National Research Council Canada
Conseil national de recherches Canada
Institute for Information Technology
Institut de Technologie de l'information

ERB-1085

NRC·CNRC

Object-Oriented Metrics: A Review of Theory and Practice

El-Emam, K.
March 2001

Canada

NRC 44190

National Research Council Canada
Institute for Information Technology

Conseil national de recherches Canada
Institut de Technologie de l'information

Object-Oriented Metrics: A Review of Theory and Practice

El-Emam, K.
March 2001

Object-Oriented Metrics: A Review of Theory and Practice

Khaled El Emam

1.1 Introduction

In today's business environment, competitive pressures demand the production of reliable software with shorter and shorter release intervals. This is more so in commercial high reliability domains such as telecommunications and aerospace. One recipe for success is to increase process capability. There is recent compelling evidence that process capability is positively associated with productivity and quality. (Clark, 1997; El-Emam and Birk, 2000a, 2000b; Flowe and Thordahl, 1994; Goldenson and Herbsleb, 1995; Jones, 1999; Krishnan and Kellner, 1999). Quantitative management of software quality is a hallmark of high process capability (El-Emam, Drouin, and Melo, 1998; Software Engineering Institute, 1995).

Quantitative management of software quality is a broad area. In this chapter we focus on only one aspect: the use of *software product metrics* for quality management. Product metrics are objective[1] measures of the structure of software artifacts. The artifacts may be, for example, source code or analysis and design models.

The true value of product metrics comes from their association with measures of important external attributes (ISO/IEC, 1996). An external attribute is measured with respect to how the product relates to its environment (Fenton, 1991). Examples of external attributes are testability, reliability and maintainability. Practitioners, whether they are developers, managers, or quality assurance personnel, are really concerned with the external attributes. However, they cannot measure many of the external attributes directly until quite late in a project's or even a product's life cycle. Therefore, they can use product metrics as leading indicators of the external attributes that are important to them. For instance, if we know that a certain coupling metric is a good leading indicator of quality as measured in terms of the number of faults, then we can minimize

[1] Objective means that if you repeatedly measure the same software artefact (and the artefect does not change), then you will get the same values. This is because in most cases the metrics are automated. The alternative is to have subjective metrics. Subjective metrics are not covered in this chapter.

coupling during design because we know that in doing so we are also reducing rework costs.

Specifically, product metrics can be used in at least three ways: making system level predictions, early identification of *high-risk* software components, and the construction of preventative design and programming guidelines. These uses allow an organization, for instance, to get an early estimate of quality, and to take early action to reduce the number of faulty software components.

Considerable effort has been spent by the software engineering research community in developing product metrics for both procedural and object-oriented systems, and empirically establishing their relationship to measures of important external attributes. The latter is known as the *empirical validation* of the metric. Once the research community has demonstrated that a metric or set of metrics is empirically valid in a number of different contexts and systems, organizations can take these metrics and use them to build appropriate prediction models and guidelines customized to their own context.

The objective of this chapter is to provide a review of contemporary object-oriented metrics. We start by describing how object-oriented metrics can be used in practice by software organizations. This is followed by an overview of some of the most popular object-oriented metrics, and those that have been studied most extensively. The subsequent section describes current cognitive theories used in software engineering that justify the development of object-oriented metrics. This is followed by a further elaboration of the cognitive theory to explain the cognitive mechanisms for metric thresholds. The empirical evidence supporting the above theories is then reviewed. The chapter is concluded with recommendations for the practical usage of object-oriented metrics, a discussion of the match between the empirical results and the theory, and directions for future research.

1.2 The Practical Use of Object-Oriented Metrics

In this section we describe how product metrics can be used by organizations for quality control and management.

1.2.1 Making System Level Predictions

Typically, software product metrics are collected on individual components for a single system. Predictions on individual components can then be aggregated to give overall system level predictions. For example, in two recent studies using object-oriented metrics, the authors predicted the proportion of faulty classes in a whole system (El-Emam, Melo, and Machado, 2001). This is an example of using predictions of fault-proneness for each class to draw conclusions about the overall quality of a system. One can also build prediction models of the total number of faults and fault density (Evanco, 1997). Similarly, another study

used object-oriented metrics to predict the effort to develop each class, and these were then aggregated to produce an overall estimate of the whole system's development cost (Briand and Wuest, 1999).

1.2.2 Identifying High-Risk Components

The definition of a high-risk component varies depending on the context. For example, a high-risk component may be one that contains any faults found during testing (Briand, Basili, and Hetmanski, 1993; Lanubile and Visaggio, 1997), one that contains any faults found during operation (Khoshgoftaar, Allen, Jones, and Hudepohl, 1999), or one that is costly to correct after an error has been found (Almeida, Lounis, and Melo, 1998; Basili, Condon, El-Emam, Hendrick, and Melo, 1997; Briand, Thomas, and Hetmanski, 1993). Recent evidence suggests that most faults are found in only a few of a system's components (Fenton and Ohlsson, 2000; Kaaniche and Kanoun, 1996; Moller and Paulish, 1993; Ohlsson and Alberg, 1996). If these few components can be identified early, then an organization can take mitigating actions. Examples of mitigating actions include focusing defect detection activities on high-risk components by optimally allocating testing resources (W. Harrison, 1988), or redesigning components that are likely to cause field failures or be costly to maintain.

Early prediction is commonly cast as a binary classification problem.[2] This is achieved through a *quality model* that classifies components into either a high or low risk category. An overview of a quality model is shown in Figure 1.1. A quality model is developed using a statistical modeling or machine learning technique, or a combination of techniques. This is done using historical data. Once constructed, such a model takes as input the values on a set of metrics (M_1 … M_k) for a particular component, and produces a prediction of the risk category (say either high or low risk) for that component.

A number of organizations have integrated quality models and modeling techniques into their overall decision making process. For example, Lyu et al. (Lyu, Yu, Keramides, and Dalal, 1995) report on a prototype system to support developers with software quality models, and the EMERALD system is reportedly routinely used for risk assessment at Nortel (Hudepohl, Aud, Khoshgoftaar, Allen, and Mayrand, 1996a, 1996b). Ebert and Liedtke describe the application of quality models to control the quality of switching software at Alcatel (Ebert and Liedtke, 1995).

---

[2] It is not, however, *always* the case that binary classifiers are used. For example, there have been studies that predict the number of faults in individual components, for example, (Khoshgoftaar, Allen, Kalaichelvan, and Goel, 1996), and that produce point estimates of maintenance effort, for example, (Jorgensen, 1995; Li and Henry, 1993).

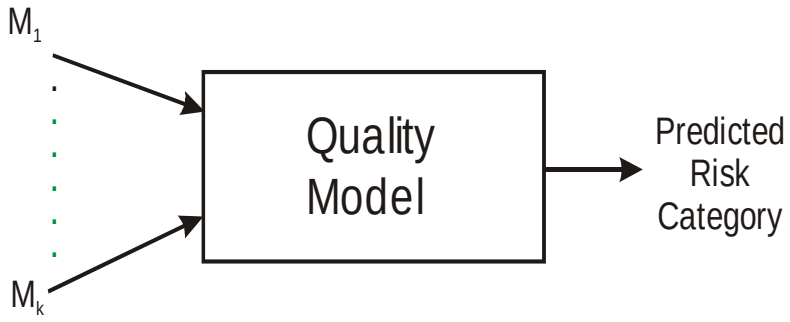

Figure 1.1. Definition of a quality model.

In the case of object-oriented metrics, an example of a quality model was presented in a recent study using design metrics on a Java application (El-Emam et al., 2001). This model was developed using logistic regression (Hosmer and Lemeshow, 1989):

$$p = \frac{1}{1 + e^{-(-3.97 + 0.464\,NAI + 1.47\,OCMEC + 1.06\,DIT)}} \qquad (2.1)$$

Here, the p value gives the predicted probability that a class will have a fault, NAI is the total number of attributes defined in the class, $OCMEC$ is the number of other classes that have methods with parameter types of this class (this is a form of export coupling), and DIT is the depth of the inheritance tree which measures how far down an inheritance hierarchy a class is. NAI, OCMEC, and DIT are examples of object-oriented metrics. In fact, in this case, all of these metrics can be easily collected from high-level designs, and therefore one can in principle use this model to predict the probability that a class will have a fault at an early stage of development. A calibration of this model, described in (El-Emam et al., 2001), indicated that if the predicted probability of a fault is greater than 0.33, then the class should be flagged for special managerial action (i.e., it is considered high risk).

