action be taken:

- The MHF metric is replaced by a metric to capture the 'unity' aspect of encapsulation and the MIF and AIF metrics are replaced by a single 'class inheritance' metric.
- Further work is put into correcting the inconsistencies and inaccuracies pointed out in this chapter and generally fine-tuning the metrics' definitions.
- Additional theoretical and empirical validation is carried out by third parties.
- Further language bindings are produced for all widely used class-based OO languages.

How useful these or any other OO-specific metrics prove to be within the software industry is dependent to a large extent on the quality of the tools that are developed to collect the metrics. Such tools cannot be accurately built until the definition of each implemented metric attains the kind of precision and clarity that was recommended in the first two chapters of this work and is further being recommended here.

5 Where to Now?

5.1 Summary

The first part of this work has explored the different object-orientated design metrics that have been developed since 1990. In particular the focus has been on the metrics suites proposed by Chidamber & Kemerer, Lorenz & Kidd, McCabe & Associates and The MOOD team. Although some interesting and important work has occurred in this field the field itself is shown to be disjointed and riddled with inaccuracies and inconsistencies. Much of the work is rooted in the function-orientated concepts of coupling and cohesion. Many metrics suffer from the absence of a measurement theory basis, while others suffer from being over-theoretical, thus lacking in intuitive appeal. The terminology required for this field is ill defined and important language issues are almost completely ignored. Furthermore, and perhaps most importantly, there is little communication between the different proponents of these measures, resulting in disagreements, not so much between what should be measured but how it should be measured. There is also the sense that the same metrics are being defined repeatedly by different proponents, which is somewhat ironic given the OO paradigm's emphasis on reuse.

That some of the work is motivated by commercialism adds to the lack of consistency and agreement. The Hatteras Inc.? web pages even go so far as to boast how their tools offer metrics that no one else offers [LORE98].

It would appear that much of the work carried out over the past nine years has been driven by a desire to conquer the market place: to be the first person or team to produce the definitive set of object-orientated design metrics. This would explain much of the lack of communication and the reluctance to utilise the work done by others. Even those responsible for testing the metrics, e.g. [HARR96, HARR98, BASI96] are neglecting to follow up the few developments that have

occurred on the suites that they test. Hitz and Montazeri have done some excellent work on developing the Chidamber & Kemerer cohesion (LCOM) metric (see Chapter 3). The new formulation is finer grained and more accurate than the original and is compliant with the principles of measurement theory but has all but been ignored by independent testers. The Hitz & Montazeri work demonstrates perfectly the value of building on the foundation of others, rather than always starting from scratch.

Despite the best efforts of the various proponents little progress has been made. Many of the proposed frameworks and models never reached beyond the walls of academia and much of the work has never been followed up - even by the authors themselves. The metrics that are making it into industry tend to be those developed *by* the industry, by companies such as McCabe & Associates. These metrics tend to be the least theoretically sound but interestingly also tend to have the highest intuitive appeal. The Chidamber & Kemerer metrics are also finding homes in a number of independently developed metrics tools. Again, they have a high intuitive appeal. How useful the implemented metrics are proving to be is not known. That is a whole other area of research.

5.2 The future

The field of object-orientated design metrics needs to wake up. If software engineers and academics are genuinely interested in developing an accurate, language-independent, industry-wide set of metrics then a new approach to the problem is called for. Interested parties need to put aside dreams of academic glory or industry dollars and begin to communicate with each other in a cooperative and supportive manner. Existing work needs to be developed and the developments need to be shared. Until this begins to happen it is unlikely that object-oriented design metrics will ever be more than a fringe activity in the software engineering industry.

Perhaps what is needed at this point is the intervention from a powerful and respected body of software professionals and academics such as the Institute of Software Engineering (ISE) or the IEEE. This professional group would need to take the situation in hand and lay down some clear and unambiguous rules for the development of new OO design metrics that all proponents should follow. Metrics developed according to the criteria laid down by this group will be given some kind of official 'seal of approval' so all practitioners know that the metric is valid. Metrics lacking such approval will slowly be left by the wayside. But until such intervention comes (if, indeed it ever does), the field of OO metrics needs to find some other way of achieving the necessary cohesion; it requires a new model for collaborative metrics development.

5.2.1 A model for development

There already exists in the field of Software Engineering an interesting model for co-operative development, a model which the field of OO metrics could, surprisingly, learn a lot from and even embrace whole-heartedly, adapting to its own circumstances as it sees fit. This model is the Open Source model of software development as described in [HECK99].

The Open Source model utilises the Internet as a development forum. It is founded in the belief that all information should be free - and should be openly shared. As stated earlier, one of the main problems in the OO metrics field is the lack of communication between proponents and the reluctance to use other people's ideas. Adopting a model that actively encourages such sharing and reuse can overcome this hurdle.

The following quotes, taken from Eric Raymond's paper "The Cathedral and the Bazaar" [RAYM99] will serve to illustrate some of the advantages of using such a model. Words in square brackets are added to put the statements into a research context.

> *"Given a large enough beta-tester and co-developer [/co-researcher] base, almost every problem will be characterized quickly and the fix obvious to someone."*
>
> *"The next best thing to having good ideas is recognizing good ideas from your users [co-researchers]. Sometimes the latter is better."*
>
> *"Often, the most striking and innovative solutions come from realizing that your concept of the problem was wrong."*
>
> *"Provided the development [research] coordinator has a medium at least as good as the Internet, and knows how to lead without coercion, many heads are inevitably better than one."*

With an open model such as this each newly proposed metric can be discussed, examined, evaluated, empirically tested and improved prior to being released. Measurement tools can be developed and tested by large groups of programmers/testers using the 'Bazaar' method of development [*ibid*] with a greater likelihood of early and accurate completion. Most importantly, a process similar to natural selection will kick in as more and more researchers and practitioners (especially the latter) come on board. It will soon become clear which metrics are most used and most useful and it will be those metrics which get more time and more attention. Automated tools will be built to collect the metrics, language bindings will be produced for more languages and the definitions will become finely honed. This in turn will ensure still wider usage and the process will continue to iterate.

5.2.2 A new context for OO metrics

The 'Hacking Culture', where the Open Source concept originally came from, is the subject of the second part of this work. The unlikely combination of OO design metrics and the hacking culture is considered and the relationships between the two fields are explored. The primary aim is to show how the use of metrics can support the hacking style of development, in particular in terms of quality assurance, i.e. ensuring (OO) design standards are upheld. A life cycle structured on this method of development is then proposed.

It must be remembered, however, that for any serious utilization of OO metrics to occur the content of the first part of this work needs to be heeded: the current work needs a major overhaul.

Part Two

OO Metrics & the Hacking Culture

Introduction to Part Two

The second part of this work looks at the re-emergence and re-acceptance of the hacking style of programming as practiced by the Open Source developers and characterized by Raymond in [RAYM99], and examines the role that software metrics might play to support the modern-day hacker.

The Hacking Culture: Chapter Six takes a brief look at the history of hacking, introduces the hacker philosophy and explores the hacker development environment.

OO Metrics for (OO) Hackers: Chapter Seven explores how the hacking development style can be supported by the OO paradigm, and more specifically by OO design metrics. It recommends a set of metrics that are applicable to any development style and highlights those that are especially useful to the OO hacker, given the iterative style of development usually employed. It contends that the quality of design can be improved by the application of such metrics throughout the development cycle, but recommends that empirical tests be carried out to verify this.

The 'Real Life!' Cycle: Chapter Eight extends this idea with the introduction of a full development life cycle founded on the hacking style of development with quality control imposed by the use of OO metrics. The life cycle, nicknamed The 'Real Life!' Cycle (RLC) is proposed as being suitable for small to medium-sized software projects. The three key phases of the RLC are described and it is claimed that this life cycle accurately reflects the way that many developers actually work - or would like to work, given the opportunity.

Part Two concludes with a look at future research directions to follow up the ideas presented here.

6 The Hacking Culture

Hacking is coming back in fashion. This technique of developing software systems, used by the software pioneers in the 1950's and 1960's and then all but stamped out in the 1970's and 80's by over-zealous management wanting more control and more of the credit [GLAS95], is finally being given the recognition and respectability it so deserves. To be more precise, it is the particular style of hacking that utilises many developers and uses the Internet as a forum that is gaining the respectability. This method of hacking, more recently dubbed 'Bazaar Development' by Eric Raymond in his widely-read paper "The Cathedral and the Bazaar" [RAYM9X], has gained its respect through the work of the Free Software/Open Source gurus such as Richard Stallman, founder of the GNU project, Tim Berners-Lee, inventor of the World Wide Web, Linus Torvalds, the driving force behind Linux, the free operating system which has recently received so much notice (and credibility), and Eric Raymond himself.

Although the Bazaar/Open Source method of development utilises a great number of programmers and testers each individual developer owes a debt of gratitude to the original MIT hackers of the 1950's. It is the style of work and, more importantly, the ethic behind the style [LEVY85] pioneered by these original hackers, that each of the Open Source developers has adopted.

