CHAPTER 6

Metric Development for Object-Oriented Software

James M. Bieman

6.1 Introduction

A programming paradigm of great interest and activity is object-oriented programming and design. The reasons to develop and use software metrics for software implemented in imperative programming languages also apply to software developed using object-oriented programming languages. Perhaps metrics are even more important in the object-oriented paradigm, because, in many ways, object-oriented programming can be even more complex than imperative programming.

A foundation like measurement theory is critical for the development of metrics for object-oriented programs. Much of the work done in developing metrics for imperative programs was ad hoc. Users and researchers have tended to look at particular problems or needs and have come up with specialized solutions. The obvious difficulty is that these problems have for the most part been approached in an ad hoc manner without any theory or foundation to give guidance in the development of the solutions. Thus, there is no reliable way to understand how metrics for imperative programs can be adapted – if they can be – for object-oriented programs, and there is no real experience except for ad hoc methods that can be used to develop metrics for a new programming paradigm such as object-orientation. Thus, the work presented in this chapter is important for its own sake and as a way of gaining

Published in *Software Measurement*, edited by Austin Melton, pp. 75–92.
Copyright ©1996 International Thompson Computer Press.

Contents and format does not exactly match that in the published book.

insight into metric development which can at least in part be used when metrics for other programming paradigms are needed.

Software metrics are derived to quantify properties of both existing and planned software. Thus, software metrics research and development tends to focus on products developed using software development methods and languages that are current at the time of the research. Halstead's software science metrics focus on the basic language units of assembly and Fortran code—operators and operands (Halstead[77]). This focus on operators and operands matches the software development process commonly used in the 1960's and early 1970's. Much of the research on control flow measurement reflects the focus of the software engineering community on structured programming during the late 1970's (McCabe[76], Fenton-Whitty[86], Whitty[89], Howatt-Baker[85b,89], Zuse[91]). Metrics developed to quantify attributes such as module cohesion assume that a module is a procedure or function (Ott[92], Ott-Bieman[92], Ott-Thuss[93], Bieman-Ott[94]). This assumption also fits the structured programming paradigm.

Software measurement research must adapt to the emergence of new software development methods and new perspectives of software engineering. These emerging methods include specification techniques and logic programming. Metrics for new language and design paradigms must be based on models that are relevant to these paradigms (Melton *et al.*[90]). Myers and Kaposi give initial results on developing metrics for Prolog programs based on a model of Prolog data (Myers-Kaposi[91]). In the chapter *Analysis and Measurement Techniques for Logic-based Languages* McCauley and Edwards discuss new techniques for developing and verifying metrics for logic languages. Whitty defines metrics of Z specifications (Whitty[89]).

Measurement should be applicable to a high-level perspective of large systems with thousands of program units. Such design metrics need not deal with all of the details of an implementation. In a comprehensive 1988 report, Ince and Sheppard conclude that "progress has been very slow" in developing appropriate software design metrics (Ince-Sheppard[88]).

Object-oriented design and object-oriented programming appear to be the *structured programming* of the 1990's. Software metrics researchers need to focus on this new paradigm (Bieman[91]). Attributes that are well understood in the world of procedural programming may not be directly relevant to object-oriented software.

For example, the concepts of coupling and cohesion need to be reformulated in order to be applied to object-oriented systems or to abstract data types (Woodward[93]).

In an object-oriented system, the basic program unit is a class rather than a procedure. A class with its encapsulation of state with associated methods (operations) is a significantly different and richer abstraction than the procedure units within procedural programs. The inclusion of inheritance in object-oriented systems further complicates the static relations between classes. Models and abstractions that are appropriate to the object-oriented paradigm are needed.

Seldom have new design techniques been evaluated in a rigorous and quantitative fashion. The object-oriented paradigm is no exception. Proponents assert that object-oriented programming is the silver bullet solving the *software problem* (Cox[90a], Cox[90b]), with little data to support such claims. Object-oriented software is supposed to be easy to reuse, yet few quantitative studies on reuse have been conducted, and metrics to quantify the amount of reuse in object-oriented software are lacking. Object-oriented proponents argue about the effects of the use of particular language constructs. For example, the effects of the use of *private* inheritance on class reuse in C++ programs are frequently debated on the comp. lang. c++ internet news group. Empirical studies could resolve such debates.

Significant data exists for empirical studies of properties of object-oriented software. Many object-oriented software libraries are in the public domain. New systems, in theory, are extensions of existing code. Thus, once metrics are rigorously defined, plenty of data exists to test hypotheses regarding quantitative properties of object-oriented software.

The object-oriented paradigm is ideal for metrics research. Data is available and hypotheses concerning quantifiable properties of object-oriented software are known. However, rigorous, well defined models suitable for defining metrics of attributes of object-oriented software are needed. In this chapter we take a measurement theory approach; the focus is on the measurement of structural properties of software artifacts.

The rest of the chapter is organized as follows: Section 2 describes the differences between procedural and object-oriented software from the perspective of deriving software metrics. These differences translate into appropriate research goals for developing

metrics for object-oriented software. In Section 3, an overview of some of the on-going work in developing object-oriented metrics is given. Section 4 provides some details of the object-oriented reuse measurement project at Colorado State University, which has studied reuse in both Ada and C++ systems. The use of measurement to quantify desirable and undesirable structural style in object-oriented software is described in Section 5. Concluding remarks are in Section 6.

6.2 How Is Object Oriented Software Different?

From all of the hype concerning object-oriented programming, we must assume that there is something special about an object-oriented program. In this chapter the structure of object-oriented programs is examined so as to identify the unique aspects of these programs. Further, object-oriented software is examined from the perspective of software metrics development, and measuring attributes of the static structure of software documents is looked at. These are the structures used in object-oriented programs, object-oriented design, and object-oriented analysis. The two central abstraction mechanisms in object-oriented software are data abstraction and inheritance. The control and communication mechanism is message passing rather than direct procedure or function invocation. Global variables are not directly accessed. Rather, variables are accessed via messages passed to objects.

