A Hierarchical Model for Object-Oriented Design Quality Assessment

Jagdish Bansiya, *Member*, *IEEE*, and Carl G. Davis, *Fellow*, *IEEE*

Abstract—This paper describes an improved hierarchical model for the assessment of high-level design quality attributes in object-oriented designs. In this model, structural and behavioral design properties of classes, objects, and their relationships are evaluated using a suite of object-oriented design metrics. This model relates design properties such as encapsulation, modularity, coupling, and cohesion to high-level quality attributes such as reusability, flexibility, and complexity using empirical and anecdotal information. The relationship, or links, from design properties to quality attributes are weighted in accordance with their influence and importance. The model is validated by using empirical and expert opinion to compare with the model results on several large commercial object-oriented systems. A key attribute of the model is that it can be easily modified to include different relationships and weights, thus providing a practical quality assessment tool adaptable to a variety of demands.

Index Terms—Quality model, quality attributes, design metrics, product metrics, object-oriented metrics.

1 Introduction

The demand for quality software continues to intensify due to our society's increasing dependence on software and the often devastating effect that a software error can have in terms of life, financial loss, or time delays. Today's software systems must ensure consistent and error free operation every time they are used. This demand for increased software quality has resulted in quality being more of a differentiator between products than it ever has been before. In a marketplace of highly competitive products, the importance of delivering quality is no longer an advantage but a necessary factor for companies to be successful. While there is uniform agreement that we need quality software, the question of how, when, and where you measure and assure quality are far from settled issues.

The switch to the object-oriented paradigm has changed the elements that we use to assess software quality. Traditional software product metrics that evaluate product characteristics such as size, complexity, performance, and quality must be changed to rely on some fundamentally different notions such as encapsulation, inheritance, and polymorphism which are inherent in object-orientation. This has led to the definition of many new metrics [8], [15], [20] to measure the products of the object-oriented approach.

However, the new object-oriented metrics are varied in what they measure, how they are used in measuring, and when they are applicable. Many of the newer metrics have only been validated with small, and sometimes nonrealistic data sets and, therefore, the practical applicability and effectiveness of the metrics on large complex projects such as those encountered in an industrial environment is not known. Finally, if the goal is assessing the external quality attributes of the product rather than simply collecting individual metrics, then there must be a well defined way of connecting the two.

Many of the metrics and quality models currently available for object-oriented software analyses can be applied only after a product is complete, or nearly complete. They rely upon information extracted from the implementation of the product. This provides information too late to help in improving internal product characteristics prior to the completion of the product. Thus, there is a need for metrics and models that can be applied in the early stages of development (requirements and design) to ensure that the analysis and design have favorable internal properties that will lead to the development of a quality end product. This would give developers an opportunity to fix problems, remove irregularities and nonconformance to standards, and eliminate unwanted complexity early in the development cycle. This should significantly help in reducing rework during and after implementation, as well as designing effective test plans and better project and resource planning.

Fortunately, the object-oriented approach naturally lends itself to an early assessment and evaluation. Object-oriented methodologies require significant effort early in the development cycle to identify objects and classes, attributes and operations, and relationships. Encapsulation, inheritance, and polymorphism require designers to carefully structure the design and consider the interaction between objects. The result of this early analysis and design process is a blueprint for implementation. Therefore, the approach provides the information needed to assess the quality of a design's classes, structure, and relationships before they are committed to an implementation.

- *J. Bansiya is with the Department of Mathematics and Computer Science, California State University, Hayward, CA 94542. E-mail: jbansiya@csuhayward.edu.*
- *C. Davis is with the Computer Science Department, University of Alabama in Huntsville, Huntsville, AL 35899. E-mail: cdavis@cs.uah.edu.*

Manuscript received 24 Nov. 1997; revised 29 Nov. 1998; accepted 27 Jan. 2000.
Recommended for acceptance by D.R. Jeffery.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 105978.

In order to provide the significant improvement that is needed to measure object-oriented software development products there must be a concerted effort to address the following barriers that currently exist. First of all, there must be a way to assess object-oriented software quality as early as possible in the development cycle. This means we must adapt or invent metrics that can be applied early and give reliable predictions. There must also be some quantifiable way to relate measurable object-oriented characteristics to the higher-level desirable software quality attributes. These relationships need to be defined and quality models developed that can be adapted for a variety of application environments so that differing ways of looking at quality can be quantified. The process of measurement also needs to be automated better and constructed in a nonintrusive manner. Tools that will automatically extract data for large software systems and can be easily adapted are needed.

There are no known comprehensive and complete models or frameworks that evaluate the overall quality of designs developed using an object-oriented approach based on its internal design properties. The new model and results presented in this article address many of the issues raised above. This new model has the lower-level design metrics well defined in terms of design characteristics, and quality is assessed as an aggregation of the model's individual high-level quality attributes. The high-level attributes are assessed using a set of empirically identified and weighted object-oriented design properties, which are derived from object-oriented metrics which measure the lowest-level structural, functional, and relational details of a design. The software tool, QMOOD++ [2], allows the design assessment to be carried out automatically, given the parameters of interest for particular evaluation. It allows rapid modification of the model to include new relationships or metrics. Lastly, the effectiveness of this initial model in predicting design quality attributes has been validated against numerous real-world projects. The quality predicted by the model shows good correlation with evaluator assessments of project designs and predicts implementation qualities well.

2 Previous Quality Models

Software quality is still a vague and multifaceted concept, which means different things to different people. Typically, the way we measure quality depends on the viewpoint we take [17]. This makes the direct assessment of quality very difficult. In order to better quantify quality, researchers have developed indirect models that attempt to measure software product quality by using a set of quality attributes, characteristics, and metrics. An important assumption in defining these quality models is that internal product characteristics (internal quality indicators) influence external product attributes (quality in use), and by evaluating a product's internal characteristics some reasonable conclusions can be drawn about the product's external quality attributes. This product-based approach is frequently adopted by software-metrics advocates because it offers an objective and context independent view of quality [17].

One of the earliest software product quality models was suggested by McCall [21] and his colleagues. McCall's quality model defines software-product qualities as a hierarchy of factors, criteria, and metrics and was the first of several models of the same form. International efforts have also led to the development of a standard for software-product quality measurement, ISO 9126. All of these models vary in their hierarchical definition of quality, but they share a common difficulty. The models are vague in their definition of the lower-level details and metrics needed to attain a quantitative assessment of product quality. This lack of specifics in these models offers little guidance to software developers who need to build quality products.

Another difficulty with earlier models was the inability to account for dependency among quality attributes. While several high-level attributes [27] are used to refer to product quality, generally, only a subset of these attributes would be relevant for each different viewpoint. Since the influence of individual attributes on overall quality is not always independent, individual or groups of quality attributes can influence the overall quality in conflicting ways. For example, a quality goal for higher flexibility makes it harder to achieve a goal of lower maintainability.