6.1 The hacker ethic

Hacking, in its original sense[16], can be described as the development of software using informal, unstructured and undisciplined methods by unmanageable individuals. However, it could also be described as the crafting of software using innovative techniques by persons with a high degree of passion for the work. It depends on your perspective.

> *"The 'original' hackers were computer professionals who [...] adopted the word 'hack' as a synonym for computer work, and particularly for computer work executed with a certain level of craftsmanship. They subsequently started to apply the noun 'hacker' to a particularly skilled computer workers who took pride in their work and found joy in doing so."* [HANN98]

But more than just developing great computer programs the original MIT hackers - by the very way they interacted with the computer: their passion and openness and their underlying belief that access to the computer was a *right*, were developing a culture and a philosophy.

> *"The precepts of this revolutionary Hacker Ethic were not so much debated and discussed as silently agreed upon. No manifestos were issued. No missionaries tried to gather converts. The computer did the converting…"* [LEVY84]

The Hacker Culture is one of openness and sharing. It is one of doing, rather than theorizing. It is a culture where the artistry in execution is as important (sometimes more important) than the observable results [HANN98].

The Hacker Ethic embraces six imperatives:

- reject hierarchies
- mistrust authority
- promote decentralization
- serve your (the hacker) community
- share information - freely
- judge only according to merit

The Hacker Ethic has been adopted since by many hacker-developers and it is this ethic, this philosophy, more-so even than dreams of fame or financial success, that has fuelled some of computing's greatest triumphs; the Linux operating system and the World Wide Web to name just two [*ibid*].

Probably the strongest advocate for the Hacker Ethic is Richard Stallman, the ex-MIT AI lab programmer who left there in 1985 to form the Free Software Foundation (FSF). The FSF have produced a huge body of free software under the name of GNU[17]. Stallman made his position crystal clear in the GNU manifesto.

> *"I consider that the golden rule requires that if I like a program I must share it with other people who like it. Software sellers want to divide the users and conquer them, making each user agree not to share with others. I refuse to break solidarity with other users in this way. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. For years I worked within the Artificial Intelligence Lab to resist such tendencies and other*

[16] The term 'hacking' has more recently been used (or misused) to describe that brand of software 'pirate' who uses his computing skills to break into other people's systems, sometimes for profit but most often simply just for the hell of it. Throughout this document the term is used in its original sense, to describe a highly skilled and creative programmer.

[17] GNU is a 'recursive acronym', (hacker humour!). It stands for 'GNUs, not Unix'.

inhospitalities, but eventually they had gone too far: I could not remain in an institution where such things are done for me against my will.

So that I can continue to use computers without dishonor, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free. I have resigned from the AI lab to deny MIT any legal excuse to prevent me from giving GNU away." [STAL85]

The concept of sharing information coupled with the hacker's natural desire to improve on imperfect systems has led to the Open Source model. More than just being freely available, the source code released in this way is intended to be continually updated, modified and improved by those who use it. The only stipulation is that any changes made to an existing source code be shared with the whole community.

To illustrate the importance that the average hacker gives to the ideals of the Open Source model is well illustrated by the Apache-IBM story. IBM wanted to use the Apache group's web server software as the cornerstone of a new Internet commerce package, WebSphere. But the twenty or so developers, scattered around the world that make up the Apache group do not legally exist as a company, and as individuals, were not interested in making financial deals. In fact, the individuals were reluctant to make any kind of deal with IBM - traditionally the enemy of hackerdom [LEVY85]; the latter had nothing that the Apache group could possibly want. Until, that is a group of IBM programmers figured out how to make Apache run faster on Microsoft's NT operating system. IBM offered this 'hack' to the Apache group, with a promise to also share future hacks, and thus an agreement was reached between the two sides [MCHU98].

6.2 A brief history

Before the days of waterfall and spiral life cycles, structured programming, formal design notations and other management-imposed techniques to control the software process hacking was all there was. Programs were written in a loose code-debug-recode cycle; they were hacked and 'bummed' to perfection before being released [LEVY85]. There were no design documents, no functional specification, little in the way of a requirements specification and even in-line comments were kept to a minimum. Many of the hackers were of the mind "it was hard to write so it should be hard to understand". (Fortunately, this attitude was non-pervasive and is less often seen today then it was back in the early days of hacking.)

With the management-imposed structuring of the software process everything swung in the opposite direction, more and more documentation was demanded as the software life-cycle became tighter and more controlled until it reached the point where the end-of-phase documents became more important to management than the programs themselves.

"Management could now check progress by examining those documents. And they could almost understand them! Instead of a few scraps of paper and an incomprehensible listing, there were now artifacts produced as part of the software process that management could read - even if it couldn't (or wouldn't) read the final product, the code itself. [...] The documents, realized the programmers, were more important than the program itself. After all, management didn't ever look at THAT!" [GLAS95].

This rigid, top-down view of software development has since gone out of favour but various other management-imposed structures have taken its place. We now have spiral life cycles, formal methods, algebraic specifications and a plethora of CASE tools to automate and routinize the software process. Even the more innovative methods such as prototyping have had

structure imposed on them. There is still an emphasis on control and formality in the development of software.

However, more recently there has been the start of a movement away from this. Not before time the software community is beginning to realise that some of the best software out there has not been built by large teams of organised (managed) personnel using formal, structured methods but by means altogether more anarchic. Well-known and much-used software products such as **Lotus 123**, the **Macintosh** operating system, **Image DBMS**, **dBase III** and the **Java** language were built by individuals or teams of 2-3 people [DALC99]. Other software artifacts, widely acknowledged to be the best of their type such as the **emacs** editor, the **Lisp**, **C**, **C++**, **HTML** and **Perl** programming languages, the **Linux** operating system, the **TeX** and **LaTeX** typesetting programs, the **Apache** web-server software and even the **Internet** have their roots in the hacker community [HANN98]. Even in cases where large teams *are* required, a more informal, innovative process is being favoured. Microsoft's approach to building software, where developers are paired with testers and trusted to build their sections of a program using whatever methods best suit them [CUSU96], has been compared to the Bazaar method by Brooks [BROO95]. Microsoft's products, while much criticized, are probably used by more PC users than any other commercial products.

It is on the wave of this movement away from formality and towards innovation that the ideas in this chapter are hoping to ride. Taking a step back from the Bazaar method of development and focussing initially on the original style of hacking (the single hacker, working alone) this chapter will look at how this style of work can be supported by the OO paradigm and, more exactly, by OO software metrics. The ideas presented here can easily be transposed to apply to the hacker working on a larger, Bazaar-style project.

6.3 The hacker as artist

The OO paradigm is known to lend itself well to an iterative life cycle, such as prototyping. Hacking is nothing if not iterative. No hacker ever sits down and writes a perfect program from scratch, just as no conventional - structured - programmer does. The main difference between the two styles is that the hacker style allows the programmer to write code immediately; he/she will work out the program design and structure through the code, whereas the structured approach insists on carefully drawn designs, functional specifications and even test cases being implemented before any coding begins. The structured programmer approaches his work as a clerk, in an ordered, methodical fashion, closely following the rules. The hacker, on the other hand, approaches his work as an artist, seeing big concepts and forming ideas, which he then realises. Rules are there as boundaries, not restrictions. They guide rather than direct.

The hacker approach is not as disorganised as at first it sounds. Before any program is written someone has to have the concept. This concept is itself some form of representation of the program. Far from just sitting down and coding wildly the hacker has this concept, this model in his mind. It may have come about via a discussion with a client or user-group or it may be a 'personal itch' of the hacker himself [RAYM99]. This model is a raw, ever-changing and malleable model. In some ways it can be parallelled to the sculptor's stone, which starts off as a raw block and is gradually chiselled and chipped into the form the sculptor desires. The hacker, likewise, chisels or 'hacks' away at his model until he has sculpted his program.

The advantage the hacker's model has over the pre-drawn designs used by a more structured methodology is that it is not fixed. It is more open to change, and change is more easily implemented.

This is doubly true when the language - or more accurately, the design - being used is object-oriented. The OO paradigm has an inherent structure that is lacking in more traditional, function-orientated languages. OO systems are built out of clearly defined components and the relationships between the components are equally clearly defined. If the paradigm is strictly adhered to this clarity of component and relationship encourages clearer, more understandable designs, allows for easier manipulation and greater adaptability during development and leads, ultimately to a more maintainable system [ETSM91].