We can view the static definition of an object-oriented software system as a collection of abstract data types called classes. A *class* is an encapsulated specification of both the persistent state of an abstract data type and its operations. An instantiation or *instance* of a class is an *object*. There may be several concurrently active objects of one class; each instantiation is a different object. Suppose a class defines a stack abstract data type. We can instantiate several stack objects, and each object may contain different values in their stack frames. To change the internal state of an object, a message with a specified state changing effect must be sent to the object. Responses to messages are specified via *methods* which are components of classes. Methods are essentially procedures which are local to a class or inherited. Methods may have parameters, assignments to local variables and persistent class variables, and may send other messages. A stack class will contain methods for common stack operations such as *push, pop, top, is empty*, etc.

An *object-based* system is one that is built around *data abstraction*; an *object-oriented* system also makes use of *inheritance*. Inheritance provides language support for specifying that "I want something just like that except ... ." A developer can modify a particular class to create a new class that behaves somewhat differently than the parent class. The original class is the *superclass* and the new class is the *subclass*. The subclass can be modified by adding new state variables, adding new methods, and/or changing existing methods. When creating a subclass, a developer need only specify the differences from the superclass.

Traditional software uses procedures and functions as the major abstraction mechanisms. The focus of abstraction is on the actions taken, the procedure, rather than the data that is acted on. A traditional module is a procedure or function, and most of the unit-level metrics have been developed with the assumption that a module is one procedure or function. Control flow measurement is based on the flowgraph model, which is appropriate for one procedure (McCabe[76], Fenton[86], Zuse[91]). A traditional system is a collection of procedures and functions.

Metrics developed for traditional software can be applied to object-based and object-oriented software. Chappell, Henry and Mayo developed a measurement tool to take a set of traditional measurements on object-based Ada code (Chappell *et al.*[90]). The metrics included in the tool are lines of code, Halstead metrics (Halstead[77]), cyclomatic complexity (McCabe[76]), review complexity (Woodfield[80]), and information flow (Henry[81]). The authors found the tool useful for evaluating programming style. However, they were unable to measure factors related to abstract data types as a unit, since the metrics treat procedures as a unit.

Tegarden, Sheets, and Monarchi applied a similar set of traditional metrics to object-oriented software (Tegarden *et al.*[92]). The metrics used were lines of code, a set of Halstead metrics, and cyclomatic complexity. They applied the metrics to one example of an object-oriented system implemented in C++. They demonstrated that the use of polymorphism through operator overloading and/or inheritance can decrease many of the metrics. The authors are not confident about the magnitude of the metrics, but feel that the ordering implied by the metrics is accurate. Thus, the metrics appear to be on an ordinal scale and not an interval or ratio scale. They also seek metrics that focus on attributes unique to object-oriented systems.

Metrics are generally based on structural components and *levels* of a software system. Tegarden, Sheetz, and Monarchi identify a set of levels for object-oriented metrics (Tegarden *et al.*[93]). The levels are

1. variable level,
2. method level,
3. object level, and
4. system level.

In this chapter a similar set of levels is used. However, since the focus is on larger software components, the variable levels are not addressed. Since objects are dynamic rather than static program entities, the generic term *unit level* or *class level* rather than *object level* referred to by (Tegarden *et al.*[93]) is used. Also, *unit level* can refer to units such as *packages* or *modules* in object-based languages such as Ada or Modula-2. In (Tegarden *et al.*[93]) inheritance issues are addressed at each level. In this chapter inheritance is treated as an independent issue (and level). Thus, the focus is on measurement for the *method-level*, *class* or *unit-level*, *systems-* or *integration-level*, and *inheritance-structure* of an object-oriented or object-based system. At each level, various models may be useful for deriving metrics.

Method-level. The method-level refers to the defined operations of a class. This is the level that most resembles procedure or function *modules* of traditional software. As a result, measurements used for traditional software are most applicable at this level. At this level, the control flowgraph model can be used to derive control complexity metrics, and a model of a method as a set of program slices can be used to derive metrics of cohesion (Ott[93]).

Class-level or unit-level. The unit of an object-oriented system is the class or abstract data type, while the unit of a procedural system is a procedure or function. A procedure or function corresponds to a class method, and methods are really sub-units in a class. Thus, most of the traditional unit-level measurements can be applied only to individual methods. Control flowgraphs are intuitive abstractions for describing and measuring properties concerning the control flow in a method. To characterize the structure of an abstract data type, we need an abstraction that combines the methods and data structure of an ADT in an intuitively satisfying way. But, there is no obvious abstraction for describing properties

of an abstract data type. A core problem in designing metrics for object-oriented software is determining appropriate unit-level abstractions. Sets can be an initial model of the Class level, where a class is considered a set of methods and instance variables.

System-level. Each program unit (class, package, module, etc) in an object-based or object-oriented system must hide details from other program units to preserve the information hiding properties of abstract data types. Abstractions that capture the name space of a system will prove useful to measure and will be helpful in analyzing information hiding attributes. Message passing determines the control and data connections between program units (objects). Call graphs are appropriate abstractions of the control flow between units for both object-oriented and traditional software. However, for object-oriented software, call graph edges represent messages rather than invocations. Booch diagrams are appropriate abstractions for object oriented designs (Booch[91]). The key difference between traditional and object-oriented systems here is the use of messages rather than invocations. Dynamic scoping in many object-oriented languages can make the derivation of a call graph at compile time difficult. Here we can model a system as a set of class inheritance hierarchies, and we may also include message targets of classes as sets of relations.

Inheritance Structure. Inheritance is unique to object-oriented systems. The class, superclass, subclass hierarchy can be represented by a class hierarchy graph. Figure 6.1 shows an example class hierarchy graph from a object-oriented data base of university records.

The inheritance graph abstraction can be used to describe some object-oriented software attributes and derive their metrics. (Note that many object-oriented languages, such as *C++* support *multiple inheritance*, and so one class may have several superclasses.)

