An Analysis of OO Software Metrics

Jer Hogan*
Department of Computer Science,
University of Warwick,
Coventry,
CV4 7AL,
United Kingdom.

2nd May 1997

Abstract

This paper is a survey of software metrics literature with particular reference to reuse, quality, and productivity, and was prepared as part of the MOOD[1] project. The paper has three main parts: the literature survey, the selected metrics list, and the information to be gathered for process metrics. In the literature survey, papers which relate to metrics are outlined; in the selected metrics list some selected metrics are outlined. This paper is useful for the MOOD project or any project with broadly similar aims, such as to measure and improve reuse, productivity and software quality.

---

[0]Jer Hogan's email address is "Jer.Hogan@dcs.warwick.ac.uk". His world-wide web page is at "http://www.dcs.warwick.ac.uk/~jer/".

[1]MOOD stands for Metrics for Object-Oriented Developments. The MOOD project will compare and contrast the effectiveness of two different object-oriented methodologies for reuse, quality, and productivity. This work is being undertaken by Parallax Solutions Ltd. and the University of Warwick, as part of the EU funded European Systems and Software Initiative (ESSI). The MOOD project world-wide web home page is at "http://www.parallax.co.uk/mood/".

Contents

1 Introduction

The objective of the MOOD project is to provide a method to choose an object-oriented development methodology and predict project time, cost and quality metrics by:

- designing an integrated set of metrics for object-oriented developments, based on clear business requirements;
- applying these metrics to a project using two different methodologies, and measuring the effect of methodology on software quality, reuse and project performance; and
- updating the quality management system to apply the new method to all (Parallax) projects.

In Section 1.1 the remaining sections in this paper are briefly outlined. In Section 1.2 software quality is defined for the purposes of this paper.

1.1 Paper Outline

In Section 2 of this paper a literature survey is presented. This survey outlines papers we evaluated in order to help us in selecting metrics which measure reuse, quality, and productivity. The papers are summarised and evaluated. Software metrics bibliographies were particularly useful to us, in finding papers to evaluate. Object-oriented software metrics bibliographies include [Whitty 96] and [Dumke 96]. In Section 3, some metrics are categorised and outlined, and the conclusions and a bibliography follow. In Appendix A the information required for the process metrics is outlined. In Appendix B the five papers reviewed in Section 2 are evaluated as to their usefulness to the MOOD project.

1.2 Software Quality

Software quality has many definitions. It may be defined as the relative number of errors [Lim 94], or in terms of a number of different characteristics [Meyer 88] [Fung 96]. The software quality characteristics we will measure in the MOOD project are listed and defined in the following text:

Correctness:

How lacking in errors the system is (how close to specification the system is).

Maintainability:

How easy it is to fix errors.

Extensibility:

How easy it is to extend a software system (add a new feature).

Adaptability:

Degree of ease system shows in being adjusted to meet a changed or deleted requirement.

Efficiency:

Measures how optimised the software system is.

Robustness:

Measures the software systems response to unforeseen circumstances.

Reusability:

How reusable components and/or architecture of software is.

Portability:

How easy it is to transfer a software system to another platform.

Fenton and Pfleeger [Fenton 96] define internal attributes as follows; "Internal attributes of a product, process or resource are those that can be measured purely in terms of the product, process or resource itself. In other words, an internal attribute can be measured by examining the product, process or resource on its own, separate from its behaviour". Using this terminology, some internal product quality characteristics are as follows:

Complexity:

There are several different definitions of software complexity. An example is Chidamber and Kemerer's (C&K) Weighted Methods per Class (WMC) metrics [Chidamber 94] which collapses to a measure of the number of methods per class, if the weighting per method is 1. McCabe's cyclomatic complexity measures the number of decisions which gives an indication of the number of test cases.

Coupling:

Defined by Chidamber and Kemerer [Chidamber 94], as the amount of connections between classes. A connection between classes exists if one class uses a method or instance variable of another. High levels of coupling indicate lower levels of quality (Coupling is inversely proportional to quality). Note that coupling can be split into many different types, of differing quality. See section 3.3.12 for details.

Cohesion:

Defined by Fenton and Pfleeger [Fenton 96] for a module[2], as "the extent to which its individual components are needed to perform the same task". Cohesion is proportional to quality.

2 Literature Survey

In this section several papers are summarised. These are papers which are of use in selecting metrics under the headings of reuse and/or productivity and/or sub-headings of quality. This section is divided into sub-sections under the headings of what is being measured, such as reuse, productivity, correctness etc.. Within these sub-sections the papers themselves are summarised, each within its own sub-section. A general evaluation is then given for each paper.

2.1 Reuse and Reusability

2.1.1 [Bieman 92]

Reuse Measure Derivation As indicated in the title, this paper gave a description of how to derive measures of software reuse in object-oriented systems. The method used to derive measures of software reuse, is derived from measurement theory and was described as follows in section 2:

1. Identify and define intuitive and well-understood attributes of software reuse. We must qualitatively understand what we want to measure.
2. Specify precisely the documents and the attributes to be measured. We must be able to describe precisely the object that we measure, and the property of the object that the measurement is to indicate.
3. Determine formal models or abstractions which capture these attributes. Formal models are required to unambiguously produce numerical measurement values.
4. Derive mappings from the models of attributes to a number system. The mappings must be consistent with the orderings implied by the model attributes.

Reuse Definitions Some definitions were then presented. *Public reuse* was defined by Fenton as "the proportion of a product which was constructed externally" [Fenton 91]. In other words:

$$Public\, reuse = \frac{length(E)}{length(P)}$$

where E is the code defined externally (libraries etc.) and P is the complete system including E. *Private reuse* was defined by Fenton as the "extent to which modules within

[2] The OO equivalent is a class.

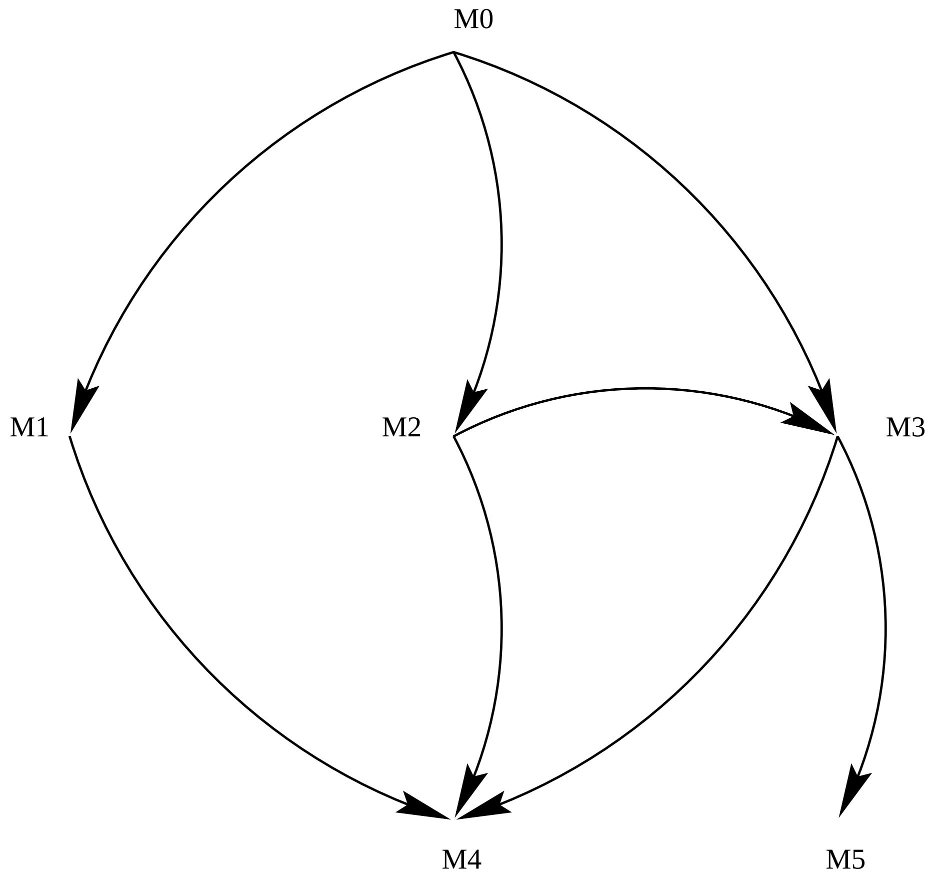

Figure 1: An example of a Call Graph

a product are reused within the same product" [Fenton 91]. In the call graph in Figure 1 there are three instances of private reuse. M4 is used three times (reused twice), and M3 is used twice (reused once). M0, M1, M2, and M5 are only used once each, and are therefore not reused. The measure of private reuse is given by:

$$r(G) = e - n + 1$$

where e is the number of edges and n is the number of nodes in the call graph (example in Figure 1). e is also the number of calls made, and n is also the number of routines. Therefore the level of private use is e. To formulate reuse (use more than once) from use it is required to take away a value corresponding to the first instance of use (calling of) of each routine. This value is n - 1 as we assume that in non-trivial systems each routine is called at least once, except the main routine. This gives the formula shown previously for r(G). *Verbatim reuse* means reusing a module/class fully. *Leveraged reuse* is reuse where the module/class is partially reused. An example is reuse by inheritance, where a class's methods can be overridden. *Direct reuse* is reuse without going through an intermediate module/class. *Indirect reuse*, on the other hand, is reuse through an intermediate module/class. For example, if module A invokes module B, which invokes module C, then A indirectly reuses C and C is indirectly reused by A. *Reuse perspectives* are perspectives gained by considering reuse from different viewpoints. The *server perspective* is the perspective of a library or a particular library component. Examples of measures of server perspective software reuse attributes are the number of times a library component is reused, average for library etc. The *client perspective* takes the point of view of the new system or a new system component. Examples of measures include the number of reuse occurrences in the system, the percentage of a system which is reused code, kinds of reuse etc.. The *system perspective* is a view of reuse in the overall system, both servers and clients. It can include types of reuse such as; system-wide private reuse, system-wide public reuse, indirect reuse, direct reuse, and verbatim reuse.

Object Oriented Reuse Object oriented reuse is then considered in the paper. In object-oriented terms public and private reuse are not considered separately in [Bieman 92] due to Bieman's difficulty in identifying external code. Bieman described this difficulty as due to the fact that "a new system is built as an extension to an existing class library". As described earlier, in object-oriented terms, leveraged reuse occurs through inheritance, when some method(s) in a class are overridden. In inheritance either a method, or methods, is/are modified or method(s)/ variable(s) are added to a class, or trivially the class is effectively the same as its ancestor. Bieman considered all non-trivial inheritance as partial, or leveraged reuse, even if the inheritance is extending rather than modifying the ancestor class. Verbatim server reuse can be measured as the number of instance creations of objects of class A in a new system and the number of instance references of class A objects in a new system. This can be determined dynamically, or statically. For public reuse one method/class usage is a reusage whereas for private reuse two class/method usages is one reusage (and three class/ method usages is two reusages). Leveraged server reuse occurs through inheritance and can be measured using an inheritance graph, which graphs inheritance relations as a tree. In the example shown in Figure 2 class Undergrad inherits from class Student, which inherits from class Person. Of course for Bieman non-trivial inheritance is always leveraged reuse. The object oriented client reusing profile can measure verbatim reuse within class A, by measuring the number of instance creations of library objects, and the percentage of objects created or referenced by A, which are from the library. Indirect client reuse measures can also be made. Leveraged reuse measures from the client perspective include the number of servers, the number of paths to indirect servers, and the average lengths of such paths. Object oriented system reuse profile measures include (as described in section 4.5):

- Percentage of the new system source text imported from the library. This requires information not contained in the call multi-graph, information related to class lengths.
- Percentage of new system classes imported verbatim from library.
- Percentage of the new system classes derived (leveraged reuse) from library classes, and the average percentage of these leveraged classes that are im-

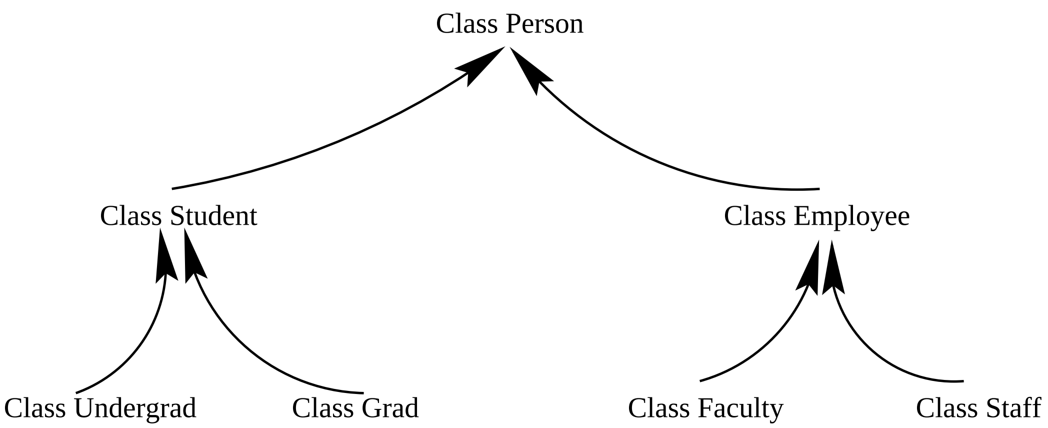

Figure 2: An Example of an Inheritance Graph

ported.

- The average number of verbatim and leveraged clients for servers, and the average number of indirect servers for clients.
- The average length and number of paths between indirect servers and clients for both verbatim and leveraged reuse.

Internal software properties which may affect reuse include:

Class size: A large class (inherited definitions are taken into account) may be harder to understand, and thus more difficult to reuse. Leveraged reuse of a larger sized class may be especially difficult.

Nature and "complexity" of control flow: A class with a complicated decision structure in the implementation of a class method may also be more difficult to reuse, especially if modifications are necessary.

Nature and "complexity" of data flow: Many data dependencies may also make reuse more difficult.

Size and "complexity" of interfaces: Many speculate that a large and complicated interface makes reuse more difficult. Bieman suspects that interface complexity will affect direct reuse of a server entity more than the above internal attributes.

In conclusion, Bieman pointed out that there is a need to know "where we stand" in relation to software reuse, which the measures he proposed would help to gauge.

Evaluation In this paper several useful definitions of reuse were presented. Even if they prove not to be directly useful they are a useful starting point for reuse measures. Lots of different reuse measures were presented in this paper, under three different perspectives;

HP Division	Manufacturing Productivity	San Diego Tech. Graphics
Quality	51% defect reduction	24% defect reduction
Productivity	57% increase	40% increase
Time to market	Data not available	42% reduction

Table 1: Quality, Productivity, and Time-to-market Profiles [Lim 94]

server, client, and system. The server perspective is that of an item to be reused. The client perspective is that of an item reusing another item. The system perspective is the overall system view. Again these measures are useful in understanding the issues involved in formulating reuse measures, and in providing a starting point for reuse measures.