Earlier models also failed to include ways of accounting for the degree of influence of individual attributes. When overall quality is assessed as an aggregate of a set of quality attributes, the significance of all attributes to the overall quality may not be equal and, therefore, the influence of an attribute may need to be changed by a weighting factor. For large organizations with sophisticated networks and real-time processing, the attributes *performance* and *reliability* may be the most important attributes [16], whereas, for organizations that are in the multiplatform business, *portability* and *extendibility* are important attributes. The identification of a set of quality attributes that completely represents quality assessment is not a trivial task and depends upon many things including management objectives, business goals, competition, economics, and time allocated for the development of the product.

However, the earlier models provided an excellent framework from which to proceed. New metrics, relationships, and weights can be evaluated and defined in the context of these earlier models.

3 Model Development

Recently, a long needed framework for building product-based quality models has been developed by Dromey [10], [11]. It addresses some of the problems of the earlier models such as McCall's and ISO 9126. The framework is a methodology for the development of quality models in a bottom-up fashion, providing an approach that will ensure that the lower-level details are well-specified and computable.

Dromey's quality framework, like the earlier models, relies on the decomposition of high-level quality attributes into tangible, quality-carrying properties of a product's components (requirements, design, and implementation). There are three principal elements to Dromey's generic quality model: product properties that influence quality, a set of high-level quality attributes, and a means of linking them [11].

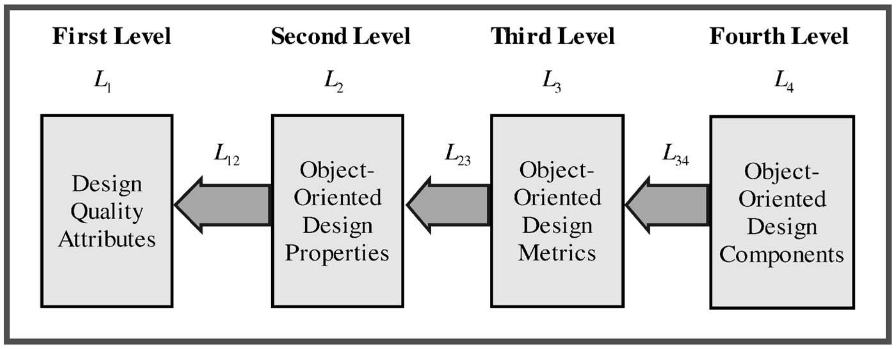
First Level
L_1
Second Level
L_2
Third Level
L_3
Fourth Level
L_4
Design Quality Attributes
L_{12}
Object-Oriented Design Properties
L_{23}
Object-Oriented Design Metrics
L_{34}
Object-Oriented Design Components

Fig. 1. Levels and links in QMOOD.

The methodology used in the development of the hierarchical **Q**uality **M**odel for **O**bject-**O**riented **D**esign (QMOOD) assessment extends Dromey's generic quality model methodology and involves the steps shown in Fig. 1 and described in the following paragraphs. Fig. 1 shows the four levels (L_1 through L_4) and three mappings (links: L_{12}, L_{23}, and L_{34}) used (to connect the four levels) in QMOOD. While defining the levels involves identifying (selecting) design quality attributes, quality carrying design properties, object-oriented design metrics, and object-oriented design components, defining the mapping involves connecting adjacent levels by relating (linking) a lower level to the next higher level.

3.1 Identifying Design Quality Attributes (L_1)

The ISO 9126 attributes—"functionality," "reliability," "efficiency," "usability," "maintainability," and "portability"—were selected as the initial set of quality attributes in the QMOOD model. This set of attributes was then individually reviewed to see if they contributed towards defining design quality and whether the set was sufficiently broad to include all aspects of design quality. The attributes "reliability" and "usability" were excluded from the set due to their obvious slant toward implementation rather than design. The term "portability" is more appropriate in the context of software implementation quality and was replaced with "extendibility," which better reflects this characteristic in designs. Similarly, the term "efficiency" was replaced with "effectiveness," which better describes this quality for designs. The term "maintainability" also implies the existence of a software product and was replaced by "understandability," which concentrates more upon design characteristics.

An important objective in adopting the object-oriented approach for design and implementation has been to develop reliable, adaptable, and flexible software systems quickly. One way to achieve this has been by encouraging reuse at all levels of development. This goal justified the inclusion of "reusability" as an important attribute of object-oriented design quality assessment.

Flexibility of software systems is also an important development and end-user characteristic. For a software system to exhibit this quality characteristic, the foundation for it should be laid in the design phase with the incorporation of software architectures which support this objective in the design. It was, therefore, decided to include "flexibility" as a quality attribute in the design model.

Thus, the initial set of design quality attributes in QMOOD are: "functionality," "effectiveness," "understandability," "extendibility," "reusability," and "flexibility." This set of attributes is broad enough to allow desirable characteristics of object-oriented systems to be identified. The validation of the model, which is described later in the paper, bears this out. However, this set of quality attributes is not exclusive, and it can be easily changed to represent different objectives and goals.

These quality attributes, just like overall quality, are in themselves abstract concepts and are therefore not directly observable. Also, like quality, there are no universally agreed-upon definitions for each of the high-level quality attributes of QMOOD. Table 1 summarizes the definitions of the quality attributes used in the development of QMOOD model.

3.2 Identifying Object-Oriented Design Properties (L_2)

Design properties are tangible concepts that can be directly assessed by examining the internal and external structure, relationship, and functionality of the design components, attributes, methods, and classes. An evaluation of a class definition for its external relationships (inheritance type) with other classes and the examination of its internal components, attributes, and methods reveals significant information that objectively captures the structural and functional characteristics of a class and its objects.

The design properties of abstraction, encapsulation, coupling, cohesion, complexity, and design size are frequently used as being representative of design quality characteristics in both structural as well as object-oriented development. Messaging, composition, inheritance, polymorphism, and class hierarchies represent new design concepts which have been introduced by the object-oriented paradigm and are thus vital to the quality of an object-oriented design. The initial version of QMOOD includes both sets of properties which are defined in Table 2.

3.3 Identifying Object-Oriented Design Metrics (L_3)

Each of the design properties identified in the QMOOD model represent an attribute or characteristic of a design that is sufficiently well defined to be objectively assessed by using one or more well-defined design metrics during the

TABLE 1
Quality Attribute Definitions

Quality Attribute	Definition
Reusability	Reflects the presence of object-oriented design characteristics that allow a design to be reapplied to a new problem without significant effort.
Flexibility	Characteristics that allow the incorporation of changes in a design. The ability of a design to be adapted to provide functionally related capabilities.
Understandability	The properties of the design that enable it to be easily learned and comprehended. This directly relates to the complexity of the design structure.
Functionality	The responsibilities assigned to the classes of a design, which are made available by the classes through their public interfaces.
Extendibility	Refers to the presence and usage of properties in an existing design that allow for the incorporation of new requirements in the design.
Effectiveness	This refers to a design's ability to achieve the desired functionality and behavior using object-oriented design concepts and techniques.

design phase. For object-oriented designs, this information should include the definition of classes, class hierarchies, and class member operation declarations, along with all the parameters types and attribute declarations.

