ASSESSING THE MAINTAINABILITY OF C++ SOURCE CODE

By

MARIUS SUNDBAKKEN

A thesis submitted in partial fulfillment of
the requirements for the degree of

Master of Science in Computer Science

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

DECEMBER 2001

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of MARIUS SUNDBAKKEN find it satisfactory and recommend that it be accepted.

_____

Chair

_____

_____

ASSESSING THE MAINTAINABILITY OF C++ SOURCE CODE

Abstract

by Marius Sundbakken, M.S.
Washington State University
December 2001

Chair: David Bakken

Maintenance refers to the modifications made to software systems after their first release. It is not possible to develop a significant software system that does not need maintenance because change, and hence maintenance, is an inherent characteristic of software systems. It has been estimated that it costs 80% more to maintain software than to develop it. Clearly, maintenance is the major expense in the lifetime of a software product. Predicting the maintenance effort is therefore vital for cost-effective design and development. Automated techniques that can quantify the maintainability of object-oriented designs would be very useful. Models based on metrics for object-oriented source code are necessary to assess software quality and predict engineering effort.

This thesis will look at C++, one of the most widely used object-oriented programming languages in academia and industry today. Metrics based models that assess the maintainability of the source code using object-oriented software metrics are developed.

Table of Contents

List of Tables

List of Figures

Chapter 1

1. Introduction

1.1. Maintenance and Maintainability

Maintenance refers to the modifications made to software systems after their first release. It is not possible to develop a significant software system that does not need maintenance because change, and hence maintenance, is an inherent characteristic of software systems [Hsia, 95]. Maintenance is defined, by the IEEE, as “the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.”[IEEE 610.12] Maintainability is defined as “the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.” [IEEE 610.12]

In 1975, Brooks stated that it typically costs 40% more to maintain software than to develop it [Brooks, 95]. Three years later, [Lientz, 78] estimate this to be 50% and in 1983 [Martin, 83] estimate it to be 80% and said that $20 billion is spent yearly worldwide on maintenance. Ten years later, [Edelstein, 93] reported this to be $70 billion. Today, the numbers are most likely even higher, although it should be noted that disagreement of what maintenance encompasses leads to estimates ranging from 40% to 90% [Mattson, 00]. Regardless of the definition, software maintenance is paramount in the software life cycle. When we study software maintenance, reducing cost is the

primary goal, although [Zhuo, 93] points out that in an industrial environment, maintainability measures may be applied in three ways:

1. As a managerial assessment to quantify the cost of maintaining existing software systems.
2. As a quality assessment and control mechanism to drive software development efforts. [Kiran, 97] sees maintainability as a quality property of software.
3. As a mechanism to enforce maintainability standards prior to acceptance and/or delivery.

Since maintenance in one way or another implies changing source code, the cost is directly related to the quality of the source code and the program design.

1.2. Analyzing C Source Code

Analyzing C source code is simpler than analyzing C++ code since C is a subset of C++. C is also procedural and source code written in C has a design that is simpler to parse and evaluate. From a software metric point of view, procedural design has little more organization than one big set of functions. C++, on the other hand, is more organized with hierarchies and encapsulation, polymorphism and genericity.

1.3. Problems Addressed

It is clear that maintenance is the major expense in the lifetime of a software product. Predicting the maintenance effort is therefore vital for cost-effective design and development. Techniques that have the ability to reveal high maintenance designs would

be very useful. Dedicated metrics for object-oriented source code are necessary, and even specialized metrics for individual languages are welcomed.

The C++ programming language is one of the most used, if not *the* most used object-oriented programming languages in use in academia and industry. Because of this, focusing on the C++ programming language is certainly beneficial.

To quantify maintainability, software metrics are employed. In a study by [Glasberg, 00], the post-release costs (maintenance) were reduced by 42% by using object-oriented metrics on the design to identify high-maintenance classes and redesigning them (if possible). [Kiran, 97] shows that object-oriented C++ implementations were easier to maintain than similar non-object-oriented C implementations.

[Li, 93] shows that when using the [Chidamber, 91] metrics:

1. There is a strong relationship between metrics and maintenance effort in object-oriented systems.
2. Maintenance effort can be predicted from combinations of metrics collected from source code.
3. The prediction is successfully cross-validated.

[Oman, 92] provides a taxonomy of software maintainability and three broad categories: (1) Management practices, (2) Hardware and software environments and (3) The target software system. This thesis will focus on software maintenance on target software

systems. By measuring the source code with a set of metrics, the results can be combined into a single value (hybrid metric) representing the maintainability of the software being evaluated.

Quantitative evaluation of object-oriented C++ code in terms of maintainability will be obtained using object-oriented software metrics. A model will be established from the relationship between the numbers provided by static metric data of the source code and maintainability surveys completed by research assistants [Oman, 92]. The construction of the assessment model will follow the study in [Zhuo, 93].

It is the author's thesis that:

It is possible to construct metric models, a Maintainability Index, that accurately predicts or indicates the maintainability of C++ source code using object-oriented software metrics.

The contributions of this thesis are:

- The maintainability of C++ source code can be mechanically assessed from non-preprocessed source files.
- The suitability of various metrics as maintainability predictors are identified.
- Candidate Models for calculating a Maintainability Index of Object oriented C++ have been developed.

1.4. Outline

In Chapter 2 related works will be examined and what has been done in maintainability assessment during the last five years. First, the concepts of object-oriented design will be discussed in order to provide a foundation on which to base the various metrics. The problems with gathering metrics will also be discussed.

In Chapter 3, the changes performed on *m2*, now *m3*, will be discussed in detail, as well as the metrics visualization tool, *mvt*, and the front-end that combines *m2* and *mvt*, namely *mmf*.

The central technique used in *m2/m3* will be explained in detail. All of the *m2* metrics collected will be defined followed by definitions of the new metrics added in *m3*. These new metrics will also be discussed.

In Chapter 4, an industry source code set will be studied for maintainability. Data will be collected from the source code using the tool developed. This data will then be correlated to the subjective evaluations performed on the source code.

Chapter 5 will summarize the thesis and suggest further work.

Chapter 2

2. Related Work

This chapter will look at related work including an overview of the principles of object-oriented design, and at the various object-oriented metrics that try to measure these design principles. A look at the various tools available for gathering metrics will follow. There will be a brief discussion of various features in the C++ language and how they compare to other popular object-oriented languages. A discussion of the assessment of object-oriented source code will follow and, finally, a discussion of the various object-oriented principles and how they fit with object-oriented metrics.

2.1. Object-Oriented Concepts

[Booch, 94] considers these to be the four major elements of the object model:

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

These are defined as follows [Booch, 94]:

Abstraction

> *An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.*

Encapsulation

Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics.

Modularity

Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

Hierarchy

Hierarchy is a ranking or ordering of abstractions.

Studying the object-oriented literature reveals variations in the precise concepts that define object-oriented design and programming. Different names and granularity are used, but the above listing is representative.

[Fowler, 00] has a helpful perspective on objects:

1. At the *conceptual level*, an object is a set of responsibilities.
2. At the *specification level*, an object is a set of methods than can be invoked by other objects or by itself.
3. At the *implementation level*, an object is code and data.

A common mistake is to think of object-oriented design only at the implementation level [Shalloway, 02]. For example, abstract classes are often described as classes that cannot be instantiated. This is correct, but focuses only at the implementation level. It is more helpful to think of abstract classes from a conceptual level where they are simply placeholders for other classes. They open up the possibility to treat a set of related classes as one concept.

2.2. Principles of Object-Oriented Design

In order to better understand the various object-oriented metrics, there will be an overview of what are considered to be the principles of object-oriented design. What exactly constitute 'good' and 'bad' design is not clearly defined in the literature. In generic terms, [Booch, 96] states that "all well structured object-oriented architectures have clearly defined layers, with each layer providing some coherent set of services through a well-defined and controlled interface." (pp. 54). [Martin, 96c] classifies bad design as:

1. **Rigidity** – It is hard to change because every change affects too many other parts of the system.
2. **Fragility** – When you make a change, unexpected parts of the system break.
3. **Immobility** – It is hard to reuse in another application because it cannot be disentangled from the current application.

The cause of rigid, fragile and immobile designs is the interdependence of the modules in that design. In [Martin, 94] a closer look at OO design is presented along with what characteristics make a design good or not. He recognizes that there are patterns of

interdependence and that some of these are desirable and some are not. The *desirable dependencies* are those that are *stable*. The question becomes how to achieve stability. Independence is one way. For example, string classes tend to be independent since they do not typically require any other classes. String classes also tend to be heavily used. This prevents us from breaking the interface of the string class because any change has a significant impact. Such classes have a strong impact on the software, and Martin calls them for "responsible" classes. The most stable classes, then, are those that are both independent and responsible, such as string classes.

This dependence often goes under the name 'coupling'. Cohesion is self-coupling, i.e. it refers to how closely the methods of a class are related.

There are several principles that work against undesirable interdependence. A closer look at these principles follows.

2.2.1. The Open Closed Principle (OCP)

This principle can be phrased as [Martin, 96a]:

> *A module should be open for extension but closed for modification.*

This principle originates from [Meyer, 88], and [Martin, 96a] calls it "the heart of object-oriented design." It means simply that modules should be extendable without requiring modification. The key to achieve this is abstraction. In practice, this usually means extension through inheritance (and, therefore, usually using polymorphism). Often, this is

from library classes where the source code is not available, and even if it is, it is still undesirable to modify it.

[Martin, 96a] explains the principle as this:

1. ‘Open for Extension’ – This means that the behavior of the module can be extended.
2. ‘Closed for Modification’ – No one is allowed to make changes to the source code of the module.

2.2.2. The Liskov Substitution Principle (LSP)

This principle can be phrased as [Martin, 96b]:

> *Subclasses should be substitutable for their base classes.*

Barbara Liskov coined the principle in her work regarding data abstraction and type theory [Martin, 00]. Better known, perhaps, is Meyer’s similar concept of Design by Contract [Meyer, 88] in which methods declare pre- and post-conditions. Unlike Eiffel, however, C++ does not have a specific language mechanism for this, although assert statements could be used to partly emulate it.

The concept is depicted in Figure 1. Derived classes should be substitutable for their base classes. The *User* class should be able to use the *Base* derived classes and function properly.

The principle can be stated in terms of contracts. A derived class is substitutable for its base class if:

1. Its preconditions are no stronger than the base class method.
2. Its post conditions are no weaker than the base class method.

In other words, derived methods should expect no more and provide no less than the method they override.

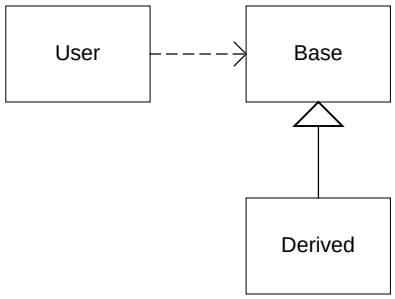

Figure 1 Liskov Substitution Principle

[Martin, 96b] discusses the principle further.

2.2.3. The Dependency Inversion Principle (DIP)

This principle can be phrased as:

> *Depend upon abstractions. Do not depend upon concretions.*

The above is stated in [Martin, 00], but a more detailed statement can be found in [Martin, 96c] where it stated that there are really two parts to this principle:

1. High-level modules should not depend upon low-level modules. Both should depend upon abstractions.
2. Abstractions should not depend upon details. Details should depend upon abstractions.

The open-closed principle can be seen as the *goal* of object-oriented architecture. The primary *mechanism* to achieve this is the dependency inversion principle. The strategy is to depend upon interfaces or abstract functions and classes, rather than upon concrete functions and classes. This is the enabling force behind COM, CORBA and EJB.

Figure 2 and Figure 3 show how the dependency structure differs in procedural design and in object-oriented design. The procedural design is intrinsically weak because the high-level modules depend directly on the low-level implementation.

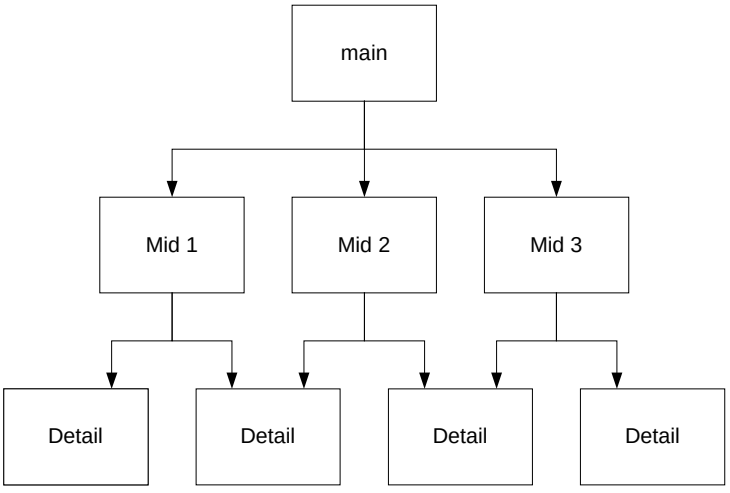

Figure 2 Dependency structure of a procedural architecture

As can be seen in Figure 3, the dependency structure in object-oriented designs is highly preferable since the high-level modules are not dependent on the lower levels. The majority of the dependencies point to abstractions. The detail implementations themselves depend on these abstractions, thus they are *inverted*.

The implication of the principle, then, is that every dependency should target an interface or an abstract class. No dependency should target a concrete class [Martin, 00]. The motivation is that anything concrete is volatile. This is frequently the case, but there are exceptions. Most common is the use of string classes such as the STL string class or string functions such as those in the standard C library.

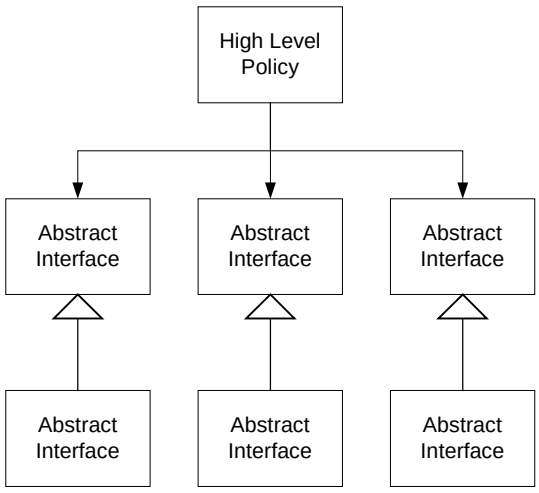

Figure 3 Dependency structure of an object-oriented architecture

[Martin, 96c] discusses this principle in detail.

2.2.4. The Interface Segregation Principle (ISP)

This principle can be phrased as [Martin, 96d]:

Many client specific interfaces are better than one general-purpose interface.

As mentioned in 0, DIP is the primary mechanism that allows for OCP. Another mechanism is ISP, although this is by far the least common. [Martin, 00] says that if you have a class that has several clients, rather than loading the class with all the methods the clientage need, create specific interfaces for each client and multiple inherit them into the class.

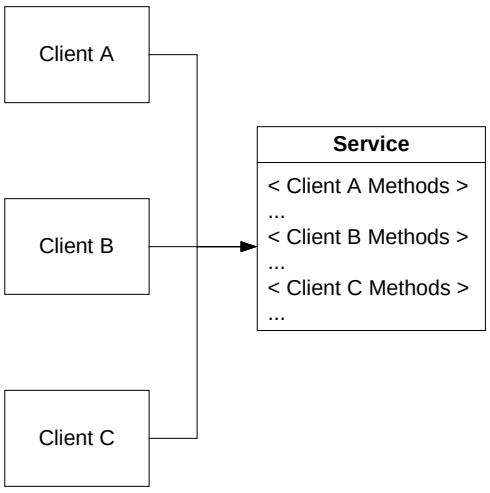

Figure 4 Fat service with integrated interfaces

With a design such as that shown in Figure 4, a change made to one of the methods that 'Client A' calls may affect 'Client B' and 'Client C'. This propagation is undesirable as it increases maintenance effort. A better solution is shown in Figure 5. Each client has its own service interface. The service interfaces are multiple inherited to the Service class

and implemented there. Any change in the interface ‘Service A’ will not affect ‘Client B’ or ‘Client C’.

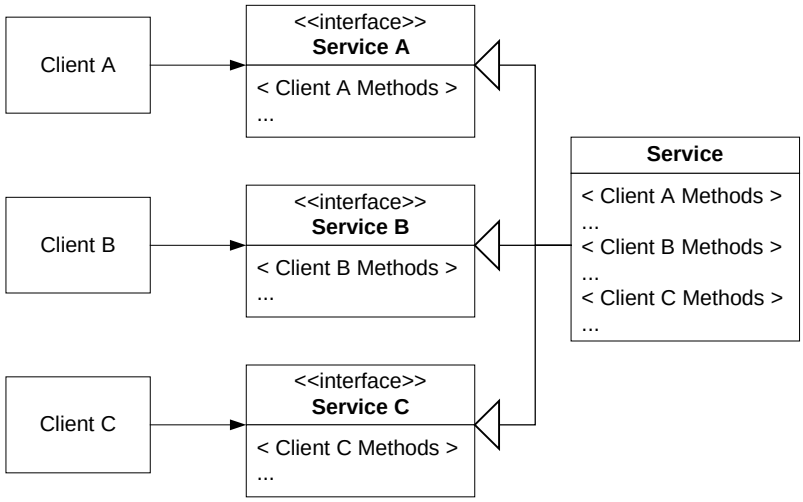

Figure 5 Segregated interfaces

The interface segregation principle is discussed further in [Martin, 96d].

It is worth noting that this principle uses the much criticized multiple inheritance feature of C++. Multiple inheritance will be discussed further in 2.6.1, but this principle shows that multiple inheritance does have its uses.

2.3. Object-Oriented Metrics

Much of the object-oriented metrics research is based on the almost classic [Chidamber, 91]. The suite of metrics proposed there are, however, based on theoretical work and measurement theory rather than practical experience. To balance this, the work by

[Lorenz, 93] is based on real-life trial and error. Lastly, the comprehensive study in [Sellers, 94] has examined many of the proposed metrics.

The following table shows the combined recommendations by:

A. Lorenz [Lorenz, 93]

B. Chidamber & Kemerer [Chidamber, 91] (The MOOSE[1] suite)

C. Henderson-Sellers & Edwards [Sellers, 94]

D. Abreu [Abreu, 95] (The MOOD[2] suite)

Metric	**Ref.**
Lack of Cohesion in Methods	B
Fan-out (coupling)	C
Average and standard deviation of the number of methods per class	C
Average and standard deviation of the cyclomatic complexity of the methods	C
Mean and maximum depths of inheritance hierarchies	C
Reuse ratio	C
Specialization ratio	C
Average and standard deviation of method size	A, C
Average number of methods per class	A
Average number of instance variables per class	A
Depth of inheritance tree	A, B
Number of subsystem/subsystem relationship	A
Number of class/class relationships within each subsystem	A
Instance variable usage	A
Average number of comment lines	A
Weighted methods per class	B
Number of children	B
Coupling between objects	B
Response for class	B
Method Inheritance Factor	D
Method Hiding Factor	D

Table 1 Recommended metrics

[1] Metrics for Object-Oriented Software Engineering

[2] Metrics for Object-Oriented Design

Not all of these metrics were implemented in this study (for the *m3* tool). More recent studies, in particular [Basili, 95], show that LCOM (Lack of Cohesion of Methods) is insignificant to detect defective classes. Although we must be careful not to view defect prediction and maintainability as the same, the results regarding LCOM are still relevant. The same study refers to problems with the cyclomatic complexity metric in object-oriented design. Other metrics were dropped because of the difficulty of implementing them in the design of the metrics tool that was used as a basis, *m2*.

2.4. Difficulty Gathering Metrics

The *m2* tool is based on the conceptual source code module technique in order to better analyze non-preprocessed C++ source code. As shown by [Penner, 99], measuring C++ source code is non-trivial. It can be accomplished in two fundamentally different ways: on preprocessed and non-preprocessed code. Preprocessed code, however, is removed from what the programmer sees. Because of this, it is preferable to measure non-preprocessed source code, but this is not an easy process. [Penner, 99] presents a technique to simplify this, called the conceptual source code model. This is implemented in the *m2* tool.

A closer look at this technique will be discussed in 3.1.

2.4.1. Tools

There are many measurement tools for C++. Many, however, are not useful on non-preprocessed C++ source code files [Penner, 99]. The tools Penner looked is given in Table 2.

Static Analysis Tool	Company	Embedded	Stand-alone	Notes
Cantata	IPL		X	Preprocesses
CMT++	Testwell		X	Handles Conditional Compilation
Codecheck	Abraxas	X		
COOL: Teamwork	Sterling	X		No Reply
Hindsight	Integrisoft	X		
Krakatau	Power Software		X	Preprocesses, Uses Fuzzy-Parsing
Logiscope	Verilog		X	Preprocesses
Metrics ONE	Number Six Software	X		No Reply
Metrics4C	+1 Software Engineering		X	Preprocesses (C code only)
Object Detail	Object Software	X		No Reply
PC-Metric	SET Labs, Inc.		X	Skips Conditional Compilation
QA C++	PR: QA		X	Preprocesses, Does Deep Flow analysis
QualGen	Scientific Toolworks		X	Preprocesses
RSM	M Squared Technologies		X	Skips Conditional Compilation
Software Metrics	McCabe & Associates	X		No Reply
Testworks METRIC	Software Research	X		
Ccount	*Freeware*		X	Skips Conditional Compilation (C code only)
MKS Code Integrity	MKS	X		No Reply
MAS	Software Engineering Test Lab		X	Ignores Conditional Compilation (C code only)
CIA	AT&T	X		Defunct

Table 2 Source code static analyzers

Of the 20 surveyed measurement tools, 11 can be classified as stand-alone C/C++ measurement tools. Of these 11 stand-alone measurement tools, nine are designed to skip or preprocess to eliminate conditional compilation from the source code file.

Stand-alone C/C++ measurement tools consider all of the source code text. Such tools are necessary to gather metrics on the source code file, but conditional compilation can easily modify the syntactical structure of the source code file causing parser-based techniques to fail. Many stand-alone C/C++ measurement tools ignore conditional compilation paths or attempt to resolve the different paths with user input since it complicates the parsing process.

A few more tools have appeared since [Penner, 99]:

TAC++	[1]
Understand for C++	[2]
CCCC	[3]
MAISA	[4]

2.5. Object-Oriented Programming Languages

There are several object-oriented languages. This thesis focuses on C++, but there are also Java, Ada, Objective-C, Eiffel, and Smalltalk to name the most well known. In order to better understand C++, it is useful to examine other languages. Understanding what is considered to be “good” or “bad” design in those languages can then be related to C++.

Some of the most common criticisms of C++ are [Jia, 00]:

1. C++ is not a pure object-oriented language. It is a hybrid in which stand-alone functions and global variables are allowed.
2. Not everything is an object. Primitive types like `int`, `char` are not classes as they are in Smalltalk.

3. Multiple inheritance as designed in C++ is considered undesirable due to the possibility of repeated inheritance[3] and the conflicts that may arise as a result.
4. No garbage collection.
5. Friends.

Although garbage collection is useful, it is not a strictly object-oriented feature and it will not be discussed here.

2.6. Assessing Maintainability of Object-Oriented Code

There is not much literature available on object-oriented metrics. Compared to the amount of literature on object-oriented analysis, design and programming, the object-oriented metrics literature is almost non-existent. One of the most noticeable books is [Sellers, 96] on which much other work is based. The last five years, it appears that work has been focused on *using* object-oriented metrics rather than inventing new ones.

In 1995, the concept of design patterns was widely noticed by software designers and developers after the release of the classic *Design Patterns* by Gamma et al [Gamma, 95]. Design patterns have since gained much popularity and attention and are now buzzwords in the software industry. The fundamental idea of design patterns is to reuse something that has been proven to work well in many different cases. In other words, design patterns have proven to be highly maintainable, most likely reducing maintenance costs. Software metrics are used in order to locate high-maintenance areas in software systems, so design

[3] The inheritance hierarchy in such cases is sometimes called the "deadly diamond of death."

patterns and software metrics have the same general aim. Since software metrics can be used to quantify maintenance, it could be useful to see how design patterns affect various software metrics. [Masuda, 99] and [Huston, 01] have studied the relationship between design patterns and software metrics.

[Masuda, 99] investigated the [Chidamber, 91] metrics in particular and how design patterns affect various metrics. They reported that specific design patterns tend to affect particular metric values so that the metrics indicate design problems. At first, this may seem of concern to people using software metrics, but a closer look at the various patterns show that the patterns use some specific object-oriented techniques to a high extent, such as inheritance. Design patterns and software metrics are *mainly* congruent [Huston, 01]. Differences do not invalidate either the metrics or the patterns, but show that the metric numbers should not be uncritically accepted. In some situations, it may certainly be preferable to design software in a way that makes particular metrics raise false warning flags. The patterns literature also typically specifies the drawbacks by choosing a particular pattern. It is up to the designer to evaluate the impact various design patterns will have on the software structure. On the other hand, the warning flag raised by a certain metric may certainly be valid and the particular pattern applied may not be appropriate.

It is important to realize that there is no silver bullet [Brooks, 95]; neither metrics nor patterns guarantee anything. Unfortunately, patterns have been a victim of this and [Vlissides, 98] list that one of the top *misconceptions* of patterns is that they “guarantee

reusable software, higher productivity […].” Obviously, this is not true. Neither metrics nor patterns should be followed blindly.

This may seem confusing, but despite some of these confusions, object-oriented software has been shown to be more maintainable than procedural software [Kiran, 97].

In order to understand object-oriented software metrics better and perceive how applicable and valuable they are, the object-oriented techniques must be fully understood. This thesis will look at:

- Inheritance
- Encapsulation
- Polymorphism

By far the most researched concept in terms of maintainability in the object-oriented paradigm, is that of inheritance. The other techniques have been studied far less.