2.2 Productivity and Correctness

2.2.1 [Lim 94]

Introduction As the title suggests, Lim presented some figures showing the effects of reuse on quality, productivity, and economics in this paper. The effects were largely positive in all three cases in the two projects evaluated. The projects were carried out by different divisions of Hewlett-Packard. The two HP divisions used different languages (Pascal and a HP proprietary language vs. C) for different types of applications (software for manufacturing resource planning vs. firmware for plotters and printers) which means that they are quite independent. Therefore the two case studies should provide twice as much proof for or against the hypothesis that reuse is "good" for quality, productivity, and economics, if the results agree.

Definitions Firstly some terms were defined for the article. *Work products* are the products or by-products of the software development process: for example, code design and test plans. *Reuse* is the use of these work products without modification in the development of other software. *Leveraged reuse* is modifying existing work products to meet specific system requirements. A *producer* is a creator of reusable work products, and the *consumer* is someone who uses them to create other software. *Time to market* is the time it takes to deliver a product from the time it is conceived. The two case studies were then presented.

Results The overall improvements due to reuse were impressive, as shown in Table 1. The number of defects per thousand non-comment source statements was reduced when considering reused code combined with new code, as against new code. See Table 2. Similarly the productivity increased when considering combined reused and new code against new code. See Table 3. Figure 3 in page 25 of the paper shows a roughly proportional relationship between amount of reuse and productivity in a third HP division, measured from

Manufacturing Productivity - New Code Only	4.1
Manufacturing Productivity - New and Reused Code	2.0
San Diego Tech. Graphics - New Code Only	1.7
San Diego Tech. Graphics - New and Reused Code	1.3

Table 2: Effects of reuse on software quality - as measured by defects per thousand non-comment source statements - in new code only versus new code combined with reused code [Lim 94]

Manufacturing Productivity - New Code Only	0.7
Manufacturing Productivity - New and Reused Code	1.1
San Diego Tech. Graphics - New Code Only	0.5
San Diego Tech. Graphics - New and Reused Code	0.7

Table 3: Effects of reuse on productivity - as measured by thousands of non-comment source statements produced per engineering month - in new code only versus new code combined with reused code [Lim 94]

1985 to 1990. The paper then went into the costs of reuse; creating or purchasing work products, libraries, and tools and implementing reuse related properties. Several papers were reviewed from the point of view of costs of reuse. The reuse program economic profiles were presented, as shown in Table 4. A reuse cost-benefit analysis was presented for the two HP divisions over 10 years in Figure 5 on page 28 of the paper. Further graphs were presented showing the savings in days and dollars which HP has achieved through reuse. From this paper it is evident that reuse is a very positive experience for Hewlett-Packard (who are of course a prestigious electronics company).

Evaluation In this paper two programs were outlined, which both reused code and achieved increased productivity and increased quality (less errors). These figures were compared with all new code (none reused) and this showed the positive effect of reuse on productivity and quality. A substantial return on investment for reuse was shown, in two different industrial projects. Useful and practical measures and measurement values were displayed for productivity, correctness and return on investment.

2.3 Product - Complexity, Cohesion and Coupling

2.3.1 [Chidamber 94]

Introduction As the title suggests, in this paper the authors (C&K) presented a metrics suite for object-oriented design. The theoretical basis for the metrics was the ontology of

	Manufacturing Productivity	San Diego Tech. Graphics
Time horizon	1983-1992 (10 years)	1987-1994 (8 years) 1994 data estimated
Start-up resources required	26 eng. months (start-up costs for 6 products) $0.3 million	107 eng. months (about 3 eng.s for 3 years) $0.3 million
Ongoing resources required	54 eng. months (about 0.5 eng. for 9 years) About $0.3 million	99 eng. months (about 1-3 eng.s for 5 years) About $0.7 million
Gross cost	80 eng. months ($1.0 million)	206 eng. months ($2.6 million)
Gross savings	328 eng. months ($4.1 million)	446 eng. months ($5.6 million)
Return on Investment (savings/cost)	410%	216%
Net present value	125 eng. months ($1.6 million)	75 eng. months ($0.9 million)
Break-even year (recoup start-up costs)	2nd year	6th year

Table 4: Reuse Program Economic Profiles [Lim 94]

Bunge [Bunge 77] [Bunge 79]. Six metrics were developed, and evaluated against Weyuker's proposed set of measurement principles [Weyuker 88]. An automated data collection tool was then developed in order to collect an empirical sample of this data at two field sites. The authors point out that since object orientation (OO) is a relatively new method of programming, there is a need for metrics as an organisation implementing a new object-oriented system will most likely not have developed established practices. The authors mentioned criticism which has been made of previous software metrics - such as;

- They lack a theoretical base.
- They lack desirable measurement properties.
- They are insufficiently generalised or too implementation technology dependent.
- They are too labour-intensive to collect.

The authors mentioned that since OO is a new technology, new metrics are required to measure key OO concepts such as;

- Classes.
- Inheritance.
- Encapsulation.
- Message passing.

The authors quickly surveyed other papers on OO metrics. The importance of class design in OO is then outlined and the selection of static class metrics is justified on the grounds of the importance of class design versus implementation. The authors then went into quite a bit of detail on the measurement theory on which they base their metrics (Bunge's ontology) [Bunge 77] [Bunge 79] and the properties (Weyuker's) [Weyuker 88] against which the metrics were evaluated. Some criticism of Weyuker's properties were outlined, but since they were a widely known formal analytical approach, they were used. The six metrics were then presented under a number of headings;

- Definition.
- Theoretical Basis.
- Viewpoints.
- Analytical Evaluation.
- Empirical Data.

Site	Metric	Median	Max	Min
A	WMC	5	106	0
B	WMC	10	346	0

Table 5: Summary Statistics for the WMC metric [Chidamber 94]

WMC Metric The first metric, Weighted Methods per Class (WMC), was then defined. Consider a class C_1 with methods $M_1, .., M_n$. Let $c_1..c_n$ be the complexity of the methods. Then:

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

The method complexity measure used is left to the developer. If the complexity (weighting) per method is unity then WMC is the number of methods per class. Some viewpoints, or reasons why the metric was chosen were then presented. The reasons given in section VI were:

1. The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class.
2. The larger the number of methods in a class the greater the potential impact on children, since children inherit all the methods defined in the class.
3. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

The results of the measurements at the two sites were then presented. These are summarised in Table 5. Most classes were found to have a small number of methods (0 to 10).

DIT Metric The next metric, Depth of Inheritance Tree (DIT), was then presented. DIT was defined as the maximum length from the node to the root of the tree. The following reasons were given for using the DIT metric in section VI;

1. The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behaviour.
2. Deeper trees constitute greater design complexity, since more methods and classes are involved.
3. The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods.

The empirical data for the DIT metric was then presented. Table 6 shows the summary statistics. In both cases the bulk of classes had a DIT value of 5 or lower. The authors suggested that a deeper value would show a more thorough use of inheritance.

Site	Metric	Median	Max
A	DIT	1	8
B	DIT	3	10

Table 6: Summary Statistics for the DIT metric [Chidamber 94]

Site	Metric	Median	Max	Min
A	NOC	0	42	0
B	NOC	0	50	0

Table 7: Summary Statistics for the NOC metric [Chidamber 94]

NOC Metric The Number of Children (NOC) metric was presented next. It was defined as the number of immediate subclasses subordinated to a class in the class hierarchy. The following reasons were given for using the NOC metric in Section VI;

1. The greater the number of children, the greater the reuse, since inheritance is a form of reuse.
2. The greater the number of children, the greater the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of sub-classing.
3. The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children, more testing may be required of the methods in that class.

The empirical data for the NOC metric was then presented. Table 7 shows the summary statistics for the NOC metric. The most common value by far for NOC was 0. It was suggested that this showed a low level of reuse.