The metrics in the above example are class-level static metrics. Object-oriented metrics can also be defined at the method-level or at the system-level. Our focus here is only on class-level metrics. Furthermore, metrics may be collected statically or dynamically. Static metrics can be collected by an analysis of the software artifact. Dynamic metrics require execution of the software application in order to collect the metric values, which makes them difficult to collect at early stages of the design. Our focus in this chapter is on static metrics.

1.2.3 Design and Programming Guidelines

An appealing operational approach for constructing design and programming guidelines using software product metrics is to make an analogy with conventional statistical quality control: identify the range of values that are acceptable or unacceptable, and take action for the components with unacceptable values (Kitchenham and Linkman, 1990). This means identifying thresholds on the software product metrics that delineate between *acceptable* and *unacceptable*. In summarizing their experiences using software product measures, Szentes and Gras (Szentes and Gras, 1986) state "the complexity measures of modules may serve as a useful early warning system against poorly written programs and program designs. .... Software complexity metrics can be used to pinpoint badly written program code or program designs when the values exceed predefined maxima or minima." They argue that such thresholds can be defined subjectively based on experience. In addition to being useful during development, Coallier et al. (Coallier, Mayrand, and Lague, 1999) present a number of thresholds for procedural measures that Bell Canada uses for risk assessment during the acquisition of software products. The authors note that the thresholds result in 2 to 3 percent of the procedures and classes that are flagged for manual examination. Instead of thresholds based on experience, some authors suggest the use of percentiles for this purpose. For example, Lewis and Henry (Lewis and Henry, 1989) describe a system that uses percentiles on procedural measures to identify potentially problematic procedures. Kitchenham and Linkman (Kitchenham and Linkman, 1990) suggest using the 75^{th} percentile as a cut-off value. More sophisticated approaches include identifying multiple thresholds simultaneously, such as in (Almeida et al., 1998; Basili et al., 1997).

In an object-oriented context, thresholds have been similarly defined by Lorenz and Kidd as (Lorenz and Kidd, 1994) "heuristic values used to set ranges of desirable and undesirable metric values for measured software." Henderson-Sellers (Henderson-Sellers, 1996) emphasizes the practical utility of object-oriented metric thresholds by stating that "An alarm would occur whenever the value of a specific internal metric exceeded some predetermined threshold." Lorenz and Kidd (Lorenz and Kidd, 1994) present a number of thresholds for object-oriented metrics based on their experiences with Smalltalk and C++ projects. Similarly, Rosenberg et al. (Rosenberg, Stapko, and Gallo, 1999) have developed thresholds for a number of popular object-oriented metrics that are used for quality management at NASA GSFC. French (French, 1999) describes a technique for deriving thresholds, and applies it to metrics collected from Ada95 and C++ programs. Chidamber et al. (Chidamber, Darcy, and Kemerer, 1998) state that the premise behind managerial use of object-oriented metrics is that extreme (outlying) values signal the presence of high complexity that may require management action. They then define a lower bound for thresholds at the 80^{th} percentile (i.e., at most 20% of the observations are considered to be above

the threshold). The authors note that this is consistent with the common Pareto (80/20) heuristic.

1.3 Object-Oriented Metrics

Structural properties that capture inter-connections among classes are believed to be important to measure (for example different types of *coupling* and *cohesion).* This is because they are considered to affect cognitive complexity (see next section). Object-oriented metrics measure these structural properties. Coupling metrics characterize the static usage dependencies among the classes in an object-oriented system (Briand, Daly, and Wuest, 1999). Cohesion metrics characterize the extent to which the methods and attributes of a class belong together (Briand, Daly, and Wuest, 1998). In addition, *inheritance* is also believed to play an important role in the understandability of object-oriented applications.

A considerable number of such inter-connection object-oriented metrics have been developed by the research community. For example, see (F. Brite e Abreu and Carapuca, 1994; Benlarbi and Melo, 1999; Briand, Devanbu, and Melo, 1997; Cartwright and Shepperd, 2000; Chidamber and Kemerer, 1994; Henderson-Sellers, 1996; Li and Henry, 1993; Lorenz and Kidd, 1994; Tang, Kao, and Chen, 1999). By far, the most popular of these is the metrics suite developed by Chidamber and Kemerer (Chidamber and Kemerer, 1994) (known as the CK metrics). In fact, it has been stated that for historical reasons the CK metrics are the most referenced (Briand, Arisholm, Counsell, Houdek, and Thevenod-Fosse, 1999), and most commercial metrics collection tools available at the time of writing also collect these metrics. Another comprehensive set of metrics that capture important structural characteristics, namely different types of coupling, have been defined by Briand et al. (Briand et al., 1997). We will focus our attention on these two sets of metrics since they have also received a considerable amount of empirical study. A summary of these metrics can found in Table 1.1. Many of the metrics can be collected at the design stage of the life cycle. Table 1.1 indicates which of these metrics can be accurately collected at the design phase. If the entry in the "Des" column is "Y", then the metric is typically available during design.

Even though some of the metrics can be collected at design time, in practice, they are frequently collected from the source code during validation studies. Out of the metrics set shown, only the CK metrics suite currently is known to have a number of commercial and public domain analyzers (for Java see (CodeWork, 2000; Metameta, 2000; Power-Software, 2000b), and for C++ see

(Devanbu, 2000; ObjectSoft, 2000; Power-Software, 2000a))[3]. In addition there is at least one tool that can be used to collect the CK metrics directly from design documents (Number-Six-Software, 2000).

Table 1.1. Summary of object-oriented metrics.

Metric Acronym	Des	Definition
CBO	N	This is the **Coupling Between Object Classes** coupling metric (Chidamber and Kemerer, 1994). A class is coupled with another if the methods of one class use the methods or attributes of the other, or vice versa. In this definition, uses can mean as a member type, parameter type, method local variable type or cast. CBO is the number of other classes with which a class is coupled. It includes inheritance-based coupling (i.e., coupling between classes related via inheritance). A variant of CBO, known as CBO', excludes inheritance based coupling (Chidamber and Kemerer, 1991).
RFC	N	This is the **Response for a Class** coupling metric (Chidamber and Kemerer, 1994). The response set of a class consists of the set M of methods of the class, and the set of methods invoked directly by the methods in M (i.e., the set of methods that can potentially be executed in response to a message received by that class). RFC is the number of methods in the response set of the class. A variant of RFC excludes methods indirectly invoked by a method in M (Chidamber and Kemerer, 1991).
DIT	Y	The **Depth of Inheritance Tree** (Chidamber and Kemerer, 1994) metric is defined as the length of the longest path from the class to the root in the inheritance hierarchy.
NOC	Y	This is the **Number of Children** inheritance metric (Chidamber and Kemerer, 1994). This metric counts the number of classes that inherit from a particular class (i.e., the number of classes in the inheritance tree down from a class).

… continued on the following page

[3] Note that this is not a comprehensive list of tools available on the market today. Also, please note that not all of the analyzer will collect all of the CK metrics; some only collect a subset.

Table 1.1. Summary of object-oriented metrics (continued from the previous page).

WMC	Y[4]	This is the **Weighted Methods per Class** metric (Chidamber and Kemerer, 1994), and can be classified as a traditional complexity metric. It is a count of the methods in a class. It has been suggested that neither methods from ancestor classes nor *friends* in C++ be counted (Basili, Briand, and Melo, 1996; Chidamber and Kemerer, 1995). The developers of this metric leave the weighting scheme as an implementation decision (Chidamber and Kemerer, 1994). Some authors weight it using cyclomatic complexity (Li and Henry, 1993). However, others do not adopt a weighting scheme (Basili et al., 1996; Tang et al., 1999). In general, if cyclomatic complexity is used for weighting then WMC cannot be collected at early design stages. Alternatively, if no weighting scheme is used then WMC becomes simply a size measure (the number of methods implemented in a class), also known as NM.
LCOM	N	This is a cohesion metric that was defined in (Chidamber and Kemerer, 1994). It measures the number of pairs of methods in the class that have no attributes in common, minus the number of pairs of methods that do. If the difference is negative, the metric value is set to zero.