Models for deriving OO Metrics

For class or unit level metrics we can represent a class as a set of methods and instance variables. We can represent methods using slices and control flowgraphs. A system can be viewed as a tuple including a set of classes, a set of inheritance relations between classes, and a set of message passing relations. The most common way to view the inheritance relations is through an inheritance

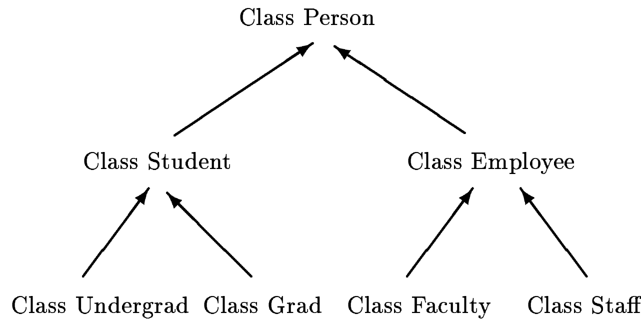

Figure 6.1 *Inheritance hierarchy graph abstraction*

tree or graph, but such a graph can be generated through a set of inheritance relations. However, some measurable attributes may require the use of several of these models. For example, a metric that indicates the number of inherited methods may be derived. Such a metric depends on both the set of inheritance relations and the names of the actual methods in each class.

Choosing the right model to use for deriving a metric depends on matching the attribute of interest to the necessary information represented in the model. Inheritance and the use of classes rather than procedures as the units of program development are the constructs that most distinguish object-oriented software from procedural software. Thus, the focus is on measurements of properties of classes and inheritance, and now we look at how object-oriented metrics have been developed (both with and without using explicit models).

6.3 Object Oriented Measurement Research

Research efforts can be evaluated in terms of how constructs unique to object-oriented software are addressed. Key concepts are unit-level measurement, inheritance, and interconnection issues such as information hiding and message passing. A key issue in deriving metrics appropriate for object-oriented software is defining abstractions or models of object-oriented software properties.

Gannon, Katz, and Basili look at metrics for object-based Ada software (Gannon *et al.*[86]). They propose simple and *more elaborate* package metrics. The simple metrics are counts of the number

of declared packages, generic packages, and instantiations of generic packages. (An instantiation of a generic package is equivalent to the declaration of a non-generic package.) A *component access metric* indicates the relative number of components of non-local data objects that are accessed in a package unit. It is calculated as the ratio of the number of non-local (declared in an external package) record fields that are accessed by a package to the package length (measured in lines of code). Here, a package is modeled as a set of record fields. The authors show via a case study that this metric is related to the difficulty of changing a package. The component access metric indicates a form of coupling. Thus a relationship with the difficulty of modifying a package is expected. The component access metric also indicates reuse; we see that there is a connection here between reuse and coupling.

Another metric proposed by Gannon, Katz, and Basili is a *package visibility metric*. This metric is based on counts of the number of

1. package units that access information in a package (*Used*)
2. units where the package is visible (*Current*),
3. units where a package could be made visible by adding a `with` clause (*Available*), and
4. units where the package are visible after moving `with` statements to the most local subunits (*Proposed*).

Package bodies are not included in the calculations. *Visibility* ratios can be computed that compare the actual visibility with the highest possible visibility, for example, *Used/Available (UA)*) and *Proposed/ Current (PC)*. These ratio measurements range in value from 0 to 1 and can indicate the level of information hiding. A high *UA(P)* for package P indicates that P is accessed in most of the units that could access P. A low *UA(P)* indicates that few of the units with access to P actually use P. A high *PC(P)* indicates that P must be visible (based on system structure) to (almost) only the units that actually access P. Since visibility is a key concept in object-based (and object-oriented) software these ratio metrics should prove useful. For these visibility metrics, we might think that sets of packages is the model used for measure derivation. However, Ada allows unbounded nesting of packages, and this nesting complicates a model based on sets.

Sheetz, Tegarden, and Monarchi derive a set of primitive counts

of attributes of object-oriented code at several levels (Sheetz *et al.*[91]) oriented software are those related to the class level (called the object-level by (Sheetz *et al*[91], Tegarden *et al.*[93])) and inheritance structure. All of these metrics can be derived from a model of an object-oriented system as a tuple with a set of classes, a set of inheritance relations, and a reference set for each class. The metrics at these two levels that are clearly defined include:

Class level metrics:

- used-by/uses: number of classes [using /used by] a class.
- fan-in/fan-out: number of unique messages [received/sent] by a class.
- fan-down: number of subclasses of a class
- fan-up: number of superclasses of a class
- class-to-root depth: maximum number of levels above a class in the inheritance hierarchy
- class-to-leaf depth: maximum number of levels below a class in the inheritance hierarchy
- local-to-parent conflicts: number of properties defined in a class with the same name as an inherited property.
- parent-to-parent conflicts: number of properties defined in multiple parents of a class with the same name.

Inheritance structure metrics:

- max depth of hierarchy: number of levels from the root to the leaf that is at the greatest distance.
- max breadth of hierarchy: maximum number of classes at any one level.
- inheritance links: number of inheritance links.

The authors provide a model of *complexity* based on the metrics, and some intuition concerning the relationship between the metrics and coupling and cohesion; they use this model to combine the primitive metrics into composites.

Lake and Cook developed a tool that counts various attributes in C++ programs (Lake-Cook[92]). The tool can generate the inheritance hierarchy tree and indicate the number of classes at each level, the depth of a class in the tree, and the number of sub-classes. They use the tool in an experiment to determine the relationship

between class depth in the inheritance tree and the ability of programmers to perform maintenance tasks. The results indicate that it is easier for programmers to perform tasks on classes at or near the root of the inheritance hierarchy tree. These results are preliminary, since the study included only eleven subjects.

Chidamber and Kemerer report efforts on developing structural metrics for object-oriented design (Chidamber-Kemerer[91]). They develop a set of metrics at the class level including:

1. a size metric based on the number of methods per class;
2. the depth of inheritance of a class;
3. the number of children;
4. non-inheritance couples with other classes;
5. the number of methods that can be invoked by a class;
6. the non-cohesion of a class based on the number of non-shared instance variables.

Chidamber and Kemerer apply these metrics to two software systems, one system implemented in C++ and one implemented in Smalltalk (Chidamber-Kemerer[92]). Of interest in these measurements is the fact that inheritance was not greatly used in these systems. The median depth of inheritance is only 1 in the C++ system, and 3 in the Smalltalk system. (Smalltalk requires a higher depth of inheritance since all classes are subclasses of a base class.)