CBO Metric The Coupling Between Object classes (CBO) metric was then presented. It was defined for a class as a count of the number of other classes to which it is coupled. A class was said to be coupled with another class if either one accessed the others methods, or variables. The following reasons were given for using the CBO metric in section VI;

1. Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application.
2. In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the

Site	Metric	Median	Max	Min
A	CBO	0	84	0
B	CBO	9	234	0

Table 8: Summary Statistics for the CBO metric [Chidamber 94]

higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult.

3. A measure of coupling is useful to determine how complex the testing of various parts of a design is likely to be. The higher the inter-object class coupling the more rigorous the testing needs to be.

It should be noted that C&K's coupling measure doesn't differentiate between the many different types of coupling. See section 3.3.12 for more details. The empirical data for the CBO metric was then presented - see Table 8 for summary statistics. Site B, which had a Smalltalk application, showed larger values than Site A, which had a C++ application. The authors suggested that this is because Smalltalk treats most things as objects, including simple scalar variables and control flow constructs.

RFC Metric The Response for a Class (RFC) metric was then presented. RFC is defined as $|RS|$ where RS is the response set for a class. The response set for a class can be expressed as:

$$\begin{aligned} RS &= \{M\} \bigcup_{all\,i} \{R_i\} \\ where\ \{R_i\} &= set\ of\ methods\ called\ by\ method\ i \\ and \{M\} &= set\ of\ all\ methods\ in\ the\ class. \end{aligned}$$

The response set of a class is a set of methods that can be potentially be executed in response to a message received by an object of that class. Membership of the response set is defined only up to the first level of nesting of method calls due to the practical considerations involved in collection of the metric. The following reasons were given for using the RFC metric in section VI;

1. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding from the tester.
2. The larger the number of methods that can be invoked from a class, the greater the complexity of the class.
3. A worst case value for possible responses will assist in appropriate allocation of testing time.

Site	Metric	Median	Max	Min
A	RFC	6	120	0
B	RFC	29	422	3

Table 9: Summary Statistics for the RFC Metric [Chidamber 94]

The empirical data was then presented. See Table 9 for the summary statistics. Site B (Smalltalk application) RFC values were higher than site A (C++ application) values which again was probably due to Smalltalk's treating virtually everything as objects.

LCOM Metric The Lack of Cohesion in Methods (LCOM) metric was then presented. If a class C_1 has n methods $M_1, .., M_n$ then $\{I_j\}$ is the set of instance variables used by method M_j. Let $|P|$ be the number of null intersections between instance variable sets. Let $|Q|$ be the number of non-empty intersections between instance variable sets. Then:

$$
\begin{aligned}
LCOM &= |P| - |Q|, if |P| > |Q| \\
LCOM &= 0\, otherwise
\end{aligned}
$$

LCOM measures the amount of method pairs which don't access the same instance variable minus the amount of method pairs which do access the same instance variable. As such it is a measure of lack of cohesion (the negative of cohesion). The following reasons were given for using the LCOM metric in section VI;

1. Cohesiveness of methods within a class is desirable, since it promotes encapsulation.
2. Lack of cohesion implies classes should probably be split into two or more classes.
3. Any measure of disparateness of methods helps identify flaws in the design of classes.
4. Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

The empirical data for LCOM was then presented. See Table 10 for the summary statistics. Both sites showed low values on the whole for LCOM indicating a high degree of cohesion between class methods.

Usefulness of the 6 Metrics Reasons were then presented as to why the six chosen metrics were good for measuring the Booch OO design (OOD) methodology. The authors cite Sharble and Cohen's paper [Sharble 93] as evidence that their (Chidamber and Kemerer's) metrics are being used by leading-edge organisations. The authors point out that

Site	Metric	Median	Max	Min
A	LCOM	0	200	0
B	LCOM	2	17	0

Table 10: Summary Statistics for the LCOM Metric [Chidamber 94]

it is possible to use their metrics to highlight differences between OO languages and environments. Another possible use outlined for the metrics is to analyse the degree to which they correlate with managerial performance indicators, such as design, test, and maintenance effort, quality, and system performance. The authors point out that Li and Henry [Li 93] found that the 6 metrics correlated better with maintenance effort than traditional metrics.

Evaluation The authors presented six object-oriented metrics which have become established, been extended [Churcher 95] [Hitz 96] and been used widely [Sharble 93] [Li 93]. These metrics have a measurement theory base, although C&K mentioned some criticism of it. These metrics are an extremely useful "base suite" from which useful object-oriented metrics can be developed.

2.3.2 [Churcher 95]

There has been some criticism of [Chidamber 94]. Churcher and Shepperd's (C&S) main ground for criticism was the ambiguity of some of the metrics. The main metric they found ambiguous was WMC, which if the weighting is unity, is a count of the number of methods in a class. C&S pointed out that there are two factors for C++ methods which C&K didn't specify; whether constructor/ destructor methods were all counted (despite having the same name in a class) and whether operators are included as methods.

Evaluation C&S presented a criticism of C&K's WMC metric based on the fact that the number of methods in a C++ class is ambiguous. C&K answer this by stating explicitly their way of counting the number of C++ methods. This paper was a useful criticism, emphasising the necessity for precise and explicit definition.

2.3.3 [Hitz 96]

Hitz and Montazeri (H&M) criticised two of C&K's metrics from a measurement theory perspective.

CBO Metric H&M stated that the CBO metric has not got a sound empirical relation system [Hitz 96]. In other words H&M believe that C&K do not differentiate bewteen the different types of coupling. H&M listed the following empirical relations not met by the CBO metric in section 2;

- Access to instance variables constitutes stronger coupling than pure message passing.
- Access to instance variables of foreign classes constitutes stronger coupling than access to instance variables of superclasses (the same holds, *mutatis mutandis,* for message passing).
- Passing a message with a wide parameter interface yields stronger coupling than passing one with a slim interface.
- Violating the Law of Demeter yields stronger coupling than restricted (Demeter conforming) message passing. The Law of Demeter, in its so-called "object" version is defined for C++ as follows: For all classes C, and for all member functions M attached to C, all objects to which M sends a message must be one of:
  1. M's argument objects, including this.
  2. A data member object of class C.
- Couples to the following types of objects should yield increasing coupling values: local objects, method parameters, sub-objects of self (= instance variables of class type), sub-objects of a super class (= inherited instance variables of class type), global objects.

LCOM Metric H&M showed cases where the LCOM metric represents intuitively equivalent cases differently. They then proposed their own graph theoretic version of LCOM. In working out H&M's LCOM for a system the first step is to graph all methods as nodes, linking nodes where the methods share at least one instance variable. H&M's LCOM is simply the number of separate method clusters shown in the graph. Where the LCOM value is 1, the cohesion can still be differentiated by counting the number of edges in the graph. H&M presented a normalised version of this as follows:

$$c = 2\frac{|E| - (n-1)}{(n-1)(n-2)}$$

where $|E|$ is the number of links (edges) and n is the number of methods. For example if methods M_1, M_2, and M_3 each access I_1, and I_2 instance variables only and the other three methods M_4, M_5, and M_6 each access instance variable I_3 only then H&M's LCOM value is 2 as there are two separate "clusters". Intuitively H&M's LCOM makes more sense than C&K's.

Measure Design Diagram H&M then presented a diagram (Figure 5 [Hitz 96]) showing an overview from their point of view of a commonly agreed procedure [Fenton 91] to design useful measures for assessing attributes used in software development.

Evaluation H&M presented a criticism of C&K's CBO and LCOM metrics. They proposed a CBO metric with more granularity and that also made more intuitive sense. Similarly they proposed an LCOM metric which made more intuitive sense than C&K's. The new metrics proposed comprise a useful extension to C&K's metrics suite.

3 A Classification of Metrics

This section lists the selected metrics which could be measured for the MOOD project. This section has four main sub-sections. These are Selected Requirements Metrics, Selected Design Metrics, Selected Code Metrics and Selected Estimation Metrics. As well as the surveyed papers [Lorenz 94] and [Henderson 96] proved useful in selecting metrics.