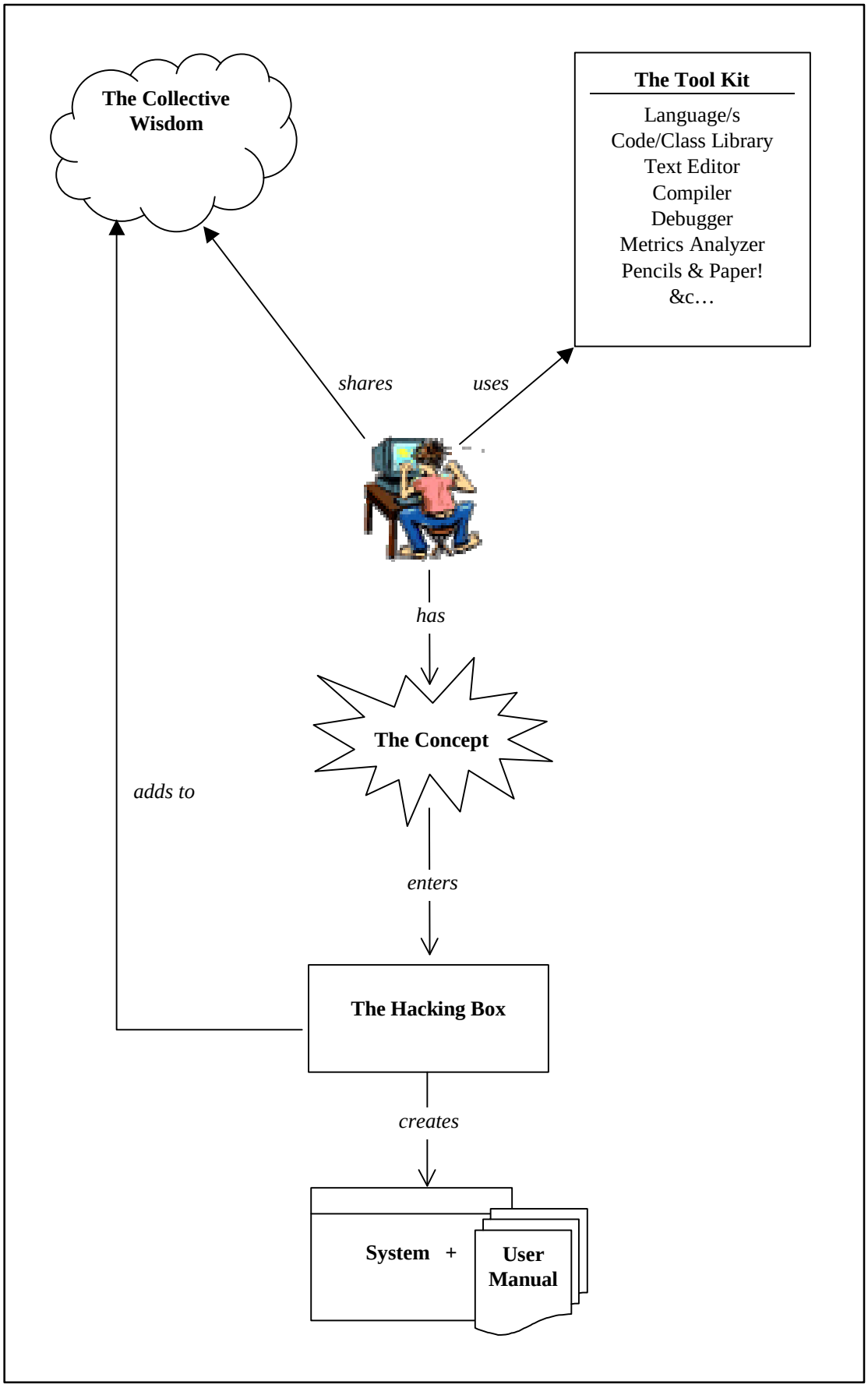


Figure 6a: The Hacking Environment

6.3.1 The hacking environment

Figure 6a shows the hacker in his development environment. The elements of the diagram are discussed in more detail below.

Collective Wisdom

The Collective Wisdom represents the hackers link with the greater hacking community. The Hacker Ethic demands that the best insights and the most up-to-date knowledge about computer programming be shared among the community. This sharing is achieved through user groups, news groups and other mechanisms supported by the Internet and the World Wide Web. An individual hacker can, for example, be warned of the latest bugs in a system, be given tips on improving algorithms, download source code and executable programs (for free, of course) and generally be offered useful tips and on-going support by others in the hacking community.

Tool Kit

The typical hacker's tool kit will consist of software tools, utilities and various functions or classes that he has either built himself or has downloaded as 'free software' from the Internet. Organisations such as the FSF exist almost solely for the purpose of supplying free software for hackers - all of it written by (other) hackers. The set of tools and utilities will continually be updated as new versions/bug fixes appear and the Code/Class Library will grow as more development work is completed and new re-usable components identified. The list of tools in Figure 1 will be familiar to most programmers, the only odd item being the Metrics Analyzer. This is a tool to automate the process of collecting metrics on source code. The majority of hackers, traditionally have not used such a tool, nor indeed taken any interest in, or seen any use for software metrics. Section 1.4 will explain its (somewhat controversial) introduction to the hacker's tool kit and provide justification for this.

Concept

The concept, or model, is the initial idea for a program or system. It was described in detail earlier on in this section.

Hacking Box

The Hacking Box represents the mental space that the hacker is in during program development. It is in this space that the creativity and craftsmanship occur. The hacker will not necessarily be sitting at a computer. He can be in the Hacking Box at any time, in any (physical) place. It simply represents the time during which the hacker is thinking about or working on the program. It is during this period that the hacker 'gives back' to the community. Problems and innovative ideas will all be taken to the user groups for discussion. Additionally, most hackers release their code as 'Open Source' (a.k.a. 'Free Software'), whether or not the actual program is free.[18]

---

[18] For a thorough understanding of the Open Source model from a business perspective see [HECK99].

7 OO Metrics for (OO) Hackers

7.1 Creating an OO design

To build within the OO paradigm it is not enough to just use an OO language. Many criticisms of the OO paradigm, e.g. [HATT98], when closely analysed, turn out to be a criticism of the language being used - in most cases this is a hybrid language such as C++ [WIEN98]. The problems with such a language stem from the programmer failing to use a pure OO model, and falling into old, function-orientated habits. One way of avoiding this is to use a language, such as Smalltalk or Java, that has removed most function-orientated features and encourage the programmer to think in OO terms. It is still possible of course to misunderstand the OO paradigm and create a bad (or at least non-OO) design. For example, with the Java language it is easy to create designs that make no use of inheritance and whose classes are over-sized and non-specialized and have an over-reliance on static methods and public variables. Such designs are clearly not built on an OO structure. The programs thus produced may work, although probably not optimally, and they have lost the other benefits of using the OO paradigm, discussed above.

One way of ensuring that a pure OO design is being created is by using OO design metrics. A small set of well-defined metrics, which can measure the source code automatically and produce summary results that are easily understood and interpreted, will allow the programmer to get a clear overview of the system he is developing. It will be clear at a glance which sub-systems need restructuring, which classes need to be redesigned (merged, split, unified, better organised,

etc.) and how well the system as a whole adheres to the OO paradigm. Information on the reusability of classes can also be gleaned from the metrics results.

7.2 Recommended OO design metrics

The main OO mechanisms of abstraction, inheritance, collaboration, encapsulation, and polymorphism are well understood by experienced OO designers and programmers, and it is recognised that making 'good use' of these mechanisms is key to producing elegant, maintainable and reusable software systems. 'Good use' of course, is relative, but there are certain criteria that most designers, e.g. [BOOC94], [LORE94], [BUDD91] would agree on, these include:

- Inheritance hierarchies should not be too deep, or too shallow. A maximum depth of around six is suggested in [LORE94].
- Multiple inheritance should be used minimally and only when essential
- Extension should be used for specialization, not convenience.
- A method that overrides another method should extend the behaviour, not change it.
- The higher the class is in a hierarchy, the more abstract it should be.
- A single class should represent a single entity.
- A class's data should be hidden from the 'outside world' and only available through access methods.
- Well-encapsulated classes are easier to understand and maintain.
- An OO system should be an intelligent balance between effective collaboration and low inter-connectedness.
- Function-orientated constructs (e.g. global variables, 'friend' functions in C++, etc.) should be avoided altogether.

Metrics that enable the developer to ensure such criteria are being adhered to will undoubtedly prove useful during the design and implementation phases of the development process.

The design metrics recommended here are those that the author has found from experience to be the most useful and accurate in quantifying an OO design. The metrics are divided into categories that coincide with the OO mechanisms listed above, namely abstraction, inheritance, collaboration, encapsulation, and polymorphism. As a general rule, metrics concerned with the relationships between classes (inter-class metrics) are more useful during the early, design-focussed stages and those concerned with the inner workings of an individual class and its methods (intra-class metrics) are more useful during the later, refinement stages.

No guide is given for the use of these metrics. The OO paradigm has clear guidelines (discussed earlier) for the use of each of its mechanisms. Additionally, each developer (or employer) will likely have his own guidelines for 'good' OO designs.

7.2.1 Metrics for the design phase

An important part of the early design phase consists of establishing the classes that will make up the system and the relationships between those classes. Relationships between classes in OO designs fall into two categories. Two classes can have either an ancestor-descedant relationship

or a client-supplier relationship. The former is described by inheritance metrics and the latter by collaboration metrics.

It is not desirable for all classes in a system to be freestanding, rather an intelligent level of 'is-a', 'has-a' and 'uses-a' relationships are essential for a well designed system. Reuse is another important aspect of OO development so for a class to have multiple children/descendants, or even multiple clients, may well be a design goal. A class does become more difficult to comprehend if it has multiple suppliers or lies deep in an inheritance hierarchy (particularly in a multiple-inheritance hierarchy). The system will need to be a compromise between good re-use/specialisation and low interconnectedness.