Li and Henry add additional metrics to those developed by Chidamber and Kemerer (Li-Henry[93]). They add a class level metric of coupling through message passing defined as the number of send-statements defined in a class and coupling through an abstract data type defined as the number of abstract data types defined in a class. They also define several class size metrics including the number of local methods and the number of semicolons (clearly a language-dependent metric). They also show that the metrics can be statistically related to maintenance effort.

The foregoing is just a sample of some of the current efforts to define metrics for object-oriented software. Some common software measurement research problems still need to be resolved (Melton *et al.*[90], Baker *et al.*[90]). Zuse and Fetcke (Zuse-Fetcke[95]) provide some new insights into how to validate scale factors for object-oriented measures.

Several of the proposed object-oriented metrics (1) combine metrics of different attributes into composites, thus losing sensitivity,

(2) adjust simple counts with *weights*, making it difficult or impossible for them to be on an interval or ratio scale, and (2) confuse measurement systems with prediction systems. However, new insights into object-oriented software measurement are provided.

6.4 The OO Reuse Measurement Research Effort

As an example of current research on object-oriented measurement, the reuse metrics project at Colorado State University (CSU) is described. Proponents assert that a major benefit of object-oriented or object-based design and programming is the generation of reusable software components (Meyer[87]). Components can be reused as is, or modified using subclassing facilities. Henderson-Sellers provides insights into the differences between process metrics for object-oriented software development and such metrics for traditional development (Henderson-Sellers[91,92]). These differences result primarily from effects of the reuse of code on process prediction models.

To support or refute claims that object-oriented or object-based software promotes software reuse and to adequately evaluate the effects of reuse on process models, one must be able to measure reuse in these systems. Current reuse metrics are not directed toward the object-oriented approach. Thus, there is a strong motivation to develop metrics to quantify reuse in object-oriented and object-based software.

New definitions of attributes, abstractions and metrics that support data abstraction, information hiding and inheritance constructs are needed to measure reuse in object-oriented systems. A research group at CSU is studying this problem. (Bieman[91]) defines the classes of software reuse, identifies important perspectives of reuse, proposes relevant reuse abstractions, and suggests reuse attributes and associated metrics applicable to object-oriented systems. In (Karunanithi-Bieman[93a]), these definitions and metrics are extended with a focus on Ada object-based software which is of primary interest to the project's sponsors (NASA, CTA, and the Colorado Advanced Software Institute).

6.4.1 Object Oriented Reuse Classifications

Reuse can be classified in one of the following ways: *public/private*, *verbatim/generic/leveraged*, and *direct/indirect*. Public reuse is

reuse of externally constructed software while private reuse is reuse of software within a product (Fenton[91]). Verbatim reuse is reuse without modifications. Leveraged reuse is reuse with modifications. These modifications can be either ad hoc modifications (modifications not supported by the programming language) or modifications with some language support. Generic reuse is reuse of generic packages. Generics are simply templates for packages or subprograms. They are general versions of processes that can be modified by parameters at compilation time. Direct reuse is reuse without going through an intermediate entity. Indirect reuse is reuse through an intermediate entity. The level of indirection is the number of intermediate entities between a client and a server. There may be different possible intermediate entities connecting a client and a server.

Object-oriented languages support reuse in the following ways:

- verbatim reuse through instantiation and use of previously defined classes,
- generic reuse through generic templates, and
- leveraged reuse through inheritance.

Leveraged reuse applies to reuse with any type of modification. Since it is difficult to measure ad hoc leveraged reuse, we consider only leveraged reuse for modifications with some language support. Object-oriented support of leveraged reuse via inheritance provides an enhanced ability to analyze and measure leveraged reuse.

6.4.2 Perspectives and Reuse Measurement

Different reuse attributes are visible when reuse is examined from different perspectives. Consider a system where individual modules access some set of existing software entities. When module M uses program unit S, M is a client and S is a server. A program unit being reused is considered a server and the unit accessing that program unit is considered a client. Reuse can be observed from the perspectives of the server, the client, and the system. Each of these perspectives is relevant for the analysis and measurement of reuse in a system. A set of potentially measurable attributes can be derived based on profiles of reuse from each perspective.

In object-oriented systems, reuse is not restricted to modules. A class can reuse another class, a global module (or subprogram), a local module, and a class module can reuse another class module.

When a class reuses another class, it can inherit from another class, instantiate a generic class, and/or use another class. Thus, metrics of the number of servers reused, the number of times a server is reused, the number of clients for a server, size of each server and each client, etc. are important. Size for the relevant attributes can be determined by source lines of code, number of bytes, or any relevant metric. Again, a client can reuse a server either directly or indirectly. Hence, the number of indirect servers, indirect clients, levels of indirection, etc., can be measured. All of these types of reuse can be measured from each of the three perspectives. Using each perspective, reuse can be categorized as either verbatim, generic, or leveraged. As a result, numerous measurable attributes can be defined. Because of the numerous potential reuse metrics, they are best presented in a tabular fashion. Table 6.1 and Table 6.2 from (Karunanithi-Bieman[93b] and Bieman-Karunanithi[95]) show the set of potential reuse metrics from the client perspective and the server perspective.

Prototype tools to collect a set of metrics for both Ada and C++ software have been developed. The Ada Reuse Metrics Analyzer (ARMA) (Karunanithi-Bieman[93b], Bieman-Karunanithi[95]) is based on the semantic analysis system in the *Anna-I* tool (Mendal[92]). ARMA generates counts of the number of packages imported and the number of times the packages and their components are referenced. The information is generated from both the client and server perspective, and can be examined from various levels of abstraction. ARMA produces a *reuse data representation* of an Ada system using an *internal forest of trees* that contains the information necessary to produce the primitive metrics. An example internal forest for an Ada system is shown in Figure 6.2.

The internal forest representation tracks the names and the number of uses of servers and clients for each package and subpackage. Developers can use the reuse data representation to produce customized reports to satisfy a wide range of measurement goals.

The CSU research group used ARMA to measure primitive reuse attributes for data sets of Ada software provided by CTA, Inc., and it also showed that ARMA can be used to generate a set of component access and package visibility metrics.