3.1 Selected Requirement Metrics

The requirement metrics are the numbers of the item listed except where otherwise specified. Individual requirements may vary widely, even though they should be unambiguous and clear. Therefore it is open to debate how effective the measures in this section are. The metrics are as follows:

3.1.1 Business Requirements

These are the general requirements of a system, which detail its functionality. Each requirement details one part of the functionality clearly and unambiguously. A count of the business requirements gives an idea of the size of the functionality although of course the same functionality could be represented by different numbers of business requirements by different people, and not all business requirements are equivalent in size or complexity.

3.1.2 Technical Requirements

These requirements are related to technical issues. An example would be: "Business requirement X must take less than 3 seconds to execute.". A high number of technical requirements would indicate a complex application requiring very careful implementation work.

3.1.3 Inputs

Inputs are anything required from the user for the system to work. An example for a compiler would be a source file. A high number of inputs does indicate a highly complex application.

3.1.4 Outputs

Outputs are anything produced by the system. An example for a linker would be an executable file. A high number of outputs tends to mean the system is very complex.

3.1.5 Maintenance Requirements

These requirements indicate details of maintenance for the system. An example would be: "This system must have all errors considered serious by the customer corrected in a new version within one week of being found.". System complexity will increase with this measure.

3.1.6 Testing Requirements

These requirements relate to testing. Examples would be: "This system must be tested by test engineer(s).", and "This system must receive two person-weeks of testing after the developers consider it ready". Higher amounts of this measure should indicate high quality.

3.2 Selected Design Metrics

The selected design metrics listed in this section are designed with the Booch methodology and the C++ language in mind. The design metrics are the numbers of the item listed except where otherwise specified. There is nothing in the surveyed literature about combining these design metrics into a complex measure.

3.2.1 Class Diagram Metrics

- All Class Associations
- Inheritance Associations
- "Has" Associations
- "Using" Associations
- Instantiations
- Friend Associations
- Virtual Associations
- Static Associations
- Public Accesses
- Protected Accesses
- Private Accesses
- Implementation Accesses
- Nested Classes
- Maximum Class Nesting Depth

- Abstract Classes
- Classes
- Parameterized Classes
- Attributes per Class
- Operations per Class
- Constraints per Class
- Primitive (not containing sub-states) States (in State Machine) per Class
- Primitive Transitions (in State Machine) per Class
- Parameters per Class (if parameterized)
- Transient Classes
- Persistent Classes (Classes, some of whose objects continue to exist after the program has exited).
- Arguments per Operation
- Static Operations
- Virtual Operations
- Pure Virtual Operations
- Const Operations
- Public Operations
- Private Operations
- Protected Operations
- Implementation Operations
- Pre-Conditions per Operation
- Post-Conditions per Operation
- Exceptions per Operation
- Storage per Operation
- Time per Operation
- Total Operation Storage

3.2.2 Object Diagram Metrics

- Object Diagrams
- Methods Called
- Links
- Global Links
- Parameter Links
- Field Links
- Local Links
- Objects
- Global Object Diagrams
- Category Object Diagrams
- Class Object Diagrams
- Operation Object Diagrams

3.2.3 Interaction Diagram Metrics

- Message Passes
- Objects

3.2.4 Module Diagram Metrics

- Modules
- Dependencies

3.3 Selected Code Metrics

The selected code metrics listed in this section are divided into sub-sections of reuse, productivity and quality which are the criteria the MOOD objective says should be measured and predicted. Some further explanation of a metric is given where required.

3.3.1 Reuse Metrics

Reuse Rate: The percentage of functions and classes which are being reused. In this category are unmodified and less than 25% modified functions and classes. The 25% figure is that used by NASA/Goddard's Software Engineering Laboratory as reported by Fenton and Pfleeger [Fenton 96]. For this measure we can define the total modified lines as the sum of the total added lines and the total deleted lines and the total changed lines.

Class Internal Reuse Rate: The number of times the methods in the class are called outside the class divided by the number of public methods in the class.

Internal Reuse Rate: The sum of the CIR over all classes divided by the number of classes.

3.3.2 Productivity Metrics

Project Productivity: The number of person-hours taken to meet the project functionality divided by the size in NCSS. See section 3.3.14.

Overall Productivity: The size of product (code, design etc.) produced per person-hour. This can be measured against time in order to take account of any learning-curve.

Class Productivity: The effort in person-hours required per class. This measure can be compared with class internal reuse to see if there's any correlation.

3.3.3 Correctness Metrics

Number of Errors: Errors are detected deviations from the specification. Fenton and Pfleeger [Fenton 96] call errors seen by the user failures, because they are failures to meet the specification.

Number of Corrections: F&P [Fenton 96] call these faults. Each error requires some number of separate places in the code to be corrected. Each one counts as a correction.

3.3.4 Maintainability Metrics

Code Maintainability: The size of the code which is changed or added per error.

Overall Maintainability: Time taken in person-hours to fix all errors encountered.

Error Maintainability: Average time taken in person-hours to fix each error.

Correction Maintainability: Time taken in person-hours per correction.

Prediction of Maintainability: Li and Henry [Li 93] were able to correlate some metrics including those of Chidamber and Kemerer [Chidamber 94] with maintenance data obtained on two Ada systems. Further work could lead to the prediction of maintainability measures based on C&K's and other metrics.

3.3.5 Extensibility Metrics

The time taken in person-hours to add completely (including design, code, test, debug and regression test) a selected new feature. In most projects new requirements will be added after design or even coding has started. If this happens in this project we can measure the extensibility.

3.3.6 Adaptability Metrics

The time taken in person-hours to change an existing requirement. If an existing requirement changes in our project we can measure adaptability.

3.3.7 Efficiency Metrics

Speed Efficiency: The time taken for the same task by two different systems of similar functionality.

Size Efficiency: The code sizes for the same functionality/task.

Memory Efficiency: The amount of memory in bytes required for the same task.

Space Efficiency: The amount of disk space in bytes required for the same task.

3.3.8 Robustness Metrics

The percentage of selected non-specified occurrences gracefully handled. The non-specified occurrences will be selected and the projects will be tested against them.

3.3.9 Reusability Metrics

Judith Barnard, a colleague on this project, has looked at reusability and selected several useful metrics:

- Number of comment lines per class
- Number of words of text documentation on the class
- Number of instance variables in the class
- Number of class variables in the class

3.3.10 Portability Metrics

The percentage of non-portable classes/non-comment source statements. "Non-portable" will be defined for each language. For C++ it means non-ANSI draft standard.

3.3.11 Cohesion Metrics

Chidamber and Kemerer's LCOM (lack of cohesion in methods) metric: See earlier or [Chidamber 94] for more details.

Hitz and Montazeri's LCOM (lack of cohesion in methods) metric: This counts separate method clusters in a graph of the methods linked by the instance variables they use. For instance if methods M1, M2, and M3 each access I1 and I2 instance variables only and M4, M5, and M6 each access I3 instance variable only then the LCOM is 2. See earlier or [Hitz 96] for more details.

Martin's H (relational cohesion) metric: This is a measure of cohesion within a class category (a group of related classes). The measure is the average number of internal relationships per class. Let R be the number of class relationships that are internal to the category. Let N be the number of classes within the category. Then $H = (R + 1)/N$. See [Martin 95] for more details.

3.3.12 Coupling Metrics

Chidamber and Kemerer's CBO (Coupling Between Object classes) metric: See earlier or [Chidamber 94] for more details.

Li and Henry's MPC (Message-passing coupling) metric: MPC is defined as the number of send statements defined in a class. See [Li 93] for more details.