IFCAIC
ACAIC
OCAIC
FCAEC
DCAEC
OCAEC
IFCMIC
ACMIC
OCMIC
FCMEC
DCMEC
OCMEC
OMMIC
IFMMIC
AMMIC
OMMEC
	Y
	Y
	Y
	Y
	Y
	Y
	Y
	Y
	Y
	Y
	Y
	Y
	N
	N
	N
	N
		These coupling metrics are counts of interactions among classes. The metrics distinguish among the class relationships (friendship, inheritance, none), different types of interactions, and the locus of impact of the interaction (Briand et al., 1997).
		The acronyms for the metrics indicate what types of interactions are counted:
		• The first or first two letters indicate the relationship:
		  • **A**: coupling to ancestor classes;
		  • **D**: coupling to descendents;
		  • **F**: coupling to friend classes;
		  • **IF**: inverse friend coupling; and
		  • **O**: other, (i.e., none of the above).
		• The next two letters indicate the type of interaction between classes c and d:
		  • **CA**: there is a class-attribute interaction between classes c and d if c has an attribute of type d;
		… continued on the following page

[4] Only the unweighted version of WMC is available during design. If weights are used, then this would depend on the characteristics of the weighting scheme. For example, cyclomatic complexity weights would certainly not be available during design.

Table 1.1. Summary of object-oriented metrics (continued from the previous page).

WMC	Y[5]	This is the **Weighted Methods per Class** metric (Chidamber and Kemerer, 1994), and can be classified as a traditional complexity metric. It is a count of the methods in a class. It has been suggested that neither methods from ancestor classes nor *friends* in C++ be counted (Basili, Briand, and Melo, 1996; Chidamber and Kemerer, 1995). The developers of this metric leave the weighting scheme as an implementation decision (Chidamber and Kemerer, 1994). Some authors weight it using cyclomatic complexity (Li and Henry, 1993). However, others do not adopt a weighting scheme (Basili et al., 1996; Tang et al., 1999). In general, if cyclomatic complexity is used for weighting then WMC cannot be collected at early design stages. Alternatively, if no weighting scheme is used then WMC becomes simply a size measure (the number of methods implemented in a class), also known as NM.

1.4 Cognitive Theory of Object-Oriented Metrics

A theoretical basis for developing quantitative models relating product metrics and external quality metrics has been provided in (Briand, Wuest, Ikonomovski, and Lounis, 1998), and is summarized in
Figure 1.2. This theory hypothesizes that the structural properties of a software component (such as its coupling) have an impact on its cognitive complexity. Cognitive complexity is defined as the mental burden of the individuals who have to deal with the component, for example, the developers, testers, inspectors, and maintainers. High cognitive complexity leads to a component exhibiting undesirable external qualities, such as increased fault-proneness and reduced maintainability. Accordingly, object-oriented product metrics that affect cognitive complexity will be related with fault-proneness.

It should be noted that if a cognitive theory is substantiated, this could have important implications. It would provide us with a clear mechanism that would explain the introduction of faults into object-oriented applications.

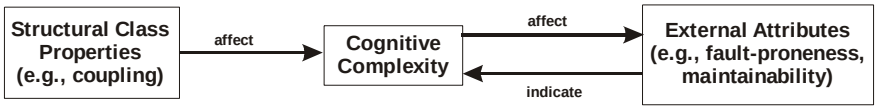

Figure 1.2. Theoretical basis for the development of object-oriented product metrics

1.4.1 Distribution of Functionality

In applications developed using functional decomposition, functionality is localized in specific procedures, the contents of data structures are accessed directly, and data central to an application is often globally accessible (Wilde, Matthews, and Huitt, 1993). Functional decomposition is believed to make procedural programs easier to understand because such programs are built upon a hierarchy in which a top-level function calls lower level functions to carry out smaller chunks of the overall task (Wiedenbeck, Ramalingam, Sarasamma, and Corritore, 1999). Hence tracing through a program to understand its global functionality is facilitated. This is not necessarily the case with object-oriented applications.

The object-oriented strategies of limiting the responsibility of a class and reusing it in multiple contexts results in a profusion of small classes in object-oriented systems (Wilde et al., 1993). For instance, Chidamber and Kemerer (Chidamber and Kemerer, 1994) found in two systems studied[6] that most classes tended to have a small number of methods (0-10), suggesting that most classes are relatively simple in their construction, providing specific abstraction and functionality. Another study of three systems performed at Bellcore[7] found that half or more of the methods are fewer than four Smalltalk lines or two C++ statements, suggesting that the classes consist of small methods (Wilde et al., 1993). Many small classes imply many interactions among the classes and a distribution of functionality across them.

In one experimental study with students and professional programmers (Boehm-Davis, Holt, and Schultz, 1992), Boehm-Davis et al. compared maintenance time for three pairs of functionally equivalent programs (implementing three different applications amounting to a total of nine programs). Three programs were implemented in a straight serial structure (i.e., one main function, or monolithic program), three were implemented following the principles of functional decomposition, and three were implemented in the object-oriented style, but without inheritance. In general, it took the students more time to change the object-oriented programs, and the professionals exhibited the same effect, although not as strongly. Furthermore, both the students and professionals noted that they found that it was most difficult to recognize program units in the object-oriented programs, and the students felt that it was also most difficult to find information in the object-oriented programs.

---

[6] One system was developed in C++, and the other in Smalltalk.

[7] The study consisted of analyzing C++ and Smalltalk systems and interviewing the developers for two of them. For a C++ system, method size was measured as the number of executable statements, and for Smalltalk size was measured by uncommented nonblank lines of code.

Widenbeck et al. (Wiedenbeck et al., 1999) make a distinction between program functionality at the local level and at the global (application) level. At the local level they argue that the object-oriented paradigm's concept of encapsulation ensures that methods are bundled together with the data on which they operate, making it easier to construct appropriate mental models and specifically to understand the individual functionality of a class. At the global level, functionality is dispersed among many interacting classes, making it harder to understand what the program is doing. They support this in an experiment with equivalent small C++ (with no inheritance) and Pascal programs whereby the subjects were better able to answer questions about the functionality of the C++ program. They then performed an experiment with larger programs. The number of correct answers for the subjects with the C++ program (with inheritance) on questions about its functionality was not much better than guessing. While this study was done with novices, it supports the general notions that high cohesion makes object-oriented programs easier to understand, and high coupling makes them more difficult to understand.

1.4.2 A Cognitive Model

Cant et al. have proposed a general cognitive theory of software complexity that elaborates on the impact of structure on understandability (Cant, Jeffery, and Henderson-Sellers, 1995). At the core of the cognitive theory proposed by Cant et al. (Cant et al., 1995) is a human memory model, that consists mainly of short-term and long-term memory.[8] In the same light, Tracz (Tracz, 1979) has claimed that "The organization and limitations of the human memory are perhaps the most significant aspects of the human thought process which affect the computer programmer." Hence, there is a some belief within the software engineering community that the human memory model is a reasonable point of departure for understanding the impact of structural properties on understandability.

Cant et al. argue that comprehension consists of both *chunking* and *tracing*. Chunking involves recognizing groups of statements and extracting from them information which is remembered as a single mental abstraction. These chunks are further grouped together into larger chunks forming a hierarchical structure. Tracing involves scanning through a program, either forwards or backwards, in order to identify relevant chunks. Subsequently, they formulate a model of cognitive complexity for a particular chunk, say D, which is the sum of three components: (1) the difficulty of understanding the chunk itself; (2) the

---

[8] Tracz (Tracz, 1979) also discusses very-short-term memory, which plays a role in attention and perception. However, this does not play a big role in cognitive theories that are used to associate software product metrics to understandability. Neither does the concept of extended-memory presented by Newell and Simon (Newell and Simon, 1972). Therefore, they will not be discussed fruther.

difficulty of understanding all the other chunks upon which D depends; and (3) the difficulty of tracing the dependencies on the chunks upon which D depends. Davis (Davis, 1984) presents a similar argument where he states "Any model of program complexity based on chunking should account for the complexity of the chunks themselves and also the complexity of their relationship."

In order to operationalize this model, it is necessary to define a *chunk*. Tracz considers a *module* to be a chunk (Tracz, 1979). However, it is not clear what exactly a module is. Cant et al. (Cant et al., 1995) make a distinction between elementary and compound chunks. Elementary chunks consist only of sequentially self-contained statements. Compound chunks are those which contain within them other chunks. Procedures containing a number of procedure calls are considered as compound chunks. At the same time, procedures containing no procedure calls may also be compound chunks. If a procedure contains more than one recognizable subunit, it is equivalent to a module containing many procedure calls in the sense that both contain within them multiple subchunks. Subsequent work by Cant et al. (Cant, Henderson-Sellers, and Jeffery, 1994) operationally defined a chunk within object-oriented software as a method. However, Henderson-Sellers (Henderson-Sellers, 1996) notes that a class is also an important type of (compound) chunk.