2.6.1. Inheritance

One central concept in the object-oriented paradigm is the use of inheritance. In *m3*, inheritance is measured with DIT, NOC and NOS. Although inheritance is far less used in C++ than expected [Bieman, 95] [Cartwright, 98], [Daly, 95] shows that, on average, object-oriented software using inheritance is approximately 20% quicker to maintain than that with no inheritance. It is clear that inheritance plays a central role in the maintainability of object-oriented software, and quantifying the quality of the inheritance tree is useful to assess maintainability [Hsia, 95]. The study by Hsia shows that a design

with a higher broadness factor was more maintainable than one with a smaller broadness factor (tall and slim trees). Similar results were shown by [Daly, 95].

[Basili, 95] reports that DIT is a very significant metric for defect detection. The larger the DIT value, the larger the probability of defect detection. Although defect detection is not the same as maintainability, it is nevertheless important. The same study also reports that NOC is very significant in defect detection, except for UI classes. The larger the NOC value the lower the probability of defect detection.

[Li, 93] sees inheritance as a form of coupling:

> "Inheritance promotes reuse, but also creates the possibility of violating encapsulation and information hiding. This violation occurs because the properties in the super class are exposed to the subclass for less restrictive access. The use of inheritance that is not well designed may introduce extra complexity to a system. The extra complexity is due to the attributes, which are encapsulated in the super class but now are exposed to less restricted access by the subclass. The more a class inherits, the more non-private attributes the class may access."

This is certainly true, but a well-designed class has *no* public or protected attributes and is fully encapsulated.

So far, only single inheritance has been the focus, but a highly criticized feature of C++ is its multiple inheritance design.

Multiple inheritance as designed in C++ is considered to be undesirable because of the conflicts it may create (see 2.5). Compared to other languages, C++ handles multiple inheritance poorly. The BETA programming language does not allow it and [Madsen, 93] states that “BETA does not have multiple inheritance, due to the lack of a profound theoretical understanding, and also because the current proposals seem technically very complicated.” Ada 95 also avoids multiple inheritance. Sun claims multiple inheritance results in problems, so Java provides it in a restricted form by using interfaces, which Sun claims provide all the desirable features of multiple inheritance. Java interfaces provide semantic multiple inheritance, or types, but not multiple inheritance of code, which as [Wegner, 91] points out, is of more practical value [Joyner, 99]. Joyner confirms Sun’s claims and points out that what seems like simple generalization of inheriting from multiple classes turns out to be nontrivial. The multiple inheritance design in C++, however, is simple and, consequently, this is where the language suffers. For example, what should the policy be if you inherit items of the same name from two or more classes? It is worth pointing out that Java does not solve all the problems with its interface approach. Constants can be declared in interfaces and the same problem as given in the example above applies [Joyner, 99].

One of the ideas behind multiple inheritance is that it might help with handling the combinatorial explosion of design choices through a small number of cleverly chosen base classes. However, “any experienced class designer knows that such a naïve design

does not work." [Alexandrescu, 01]. The problem in C++ with assembling separate features by using multiple inheritance can be briefly categorized as follows:

1. **Mechanism**. The simple superposition in combining the base classes and the simple set of rules for accessing their members is unacceptable except for the simplest cases. Most of the time, there is a need to orchestrate the workings of the inherited classes carefully to obtain the desired behavior.
2. **Type information**. The base classes do not have enough type information to carry on their tasks.
3. **State manipulation**. Various behavioral aspects implemented with base classes must manipulate the same state. This means that they must use virtual inheritance to inherit a base class that holds the state. This complicates the design and makes it more rigid because the premise was that user classes inherit library classes, not vice versa.

Although combinatorial in nature, multiple inheritance cannot by itself address the multiplicity of design choices. One way to tackle this with C++ is by using templates. It is, however, beyond the scope of this thesis to discuss templates or any of the features mentioned above in any more detail.

Much has been written about the problems of multiple inheritance [Joyner, 99]. There are advocates for multiple. Eiffel, for example, handles multiple inheritance well. Multiple inheritance can be misused, but so can single inheritance. In fact, single inheritance can be seen as a special case of multiple inheritance. Even though there are problems with

multiple inheritance, it is too simple to rule it out as “bad.” It may certainly be appropriate, even in C++ source code, to use it.

As [Meyer, 97] points out, the most common problem with multiple inheritance is name clashing, that is, cases in which different features, inherited from different classes, have the same name. C++ solves this poorly while Eiffel is well defined in this respect. Another problem is that of repeated inheritance as shown in Figure 6. Again, there need to be explicit policies to handle this.

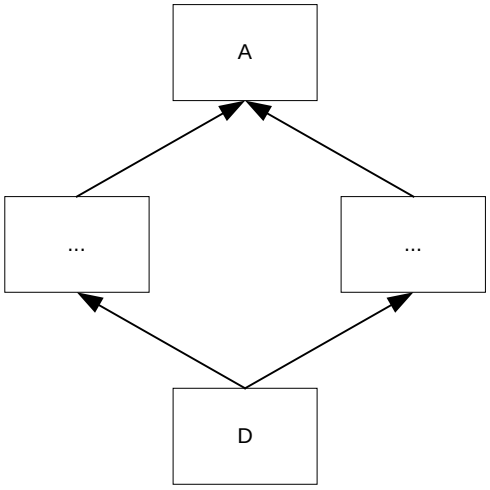

Figure 6 Repeated inheritance

[Meyer, 97] goes into great detail when discussing multiple inheritance and how to solve the various problems as is done in Eiffel. [Joyner, 99] confirms that the design of Eiffel works well. The focus here, however, is on C++, which may not deal well with these issues. Software metrics, then, should raise warning flags when multiple inheritance is used in C++ source code.

2.6.2. Encapsulation

The encapsulation part of object-oriented design has been given less attention in the metrics literature. This is not too surprising as encapsulation is an easily understood concept and is easily implemented as well. Encapsulation is a well-understood concept and easy to achieve, so systems tend to be fully encapsulated. There is, however, more to encapsulation than one might first think, and this will be briefly discussed here.

C++ does provide a particular mechanism for breaking encapsulation: the `friend` keyword. The way friends break encapsulation is a widely known issue in C++ circles and the language is frequently attacked for having such a destructive feature. Skilled C++ programmers, however, are aware of this and use friends sparingly. It is usually used in combination with operator overloading and, less frequently, with very tightly coupled classes, such as iterator classes and containers (Iterator pattern [Gamma, 94]). Although the `friend` keyword is considered to be best avoided, the use of it is limited and easily located, i.e. it is typically found where expected and rarely elsewhere.

Design patterns are generally considered to be good design practice. In particular, for the discussion of encapsulation and friends, the Iterator pattern has already been mentioned. More apparent is the Memento pattern, which has the intent that “without violating encapsulation, [to] capture and externalize an object’s internal state so that the object can be restored to this state later.” [Gamma, 94]. The way this is achieved is by letting the Originator class be a *friend* of the Memento class. Here we see an example where, in fact, the friend keyword is used *not* to violate encapsulation.

Encapsulation is often thought to be equivalent to data hiding. It is true that data hiding is encapsulation, but encapsulation is more than just that. The patterns community is talking about “find what varies and encapsulate it” [Shalloway, 02], meaning it is more than just data that is being encapsulated. A common object-oriented example is one that is shown in Figure 7. Except for the expected data hiding, the client is unaware of the various types of shapes. The concrete classes that the client is dealing with are hidden from the client. The variation of the shapes are encapsulated “behind” an abstract class and hidden. The power of this approach and the high frequency of undesirably large class hierarchies have led the patterns community to favor composition over inheritance to deal with variations. The Bridge pattern is a good example of contained variation. Commonality and variability analysis [Coplien, 98] is a way to find variations in the problem domain and identify what is common across the domain.

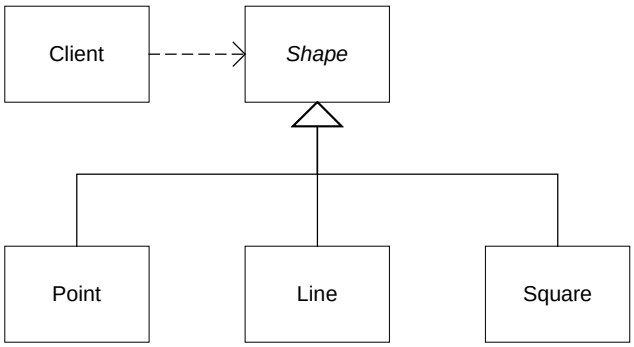

Figure 7 A common object-oriented example

Encapsulation should not be viewed strictly as data hiding, but as any kind of hiding, be it data, implementations, or derived classes. Design patterns use encapsulation to create

layers between objects, allowing the designer to change things on one side of the layers without affecting the other side. This promotes loose coupling. Recall from 2.2 that unstable dependencies are undesirable, i.e. exactly what this type of encapsulation prevents.

The use of composition instead of inheritance is something that would be useful to examine further with regards to software metrics.

In the case of C++ and friends, breach of encapsulation can be indicated with a count of the number of friends and their locations. It is more important to point to any public or protected variables. Any variables that fall within this category need to be examined and most likely moved to the private area, if necessary with matching functions to set and retrieve the value.

2.6.3. Polymorphism

Polymorphism is another concept in object-oriented design. It is, however, not as trivial as encapsulation. It is hard to quantify the impact polymorphism has on a system, and even harder to determine a desirable quantity of polymorphism.

The term polymorphism is used to describe how objects of different classes can be manipulated uniformly. Through polymorphic behavior, it is possible to write code that manipulates many different forms of objects in a uniform and consistent manner without regard to their individual differences. The flexibility and generality of polymorphic structures is one of the significant advantages of object-oriented programming [Kafura,

98]. Polymorphism is strongly connected to inheritance. It exists when the features of inheritance and dynamic binding interact [Booch, 94]. Consequently, it could be helpful to study the combination of inheritance and polymorphism from a software metrics point of view, particularly when taking into the account some of the problems with polymorphism in C++.

In C++, polymorphism is achieved through the `virtual` keyword. The problem with the approach C++ has taken is that the base class designer has to *guess* that a function might be polymorphic in one or more derived classes. If a function is not declared virtual, the possibility of polymorphism is closed. Unfortunately, virtual functions, coupled with the independent notion of overloading, leads to an error-prone combination in C++. Other object-oriented languages, such as Smalltalk, Objective C, Java, and Eiffel all use a different mechanism to implement polymorphism [Joyner, 99]. Simply stated, the problem with the virtual mechanism of C++ is that function redefinition and virtual functions are separate concepts [Meyers, 92]. In Java, everything is virtual, so there is no problem redefining non-virtual functions. In Eiffel, only redefined routines become virtual.

For completeness, the concepts of overloading and overriding will be explained here. Overloaded functions are different from polymorphic functions; the former is decided at compile time, the latter at run time. Overloading arises when two or more functions share a name. The number and types of the arguments separate these. Polymorphism, on the other hand, has one definition, and all types are subtypes of a principal type [Joyner, 99].

Because of the problem of overloading and overriding in C++, a metric that takes this into account could be useful.

Chapter 3

3. Gathering Object-Oriented Metrics

Penner's *m2* tool was chosen as a basis and extended to support various object-oriented metrics. The tool was chosen because the technique it uses has been proven to be stable and reliable. Moreover, the source code was available and under no restricting license.

3.1. Conceptual Source Code Module

The Conceptual Source Code Module (CSCM) technique provides a flexibility that cannot be achieved by complete language parser-based techniques. This flexibility comes from the minimal syntax recognized by the CSCM technique. The flexibility of the CSCM technique allows software measurement tools to overcome many of the problems encountered by tools using language based parsing techniques.

The CSCM technique only recognizes specific structures of the source code language. By minimizing the dependence of the processing technique on the language grammar, focus can shift from locating to measuring the code module. The CSCM technique minimizes its dependence on the C/C++ language by identifying constructs basic to the grammar. Examples of "basic constructs" are keywords, string literal, character literal, and structural constructs like blocks. By recognizing only minimal language syntax, it is possible to construct modules that can be measured regardless of the preprocessor.

The minimal syntax utilized by the CSCM technique enables the analysis tool to recognize and measure high-level patterns rather than correcting parsing errors. The design of the CSCM technique, discussed in the next section, builds on this minimal syntax to develop an architectural hierarchy of measurable code modules.

3.1.1. Concept and Design

When working with non-preprocessed source code files, traditional parsing techniques fail to perform reliably or produce valid and reliable measurements [Murphy, 96]. The inadequacies of parsing techniques come from the undefined grammar of the non-preprocessed C++ source code file. Specific problems identified for parsing techniques include: preprocessor parameterized macro return types, multiple function entry points based on preprocessor conditionals, and specialized source code specific to compiler implementations.

The CSCM technique is a process that aids in the measurement of C++ source code files without the need for any preprocessing.

Measuring non-preprocessed C++ source code files is a problem, and can be shown to be literally impossible from a deterministic point of view [Penner, 99]. However, estimates of complexity are possible, given certain assumptions. Most metrics tools avoid this problem by preprocessing, and those that do not preprocess do a relatively poor job of estimating complexity from non-preprocessed source code files. Better estimating tools are needed to successfully generate measurements on non-preprocessed source code files [Pearse, 95].

The CSCM technique is an effective method to aid measurement tools when analyzing C++ source code. When using the CSCM technique, the measurement process must be split into two phases. The first phase separates the source code file into modules and the second measures the modules that have been produced. Figure 8 shows the phases of source file measurement process. In the first phase the CSCM technique is used to separate the source code file into code modules. In the second phase, each code module is examined and metrics for that module are calculated. In *m3*, the second phase has been extended with a phase in which the graph created from the parsing is traversed.

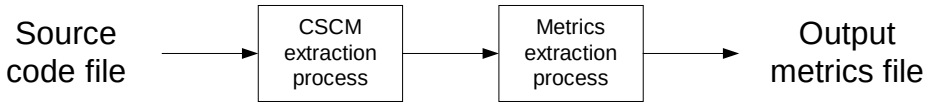

Figure 8 Source code file measurement process

A source code is defined as a measurable unit of the source code file. Better results can be achieved from separating the source code file into measurable units (functions, classes, and data) than by using traditional lexicographical and grammatical techniques.

3.1.2. The Code Module

The CSCM technique is based on the assumption that each source code file contains a collection of code modules. Four types of C++ source code modules are defined by the CSCM technique: Functions, Objects, Containers, and Data. To correctly separate the source code file into a collection of code modules the CSCM technique assumes the source code file will correctly compile. Hence, only complete implementations of

Function, Object, Container, and Data code modules can be used if the CSCM technique is going to be successful in processing the source file.

- **Function.** This module encapsulates the definition of a single function. A function definition includes the function header and the code that implements the function. Function prototypes are not considered a Function module.
- **Object.** This module encapsulates the definition of a single object. Objects include classes, structures, unions, and enumerations. The definition of an object includes the definition of both data and function members. The function members of a class or structure need not be defined but only prototyped. The Object module may contain additional modules including other Object modules. In *m2*, forward declarations of objects do not constitute Object modules. In *m3*, "dummy" object modules are created in order to build the module graph. The dummy is completed when (if) the definition is found.
- **Container.** This module is specific to C++ language. A Container module is a namespace *definition* and other contained modules. Namespace *usage* and alias definitions are not Container modules.
- **Data.** This module includes all data declarations as well as function prototypes and forward declarations of objects and containers.

3.2. Functionality in m2

The following metrics are supported in *m2*:

- Function Fan-in
- Function Fan-out
- Virtual Functions
- Member Functions
- Data Members
- Number of Comments (inside)
- Lines of Comments (inside)
- Number of Comments (outside)
- Lines of Comments (outside)
- Total Number of Comments
- Total Lines of Comments
- Blank LOC (inside)
- Non-blank LOC (inside)
- Blank LOC (outside)
- Non-blank LOC (outside)
- Total Number of Blank LOC
- Total Number of Non-blank LOC
- Total LOC
- Preprocessor Directives
- Cyclomatic Complexity
- Extended Cyclomatic Complexity
- Total Operator Count
- Total Operand Count
- Unique Operator Count
- Unique Operand Count
- Halstead's Volume
- Halstead's Effort
- Maximum Nesting Level
- Average Nesting Level
- Total Nesting Level
- Nesting Level Count
- Maximum Variable Length
- Average Variable Length

Table 3 Metrics in m2

3.3. Metrics gathered

The following metrics are those that are collected in *m2*. They are still available in *m3*.

Function Fan-in

Function fan-in is an inter-module coupling metric. Fan-in refers to the number of locations from which control is passed into the module (i.e. calls to the module) plus the number of reads of global data [Sellers, 96].

[Fenton, 96] notes that the fan-in metric is an information flow metric and defines fan-in of a module M as the number of local flows that terminate at M plus the number of data structures from which information is retrieved by M.

[Zuse, 98] defines fan-in as the number of edges going into a module. This includes the call of a module, parameters transferred by the call of the module, and reading global variables.

Based on [Zuse, 98], we can define fan-in formally as:

$$f_i(M) = c(M) + p(c(M)) + r_g(M)$$

where $f_i(M)$ is the fan-in of the module M, $c(M)$ is the calls of the module M, $p(c(M))$ is the parameters transferred by the calls of the module M and $r_g(M)$ is the reading of global variables accessed by M.

[Penner, 99] defines fan-in as the number of times the short module name is referred to by other modules in the source code files. This counts the number of times the given function module is called within the file.

Function Fan-out

Function fan-out is an inter-module coupling metric. Fan-out measures the number of other modules required plus the number of data structures that are updated by the module being studied (such as global variables) [Sellers, 96].

[Fenton, 96] notes that the fan-out metric is an information flow metric and defines fan-out of a module M as the number of local flows that emanate from M, plus the number of data structures that are updated by M.

[Zuse, 98] defines fan-out as the number of edges going out of a module. This includes the call of another module, parameters transferred by the call of the module, and writing into global variables.

Based on [Zuse, 98], we can define fan-in formally as:

$$f_o(M) = c + p(c) + w_g(M)$$

where $f_o(M)$ is the fan-out of the module M, c is the calls of another module, $p(c)$ is the parameters transferred by the calls of the module M and $w_g(M)$ is the writing of global variables accessed by M.

[Penner, 99] defines fan-out as the number of function calls made in a function module. This counts the number of function calls the given function module makes.

Virtual Functions

"Virtual functions" is a count of the number of virtual functions in a class. This does not include inherited functions. Formally:

$$f_{vf}(M) = \sum_{i=0}^{n} f_v$$

where $f_{vf}(M)$ is the virtual functions in the class M, n is the number of functions in M and f_v is a virtual function.

Member Functions

"Member functions" is a count of the number of functions in a class. This does not include inherited functions. Formally:

$$f(M) = \sum_{i=0}^{n} f$$

where $f(M)$ is the functions in the class M, n is the number of functions in M and f is a function.

Data Members

"Data members" is a count of the number of variables in a class. This does not include inherited variables. Formally:

$$v(M) = \sum_{i=0}^{n} v$$

where $v(M)$ is the variables in the class M, n is the number of variables in M and v is a function.

Number of Comments (inside)

"Number of comments (inside)" counts the number of comments within the outermost curly brace set, i.e. it counts the number of comments in a function or within a class declaration.

Lines of Comments (inside)

"Lines of comments (inside)" counts the number of comment lines within the outermost curly brace set, i.e. it counts the number of comment lines in a function or within a class declaration.

Number of Comments (outside)

"Number of comments (outside)" counts the number of comments that are outside of any curly braces.

Lines of Comments (outside)

"Lines of comments (outside)" counts the number of comment lines that are outside of any curly braces.

Total Number of Comments

"Total number of comments" is the sum of the number of comments inside and number of comments outside.

Total Lines of Comments

"Total lines of comments" is the sum of lines of comments inside and lines of comments outside.

Blank LOC (inside)

Inside blank lines of code are equal to the count of new-line characters appearing after the first open curly brace that only contains white space characters. "Blank LOC (inside)" counts the number of lines within a module that contains nothing but white space. This metric is only valid for objects and functions [Penner, 99].

Non-blank LOC (inside)

The number of inside non-blank lines of code is the count of lines after the first curly brace of a module. This counts the number of lines within a module that contains something other than white space. This metric is only valid for objects and functions.

Blank LOC (outside)

The number of outside blank lines of code is the count of lines before the first open curly brace of any module that only contains white space characters. This counts the number of lines outside of a module that contain nothing but white space. This metric is only valid for objects and functions.

Non-blank LOC (outside)

The number of outside non-blank lines of code is the count of lines before the first open curly brace of any module. This counts the number of lines outside of a module that contains something other than white space. This metric is only valid for objects and functions.

Total Number of Blank LOC

“Total number of blank LOC” is the sum of blank LOC inside and blank LOC outside.

Total Number of Non-blank LOC

“Total number of non-blank LOC” is the sum of non-blank LOC inside and non-blank LOC outside.

Total LOC

“Total LOC” is the sum of total number of blank LOC and non-blank LOC.

Preprocessor Directives

“Preprocessor directives” is the count of the number of preprocessor directives found in a given file.

Cyclomatic Complexity

The cyclomatic complexity metric is the count of conditional branches within a function module. This counts the number of different branching choices in a given module [Penner, 99]. [Sellers, 96] defines the cyclomatic complexity formally as:

$$V(G) = e - n + p$$

where e are the number of edges, n the number of nodes and p the number of connected components.

Extended Cyclomatic Complexity

The extended cyclomatic complexity metric is the count of all conditional statements including logical "&&" and "||" predicates. This counts the number of different branching choices in a given module using branching statements, logical AND predicates and logical OR predicates.

Total Operator Count

"Total operator count" counts the total number of operators found within a given module denoted by N_1.

Total Operand Count

"Total operand count" counts the total number of operands found within a given module denoted by N_2.

Unique Operator Count

"Unique operator count" counts the number of unique, individual operators within a given module, denoted by η_1.

Unique Operand Count

"Unique operand count" counts the number of unique, individual operands within a given module, denoted by η_2.

Halstead's Volume

"Halstead's volume" calculates the size of the module using the following formula [Penner, 99]:

$$V = N \times \log_2(\eta)$$

where

$$N = N_1 + N_2$$

and

$$\eta = \eta_1 + \eta_2$$

[Fenton, 99] notes that the volume of a program is akin to the number of mental comparisons needed to write a program of length N. The program vocabulary is represented by η.

Halstead's Effort

"Halstead's effort" calculates the estimated amount of effort that went into writing the module. The value is found by the following formula [Penner, 99]:

$$E = \frac{V^2}{(2+\eta_2) \cdot \log_2(2+\eta_2)}$$

The formula basically does a calculation of the volume and difficulty of the code to quantify the effort required to understand a function.

Maximum Nesting Level

"Maximum nesting level" counts the highest number of blocks and sub-blocks that are nested within each other within a module.

Average Nesting Level

"Average nesting level" counts the average number of blocks and sub-blocks that are nested within each other within a module.

Total Nesting Level

Total nesting level sums the different nesting levels present within a module.

Maximum Variable Length

"Maximum variable length" counts the number of characters in the longest variable within a module.

Average Variable Length

"Average variable length" counts the average number of characters in each variable within a module.

3.4. Introducing m3

The *m2* tool was a result of Penner's work [Penner, 99]. The number '2' in the name means the second version. The name is informal, but stuck. The changes made to *m2* can be seen as the third version, thus named *m3*.

3.4.1. Changes

The metrics in *m2* deal primarily with procedural design. Several new metrics needed to be introduced in order to measure object-oriented design. The current metrics (Table 3) are mostly single-class counts and therefore give a considerably restricted, one-dimensional view of the system. These metrics, typically considered “basic” in the metrics literature, are useful, but important information, such as coupling, is not gathered. Inter-module metrics are fundamental in order to understand multidimensional properties such as reusability, coupling and complexity, all of which are related to maintenance. That being said, further basic metrics can also be added. The focus so far has been on primarily code metrics. For this thesis, some design metrics have been added.

Depending on the metric to add, the calculation can be accomplished in one of two ways: (1) calculate a value from the already present information (trivial), or (2) add new information to gather. The latter may require a substantial amount of work depending on the information needed and how it fits with the design of *m2*. The metrics listed later in this section were reasonable to add to *m2*. The metrics have been restricted to those that will fit well with C++ and object-oriented design. Some traditional metrics based on structured languages do not fit well with object-oriented languages, and have therefore been ignored.

We will focus on object-oriented metrics, but unfortunately, most of them are fairly generic and do not take into account special C++ features such as the `protected` and `friend` keywords. Although these keywords are recognized and counted by *m2*, there is

no metric calculation based on these numbers. Further study is needed in this area. See 2.6 for some discussion on this.

The fundamental idea of *m2* is the conceptual source code module (CSCM) [Penner, 99]. The various metrics will, therefore, be categorized based on which module they fit in. Some metrics, such as the number of classes, is a system wide metric, that is, it takes into account the entire source code. There is no module in *m2* that represents the entire project. However, the proposed addition of a set module will take this into account; simply let all the files in the project be a set.

The modules in CSCM are:

- File
- Container
- Object
- Function
- Data
- Set (proposed)

[Briand, 97] classify cohesion measures in domains, which is similar to CSCM's modules. The domains proposed are:

- Attribute
- Method
- Class
- Set of classes
- System

The study [Briand, 97] showed that most cohesion measures fit only in the class domain, and measuring cohesion from a set point of view is restricted to the information-flow based cohesion metric (ICH) as defined by [Lee, 95].

From an implementation point of view, the Set module is simply a global structure in the *m3* code that is updated as other modules are parsed and measured.

Arguably, the biggest change to *m2* was moving away from the idea of just keeping a single metric structure in memory at any given time. Previously, *m2* would read one file, gather metrics from it, write the output to file and then flush the structure from memory. This limited the possibility to gather metrics, as many metrics require "neighboring" entities to be examined as well. When calculating the depth of the inheritance tree, for example, the entire inheritance hierarchy must be kept in memory. To achieve this, *m2*

was modified to link together the metric structures into a graph similar to the inheritance hierarchy (Figure 9).