Li and Henry's DAC (Data Abstraction Coupling) metric: DAC is defined as the number of classes which are types (or part of types) of variables declared in a class. See [Li 93] for more details.

Hitz and Montazeri CBO metrics Hitz and Montazeri [Hitz 96] have proposed adding granularity to Chidamber and Kemerer's [Chidamber 94] CBO metric. The following is a list of metrics based on Hitz and Montazeri's categories of coupling:

CBO1 metric: The number of message passes to objects of a super class. The number of parameters + 1 is added to CBO1 for each message pass.

CBO2 metric: The number of message passes to objects of a foreign class. The number of parameters + 1 is added to CBO2 for each message pass.

CBO3 metric: The number of accesses of an instance variable of super class object.

CBO4 metric: The number of accesses of an instance variable of foreign class object.

CBO5 metric: The number of accesses of a local object.

CBO6 metric: The number of accesses which are violations of the Law of Demeter.

CBO7 metric: The number of accesses to a method parameter object.

CBO8 metric: The number of accesses to a sub-object of self.

CBO9 metric: The number of accesses to a sub-object of a super-class.

CBO10 metric: The number of accesses to a global object.

Hitz and Montazeri Definitions The following are definitions of some of the terms used by Hitz and Montazeri. See earlier or [Hitz 96] for more details.

- Super Class: An ancestor class in inheritance terms.
- Foreign Class: Neither this class nor a super class.
- Local Object: An object whose scope is local to this class.
- Law of Demeter: The LOD ("object version") states that for all classes C and all member functions M attached to C, all objects to which M sends a message must be one of :
    1. M's argument objects including this in C++.
    2. A data member object of class C.
- Sub-object of Self: An instance variable of class type.
- Sub-object of a Super Class: An inherited instance variable of class type.

Martin's C_a metric (afferent coupling): This measure is given by the number of classes from other categories that depend upon classes within the subject category. Dependencies are class relationships. For more details see [Martin 95].

Martin's C_e metric (efferent coupling): This measure is given by the number of classes from other categories that classes within the subject category depend upon. Dependencies are class relationships. For more details see [Martin 95].

Abstract and Concrete Coupling All the measures in this section can be split into abstract and concrete measures where an abstract measure involves coupling with at least one abstract class. Concrete coupling leads to lower quality than abstract coupling.

3.3.13 Complexity Metrics

McCabe's Cyclomatic Complexity: This is described in [Basili 94]. The value is given by the number of decisions + 1 where a decision is the conditional of an if, while etc.

Chidamber and Kemerer's WMC (Weighted Methods per Class) metric: See earlier or [Chidamber 94] for more details. The weighting can be the McCabe number or a measure of size like SLOC or the number of parameters.

3.3.14 Size Metrics

Number of Classes

NCSS: Number of non-comment source statements.

SLOC: Number of source (non-comment) lines of code.

Number of Attributes: Public, Private and Protected.

3.3.15 Inheritance Metrics

Chidamber and Kemerer's DIT (Depth of Inheritance Tree) and NOC (Number of Children) metrics: See earlier or [Chidamber 94] for more details.

3.3.16 Confidence Metrics

Developer Confidence metric: Rating of confidence in system given by developer from 1 to 5.

3.3.17 Miscellaneous Metrics

Chidamber and Kemerer's RFC (Response for a Class) metric: See earlier or [Chidamber 94] for more details.

Martin's A (abstractness) metric: This is the ratio of the number of abstract classes in the category to the total number of classes in the category. See [Martin 95] for more details.

Li and Henry's NOM metric: The number of methods in a class. See [Li 93] for more details. This can be sub-divided into numbers of public, private, and protected methods.

3.4 Estimation Metrics

Function Points: See [Fenton 96] p258-265 for more details. Fenton & Pfleeger mention some problems in theory and practice with function point analysis but they also mention that function points have been used to good effect in a number of industrial applications.

3.5 Classification Table

3.5.1 Key

Se. - Section This is the section of this document in which the metric is described.

T. - Type This is the type of the metric. The possible types are defined by Fenton and Pfleeger [Fenton 96] as follows :-

- [*Ps. – Process.*] Processes are collections of software-related activities.
- [*Pt. – Product.*] Products are any artifacts, deliverables or documents that result from a process activity.
- [*Re. – Resource.*] Resources are entities required by a process activity.

I/E - Internal/External This is defined by Fenton and Pfleeger [Fenton 96] as follows :-

- **Internal attributes** of a product, process or resource are those that can be measured purely in terms of the product, process or resource itself. In other words, an internal attribute can be measured by examining the product, process or resource on its own, separate from its behaviour.
- **External attributes** of a product, process or resource are those that can be measured only with respect with how the product, process or resource relates to its environment. Here, the behaviour of the product, process or resource is important, rather than the entity itself.

Sc. - Scale

- N - Nominal Scale. Fenton and Pfleeger [Fenton 96] list the following major characteristics of the nominal scale:-
  - The empirical relation system consists only of different classes; there is no notion of ordering among the classes.
  - Any distinct numbering or symbolic representation of the classes is an acceptable measure, but there is no notion of magnitude associated with the numbers or symbols.

Metric	Se.	T.	I/E	Sc.	E.	S/O	M/P
# Business Requirements	3.1.1	Pt.	I	A	1	O	M
# Technical Requirements	3.1.2	Pt.	I	A	1	O	M
# Inputs	3.1.3	Pt.	I	A	1	O	M
# Outputs	3.1.4	Pt.	I	A	1	O	M
# Maintenance Requirements	3.1.5	Pt.	I	A	1	O	M
# Testing Requirements	3.1.6	Pt.	I	A	1	O	M
# All Class Associations	3.2.1	Pt.	I	A	1	O	M
# Inheritance Associations	3.2.1	Pt.	I	A	1	O	M
# "Has" Associations	3.2.1	Pt.	I	A	1	O	M
# "Using" Associations	3.2.1	Pt.	I	A	1	O	M
# Instantiations	3.2.1	Pt.	I	A	1	O	M
# Friend Associations	3.2.1	Pt.	I	A	1	O	M
# Virtual Associations	3.2.1	Pt.	I	A	1	O	M
# Static Associations	3.2.1	Pt.	I	A	1	O	M
# Public Accesses	3.2.1	Pt.	I	A	1	O	M
# Protected Accesses	3.2.1	Pt.	I	A	1	O	M
# Private Accesses	3.2.1	Pt.	I	A	1	O	M
# Implementation Accesses	3.2.1	Pt.	I	A	1	O	M
# Nested Classes	3.2.1	Pt.	I	A	1	O	M
Maximum Class Nesting Depth	3.2.1	Pt.	I	R	1	O	M
# Abstract Classes	3.2.1	Pt.	I	A	1	O	M
# Classes	3.2.1	Pt.	I	A	1	O	M
# Parameterized Classes	3.2.1	Pt.	I	A	1	O	M
# Attributes per Class	3.2.1	Pt.	I	A	1	O	M
# Operations per Class	3.2.1	Pt.	I	A	1	O	M
# Constraints per Class	3.2.1	Pt.	I	A	1	O	M
# Primitive States per Class	3.2.1	Pt.	I	A	1	O	M
# Primitive Transitions per Class	3.2.1	Pt.	I	A	1	O	M
# Parameters per Class	3.2.1	Pt.	I	A	1	O	M
# Transient Classes	3.2.1	Pt.	I	A	1	O	M
# Persistent Classes	3.2.1	Pt.	I	A	1	O	M
# Arguments per Operation	3.2.1	Pt.	I	A	1	O	M
# Static Operations	3.2.1	Pt.	I	A	1	O	M
# Virtual Operations	3.2.1	Pt.	I	A	1	O	M
# Pure Virtual Operations	3.2.1	Pt.	I	A	1	O	M