One factor that is contended to have an impact on complexity is chunk familiarity (Henderson-Sellers, 1996). It is argued that chunks that are referenced more often (i.e., high export coupling) will be more familiar since they are used more often. A similar argument is made by Davis for procedural programs (Davis, 1984). Therefore, when tracing other chunks more traces will lead to those with the highest export coupling. Furthermore, Henderson-Sellers (Henderson-Sellers, 1996) applies the concept of cohesion to chunking by stating that a chunk with low cohesion will be more difficult to recognize since functions performed by the chunk will be unrelated, and hence more difficult to understand.

Henderson-Sellers (Henderson-Sellers, 1996) notes that tracing disrupts the process of chunking. This occurs when it becomes necessary to understand another chunk, as when a method calls another method in a different class (method-method interaction), or when an inherited property needs to be understood. Such disruptions may cause knowledge of the original chunk to be lost. This then is contended to have a direct effect on complexity. In fact, tracing dependencies is a common task when understanding object-oriented software.

Cant et al. (Cant et al., 1994) also performed an empirical study whereby they compared subjective ratings by two expert programmers of the complexity of understanding classes with objective measures of dependencies in an object-oriented system. Their results demonstrate a concordance between the objective measures of dependency and subjective ratings of understandability.

Wilde et al.'s (Wilde et al., 1993) findings are also concordant with this conclusion, in that programmers have to understand a method's context of use by tracing back through the chain of calls that reach it, and tracing the chain of

methods it uses. Their findings were from an interview study of two C++ object-oriented systems at Bellcore and a PC Smalltalk environment. The three systems investigated span different application domains.

Related work on mental representation of object-oriented software provides further insights into the structural properties that are most difficult to understand. These works build on theories of text comprehension. Modern theories of text comprehension propose three levels of mental representation (Dijk and Kintsch, 1983; Kintsch, 1986). The first level, the *verbatim representation* consists of the literal form of the text. The second level, the *propositional textbase* consists of the propositions of the text and their relationships. The third level, the *situation model* represents the situation in the world that the text describes. Pennington (Pennington, 1987a, 1987b) subsequently applied this model to the comprehension of procedural programs, where she proposed two levels of mental representation. the program model and the domain model, which correspond to the latter two levels of the text comprehension model above. The program model consists of elementary operations and control flow information. The domain model consists of data flow and program function information.

Burkhardt et al. (Burkhardt, Detienne, and Wiedenbeck, 1997) applied this three level model to object-oriented software. For the situation model they make a distinction between a static part and a dynamic part. The static part consists of: (a) the problem objects which directly model objects of the problem domain; (b) the inheritance/composition relationships between objects; (c) reified objects; and (d) the main goals of the problem. The dynamic part represents the communication between objects and variables. The static part corresponds to client-server relationships, and the dynamic part corresponds to data flow relationships. Based on this model, Burkhardt et al. performed an experiment. They asked their subjects to study an object-oriented application and then answer questions about it. Subsequently the subjects were asked to perform either a documentation or a reuse task. The authors of the study found that the static part of the situation model is better developed than the dynamic, even for experts. Furthermore, there was no difference between experts and novices in their understanding of the dynamic part. Their findings suggest that inheritance and class-attribute coupling may have less of an impact on understandability than both cohesion and coupling.

Even though the above studies suggest that inheritance has less impact on understandability, within the software engineering community inheritance is strongly believed to make the understandability of object-oriented software difficult. According to a survey of object-oriented practitioners (Daly, Miller, Brooks, Roper, and Wood, 1995), 55% of respondents agree that inheritance depth is a factor in understanding object-oriented programs (Daly, Miller et al., 1995). “Inheritance gives rise to distributed class descriptions. That is, the complete description for a class D can only be assembled by examining D as well as each of D’s superclasses. Because different classes are described at different places in the source code of a program (often spread across several

different files), there is no single place a programmer can turn to get a complete description of a class" (Leijter, Meyers, and Reiss, 1992). While this argument is stated in terms of source code, it is not difficult to generalize it to design documents. The study by Wilde et al (Wilde et al., 1993) indicated that, to understand the behavior of a method, one has to trace inheritance dependencies, which may be considerably complicated due to dynamic binding. A similar point was made in (Leijter et al., 1992) about the understandability of programs in such languages as C++ that support dynamic binding.

In a set of interviews with 13 experienced users of object-oriented programming, Daly et al. (Daly, Wood, Brooks, Miller, and Roper, 1995) noted that if the inheritance hierarchy is designed properly then the effect of distributing functionality over the inheritance hierarchy would not be detrimental to understanding. However, it has been argued that there exists increasing conceptual inconsistency as one travels down an inheritance hierarchy (i.e., deeper levels in the hierarchy are characterized by inconsistent extensions or specializations of super-classes) (Dvorak, 1994). Therefore inheritance hierarchies are likely to be improperly designed in practice. One study by Dvorak (1994) supports this argument. Dvorak found that subjects were more inconsistent in placing classes deeper in the inheritance hierarchy than they were in placing them lower levels in the inheritance hierarchy.

1.4.3 Summary

This section provided a theoretical framework that explains the mechanism by which object-oriented metrics would be associated with fault-proneness. If this hypothesized mechanism matches reality, then we would expect object-oriented metrics to be good predictors of external quality attributes, in particular fault-proneness. In the subsequent sections, we will review the empirical studies that test these associations.

It must be recognized that the above cognitive theory suggests only one possible mechanism of what would impact external metrics. Other mechanisms can play an important role as well. For example, some studies showed that software engineers experiencing high levels of mental and physical stress tend to produce more faults (Furuyama, Arai, and Iio, 1994, 1997). Reducing schedules and many changes in requirements may induce mental stress. Physical stress may be a temporary illness, such as a cold. Therefore, cognitive complexity due to structural properties, as measured by object-oriented metrics, can never be the reason for all faults. For instance, the developers of a particular set of core functionality in a system may be placed under schedule pressure since there are many dependencies on their output. These developers may introduce more faults into the core classes due to stress.

It is not known whether the influence of object-oriented metrics dominates other effects. The only thing that can reasonably be stated is that the empirical relationships between object-oriented metrics and external metrics are not very

likely to be strong. This is due to other effects that are not accounted for, but as has been demonstrated in a number of studies, they can still be useful in practice.

1.5 Object-Oriented Thresholds

As noted in above, the practical utility of object-oriented metrics would be enhanced if meaningful thresholds could be identified. The cognitive theory described above can be expanded to include threshold effects. Hatton (Hatton, 1997) has proposed a cognitive explanation as to why a threshold effect would exist between *complexity* metrics and faults. [9]

Hatton argues that Miller (Miller, 1957) shows that humans can cope with around 7 +/- 2 pieces of information (or chunks) at a time in short-term memory, independent of information content. He then refers to the text of Hilgard et al. (Hilgard, Atkinson, and Atkinson, 1971). Hilgard et al. note that the contents of long-term memory are in a coded form and the recovery codes may get scrambled under some conditions. Short-term memory incorporates a rehearsal buffer that continuously refreshes itself. Hatton suggests that anything that can fit into short-term memory is easier to understand and less fault-prone. Pieces that are too large or too complex overflow, involving use of the more error-prone recovery code mechanism used for long-term storage. In a subsequent article, Hatton (Hatton, 1998) extends this model to object-oriented development. If we take a class as a definition of a chunk, then if the class dependencies exceed the short-term memory limit, one would expect designers and programmers to make more errors.

1.5.1 Size Thresholds

A reading of the early software engineering literature suggests that when software components exceed a certain size, fault-proneness increases rapidly. This is in essence a *threshold effect*. For instance, Card and Glass (Card and Glass, 1990) note that many programming texts suggest limiting component size to 50 or 60 SLOC. A study by O'Leary (1996) of the relationship between size and faults in knowledge-based systems found no relationship between size and faults for small components, but a positive relationship for large components; again suggesting a threshold effect. A number of standards and organizations had defined upper limits on components size (Bowen, 1984), for example, an

[9] Hatton also suggests that components that are of low complexity do not use short-term memory efficiently, and that failure to do so also leads to increased fault-proneness. However, this aspect of his model has been criticised recently (El-Emam, Benlarbi, Goel, Melo et al., 2000) and therefore will not be considered further.

upper limit of 200 source statements in MIL-STD-1679, 200 HOL executable statements in MIL-STD-1644A, 100 statements excluding annotation in RADC CP 0787796100E, 100 executable source lines in MILSTAR/ESD Spec, 200 source statements in MIL-STD-SDS, 200 source statements in MIL-STD-1679(A), and 200 HOL executable statements in FAA ER-130-005D. Bowen (Bowen, 1984) proposed component size thresholds between 23-76 source statements based on his own analysis. After a lengthy critique of size thresholds, Dunn and Ullman (Dunn and Ullman, 1979) suggest two pages of source code listing as an indicator of an overly large component. Woodfield et al. (1981) suggest a maximum threshold of 70 LOC.