As a result, *m3* now requires more memory than before. In fact, there is now a limit to how much *m3* can parse. Earlier, *m2* could, theoretically, parse an infinite amount of source code and use a roughly constant amount of memory, but this is no longer possible. The new design requires more and more memory as more files are parsed. In practice, however, the increased memory usage is not a problem. The added metrics also makes *m3* slower than before as more calculation is performed.

Figure 9 shows how the metric structures are connected together in memory after having processed three classes (`object`, `string`, `unicode`) and the six files used to declare and define these classes. Table 4 shows the header files, which are the only files needed in order to create a memory structure as shown in Figure 9. Let the implementation files (not shown) be stubs.

```
class object
{
public:
   object();
   virtual ~object();
   virtual void f();
};
```

```
class string : public object
{
public:
   string();
   virtual ~string();
   void append(const string&);
   void append(const char*);
};
```

```
class unicode : public string
{
   unicode();
   virtual ~unicode();
   virtual f();
};
```

Table 4 Source code for object.h, string.h and unicode.h.

As we see from the code, all the destructors are virtual and, from the figure, we see that the boxes corresponding to the constructors are grayed out in order to indicate that they are virtual. The method `f()` is also virtual and is designated in the same way. From the code we can also see that `unicode` derives from `string` which derives from `object`. This is also shown to the right in the figure.

Except for the inheritance hierarchy shown to the right in the figure, each box represents a metric structure in memory. There will be one structure for each file, one for each class name and one for each function. The metrics structure holds data about the module it represents.

The classes are declared in the header files, so the metric structures representing those files each point to a structure representing the class that was declared in that file. The class metric structure points to the first method found in the class, typically the constructor, and then that method metric structure points to the next method and so on. The implementation file also points to the first method found.

There are also backward pointers. These point to parents or overridden methods. For example, each derived class is pointing to its parent class. Each method points to the method that was overridden, if any. For example, all the destructors, that are marked virtual, point to the destructor of the parent class.

The graph created by the connected metrics structures is traversed and analyzed at the end of the parsing of all the source code. The various metrics requiring information from such a graph is then gathered. Examples of such metrics are given in Figure 10.

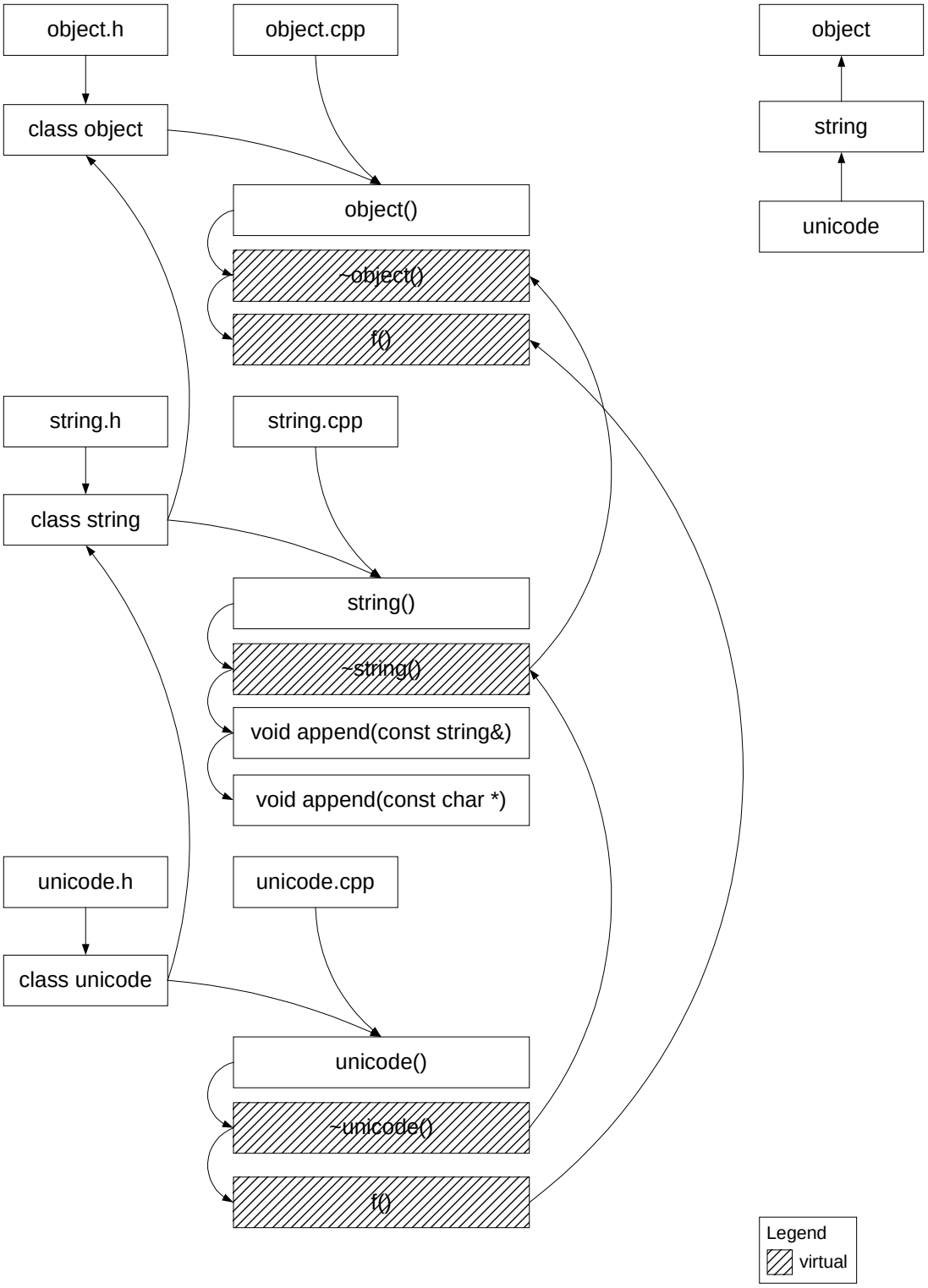

Figure 9 How metric structures are connected in m3

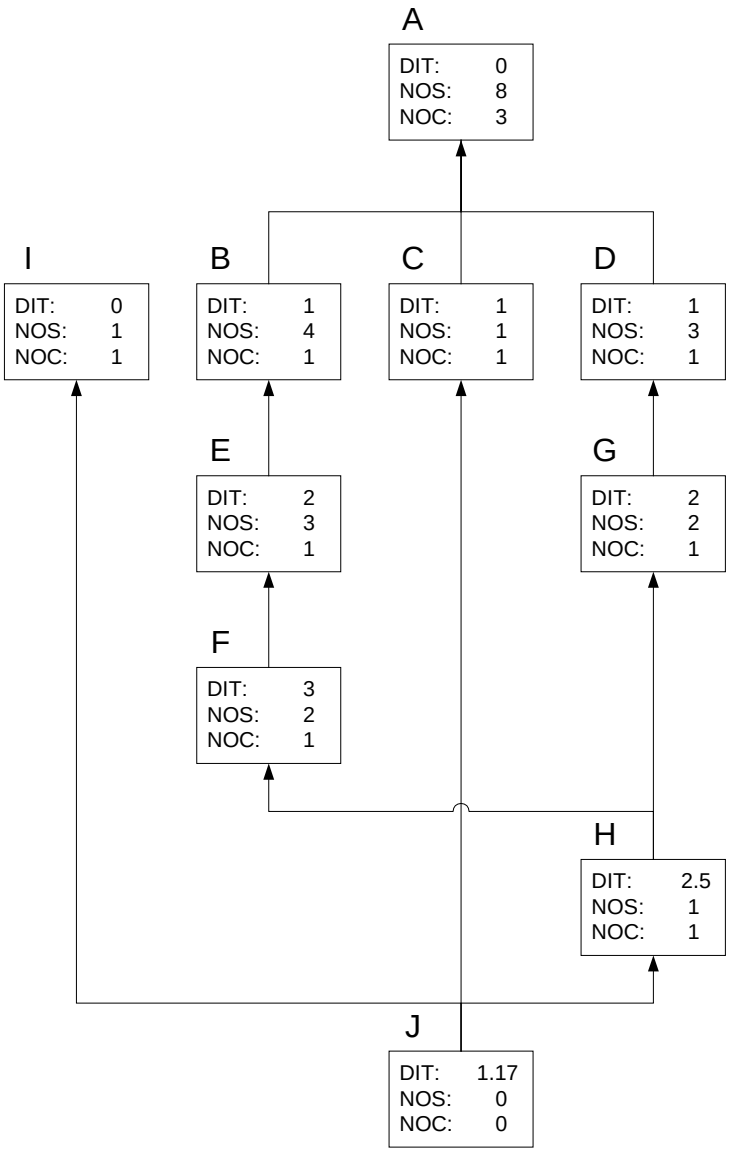

Figure 10 A sample inheritance hierarchy

While Figure 9 shows a detailed level of the *m3* internals, Figure 10 shows an inheritance hierarchy of 10 classes named A, B, C, …, J. Each class has three sample metrics printed in them. Figure 11 shows how this inheritance hierarchy appears within *m3*. The graph can be fully traversed in any direction from any point.

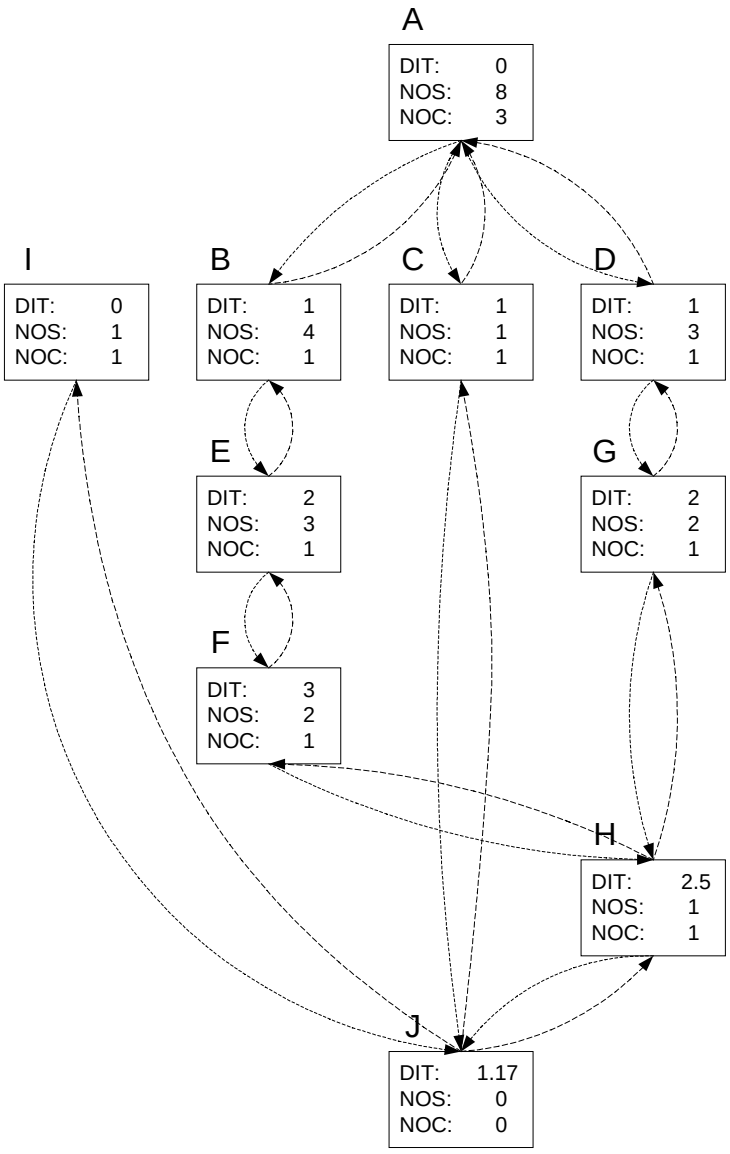

Figure 11 How m3 connects class structures together.

3.4.2. Sets

It is necessary to refine how the CSCM technique works on files, especially in large-scale software. We introduce the concept of a set in CSCM:

> *Set: A set is a list of file names. A list can consist of both header and source code files.*

By using sets, we introduce the possibility to restrict measurement to file scope:

- Fan-in and fan-out
- Better estimate on C++ classes (header and source files can be measured together)
- Multiple files: inheritance, composition

In *m3*, the set is simply the files that are sent to *m3*. If different sets are required, the tool needs to be run several times, once for each set.

3.5. Object-Oriented Metrics Gathered

The object-oriented metrics gathered by *m3* are given in Table 5. The dependencies between the various metrics are given in Figure 15.

Metric	Abbreviation
Number of Children	NOC
Number of Subclasses	NOS
Number of Methods Added	NMA
Number of Methods Overridden	NMO
Number of Methods	NOM
Information Flow Metrics	IFM
Depth of Inheritance Tree	DIT
Halstead's Program Vocabulary	HPV
Halstead's Program Length	HPL
Number of Classes	NOCL
Reuse Ratio	RR
Specialization Ratio	SR
Specialization Index	SIX
Maximum Number of Parameters	MNOP
Average Parameters per Method	APPM
Weighted Methods per Class	WMC
Comment Density	CD
Method Hiding Factor	MHF
Method Inheritance Factor	MIF
Maximum Size of Operation	MSOO

Table 5 Metrics added

NOC

The NOC metric (Number of Children) counts the number of immediate children a particular class has. An example class hierarchy is given in Figure 12 and shows how the NOC metric works:

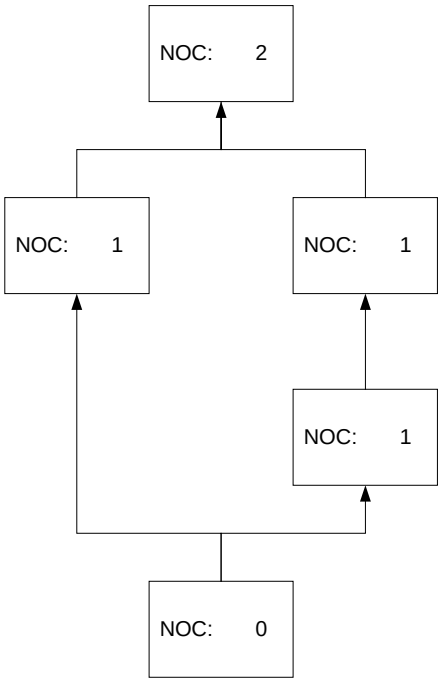

Figure 12 Example of the NOC metric

Classes with a large number of children are difficult to modify and usually require more testing because the class potentially affects all of its children. Thus, a class with numerous children has to provide services in a larger number of contexts and must be more flexible. It is suspected that this introduces more complexity into the class design [Basili, 95].

Although not the same as maintainability [Basili, 95] found that the NOC metric is very significant in fault detection. The results were surprising: The larger the NOC, the lower the probability of defect detection. [Basili, 95] explains this as occurring because most classes do not have more than one child and verbatim-reused classes are somewhat associated with a large NOC. [Basili, 95] found, in another study, that reuse was a significant factor in fault density and, as such, explains why large NOC classes are less fault-prone.

[Chidamber, 91] reports that 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 the class.

As a side note, [Basili, 95] says that only the NOC metric is somewhat unstable across different problem domains.

NOS

The NOS metric (Number of Subclasses) counts the total number of children of a particular class. An example hierarchy is given in Figure 13 and shows how the NOS metric works:

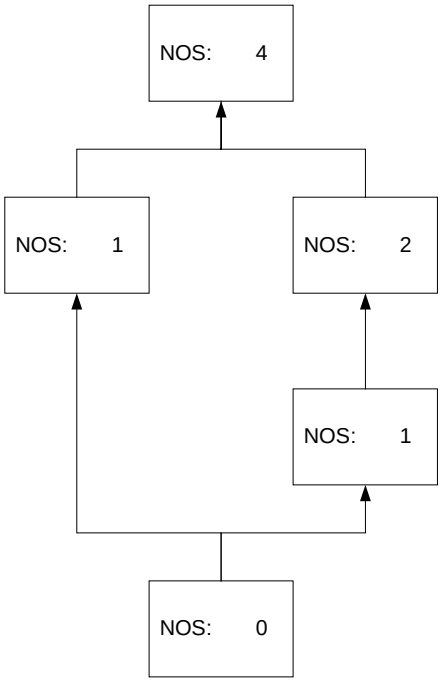

Figure 13 Example of the NOS metric

Its usage is similar to that of NOC.

NMA

The NMA metric (Number of Methods Added) is the number of new methods in a particular class, i.e., methods that were not inherited or overridden from the parent. The higher the number, the more distinct the class is from its parent and one may consider whether it should be a child class at all (or split into several classes deriving from the parent).

NMO

The NMO metric (Number of Methods Overridden) is simply the number of methods redefined in a particular class. In Figure 9 these are the functions that are grayed out.

NOM

The NOM metric (Number of Methods) is a simple count of the number of local methods. This has been refined in *m3* and is divided into number of public, protected and private methods.

[Li, 93] uses NOM as a "class interface increment metric" to measure the complexity of a class.

IFM

The IFM metric (Information Flow Metric) proposed by [Henry, 81]:

$$S_s \times (fanin \times fanout)^2$$

where S_s is the module size in lines of code. According to [Fenton, 96], however, the Henry-Kafura metric has been studied and an improved version is simply:

$$(fanin \times fanout)^2$$

This is how *m3* calculates IFM.

The IFM metric is meant to capture information on system design complexity. It is an inter-module metric. It measures complexity in terms of data connections "fanning in" and "fanning out" from modules. [Henry, 81] reported an impressive 0.98 correlation between percent-of-modules-changed and the latter definition. Another study by Henry gave the IFM metric 95% confidence level in predicting complexity.

DIT

The DIT metric (Depth of Inheritance Tree) measures at what depth the current class is. It was first introduced by [Chidamber, 91]. An example hierarchy is shown in Figure 14 and shows how the DIT metric is calculated.

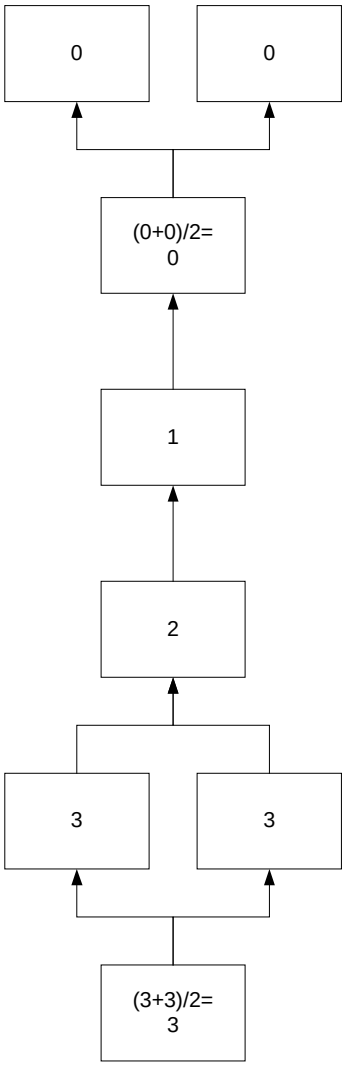

Figure 14 Example of the DIT metric

The assumption behind this metric is that well-designed object-oriented systems are those structured as forests of classes, rather than as one very large inheritance lattice.

Consequently, the larger the DIT value for a class, the greater the number of variables and methods it is likely to inherit, and therefore the more difficult it is to predict its behavior.

Design patterns also tend to have very shallow trees, with few classes having a depth of more than 3 [Shalloway, 02]. One reason for this is that, as [Masuda, 99] notes, some design patterns promote object composition rather than class inheritance. Consequently, the DIT values become low and may give a wrong indication about the design. In fact, [Shalloway, 02] notes that favoring composition to inheritance is a mantra in the design patterns community. One of the reasons for this is because of the sometimes-undesirable large class hierarchies that result of dealing with variations.

[Basili, 95] found the DIT metric to be very significant in fault detection.

[Hsia, 95] indicates that designs with a smaller broadness factor yield tall and slim inheritance trees that result in greater dependency among objects. This introduces complexity into the system and makes maintenance difficult. Implied here is: more reuse does not necessarily mean more maintainability. [Hsia, 95] reports that systems with a larger broadness factor (i.e. less reuse) are more maintainable than systems with a smaller broadness factor (i.e. more reuse). Consequently, reuse and maintainability are two factors of object-oriented design that may not be possible to increase simultaneously.

When designing object-oriented software, developers must deal with a conflict between the advantages of inheritance (increased reuse, and improved similarity of implementation and problem structure) and its disadvantages (increased coupling and complexity).

[Booch, 94] recommends a maximum number of inheritance levels at 7±2 (a maximum inheritance depth of 6±2). [Harrison, 98] reports low to moderate use of inheritance, although this does not refer to the DIT metric, but rather the MIF and AIF metrics from the MOOD suite.

It is worth pointing out that the metrics literature does not make a distinction of (in fact, does not even mention) what type of inheritance is used. C++ supports public, protected and private inheritance, even virtual inheritance. Although public inheritance is by far the most used type, *m3* keeps track of which kind of inheritance is being used. Multiple inheritance is another feature that is rarely discussed in the metrics literature, but *m3* keeps track of it. This information may be used in the future when multiple inheritance has been studied closer with regard to software metrics.

See also the RR metric, and the SR metric.

HPV

The HPV metric (Halstead Program Vocabulary) is calculated as the number of unique operators plus the number of unique operands:

$$HPV = OPR_U + OPD_U$$

HPL

The HPL metric (Halstead Program Length) is calculated as the number of operators plus the number of operands:

$$HPL = OPR + OPD$$

NOCL

The NOCL metric (Number of Classes) is simply the total count of classes.

RR

The RR metric (Reuse Ratio) is defined as:

$$\frac{NOS}{TOC}$$

where NOS is the number of subclasses and TOC is the total number of classes. A value near one is characteristic of a linear hierarchy and a value of near zero indicates a shallow depth and large number of leaf classes.

See also the DIT metric.

SR

The SR metric (Specialization Ratio) is defined as:

$$\frac{NOS}{NOSC}$$

where NOS is the number of subclasses and NOSC is the number of super classes. This metric measures the extent to which a super class has captured the abstraction. A large value indicates a high degree of reuse by subclassing.

SIX

The SIX metric (Specialization Index) is suggested by [Lorenz, 94] and is defined as:

$$\frac{NMO \cdot DIT}{NOM}$$

where NMO is the number of overridden methods, DIT is the depth of the inheritance tree and NOM is the number of methods in the class. This metric is an indicator of where a class should be placed in the inheritance hierarchy and for design problems.

MNOP

The MNOP metric is simply the maximum number of parameters in the given class. It can be used to locate functions that may need to be redesigned.

APPM

The APPM metric is the average number of parameters per method in a class. The metric is suggested by [Lorenz, 94] as he states that good object-oriented design should pass relatively few objects as parameters to messages. [Lorenz, 94] suggests an upper limit of about 0.7.

WMC

The WMC metric (Weighted Methods per Class) is invented by [Chidamber, 91]. It tries to measure the static complexity of all methods in a class. For a class C with methods $M_1, M_2, \ldots, M_n$ weighted respectively with complexity $c_1, c_2, \ldots, c_n$ the measure is calculated as:

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

In other words, WMC is the sum of the complexities for all methods of the class. The larger the WMC value for a class, the more complicated and expensive it is to maintain. That is, the assumption behind this metric is that a class with significantly more member functions than other classes in the same set is more complex and, therefore, tends to be more error-prone.

The complexity of each method is calculated using McCabe's cyclomatic complexity.

[Basili, 95] found that WMC was "somewhat significant" in detecting fault probability. The larger the WMC the larger the probability of fault detection. The results can be explained by the fact that the internal complexity does not have a strong impact if the class is reused verbatim or with very slight modifications. In this case, the class interface properties will have the most significant impact [Basili, 95].

As noted by [Masuda, 99], WMC may not always give the correct impression. In particular, some design patterns, such as Command, Strategy, and Factory Method

[Gamma, 95], make object granularity small with a low WMC value. An application designed heavily with design patterns such as these will, therefore, have a relatively low WMC value, but may, in fact, have a high complexity.

[Chidamber, 91] reports that the number of methods and their complexity is an indicator how much time and effort is required to maintain the class. Also, the more methods a class has, the more impact it has on its children since the public and protected methods will be inherited. Lastly, classes with a large number of methods are more likely to be application specific and therefore limit the possibility for reuse.

Other patterns, on the other hand, such as the Meditator pattern [Gamma, 95], will give high WMC values. The Meditator pattern makes coupling between colleagues loose, but centralizes control to ConcreteMeditator.

CD

The CD metric (Comment Density) is proposed by [Fenton, 96]. It measures the density of comments in a program:

$$\frac{CLOC}{LOC}$$

where CLOC is the number of comment lines. This is a simple metric, but comments are important in order to understand a program. [Fenton, 96] does not suggest any threshold for CD, but it should be fairly high. Object-oriented functions tend to be small and each function should be commented.

MHF

The MHF metric (Method Hiding Factor) is from the MOOD suite [Abreu, 95] and is calculated as the number of invisible operations in the project divided by the number of visible operations. It measures the encapsulation part of object-oriented techniques. A low value indicates lack of information hiding. It is expressed as a percentage ranging from 0% (no use) to 100% (full use). The metric is dimensionless which avoids any artificial units that dominate the metrics literature.

[Harrison, 98] points out the trouble with this metric and C++ due to its 'protected' feature. A simplified approach has been taken in *m3*: protected functions are considered to be invisible.

[Abreu, 96] reported that MHF has a moderate negative correlation with defect density and rework. That means that once MHF increases, the defect density and the effort spent to fix defects will be expected to decrease.