A survey of existing design metrics [1] revealed that there are several metrics that can be modified and used in the assessment of some design properties, such as abstraction, messaging, and inheritance. However, there are several other design properties, such as encapsulation and composition, for which no object-oriented design metrics exist. Also, while metrics to assess complexity, cohesion, and coupling have already been defined [8], [20], these metrics require a nearly complete implementation of classes before they can be calculated and,

TABLE 2
Design Property Definitions

Design Property	Definition
Design Size	A measure of the number of classes used in a design.
Hierarchies	Hierarchies are used to represent different generalization-specialization concepts in a design. It is a count of the number of non-inherited classes that have children in a design.
Abstraction	A measure of the generalization-specialization aspect of the design. Classes in a design which have one or more descendants exhibit this property of abstraction.
Encapsulation	Defined as the enclosing of data and behavior within a single construct. In object-oriented designs the property specifically refers to designing classes that prevent access to attribute declarations by defining them to be private, thus protecting the internal representation of the objects.
Coupling	Defines the interdependency of an object on other objects in a design. It is a measure of the number of other objects that would have to be accessed by an object in order for that object to function correctly.
Cohesion	Assesses the relatedness of methods and attributes in a class. Strong overlap in the method parameters and attribute types is an indication of strong cohesion.
Composition	Measures the "part-of," "has," "consists-of," or "part-whole" relationships, which are aggregation relationships in an object-oriented design.
Inheritance	A measure of the "is-a" relationship between classes. This relationship is related to the level of nesting of classes in an inheritance hierarchy.
Polymorphism	The ability to substitute objects whose interfaces match for one another at run-time. It is a measure of services that are dynamically determined at run-time in an object.
Messaging	A count of the number of public methods that are available as services to other classes. This is a measure of the services that a class provides.
Complexity	A measure of the degree of difficulty in understanding and comprehending the internal and external structure of classes and their relationships.

TABLE 3
Design Metrics Descriptions

METRIC	NAME	DESCRIPTION
DSC	Design Size in Classes	This metric is a count of the total number of classes in the design.
NOH	Number of Hierarchies	This metric is a count of the number of class hierarchies in the design.
ANA	Average Number of Ancestors	This metric value signifies the average number of classes from which a class inherits information. It is computed by determining the number of classes along all paths from the "root" class(es) to all classes in an inheritance structure.
DAM	Data Access Metric	This metric is the ratio of the number of private (protected) attributes to the total number of attributes declared in the class. A high value for DAM is desired. (Range 0 to 1)
DCC	Direct Class Coupling	This metric is a count of the different number of classes that a class is directly related to. The metric includes classes that are directly related by attribute declarations and message passing (parameters) in methods.
CAM	Cohesion Among Methods of Class	This metric computes the relatedness among methods of a class based upon the parameter list of the methods [3]. The metric is computed using the summation of the intersection of parameters of a method with the maximum independent set of all parameter types in the class. A metric value close to 1.0 is preferred. (Range 0 to 1)
MOA	Measure of Aggregation	This metric measures the extent of the part-whole relationship, realized by using attributes. The metric is a count of the number of data declarations whose types are user defined classes.
MFA	Measure of Functional Abstraction	This metric is the ratio of the number of methods inherited by a class to the total number of methods accessible by member methods of the class. (Range 0 to 1)
NOP	Number of Polymorphic Methods	This metric is a count of the methods that can exhibit polymorphic behavior. Such methods in C++ are marked as virtual.
CIS	Class Interface Size	This metric is a count of the number of public methods in a class
NOM	Number of Methods	This metric is a count of all the methods defined in a class.

therefore, cannot be used in QMOOD. This led to the definition of five new metrics, the data access metric (DAM), the direct class coupling metric (DCC), the cohesion among methods of class metric (CAM), the measure of aggregation metric (MOA), and the measure of functional abstraction metric (MFA) that could be calculated from design information only. The complete suite of metrics used in QMOOD are listed and described in Table 3.

3.4 Identifying Object-Oriented Design Components (L_4)

The design components that are identifiable and define the architecture of an object-oriented design are objects, classes, and the relationships between them. Objects encapsulate data structures which represent the attributes of a class of objects. A set of operations (methods) defined in classes can operate on data encapsulated by the object. The quality of an object is determined by its constituents, which includes attributes, methods, or other objects (composition). Another component that can be identified in object-oriented designs are generalization-specialization structures, or class hierarchies that organize families of related classes. Thus, a set of components which can help analyze, represent, and implement an object-oriented design should include attributes, methods, objects (classes), relationships, and class hierarchies.

Generally, all object-oriented programming languages provide syntactic constructs to represent these fundamental design components. Since an object-oriented programming language (C++) was used for design representation, the quality of a design can be readily assessed by evaluating the quality of these automatically detectable components. An automated tool, QMOOD++ [2], makes it easy to collect metrics data from the components of a design represented in C++.

3.4.1 *Identifying Quality-Carrying Properties of Components*

Arriving at a set of quality-carrying properties for each design component is an empirical process [10]. The process was guided by a series of questions, such as: "What role does the component play in the design (purpose) ?" "What is the significance of the component in the design ?" "How is the component represented ?" "What are the different forms of the component ?"

Important considerations in including a quality-carrying property were the requirement that the information to evaluate the property should be available during the design phase, and that there is enough information in the representation of the component to identify and evaluate the quality carrying property in an unambiguous way. The quality carrying properties of the attributes, methods, and class components that were used for design quality assessment were identified [1] in the context of a C++ representation.

Attributes are the most fundamental components in an object definition and their representation is directly supported in object-oriented programming languages. A set of attributes' (data) declaration properties that can directly influence the quality of an object and, therefore, the overall design quality, should include: name (self-descriptive), encapsulation (public, private, protected), qualifiers (static/constant), initialization (on creation ? yes/no), size (in bytes), type (primitive or user defined), relationship (yes/no), and enumeration (value or pointer/reference).

Method declarations are also directly supported in programming languages. A set of identifiable and assessable properties of a method that, either directly or indirectly, influences the quality of the class inside which it is declared should include: name (self-descriptive), encapsulation (private, protected, public), parameter types, number of parameters, parameter passing mechanism (by value, reference or address), resolution (static/dynamic or nonvirtual/virtual), and type (constructor, destructor, constant, inline, static).