Hatton (Hatton, 1998) argues that the concept of encapsulation, central to object-oriented development, lets us think about an object in isolation. If the size of this object is small enough to fit into short-term memory, then it will be easier to understand and reason about. Objects that are too large and overflow the short-term memory would tend to be more fault-prone.

1.5.2 Inheritance Thresholds

According to the above threshold theory, objects that are manipulated in short-term memory possessing inherited properties require referencing the ancestor objects. If the ancestor objects are in short-term memory then this tracing does not increase cognitive burden. However, if the ancestor objects are already encoded in long-term storage, access to long-term memory breaks the train of thought and is inherently less accurate. Accordingly, it is likely that classes will be more fault-prone if they reference inherited chunks that cannot be kept in short-term storage, and this fault-proneness increases as the extent of inheritance increases. An implication is that a certain amount of inheritance does not affect cognitive burden, it is only when inheritance increases beyond the limitations of short-term memory that understandability deteriorates. For example, Lorenz and Kidd (Lorenz and Kidd, 1994), based on their experiences with Smalltalk and C++ projects, recommended an inheritance nesting level threshold of 6, indicating that inheritance up to a certain point is not detrimental.

1.5.3 Coupling Thresholds

When there is a diffusion of functionality, then an object in short-term memory may be referencing or be referenced by many other objects. If each of these other objects is treated as a chunk and they are within short-term memory, then tracing does not increase cognitive burden. However, if more objects need to be traced than can be held in short-term memory, this requires retrieval (and pattern-matching in the case of polymorphism) of many other objects in long-term memory. Hence, the ensuing disruption leads to comprehension difficulties, and therefore greater fault-proneness. Therefore, one can argue that

when the interacting objects overflow short-term memory, this will lead to an increase in fault-proneness. The implication of this is that a certain amount of coupling does not affect cognitive burden, until a non-zero coupling threshold is exceeded.

1.6 Empirical Evidence

A considerable number of empirical studies have been performed to validate the relationship between object-oriented metrics and class fault-proneness. Some studies covered the metrics that were described earlier in this chapter, such as (Basili et al., 1996; Briand et al., 1997; Briand, Wuest, Daly, and Porter, 2000; Briand, Wuest et al., 1998; Tang et al., 1999). Other studies validated a set of polymorphism metrics (Benlarbi and Melo, 1999), a coupling dependency metric (Binkley and Schach, 1998), a set of metrics defined on Shlaer-Mellor designs (Cartwright and Shepperd, 2000), another metrics suite (F. Brito e Abreu and Melo, 1996), and a set of coupling metrics (R. Harrison, Counsell, and Nithi, 1998). Other external measures of interest that have been studied are productivity (Chidamber et al., 1998), maintenance effort (Li and Henry, 1993), and development effort (Chidamber et al., 1998; Misic and Tesic, 1998; Nesi and Querci, 1998). However, here we will focus on the fault-proneness external measure.

It would seem that with such a body of work we would also have a large body of knowledge about which metrics are related to fault-proneness. Unfortunately, this is not the case. A recent study (El-Emam, Benlarbi, Goel, and Rai, 2000) has demonstrated a confounding effect of class size on the validity of object-oriented metrics. This means that if one does not control the effect of class size when validating metrics, then the results would be quite optimistic. The reason for this argument is illustrated in Figure 1.3. Class size is correlated with most product metrics, and it is also a good predictor of fault-proneness*: Bigger classes are simply more likely to have a fault.*

Empirical evidence supports an association between object-oriented product metrics and size. For example, in (Briand et al., 2000) the Spearman rho correlation coefficients go as high as 0.43 for associations between some coupling and cohesion metrics with size, and 0.397 for inheritance metrics. Both results are statistically significant (at an alpha level of say 0.1).

Similar patterns emerge in other studies. One study by Briand et al. (Briand, Wuest et al., 1998) reports relatively large correlations between size and object-oriented metrics. In another study (Cartwright and Shepperd, 2000) the authors display the correlation matrix showing the Spearman correlation between a set of object-oriented metrics that can be collected from Shlaer-Mellor designs and C++ LOC. The correlations range from 0.563 to 0.968, all statistically significant at an alpha level 0.05. This result also indicates very strong correlations with size. In further support of this hypothesis, the relationship

between size and defects is clearly visible in the study by Cartwright and Shepperd (Cartwright and Shepperd, 2000), where the Spearman correlation was found to be 0.759 and statistically significant. Another study of image analysis programs written in C++ (R. Harrison, Samaraweera, Dobie, and Lewis, 1996) found a Spearman correlation of 0.53 between size in LOC and the number of errors found during testing, also statistically significant at an alpha level of 0.05. Finally, Briand et al. (Briand et al., 2000) find statistically significant associations between 6 different size metrics and fault-proneness for C++ programs, with a change in odds ratio going as high as 4.952 for one of the size metrics.

A number of validation studies did not control for size, such as (Binkley and Schach, 1998; Briand et al., 2000; Briand, Wuest et al., 1998; R. Harrison et al., 1998; Tang et al., 1999). This means that if an association is found between a particular metric and fault-proneness, this may be due to the fact that higher values on that metric also mean higher size values. In the following sections, we therefore only draw conclusions from studies that *did* control for size, either statistically or experimentally.

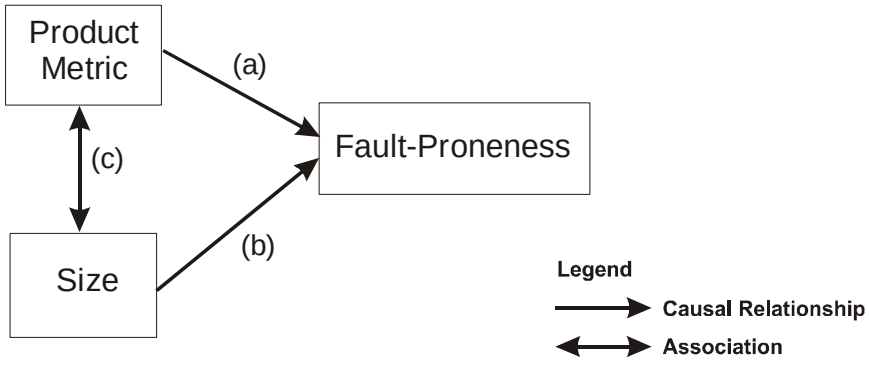

Figure 1.3. Illustration of confounding effect of class size.

1.6.1 Inheritance Metrics

As noted in (Deligiannis and Shepperd, 1999), the software engineering community has been preoccupied with inheritance and its effect on quality. Many studies have investigated that particular feature of the object-oriented paradigm.

An experimental investigation found that making changes to a C++ program with inheritance consumed more effort than a program without inheritance, and the author attributed this to the subjects finding the inheritance program more

difficult to understand based on responses to a questionnaire (Cartwright, 1998). Another study by Cartwright and Shepperd (Cartwright and Shepperd, 2000) found that classes with inheritance tend to be more fault prone. This suggests that, holding everything else equal, understandability of classes is stable when there is no inheritance, but falls if there is any inheritance.

In two further experiments (Unger and Prechelt, 1998), subjects were given three equivalent Java programs to modify, and the maintenance time was measured. One of the Java programs was *flat*, in that it did not take advantage of inheritance; one had an inheritance depth of 3; and one had an inheritance depth of 5. In an initial experiment, the programs with an inheritance depth of 3 on the average took longer to maintain than the flat program, but the program with an inheritance depth of 5 took as much time as the flat program. The authors attribute this to the fact that the amount of changes required to complete the maintenance task for the deepest inheritance program was smaller. The results for a second task in the first experiment and the results of the second experiment indicate that it took longer to maintain the programs with inheritance. This was attributed to the need to trace call sequences up the inheritance hierarchy in order to understand what a class is doing.