There is a related metric in the MOOD suite: AHF (Attribute Hiding Factor). This has not been implemented. Information hiding implies that attributes should always be private. Therefore, AHF should always be 100% as is also acknowledged by [Harrison, 98].

Formally, MHF is defined as [Abreu, 96]:

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

where

$$V(M_{mi}) = \frac{\sum_{j=1}^{TC} is\_visible(M_{mi}, C_j)}{TC - 1}$$

and

$$is\_visible(M_{mi}, C_j) = \begin{cases} 1 \quad iff \quad \begin{cases} j \neq i \\ C_j \ may\ call\ M_{mi} \end{cases} \\ 0 \quad otherwise \end{cases}$$

and TC is the total class count and $M_d(C_i)$ is all methods declared in the class C_i (not inherited). $V(M_{mi})$ refers to the visibility, i.e. the percent of the total classes from which the method M_{mi} is visible.

MIF

The MIF metric (Method Inheritance Factor) is from the MOOD suite [Abreu, 95] and is calculated as the number of inherited methods in the project divided by the total number of methods. It measures the inheritance part of object-oriented techniques. It is expressed as a percentage ranging from 0% (no use) to 100% (full use). The metric is dimensionless which avoids any artificial units that dominate the metrics literature.

[Abreu, 96] reported that MIF has a moderate negative correlation with defect density and high negative correlation with both defect density and normalized rework measure. That means that once MIF increases, the defect density and the effort spent to fix defects will be expected to decrease. The report goes on to point out that inheritance appears to be an appropriate technique to reduce defect density and rework when used sparingly. Very high values of MIF are believed to reverse this beneficial effect.

There is a related metric in the MOOD suite: AIF (Attribute Inheritance Factor). This has not been implemented. See section on AHF.

Formally, MIF is defined as [Abreu, 96]:

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

where

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

where $M_a(C_i)$ is available methods, $M_d(C_i)$ is defined methods and $M_i(C_i)$.

MSOO

The MSOO metric (Maximum Size of Operation) is simply the highest cyclomatic complexity of the functions in a class.

NOF

The NOF metric (Number of Friends) is a metric suggested by the author and is simply the count of friend methods. Since 'friends' are considered undesirable in C++ programs, a dedicated metric may be interesting.

[Stroustrup, 97] advises "don't use friends" (pp. 16) except to avoid global data and public data. The reason is that friends have access to the private part of the class declaration. In other words, friend functions break encapsulation.

[Joyner, 99] points out that friends have three problems:

1. They can change the internal state of objects from outside the definition of the class.
2. They introduce extra coupling between components and therefore should be used sparingly.
3. They have access to everything, rather than being restricted to the members of interest to them.

Figure 15 shows the dependencies between the various metrics. It was useful during the development of *m3* to see in which order the metrics had to be implemented and calculated. The figure should be useful for any future work on *m3* as well.

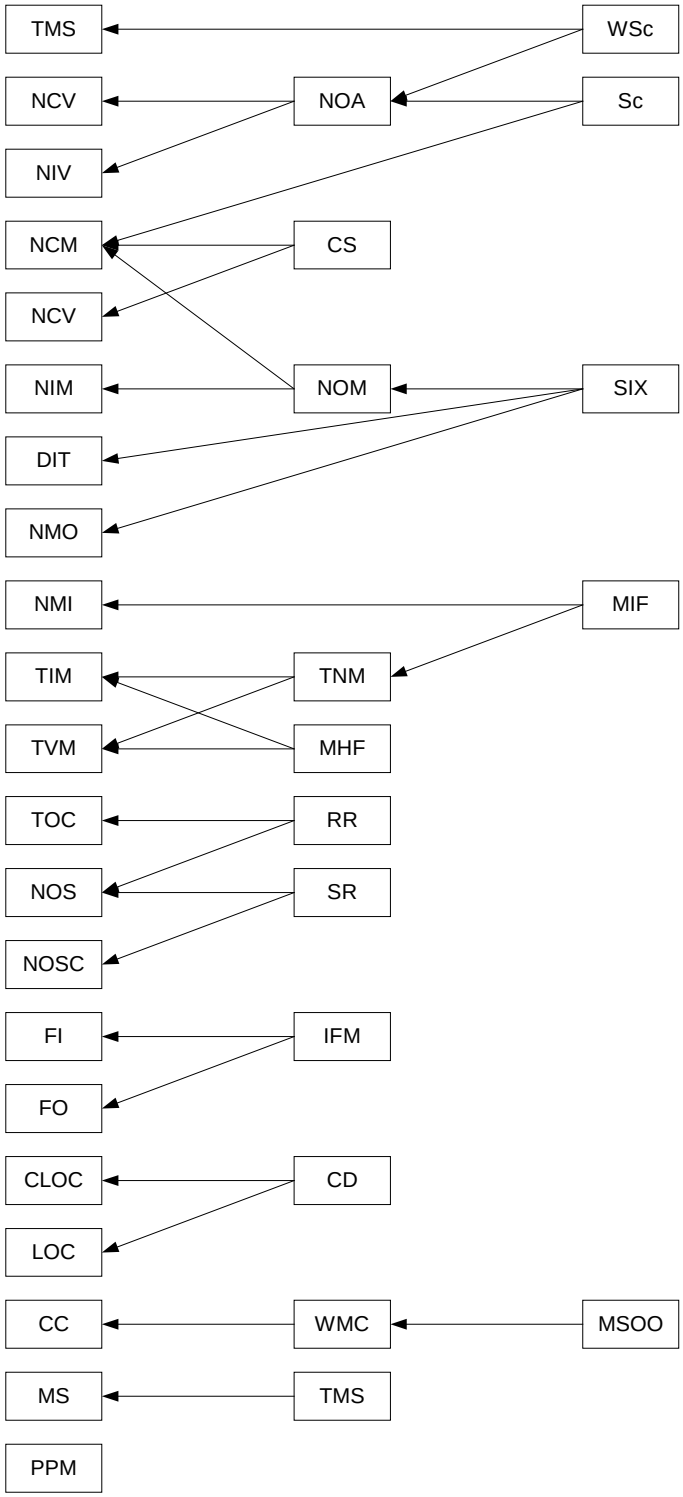

Figure 15 Metrics dependencies

3.6. mvt

The output produced by *m3* is in XML format. The file can be browsed using the *m3* visualization tool, *mvt*. The tool was a senior design project and was developed in conjunction with Intel.

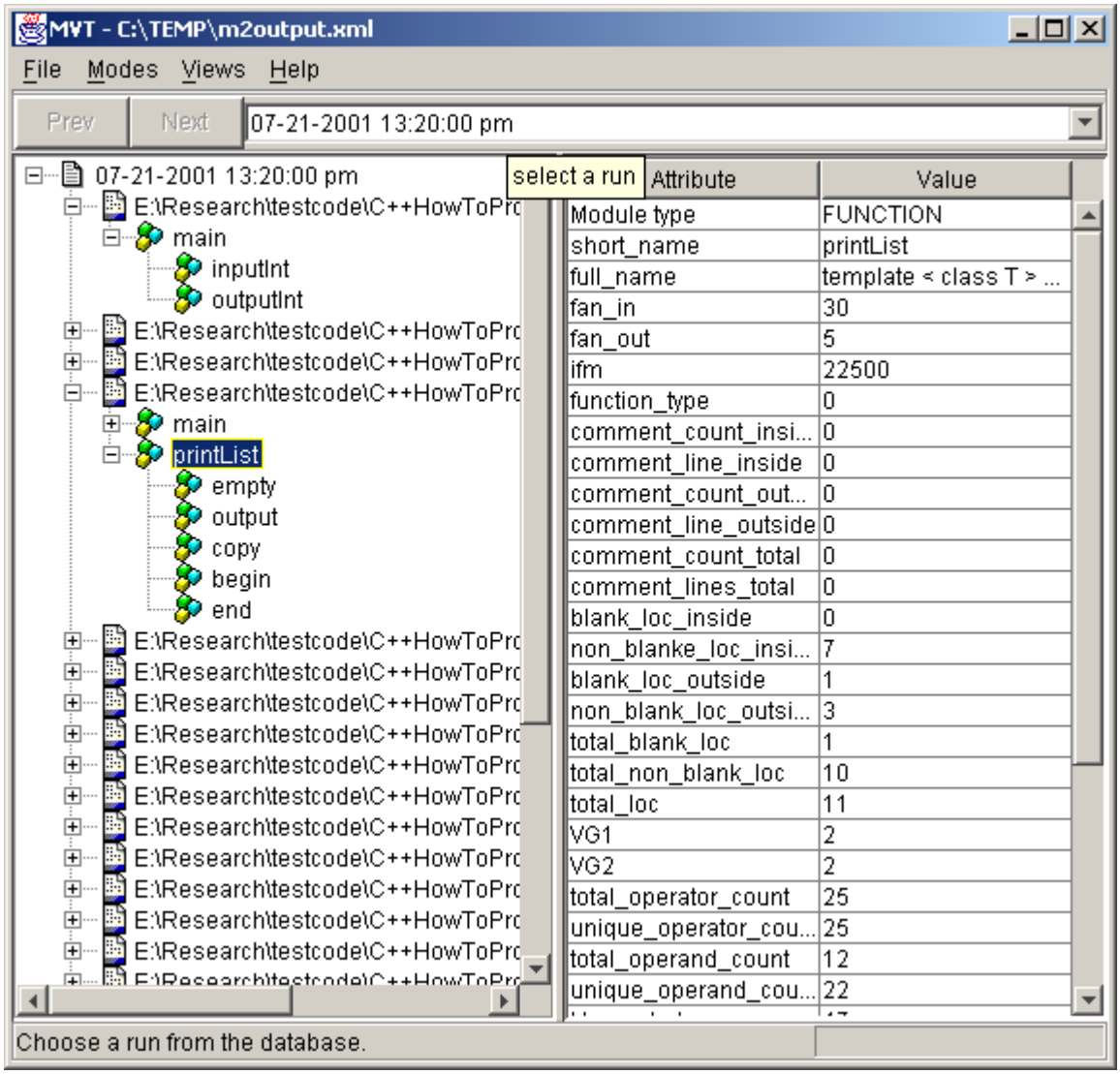

Figure 16 Screenshot of mvt

3.7. mmf

To ease the usage of *m3* and *mvt*, a simple front-end was developed: *mmf* is the *m3* & *mvt* front-end. With it, a list of source code can be built to feed to *m3* and execute *mvt* to browse the result. The tool is developed in Java using Swing for its user interface.

Figure 17 shows how a set (see 3.4.2) is defined and sent to *m3*.

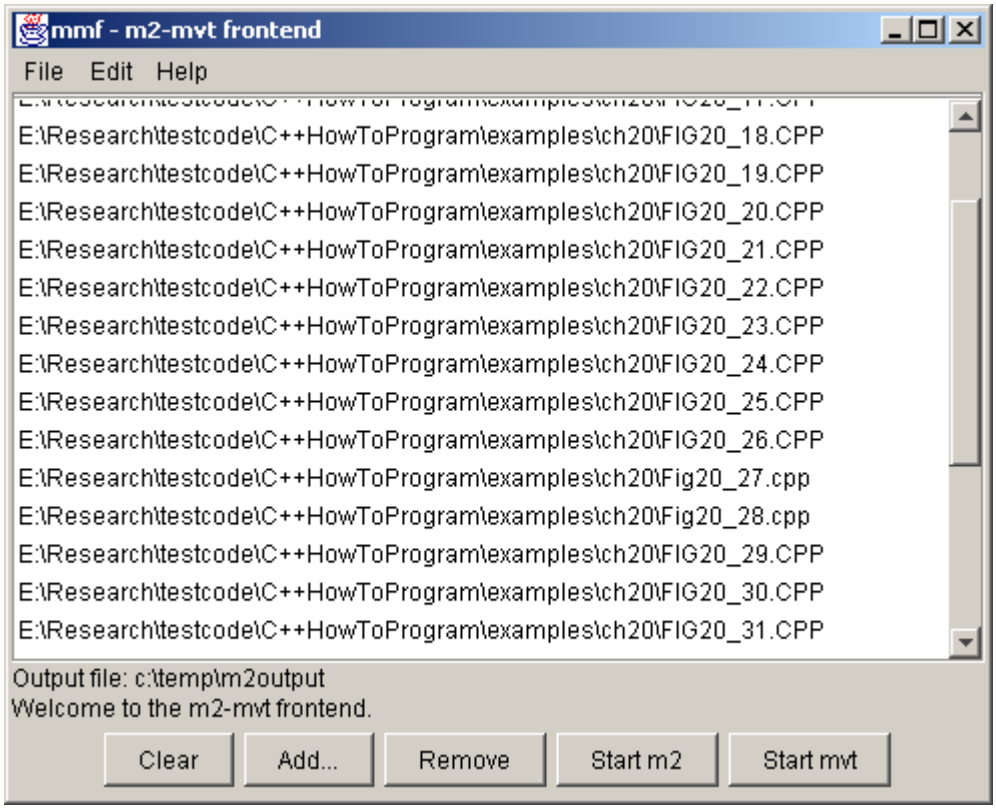

Figure 17 Screenshot of mmf

Chapter 4

4. Developing Maintainability Index Candidate Models

The goal of this research is to develop maintainability index (MI) models that can be easily computed with tools that will indicate the maintainability of large object oriented software systems. That is, the MI should give a scalar indication of the effort that may be required to perform maintenance activities on the software. The first step is to develop a set of candidate maintainability index models. This is done by extracting metric information about the software system with a tool, such as *m3*, and discovering a relationship with the maintainability of the software as determined by software developers. A statistical analysis process is used to discover the relationship between source code metrics and the maintainability of the software.

The first step is gathering an initial sample of object-oriented C++ program modules for evaluation in order to determine what source code metrics may be good indicators of software maintainability. Research assistants, who are fluent in C++ and object-oriented programming, determine the maintainability of each source file. A maintainability survey (see Appendix C) based on [Hagemeister, 92] but modified to include object-oriented evaluations was completed by each research assistant for each code set. This establishes a ranking of the sample set which is used as the dependent variable in the analysis. The second step is to reduce the number of metrics to use and identify those metrics that best represent the complete metrics set for each sample module. This is done with principal component analysis. Once candidate metrics have been identified, linear regressions are

completed to develop the relationship models to the programmer ranking to create the MI candidates. This chapter documents the process of developing candidate MI models.

4.1. Source Code Sample Set

Source code of varying quality was identified by posting at various places on the Internet, such as the newsgroups *gnu.misc.discuss, comp.lang.c++* and *comp.object*. The software modules selected as the sample set was based on suggestions received from the request and by an initial review. The sample modules come from the following systems, which are all in the public domain open source systems under the GNU General Public License [12].

NiL [5] is a 2D computer game developed primarily by Flemming Frandsen. Jan Borsodi suggested it as a sample set.

Qwt [6] is a graphics extension to the *Qt* [7] GUI application framework primarily developed by Josef Wilgen. It provides 2D plotting widgets and other widgets not present in *Qt.*

DOC++ [8] is a documentation system for C, C++, IDL and Java, generating both TeX output for high quality hardcopies and HTML output for sophisticated online browsing of documentation. Primarily developed by Dragos Acostachioaie.

Qt [9] is a multi-platform application framework library. *Qt* provides a platform-independent API to all central platform functionality: GUI, database access, networking, file handling, etc.

4.2. Evaluations of the Sample Sets

The sample software was partitioned into 8 different samples for evaluation. It was partitioned initially into four categories; very low maintainability, low maintainability, medium maintainability, and high maintainability. Based on the initial screening three samples were very low, one sample was low, one sample was medium, and two samples were high. They are briefly described below.

Sample 0/A

Rated: Very low maintainability

The design and organization of this code module is generally confusing. The class design is consistently poor, with all variables declared as `protected` (the `private` keyword is not even used); there is no use of polymorphism, and consistent lack of virtual destructors effectively prevent any proper inheritance from the classes. The class design is frequently divided into a constructor and an initialize function with no apparent reason for this being the case. This leads to the possibility of invalid objects, i.e. objects created but not initialized. There are several instances of odd inheritance usage from STL classes. Their function is not commented and neither is the use of the inheritance. It appears that inheritance has been used as a replacement for `typedef`. Although `typedefs` are generally best avoided, they are commonly used together with STL and would there not

concern or confuse a maintainer. Instead, inheritance has been used here. Another confusing element is the mix between structures and classes. Although a structure is identical to a class in C++ except for the default visibility, it is inconsistent and confusing to a maintainer to suddenly use structures instead of classes. The naming convention is poor with no difference between the naming of class variables and local variables and function parameters. This leads to the convention that function parameter names that conflict with class names has an underscore appended. Since far from all function parameters conflict with class variables, there is an inconsistency of the function parameters; some with underscore, some without. There are few or no relevant comments. The comments generally explain details in the code rather then design and dependency information. The functions and their formal parameters are not commented. The purpose of each class is not documented. Finally, there is an inconsistent use of forward declarations versus include statements.

Sample 1/A

Rated: Low maintainability

This is source code from the same software as Sample 0/A. The overall feel of the source code and design is the same so a lot of the problems identified above apply here too, but this set it slightly better. The encapsulation is better. There are some private variables. The use of inheritance is understandable, although still somewhat confusing and it does not promote reuse. A major problem with this set is that a few functions are extremely long, covering several pages. Following the logic in these functions is hard. The code itself is computationally heavy. The math is not complicated, but there is quite a bit of it.

The application must accomplish the computation, but it is excessively obfuscated from the maintainer what the required operation is. Extra care should be taken to not overwhelm the maintainer. At the very least, more comments are required. Other functions are complex, with many arrays, maps, lists etc, as well as a complicated flow of program control.

Sample 2/A

Rated: Medium maintainability

This set contains some source code from the same application as the two previous samples, but primarily it consists of source code from another application. This makes the overall feeling of the code a little inconsistent as the code related to Sample 0/A and Sample 1/A has a lower quality than the rest. The lowest quality part lacks encapsulation; now even with a public variable, and the rest remain protected. This one class is still better than many of the classes in the two other sets as this one actually opens up for inheritance by the use of virtual keywords. The inheritance used is understandable. The remaining part of the set, the majority of the code, is of generally higher standard. The code is impressively well documented, with extensive explanation of each function and all function arguments and return values. There is not, however, much comments in the code itself, but this is far less important than a general explanation of all the functions. The classes are also commented well. The classes are fairly well designed, however the function names tend to be unnecessarily abbreviated, which makes the code a little less readable. The classes also implement copy constructors and the equality operator, two parts of C++ class design that is frequently overlooked. The biggest problem with the

design is too long. The classes seem rather large and splitting them into smaller classes may be beneficial. The code worryingly lacks any kind of asserts or debug checks/outputs. Although many of the assumptions are clearly documented, they are just that and not checked for. Simple asserts could improve this. Another problem, although much smaller, is the occasional weak encapsulation. Several class variables are just protected and not private. The class interfaces also are short on the use of virtual. This may be a good thing in this particular case, but it makes extension through inheritance next to impossible. It also worth mentioning that there is almost no use of abstract classes (interfaces), again which makes it hard to extend the functionality. The code is relatively heavy on the math side, but since this is a computationally heavy application, this is expected. To summarize, the major weakness is lack of use of abstract classes or at least making it difficult to extend the current classes. Splitting up the classes may improve the code. If not that, splitting some of the very long functions would greatly improve the readability.

Sample 3/A

Rated: High maintainability

The code is highly maintainable, the functions are reasonably size, there level of encapsulation is high, and commenting provides assistance in understanding the functionally. However, the design doesn't provide good extensibility and may be difficult to reuse. There is poor separation between interface and implementation. The interface design is the primary limit on the extensibility. Most functions are reasonably sized, which makes them easy to understand. There are a few exceptions, especially the ones

that calculate and draw lines, “splines”, but this is understandable since they implement complex algorithms. Overall, encapsulation is fairly good, but there are still non-private variables. In fact, there are quite a few. Use of assertions and similar debug techniques would help. To summarize, the quality of this set is similar to that of Sample 2/A, but the classes are smaller in this set and there are fewer oversized functions. Both of these factors improve the maintainability to a higher level.

Sample 0/B

Rated: Very low maintainability

The classes are badly encapsulated with most variables public, a few protected and none private. There is inconsistency with the use of inline functions and non-inline functions. Those that are inline appear to be so at random. There is little use of polymorphism, and the class design does not permit extension. There are some long and complicated functions and also some strange inheritance usage as a replacement for `typedefs`, similar to that in Sample 0/A. No use of assertions to verify assumptions, although the code is scattered with output to an error log if something fails. The documentation is non-existent. Some comments in the code itself, but all functions are mostly uncommented and no function arguments are explained. The code itself is complex. This is primarily because the functions are too long, covering several pages. Splitting them up into helper functions would help.

Sample 0/C

Rated: Very low maintainability

There is no encapsulation. The protected or private keywords are not used, leaving everything public. This code set includes only one class, which is extremely large, and there are many public variables. Many of the functions appear to be helper functions and have no need to be exposed to the outside. There is no use of `const` functions, making the whole class construct brittle. The class itself is far too large. It should be redesigned into several classes. The one class is also not designed for extension. There is a mix of procedural and object-oriented design. There are also many global variables, mostly flags that are set according to command line arguments passed to the program. An object-oriented solution would use a singleton for this. Instead of writing member functions to evaluate or reset an object, macros accessing the public variables are used. The header file has some sparse comments on the functions and variables, but the implementation file has almost no commenting. The functions and their parameters are not commented. The use of asserts and similar debug techniques are non-existent.

Sample 4/A

Rated: High maintainability

This set is exceptional compared to the other sets. It is extremely well commented, has a clear and consistent programming style, appears very well made, with a logical and

complete interface. The class is fully encapsulated. It has high cohesion and low coupling. The one major problem with the code is the somewhat extensive use of conditional compilation with `ifdefs,` but this is necessary to enable/disable the various debugging functions as well as enable/disable various other features for portability and compactness. Another surprising design is that the class is divided into a public and a private class. That is, this is a class as part of a library. The private class is not exported and cannot be used by the library users. The private class is a helper class. It's simple, almost to the point of being a C `struct`. This particular class is not very well documented. The class is multiple inherited, but a study of the two base classes (not included in the set because it would make the set too large) reveals that it's a reasonable class design. The use of virtual and protected indicates a design towards extendibility. The class uses `const` functions extensively, which should increase the maintainability.

4.3. Establishing Maintainability of the Sample Set

Research assistants were given the sample sets and a maintainability survey (see Appendix C) for each set. The results from each survey were then added together. This value represents the maintainability of the sample (source code) seen from a developer's viewpoint.

4.4. Metrics Analysis

The *m3* tool provides a large set of function level, object level, container level, and file level metrics for object oriented C++ programs or code modules. To identify a reasonable set metrics that can be provided by metrics tools that may be good predictors of object oriented software maintainability, the number of metrics was reduce to a subset that are

good predictors of the variance in all of the metrics. To reduce the number of metrics for modeling, principal component analysis was used.

Principal component analysis (PCA) is a method to analyze multivariate data. Often when a multivariate data set is analyzed, one notices that there is a marked correlation between variables. Thus, there exist a redundancy in the data that can be used to find outliers in the data or to reduce the number of variables in further analysis. The aim is to find a set of *M* orthogonal vectors in data space that account for as much as possible of the data's variance. Projecting the data from their original *N*-dimensional space onto the *M*-dimensional subspace spanned by these vectors then performs a dimensionality reduction that often retains most of the intrinsic information in the data. The first principal component is taken to be along the direction with the maximum variance. The second principal component is constrained to lie in the subspace perpendicular to the first. Within that subspace, it points in the direction of the maximum variance. Then, the third principal component (if any) is taken in the maximum variance direction in the subspace perpendicular to the first two, and so on [10]. The first principal component accounts for as much of the variability in the data as possible, and each succeeding component accounts for as much of the remaining variability as possible [11].

The data set is the matrices of metrics generated by *m3*. The metrics produced by the level from *m3*. In this sample there were no containers. Principal component analysis was preformed on file, function and object level metrics.

The file module metrics are discarded from consideration, as the principal component analysis showed that only the size was significant and this is covered by the size metrics in both the object- and function-modules. The object and function module principal component analysis is shown in Appendix E. The following metrics were identified as the best candidate metrics to predict the data gathered from a source code:

Object:

- Total lines of code
- Average lines of code
- Total comment lines
- Number of member functions
- Number of members added
- Data members

Function:

- Halstead's effort
- Information flow metric (IFM)
- Nonblank lines of code
- Total lines of code
- Average lines of code

The function level metrics were consistent with Hagemeister and Oman [Oman, 92] [Oman, 93] [Oman, 94]. We chose to not investigate complexity and function comment

even though those had previously been good indicators for procedural C systems. With a candidate set of indicators identified, the next step is to create and evaluate regression models with those metrics against the maintainability survey results.

4.5. Maintainability Index Candidate Models

4.5.1. Single-Metric Models

Microsoft Excel was used for the linear regression analysis, which is documented as follows [Microsoft Excel documentation on “Regression analysis tool”]:

> “This analysis tool performs linear regression analysis by using the "least squares" method to fit a line through a set of observations.”

The regression lines in the graphs are linear, but they do not appear that way since the x-axis values are not uniformly distributed.

Total LOC for Objects

The total lines of code for an object do not appear to be a good maintainability predicator. The R^2 value is just 0.0070, from Table 6, indicating no correlation between the total lines of code and the estimates from the maintainability survey. The total lines of code for an object are just function declarations and variables and unless the class is exceptionally large or exceptionally small, the variations of the headers were too small to be a significantly correlated to the maintainability.