Classes in most object-oriented programming languages are described by using syntactic constructs that are easily identifiable. The "Goodness" of a class can be analyzed based on its constituents (attributes and methods), and its structural and communicational relationship (interaction) with other classes. A set of class properties that may influence the overall quality of a design includes: name (self-descriptive), inheritance type and encapsulation (public, private, or protected inheritance), inheritance (base, derived, single, multiple, abstract, or concrete), number of parents, number of children, depth of inheritance, template class, class (object) size, number of methods (public, private, protected), number of attributes, number of virtual methods, number of constructors, destructors, operators, constant and static methods, number of inline functions, coupling, and cohesion.

3.5 Mapping Quality-Carrying Component Properties to Design Properties (L_{34})

The quality-carrying properties of the design components are classified based on the design properties that they influence. While the set of quality-carrying properties of the fundamental (attributes, methods, and classes) components is large, it is highly overlapping. For instance, attributes, methods, and class components have a *name* that is a quality-carrying property. When *names* are self-descriptive, they help in better understandability and, therefore, influence (reduce) the design property complexity. The quality-carrying property, *encapsulation,* identified for attributes, methods, and classes, is the same as the general design property, encapsulation. Similarly, the remaining quality-carrying properties identified for attributes, methods, and classes can all be grouped into the smaller set of eleven fundamental design properties described in Table 2.

3.6 Assigning Design Metrics to Design Properties (L_{23})

The goal of this assignment was to find a reasonably small set of metrics which contain sufficient information to allow accurate determination of design properties. The metrics *Design Size in Classes* (DSC) and *Number of Hierarchies* (NOH) are used to assess the two design properties *Design Size* and *Hierarchies* in QMOOD. Abstraction refers to the generalization-specialization structure of a design and is assessed by the *Average Number of Ancestors* (ANA) metric. The definition of the encapsulation design property in Table 2 refers to access control of attribute declarations in a class which is reflected in the description of the DAM metric. The *Direct Class Coupling* (DCC) metric is a measure of the classes directly related to a class and, thus, is used to assess the *Coupling* design property. The metrics *Cohesion Among Method of Class* (CAM) [3], *Measure of Aggregation* (MOA), and *Class Interface Size* are used to measure *Cohesion*, *Aggregation*, and *Messaging* design properties. *Inheritance* refers to the extent of reuse of functionality (measured by the MFA metric) that can be achieved by creating subclasses of existing classes and, therefore, the MFA metric was used as a measure for the *Inheritance* design property in QMOOD. For an object-oriented design represented in C++ syntax, the design property of *Polymorphism* is a measure of the *Virtual* methods in a class and is assessed by the metric *Number of Polymorphic Methods* (NOP). The *Number of Methods* (NOM) metric has been used as a measure of class complexity by Chidamber and Kemerer [8] in the Weighted Methods Per Class (WMC) metric of their design metric suite. When all methods are weighted equally, the WMC metric has the same measure as the *Number of Methods* (NOM) in a class. Table 4 summarizes the design metrics used to assess the eleven design properties of Table 3.

3.7 Linking Design Properties to Quality Attributes (L_{12})

In order to incorporate various views as to how design properties may influence design quality, an extensive review of object-oriented development books [6], [9], [13], [14], [16], [24] and publications [4], [8], [10], [11], [17], [18], [25] was conducted in order to develop a basis for relating product characteristics to quality attributes.

The reviewed information indicates that the abstraction design property has a significant influence on the flexibility, effectiveness, functionality, and extendibility quality

TABLE 4
Design Metrics for Design Properties

Design Property	Derived Design Metric
Design Size	Design Size in Classes (DSC)
Hierarchies	Number of Hierarchies (NOH)
Abstraction	Average Number of Ancestors (ANA)
Encapsulation	Data Access Metric (DAM)
Coupling	Direct Class Coupling (DCC)
Cohesion	Cohesion Among Methods in Class (CAM)
Composition	Measure of Aggregation (MOA)
Inheritance	Measure of Functional Abstraction (MFA)
Polymorphism	Number of Polymorphic Methods (NOP)
Messaging	Class Interface Size (CIS)
Complexity	Number of Methods (NOM)

attributes of a design. Encapsulation is viewed to promote flexibility, reusability, and understandability. While low coupling is considered good for understandability, extendibility, and reusability, higher measures of coupling are viewed to adversely influence these quality attributes. Cohesion is viewed to have a significant effect on a design's understandably and reusability.

Objects communicate by message passing and, therefore, messaging directly influences functionality and effectiveness and helps promotes reusability. While the use of inheritance promotes internal reusability, functionality, extendibility, and effectiveness, it has the potential to adversely influence flexibility and understandability. Similarly, while the careful use of object composition can significantly increase internal reusability, functionality, and flexibility, its excessive and incorrect usage can make a design harder to understand. The use of composition may also influence effectiveness and extendibility. While the use of polymorphism is viewed to increase the flexibility, extendibility, effectiveness, and functionality attributes of a design, it is also viewed to make designs harder to understand. Complexity is viewed as an indication of the understandability of a design. In general, the more complex a design, the harder it is to understand, which can also influence the designs flexibility and reusability.

Table 5 shows the influence of each of the design properties on the quality attributes. An up arrow symbol (↑) indicates that the design property has a positive influence on the quality attribute, and a down arrow symbol (↓) indicates that the design property has a negative influence on the quality attribute.

3.7.1 Weighting the Linkages from Design Properties to Quality Attributes

Based upon the design property to quality attribute relationship matrix in Table 5, the relative significance of individual design properties that influence a quality attribute is weighted proportionally so that the computed values of all quality attributes have the same range. A range of 0 to ±1 was selected for the computed values of the quality attributes.

For an initial pass in setting model parameters, a weighted value of +1 or +0.5 was used for the positive influences, and a value of -1 or -0.5 was used for the negative influences. The initial weighted values of design property influences on a quality attribute were then proportionally changed to ensure that the sum of the new weighted values of all design property influences on a quality attribute added to ±1, the selected range for the computed values of quality attribute. This results in the relationships shown in Table 6. This scheme for weighting the influences was chosen because it was simple and straightforward to apply.