Inheritance metrics

Many metrics exist to capture the inheritance aspect of a system. Most of these are simple counts of ancestors or descendants (both immediate and non-immediate) or are mean values of these counts taken across a system.

The first four candidate metrics, below, are taken from the metrics defined in [CHID94] and [BANS97]. They are all simple counting measures that are defined for single classes. Their names accurately describe their usage. The fifth metric is introduced and described in Chapter Four of this work.

Number of Immediate Parents (NOP)

Number of Ancestors (NOA)

Number of Children (NOC)

Number of Descendants (NOD)

Class Inheritance Factor (CIF)

Collaboration metrics

There are essentially three different ways that two classes can be in a (direct) client/supplier relationship. The following relationships are possible between two classes, **P** (the client) and **Q** (the supplier):

1. **P** declares a data attribute (aka field/variable) of type **Q**
2. a method of **P** has a local variable of type **Q**
3. a method of **P** accepts a parameter of type **Q**

An additional relationship type is sometimes mentioned, that of a method of **P** returning a value of type **Q** (including Exception or Error classes that are 'thrown') but for this to occur one of the previous three relationships must already apply.

One further relationship type does exist in languages that support class-wide properties[19] (known as 'static' in Java and C++):

4. a method of **P** references a class-wide property (method or attribute) of **Q**

Note that in the first case the **Q** object potentially can be used by all of **P**'s methods. In the other cases only a single method of **P** is using the **Q** object. Certain metricians suggest measuring the different strengths of these relationships, stating (quite correctly) that the accessing of a public variable represents stronger "coupling" than the accessing of a method [HITZ96]. However, as the use of public variables breaks the data-hiding principle of OO design

[19] Class-wide properties can be accessed independently of an object reference.

this fact should be captured by a separate metric and not allowed to confuse the issue when measuring collaboration.

In all cases above (including case 4) **P** relies on the existence (in the system) of **Q** and the person who implements, tests or maintains class **P** would need knowledge of class **Q**. Therefore, it may not be necessary to differentiate between these relationship types when measuring the number of suppliers for a class. If differentiation is made it is better to design different metrics to capture each different relationship-type rather than apply some form of 'weighting', using what are really no more than random numbers.

A class that is an ancestor or descendant of a particular class can also be in a client-supplier relationship with the same class, but only if it is explicitly declared to be so. For example, class **A** has a descendant, **B** and **B** has a field of type **A**, therefore **A** is an ancestor and a supplier of **B**, and likewise **B** is a descendant and a client of **A**. **B**'s use of **A** as a supplier is a completely separate relationship to **A** as an ancestor of **B**. Such cases, while not common, certainly do exist and each relationship type should be measured separately. An example, taken from the Java Development Kit (JDK), was given in [4.3.4].

Candidate metrics for measuring collaboration are:

Number of Client Classes (NCC)

Number of Supplier Classes (NSC)

Each of these can be made both in a direct way and an indirect way. In the following system (where ?**--** indicates a supplier-to-client relationship):

A?--**B**?--**C**
?--**D**

Class **A** has one direct client (**B**) and two indirect clients (**C** and **D**), giving a total of three clients; likewise **C** and **D** both have one direct supplier (**B**) and one indirect supplier (**A**), giving a total of two suppliers each. Class **B** is only engaged in direct relationships: as a client of **A** and a supplier to **C** and **D**.

Abstraction metrics

> *"Abstraction is a mechanism for focusing on the important (or essential) details of a concept or item, while ignoring the inessential details."* [BERA96]

Abstraction can occur at a class or method level. In general, the higher the class is in a hierarchy the more abstract it is likely to be, i.e. the greater percentage of abstract methods it is likely to have.

Percent of Abstract Classes (ACP)

> Measured at a class level. The number of abstract classes in a system divided by the total number of classes. The recommended threshold for ACP is 10-15% [LORE94].

Percent of Abstract Methods (AMP)

> Measured at a class level. The number of abstract methods in a class divided by the total number of methods.

Summary of design metrics

The abstraction, inheritance and collaboration metrics will quantify, and provide an interesting, numeric description of a system, identifying those classes that may need more thorough testing or are likely to be problematic. They are, however, most useful for large systems that the developer is unable to keep track of in his mind. For smaller systems - of the type that a single,

hacking-style developer is likely to be working on - their use is limited. The metrics for encapsulation and polymorphism are more likely to be found useful in an immediate, practical way.

7.2.2 Metrics for the implementation phase

Encapsulation metrics

Encapsulation is concerned with the packaging of data and behaviour that together represent a single entity. Ideally the implementation details should be concealed and access to the encapsulated entity only available through a public interface.

To properly assess the quality of encapsulation requires human understanding; the design needs to be fully comprehended. There are however two aspects of encapsulation that can be assessed automatically. The first concerns the degree to which a single class represents a single entity and the second relates to the visibility, or containment, of a class's data.

Class Unity (UNI)

UNI is a measure of the similarity of methods in a class. Two methods are considered similar if they access one or more of the same instance fields. UNI counts distinct clusters of methods, where a cluster is defined as a group of methods that are linked to each other, either directly or indirectly, through accessing the same field or set of fields, any one method in the cluster accessing at least one field which is accessed by at least one of the other methods in the cluster. A cluster can be conceived as a graph where the nodes are the methods. Two nodes are connected by an edge if the two methods (that the nodes represent) both access the same instance variable/s. A measure of 1 indicates full unity (all methods connected); higher numbers imply progressively less unity.

Note: UNI is essentially a re-naming of the Chidamber & Kemerer LCOM metric [CHID94], however the definition used here is that proposed by Hitz and Montazeri in their correspondence paper [HITZ95] and not that originally defined by Chidamber & Kemerer.

Level of data Visibility (VIS)

VIS is a measure of how visible (and therefore directly accessible) a class's data is. A class's data is considered hidden (invisible) if all its instance fields are private and can only be manipulated via methods. VIS uses a scale of 0-1 for each field, where 0=fully hidden (private), 1=fully visible (public) and the value for partially visible (e.g. protected) fields lies between 0 and 1. An intuitive value can be assigned (e.g. 0.5) or an exact value calculated by dividing the number of (other) classes in the system that can access the field by the total number of classes in the system, less one (i.e. the class itself). Which of these methods is chosen depends on the level of granularity required for this metric. The latter is the method used for the Attribute Hiding Factor (AHF) metric proposed by the MOOD Metrics team [ARBR96].

The final VIS value for a class is the average visibility across all fields in the class. The closer the total measurement is to zero the more hidden the data.

Ideally a class will have a value of 0 for VIS and a value of 1 for UNI. Classes with higher values should be examined to ensure they couldn't be split into smaller classes with a parent-descendant relationship.

Both UNI and VIS can be extremely helpful to a developer in ensuring that the system being developed does not stray from the OO paradigm. The UNI metric in particular identifies an

attribute that is not immediately obvious from looking at the source code, i.e. how well a single class represents a single entity.

Polymorphism metrics

Polymorphism is the most difficult of the OO mechanisms to devise meaningful metrics for. For this reason no candidate metrics are recommended here and only a general guide is given to the aspects of polymorphism that are potentially measurable.[20]

Polymorphism means 'many forms'. There are two basic ways of implementing polymorphism, overriding and overloading. A method that overrides another method cannot be in the same class as the overridden method. It has the same name, parameter list and return type. A method that overloads another method has the same name and return type but a different parameter list. Methods can overload other methods in the same class. More commonly, polymorphism refers to the former definition, and it is that definition that is considered here.

When a method overrides another method it can either extend the behaviour of that method or change it. The latter case is considered undesirable, producing brittle relationships [LORE94] and leading to the same type of problems created by the use of unconstrained goto statements [TEGA93].

In assessing polymorphism we would want to know firstly, if the mechanism is being used, and secondly, if it is being used correctly. Use of Polymorphism can be determined by simply counting the number of methods in a hierarchy that share the same name and parameter list. Assessing the quality of polymorphism is more difficult and to be done properly needs human understanding. The code, or design, would need to be studied to determine if the overriding method is indeed 'specialising' the overridden method. There are however two indicators that can be relied on to show that specialisation is occurring. The first is if the method is overriding an abstract - or empty - method (i.e. no previous behaviour, nothing to change) and the second is if the overriding method invokes the superclass' method from within its code body.

Due to the dynamic binding of methods to messages there is no way of knowing, with static metrics, if the polymorphic methods are actually being used in a polymorphic way. They may simply be called like an ordinary method (using static, compile-time, binding). Only by using dynamic (run-time) metrics can we determine the extent to which the polymorphism mechanism is being used.[21]

7.2.3 Applying the metrics

The use of metrics to capture the attributes described above will allow a hacker to continually improve the quality of the system under development.