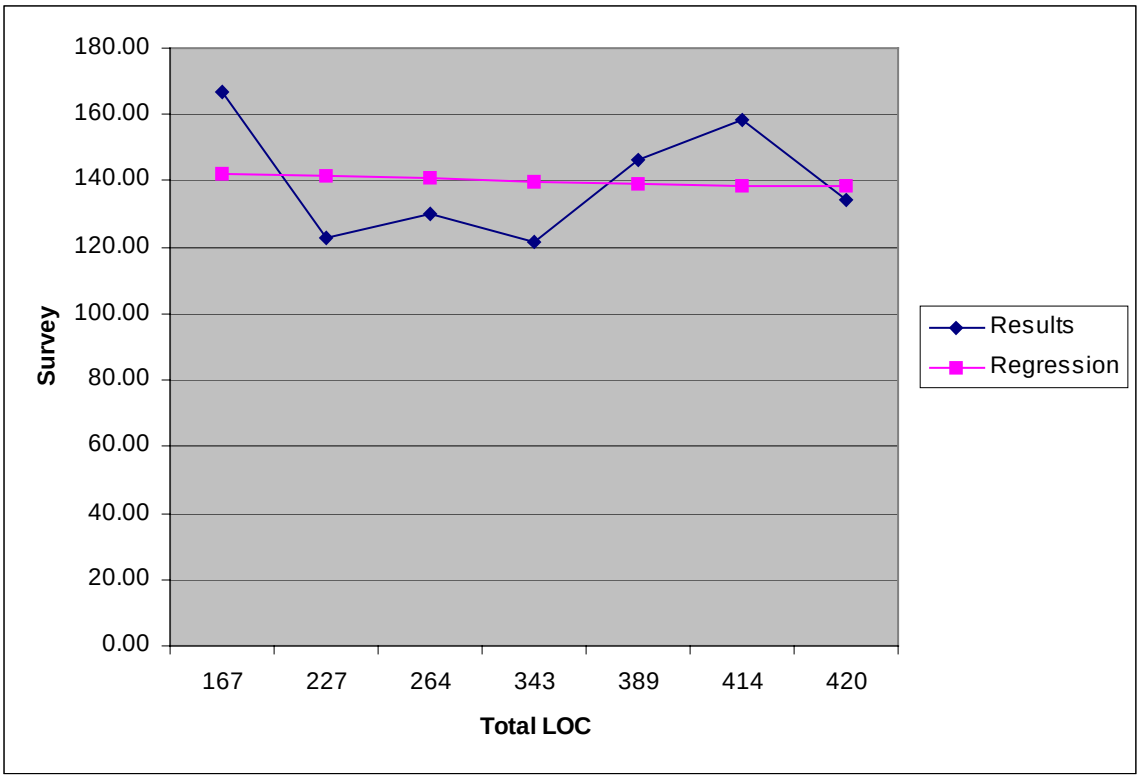

Figure 18 Results and linear regression for total lines of code for objects.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.0835
R Square	0.0070
Adjusted R Square	-0.1916
Standard Error	19.2631
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	139.6015	-18.1015
2	140.7722	-10.7722
3	141.3205	-18.3205
4	138.4604	-4.4604
5	138.9198	7.0802
6	138.5493	19.7840
7	142.2097	24.7903

Table 6 Analysis #1 for total lines of code for objects.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	144.6845	26.1692	5.5288	0.0027	77.4145	211.9546
Total LOC	-0.0148	0.0791	-0.1873	0.8588	-0.2182	0.1886

Table 7 Analysis #2 for total lines of code for objects.

Average LOC for Objects

The average lines of code for objects appear to be a good maintainability predicator. The R^2 value is 0.7073, from Table 8, indicating a significant correlation between average lines of code for objects and the estimates from the maintainability survey. As can be seen from Figure 19, there is a good match between the linear regression and the maintainability survey results.

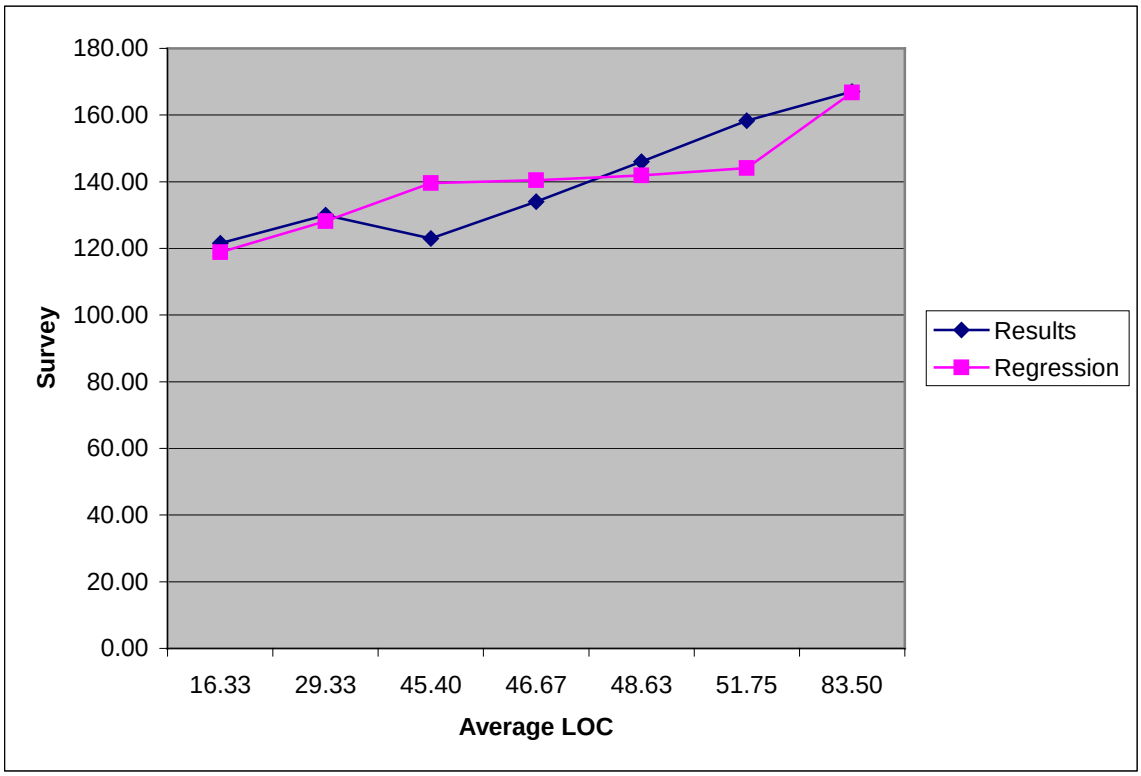

Figure 19 Results and linear regression for average lines of code for objects.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.8410
R Square	0.7073
Adjusted R Square	0.6488
Standard Error	10.4580
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	118.8856	2.6144
2	128.1450	1.8550
3	139.5887	-16.5887
4	140.4909	-6.4909
5	141.8857	4.1143
6	144.1116	14.2218
7	166.7259	0.2741

Table 8 Analysis #1 for average lines of code for objects.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	107.2519	10.2104	10.5042	0.0001	81.0052	133.4986
Avg LOC	0.7123	0.2049	3.4760	0.0177	0.1855	1.2390

Table 9 Analysis #2 for average lines of code for objects.

Comment Lines for Objects

The comment lines for an object do not appear to be a good maintainability predicator. The R^2 value is just 0.0896, from Table 10, indicating little correlation between comment lines for objects and the estimates from the maintainability survey. At first, this may seem surprising, but it doesn't help maintainability of the entire source code if just a few of the classes are commented. It is better to comment a little on all the classes, as can be seen from the Average Comment Lines results below.

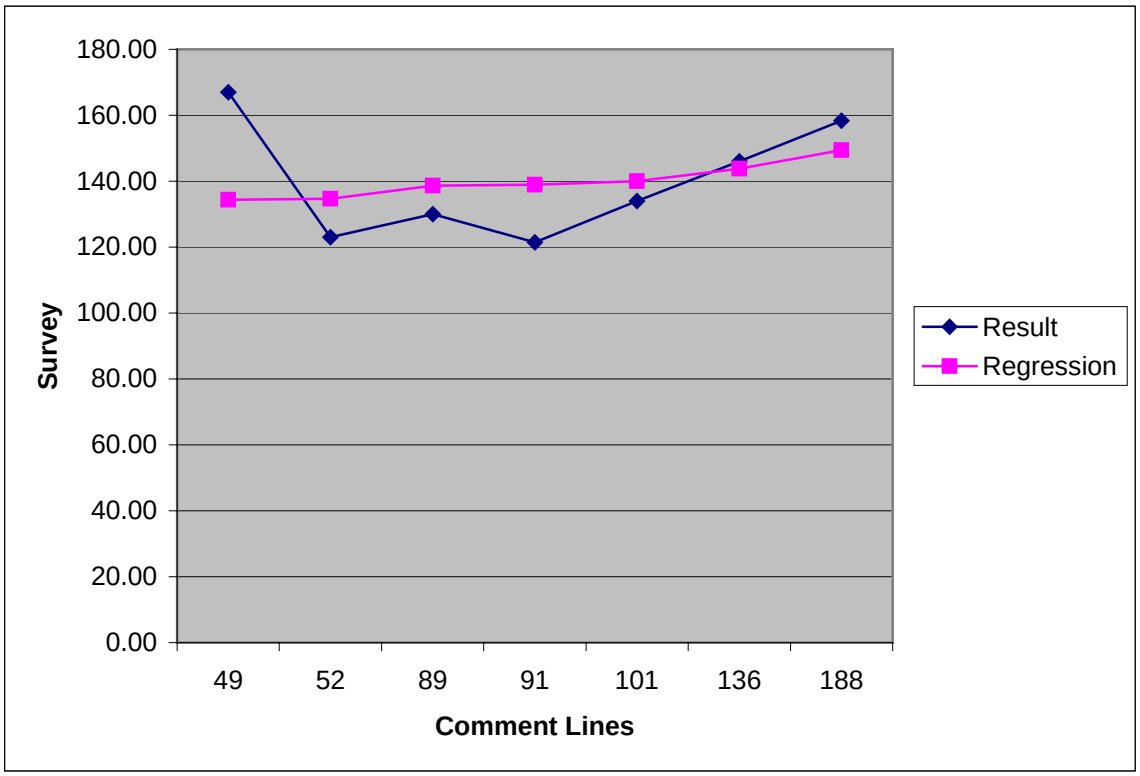

Figure 20 Results and linear regression for comment lines for objects.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.2994
R Square	0.0896
Adjusted R Square	-0.0924
Standard Error	18.4439
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	138.9036	-17.4036
2	138.6859	-8.6859
3	134.6597	-11.6597
4	139.9917	-5.9917
5	143.8003	2.1997
6	149.4588	8.8746
7	134.3333	32.6667

Table 10 Analysis #1 for comment lines for objects.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	129.0013	17.1251	7.5329	0.0007	84.9799	173.0226
Comment Lines	0.1088	0.1551	0.7016	0.5142	-0.2899	0.5075

Table 11 Analysis #2 for comment lines for objects.

Average Comment Lines for Objects

The average comment lines for an object appear to be a good maintainability predicator. The R^2 value is 0.9437, from Table 12, indicating a close correlation between the average comment lines for objects and the estimates from the maintainability survey. As can be seen from Figure 21, there is a close match between the linear regression and the results from the maintainability survey. Not surprisingly, consistently commenting all the classes increases the maintainability.

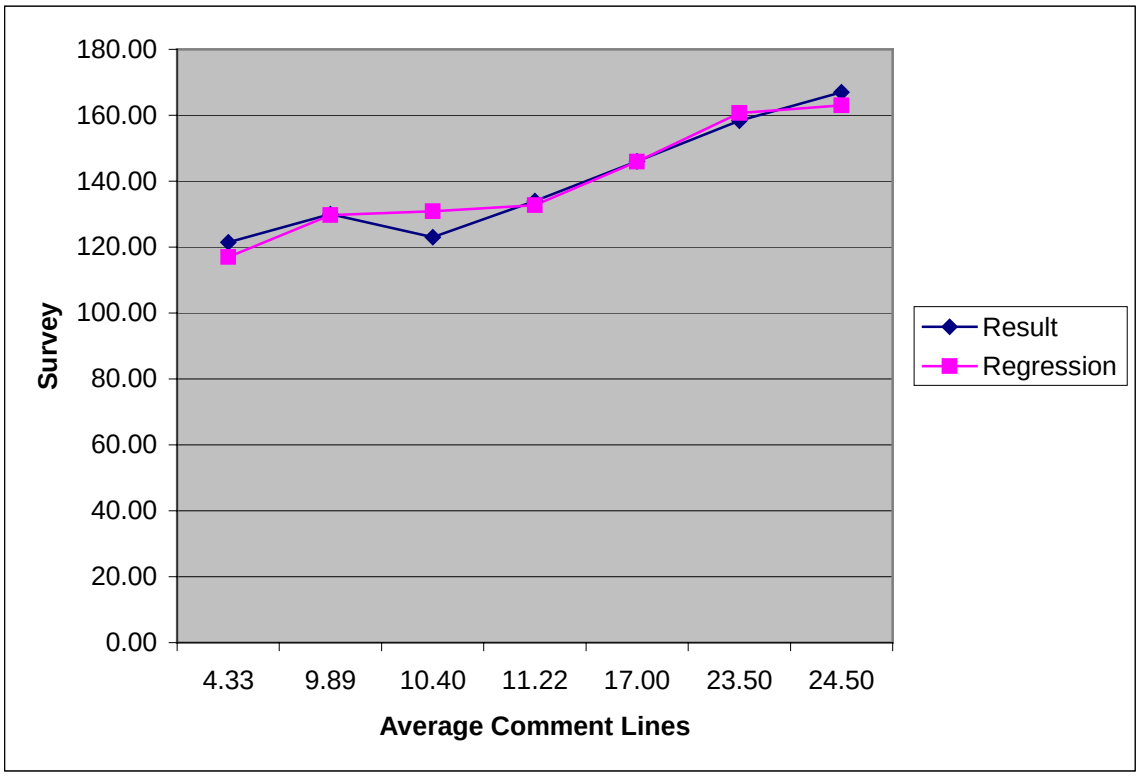

Figure 21 Results and linear regression for average comment lines for objects.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.9715
R Square	0.9437
Adjusted R Square	0.9325
Standard Error	4.5858
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	117.0159	4.4841
2	129.6792	0.3208
3	130.8442	-7.8442
4	132.7183	1.2817
5	145.8881	0.1119
6	160.7041	-2.3708
7	162.9835	4.0165

Table 12 Analysis #1 for average comment lines for objects.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	107.1386	3.9831	26.8981	0.0000	96.8996	117.3775
Avg. Comment Lines	2.2794	0.2489	9.1566	0.0003	1.6395	2.9193

Table 13 Analysis #2 for average comment lines for objects.

NMA for Objects

The number of members added to an object does not appear to be a good maintainability predicator. The R^2 value is just 0.0660, from Table 14, indicating no correlation between NMA and the estimates from the maintainability survey. This contradicts intuition, but the reason for this result may be related to the fairly small source code samples. This is also a metric that is dependent on a hierarchy, but the small source code set did not contain much inheritance.

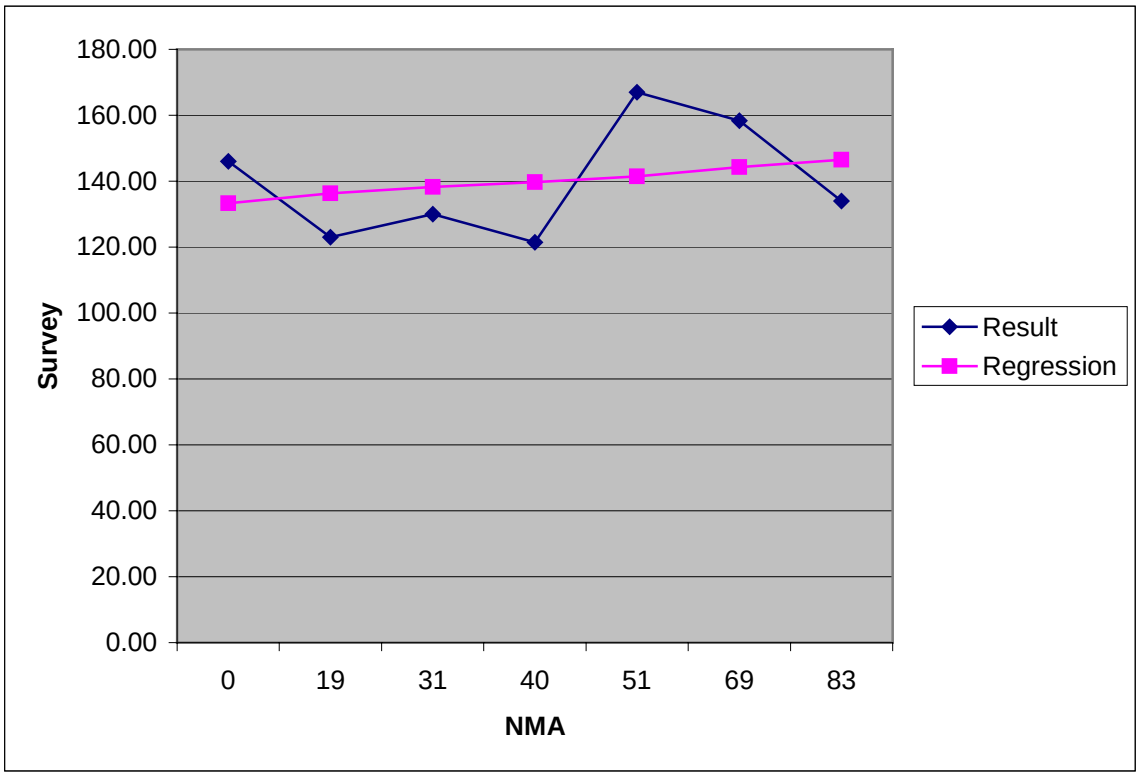

Figure 22 Results and linear regression for number of methods added.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.2569
R Square	0.0660
Adjusted R Square	-0.1208
Standard Error	18.6819
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	139.6819	-18.1819
2	138.2555	-8.2555
3	136.3537	-13.3537
4	146.4967	-12.4967
5	133.3425	12.6575
6	144.2779	14.0554
7	141.4252	25.5748

Table 14 Analysis #1 for number of methods added.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	133.3425	13.2070	10.0964	0.0002	99.3929	167.2921
NMA	0.1585	0.2666	0.5944	0.5781	-0.5269	0.8439

Table 15 Analysis #2 for number of methods added.

Member Functions for Objects

The number of member functions for an object does not appear to be a good maintainability predicator. The R^2 value is just 0.0622, from Table 16, indicating no correlation between the member functions for objects and the estimates from the maintainability survey. This is contradicts intuition, where a large number of functions in a class would intuitively make it less maintainable. As with the results of the NMA metric, this may have something to do with the sample source code set.

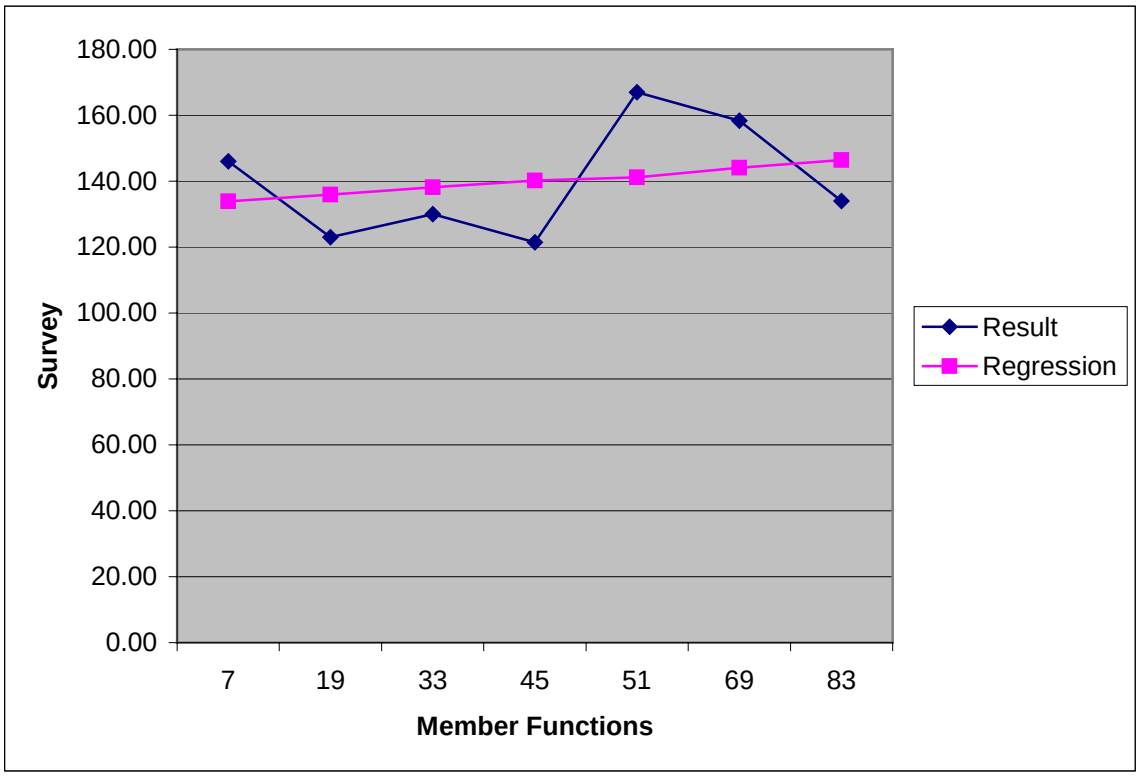

Figure 23 Results and linear regression for member functions for objects.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.2494
R Square	0.0622
Adjusted R Square	-0.1254
Standard Error	18.7197
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	140.1638	-18.6638
2	138.1935	-8.1935
3	135.8948	-12.8948
4	146.4031	-12.4031
5	133.9245	12.0755
6	144.1045	14.2289
7	141.1490	25.8510

Table 16 Analysis #1 for member functions.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	132.7752	14.3672	9.2415	0.0002	95.8431	169.7073
Member functions	0.1642	0.2851	0.5759	0.5896	-0.5687	0.8971

Table 17 Analysis #2 for member functions.

Data Members for Objects

The number of data members for an object does not appear to be a good maintainability predicator. The R^2 value is just 0.1131, from Table 18, indicating little correlation between the data members for objects and the estimates from the maintainability survey. Like the results for member functions, one would expect the number of data members to affect maintainability. However, if the class has a very high cohesion, not currently measured in *m3*, the number of data members is likely to matter less than with a low cohesion.

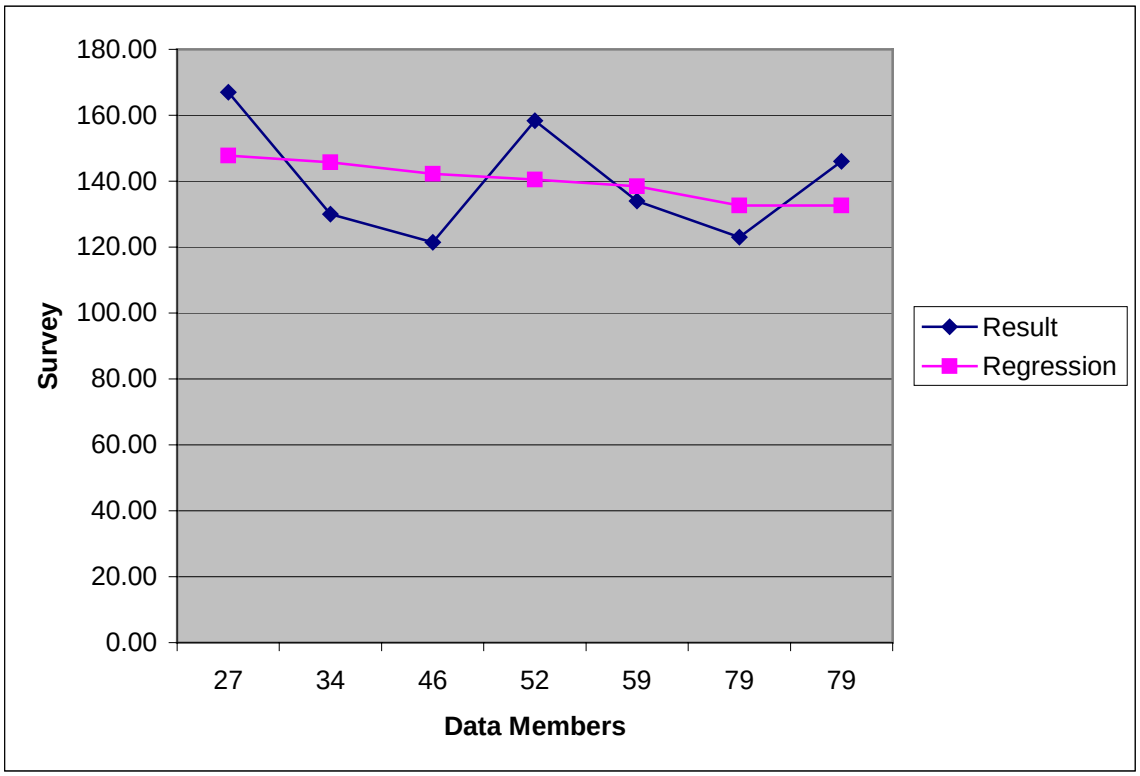

Figure 24 Results and linear regression for data members for objects.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.3363
R Square	0.1131
Adjusted R Square	-0.0643
Standard Error	18.2045
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	142.2321	-20.7321
2	145.7412	-15.7412
3	132.5819	-9.5819
4	138.4305	-4.4305
5	132.5819	13.4181
6	140.4775	17.8558
7	147.7882	19.2118

Table 18 Analysis #1 for data members.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	155.6837	20.8383	7.4710	0.0007	102.1173	209.2501
Data members	-0.2924	0.3662	-0.7986	0.4608	-1.2337	0.6489

Table 19 Analysis #2 for data members.