The CSU research group has also developed tools to analyze C++ programs. One tool, Jasmin, generates information concerning the inheritance hierarchy. Jamin was used to measure the use of inheritance in more than 50 Mbytes of software from 19 C++

	Candidate Metrics	Definition
	Leveraged	**Inheritance relationship metric**
1.	# Direct server classes	# direct superclasses.
2.	# Indirect server classes	# classes that direct servers have Using, Instantiation and Inheritance indirect relationships.
3.	# Indirect parent servers	# indirect superclasses for client.
4.	# Direct server methods inherited	# methods from server class available for client.
5.	# Direct server methods extended	# methods in client extended from corresponding server methods.
6.	# Direct server methods overridden	# methods in client overridding corresponding server methods.
7.	# Direct server overloaded methods	# methods in client overloading server methods.
8.	Size: each direct server method that is reused / extended	
9.	Size: direct server interface	
10.	Size: global defs in server interface	
11.	Size: global defs in server body	
12.	Size: each client method	
13.	Size: client interface	
14.	Size: global defs in client interface	
15.	Size: global defs in client body	
16.	# Paths to indirect servers	# paths connecting client and indirect servers.
17.	Length of paths to indirect servers	# edges in a path connecting client and indirect servers.
18.	# Paths to indirect parent servers	# paths connecting client and indirect parent servers.
19.	Length of paths to indirect parent servers	# edges in a path connecting client and indirect parent servers.

Table 6.1 *Leveraged reuse metrics from client perspective*

	Candidate Metrics	Definition
	Leveraged	**Inheritance Relationship metric**
1.	# Direct clients	# direct subclasses.
2.	# Indirect clients	# classes with Using, Instantiation, and Inheritance indirect relationships with direct clients.
3.	# Indirect child clients	# clients that have inherited indirectly from server.
4.	# Client invocations of server method	# times a method in server is reused in all all its clients.
5.	Size of server methods	
6.	Size of server interface	
7.	Size of global definitions in in server interface	
8.	Size of global definitions in in server body	
9.	Paths to indirect clients	# paths connecting server and clients.
10.	Length of paths to indirect clients	# edges in a path connecting client and server.
11.	Paths to indirect child clients	# paths connecting server. and child clients.
12.	Length of paths to indirect child clients	# edges in a path connecting a server and child client.

Table 6.2 *Leveraged reuse metrics from server perspective*

systems with a total of more than 265,000 non-commented source lines of code (NCSL) and 2,744 separate C++ classes (Bieman-Zhao[95]). Although inheritance is used extensively in many of the measured systems, it is used very infrequently in others. The average depth of inheritance for classes was less than 1.0 in half of the systems. The greatest use of inheritance is in the GUI applications, perhaps because GUI applications tend to model a hierarchical world of user interface objects such as icons and windows. Reuse library classes use inheritance much more than the applications system classes. The measurements also indicate that there are many classes that exist independently with no parent or child classes.

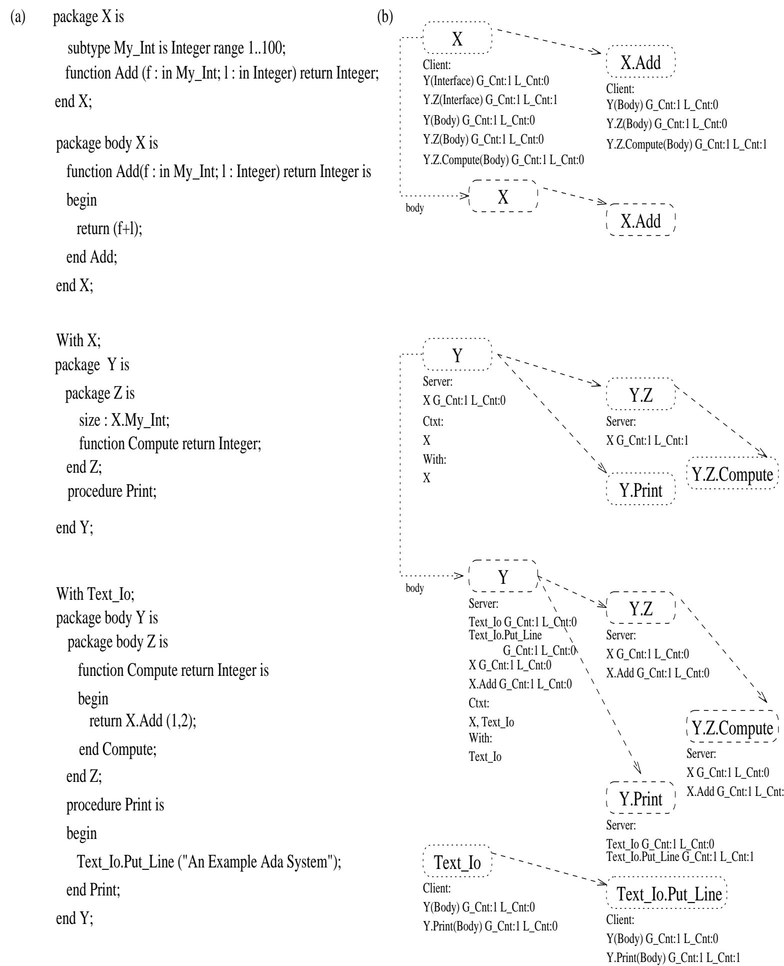

Figure 6.2 *An Ada System (a) and its internal representation (b)*

Figure 6.3 from (Bieman-Zhao[95]) shows that few of the 19 measured systems have the 7 ± 2 maximum class inheritance depth recommended by Booch (Booch[94]). Two of the systems exceed Booch's maximum, while most of the systems never reach this maximum. We did not measure the maximum width of individual inheritance trees; however, 32% of the systems had classes with more

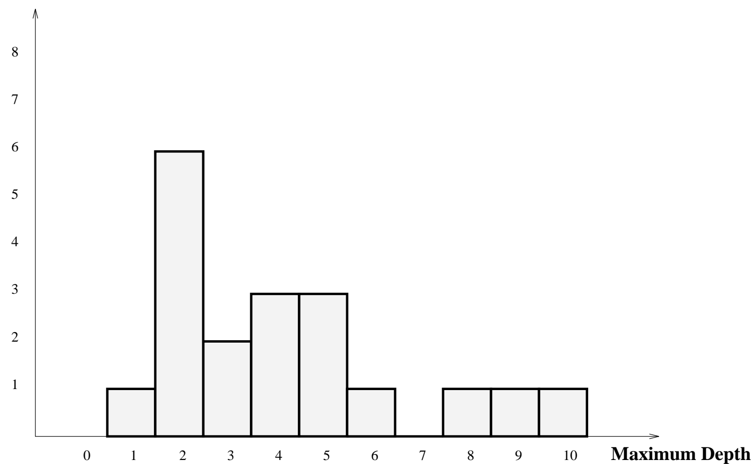

Figure 6.3 *Maximum Class Inheritance Depths in 19 C++ Systems*

than 16 children. We also found that multiple inheritance is not used very often.