3.8 Refining and Adapting the Model

The QMOOD quality model allows changes to be easily made to the model to address different weightings, other perspectives, and new goals and objectives. At the lowest level, the metrics used to assess design properties may be changed, or a different set of design properties may be used to assess quality attributes, or even the set of quality attributes to be assessed can be changed. The influence relationships from design properties to quality attributes

TABLE 5
Quality Attributes—Design Property Relationships

	Reusability	Flexibility	Understandability	Functionality	Extendibility	Effectiveness
Design Size	↑			↑		
Hierarchies				↑		
Abstraction					↑	↑
Encapsulation		↑	↑			↑
Coupling						
Cohesion	↑		↑	↑		
Composition		↑				↑
Inheritance					↑	↑
Polymorphism		↑		↑	↑	↑
Messaging	↑			↑		
Complexity						

TABLE 6
Computation Formulas for Quality Attributes

Quality Attribute	Index Computation Equation
Reusability	-0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * Design Size
Flexibility	0.25 * Encapsulation - 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism
Understandability	-0.33 * Abstraction + 0.33 * Encapsulation - 0.33 * Coupling + 0.33 * Cohesion - 0.33 * Polymorphism - 0.33 * Complexity - 0.33 * Design Size
Functionality	0.12 * Cohesion + 0.22 * Polymorphism + 0.22 Messaging + 0.22 * Design Size + 0.22 * Hierarchies
Extendibility	0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism
Effectiveness	0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism

and the weights can be experimented with to best reflect organizational objectives and goals.

4 Model Validation and Results

The validation of the QMOOD quality model was carried out at two levels: validation of the individual quality attributes' effectiveness and validation of the overall software quality estimation.

4.1 QMOOD Individual Attribute Validation

In order to verify that the computed values of the quality attributes are within valid ranges, it was desirable that the quality attribute values be computed for several designs and compared with expected results. Since the design metrics listed in Table 3 will vary greatly across design domains, it was decided that any comparisons be done between designs that had been developed for similar requirements and objectives.

4.1.1 Selecting a Validation Suite

Several versions of two popular WINDOWS application frameworks, Microsoft Foundation Classes (MFC), and Borland Object Windows Library (OWL), provided readily-available designs that have individually been developed with consistent requirements and objectives. The MFC and OWL framework solutions primarily provide capabilities for the development of WINDOWS-based applications. They are very successful and popular commercial object-oriented systems that are extensively used in real-world software development and several versions of design exist for comparison within each framework. Five publicly released versions of MFC and four versions of OWL, Table 7, were evaluated using the suite of design metrics in Table 3 and the quality attributes in Table 6.

4.1.2 Expected Results

Since the quality attributes in QMOOD are important development characteristics of most software systems, they should indicate desirable trends that are normally expected of reusable, flexible, extendible, and effective frameworks applications.

Typically, there are two overriding reasons for the release of new versions of an existing software product; the addition of new features or the incorporation of fixes to

TABLE 7
Windows Frameworks Analyzed to Study Quality Attributes

Windows Application Frameworks	Release Date
Microsoft Foundation Classes, MFC 1.0	1992, with C/C++ 7.0 and Windows NT
Microsoft Foundation Classes, MFC 2.0	Early 1993, with Visual C++ version 1.0 for Windows
Microsoft Foundation Classes, MFC 3.0	Late 1994, with Visual C++ version 2.0
Microsoft Foundation Classes, MFC 4.0	Late 1995, with Visual C++ 4.0
Microsoft Foundation Classes, MFC 5.0	Early 1997, with Visual C++ 5.0
Object Windows Library, OWL 4.0	Mid 1994, with Borland C++ 4.0
Object Windows Library, OWL 4.5	Late 1995, with Borland C++ 4.5
Object Windows Library, OWL 5.0	Mid 1996, with Borland C++ 5.0
Object Windows Library, OWL 5.2	Mid 1997, with Borland C++ 5.2

TABLE 8
Actual Metric Values for MFC and OWL Frameworks

METRIC	MFC 1.0	MFC 2.0	MFC 3.0	MFC 4.0	MFC 5.0	OWL 4.0	OWL 4.5	OWL 5.0	OWL 5.2
Design Size	72.00	92.00	132.00	206.00	233.00	82.00	142.00	357.00	356.00
Hierarchies	1.00	1.00	1.00	5.00	6.00	12.00	16.00	29.00	27.00
Abstraction	1.68	2.11	2.45	2.33	2.30	1.00	0.96	1.60	1.55
Encapsulation	0.61	0.48	0.54	0.56	0.58	0.71	0.66	0.62	0.63
Modularity	0.80	0.90	0.92	0.97	1.02	1.30	1.30	1.40	1.40
Coupling	6.03	7.59	9.02	9.33	8.94	1.90	2.30	4.70	4.71
Cohesion	0.20	0.15	0.16	0.15	0.16	0.39	0.41	0.38	1.00
Aggregation	0.26	0.91	1.39	1.38	1.45	0.37	0.70	1.40	1.36
Inheritance	0.07	0.35	0.43	0.50	0.49	0.43	0.39	0.45	0.45
Polymorphism	6.83	18.20	19.30	28.30	28.60	1.90	3.10	7.00	6.78
Messaging	44.60	75.60	95.00	114.00	109.00	28.00	24.00	37.00	54.60
Complexity	56.50	102.00	126.00	150.00	144.00	30.00	29.00	56.00	36.80

errors previously discovered. Generally, the early versions of new software are features released as the software is modified to enhance capabilities and add new features or incorporate additional requirements. These early releases may also improve the usability and user friendliness of the software product. As a result, these early versions of the software may have significant changes in the structure of the software. The "overall quality" of these early releases have generally been observed to be significantly better than their predecessors. After the first few initial releases of the software, new versions of mature software are generally released to make available "bug" fixes, and improve the robustness and reliability of the software. These releases may also attempt to reduce the complexity of the software. While these new releases of mature software have also been observed to improve the quality of a product over its predecessor, the improvements are small and not usually very dramatic.

In order to obtain a measure of the validity of the six high-level quality attributes in the QMOOD model, it was expected that the quality characteristics for each version of the two framework systems evaluated should match the generally expected trends from one version to the next. Specifically, it was expected that the quality attributes reusability, flexibility, functionality, extendibility, and effectiveness should increase from one release to the next. Also, for the early releases of a framework system, it was expected that understandability should initially decrease because the initial releases would add significant new features and make extensions to the capabilities of the frameworks by adding new classes and methods to existing classes from one release to the next. The early structure of the framework may also be unstable and undergo significant reworks during the initial releases. After several initial releases, a framework is expected to mature, having incorporated most required capabilities. Releases of a mature framework are expected to reverse the trend of the understandability measure since development efforts in mature frameworks can be expected to improve their usability, reduce complexity, and make them easier to understand.

4.1.3 Gathering and Normalizing Metric Data

The framework designs were analyzed by the tool, QMOOD++ [2], using the C++ class specifications as inputs. The metric data collected for the eleven metrics of Table 3 for the five versions of MFC and four versions of OWL is shown in Table 8.

Because actual metric values of different ranges are combined in the computation of the quality attribute indices, the metric values for each framework were normalized with respect to the metrics' values in the first version of the frameworks. The normalization of the two frameworks was done separately because the systems have been developed by two independent vendors and were felt to be different to permit any meaningful metrics based comparison between them. The actual metric measures of the five MFC and four OWL systems shown in Table 8 are replaced by their normalized values (computed by dividing a metric value with the metrics' value in the first version) in Table 9. This normalization (and, therefore, comparison) is valid and meaningful because the normalization is based on the different versions of the same framework solution. However, this technique can not be extended to cases in which dissimilar systems influence the normalization.