Table 11: Metrics Classification Part 1

Metric	Se.	T.	I/E	Sc.	E.	S/O	M/P
# Const Operations	3.2.1	Pt.	I	A	1	O	M
# Public Operations	3.2.1	Pt.	I	A	1	O	M
# Private Operations	3.2.1	Pt.	I	A	1	O	M
# Protected Operations	3.2.1	Pt.	I	A	1	O	M
# Implementation Operations	3.2.1	Pt.	I	A	1	O	M
# Pre-Conditions per Operation	3.2.1	Pt.	I	A	1	O	M
# Post-Conditions per Operation	3.2.1	Pt.	I	A	1	O	M
# Exceptions per Operation	3.2.1	Pt.	I	A	1	O	M
Storage per Operation	3.2.1	Pt.	I	R	1	O	M
Time per Operation	3.2.1	Pt.	I	R	5	O	M
Total Operation Storage	3.2.1	Pt.	I	A	1	O	M
# Object Diagrams	3.2.2	Pt.	I	A	1	O	M
# Methods Called	3.2.2	Pt.	I	A	1	O	M
# Links	3.2.2	Pt.	I	A	1	O	M
# Global Links	3.2.2	Pt.	I	A	1	O	M
# Parameter Links	3.2.2	Pt.	I	A	1	O	M
# Field Links	3.2.2	Pt.	I	A	1	O	M
# Local Links	3.2.2	Pt.	I	A	1	O	M
# Objects	3.2.2	Pt.	I	A	1	O	M
# Global Object Diagrams	3.2.2	Pt.	I	A	1	O	M
# Category Object Diagrams	3.2.2	Pt.	I	A	1	O	M
# Class Object Diagrams	3.2.2	Pt.	I	A	1	O	M
# Operation Object Diagrams	3.2.2	Pt.	I	A	1	O	M
# Message Passes	3.2.3	Pt.	I	A	1	O	M
# Objects	3.2.3	Pt.	I	A	1	O	M
# Modules	3.2.4	Pt.	I	A	1	O	M
# Dependencies	3.2.4	Pt.	I	A	1	O	M
Reuse Rate	3.3.1	Pt.	I	R	1	O	M
Class Internal Reuse Rate	3.3.1	Pt.	I	R	2	O	M
Internal Reuse Rate	3.3.1	Pt.	I	R	2	O	M

Table 12: Metrics Classification Part 2

Metric	Se.	T.	I/E	Sc.	E.	S/O	M/P
Project Productivity	3.3.2	Ps.	I	R	1/2	O	M
Overall Productivity	3.3.2	Pt.	I	R	1	O	M
Class Productivity	3.3.2	Ps.	I	R	1	O	M
# Errors	3.3.3	Ps.	I	A	1	O	M
# Corrections	3.3.3	Ps.	I	A	1	O	M
Code Maintainability	3.3.4	Pt.	I	R	1	O	M
Overall Maintainability	3.3.4	Ps.	I	R	1	O	M
Error Maintainability	3.3.4	Ps.	I	R	1	O	M
Correction Maintainability	3.3.4	Ps.	I	R	1	O	M
Extensibility	3.3.5	Ps.	E	R	3	O	M
Adaptability	3.3.6	Ps.	E	R	3	O	M
Speed Efficiency	3.3.7	Ps.	E	R	3	O	M
Size Efficiency	3.3.7	Pt.	E	R	3	O	M
Memory Efficiency	3.3.7	Re.	E	R	3	O	M
Space Efficiency	3.3.7	Re.	E	R	3	O	M
Robustness	3.3.8	Ps.	E	R	3	O	M
# comment lines per class	3.3.9	Pt.	I	A	1	O	M
# words text documentation per class	3.3.9	Pt.	I	A	1	O	M
# instance variables per class	3.3.9	Pt.	I	A	1	O	M
# class variables per class	3.3.9	Pt.	I	A	1	O	M
Portability	3.3.10	Pt.	I	R	2	O	M
C&K's LCOM	3.3.11	Pt.	I	O	2	O	M
H&M's LCOM	3.3.11	Pt.	I	A	2	O	M
H	3.3.11	Pt.	I	O	2	O	M
CBO	3.3.12	Pt.	I	A	2	O	M
MPC	3.3.12	Pt.	I	A	2	O	M
DAC	3.3.12	Pt.	I	A	2	O	M
CBO1	3.3.12	Pt.	I	A	2	O	M
CBO2	3.3.12	Pt.	I	A	2	O	M
CBO3	3.3.12	Pt.	I	A	2	O	M
CBO4	3.3.12	Pt.	I	A	2	O	M
CBO5	3.3.12	Pt.	I	A	2	O	M
CBO6	3.3.12	Pt.	I	A	2	O	M

Table 13: Metrics Classification Part 3

Metric	Se.	T.	I/E	Sc.	E.	S/O	M/P
CBO7	3.3.12	Pt.	I	A	2	O	M
CBO8	3.3.12	Pt.	I	A	2	O	M
CBO9	3.3.12	Pt.	I	A	2	O	M
CBO10	3.3.12	Pt.	I	A	2	O	M
C_a	3.3.12	Pt.	I	A	2	O	M
C_e	3.3.12	Pt.	I	A	2	O	M
McCabe	3.3.13	Pt.	I	I	1 or 2	O	M
WMC	3.3.13	Pt.	I	O	2	O	M
# Classes	3.3.14	Pt.	I	A	1	O	M
NCSS	3.3.14	Pt.	I	A	2	O	M
SLOC	3.3.14	Pt.	I	A	2	O	M
# Attributes	3.3.14	Pt.	I	A	1 or 2	O	M
DIT	3.3.15	Pt.	I	R	2	O	M
NOC	3.3.15	Pt.	I	A	2	O	M
Developer Confidence	3.3.16	Re.	E	O	3	S	M
RFC	3.3.17	Pt.	I	A	2	O	M
A	3.3.17	Pt.	I	R	2	O	M
NOM	3.3.17	Pt.	I	A	1 or 2	O	M
Function Points	3.4	Pt.	E	X	4	S	P

Table 14: Metrics Classification Part 4

- O - Ordinal Scale. F&P list the following characteristics of the ordinal scale:-
  - The empirical relation system consists of classes that are ordered with respect to the attribute.
  - Any mapping that preserves the ordering (that is, any monotonic function) is acceptable.
  - The numbers represent ranking only, so addition, subtraction, and other arithmetic operations have no meaning.
- I - Interval Scale. F&P characterize the interval scale in the following way:-
  - An interval scale preserves order, as with an ordinal scale.
  - An interval scale preserves differences but not ratios. That is, we know the difference between any of the two ordered classes in the range of the mapping, but computing the ratio of two classes in the range does not make sense.
  - Addition and subtraction are acceptable on the interval scale, but not multiplication and division.
- R - Ratio Scale. F&P give the following characteristics for the ratio scale:-
  - It is a measurement mapping that preserves ordering, the size of intervals between entities, and ratios between entities.
  - There is a zero element, representing total lack of the attribute.
  - The measurement mapping must start at zero and increase at equal intervals, known as units.
  - All arithmetic can be meaningfully employed to the classes in the range of the mapping.
- A - Absolute Scale. F&P state that the absolute scale has the following characteristics:-
  - The measurement for an absolute scale is made simply by counting the number of elements in the entity set.
  - The attribute always takes the form "number of occurences of x in the entity".
  - There is only one possible measurement mapping, namely the actual count.
  - All arithmetic analysis of the resulting count, is meaningful.
- X - no valid scale.