The relationship between class cohesion and private reuse was also studied by the CSU group (Bieman-Kang[95]). To do this, sensitive class cohesion measures were developed. Cohesion refers to the "relatedness" of module components. Method and instance variable class components are treated as the key class units that may or may not be connected. Two methods are related (connected) through instance variable(s) if both methods use the instance variable(s). Using this orientation, class cohesion can be measured by the relative connectivity (through instance variables) of the methods.

Two measures of class cohesion are defined based on the direct and indirect connections of method pairs. Let $NP(C)$ be the total number of pairs of visible methods in a class C, NP be the maximum possible number of direct or indirect connections in a class, $NDC(C)$ be the number of direct connections, and $NIC(C)$ be the number of indirect connections in class C. The two measures are *Tight class cohesion* and *Loose class cohesion*:

1. *Tight class cohesion* (TCC) is the relative number of directly

connected methods:

$$\mathsf{TCC}(C) = NDC(C)/NP(C)$$

2. *Loose class cohesion* (LCC) is the relative number of directly or indirectly connected methods:

$$\mathsf{LCC}(C) = (NDC(C) + NIC(C))/NP(C)$$

The CSU research group applied the class cohesion measures to the InterViews system, a reasonably large C++ system developed at Stanford University. It consists of more than 25,000 noncommented lines of code. 14% of the classes in InterViews do not have any methods; these classes were excluded from our measurements. Also removed were all virtual methods with empty bodies. Only local cohesion was measured—inherited methods were not included in the measurement. Reuse was measured by counting the number of descendents of a particular class.

No relationship was found between class cohesion and instantiation reuse in the InterViews system. However, significant relationships were found between cohesion and inheritance reuse. Figure 6.4 from (Bieman-Kang[95]) shows the relationship between the number of descendents and local class cohesion. Average values of tight class cohesion and loose class cohesion are provided for four different categories based on the number of descendents. Figure 6.4 clearly shows that the classes that are reused more frequently exhibit lower cohesion. It was also found that this relationship holds generally for all levels of depth in the inheritance hierarchy. A T-test and the Wilcoxon rank-sum test were used to evaluate the significance of the results. A T-test can be used for data with a normal distribution and an interval scale, and the Wilcoxon rank-sum test can be used if there is a question concerning distributions or if the data is ordinal. Both tests shows that the relationship we see in Figure 6.4 is significant (to the .05 level) and not due to chance.

Although, the most cohesive classes in InterViews tend to have fewer descendents, most of the classes are quite cohesive. The mean TCC is 0.75 and median is 1.0; the LCC mean is 0.8 and median is 1.0. Thus, most of the pairs of methods in most of the classes are connected.

An alternative approach for measuring cohesion is based on generating "slices" for each instance variable over all class methods

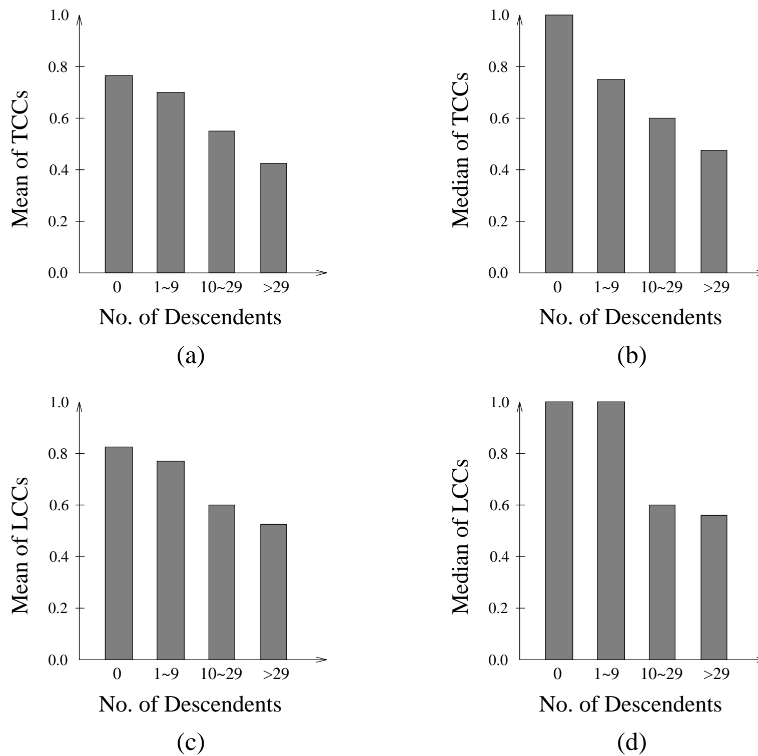

Figure 6.4 *Number of descendents and Class Cohesion in InterViews*

(Ott *et al.*[95]). Then the similarity of the slices is used to determine cohesion. Two measures are proposed, *strong data cohesion* and *weak data cohesion*, following a derivation similar to that used by Bieman and Ott to develop functional cohesion measures (Bieman-Ott[94]). This alternative approach is now under evaluation.