However, another study (Daly, Brooks, Miller, Roper, and Wood, 1996) contradicts these findings. The authors conducted a series of classroom experiments comparing the time to perform maintenance tasks on a flat C++ program and a C++ program with three levels of inheritance. The result was a significant reduction in maintenance effort for the inheritance program. An internal replication by the same authors found the results to be in the same direction, albeit the p-value was larger. This suggests an inverse effect for inheritance depth to the one described above.

More recent studies also reported similar contradictory results. Two studies found that there is a relationship between the depth of inheritance tree and fault-proneness in Java programs (El-Emam et al., 2001; Glasberg, El-Emam, Melo, Machado, and Madhavji, 2000). However, two other studies found no such effect with C++ programs (El-Emam, Benlarbi, Goel, and Rai, 1999, 2000).

Overall, then, it seems that the evidence as to the impact of inheritance depth on fault-proneness is rather equivocal. This is usually an indication that there is another effect that is confounded with inheritance depth. Further research is necessary to identify this confounding effect and disentangle it from inheritance depth in order to assess the effect of inheritance depth by itself.

1.6.2 Coupling Metrics

The most promising results with object-oriented metrics were obtained using coupling metrics. A summary of three recent results is given in Table 1.2. The "*" indicates that for this particular study ACMIC was not evaluated because it had too few observations that were non-zero, and hence lacked variation. It can be seen that both import and export coupling metrics tend to be associated with

fault-proneness. The type of coupling depends on the system, likely a reflection of the overall design approach.

Table 1.2. Summary of validation results for coupling metrics.

	(El-Emam et al., 1999) (C++ system)	(El-Emam et al., 2001) (Java system)	(Glasberg et al., 2000) (Java system)
CBO	X	Not evaluated	Not evaluated
OCAEC	X	X	No association found
ACMIC	X	Not evaluated *	X
OCMEC	X	X	No association found
OMMEC	X	Not evaluated	Not evaluated
OCMIC	No association found	X	X
OCAIC	No association found	No association found	X

1.6.3 Cohesion Metrics

Three studies that evaluated the effect of cohesion, in the form of the LCOM metric, found no effect of cohesion on fault-proneness (Benlarbi, El-Emam, Goel, and Rai, 2000; El-Emam et al., 1999; El-Emam, Benlarbi, Goel, and Rai, 2000). This is not surprising given that the concept of cohesion is not well understood.

1.6.4 Thresholds

A recent series of studies led by the author investigated thresholds for object-oriented metrics (Benlarbi et al., 2000; El-Emam, Benlarbi, Goel, Melo et al., 2000; Glasberg et al., 2000). The first study demonstrated that an absence of size thresholds for object-oriented classes (El-Emam, Benlarbi, Goel, Melo et al., 2000). The remaining two studies demonstrated that an absence of threshold effects for a subset of the metrics described earlier (Benlarbi et al., 2000; Glasberg et al., 2000). The results are consistent across all of the three studies: there are no thresholds for contemporary object-oriented metrics, including class size.

Absence of thresholds does not mean that the claims of limits on short-term memory are inapplicable to software engineering. However, the applicability of this cognitive model to object-oriented applications needs to be refined further. It is plausible that a *chunk* in the object-oriented paradigm is a method rather than a class. It is also plausible that dependencies between chunks need to be weighted according to the complexity of the dependency. These hypotheses

require further investigation. The main result remains, however, that the existence of thresholds for contemporary object-oriented metrics lacks evidence.

The existing object-oriented thresholds that have been derived from experiential knowledge, such as those of Lorenz and Kidd (Lorenz and Kidd, 1994) and Rosenberg et al. (Rosenberg et al., 1999), may, however, still be of some practical utility despite these findings. Even if there is a continuous (i.e., no threshold) relationship between these metrics and fault-proneness as we have found, if you draw a line at a high value of a measure and call this a threshold, classes that are above the threshold will still be the *most fault-prone*. This situation is illustrated in the left panel of Figure 1.4 (where p is the probability of a fault). Therefore, for the purpose of identifying the most fault-prone classes, such thresholds will likely work. Yet classes with values below the threshold can still mean high fault-proneness, just not the highest.

Had a genuine threshold effect been identified, then classes with values below the threshold represent a *safe* region whereby designers deliberately restricting their classes within this region can have some assurance that the classes will have, everything else being equal, *minimal fault-proneness*. This genuine threshold effect is illustrated in the right panel of Figure 1.4.

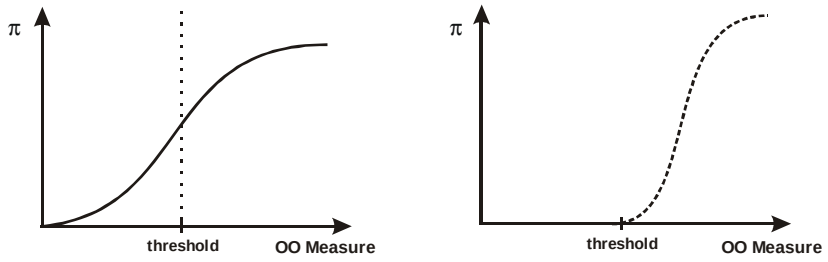

Figure 1.4. Different types of thresholds. An arbitrarily chosen threshold is illustrated on the left. A genuine threshold effect is illustrated on the right.

1.7 Conclusions

This chapter reviewed contemporary object-oriented metrics, the theory behind them, and the empirical evidence that supports their use. The results obtained thus far can provide the basis for concrete guidelines for quality management in object-oriented applications. These can be summarized as follows:

- The most important metrics to collect seem to be those measuring the different types of export and import coupling. Most of these metrics have

the advantage that they can be collected at the early design stages, allowing for early quality management. Assign your best people to work on classes with high values on the coupling metrics.

- If historical data is available, it would be even better to rank your classes by their predicted fault-proneness. This involves constructing a logistic regression model using the above coupling metrics (and a measure of size). This model would predict the probability of a fault in each class. Assign your best people to work on classes with the largest predicted fault-proneness.
- Other managerial actions that can be taken are: larger and more experienced inspection teams for classes with high fault-proneness, and development of more test cases for these classes. Given that these classes are expected to be the most fault-prone, such defect detection activities will help identify and remove these faults before the software is released.

It is clear from the above studies that we are not yet at the stage were precise prescriptive or proscriptive design guidelines can be developed. However, the findings so far are a useful starting point.

The results do not, in general, disconfirm the cognitive complexity theory presented earlier. We did not find compelling evidence that the depth of inheritance tree is a major contributor to fault-proneness. However, this may be due to other ancestor-based coupling metrics being the main effect predicted by the theory rather than inheritance depth itself.

From a research perspective, the following conclusions can be drawn:

- Contemporary cohesion metrics tend not to be good predictors of fault-proneness. Further work needs to be performed at defining cohesion better, and developing metrics to measure it.
- The evidence as to the impact of inheritance depth itself on fault-proneness is equivocal. This is an issue that requires further investigation.
- No threshold effects were identified. This most likely means that the manner in which theories about short and long term human memory have been adapted to object-oriented applications needs further refinement.

In closing, it is important to note that the studies from which these recommendations came from looked at commercial systems (i.e., not student applications). This makes the case that the results are applicable to actual projects more convincing. Furthermore, they focused only on fault-proneness. It is plausible that studies that focus on maintenance effort or development effort would give rise to different recommendations.

1.8 Acknowledgments

Discussions with Janice Singer and Norm Vinson have contributed towards improving the cognitive theory elements of this chapter. I also wish to thank them for providing directions through the cognitive psychology literature.

Review comments from Hakan Erdogmus and Barbara Kitchenham have been very useful for making improvements to the chapter.

1.9 References

Abreu, F. B. e., and Carapuca, R. (1994). *Object-Oriented Software Engineering: Measuring and Controlling the Development Process.* In Proceedings of the 4th International Conference on Software Quality.

Abreu, F. B. e., and Melo, W. (1996). *Evaluating the Impact of Object-Oriented Design on Software Quality.* In Proceedings of the 3rd International Software Metrics Symposium, pp. 90-99.

Almeida, M., Lounis, H., and Melo, W. (1998). *An Investigation on the Use of Machine Learned Models for Estimating Correction Costs.* In Proceedings of the 20th International Conference on Software Engineering, pp. 473-476.

Basili, V., Briand, L., and Melo, W. (1996). A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering, 22*(10), 751-761.