4.1.4 Analyzing and Comparing with Expected Results

Table 10 shows the computed values of the six quality attributes for the two framework systems based on the normalized values of the metrics in Table 9. Fig. 2 shows a plot of the reusability, flexibility, functionality, extendibility, and effectiveness quality attribute values (in Table 9) for the MFC framework versions. The expected increase in values of these quality attributes is compatible with the hypothesis that these quality attributes should improve with new releases in framework-based systems.

Understandability agrees with the expectation that early releases characterize a developing framework that takes on additional functionality in each new revision and is therefore expected to be harder to learn and understand. It decreases significantly in every version of MFC from 1.0 to 4.0. For MFC version 5.0, however, the understandability measure is only slightly greater than in version 4.0. This corroborates the observation that MFC version 5.0 is a mature framework that is characterized by the addition of

TABLE 9
Normalized Metric Values for MFC and OWL Frameworks

METRIC	MFC 1.0	MFC 2.0	MFC 3.0	MFC 4.0	MFC 5.0	OWL 4.0	OWL 4.5	OWL 5.0	OWL 5.2
Design Size	1.00	1.28	1.83	2.86	3.24	1.00	1.73	4.35	4.34
Hierarchies	1.00	1.00	1.00	5.00	6.00	1.00	1.33	2.42	2.25
Abstraction	1.00	1.26	1.46	1.39	1.37	1.00	0.97	1.56	1.55
Encapsulation	1.00	0.79	0.88	0.92	0.94	1.00	0.94	0.88	0.89
Modularity	1.00	1.13	1.15	1.21	1.28	1.00	1.01	1.04	1.04
Coupling	1.00	1.26	1.50	1.55	1.48	1.00	1.21	2.53	2.52
Cohesion	1.00	0.74	0.81	0.79	0.83	1.00	1.00	1.00	1.00
Aggregation	1.00	3.46	5.27	5.23	5.49	1.00	1.90	3.85	3.72
Inheritance	1.00	5.39	6.55	7.61	7.57	1.00	0.92	1.06	1.06
Polymorphism	1.00	2.66	2.83	4.14	4.19	1.00	1.66	3.71	3.59
Messaging	1.00	1.81	2.23	2.65	2.55	1.00	0.95	1.83	1.80
Complexity	1.00	1.70	2.13	2.56	2.44	1.00	0.87	1.34	1.33

TABLE 10
Computed Quality Attribute Indices for MFC and OWL

Quality Index	MFC 1.0	MFC 2.0	MFC 3.0	MFC 4.0	MFC 5.0	*OWL 4.0*	*OWL 4.5*	*OWL 5.0*	*OWL 5.2*
Reusability	1.00	1.41	1.86	2.57	2.73	*1.00*	*1.29*	*2.71*	*2.69*
Flexibility	1.00	2.94	3.89	4.53	4.71	*1.00*	*1.72*	*3.37*	*3.24*
Understandability	-0.99	-2.18	-2.66	-3.56	-3.61	*-0.99*	*-1.48*	*-3.83*	*-3.77*
Functionality	1.00	1.57	1.83	3.32	3.61	*1.00*	*1.37*	*2.83*	*2.76*
Extendibility	1.00	4.03	4.67	5.80	5.82	*1.00*	*1.17*	*1.90*	*1.84*
Effectiveness	1.00	2.71	3.40	3.86	3.91	*1.00*	*1.28*	*2.21*	*2.16*

fewer new features than previous releases and more emphasis on improvements in order to make the system easier to understand and use.

Similar results are observed with the computed quality attribute values of the OWL framework versions (Table 10). Reusability, flexibility, functionality, extendibility, and effectiveness attributes (plotted in Fig. 3) agree with the expected increase in values of these attributes in the early versions (4.0, 4.5, 5.0) of the OWL frameworks. The values of understandability quality attribute also performs as expected in the early versions. The values of the six quality attributes in OWL version 5.2 show no significant change (only a very small decrease) from the computed values of the quality attributes in version 5.0.

4.2 QMOOD Validation

Another aspect of evaluating QMOOD was to assess how well the model is able to predict the "overall quality" of an object-oriented software design. The internal characteristics of a design can be significantly different based on the domain and the objectives of the design. Because the characteristics of a design influence the quality attributes and, therefore, the overall quality, it was only meaningful to compare software designs that have been developed for similar or closely related objectives. Therefore, the validation

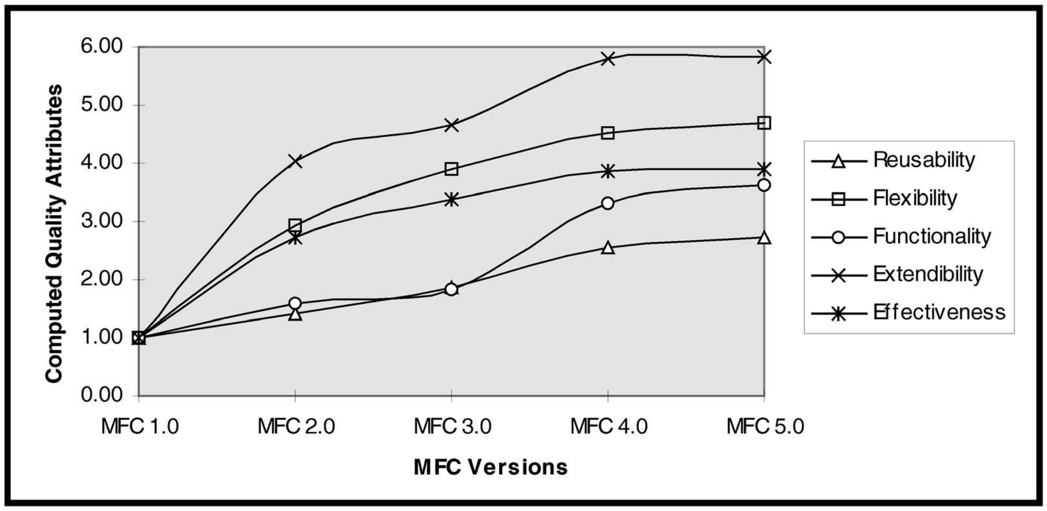

Fig. 2. Plot of computed quality attribute values for MFC versions.