In the early stages of development it is the structure of the system that is the prime concern. The use of a creative technique, such as CRC cards [RUBI98] or Scenarios [BUDD91] is recommended to support the structuring of the system. The use of abstraction and inter-class metrics alongside will ensure that the system's responsibilities are sensibly distributed and the connections in the system are kept at an understandable and manageable level; for example, classes with excessive dependence on other classes will be highlighted and that part of the system can be restructured.

---

[20] Actual polymorphism metrics are proposed by Lorenz & Kidd [LORE94] and by the MOOD team [ARBR95] but the usefulness and accuracy of both these metrics is called into doubt in the first part of this work.

[21] The author is grateful to Peter Rosner for this observation.

Once the design is reasonably clear the hacker will commence with building the individual classes. At this stage in the development the metrics for capturing the attributes of encapsulation and polymorphism will be more useful. The hacker can incrementally improve the design of each class, knowing that the OO paradigm (and any developer-specific guideline) is being closely adhered to. As pointed out earlier, the use of these metrics - especially the inner-class metrics - assumes an iterative development cycle of the kind that most hackers use. There is little point in identifying anomalous classes if it is not possible to make the required changes to those classes (e.g. because the development phase is considered 'complete').

7.3 The next stage

Right now, this is a theory. Formally worded it reads:

> **The application of OO design metrics by a hacker-style (OO) developer during the implementation phase of a system will serve to improve the quality of the design of that system.**

It appears intuitively correct but in order for it to be accepted as correct the theory need to be tested. A series of tests need to be set up where two developers are each developing identical systems, using the same language. One will apply the metrics and the other will not. At the end of the development each system will be assessed as to its quality. If enough cases show that the use of metrics leads to a system of higher quality the theory gains credence. It will not be proven, as such but it will stand stronger than it does without such evidence. It goes without saying that the term 'quality' needs to be very clearly defined before the tests commence - itself no easy task!

8 The 'Real Life!' Cycle

This chapter extends some of the ideas from the previous chapter to propose - or rather recognise and formalise - a life cycle for the development of software systems, a life cycle that is oft used but - due to the importance attached to structured methodologies - is seldom admitted to. The life cycle is named, for the purposes of this document, "The 'Real Life!' Cycle (RLC).

The ideas in this chapter are not fully developed at this stage but have progressed far enough to warrant inclusion in this work

8.1 The rational underlying the RLC

"I don't know what I think until I see what I say." (E.M. Forster)

"I don't know how I'm going to do it until I work out how it can be done." (Anon)

The 'Real Life!' Cycle is built on two simple concepts, both of which, interestingly enough, are much maligned by the Software Engineering Industry. By taking a fresh look at these concepts, and utilising them for the purpose of a development life cycle it is hoped that a new credibility can be attached to them. The two concepts are usually known by the derogatory names of 'Hacking' and 'Faking'. Hacking was introduced in the first part of this paper. The term Faking was used by Parnas and Clements to describe the process of developing design documents after the system has been developed, to fake a top-down approach [PARN86]. The logic of this is

clarified by Glass who observes that while most designers tend to use a 'hard-part-first' approach to designing a system the documents of such a process are close to impossible for others to comprehend and it is necessary to re-write the documentation to *fake* a 'top-down' approach in order to overcome this problem [GLAS95].

The Development phase of the RLC is rooted in the concept that a valid OO development life cycle can be built on the Hacking process. Quality control will be imposed by the use of OO design metrics, as described in the first part of this paper. Managers have traditionally shied away from the archetypal hacker, fearing the absence of control was synonymous with the lack of surety of a quality product. By using code metrics at regular stages through the iterative, hacker-style development cycle the project manger has the assurance that the system is being developed according to pre-determined guidelines and the developer is assisted in producing a well-designed system that is in keeping with the OO paradigm.

The Design Derivation phase is founded on the concept that Design documents are best produced *after* the system has been developed. This ordering is intended to assure the correctness and easy readability of the design documents for the purpose of system testing and maintenance. Far from being a 'fake' such documents are more likely to be a true model of the system than documents drawn up prior to implementation.

The traditional way of developing a system is to produce a finished set of designs and a functional specification before any coding begins. The problem with this method of development is that it is almost impossible to get a system right first time and it is inevitable that changes to the initial design and specification will be made. It is often the case that the original design document is not updated to reflect those changes. There are a number of reasons for this; for example the programmer or project manager making the changes may not have write-access to the original design document, the designer may have moved on to another project and no longer has the time needed to make the changes, or it may have become too difficult (and messy!) to keep writing and rewriting the changes (laziness may also come into the equation). The result of this is a set of design documents that do not correspond to the system. Such documents are effectively useless.

As stated, the RLC utilises a hacking-style model for the development of a system. Hackers tend to work not from designs but from a set of requirements. The design is done alongside the coding, as the system is being built. There is clearly no need for a design document at the development stage. The greatest use for a design document - in any life cycle - is after the system has been delivered, i.e. during the system's maintenance phase. By focusing on producing a design document that matches the system that has just been built the 'Real Life!' Cycle model ensures that the system maintenance team will be working from an accurate and meaningful representation of the system rather than from a set of designs that have been developed beforehand, changed (usually improved) during coding, and never updated - a situation that frequently occurs according to [BESK98].

> *"...it is rare for a design to stay the same when coding begins, since problems always surface. [...] I don't think I have ever seen a design document that was up to date and accurate enough to really bother with. High level documents are about the best one can hope for. Any design doc below that is next to useless, if you even have one. So, the code ends up being the only source of information."*

J. Beskin, Chief Tester at Reliable Software Technologies (commenting on the need to read the entire source code in order to understand a system)

8.2 Application of the RLC

Glass [GLAS95] states an important truth about the development of software: ***not all software projects are the same***.

While this is glaringly obvious (i.e. once it has been said!) it is clearly ignored by those who propose new methodologies, CASE tools, formal methods and so on. These people would have us believe that their new idea/method/tool is the answer to all of software's problems, regardless of the size, complexity or domain of the project. A one-size-fits-all answer. Then, when this is finally found - as it inevitably will be found - not to be true, practitioners lose interest and confidence in the thing and it is trashed, along with so many others before it [*ibid*]. If the proponents of these methods only aimed a little lower and considered the limitations of their proposals then perhaps some of these now-abandoned methods may still be in use (and possibly even useful) today.

In order to avoid making the same error it is important that we be clear from the start just which type of software project the 'Real Life!' Cycle is intended to be applicable to. In order to do that it is first necessary to define a simple taxonomy framework to describe software projects. The following is adapted from Glass [*ibid*].

Project Characteristic	**Candidate Groupings**
Domain	Business Scientific System Real-Time Other
Size	Very Large Large Medium Small
Commonality of the problem	Unique Rare Common
Criticality of the solution	High Medium Low

Other characteristics such as System Complexity are not likely to be known at the beginning of a project, so are excluded from this framework. Still others such as Developer Experience, Management Style, etc. are excluded due to their being considered external to the system; they are not inherent.

This particular taxonomy of just four characteristics gives 180 different system types (5 x 4 x 3 x 3) ranging from (e.g.) a small-size, non-critical system tool similar in type (common) to other, existing system tools to a very large, highly critical real-time system to solve a unique (i.e. never-before-solved) problem.

The 'Real Life!' Cycle is recommended, in the first instance, for small to medium size systems with low criticality that require high innovation (i.e. rare to unique commonality) in any domain, except possibly real-time systems and large business systems. The reasoning behind this recommendation will become clear from the rest of the paper but briefly it is as follows.

Size = Small to Medium:

It is important that each developer has a complete overview and understanding of the problem domain and of the developing system. The larger the number of developers the harder this becomes. Less programmer-programmer interfacing will allow individual programmers more freedom to experiment and apply individual creativity.

Criticality = Low (non-critical)

There is no sure and safe way to develop critical systems but the methods currently in use are likely to be safer than this method, which is, as yet, untried.

Commonality = Rare to Unique

The more innovation and creativity required to develop a system the better suited is the RLC. Its absence of formal rules for development allows the developers the freedom to experiment and thus arrive at novel solutions to as-yet unsolved problems. Systems where the problems to be solved are well known (common) problems require less creativity. They can be formally defined and the solutions highly automated.