Basili, V., Condon, S., El-Emam, K., Hendrick, R., and Melo, W. (1997). *Characterizing and Modeling the Cost of Rework in a Library of Reusable Software Components.* In Proceedings of the 19th International Conference on Software Engineering, pp. 282-291.

Benlarbi, S., El-Emam, K., Goel, N., and Rai, S. (2000). *Thresholds for Object-Oriented Measures.* NRC/ERB 1073. (National Research Council of Canada).

Benlarbi, S., and Melo, W. (1999). *Polymorphism Measures for Early Risk Prediction.* In Proceedings of the 21st International Conference on Software Engineering, pp. 334-344.

Binkley, A., and Schach, S. (1998). *Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures.* In Proceedings of the 20th International Conference on Software Engineering, pp. 452-455.

Boehm-Davis, D., Holt, R., and Schultz, A. (1992). The Role of Program Structure in Software Maintenance. *International Journal of Man-Machine Studies, 36*, 21-63.

Bowen, J. (1984). Module Size: A Standard or Heuristic? *Journal of Systems and Software, 4*, 327-332.

Briand, L., Arisholm, E., Counsell, S., Houdek, F., and Thevenod-Fosse, P. (1999). Empirical Studies of Object-Oriented Artifacts, Methods, and Processes: State of the Art and Future Direction. *Empirical Software Engineering: An International Journal, 4*(4), 387-404.

Briand, L., Basili, V., and Hetmanski, C. (1993). Developing Interpretable Models with Optimized Set Reduction for Identifying High-Risk Software Components. *IEEE Transactions on Software Engineering, 19*(11), 1028-1044.

Briand, L., Daly, J., and Wuest, J. (1998). A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering: An International Journal, 3*, 65-117.

Briand, L., Daly, J., and Wuest, J. (1999). A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering, 25*(1), 91-121.

Briand, L., Devanbu, P., and Melo, W. (1997). *An Investigation into Coupling Measures for C++*. In Proceedings of the 19th International Conference on Software Engineering.

Briand, L., Thomas, W., and Hetmanski, C. (1993). *Modeling and Managing Risk Early in Software Development.* In Proceedings of the International Conference on Software Engineering, pp. 55-65.

Briand, L., and Wuest, J. (1999). *The Impact of Design Properties on Development Cost in Object-Oriented Systems.* ISERN-99-16. (International Software Engineering Research Network).

Briand, L., Wuest, J., Daly, J., and Porter, V. (2000). Exploring the Relationships Between Design Measures and Software Quality in Object Oriented Systems. *Journal of Systems and Software, 51*, 245-273.

Briand, L., Wuest, J., Ikonomovski, S., and Lounis, H. (1998). *A Comprehensive Investigation of Quality Factors in Object-Oriented Designs: An Industrial Case Study.* ISERN-98-29. (International Software Engineering Research Network).

Burkhardt, J.-M., Detienne, F., and Wiedenbeck, S. (1997). Mental Representations Constructed by Experts and Novices in Object-Oriented Program Comprehension. In *Human-computer Interaction: INTERACT'97*, pp. 339-346.

Cant, S., Henderson-Sellers, B., and Jeffery, R. (1994). Application of Cognitive Complexity Metrics to Object-Oriented Programs. *Journal of Object-Oriented Programming, 7*(4), 52-63.

Cant, S., Jeffery, R., and Henderson-Sellers, B. (1995). A Conceptual Model of Cognitive Complexity of Elements of the Programming Process. *Information and Software Technology, 7*, 351-362.

Card, D., and Glass, R. (1990). *Measuring Software Design Quality*. (Prentice-Hall).

Cartwright, M. (1998). An Empirical View of Inheritance. *Information and Software Technology, 40*, 795-799.

Cartwright, M., and Shepperd, M. (2000). An Empirical Investigation of an Object-Oriented Software System. *IEEE Transactions on Software Engineering (to appear)*.

Chidamber, S., Darcy, D., and Kemerer, C. (1998). Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis. *IEEE Transactions on Software Engineering, 24*(8), 629-639.

Chidamber, S., and Kemerer, C. (1991). *Towards a Metrics Suite for Object-Oriented Design*. In Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'91), pp. 197-211.

Chidamber, S., and Kemerer, C. (1994). A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering, 20*(6), 476-493.

Chidamber, S., and Kemerer, C. (1995). Authors' Reply. *IEEE Transactions on Software Engineering, 21*(3), 265.

Clark, B. (1997). *The Effects of Software Process Maturity on Software Development Effort.* Unpublished PhD Thesis, University of Southern California.

Coallier, F., Mayrand, J., and Lague, B. (1999). Risk Management in Software Product Procurement. In K. El-Emam and N. H. Madhavji (Eds.), *Elements of Software Process Assessment and Improvement*. (IEEE CS Press).

CodeWork. (2000). *JStyle*. Available: http://www.codework.com/, 20th April 2000.

Daly, J., Brooks, A., Miller, J., Roper, M., and Wood, M. (1996). Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software. *Empirical Software Engineering: An International Journal, 1*(2), 109-132.

Daly, J., Miller, J., Brooks, A., Roper, M., and Wood, M. (1995). *Issues on the Object-Oriented Paradigm: A Questionnaire Survey*. EFoCS-8-95, Department of Computer Science - University of Strathclyde.

Daly, J., Wood, M., Brooks, A., Miller, J., and Roper, M. (1995). *Structured Interviews on the Object-Oriented Paradigm*. EFoCS-7-95, Department of Computer Science - University of Strathclyde.

Davis, J. (1984). Chunks: A Basis for Complexity Measurement. *Information Processing and Management, 20*(1), 119-127.

Deligiannis, I., and Shepperd, M. (1999). *A Review of Experimental Investigations into Object-Oriented Technology.* In Proceedings of the Fifth IEEE Workshop on Empirical Studies of Software Maintenance, pp. 6-10.

Devanbu, P. (2000). *Gen++*. Available: http://seclab.cs.ucdavis.edu/~devanbu/genp/, April 20th 2000.

Dijk, T. v., and Kintsch, W. (1983). *Strategies of Discourse Comprehension.* (Academic Press).

Dunn, R., and Ullman, R. (1979). *Modularity Is Not a Matter of Size.* In Proceedings of the 1979 Annual Reliability and Maintainability Symposium, pp. 342-345.

Dvorak, J. (1994). Conceptual Entropy and Its Effect on Class Hierarchies. *IEEE Computer*, 59-63.

Ebert, C., and Liedtke, T. (1995). *An Integrated Approach for Criticality Prediction.* In Proceedings of the 6th International Symposium on Software Reliability Engineering, pp. 14-23.

El-Emam, K., Benlarbi, S., Goel, N., Melo, W., Lounis, H., and Rai, S. (2000). *The Optimal Class Size for Object-Oriented Software: A Replicated Study* NRC/ERB 1074. (National Research Council of Canada).

El-Emam, K., Benlarbi, S., Goel, N., and Rai, S. (1999). *A Validation of Object-Oriented Metrics*. NRC/ERB 1063. (National Research Council of Canada).

El-Emam, K., Benlarbi, S., Goel, N., and Rai, S. (2000). The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. *IEEE Transactions on Software Engineering (to appear).*

El-Emam, K., and Birk, A. (2000a). Validating the ISO/IEC 15504 Measures of Software Development Process Capability. *Journal of Systems and Software, 51*(2), 119-149.

El-Emam, K., and Birk, A. (2000b). Validating the ISO/IEC 15504 Measures of Software Requirements Analysis Process Capability. *IEEE Transactions on Software Engineering (to appear).*

El-Emam, K., Drouin, J.-N., and Melo, W. (1998). *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination*. (IEEE CS Press).

El-Emam, K., Melo, W., and Machado, J. (2001). The Prediction of Faulty Classes Using Object-Oriented Design Metrics. *Journal of Systems and Software (to appear).*

Evanco, W. (1997). Poisson Analyses of Defects for Small Software Components. *Journal of Systems and Software, 38*, 27-35.

Fenton, N. (1991). *Software Metrics: A Rigorous Approach.* (Chapman and Hall).

Fenton, N., and Ohlsson, N. (2000). Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering (to appear).*

Flowe, R., and Thordahl, J. (1994). *A Correlational Study of the SEI's Capability Maturity Model and Software Development Performance in DoD*

Contracts. Unpublished MSc Thesis, The Air Force Institute of Technology.

French, V. (1999). *Establishing Software Metrics Thresholds.* In Proceedings of the 9th International Workshop on Software Measurement.