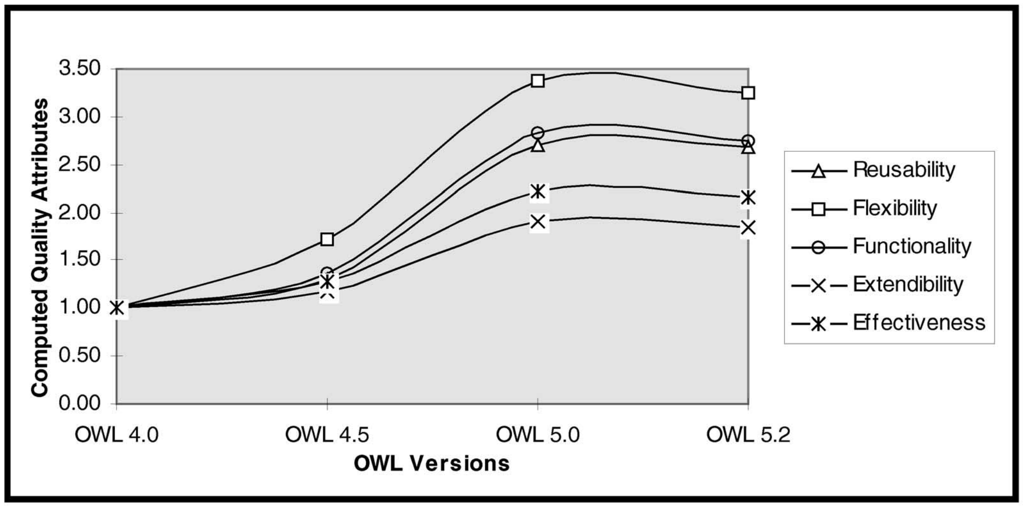

Fig. 3. Plot of computed quality attribute values for OWL versions.

of the predictability of QMOOD required that several separate object-oriented designs that had been developed for the same set of requirements be evaluated and compared. This assessment of the overall quality of designs determined by QMOOD needed to agree with the generally accepted requirements or characteristics of overall quality designs as perceived by analysts, developers, and customers. Based on these objectives, the following process was adopted for the validation of the overall software quality assessment produced by QMOOD.

4.2.1 Selecting a Model Validation Suite

Since perceptions of "quality" vary widely and depend upon the assessors' viewpoint, it thus becomes necessary to employ a variety of human evaluators who can factor those variations into an evaluation. A medium sized C++ project was used in the validation of QMOOD, using the overall or total quality index (TQI) measure [1]. This project was selected because several designs of the project that addressed the same set of requirements were easily available. The size of these projects, reflected by the number of classes (10 to 29), made them an ideal candidate for both QMOOD and human-evaluator-based assessment. The requirements for the C++ project were to implement an interpreter for a fictitious programming language named "COOL." An initial set of requirements (revision 1.0) was provided at the start of the project and was followed by revision 2.0 which defined extensions to the original requirements. The projects were independently completed over a period of four months. The deliverables included a design document and an implementation that met both revision 1.0 and revision 2.0 requirements [1]. A suite of 14 projects were developed by different individuals for the study.

4.2.2 Evaluator's Assessment of "COOL" Projects Overall Quality

A group of 13 independent evaluators was used to study the quality of the 14 "COOL" projects in the validation suite. All evaluators had two to seven years experience in commercial software development, had knowledge of the object-oriented paradigm, and had developed software using C++. The study was done over a period of four months. The participants were initially required to independently develop a set of design and implementation heuristics that indicate or influence the development of quality object-oriented software [1]. The participants then assigned their assessment heuristics to one or more of the six high-level quality attributes in Table 1, i.e., reusability, flexibility, understandability, functionality, extendibility, and effectiveness. For example, one of the most common heuristics used by the evaluators was to examine the access protections defined on the data attribute declarations in classes. This heuristic was then assigned by the evaluators to assess quality attributes such as flexibility or understandability or both.

The evaluators then analyzed each project's design and reviewed its implementation, looking for evidence that indicated conformance or nonconformance to their set of heuristics. Participants scored each heuristic on a ordinal scale of 0, 1, or 2; 0 indicating no evidence to satisfy the heuristic, 1 to indicate partial evidence, and 2 to indicate that there was sufficient evidence that the heuristic's criteria's were satisfied. The heuristic-based evaluation scores for all six high-level quality attributes were summed separately for each project in the validation suite and normalized to give the evaluator's Total Quality Score (TQS) for a project. The projects were ranked 1 through 14 based on their decreasing TQS scores by each evaluator as shown in Table 11.

The rankings of the validation suite projects indicated a close agreement between the assessments done by the evaluators. There was a very strong agreement between the evaluators in the assessment of projects 5 and 14. Also, projects 1, 2, 3, 7, and 10 are very closely ranked by at least 10 of the 13 evaluators. Results of the evaluations also indicate that there was considerable disagreement in the rankings of project number 13 between the evaluators. This disagreement in the rank can be attributed to the set of heuristics used by the evaluators. The number of classes in this project (Table 12) is significantly less than the other

TABLE 11
Evaluators and Model Rankings of "COOL" Projects

PROJECT	QMOOD	EVALUATOR RANKINGS												
NUMBER	**RANKING**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
1	*13*	12	9	11	11	13	12	12	13	13	13	11	8	9
2	*9*	13	12	12	12	12	6	13	6	10	12	12	12	10
3	*3*	2	6	2	1	7	1	4	1	2	7	4	2	1
4	*12*	9	7	9	10	2	9	10	9	6	11	10	5	6
5	*1*	1	1	1	2	1	7	1	2	1	2	2	3	2
6	*5*	8	11	10	7	9	11	6	12	11	9	3	11	12
7	*11*	4	13	13	13	8	13	11	8	12	4	1	13	8
8	*8*	3	8	9	8	11	10	9	10	8	3	7	6	11
9	*7*	5	10	7	9	10	3	5	4	5	6	6	7	7
10	*2*	6	2	3	3	4	5	2	3	3	1	5	1	5
11	*4*	11	3	5	6	5	8	3	11	9	8	8	10	13
12	*6*	7	5	6	4	3	2	8	5	7	5	9	9	3
13	*10*	10	4	4	5	6	4	7	7	4	10	13	4	4
14	*14*	14	14	14	14	14	14	14	14	14	14	14	14	14