6.5 Quantifying Structural Style

We see that there are many metrics that can be used to characterize object-oriented systems. The correct metrics to use depend on the software attributes of interest. Rombach can help determine the appropriate metrics for a particular purpose (Basili-Rombach[88]). The GQM paradigm requires that software measurements be tied

to a software engineering *goal*. For example, the goal might be how to characterize reuse in an object-oriented system. A goal leads to *questions*. For example, "What structures are used to facilitate reuse?" The answers to such questions lead to *metrics*. In the reuse example, metrics can indicate the number of times that reuse structures appear in a software artifact.

Many software metrics have been derived to quantify attributes of the traditional software unit, the procedure or function. Yet in an object-oriented or object-based system, the software unit is a class or abstract data type. If we want to answer questions about common concerns such as control flow complexity and cohesion in object-oriented or object-based systems, we need to redefine our metrics so that they can apply to classes and abstract data types.

It would be really good to use measurement to help indicate the structural quality of a system. For example, inheritance is supposed to be a useful tool to support reuse. Thus the inheritance graph structure should help indicate how inheritance supports reuse. Metrics that indicate the structure of the inheritance hierarchy should prove useful. For example, potential metrics include the maximum depth or width and other attributes of an inheritance hierarchy graph. The structure of an inheritance hierarchy graph can be used to indicate the quality of a software design. However, first we need to determine what is a good way to use inheritance and what kinds of inheritance structures should be avoided.

Good and bad structures can be determined through experience and through more formal experimentation. Lake and Cook use experimentation to show the relationship between the size of inheritance trees and the comprehension of C++ programs (Lake-Cook[92]). Li and Henry also find that many structural attributes are related to the effort to maintain a system (Li-Henry[93]). They show that size alone is not enough to predict the number of changes.

The component access metric and the visibility metrics of Gannon, Katz, and Basili quantify some aspects of programming style (Gannon *et al.*[86]). The component access metric indicates the non-local coupling, and the visibility metrics indicate the amount of unnecessary visibility. Both coupling and visibility should be minimized according to style rules.

Software development experience can provide insights about desirable software structure. Lieberherr, Holland and Riel suggest that limiting references to non-local objects will ease modification

effort (Lieberherr *et al.*[88]). They define an object-oriented software design style, the *Law of Demeter*:

> For all classes C, and for all methods M attached to C, all objects to which M sends a message must be instances of classes associated with the following classes:
>
> 1. The argument classes of M (including C).
> 2. The instance variable classes of C.
>
> (Objects created by M, or by functions or methods which M calls, and objects in global variables are considered as arguments of M.)

Lieberherr *et al.* argue that by following the Law of Demeter a developer can control coupling, control access to information, narrow interfaces, and facilitate semantic analysis. These arguments are intuitive, and we can use metrics to provide empirical support. We can design measurement tools to indicate the classes that violate the Law of Demeter (and corollaries described by (Lieberherr, Holland, Riel[88])), and then empirically validate the claims. We can also use metrics that indicate violations of the law as a mechanism to identify classes that need inspection or redesign.

In his recent book, Cargill provides a set of guidelines for good C++ programming style (Cargill[92]). Included in his numerous style recommendations are that developers should limit polymorphism and inheritance, design consistent structures, and reduce coupling. Many of the recommendations can be used to derive structural metrics.

With structural style metrics, we can empirically determine the benefit of alternative design styles. Once we really understand the structural styles that promote good quality — reliability, maintainability, reusability, etc — we can use measurement to ensure that development is consistent with desired structural style.

6.6 Conclusions

The utility of software development paradigms, such as object-oriented or object-based programming, design, and analysis can be evaluated through quantitative studies. Measurement tools developed specifically for the special characteristics of object-oriented software are needed.

The features that are unique to object-based systems include reliance on data abstraction and generic modules. Object-oriented

software also uses inheritance and message passing. We need software metrics that treat a class or abstract data type rather than a procedure or function as a basic unit. We also need software metrics that quantify attributes of inheritance and inheritance hierarchies.

Researchers have derived metrics to quantify the amount of nonlocal references, the visibility of components, inter-class dependencies, the inheritance structure, and the structure of individual classes. Metrics of interest focus on program units — packages and classes, and inheritance hierarchies.

The most general model for deriving metrics appropriate for object-oriented systems is a tuple that includes a set of classes, a set of inheritance relations between classes, and a set of class relations of interest. Classes must also be modeled as sets of methods and instance variables. More detailed models may be necessary to derive specific metrics, and nested units, such as Ada packages, make the use of sets to model systems more difficult.

There is no single metric to indicate the amount of reuse in a system. Rather, a vector of metrics is needed to quantify various attributes of reuse. In a similar manner, other researchers find many ways to quantify structural attributes of object-oriented and object-based software.

Having numerous metrics to quantify the various attributes of a system provides flexibility to those analyzing a system. A developer can use the metric or combination of metrics that are appropriate to answer specific questions about a system. Questions about reuse, the use of inheritance, and visibility can be answered by selecting the particular metrics that quantify relevant attributes.

We can use measurement to evaluate development style. Intuition about good structural style can be formalized into quantitative metrics, and then we can determine if a system is consistent with the desired style.

Future research ought to focus on quantitative evaluations of structural style, and the relation between style and (hopefully quantifiable) attributes such as reuse, reliability, and maintainability. We also need better models for deriving object-oriented metrics — models that match the unique characteristics of object-oriented software.

References

[Note] **References for all chapters were placed in one list at the end of the book. The following list of references should match the references in the chapter. However some references may have been updated or added, and a different labelling scheme is used.**

[BBF+90] A.L. Baker, J.M. Bieman, N. E. Fenton, A.C. Melton, and R.W. Whitty. A philosophy for software measurement. *Journal of Systems and Software*, 12(3):277–281, July 1990.

[Bie92] J.M. Bieman. Deriving measures of software reuse in object-oriented systems. In T. Denvir, R. Herman, and R. Whitty, editors, *Formal Aspects of Measurement. (Proc. BCS-FACS Workshop on Formal Aspects of Measurement)*, pages 79–82. Springer-Verlag, 1992.

[BO93] J.M. Bieman and L.M. Ott. Measuring functional cohesion. *IEEE Trans. Software Engineering*, August 1994.

[Boo91] G. Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings, Redwood City, CA, 1991.