Summary for Objects

Object Metric	R^2 value
Average Comment Lines	0.9437
Average LOC	0.7073
Data Members	0.1131
Comment Lines	0.0896
NMA	0.0660
Member Functions	0.0622
Total LOC	0.0070

Table 20 The R^2 values for the one metric, object models.

Total LOC for Functions

The total lines of code for functions do not appear to be a good maintainability predicator. The R^2 value is just 0.0025, from Table 21, indicating no correlation between the total lines of code for functions and the estimates from the maintainability survey. The linear regression line as shown in Figure 25 has a very small slope, in fact, just -0.0024 as shown in Table 22. A large amount of code, but with well-defined functions can be well maintainable. The problems arise when the size of the functions grow large. Hence, the average number of lines of code per function matters, which is reflected by the average lines of code for functions results below.

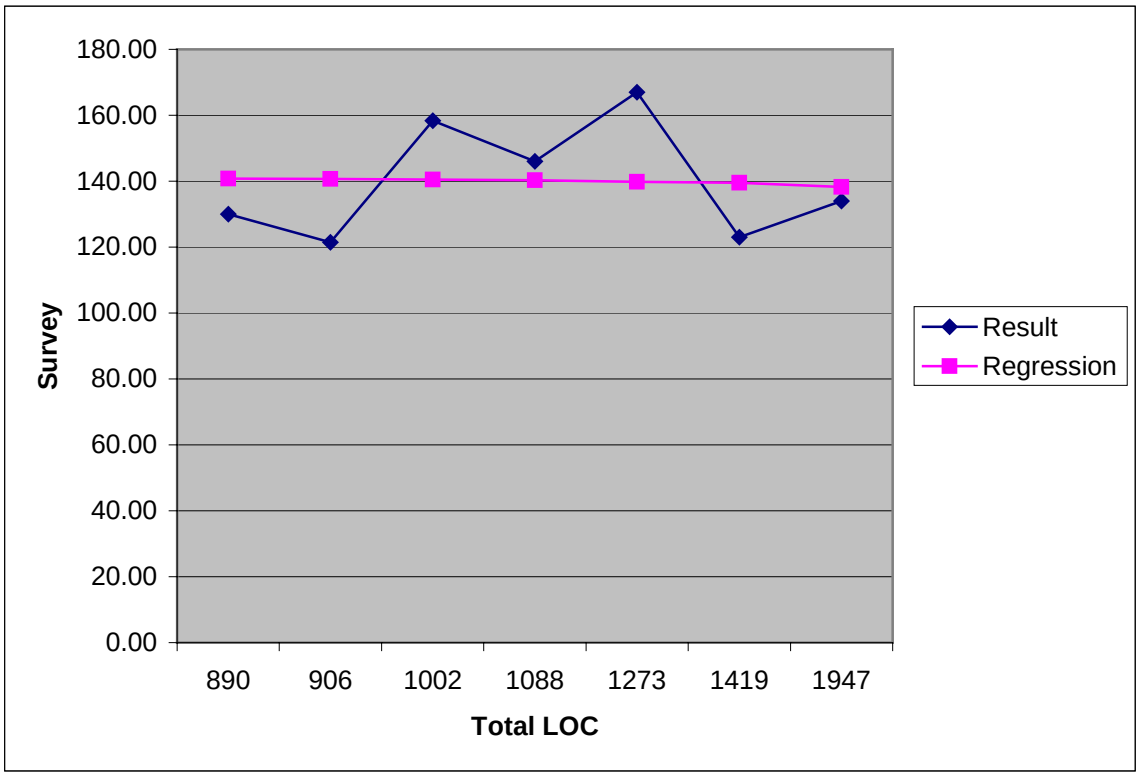

Figure 25 Results and linear regression for total lines of code for functions.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.0502
R Square	0.0025
Adjusted R Square	-0.1970
Standard Error	19.3062
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	140.7135	-19.2135
2	140.7514	-10.7514
3	139.5006	-16.5006
4	138.2522	-4.2522
5	140.2832	5.7168
6	140.4866	17.8468
7	139.8458	27.1542

Table 21 Analysis #1 for total lines of code for functions.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	142.8557	26.6224	5.3660	0.0030	74.4207	211.2907
Total LOC	-0.0024	0.0210	-0.1125	0.9148	-0.0564	0.0517

Table 22 Analysis #2 for total lines of code for functions.

Average LOC for Functions

The average lines of code for functions appear to be a maintainability predicator, although not a good one. The R^2 value is 0.4787, from Table 23, indicating a correlation between the average lines of code for functions and the estimates from the maintainability survey. As can be seen from Figure 26 below, there is a close match between the linear regression and the maintainability survey results, but the one peak in the middle of the survey results destroy some of the correlation.

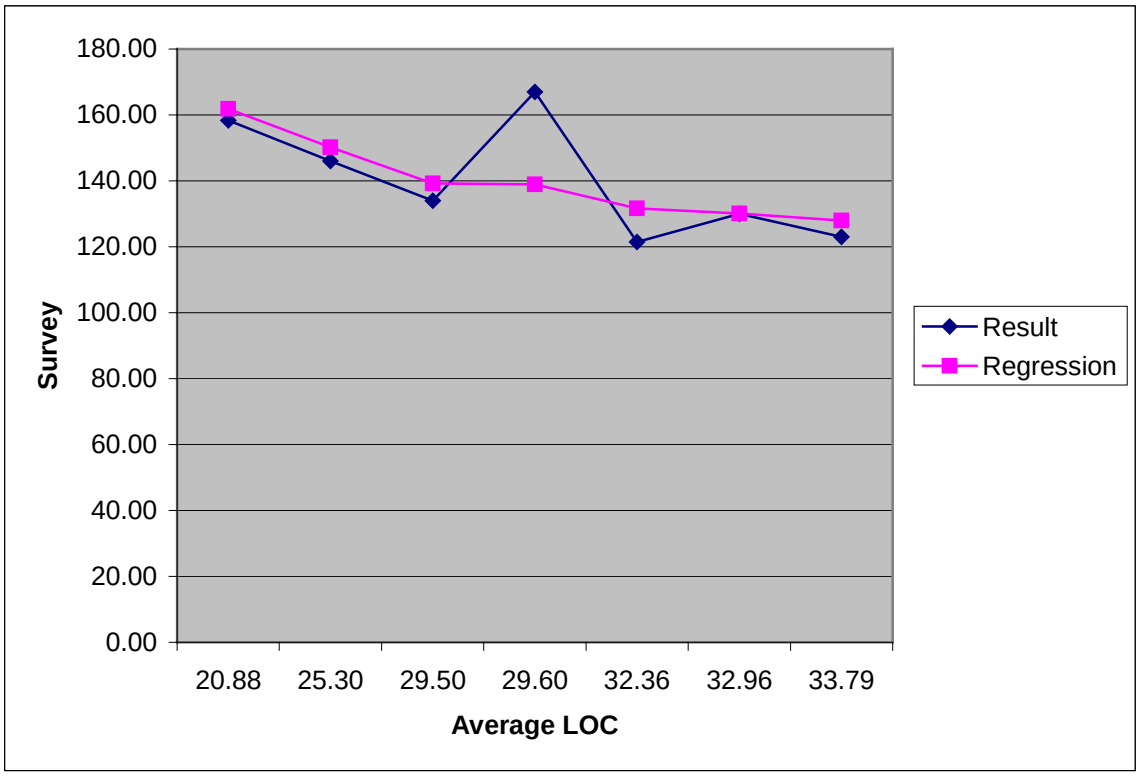

Figure 26 Results and linear regression for average lines of code for functions.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.6919
R Square	0.4787
Adjusted R Square	0.3745
Standard Error	13.9562
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	131.6872	-10.1872
2	130.0975	-0.0975
3	127.9386	-4.9386
4	139.1844	-5.1844
5	150.1992	-4.1992
6	161.8166	-3.4832
7	138.9098	28.0902

Table 23 Analysis #1 for average lines of code for functions.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	216.5930	36.1397	5.9932	0.0019	123.6931	309.4929
Avg LOC	-2.6240	1.2245	-2.1430	0.0850	-5.7716	0.5236

Table 24 Analysis #2 for average lines of code for functions.

Effort for Functions

The effort for functions does not appear to be a good maintainability predicator. The R^2 value is 0.2585, from Table 25, indicating some correlation between the effort for functions and the estimates from the maintainability survey. Overall, the effort is not a significant maintainability predicator. It does matter, however, for each individual function, as we can see from average effort for functions results below.

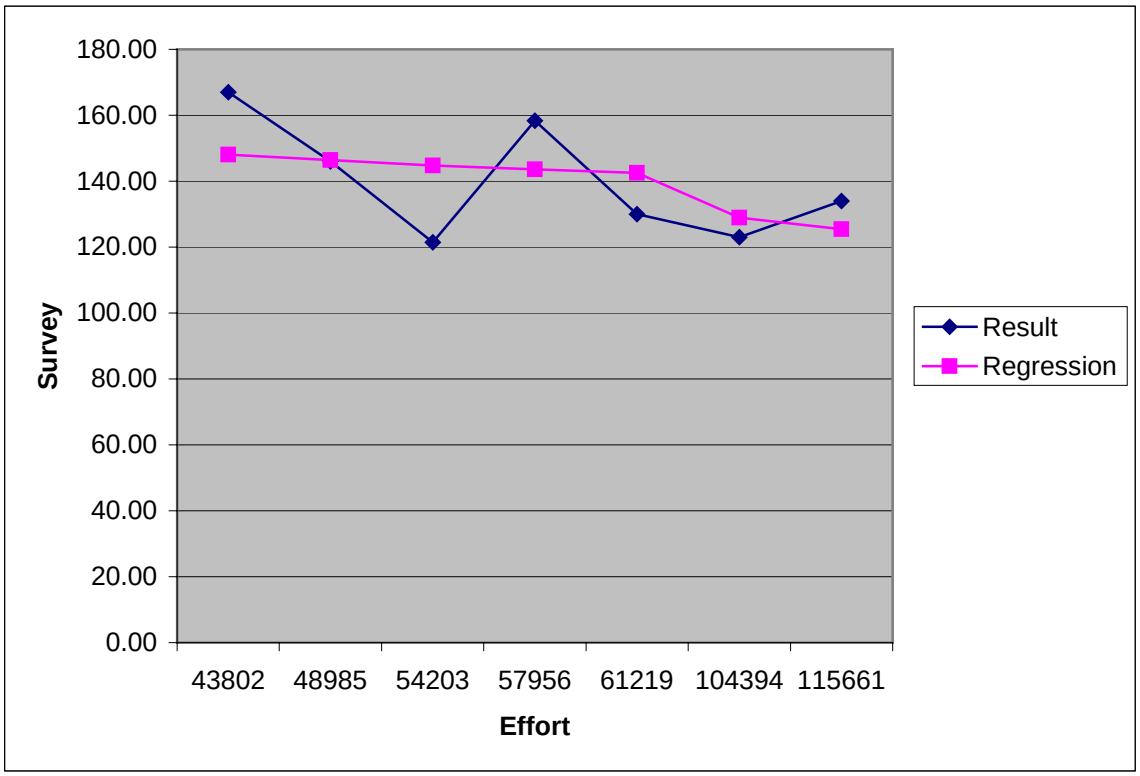

Figure 27 Results and linear regression for effort for functions.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.5084
R Square	0.2585
Adjusted R Square	0.1102
Standard Error	16.6456
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	144.7831	-23.2831
2	142.5726	-12.5726
3	128.9698	-5.9698
4	125.4200	8.5800
5	146.4271	-0.4271
6	143.6007	14.7327
7	148.0601	18.9399

Table 25 Analysis #1 for effort for functions.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	161.8605	17.7295	9.1295	0.0003	116.2855	207.4355
Effort	-0.0003	0.0002	-1.3203	0.2440	-0.0009	0.0003

Table 26 Analysis #2 for effort for functions.

Average Effort for Functions

The average effort for functions appears to be a good maintainability predicator. The R^2 value is 0.7880, from Table 27, indicating a significant correlation between the average effort for functions and the estimates from the maintainability survey. As can be seen from Figure 28, there is a close match between the linear regression and the estimates from the maintainability survey.

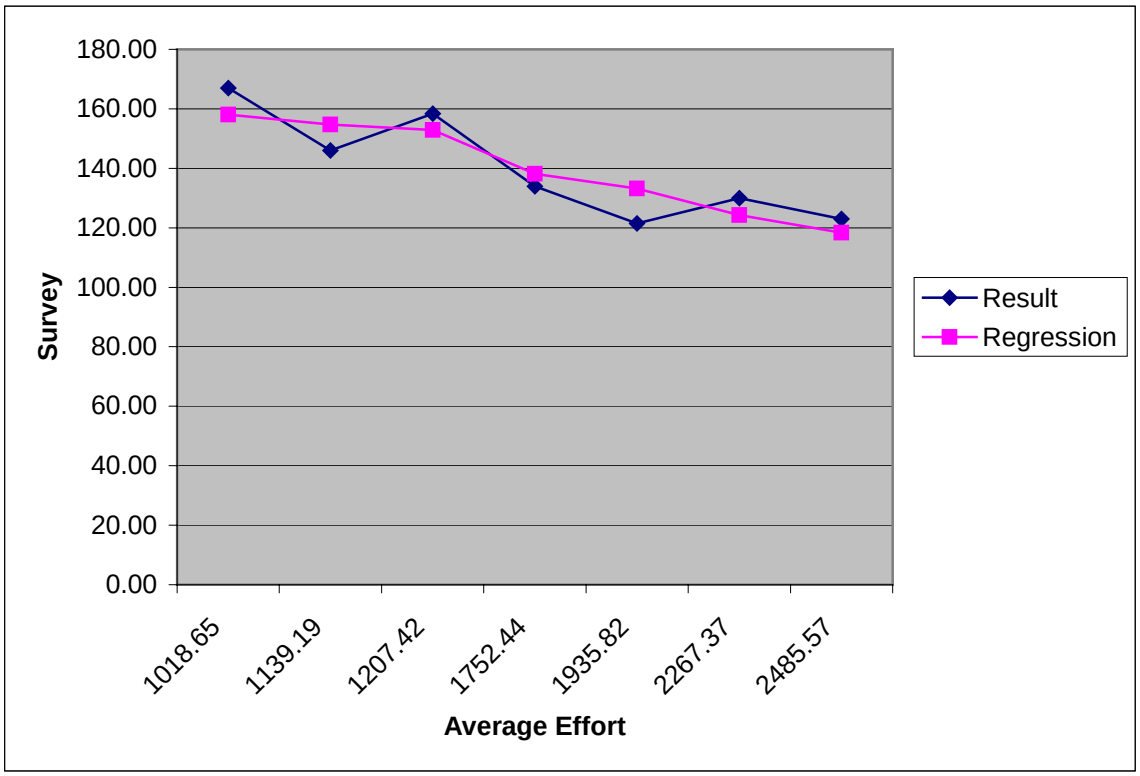

Figure 28 Results and linear regression for average effort for functions.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.8877
R Square	0.7880
Adjusted R Square	0.7456
Standard Error	8.9010
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	133.2449	-11.7449
2	124.2887	5.7113
3	118.3944	4.6056
4	138.1986	-4.1986
5	154.7646	-8.7646
6	152.9215	5.4119
7	158.0206	8.9794

Table 27 Analysis #1 for average effort for functions.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	185.5377	11.0919	16.7273	0.0000	157.0251	214.0502
Avg Effort	-0.0270	0.0063	-4.3107	0.0076	-0.0431	-0.0109

Table 28 Analysis #2 for average effort for functions.

IFM for Functions

The information flow metric for functions does not appear to be a good maintainability predicator. The R^2 value is 0.1830, from Table 29, indicating little correlation between the IFM for functions and the estimates from the maintainability survey. The IFM metric showed up in the principal component analysis, but a linear regression analysis reveals that it is not a good maintainability predicator. The reason is probably because the values for IFM vary enormously between the sample sets, with an unpredictable pattern. The reason why this metric showed up is probably because of the stand-alone functions (i.e. typical C code).

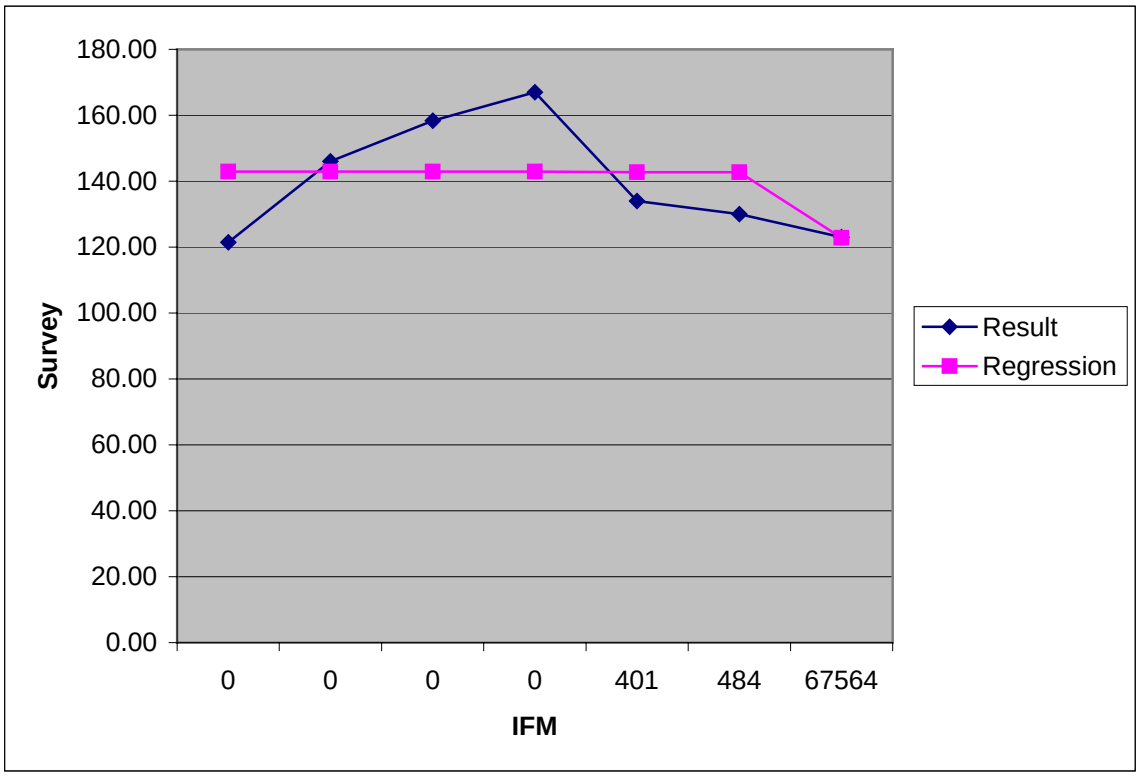

Figure 29 Results and linear regression for information flow metric for functions.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.4278
R Square	0.1830
Adjusted R Square	0.0196
Standard Error	17.4724
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	142.8731	-21.3731
2	142.7297	-12.7297
3	122.8569	0.1431
4	142.7543	-8.7543
5	142.8731	3.1269
6	142.8731	15.4602
7	142.8731	24.1269

Table 29 Analysis #1 for information flow metric for functions.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	142.8731	7.1487	19.9858	0.0000	124.4967	161.2495
IFM	-0.0003	0.0003	-1.0583	0.3383	-0.0010	0.0004

Table 30 Analysis #2 for information flow metric for functions.

Nonblank LOC for Functions

The number of nonblank lines of code for function does not appear to be a good maintainability predicator. The R^2 value is 0.0000103898703609658, indicating no correlation between the nonblank lines of code for functions and the estimates from the maintainability survey. As can be seen from Figure 30 the slope of the linear regression is very small, just 0.0002 as can be seen from Table 32.

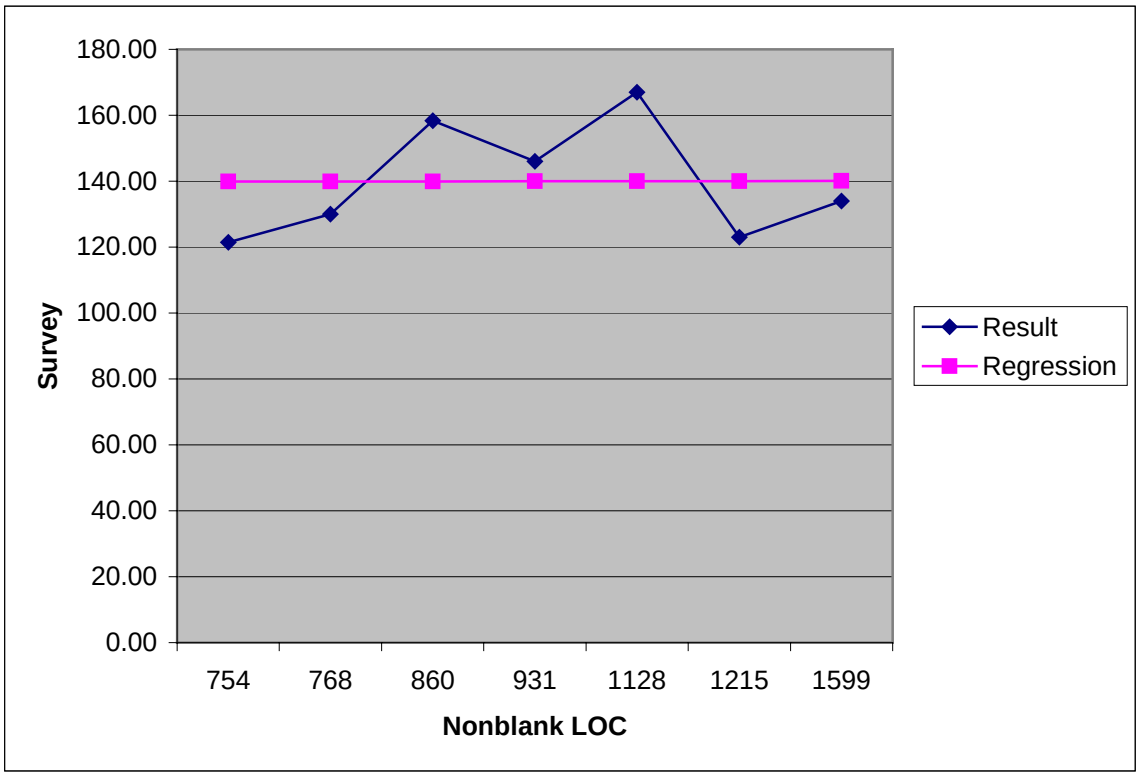

Figure 30 Results and linear regression for nonblank lines of code for functions.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.0032
R Square	0.0000
Adjusted R Square	-0.2000
Standard Error	19.3305
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	139.9232	-18.4232
2	139.9258	-9.9258
3	140.0097	-17.0097
4	140.0818	-6.0818
5	139.9564	6.0436
6	139.9431	18.3903
7	139.9934	27.0066

Table 31 Analysis #1 for nonblank lines of code for functions.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	139.7816	27.9658	4.9983	0.0041	67.8933	211.6700
Nb LOC	0.0002	0.0260	0.0072	0.9945	-0.0668	0.0671

Table 32 Analysis #2 for nonblank lines of code for functions.

Average Nonblank LOC for Functions

The average nonblank lines of code for functions do not appear to be a good maintainability predictor. The R^2 value is 0.3512, from Table 33, indicating some correlation between average nonblank lines of code for functions and the estimates from the maintainability survey.

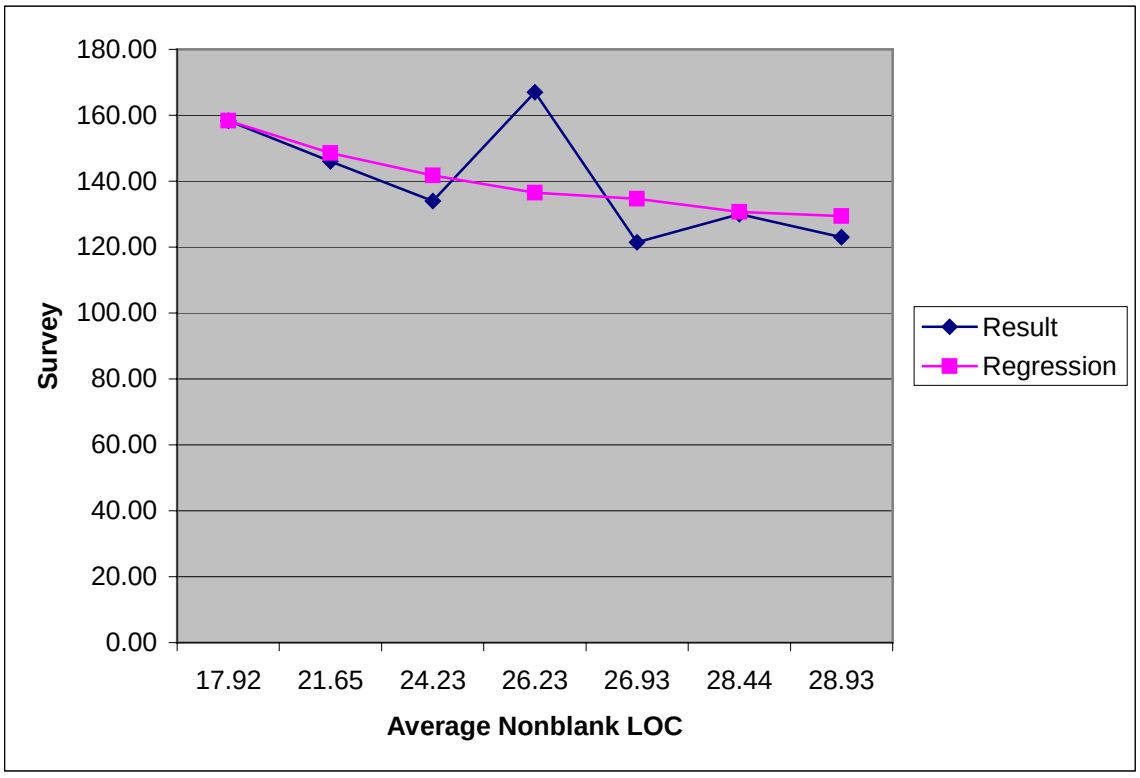

Figure 31 Results and linear regression for avg. nonblank lines of code for functions.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.5926
R Square	0.3512
Adjusted R Square	0.2215
Standard Error	15.5702
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	134.6405	-13.1405
2	130.6451	-0.6451
3	129.3690	-6.3690
4	141.7603	-7.7603
5	148.5502	-2.5502
6	158.3933	-0.0599
7	136.4750	30.5250

Table 33 Analysis #1 for average nonblank lines of code for functions.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	205.6165	40.3295	5.0984	0.0038	101.9465	309.2866
Avg Nb LOC	-2.6357	1.6021	-1.6452	0.1608	-6.7539	1.4825

Table 34 Analysis #2 for average nonblank lines of code for functions.