E. - Ease This gives the ease of measurement of the metric. The valid values are as follows:-

- 1 - These metrics can be measured easily from existing products and forms.
- 2 - These metrics can be measured easily by a parse-tree traversing tool.
- 3 - These metrics can be measured easily through experiment.
- 4 - These metrics can be easily gathered by a function point analysis tool.
- 5 - These methods require the ability to measure timings for individual method calls.

S/O - Subjective/Objective Objective metrics will be the same no matter how many times measured, but subjective metrics depend on something intangible such as the opinion of a developer.

M/P - Measurement/Predictive Measurement metrics are a measure of an attribute whereas predictive metrics are used in making a prediction about a product, process, or resource. Typical predictive metrics are function points, used to predict development time.

4 Conclusions

We have surveyed the literature and selected a suite of metrics and information required to gather process metrics. This gives a firm basis on which to select metrics for the MOOD project as well as any other project with similar aims.

References

[Basili 94] Basili, V.R., Caldiera, G. and Rombach, H.D.
Measurement.
Encyclopedia of Software Engineering, volume 1, p646–661, John J. Marciniak editor, John Wiley & Sons, 1994.

[Bieman 92] Bieman, J.
Deriving Measures of Software Reuse in Object-Oriented Systems.
Formal Aspects of Measurement (T. Denvir, R. Herman and R. Whitty ed.s) Springer-Verlag, London 1992, p63–83.

[Bunge 77] Bunge, M.
Treatise on Basic Philosophy: Ontology I: The Furniture of the World.
Boston: *Riedel* 1977.

[Bunge 79] Bunge, M.
Treatise on Basic Philosophy: Ontology II: The World of Systems.
Boston: *Riedel* 1979.

[Chidamber 94] Chidamber, S. and Kemerer, C.
A Metrics Suite for Object Oriented Design.
IEEE Transactions on Software Engineering vol. 20 no. 6 June 1994 p476–493.

[Churcher 95] Churcher, N. and Shepperd, M.
Comments on "A Metrics Suite for Object Oriented Design".
IEEE Transactions on Software Engineering vol. 21 no. 3 March 1995 p263–265.

[Dumke 96] Dumke, R.
Software Metrics A Subdivided Bibliography.
Research Report IRB-007/92 *Otto von Guericke University of Magdeburg*, Postfach 4120, D-39016 Magdeburg, Germany, February 1996.

[Fenton 91] Fenton, N.
Software Metrics: A Rigorous Approach.
Chapman & Hall, London, 1991.

[Fenton 96] Fenton, N. E., and Pfleeger, S. L.
Software Metrics A Rigorous and Practical Approach
International Thomson Computer Press, London, 1996.

[Fung 96] Fung, M., Henderson–Sellers, B. and Yap, L.–M.
A Comparative Evaluation of OO Methodologies from a Business Rules and Quality Perspective.
to appear in *Australian Computer Journal* 1996.

[Goldberg 95] Goldberg, A. and Rubin, K.S.
Succeeding with Objects: Decision Frameworks for Project Management
Addison-Wesley, Reading, Massachusets, USA, 1995.

[Henderson 96] Henderson–Sellers, B.
Object-Oriented Metrics Measures of Complexity.
Prentice Hall, New Jersey, 1996.

[Hitz 96] Hitz, M. and Montazeri, B.
Chidamber & Kemerer's Metrics Suite: A Measurement Theory Perspective.
to appear in *IEEE Transactions on Software Engineering*, vol. 22 no. 4, April 1996.

[Li 93] Li, W. and Henry, S.
Object-Oriented Metrics that Predict Maintainability.
Journal of Systems and Software, vol. 23 no. 2, 1993, p111-122.

[Lim 94] Lim, W.
Effects of Reuse on Quality, Productivity, and Economics.
IEEE Software September 1994 p23–30.

[Lorenz 94] Lorenz, M. and Kidd, J.
Object-Oriented Software Metrics.
Prentice-Hall, New Jersey, 1994.

[Martin 95] Martin, R. C.
Designing OO C++ Applications using the Booch Method
Prentice Hall, New Jersey, 1995.

[Meyer 88] Meyer, B.
Object-oriented Software Construction.
Prentice Hall, Englewood Cliffs, NJ, 1988.

[Sharble 93] Sharble, R. and Cohen, S.
The Object-oriented Brewery: A Comparison of Two Object-oriented Development Methods.
SIGSOFT Software Engineering Notes 18 (2) 1993, p60–73.

[Weyuker 88] Weyuker, E.
Evaluating Software Complexity Measures.
IEEE Transactions on Software Engineering vol. 14 no. 9 September 1988 p1357–1365.

[Whitty 96] Whitty, R.
Object-Oriented Metrics: an Annotated Bibliography.
available on the world-wide web at "http://www.sbu.ac.uk/~csse/publications/OOMetrics.html".

A Process Metric Information

A.1 Information Gathered in DMS

The DMS is Parallax's Delivery Management System. The only process metrics specified to be gathered by the DMS are information on work product defects gathered at reviews. The information the DMS specifies to gather is as follows:

- Section (Development Phase)
- Description

- Person Assigned
- Date Closed

I think that MOOD only needs information on errors in the working of the final product.

A.2 Extra Error Process Metrics

I propose that the following extra information should be gathered on errors (defects in working of final system) by use of forms extra to the DMS:

- Size added/modified/deleted (code in non-blank non-comment lines of code, design in number of methods)
- Number of corrections (separate changes in the code/design)
- Time in person-hours spent fixing the error

This information will enable MOOD to measure maintainability metrics.

A.3 Weekly Process Metrics

I reckon that we should record the following weekly information for gathering process metrics:

- Time spent in each development phase. This will enable a complete breakdown to be made of overall time spent in each development phase.
- Name of any classes being reused. This will enable MOOD to keep track of all classes which are reused, for use in formulating reuse metrics.

B Evaluation of Literature for MOOD

In this section the five papers reviewed in this report are evaluated for their usefulness to the MOOD project.

B.1 [Bieman 92]

In our case unlike Bieman we should know which are library classes (originally designed for another system) and new classes. Bieman considered all non-trivial inheritance as partial, or leveraged reuse, even if the inheritance is extending rather than modifying the ancestor class. I think it makes more sense for our purposes to consider extending and not modifying a class through inheritance, as verbatim reuse. Bieman measured non-inheritance reuse (Verbatim server reuse) as the number of instance creations of objects of the class and the number of instance references of objects of the class. An alternative (which I think is better) way of measuring non-inheritance reuse is through measuring a method invoked in a class, as a usage. Otherwise Bieman makes some useful definitions, which we will be able to use in the course of the MOOD project. This paper also helped to crystallize the preliminary reuse and reusability metrics which we have selected, by providing the definition of so many reuse metrics.

B.2 [Lim 94]

Considering this paper from the point of view of the MOOD project objectives:

- This paper shows some instances of projects in which reuse aided productivity, and a component of software quality; correctness.
- This paper shows a way of measuring productivity, and correctness.

B.3 [Chidamber 94]

Some points from this paper which relate to the MOOD project objectives are as follows:

- The six metrics outlined all relate to aspects of software quality, according to the authors.
- The paper is an established object-oriented metrics paper published in the "IEEE Transactions on Software Engineering", as are two commentaries/criticisms [Churcher 95] [Hitz 96].
- The six metrics have been used by other people [Sharble 93] [Li 93].
- In summary, these metrics relate to quality and are used widely. They are therefore worth considering for MOOD.

B.4 [Churcher 95]

It is important that all our metrics are precisely and explicitly defined.

B.5 [Hitz 96]

From the point of view of the MOOD objectives, from [Hitz 96] it makes intuitive sense to change the CBO and LCOM metrics, which we will measure, as recommended by H&M. Also the method shown for designing useful measures has proved useful in designing metrics for MOOD and no doubt will continue to prove useful.