[BR88] V.R. Basili and H.D. Rombach. The tame project: Towards improvement-oriented software environments. *IEEE Trans. Software Engineering*, SE-14(6):758–773, June 1988.

[Car92] T. Cargill. *C++ Programming Style*. Addison-Wesley, Reading, MA, 1992.

[CHM90] B.L. Chappell, S. Henry, and K.A. Mayo. Measurement of Ada throughout the software development life cycle. *Proc. 8th Conf. on Ada Technology*, pages 525–531, 1990.

[CK91a] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. Technical Report CISR WP No. 249, Sloan WP No. 352-93, Center for Information Systems Research, Sloan School of Management, MIT, Cambridge, MA, 1991.

[CK91b] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. *Proc. OOPSLA'91*, pages 197–211, October 1991.

[Cox90a] B. J. Cox. Planning the software industrial revolution. *IEEE Software*, 7(6):25–33, November 1990.

[Cox90b] B. J. Cox. There is a silver bullet. *Byte*, 15(10):209, October

1990.

[Fen91] N. Fenton. *Software Metrics - A Rigorous Approach.* Chapman and Hall, London, 1991.

[FM90] N. Fenton and A. Melton. Deriving structurally based software measures. *Journal of Systems and Software*, 12(3):177–187, July 1990.

[FW86] N. E. Fenton and R. W. Whitty. Axiomatic approach to software metrication through program decompostion. *The Computer Journal*, 29(4):329–339, 1986.

[GKB86] J.D. Gannon, E.E. Katz, and V.R. Basili. Metrics for Ada packages: An initial study. *Communications of the ACM*, 29(7):616–623, July 1986.

[Hal77] M. H. Halstead. *Elements of Software Science.* Elsevier, New York, 1977.

[HB85] J. W. Howatt and A. L. Baker. Definition and design of a tool for program control structure measures. *Proc. of the IEEE Computer Society's Ninth International Computer Software & Applications Conference (COMPSAC85)*, pages 214–220, 1985.

[HB89] J. Howatt and A. Baker. Rigorous definition and analysis of program complexity measures: An example using nesting. *The Journal of Systems and Software*, 10(2):139–150, 1989.

[HK81] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Trans. Software Engineering*, SE-7(5):510–518, 1981.

[HS91] B. Henderson-Sellers. Some metrics for object-oriented software engineering. *Proc. TOOLs Pacific '91 (Technology of Object-Oriented Languages and Systems*, pages 131–139, 1991.

[HS92] B. Henderson-Sellers. Managing and measuring object-oriented information systems development. *Proc. Workshop on Object-Oriented Software Engineering Practice*, February 1992.

[IS88] D. C. Ince and M. J. Sheppard. System design metrics: A review and perspective. *Proc. 2nd IEE/BCS Conf. on Software Engineering*, pages 23–27, 1988.

[KB93a] S. Karunanithi and J.M. Bieman. Candidate reuse metrics for object oriented and Ada software. *IEEE-CS Int. Symp. Software Metrics*, 1993.

[KB93b] S. Karunanithi and J.M. Bieman. Measuring software reuse in object oriented systems and ada software. Technical Report CS–93–125, Computer Science Dept., Colorado State Univ., Fort Collins, CO, 1993.

[LC92] A. Lake and C. Cook. A software complexity metric for C++. Technical Report 92-60-03, Computer Science Dept., Oregon State Unviersity, Corvallis, Oregon, 1992.

[LH93] W. Lei and S. Henry. Maintenance metrics for the object oriented paradigm. *Proc. IEEE-CS Int Software Metrics Symp.*, page to

appear, May 1993.

[LHR88] K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: An objective sense of style. *Proc. OOPSLA'88*, pages 323–334, September 1988.

[McC76] T. J. McCabe. A complexity measure. *IEEE Trans. Software Engineering*, SE-2(4):308–320, 1976.

[Men92] G. Mendal. *The Anna-I User's Guide and Installation Manual version 1.4 edition.* Stanford University, Computer Systems Lab, Stanford, CA, 1992.

[Mey87] B. Meyer. Reusability: The case for object oriented design. *IEEE Software*, 4(2):50–64, March 1987.

[MGBB90] A.C. Melton, D.A. Gustafson, J.M. Bieman, and A.L. Baker. A mathematical perspective for software measures research. *Software Engineering Journal*, 5(5):246–254, 1990.

[MK91] M. Myers and A. Kaposi. Modelling and measurement of Prolog data. *Software Engineering Journal*, pages 413–434, November 1991.

[OB92] Linda M. Ott and James M. Bieman. Effects of software changes on module cohesion. *Proc. IEEE/ACM Conf. on Software Maintenance*, pages 345–353, November 1992.

[Ott92] L. M. Ott. Using slice profiles and metrics during software maintenance. *Proc. 10th Annual Software Reliability Symp*, pages 16–23, June 1992.

[Ott93] L. M. Ott and J.J. Thuss. Slice based metrics for estimating cohesion. *Proc. 1993 IEEE-CS Int. Software Metrics Symp.*, pages 71–81, May 1993.

[STM91] S. D. Sheetz, D. P. Tegarden, and D. E. Monarchi. Measuring object-oriented system complexity. *Proc. 1st Workshop on Information Technologies and Systems*, December 1991.

[TSM92] D. P. Tegarden, S. D. Sheetz, and D. E. Monarchi. Effectiveness of traditional software metrics for object-oriented systems. *Proc. 25th Hawaii Int. Conf. Systems Sciences (HICSS-25)*, January 1992.

[TSMar] D. P. Tegarden, S. D. Sheetz, and D. E. Monarchi. A software complexity model of object-oriented systems. *Decision Support Systems: The International Journal*, to appear.

[Whi89] R. Whitty. Structural metrics for Z specifications. *Proc. 4th Z User Meeting*, pages 186–191, December 1989.

[Woo80] S.N. Woodfield. *Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors.* PhD thesis, Purdue University, W. Lafayette, IN, 1980.

[Woo93] M. R. Woodward. Difficulties using cohesion and coupling as quality indicators. *Software Quality Journal*, 2(2):109–127, June 1993.

[Zus91] H. Zuse. *Software Complexity Measures and Methods.* W. de Gruyter, Berlin, 1991.