Furuyama, T., Arai, Y., and Iio, K. (1994). Fault Generation Model and Mental Stress Effect Analysis. *Journal of Systems and Software, 26*, 31-42.

Furuyama, T., Arai, Y., and Iio, K. (1997). Analysis of Fault Generation Caused by Stress During Software Development. *Journal of Systems and Software, 38*, 13-25.

Glasberg, D., El-Emam, K., Melo, W., Machado, J., and Madhavji, N. (2000). An Empirical Validation of Object-Oriented Design Measures. *(submitted for publication).*

Goldenson, D. R., and Herbsleb, J. (1995). *After the Appraisal: A Systematic Survey of Process Improvement, its Benefits, and Factors that Influence Success*. CMU/SEI-95-TR-009, Software Engineering Institute.

Harrison, R., Counsell, S., and Nithi, R. (1998). *Coupling Metrics for Object Oriented Design.* In Proceedings of the 5th International Symposium on Software Metrics, pp. 150-157.

Harrison, R., Samaraweera, L., Dobie, M., and Lewis, P. (1996). An Evaluation of Code Metrics for Object-Oriented Programs. *Information and Software Technology, 38*, 443-450.

Harrison, W. (1988). Using Software Metrics to Allocate Testing Resources. *Journal of Management Information Systems, 4*(4), 93-105.

Hatton, L. (1997). Re-examining the Fault Density - Component Size Connection. *IEEE Software*, 89-97.

Hatton, L. (1998). Does OO Sync with How We Think ? *IEEE Software*, 46-54.

Henderson-Sellers, B. (1996). *Object-Oriented Metrics: Measures of Complexity.* (Prentice-Hall).

Hilgard, E., Atkinson, R., and Atkinson, R. (1971). *Introduction to Psychology.* (Harcourt Brace Jovanovich).

Hosmer, D., and Lemeshow, S. (1989). *Applied Logistic Regression*. (John Wiley and Sons).

Hudepohl, J., Aud, S., Khoshgoftaar, T., Allen, E., and Mayrand, J. (1996a). EMERALD: Software Metrics and Models on the Desktop. *IEEE Software, 13*(5), 56-60.

Hudepohl, J., Aud, S., Khoshgoftaar, T., Allen, E., and Mayrand, J. (1996b). *Integrating Metrics and Models for Software Risk Assessment.* In Proceedings of the 7th International Symposium on Software Reliability Engineering, pp. 93-98.

ISO/IEC. (1996). *Information Technology - Software Product Evaluation; Part 1: Overview*. ISO/IEC DIS 14598-1. (International Organization for Standardization and the International Electrotechnical Commission).

Jones, C. (1999). The Economics of Software Process Improvements. In K. El-Emam and N. H. Madhavji (Eds.), *Elements of Software Process Assessment and Improvement*. (IEEE CS Press).

Jorgensen, M. (1995). Experience with the Accuracy of Software Maintenance Task Effort Prediction Models. *IEEE Transactions on Software Engineering, 21*(8), 674-681.

Kaaniche, M., and Kanoun, K. (1996). *Reliability of a Commercial Telecommunications System.* In Proceedings of the International Symposium on Software Reliability Engineering, pp. 207-212.

Khoshgoftaar, T., Allen, E., Jones, W., and Hudepohl, J. (1999). *Classification Tree Models of Software Quality Over Multiple Releases.* In Proceedings of the International Symposium on Software Reliability Engineering, pp. 116-125.

Khoshgoftaar, T., Allen, E., Kalaichelvan, K., and Goel, N. (1996). The Impact of Software Evolution and Reuse on Software Quality. *Empirical Software Engineering: An International Journal, 1*, 31-44.

Kintsch, W. (1986). Learning from Text. *Cognition and Instruction, 3,* 87-108.

Kitchenham, B., and Linkman, S. (1990). Design Metrics in Practice. *Information and Software Technology, 32*(4), 304-310.

Krishnan, M. S., and Kellner, M. (1999). Measuring Process Consistency: Implications for Reducing Software Defects. *IEEE Transactions on Software Engineering, 25*(6), 800-815.

Lanubile, F., and Visaggio, G. (1997). Evaluating Predictive Quality Models Derived from Software Measures: Lessons Learned. *Journal of Systems and Software, 38*, 225-234.

Leijter, M., Meyers, S., and Reiss, S. (1992). Support for Maintaining Object-Oriented Programs. *IEEE Transactions on Software Engineering, 18*(12), 1045-1052.

Lewis, J., and Henry, S. (1989). *A Methodology for Integrating Maintainability Using Software Metrics.* In Proceedings of the International Conference on Software Maintenance, pp. 32-39.

Li, W., and Henry, S. (1993). Object-Oriented Metrics that Predict Maintainability. *Journal of Systems and Software, 23*, 111-122.

Lorenz, M., and Kidd, J. (1994). *Object-Oriented Software Metrics*. (Prentice-Hall).

Lyu, M., Yu, J., Keramides, E., and Dalal, S. (1995). *ARMOR: Analyzer for Reducing Module Operational Risk.* In Proceedings of the 25th International Symposium on Fault-Tolerant Computing, pp. 137-142.

Metameta. (2000). *Metameta Metrics.* Available: http://www.metamata.com, 20th April.

Miller, G. (1957). The Magical Number 7 Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review, 63*, 81-97.

Misic, V., and Tesic, D. (1998). Estimation of Effort and Complexity: An Object-oriented Case Study. *Journal of Systems and Software, 41*, 133-143.

Moller, K.-H., and Paulish, D. (1993). *An Empirical Investigation of Software Fault Distribution.* In Proceedings of the First International Software Metrics Symposium, pp. 82-90.

Nesi, P., and Querci, T. (1998). Effort Estimation and Prediction of Object-Oriented Systems. *Journal of Systems and Software, 42*, 89-102.

Newell, A., and Simon, H. (1972). *Human Problem Solving*. (Prentice-Hall).

Number-Six-Software. (2000). *Metrics One*. Available: http://www.numbersix.com/metricsone/index.htm, April 20th 2000.

ObjectSoft. (2000). *ObjectDetail*. Available: http://www.obsoft.com, 20th April 2000.

Ohlsson, N., and Alberg, H. (1996). Predicting Fault-Prone Software Modules in Telephone Switches. *IEEE Transactions on Software Engineering, 22*(12), 886-894.

O'Leary, D. (1996). The Relationship Between Errors and Size in Knowledge-Based Systems. *International Journal of Human-Computer Studies, 44*, 171-185.

Pennington, N. (1987a). *Comprehension Strategies in Programming*. In Empirical Studies of Programmers, 2nd Workshop, pp. 100-113.

Pennington, N. (1987b). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology, 19*, 295-341.

Power-Software. (2000a). *Krakatau for C/C++*. Available: http://www.power-soft.co.uk/.

Power-Software. (2000b). *Krakatau Java*. Available: http://www.power-soft.co.uk/.

Rosenberg, L., Stapko, R., and Gallo, A. (1999). *Object-Oriented Metrics for Reliability*. Presented at IEEE International Symposium on Software Metrics.

Software Engineering Institute. (1995). *The Capability Maturity Model: Guidelines for Improving the Software Process*. (Addison Wesley).

Szentes, J., and Gras, J. (1986). *Some Practical Views of Software Complexity Metrics and a Universal Measurement Tool.* In Proceedings of the First Australian Software Engineering Conference, pp. 83-88.

Tang, M.-H., Kao, M.-H., and Chen, M.-H. (1999). *An Empirical Study on Object Oriented Metrics.* In Proceedings of the Sixth International Software Metrics Symposium, pp. 242-249.

Tracz, W. (1979). Computer Programming and the Human Thought Process. *Software - Practice and Experience, 9*, 127-137.

Unger, B., and Prechelt, L. (1998). *The Impact of Inheritance Depth on Maintenance Tasks - Detailed Description and Evaluation of Two Experiment Replications* 19/1998. (Fakultat fur Informatik - Universitaet Karlsruhe).

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., and Corritore, C. (1999). A Comparison of the Comprehension of Object-Oriented and Procedural Programs by Novice Programmers. *Interacting with Computers, 11*(3), 255-282.

Wilde, N., Matthews, P., and Huitt, R. (1993). Maintaining Object-Oriented Software. *IEEE Software*, 75-80.

Woodfield, S., Shen, V., and Dunsmore, H. (1981). A Study of Several Metrics for Programming Effort. *Journal of Systems and Software, 2*, 97-103.