Domain = All (except Real-Time & certain Business systems)

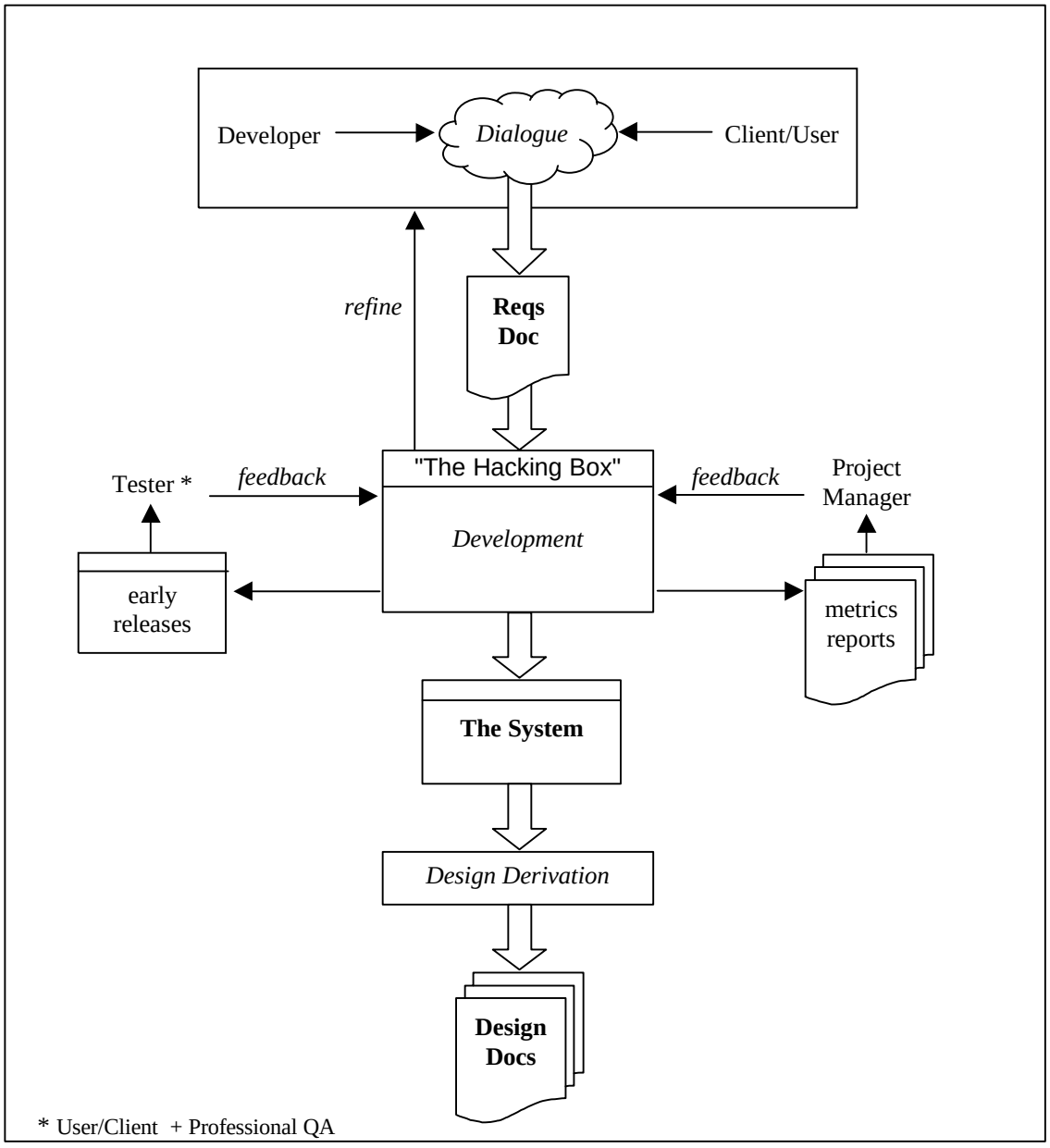
Most real-time systems have a high level of criticality, which in itself excludes this type of system. Business systems that require an in-depth analysis of an existing system are also not suited, as the RLC does not allow for such an analysis in its cycle. However, as such systems are usually large systems (which are excluded anyway) this makes little difference. Due to the inherent flexibility of the RLC it can be adapted to suit the building of systems from most other domains.

8.2.1 The OO aspect

The RLC is intended specifically for OO development. There are a number of important reasons for this:

1. The OO paradigm lends itself well to iterative-type development cycles. Etsminger, in *The Tao of Objects* [ETSM91], declares that the best way to design an OO system is to do it: to get on with the coding and to design as you go along with support from such tools as CRC cards [RUBI98].
2. OO systems are built out of clearly defined components with clearly defined interfaces and the relationships between classes are equally clearly defined. This clarity of component and relationship makes future maintenance of the system potentially easier.
3. OO systems can be clearly described, in a quantitive way, by metrics. The field of OO metrics is immature and at present many of the proposed metrics are unclearly defined. However, the paradigm does have the potential to breed metrics that are precise and absolute and not open to the same amount of (mis)interpretation that functional (structured) program metrics have historically been open to [MAYE99c].
4. Most OO languages are self-documenting. The Eiffel language provides support for specification at the implementation level. Java allows document-coments to be inserted in the code and the programmer can then auto-generate specification files from these comments at a class, package and hierarchy level. Such features are not generally available in non-OO languages.

Figure 8a: The 'Real Life!' Cycle



8.3 The RLC described

8.3.1 The phases (overview)

There are essentially three phases to the RLC (shown as large, italic text on the diagram) that always occur in the same, following order:

Phase	Phase-Ending Product
1. Client-Developer Dialog	The Requirements Document
2. System Development	The System
3. Design Derivation	Design Documents & Functional Specification

The System Development phase occurs within the boundaries of 'The Hacking Box'. This is an imaginary space in which the developer 'lives' while creating the software. The phase covers the design, implementation, unit testing (and system testing for very small systems) and informal documentation processes - but not necessarily in that order, or indeed in any sequential order.

8.3.2 The phases (in detail)

The RLC is described here in terms of single-developer projects (with or without a project manager).

Phase 1: Requirements Dialog (Client-Developer Dialog)

The developer meets with the client and the eventual user (who may be the same person). Ideally this meeting would be face-to-face. Failing that, video conferencing would be a good second best. Distanced communication such as email or telephone is not advisable; the conversational/argumental aspect that is so important to the process is lost when such mediums are used.

It is a well-known software engineering truth that the client either doesn't know what he wants or does know, but is unable to express it [SUMM95]. The developer's task is to draw the requirements out of the client and (should there be one - or more) the user. In order to do this it is clear that the developer needs excellent communication skills and a good understanding of the domain in which the system will exist. Some systems will be sufficiently complex to demand a certain amount of prototyping before the requirements are fully understood. The advantages of having the developer be a key player at this stage will be obvious. The developer will make the decision whether or not to utilise the prototype code in the actual development phase.

In certain cases the developer may be working under the guidance of a project manager. If this is the case the PM should also be present for the Requirements Dialog, if not all the way through then certainly at the stage where the requirements actually get written down and signed by both parties.

The Requirements Dialog will eventually produce the Requirements Document. It is essential that this document is drawn up with great care and agreed to by both parties. It should cover everything that it needs to cover and nothing more [PARN86]. Nothing should be taken for granted or left unsaid. At the same time nothing should appear in the document that is superfluous or a repetition. Repetitions, especially are causes of developer confusion: not knowing which of the directives to follow.

Phase 2: System Development: 'The Hacking Box'

The Hacking Box was discussed in the previous chapter; it represents the mental space that the hacker is in whilst developing the system. As far as management - (including for the most part) the project manager - is concerned The Hacking Box is a black box. Its input is the Requirements Document and its output is the fully developed system. How the one becomes the other should not be the concern of anyone who is outside the black box (in the case of single-developer projects that is everyone except the single developer). Should the developer require support or advice then the project manager will be allowed to 'enter the box', but this must always be at the request of the developer. It should never be imposed.

In order for this to work there are 3 important pre-requisites.

Pre-requisites for the System Development phase

1. The developer must be excellent. It is not enough to have just a 'good' or 'adequate' developer. The RLC developer must have exceptional programming and general problem-solving skills, a complete and unmuddied understanding of the OO paradigm, knowledge of the domain and experience of developing in that domain. He does *not* need any particular knowledge of formal methods or of any design methodologies/notations. Neither is he required to produce any formal, phase-ending-type documentation. All he has to do is build a system that meets the requirements - in the best way that he knows how. Ideally, he will deliver that system on time and within budget, but this should be no more expected than it is under other circumstances.

2. The developer must be trusted. He will have been employed on the basis of his excellence, as described above. There should be a short trial period in which management is allowed to enter the black box to ensure correctness of development but once this (formally defined) period has expired, and the developer has proved himself, then he is to be completely trusted from that point. If the developer feels he is mistrusted his output is likely to be less than optimal as he will be trying to please his manager, rather than please himself.

3. The developer must have a sound knowledge of company design and coding standards and must agree to follow them.

Taking the requirements, which he was instrumental in defining, the developer will proceed with the creation of the system, working in whichever way best suits him. As already stated the method of development is not imposed but there are certain criteria that need to be adhered to.

Development criteria

1. Measurement: The design must be subject to measurement at regular intervals and the reports of those measurements delivered to management. The purpose of this is twofold:

   i) to assure management that company standards in terms of design are being adhered to.

   ii) as important feedback for the developer in order that he understand and improve the design on an ongoing basis.

   The metrics used will be agreed on before hand and may be different from project to project. They will however all be static (i.e. design) metrics that are clearly defined, OO-specific and language (i.e. OO language) independent. The use of metrics to support the hacking process was discussed in the previous chapter. Metrics to ensure both the correctness of the system as a whole and of individual classes should be used.