Summary for Functions

Function Metric	R^2 value
Average Effort	0.7880
Average LOC	0.4787
Average Nonblank LOC	0.3512
Effort	0.2585
IFM	0.1830
Total LOC	0.0025
Nonblank LOC	0.0000

Table 35 The R^2 values for the one metric, function models.

4.5.2. Two-Metric Models

Object Average Comment Lines & Function Average Effort

The object average comment lines and the function average effort appear to be a good maintainability predicator. The R^2 value is 0.9754, from Table 36, indicating a close correlation between the model and the estimates from the maintainability survey. This is a very good maintainability predicator, the best between the two-metric models. Commented, low-effort functions are a good way to make maintainable code.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.9876
R Square	0.9754
Adjusted R Square	0.9632
Standard Error	3.3871
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	120.4510	1.0490
2	126.9355	3.0645
3	125.8174	-2.8174
4	133.9226	0.0774
5	149.4186	-3.4186
6	159.9257	-1.5924
7	163.3625	3.6375

Table 36 Analysis #1 for object average comment lines and function average effort.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	130.7148	10.7824	12.1230	0.0003	100.7780	160.6516
Obj. Avg Com. Lines	1.7124	0.3099	5.5254	0.0052	0.8519	2.5728
Func. Avg Eff	-0.0091	0.0040	-2.2728	0.0855	-0.0203	0.0020

Table 37 Analysis #2 for object average comment lines and function average effort.

Object Average Comment Lines & Function Average LOC

The object average comment lines and function average lines of code appear to be a good maintainability predicator. The R^2 value is 0.9437, from Table 38, indicating a close correlation between the model and the estimates from the maintainability survey. This is a very good maintainability predicator, the second best between the two-metric models. Commented, reasonably sized functions are a good way to make maintainable code.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.9715
R Square	0.9437
Adjusted R Square	0.9156
Standard Error	5.1269
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	117.0017	4.4983
2	129.6987	0.3013
3	130.8753	-7.8753
4	132.7061	1.2939
5	145.8578	0.1422
6	160.6566	-2.3232
7	163.0372	3.9628

Table 38 Analysis #1 for object avg. comment lines and function avg. lines of code.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	106.7460	23.2669	4.5879	0.0101	42.1467	171.3453
Obj. Avg Com. Lines	2.2843	0.3973	5.7489	0.0045	1.1811	3.3874
Func. Avg LOC	0.0110	0.6422	0.0172	0.9871	-1.7720	1.7941

Table 39 Analysis #2 for object avg. comment lines and function avg. lines of code.

Object Member Functions & Function Average Effort

The object member functions and the function average effort appear to be a maintainability predicator. The R^2 value is 0.7917, from Table 40, indicating a correlation between the model and the estimates from the maintainability survey. The result is not as good as some of the other two metric models, but still significant. A reasonable amount of low-effort functions help maintainability, but a higher number of member functions is acceptable as long as they are commented.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.8898
R Square	0.7917
Adjusted R Square	0.6875
Standard Error	9.8640
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	133.3931	-11.8931
2	124.0786	5.9214
3	117.6978	5.3022
4	139.8333	-5.8333
5	153.0282	-7.0282
6	153.7597	4.5736
7	158.0428	8.9572

Table 40 Analysis #1 for object member functions and function average effort.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	183.0510	15.4195	11.8714	0.0003	140.2396	225.8624
Obj. Mem. Func.	0.0411	0.1538	0.2671	0.8026	-0.3859	0.4681
Func. Avg Eff	-0.0266	0.0071	-3.7427	0.0201	-0.0463	-0.0069

Table 41 Analysis #2 for object member functions and function average effort.

Object Member Functions & Function Average LOC

The object member functions and the function average lines of code appear to be a maintainability predicator. The R^2 value is 0.4829, from Table 43, indicating some correlation between the model and the estimates from the maintainability survey. It is not a good predicator, but still significant. The number of member functions and their average size is not as significant to maintainability as making them commented and keeping the effort low.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.6949
R Square	0.4829
Adjusted R Square	0.2244
Standard Error	15.5406
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	131.9558	-10.4558
2	129.8761	0.1239
3	127.1536	-4.1536
4	140.9398	-6.9398
5	148.2971	-2.2971
6	162.3565	-4.0232
7	139.2544	27.7456

Table 42 Analysis #1 for object member functions and function average lines of code.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	212.6349	45.8431	4.6383	0.0097	85.3538	339.9161
Obj. Mem. Func.	0.0443	0.2458	0.1803	0.8657	-0.6383	0.7269
Func. Avg LOC	-2.5550	1.4162	-1.8042	0.1455	-6.4870	1.3770

Table 43 Analysis #2 for object member functions and function average lines of code.

Object Average LOC & Function Average Effort

The object average lines of code and the function average effort appear to be a good maintainability predicator. The R^2 value is 0.9115, from Table 44, indicating a close correlation between the model and the estimates from the maintainability survey.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.9547
R Square	0.9115
Adjusted R Square	0.8672
Standard Error	6.4307
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	123.9630	-2.4630
2	123.0604	6.9396
3	125.3912	-2.3912
4	139.0738	-5.0738
5	150.8690	-4.8690
6	150.8584	7.4750
7	166.6176	0.3824

Table 44 Analysis #1 for object average lines of code and function average effort.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	152.4291	16.1458	9.4408	0.0007	107.6010	197.2572
Obj. Avg LOC	0.3894	0.1649	2.3621	0.0775	-0.0683	0.8471
Func. Avg Eff	-0.0180	0.0059	-3.0371	0.0385	-0.0344	-0.0015

Table 45 Analysis #2 for object average lines of code and function average effort.

Object Average LOC & Function Average LOC

The object average lines of code and function average lines of code appear to be a maintainability predicator. The R^2 value is 0.8893, from Table 46, indicating a significant correlation between the model and the estimates from the maintainability survey, but the correlation is not as good as some of the other two-metric models.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.9430
R Square	0.8893
Adjusted R Square	0.8339
Standard Error	7.1912
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	117.3946	4.1054
2	123.8713	6.1287
3	131.7481	-8.7481
4	139.8738	-5.8738
5	148.2478	-2.2478
6	157.6929	0.6404
7	161.0048	5.9952

Table 46 Analysis #1 for object avg. lines of code and function avg. lines of code.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	163.7597	23.1291	7.0803	0.0021	99.5429	227.9764
Obj. Avg LOC	0.5786	0.1502	3.8513	0.0183	0.1615	0.9957
Func. Avg LOC	-1.7250	0.6727	-2.5641	0.0624	-3.5928	0.1428

Table 47 Analysis #2 for object avg. lines of code and function avg. lines of code.

Summary for Two-Metric Models

Object \ Func	Average Effort	Average LOC
Average Comment Lines	0.9754	0.9437
Member Functions	0.7917	0.4829
Average LOC	0.9115	0.8893

Table 48 The R^2 values for the two-metric models.

4.5.3. Three-Metric Models

The metric common for all of the three metric models is average comment lines (object).

Object Member Functions & Function Average Effort

The average comment lines for an object, the number of object member functions and the function average effort appear to be a good maintainability predicator. The R^2 value is 0.9785, from Table 49, indicating a close correlation between the model and the estimates from the maintainability survey.

A commented class with a reasonable number of member functions with, on average, a low effort makes for maintainable code. This is consistent with our intuition.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.9892
R Square	0.9785
Adjusted R Square	0.9570
Standard Error	3.6607
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	120.6070	0.8930
2	126.7408	3.2592
3	125.1742	-2.1742
4	135.4096	-1.4096
5	147.8560	-1.8560
6	160.6725	-2.3392
7	163.3734	3.6266

Table 49 Analysis #1 for object average comment lines, object member functions and function average effort.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	128.5575	12.1150	10.6115	0.0018	90.0022	167.1128
Obj. Avg Com. Lines	1.7095	0.3350	5.1032	0.0146	0.6434	2.7755
Obj. Mem. Func.	0.0372	0.0571	0.6514	0.5612	-0.1445	0.2188
Func. Avg Eff	-0.0088	0.0044	-2.0111	0.1378	-0.0227	0.0051

Table 50 Analysis #2 for object average comment lines, object member functions and function average effort.

Object Member Functions & Function Average LOC

The average comment lines for an object, the number of member functions in an object and the function average lines of code appear to be a good maintainability predictor. The R^2 value is 0.9498, from Table 51, indicating a close correlation between the model and the estimates from the maintainability survey.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.9746
R Square	0.9498
Adjusted R Square	0.8996
Standard Error	5.5914
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	117.2925	4.2075
2	129.4315	0.5685
3	129.9375	-6.9375
4	134.8034	-0.8034
5	143.5602	2.4398
6	161.3035	-2.9701
7	163.5047	3.4953

Table 51 Analysis #1 for object average comment lines, object member functions and function average lines of code.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	101.7435	26.6986	3.8108	0.0318	16.7764	186.7105
Obj. Avg Com. Lines	2.2893	0.4334	5.2820	0.0132	0.9100	3.6686
Obj. Mem. Func.	0.0533	0.0885	0.6025	0.5893	-0.2282	0.3349
Func. Avg LOC	0.0998	0.7157	0.1395	0.8979	-2.1779	2.3776

Table 52 Analysis #2 for object average comment lines, object member functions and function average lines of code.

Object Average LOC & Function Average Effort

The average comment lines for an object, the object average lines of code and the function average effort appear to be a good maintainability predicator. The R^2 value is 0.9766, from Table 53, indicating a close correlation between the model and the estimates from the maintainability survey.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.9882
R Square	0.9766
Adjusted R Square	0.9533
Standard Error	3.8151
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	120.2408	1.2592
2	126.5035	3.4965
3	126.1828	-3.1828
4	134.4531	-0.4531
5	149.3281	-3.3281
6	158.9617	-0.6284
7	164.1634	2.8366

Table 53 Analysis #1 for object average comment lines, object average lines of code and function average effort.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	130.8206	12.1479	10.7690	0.0017	92.1604	169.4807
Obj. Avg Com. Lines	1.5529	0.5369	2.8922	0.0629	-0.1558	3.2616
Obj. Avg LOC	0.0588	0.1504	0.3909	0.7220	-0.4199	0.5376
Func. Avg Eff	-0.0094	0.0046	-2.0548	0.1322	-0.0241	0.0052

Table 54 Analysis #2 for object average comment lines, object average lines of code and function average effort.

Object Average LOC & Function Average LOC

The average comment lines for an object, the object average lines of code and the function average lines of code appear to be a good maintainability predicator. The R^2 value is 0.9437, from Table 55, indicating a close correlation between the model and the estimates from the maintainability survey.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.9715
R Square	0.9437
Adjusted R Square	0.8875
Standard Error	5.9193
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	116.9785	4.5215
2	129.5983	0.4017
3	130.8962	-7.8962
4	132.8155	1.1845
5	145.8898	0.1102
6	160.6035	-2.2702
7	163.0515	3.9485

Table 55 Analysis #1 for object average comment lines, object average lines of code and function average lines of code.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	107.4974	38.1134	2.8205	0.0667	-13.7966	228.7913
Obj. Avg Com. Lines	2.2499	1.3203	1.7040	0.1869	-1.9520	6.4518
Obj. Avg LOC	0.0099	0.3559	0.0278	0.9796	-1.1228	1.1426
Func. Avg LOC	-0.0133	1.1470	-0.0116	0.9915	-3.6637	3.6371

Table 56 Analysis #2 for object average comment lines, object average lines of code and function average lines of code.

Summary for Three Metric Models

The metric common for all of the three-metric models is average comment lines (object).

Object \ Func	Average Effort	Average LOC
Member Functions	0.978482455150338	0.949799876770203
Average LOC	0.976629512787467	0.94373969513615

Table 57 The R^2 values for the three-metric models.

Five-Metric Model

Object Average Comment Lines, Object Average LOC, Function Average LOC, Object Member Functions & Function Average Effort

The object average comment lines, object average lines of code, the function average lines of code, the object member function and the function average effort appear to be a good maintainability predicator. The R^2 value is 0.9988, from Table 58, indicates a very close correlation between the model and the estimates from the maintainability survey. It is the best predicator of all the models.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.9994
R Square	0.9988
Adjusted R Square	0.9927
Standard Error	1.5091
Observations	7

RESIDUAL OUTPUT

Observation	*Predicted Survey*	*Residuals*
1	122.2393	-0.7393
2	129.0881	0.9119
3	123.5660	-0.5660
4	133.4694	0.5306
5	145.6172	0.3828
6	158.6817	-0.3484
7	167.1715	-0.1715

Table 58 Analysis #1 for five-metric model.

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	91.8875	10.5413	8.7169	0.0727	-42.0524	225.8273
Obj. Avg Com. Lines	2.6406	0.3471	7.6072	0.0832	-1.7699	7.0512
Obj. Avg LOC	-0.2582	0.0995	-2.5945	0.2342	-1.5225	1.0061
Func. Avg LOC	1.4463	0.3649	3.9631	0.1574	-3.1908	6.0835
Obj. Mem. Func.	0.0653	0.0246	2.6550	0.2293	-0.2472	0.3778
Func. Avg Eff	-0.0137	0.0022	-6.3251	0.0998	-0.0414	0.0139

Table 59 Analysis #2 for five-metric model.

Chapter 5

5. Summary

5.1. Conclusions

This research has identified software metrics that can be used as maintainability predicators. Candidate models for to calculate a Maintainability Index have been developed using those metrics either singularly, or in combinations show good promise in predicting the maintainability and thus the maintenance effort required for samples of object oriented C++ program modules.

The initial results indicate that single metric models are generally weak indicators but several still show promise and require further evaluation. The best single metric model is based on the "Average Halstead's Effort" metric[4]. The Halstead's Effort metric tries to quantify the effort required to understand a particular function by using its size and difficulty as parameters. This is consistent with previous work. None of the object-oriented metrics appear to by strong maintainability indicators by themselves. The best indicator is the average commenting in an object definition.

Two-metric models seem to account for much of the variance in the sample set. Average comment lines for an object together with: 1) average lines of code per function, and 2) the average effort per function, appear to be good maintainability predicators. The

[4] The R^2 value is 0.7880.

conclusion of this is that having comments for each class and keeping the member functions reasonably sized and not overly complex makes the code more maintainable. This is consistent with our intuition.

The three-metric models are even better maintainability predicators. The best model is the model with average comment lines per object, member functions for an object and the average effort for functions. This is consistent with our intuition that every class should be commented; there should be a reasonable number of member functions, each with a reasonable effort.

The five-metric model is the best maintainability predicator. Basically, it tells us that comments for classes, class size (i.e. number of lines of the header), the number of class methods, the average method size (in lines of code) and the average effort for the methods is a good predicator. Interestingly, none of these metrics are “advanced”; most of them they are just simple counts. This is not to say that the more complicated metrics are not useful. Some of them were not implemented and the sample source code sets may not have been large enough to trigger correlation with the other metrics, such as inheritance.

Using a single metric create a maintainability index is not promising. Multi-metric models Show considerable promise. The metrics for the models can be easily and quickly calculated with developed tools. In this research, 11 such models have been investigated where 8 were identified as good maintainability predicators. This research

shows that the maintainability can be mechanically estimated using tools and multi-metric models.

5.2. Further Work

The [Chidamber, 91] metrics seems to be good metrics to support according to [Basili, 95] and others. The remaining metrics should be added to *m3*, but it is the author's opinion that *m3* needs some redesign. The inline documentation (commenting) for *m3* is sparse and a redesign may be appropriate. An improved metric structure is suggested, perhaps using an object-oriented language and creating a class hierarchy of metrics classes corresponding to the various modules. The CBO (Coupling Between Objects) metric [Chidamber, 91] was considered, but was found unfeasible to implement in the current (*m2*) design. Future work should look at CBO, and also COF (Coupling Factor) from the MOOD suite, as well as the RFC (Response For Class) metric.

It is the author's impression that most research on object-oriented software metrics and maintainability has been focusing on the inheritance part of the object-oriented paradigm. Other concepts, such as polymorphism and metrics need more study. These concepts have not been overlooked and the MOOD suite consists of metrics that intend to measure encapsulation, inheritance, coupling and polymorphism. [Harrison, 98] notes that the MOOD suite is more useful for managers while the MOOSE [Chidamber, 91] suite is more useful for designers and programmers.

Many professionals consider C++ to be a fairly complex language, largely because of its hybrid design. One of the consequences is that cleaner languages like Java gain

popularity. However, C++ is powerful and, once the complexity is mastered, it is indeed a favorable language for many problem domains due to its flexibility, power, speed, availability etc. In order to better understand C++ programs and how to improve them, metrics specialized for C++ should be investigated further. For example, most metrics are generic, but metrics taking into account `public`, `protected` and `private` keywords of C++ (and Java) could be used further. The same goes for `friend` and operator overloading, both often viewed with suspicion as features that can add considerable complexity or lead to error-prone software. For a discussion on friends, see [Page-Jones, 95] and [Joyner, 99]. Namespaces is another feature in C++ that may be taken into consideration. Other C++ code qualities are extensive use of `const`, both in terms of constant member functions and constant references. Another issue to examine would be the use of pointers versus references, the former more error-prone due to the risk of null pointers or uninitialized pointers. Metrics differing between public, protected, private and virtual inheritance is another topic, although in practice, public inheritance is by far the most used.

Another language feature to consider more closely is templates. STL is becoming increasingly popular as more and more compilers mature. Advanced use of templates is getting more and more attention and templates are now being used as more than just generic containers [Alexandrescu, 01] [Czarnecki, 00].

It is the author's belief that far more work needs to be done in this area, finding pragmatic, C++ specific metrics rather than generic ones. [Basili, 95] also recognizes this.

Further study of a combination of design patterns and metrics would also be useful. Design patterns seem to be popular in the industry while metrics still is mostly used in the academia. If metrics can suggest design patterns, for example, metrics might be more supported by the industry [Tonella, 99].

Effort should be focused on object-oriented *design* metrics rather than *code* metrics. [Basili, 95] has shown that object-oriented design metrics are better indicators of faults than code metrics. [Hsia, 95] distinguishes between micro level and macro level. [Martin, 94] proposes some design quality metrics.

Further research on models with larger set of source code is probably necessary to identify more of the object-oriented metrics as maintainability predicators. For this research, for example, the sample sets were probably too small for metrics such as DIT (Depth of Inheritance Tree) to be effective. Lastly, the *m3* tool may be modified to actually include a maintainability evaluation based on the research for this thesis and the multi-metric models developed.

Bibliography

[Abreu, 95] F. B. Abreu, "The MOOD Metrics Set," *Proceedings ECOOP'95 Workshop on Metrics*, 1995.

[Abreu, 96] F. B. Abreu & W. Melo, "Evaluating the Impact of Object-Oriented Design on Software Quality," *Proceedings of the 3rd International Software Metrics Symposium (METRICS '96)*, Berlin, Germany, March 1996.

[Alexandrescu, 01] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001.

[Arthur, 89] J. Arthur & K. Stevens, "Assessing the Adequacy of Documentation Through Document Quality Indicators," *Proceedings Conference on Software Maintenance – 1989*, (Oct. 16-19, Miami, Florida), IEEE Computer Society Press, Washington DC, 1989, pp. 40-49.

[Basili, 95] V. R. Basili, L. Briand & W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, Oct. 1996, 22(10).

[Bieman, 95] J. M. Bieman & J. X. Zhao, "Reuse Through Inheritance: A Quantiative Study of C++ Software," *Proceedings of the 17th International Conference on Software Engineering on Symposium on Software Reusability*, (1995, Seattle).

[Booch, 94] G. Booch, *Object-oriented Analysis and Design with Applications*, 2nd Ed., Benjamin/Cummings, 1994.

[Booch, 96] G. Booch, *Object Solutions*, Addison-Wesley, 1996.

[Briand, 97] L. C. Briand, J. W. Daly & J. Wüst, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Proceedings of the Fourth Software Metrics Symposium*, (Nov. 5-7, 1997, Albuquerque, New Mexico).

[Brooks, 95] F. P. Brooks, *The Mythical Man-Month*, Addison-Wesley Longman 1995.

[Cartwright, 98] M. Cartwright, "An empirical view of inheritance," *Information and Software Technology*, 40(14), Dec. 1998, pp. 795-799.

[Chidamber, 91] S. Chidamber & C. Kemerer, “Towards a metric suite for object-oriented design,” *Proceedings OOPSLA ’91*, Sigplan Notices, 26(11), pp. 197-211.

[Chidamber, 94] S. Chidamber & C. Kemerer, “A metrics suite for object-oriented design,” *IEEE Transactions on Software Engineering*, 1994, 20(6), pp. 476-493.

[Coplien, 98] J. Coplien, *Multi-Paradigm Design for C++*, Addison-Wesley, 1998.

[Czarnecki, 00] K. Czarnecki & U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[Daly, 95] J. Daly, A. Brooks, J. Miller, M. Roper & M. Wood, “The Effect of Inheritance on the Maintainability of Object-Oriented Software: An Empirical Study,” *1995 Conference on Software Maintenance*, (Oct. 17-20, 1995, Opio (Nice), France).

[Edelstein, 93] D. Edelstein, “Report on the IEEE STD 1219-1993-standard for software maintainance,” *ACM SIGSOFT Software Engineering Notes*, 1993, 18(4), pp. 94-95.

[Fenton, 96] N. E. Fenton, *Software Metrics: A Rigorous and Practical Approach*, International Thomson Computer Press, 1996.

[Fowler, 00] M. Fowler & K. Scott, *UML Distilled Second Edition: A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley, 2000.

[Gamma, 95] E. Gamma, R. Helm, R. Johnson & J. Vlissides, *Design Patterns*, Addison-Wesley, 1995.

[Glasberg, 00] D. Glasberg, K E. Emam, W. Melo & N. Madhavji, “Validating Object-oriented Design Metrics on a Commerical Java Application,” National Research Council of Canada, Sept. 2000.

[Hagemeister, 92] J. Hagemeister, “A Metric Approach to Assessing the Maintainability of Software,” 1992.

[Harrison, 98] R. Harrison, S. J. Counsell & R. V. Nithi, “An Evaluation of the MOOD Set of Object-Oriented Software Metrics,” *IEEE Transactions on Software Engineering*, 24(6), June 1998.

[Henry, 81] S. Henry & D. Kafura, “Software structure metrics based on information flow,” *IEEE Transactions on Software Engineering*,

1981, 7(5), pp. 510-518.

[Hogan, 97] J. Hogan, "An Analysis of OO Software Metrics," Research Report CS-RR-324, Department of Computer Science, University of Warwick, Coventry, UK, May 1, 1997.

[Hsia, 95] P. Hsia, A. Gupta, C. Kung, J. Peng & S. Liu, "A Study on the Effect of Architecture on Maintainability of Object-Oriented Systems," *1995 IEEE Conference on Software Maintenance*, (Oct. 17-20, 1995, Opio (Nice), France), pp. 4-11.

[Huston, 01] B. Huston, "The effects of design pattern application on metric scores," *The Journal of Systems and Software*, 58(3), Sep. 2001, pp. 261-269.

[IEEE 1045] *IEEE Standard for Software Productivity Metrics*, IEEE Std 1045-1992.

[IEEE 1061] *IEEE Standard for a Software Metrics Methodology*, IEEE Std 1061-1992.

[IEEE 610.12] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12-1990.

[IEEE 982.1] *IEEE Standard Dictionary of Measures to Produce Reliable Software*, IEEE Std 982.1-1988.

[IEEE 982.2] *IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*, IEEE Std 982.2-1998.

[Jia, 00] X. Jia, *Object-Oriented Software Development using Java*, Addison-Wesley, 2000.

[Joyner, 99] I. Joyner, *Objects Unencapsulated: Java, Eiffel, and C++??*, Prentice Hall, 1999.

[Kafura, 98] D. Kafura, *Object-Oriented Software Design and Construction with C++*, Prentice-Hall, 1998.

[Kiran, 97] G. A. Kiran, S. Haripriya & P. Jalote, "Effect of Object Orientation on Maintainability of Software," *Proceedings International Conference on Software Maintenance – 1997*, (Oct. 1-3, Bari, Italy), IEEE Computer Society, Los Alamitos, California, 1997, pp.114-121.

[Lee, 95] Y.-S. Lee, B.-S. Liang, S.-F. Wu & F.-J. Wang, "Measuring the

Coupling and Cohesion of an Object-Oriented Program Based on information Flow", *Proc. International Conference on Software Quality*, Maribor, Slovenia, 1995.

[Li, 93] W. Li & S. Henry, "Object-Oriented Metrics that Predict Maintainability," Journal of Systems Software 1993, (23), pp. 111-122.