   Providing the organisation has clear design standards that are known to improve system quality (e.g. maintainability and reliability) and these standards are agreed on by all developers then the use of OO design metrics will be a very precise method of ensuring that different developers on different projects, possibly using different languages, are developing systems that meet the standards. The need for management intervention, code reviews, etc. is diminished.

2. Testing: On single-developer projects the developer is responsible for testing his own code. However, at the point when early system builds are possible these must be delivered to whoever is responsible for integrated system testing. It is recommended that the test team be a combination of client/user and professional QA personnel. The former will be more responsible for testing the user functionality (GUI, etc) and the latter for ensuring the correctness of the product. (If all is going well in the development phase there should be little to correct - this is just a safety net.) Both tester-types will be responsible for feeding

back reports on problems that have been found. The developer is then obliged to work on changing those problems (removing bugs, improving usability, etc) before the next release.

3. Requirements Change: If at any stage during the development phase either the developer or the client needs to make actual changes to the requirements then the development phase should be temporarily halted and a new Requirements Dialog set up. Each new Requirements Dialog will proceed in the same way as the first one and (whenever possible) with the same group of people. A new Requirements Document will again be produced at the end. The development then starts up again using the new document. It is possible that some major changes will now be needed: some code will have become redundant, designs will need to be reconsidered, additional classes/sub-systems developed, &c. It is also probable that additional time will need to be allocated to the project. It is important that the Requirements Dialog and the Development phase are treated as discrete units; they should not occur concurrently. The finished system must match the latest Requirements document - exactly. It must contain everything stated in that document and nothing more. All previous requirements documents become redundant, and should only be retained for historical purposes.

Phase 3: Design Derivation (Faking)

Once the system has been developed and tested as described above it is ready for release. At this stage, apart from the Requirements Document, there is no official system documentation. This is fine. Up to this point it has not been needed. However, once the system has been released it will eventually require maintenance and those responsible for maintaining the system will need (in addition to the requirements document and the source code) the following design documents:

- *Class Hierarchy Diagrams*
- *Collaboration Diagrams*
- *Class Diagrams*
- *Object Interaction Diagrams*
- *Functional Specification*

Such documents should be derived using a reverse engineering process. This is not meant to imply that a fully automated process be used. Rather it means that the source code and the system itself should be studied/analysed and the documents derived from that study. Parts of this process may be automated but greater accuracy is likely to be achieved using a manual process.

The system itself will be ready for release on completion. Its release will not be dependent on the documentation (other than the user manual) being complete. This is likely to mean that systems developed in this way are delivered to customers sooner than if they had been developed using a more traditional - structured - life cycle.

8.4 Adopting the RLC

One of the key attributes of the RLC is its flexibility. Different developers can adopt the style of development that best suits them. This is likely to result in overall better quality systems than if the developers are being forced to use a style that is unfamiliar to them, or that they find counter-intuitive. But it is not just the development phase that exhibits this flexibility. The entire life cycle, from requirements to release has only a loose structure that is easily adaptable to suit the culture of a particular organisation. Adopting this life cycle does not require a

company to undergo major structural change. The way that management carry out their feasibility studies, risk analysis and other parts of the management process need not be subjected to a great deal of change, if any at all. Developers, analysts and designers can continue to use the same CASE tools and the final documentation can follow the same format that is has always done. From a maintenance point of view life should in fact become easier. The development hierarchy of project managers, senior managers and developers can be retained, but a more hands-off approach is required from management. And that is the crux. It is not a structural but an attitude change that is required. The organisation as a whole will need to adopt a more open and trusting culture if the RLC is to prove successful.

8.5 Future directions

The RLC is a very new concept and this chapter should not be considered as anything more than the germ of an idea to increase the presence of creativity in software engineering. It is certainly not 'an answer'. Much has still to be done to go from the ideas presented here to a valid software development life cycle. Additionally, and as discussed in the first part of this work OO metrics are not yet mature enough to be trusted as the sole QA process for a software project. The metrics aspect is key to the success of the RLC so if the latter is ever to become a valid replacement for other, more traditional life cycles a good deal more effort needs to be put in to developing a set of well-formed, finely-grained and intuitively appealing OO design metrics.

And that thought, which takes us full circle and right back to the beginning of this work, is an appropriate place on which to end.

Afterword

The decision, more than half way through this research project, to expand the research base and incorporate the hacking culture alongside the OO metrics work threw up a number of unexpected ideas for further research. Certainly, more time can be put in to formulating the 'Real Life!' Cycle and to exploring how OO metrics can directly support the hacker. Both areas require the improvement of OO metrics and both can be described as areas where hacking is supported by OO metrics. But it may be remembered that the original intention of widening the research base was to see if there was something out there that could support OO metrics and give it the much-needed injection of cohesiveness and correctness that the first part of this work identified. Can the hacking culture support OO metrics? The answer to that, I strongly believe, is YES! At the end of Part One in the section, 'a model for development', I touched briefly on the idea of an Open Research forum, along the lines of 'Open Source', for the further development of OO metrics. 'Open Source' has grown directly out of the hacking culture. It thrives on the principle that information should be free and widely shared, which is precisely the remedy that this research prescribes for the field of OO metrics.

Setting up such a forum is a major task; indeed without the Internet as a base it would be impossible. It requires a dedicated enthusiast to coordinate it, someone with excellent Web authoring skills, an interest in OO measurement and a belief in the Hacker Ethic. Above all, it needs someone with a big chunk of time to spare coupled with an altruistic spirit. There is little chance of personal glory or financial gain. The concept will be offered as a final year project at South Bank University next academic year (1999-2000).

If this work is remembered for one thing only I hope it will be the message that compartmentalizing for the sake of control and understandability is not necessarily the best option - and it is certainly not the most creative. Seeking connections between apparently disparate entities or concepts - between '*the beast and the monk*' - can lead to a new awareness of old problems and pave the way for refreshing and innovative solutions where previously none were deemed possible.

References

Software Metrics & Related Subject Matter

[ABRE93] F. B. e Abreau and R. Carapuça, "Candidate metrics for object-oriented software within a taxonomy framework", *Proc. AQUIS'93 Conference*, Venice, Italy, Oct. 1993

[ABRE94] F. B. e Abreau and R. Carapuça, "Object-oriented software engineering: measuring and controlling the development process", *Proc. 4th Int. Conf. On Software Quality*, McLean, VA, USA, Oct. 1994

[ABRE95] F. B. e Abreau, M. Goulão and R. Esteves, "Toward the design quality evaluation of object-oriented software systems", *Proc.* 5th *Int. Conf. On Software Quality, 1995*

[ABRE96] F. B. e Abreau and W. Melo, "Evaluating the impact of object-oriented design on software quality", *Proc. 3rd International Software Metrics Symposium (METRICS'96),* IEEE, Berlin, Germany, Mar. 1996

[BAKE90] A. L. Baker, J. M. Bieman, N. Fenton, D. A. Gustafson, A. Melton and R. Whitty, "A philosophy for software measurement", *J. Systems Software, 1990; 12: pp277-281*

[BANS97] J. Bansiya and C. Davis, "Automated metrics and object-oriented development", *Dr Dobb's Journal*, Dec. 1997

[BANS99] J. Bansiya, L.Etzkorn, C. Davis and W. Li, "A class cohesion metric for object-oriented designs", *JOOP*, January 1999

[BASI96] V. R. Basili, L. C. Briand and W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators", *IEEE Transactions on Software Engineering*, 22(10), October 1996

[BERA96] E. V. Berard, "Metrics for object-oriented software engineering", *The Object Agency, Inc., http://www.toa.com/pub/html/moose.html*, 1996

[BIEM91] J. M. Bieman, "Deriving measures of software reuse in object orientated systems", *Formal Aspects of Measurement*, T. Denvir, R. Herman and R. Whitty (Eds), London, Springer-Verlag, 1992

[BINK96] A. B. Binkley and S. R. Schach, "Impediments to the effective use of metrics within the object-oriented paradigm", in *Proc. OOPSLA'96*, UK, 1996

[BINK97] A. B. Binkley and S. R. Schach, "A classical view of object-oriented cohesion and coupling", 1997

[BUNG97] M. Bunge, *Treatise on basic philosophy: Ontology I: The furniture of the world.* Boston: Riedel, 1977

[BUNG99] M. Bunge, *Treatise on basic philosophy: Ontology II: The world of systems.* Boston: Riedel, 1979

[CHID91] S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object-oriented design*", Proc. Sixth OOPSLA Conference*, 1991, pp. 197-211.

[CHID94] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object-oriented design*", IEEE Transactions on Software Engineering*, 20(6), 1994, pp. 476-493.

[CHID98] S. R. Chidamber, D. P. Darcy and C. F. Kemerer, "Managerial use of metrics for object-oriented software: an exploratory analysis, *IEEE Transactions on Software Engineering Vol. 24, No. 8*, August 1998, pp 629-639

[CHER91] J. C. Cherniavsky and C.H. Smith, "On Weyuker's axioms for software complexity metrics", *IEEE Transactions on Software Engineering*, 17(6), 1991, pp. 636-638.

[CHUR94] N. I. Churcher and M. J. Shepperd, "Towards a conceptual framework for object orientated software metrics", (*published? - 1994*)

[CHUR95] N. I. Churcher and M. J. Shepperd, "Comments on ' A metrics suite for object orientated design'", *IEEE Transactions on Software Engineering*, 21(3), 1995, pp. 263-265.

[FENT92] N. E. Fenton, "Software measurement: Why a formal approach?", *Formal Aspects of Measurement*, T. Denvir, R. Herman and R. Whitty (Eds), London, Springer-Verlag, 1992

[FENT96] N. Fenton & S Pfleeger, *Software Metrics: A rigorous and practical approach (2nd Ed.)*, Thompson, 1996

[GUST91] D. A. Gustafson and B. Prasad, "Properties of Software Measures", *Formal Aspects of Measurement*, Eds. T. Denvir, R. Herman and R. Whitty, Springer-Verlag, 1991

[HARR96] R. Harrison and R. Nithi, "An empirical evaluation of object-oriented design metrics", in *Proc. OOPSLA'96*, UK, 1996

[HARR98] R. Harrison, S. J. Counsell and R. V. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics", *IEEE Transactions on Software Engineering*, 24(6), June 1998

[HITZ95] M. Hitz and B. Montazeri, "Measuring coupling and cohesion in object orientated systems", *Proc. International Symposium forApplied Corporate Computing (ISACC '95), Monterrey, Mexico*, Oct. 25-27, 1995.

[HITZ96] M. Hitz and B. Montazeri, "Chidamber and Kemerer's metrics suite: A measurement perspective", *IEEE Transactions on Software Engineering*, 22(4), 1996, pp. 267-271.

[LAKE92] A. Lake and C. Cook, *"A software complexity metric for C++"*, Tech. Report, 92-60-03, Oregon State Univ., 1992

[LIEB88] K. Lieberherr, I. Holland and A. Riel, "Object-oriented programming: an objective sense of style", *Proc. Third Annual ACM Conference Object Orientated Prog., Syst., Lang. and Appl.* (OOPSLA), pp323-334, 1988

[LIST82] A. M. Lister, "Software Science - the emperor's new clothes?", *The Australian Computer Journal*, 14(2), 1995, pp. 66-70

[LORE94] M. Lorenz and J. Kidd, "Object-oriented software metrics", *Prentice Hall Object Orientated Series, Englewood Cliffs, NJ 07632*, 1994.

[LORE98] M. Lorenz, "OO Metric Descriptions", *http://www.hatteras.com/metr_des.htm*, 1998.

[MAYE98] T.G. Mayer, "A fresh look at object-oriented design metrics", *UKSMA 10th Anniversary Conference*, London, Oct. 1998

[MAYE99a] T. G. Mayer and T. Hall, "A critical analysis of current OO design metrics", *Software Quality management VII: Managing Quality*, C. Hawkins, G. King, M. Ross, G. Staples (Eds.), London, British Computer Society, 1999, pp.147-160

[MAYE99b] T. G. Mayer and T. Hall, "Measuring OO systems: a critical analysis of the MOOD metrics", *Procs. TOOLS Europe'99*, Nancy, France, 7-10 June 1999

[MCCA77] T. J. McCabe, "A complexity measure", *IEEE Transactions on Software Engineering*, SE-2(4), 1976, pp. 308-320

[MCCA94] T. J. McCabe, L. A. Dreyer, A. J. Dunn and A. H. Watson, "Testing an object-oriented application", *The Journal*, October 1994, pp. 21-27

[MCCA98] McCabe & Associates, "McCabe Object-Oriented Software Metrics", *http//:www.mccabe.com/features/complex*, 1998

[MOOD98] L. Ochoa and P. Catelas, "MOOD Project", *ftp://albertina.inesc.pt/pub/esw/mood/index.html*, 1998

[MORR89] K. L. Morris, "Metrics for object-oriented software development environments", Master's Thesis, M. I. T. Sloan School of Management, 1989

[RUSS91] M. Russell, "The mathematics of measurement in software engineering", *Formal Aspects of Measurement*, T. Denvir, R. Herman and R. Whitty (Eds), London, Springer-Verlag, 1992

[SHEN83] V. Y. Shen, S. D. Conte and H. E. Dunsmore, "Software Science revisited: A critical analysis of the theory and its empirical support", *IEEE Transactions on Software Engineering, SE-9(2)*, 1983, pp. 155-165

[SHEP88] M. J. Shepperd, "A critique of cyclomatic complexity as a software metric", *Software Engineering Journal*, March 1988, pp. 30-36.

[TEGA91] D. P. Tegarden, S. D. Sheetz and D. E. Monarchi, "Effectiveness of traditional software metrics for object-oriented systems", *Proceedings of the 25th Hawaii International Conference on System Sciences*, 1992

[TEGA92] D. P. Tegarden and S. D. Sheetz, "Object-oriented system complexity: an integrated model of structure and perceptions", *Proc. OOPSLA '92 Workshop on Metrics for OO Software Development*, 1992

[TEGA93] D. P. Tegarden, S. D. Sheetz and D. E. Monarchi, "A software complexity model of object-oriented systems", 1993

[WAND89] Y. Wand, "A proposal for a formal model of objects", in *Object-Oriented Concepts, Databases and Applications*, W. Kim and F. Lochovsky, Eds. Reading, MA: Addison-Wesley, 1989

[WEBE91] R. Weber and Y. Zhang, "An ontological evaluation of Niam's grammar for conceptual schema diagrams", in Proc. Twelfth International Conference on Information Systems, New York, 1991, pp 75-82

[WEGN90] P. Wegner, "Concepts and paradigms of object-oriented programming", *OOOPS Messenger* (1,1), pp 7-87, 1992

[WEYU88] E. Weyuker, "Evaluating software complexity measures", *IEEE Transactions on Software Engineering*, vol. 14, 1983, pp. 1357-1365

[ZUSE94] H. Zuse, "Foundations of the validation of object-oriented software measures", Research paper, Technische Universität, Berlin Germany

continued…

Hacking, Object-Orientation & General Software Engineering

[BESK98] J. Beskin, various emails, (personal correspondence with the author), 1998

[BOOC94] G. Booch, *Object-oriented analysis and design with applications (2nd Ed.)*, CA, USA: Addison-Wesley, 1994

[BROO95] F. P. Brooks, *The Mythical Man Month (2nd Ed.)*, Reading MA, USA: Addison-Wesley, 1995

[BUDD91] T. Budd, *An Introduction to Object-Oriented Programming*, Addison-Wesley Publishing Compnany, Inc., 1991

[CUSU96] M. A. Cusumano and R. W. Selby, *Microsoft Secrets*, HarperCollinsBusiness, UK, 1996

[DALC97] D. Dalcher, course notes for "Software Engineering Paradigms & Tools" unit, South Bank University, version: 1997

[DALC99] D. Dalcher, course notes for "Software Engineering Management" unit, South Bank University, version: 1999

[ETSM91] G. Etsminger, *The Tao of Objects*, CA, USA, M&T Publishing, Inc./England, Prentice Hall International, 1991

[FLAN97] D. Flanagan, *Java in a Nutshell (2nd Ed.)*, O'Reilly & Associates, Inc., 1997

[GLAS95] R. L. Glass, *Software Creativity*, Prentice Hall Inc., NJ, USA, 1995

[HANN98] G. Hannemyr, "Hacking Considered Constructive", (Position paper) *1997 Oksnøen Symposium on Pleasure and Technology*, revised 1.6.98

[HATT98] L. Hatton, "Does OO Sync with How We Think?", *IEEE Software*, May/June 1998

[HECK99] F. Hecker, "Setting up Shop: The Business of Open-Source Software", *IEEE Software*, Jan/Feb 1999

[LEVY84] S. Levy, *Hackers: Heroes of the Computer Revolution*, Dell Publishing, 1984 (Penguin Books, 1994)

[MCHU98] J. McHugh, "For the Love of Hacking", *Forbes Magazine*, August 1998

[MOOG98] G. Moody, "The Wild Bunch", *New Scientist*, 12 December 1998

[PARN86] D. L. Parnas and P. C. Clements, "A Rational Design Process: How and Why to Fake It", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986

[RAYM98a] E. S. Raymond, "The Cathedral and the Bazaar", http://www.tuxedo.org/~esr/writings/cathedral-bazaar/, latest version: Nov. 1998

[RAYM98b] E. S. Raymond, "Homesteading the Noosphere", http://www.tuxedo.org/~esr/writings/homesteading/, latest version: April 1998

[RAYM99] E. S. Raymond, "How To Become A Hacker", http://www.tuxedo.org/~esr/faqs/hacker-howto.html, latest version: March 1999

[RUBI98] D. M. Rubin, *Introduction to CRC Cards*, Softstar Research, Inc., revision: January 1998

[SOMM96] I. Sommerville, *Software Engineering* (*5th Ed.*), Addison-Wesley, 1996

[STAL85] R. M. Stallman, *The GNU Manifesto*, http:\\www.fsf.com, 1985

Index

J

L

M

N

O

P

Q

R

S

T

U

V

W

Z