[Lientz, 78] B. Lientz, E. Swanson & G. Tompkins, "Characteristics of Application Software Maintenance," *Communications of the ACM*, 21(6), June 1978, pp. 466-471.

[Lientz, 80] B. Lientz & E. Swanson, *Software Maintenance Managament*, Addison-Wesley, 1980.

[Lorenz, 93] M. Lorenz, *Object-Oriented Software Development: A Practical Guide*, Prentice Hall, 1993.

[Lorenz, 94] M. Lorenz & J. Kidd, *Object-Oriented Software Metrics*, Prentice Hall, 1994.

[Madsen, 93] O. L. Madsen, Møller-Pedersen & Nygaard, *Object-Oriented Programming in the BETA Programming Language*, Addison-Wesley, 1993.

[Martin, 00] R. C. Martin, "Design Principles and Design Patterns," 2000. http://www.objectmentor.com

[Martin, 83] J. Martin & C. McClure, *Software Maintenance: The Problem and Its Solutions*, Prentice-Hall, 1983.

[Martin, 94] R. C. Martin, "OO Design Quality Metrics – An Analysis of Dependencies," http://www.objectmentor.com, 1994.

[Martin, 96a] R. C. Martin, "The Open Closed Principle," C++ Report, Jan. 1996.

[Martin, 96b] R. C. Martin, "The Liskov Substition Principle," C++ Report, March 1999.

[Martin, 96c] R. C. Martin, "The Dependency Inversion Principle," C++ Report, May 1996.

[Martin, 96d] R. C. Martin, "The Interface Segregation Principle," C++ Report, Aug. 1996.

[Masuda, 99] G. Masuda, N. Sakamoto & K. Ushijima, "Investigating Metrics for

Applying Design Patterns in Object-Oriented Software Design," *Proceedings of IASTED International Conference Software Engineering and Applications*, (Oct. 6-8, 1999, Scottsdale, Arizona), pp.144-148.

[Mattsson, 00] M. K. Mattsson, "Preventice Maintenance! Do we know what it is?" *2000 IEEE International Conference on Software Maintenance*, pp. 12-14.

[Meyer, 88] B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, 1988.

[Meyer, 97] B. Meyer, *Object-Oriented Software Construction*, 2nd Ed., Prentice Hall, 1997.

[Meyers, 92] S. Meyers, *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*, Addison-Wesley, 1992.

[Murphy, 96] G. Murphy and D. Notkin, "Lightweight Lexical Source Model Extraction," *ACM Transactions on Software Engineering and Methodology*, 5(3), July 1996, pp. 262-292.

[Oman, 92] P. Oman & J. Hagemeister, "Metrics for Assessing a Software System's Maintainability," *Proceedings 1992 IEEE Conference on Software Maintenance*, (Nov. 9-12, Orlando, Florida), IEEE Computer Society, pp. 337-344.

[Oman, 93] P. Oman & J. Hagemeister, "Construction and Validation of Polynomials for Predicting Software Maintainability," *Proceedings of the Fifth Annual Oregon Workshop on Software Metrics*, (Mar. 21-23, Silver Falls, Oregon), 1993.

[Oman, 94] P. Oman & J. Hagemeister, "Using Halstead Metrics for Assessing Software Maintainability," *Journal of Systems and Software*, 24(3), Mar. 1994, pp. 251-266.

[Page-Jones, 95] M. Page-Jones, *What Every Programmer Should Know About Object-Oriented Design*, Dorset House Publishing, 1995.

[Pearse, 95] T. Pearse & P. Oman, "Maintainability Measurements on Industrial source Code Maintenance Activities," *Proceedings 1995 IEEE Conference on Software Maintenance*, (Oct. 16-20, Nice, France), IEEE Computer Society.

[Penner, 99] M. Penner, "The Conceptual Source Code Module: An Approach to Processing C and C++ Source Code Files," 1999.

[Penner, 99b] M. Penner, P. Oman & J. Hagemeister, "Measuring the Complexity of Non-preprocessed C/C++ Source Code Files," *Proceedings of the IASTED International Conference Software Engineering and Applications*, (Oct. 6-8, 1999, Scottsdale, Arizona).

[Sellers, 94] B. Henderson-Sellers and J. M. Edwards, *BOOKTWO of Object-Oriented Knowledge: The Working Object*. Prentice Hall, 1994.

[Sellers, 96] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*, Prentice Hall, 1996.

[Shalloway, 02] A. Shalloway & J. R. Trott, *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Addison-Wesley, 2002.

[Stroustrup, 97] B. Stroustrup, *The C++ Programming Language*, 3rd Ed., Addison-Wesley, 1997.

[Tonella, 99] P. Tonella & G. Antoniol, "Object Oriented Design Pattern Inference," *1999 IEEE International Conference on Software Maintenance*, (Aug. 30 – Sep. 3, 1999, Oxford, England)

[Vlissides, 98] J. Vlissides, *Pattern Hatching: Design Patterns Applied*, Addison-Wesley, 1998.

[Wegner, 91] P. Wegner, "Concepts and Paradigms of Object-Oriented Programming," *ACM SIGPLAN OOPS Messenger*, 1(1), Aug. 1990.

[Zhuo, 93] F. Zhuo, B. Lowther, P. Oman & J. Hagemeister, "Constructing and Testing Software Maintainability Assessment Models," *Proceedings of the First International Software Metrics Symposium, IEEE Computer Society Press*, 1993, pp. 61-70.

[Zuse, 98] H. Zuse, *A Framework of Software Measurement*, Walter de Gruyter, 1998.

References

[1] "TAC++ - A Tool for Assessment of C++ Applications"
http://www.dsi.unifi.it/~nesi/wwwtac++nuova/

[2] "Understand for C++"
http://www.scitools.com/ucpp.html

[3] "CCCC – C and C++ Code Counter"
http://cccc.sourceforge.net/

[4] "MAISA – Metrics for Analysis and Improvement of Software Architectures"
http://www.cs.helsinki.fi/group/maisa/

[5] "NiL – NiL Isn't Liero"
http://nil.sourceforge.net/

[6] "Qwt"
http://qwt.sourceforge.net/

[7] "Trolltech AS"
http://www.trolltech.com/

[8] "DOC++"
http://sourceforge.net/projects/docpp/

[9] "Principal component analysis"
http://people.imt.liu.se/~magnus/cca/tutorial/node17.html

[10] "Principal component analysis"
http://diwww.epfl.ch/mantra/tutorial/english/pca/html/

[11] "Principal component analysis"
http://fonsg3.let.uva.nl/praat/manual/Principal_component_analysis.html

[12] "GNU General Public License"
http://www.gnu.org/licenses/gpl.html

Appendix

A. Abbrevations

Abbreviation	Metric
AC	Association Complexity
AHF	Attribute Hiding Factor
AIF	Attribute Inheritance Factor
AMC	Average Method Coupling
CBO	Coupling Between Objects
CC	Class Complexity / Class Coupling / Cyclomatic Complexity
CCM	Cognitive Complexity Model
CCO	Class Cohesion
CCP	Class Coupling
CF	Coupling Factor
CIC	Class Inheritance-related Coupling
CLOC	Comment Lines of Code
CLM	Comment Lines per Method
CNIC	Class Non inheritance-related Coupling
COF	Coupling Factor
CPC	Class Path Coupling
CR	Comment Ratio
CRE	Class Reuse
CSCM	Conceptual Source Code Module
DAC	Data Abstraction Coupling
DIT	Depth of Inheritance (also called DOIH)
DOIH	Depth of Inheritance Hierarchy (also called DIT)
FFU	Friend Functions
FO	Fan-Out
FOC	Function-Oriented Code
FPA	Function Point Analysis
GUS	Global Usage
HNL	Hierarchy Nesting Level
IVU	Instance Variable Usage
LCOM	Lack of Cohesion of Methods
LOC	Lines of Code
MC	Method Coupling
MCX	Method Complexity
MDF	Multidimensional Framework (for OO metrics)
MHF	Method Hiding Factor
MNOP	Maximum Number of Parameters
MNOL	Maximum Number of Levels

MOD	Average module size in lines of code
MIF	Method Inheritance Factor
MS	Method Size
MSOO	Maximum Size of Operation
MPC	Message Passing Coupling
MUI	Multiple inheritance
NCV	Number of Class Variables
NCM	Number of Class Methods
NEM	Number of External Methods
NHM	Number of Hidden Methods
NIM	Number of Instance Methods
NIV	Number of Instance Variables
NLM	Number of Local Methods
NMA	Number of Methods added
NMI	Number of Methods Inherited
NMO	Number of Methods Overridden
NOA	Number of Attributes
NOA	= NIV + NCV
NOAM	Number of Added Methods
NOC	Number of Children / Number of Classes
NOCC	Number of Child Classes
NOIS	Number of Import Statements
NOO	Number of Operations
NOOM	Number of Overridden Methods
NOM	Number of (public) Methods (per class)
NOM	= NHM + NEM = NIM + NCM
	Number of Members (attributes and operations)
	Number of messages sends
NORM	Number of Remote Methods (see NRM)
NOT	Number of Traps
NRM	Number of Remote Methods (see NORM)
O/C	Object/Class
PCM	Percentages of Commented Methods
PF	Polymorphism Factor
PIM	Number of Public Instance Methods
POF	Polymorphism Factor
PPM	Parameters per Method
RFC	Response for Class
	= NLM + NRM
RS	Response Set
S/C	Size/Complexity
SIX	Specialization Index
TCR	True Comment Ratio
TMS	Total Method Size
SLOC	Source Lines of Code
VOD	Violations of the Law of Demeter

WAC	Weighted Attributes per Class
WMC	Weighted Methods per Class

Table 60 Terms collected from the metrics literature

B. Compatibility

The *m3* tool was successfully compiled on:

- Windows NT & 2000 using Microsoft Visual Studio 6.0.
- HP-UX B.10.20 and the cc compiler provided with the OS.
- Linux 2.4.3-12, gcc 2.96.

C. Survey

Source Listing

	completely disagree	strongly disagree	generally disagree	generally agree	strongly agree	completely agree
1) Functionally related data elements have been organized into logical classes.	❑	❑	❑	❑	❑	❑
2) The number of exit points within classes of the program is not excessive.	❑	❑	❑	❑	❑	❑
3) Each module performs only related (similar) functions.	❑	❑	❑	❑	❑	❑
4) The quantity of comments does not detract from the legibility of the source listings.	❑	❑	❑	❑	❑	❑
5) Transfers of control and destinations are clearly explained.	❑	❑	❑	❑	❑	❑
6) Each variable in this program is considered to be of one (and only one) data type for all occurrences.	❑	❑	❑	❑	❑	❑
7) Each variable in this program has only one use.	❑	❑	❑	❑	❑	❑
8) Knowledge beyond basic mathematics is not required to understand the functions of this program.	❑	❑	❑	❑	❑	❑
9) The number of executable statements in each module of this program is manageable.	❑	❑	❑	❑	❑	❑
10) There is a minimal mixing of I/O functions and other application functions in this program.	❑	❑	❑	❑	❑	❑
11) The size of any data structures that affects the processing logic of this program is parameterized.	❑	❑	❑	❑	❑	❑
12) Any constraints (e.g. accuracy, convergence, and timing) which affect processing in this program are parameterized.	❑	❑	❑	❑	❑	❑
13) Functional modules can be easily inserted, deleted, or replaced within this program.	❑	❑	❑	❑	❑	❑

	completely disagree	strongly disagree	generally disagree	generally agree	strongly agree	completely agree
14) Diagnostic messages/error codes are output when an illegal input to this program is encountered.	❑	❑	❑	❑	❑	❑
15) Diagnostic messages/error codes are output wherever an internal program failure could occur.	❑	❑	❑	❑	❑	❑
16) The class hierarchy is logical and understandable.	❑	❑	❑	❑	❑	❑
17) The class hierarchy promotes reuse.	❑	❑	❑	❑	❑	❑
18) The classes are well encapsulated.	❑	❑	❑	❑	❑	❑
19) The use of friend functions is normal and expected.	❑	❑	❑	❑	❑	❑
20) Overloading of operators are logical and understandable.	❑	❑	❑	❑	❑	❑
21) The abstractions are sound and understandable with proper separation of interface and implementation.	❑	❑	❑	❑	❑	❑
22) The design is purely object-oriented.	❑	❑	❑	❑	❑	❑
23) The source code is reusable.	❑	❑	❑	❑	❑	❑
24) The source code uses abstract data types well.	❑	❑	❑	❑	❑	❑
25) The classes have high cohesion.	❑	❑	❑	❑	❑	❑
26) The classes have low coupling.	❑	❑	❑	❑	❑	❑
27) The number of methods in the class is reasonable.	❑	❑	❑	❑	❑	❑

For questions 28 through 33, apply the given definition to the software system being evaluated and then respond to the question.

Definition: "Software possesses the characteristic of *modularity* to the extent a logical partitioning of software into parts, components, and/or modules has occurred."

28) Modularity as reflected in the source code listing contributes to the maintainability of the program.

completely disagree	strongly disagree	generally disagree	generally agree	strongly agree	completely agree

---

Definition: "Software possess the characteristic of *descriptiveness* to the extent that it contains information regarding its objects, assumptions, inputs, processing, outputs, components, revision status, etc."

29) Descriptiveness as reflected in the source code listing contributes to the maintainability of the program.

completely disagree	strongly disagree	generally disagree	generally agree	strongly agree	completely agree

---

Definition: "Software possesses the characteristic of *consistency* to the extent that the software documentation products correlate and contain uniform notation, terminology, and symbology."

30) Consistency as reflected in the source code listing and between the source listing and the documentation contributes to the maintainability of the program.

completely disagree	strongly disagree	generally disagree	generally agree	strongly agree	completely agree

---

Definition: "Software possesses the characteristic of *simplicity* to the extent that it reflects the use of singular concepts and fundamental structures."

31) Simplicity as reflected in the source code listing contributes to the maintainability of the program.

❑	❑	❑	❑	❑	❑
completely disagree	strongly disagree	generally disagree	generally agree	strongly agree	completely agree

---

Definition: "Software possesses the characteristic of *expandability* to the extent that a physical change to information, computational functions, data storage or execution time, can be easily accomplished once the nature of what is to be changed is understood."

32) Expandability as reflected in the source code listing contributes to the maintainability of the program.

❑	❑	❑	❑	❑	❑
completely disagree	strongly disagree	generally disagree	generally agree	strongly agree	completely agree

---

Definition: "Software possesses the characteristic of *testability* to the extent that it contains aids which enhance testing."

33) Testability as reflected in the source code listing to the maintainability of the program.

❑	❑	❑	❑	❑	❑
completely disagree	strongly disagree	generally disagree	generally agree	strongly agree	completely agree

---

Question 34 pertains to the whole program source code.

34) Overall it appears that the characteristics of the program source code contribute to the maintainability of the program.

❑	❑	❑	❑	❑	❑
completely disagree	strongly disagree	generally disagree	generally agree	strongly agree	completely agree

---

Now please categorize this software system as generally having one of the following:

- ❑ Very low maintainability
- ❑ Low maintainability
- ❑ Medium maintainability
- ❑ High maintainability
- ❑ Very high maintainability

D. Output Format

Object

1. Public functions
2. Protected functions
3. Private functions
4. Number of children
5. Average lines of code
6. Public data
7. Protected data
8. Private data
9. Virtual functions
10. Member functions
11. Data members
12. Comments inside
13. Comment lines inside
14. Comments outside
15. Comment lines outside
16. Comments
17. Comment lines
18. Blank LOC inside
19. Nonblank LOC inside
20. Blank LOC outside
21. Nonblank LOC outside
22. Blank LOC
23. Nonblank LOC
24. Total LOC
25. Max variable length
26. Average variable length
27. Highest count of function parameters
28. Average number of parameters per method
29. Number of subclasses
30. Specialization index
31. Depth
32. Parent count
33. Number of methods overridden
34. Number of methods added
35. Weighted methods per class
36. Maximum size of operation
37. Comment density

Function

1. Fan-in
2. Fan-out
3. Information flow metric
4. Function type
5. Comments inside
6. Comment lines inside
7. Comments outside
8. Comment lines outside
9. Comments
10. Comment lines
11. Blank LOC inside
12. Nonblank LOC inside
13. Blank LOC outside
14. Nonblank LOC outside
15. Blank LOC
16. Nonblank LOC
17. Total LOC
18. Total VG1
19. Total VG2
20. Total operators
21. Unique operators
22. Total operands
23. Unique operands
24. Halstead vocabulary
25. Halstead length
26. Halstead volume
27. Halstead effort
28. Maximum nesting level
29. Average nesting level
30. Total nesting level
31. Nesting level
32. Maximum variable length
33. Average variable length
34. Function parameters

Container

1. Virtual functions
2. Member functions
3. Data members
4. Comments inside
5. Comment lines inside
6. Comments outside
7. Comment lines outside
8. Comments
9. Comment lines
10. Blank LOC inside
11. Nonblank LOC inside
12. Blank LOC outside
13. Nonblank LOC outside
14. Blank LOC
15. Nonblank LOC
16. Total LOC
17. Maximum variable length
18. Average variable length

File

1. Preprocessor directives
2. Member functions
3. Comments
4. Comment lines
5. Blank LOC
6. Nonblank LOC
7. Total LOC

E. Principal Component Analysis Results

Files PCA

	Prin 1	Prin 2	Prin 3	Prin 4	Prin 5	Prin 6	Prin 7
Preprocessor directives	0.0158	-0.1214	0.0498	-0.3194	**0.9221**	-0.1739	0.0000
Member functions	0.0262	0.0255	0.0354	-0.1574	-0.2342	**-0.9580**	-0.0000
Comments	0.0564	-0.2151	**-0.7164**	**-0.6241**	-0.1847	0.1171	-0.0000
Comment lines	0.1435	**-0.9476**	0.0655	0.2688	-0.0473	-0.0514	0.0000
Blank LOC	0.1016	0.1226	**-0.5569**	**0.5201**	0.1967	-0.1481	-0.5774
Nonblank LOC	**0.6420**	0.0330	0.3608	-0.3130	-0.1136	0.1110	-0.5774
Total LOC	**0.7435**	0.1556	-0.1961	0.2071	0.0831	-0.0371	0.5774

Functions PCA

	Prin 1	Prin 2	Prin 3	Prin 4	Prin 5	Prin 6	Prin 7
Fan-in	0.0001	-0.0003	0.0002	-0.0027	0.0087	-0.0016	0.0217
Fan-out	0.0024	0.0010	-0.0281	-0.0840	0.0017	-0.0316	-0.2957
Information flow metric	**0.3132**	**-0.9497**	-0.0065	0.0009	0.0000	0.0002	-0.0002
Function type	0.0000	0.0000	-0.0002	-0.0009	0.0018	-0.0037	-0.0001
Comments inside	0.0008	0.0002	-0.0027	-0.0191	-0.0414	-0.1083	-0.0475
Comment lines inside	0.0008	0.0002	-0.0025	-0.0231	-0.0516	-0.1238	-0.0358
Comments outside	-0.0000	-0.0000	0.0001	-0.0055	-0.0315	0.0352	-0.0107
Comment lines outside	-0.0001	-0.0000	0.0027	-0.0391	-0.2943	0.3793	-0.0677
Comments	0.0008	0.0002	-0.0025	-0.0246	-0.0729	-0.0731	-0.0582
Comment lines	0.0007	0.0002	0.0002	-0.0622	-0.3459	0.2555	-0.1034
Blank LOC inside	0.0013	0.0004	-0.0037	-0.0179	-0.0231	-0.0786	0.0100
Nonblank LOC inside	0.0067	0.0021	-0.0216	-0.1758	-0.1346	**-0.5501**	0.1935
Blank LOC outside	-0.0000	0.0000	-0.0002	-0.0055	-0.0380	0.0350	0.0047
Nonblank LOC outside	-0.0001	-0.0000	0.0029	-0.0419	-0.3388	**0.4281**	-0.0632
Blank LOC	0.0012	0.0004	-0.0039	-0.0235	-0.0611	-0.0436	0.0147
Nonblank LOC	0.0066	0.0021	-0.0187	-0.2177	**-0.4734**	-0.1220	0.1304
Total LOC	0.0078	0.0025	-0.0226	-0.2412	**-0.5345**	-0.1657	0.1451
Total VG1	0.0013	0.0002	-0.0017	-0.0474	0.0011	-0.0571	-0.0268
Total VG2	0.0015	0.0002	-0.0018	-0.0463	0.0045	-0.0561	0.0271
Total operators	0.0245	0.0081	-0.0455	-0.3482	0.0884	-0.0516	-0.2656
Unique operators	0.0245	0.0081	-0.0455	-0.3482	0.0884	-0.0516	-0.2656
Total operands	0.0147	0.0049	-0.0317	-0.0583	0.1177	0.2256	**0.6661**
Unique operands	0.0195	0.0069	-0.0879	-0.2263	0.1211	0.1623	0.0746
Halstead vocabulary	0.0440	0.0150	-0.1334	**-0.5745**	0.2095	0.1107	-0.1909
Halstead length	0.0392	0.0130	-0.0772	**-0.4064**	0.2061	0.1740	0.4005
Halstead volume	**0.3745**	**0.1298**	**-0.8936**	0.1965	-0.0482	-0.0233	-0.0143
Halstead effort	**0.8696**	**0.2840**	**0.4032**	-0.0059	0.0011	0.0001	-0.0033
Maximum nesting level	0.0002	0.0001	-0.0001	-0.0237	0.0065	-0.0251	0.0425
Average nesting level	0.0001	0.0000	-0.0000	-0.0127	0.0038	-0.0082	0.0177
Total nesting level	0.0022	0.0006	-0.0052	-0.0753	-0.0330	-0.2792	0.1201
Nesting level	0.0011	0.0002	-0.0028	-0.0424	0.0049	-0.0703	0.0399
Maximum variable length	0.0004	0.0001	-0.0029	-0.0579	0.0187	0.0545	0.0544
Average variable length	-0.0000	-0.0000	-0.0006	-0.0010	-0.0111	0.0068	-0.0087
Function parameters	0.0000	0.0000	-0.0002	-0.0016	0.0027	-0.0001	-0.0004

Objects PCA

	Prin 1	Prin 2	Prin 3	Prin 4	Prin 5	Prin 6	Prin 7
Public functions	0.0978	-0.2132	0.2960	-0.0377	0.0925	-0.1286	0.0286
Protected functions	0.0030	-0.0064	0.0101	-0.0052	-0.0023	-0.0183	0.0987
Private functions	0.0131	-0.0025	0.0587	-0.0215	0.0954	0.1215	-0.0202
Number of children	-0.0007	-0.0006	-0.0005	0.0022	0.0110	0.0032	0.0076
Average lines of code	0.0207	-0.1710	-0.2775	**0.4042**	**0.6013**	**-0.3576**	0.2924
Public data	-0.0000	-0.0000	-0.0000	-0.0000	0.0000	0.0000	0.0000
Protected data	-0.0000	-0.0000	-0.0000	-0.0000	-0.0000	0.0000	-0.0000
Private data	-0.0000	-0.0000	-0.0000	-0.0000	-0.0000	-0.0000	0.0000
Virtual functions	0.0052	-0.0392	0.0516	-0.0427	0.0613	-0.0601	-0.0123
Member functions	0.1100	-0.2311	**0.3554**	-0.1415	0.2313	0.0556	0.1925
Data members	0.1210	-0.1640	-0.1717	-0.2046	0.0330	**0.7017**	0.2355
Comments inside	-0.0000	-0.0000	-0.0000	-0.0000	-0.0000	-0.0000	-0.0000
Comment lines inside	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Comments outside	0.0661	-0.0877	-0.2512	-0.2373	-0.2478	-0.2086	0.3960
Comment lines outside	0.2147	**0.4970**	-0.1884	-0.1016	0.1452	-0.0672	-0.0343
Comments	0.0806	-0.0905	-0.2005	-0.1032	-0.3355	-0.1333	0.5976
Comment lines	0.2473	**0.4937**	-0.0694	0.2180	0.0015	0.1555	0.2042
Blank LOC inside	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Nonblank LOC inside	0.0000	-0.0000	-0.0000	-0.0000	-0.0000	-0.0000	-0.0000
Blank LOC outside	0.1035	-0.1918	-0.1717	-0.2224	-0.1371	-0.2086	-0.2130
Nonblank LOC outside	**0.4180**	0.1063	-0.1614	**-0.6118**	0.3711	-0.1395	-0.1673
Blank LOC	0.1195	-0.2217	-0.0981	-0.0224	-0.2416	-0.2144	-0.2174
Nonblank LOC	**0.4964**	0.0451	0.1879	0.2911	-0.0357	0.1218	0.0636
Total LOC	**0.6159**	-0.1766	0.0898	0.2688	-0.2773	-0.0926	-0.1538
Max variable length	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Average variable length	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Highest count of func. param.	0.0237	-0.0071	0.0366	0.0136	0.0330	-0.0963	0.1292
Avg num of param per method	0.0026	-0.0122	0.0105	-0.0001	0.0150	-0.0237	-0.0190
Number of subclasses	-0.0007	-0.0006	-0.0005	0.0022	0.0110	0.0032	0.0076
Specialization index	-0.0000	-0.0006	-0.0008	0.0029	0.0052	-0.0041	0.0018
Depth	-0.0023	-0.0049	-0.0012	0.0096	0.0088	-0.0199	-0.0336
Parent count	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Number of methods overridden	-0.0002	-0.0023	-0.0029	0.0108	0.0197	-0.0137	0.0072
Number of methods added	0.1101	-0.2287	**0.3583**	-0.1524	0.2115	0.0693	0.1853
Weighted methods per class	0.0803	-0.3776	**-0.5203**	0.1760	0.1247	0.3196	-0.2299
Maximum size of operation	0.0167	-0.1060	-0.1440	0.1010	0.1094	-0.0410	0.0079
Comment density	0.0009	-0.0001	0.0019	0.0085	0.0023	0.0011	0.0122