TABLE 12
COOL Project Design Property Metric Values

METRIC	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Design Size	12.0	10.0	15.0	22.0	24.0	26.0	29.0	23.0	20.0	20.0	24.0	25.0	12.0	3.00
Hierarchies	1.0	1.0	1.0	1.0	1.0	2.0	2.0	4.0	1.0	1.0	2.0	1.0	1.0	0.00
Abstraction	0.7	1.2	2.1	1.2	1.4	0.7	0.9	1.0	1.3	1.4	0.7	1.3	1.3	0.00
Encapsulation	1.0	0.6	0.8	0.5	0.9	0.8	1.0	0.8	0.8	1.0	1.0	0.6	0.7	0.63
Modularity	0.7	0.5	0.4	0.4	0.5	0.4	0.6	0.5	0.4	0.4	0.3	0.3	0.4	0.26
Coupling	1.5	0.8	0.0	1.0	0.6	1.2	1.2	1.8	1.8	0.9	1.1	1.1	1.7	0.67
Cohesion	0.5	0.6	0.6	0.4	0.7	0.6	0.4	0.5	0.3	0.5	0.6	0.6	0.5	0.53
Aggregation	0.3	0.0	0.0	0.4	0.1	0.8	0.2	1.1	0.3	0.1	0.6	0.6	0.1	0.67
Inheritance	0.2	0.4	0.8	0.5	0.7	0.6	0.4	0.5	0.5	0.7	0.5	0.2	0.4	0.00
Polymorphism	2.4	3.8	1.5	1.5	2.5	2.3	2.8	3.3	5.1	4.3	2.3	3.7	5.6	0.00
Messaging	8.9	6.9	11.0	9.6	7.5	9.0	8.4	14.0	17.0	12.0	8.1	7.2	9.8	5.30
Complexity	8.9	7.7	9.9	9.6	7.5	9.0	8.4	14.0	17.0	12.0	8.1	7.2	9.8	5.30

projects, and evaluators who did not include a heuristic for the number of classes in their evaluations ranked the project higher than evaluators who included a heuristic for this design size in their assessments.

4.2.3 Evaluating "COOL" Project Designs Using QMOOD

The 14 projects of the validation suite were also evaluated using QMOOD++. The metric values collected for the 11 metrics used to assess the design properties are shown in Table 12. Since the metric values that are to be combined are all not on a uniform scale, the values of the metrics were normalized by ranking them to ensure a meaningful evaluation. The ranks were determined based on a metric's (increasing) relative value among the 14 projects. Because relative ranking is used for normalization in this validation, the normalizing range equals the number of projects in the validation suite. Table 13 shows the normalized values (from 1 through 14) of the metrics. This normalization is meaningful because all the projects used in the normalization address the same set of requirements and are thus comparable.

TABLE 13
COOL Project Normalized Design Property Metric Values

Metrics	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Design Size	4	2	5	8	11	13	14	9	7	7	11	12	4	1
Hierarchies	10	10	10	10	10	13	13	14	10	10	13	10	10	1
Abstraction	3	8	14	8	13	2	5	6	11	13	4	11	11	1
Encapsulation	14	3	8	1	10	7	14	6	9	14	14	2	5	4
Modularity	14	11	6	5	10	4	13	12	7	9	2	3	9	1
Coupling	11	4	1	6	2	10	10	14	14	5	8	8	12	3
Cohesion	5	11	13	3	14	10	2	5	1	6	12	10	8	8
Aggregation	7	2	2	9	5	13	6	14	8	3	11	10	4	12
Inheritance	3	5	14	9	13	11	4	7	9	12	10	2	6	1
Polymorphism	6	11	3	3	7	5	8	9	13	12	5	10	14	1
Messaging	7	2	11	9	4	8	6	13	14	12	5	3	10	1
Complexity	7	4	11	9	3	8	6	13	14	12	5	2	10	1

TABLE 14
COOL Project Design Quality Attribute Indices

Quality	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Reusability	4.0	3.7	11.0	7.7	10.5	10.5	8.0	8.7	7.2	9.7	9.0	8.0	6.0	2.2
Flexibility	7.2	6.2	4.2	4.7	8.0	8.2	8.0	9.5	9.2	9.7	9.5	8.5	7.2	6.7
Understandability	-3.9	-4.9	-4.2	-9.9	-3.9	-6.9	-8.9	-13.2	-16.1	-9.5	-2.3	-10.2	-12.5	-1.6
Functionality	6.5	6.8	7.9	6.9	8.7	9.7	9.2	10.5	9.8	9.7	8.9	8.9	9.3	1.8
Extendibility	0.5	10.0	15.0	7.0	15.5	4.0	3.5	4.0	9.5	16.0	5.5	7.5	9.5	0.0
Effectiveness	6.6	5.8	8.2	6.0	9.6	7.6	7.4	8.4	10.0	10.8	8.8	7.0	8.0	3.8
TQI	**20.9**	**27.6**	**42.1**	**22.5**	**48.3**	**33.2**	**27.2**	**27.9**	**29.6**	**46.4**	**39.4**	**29.6**	**27.5**	**16.2**

4.2.4 Computing Total Quality Index for "COOL" Projects

The values for the six high-level quality attributes are computed using the equations in Table 6 and shown in Table 14. The values of the six quality attribute measures for each project in Table 14 are summed to provide the Total Quality Index (TQI) for the project, which was used as a basis for (decreasingly) ranking the project designs (1 through 14) as shown in the column labeled, QMOOD RANKING, of Table 11.

4.2.5 Spearman's Rank Correlation Coefficient Test For Result Comparison

The Spearman's rank correlation coefficient, r_s, was used to test the significance of the correlation between QMOOD's design-based quality assessment and the evaluator's implementation-based assessment of the projects in the validation suite. It provides a nonparametric significance test that works well with ranked data without precise proportional scaling and can be used to detect relationships other than linear ones. The cutoffs used correspond to a level of significance of $\alpha = 5\%(0.05)$. For the validation suite with 14 projects ($n = 14$) a cutoff of ± 0.55 was required.

If E_1 and E_2 are two independent evaluations of "n" items that are to be correlated, then the values of E_1 and E_2 are ranked (either increasingly or decreasingly) from 1 to "n" according to their relative size within the evaluations. For each E_1, E_2 pair in the relative rankings, the difference in the ranks "d" is computed. The sum of all the d^2s, denoted by $\sum d^2$ is used to compute r_s using the following formula:

$$r_s = 1 - \frac{6\sum d^2}{n(n^2-1)} \qquad -1.00 \leq r_s \leq +1.00.$$

4.2.6 Hypothesis for Model Validation

$H_0 : \rho = 0$. Null hypothesis, $\rho = 0$. No significant correlation exists between relative rankings of the design's quality indicators evaluated using QMOOD and an assessment of the design and its' implementation done by human evaluators.

$H_1 : \rho \neq 0$. The alternative hypothesis, $\rho \neq 0$. Significant positive correlation exists between relative rankings of the design's quality indicators evaluated using QMOOD, and an assessment of the design and its' implementation done by human evaluators. For significance level of 5 percent (i.e., $\alpha = 0.05$).

4.2.7 Comparing Results for Significance of r_s

Table 15 shows the correlation values between QMOOD-determined design quality rankings of the 14 COOL projects and the 13 evaluators' assessments of overall quality of these projects based on design and implementation shown in Table 11. Pairs of QMOOD—Evaluator with correlation values (r_s) above cutoff (0.55) are "checked" in the Table 15.

An examination of the heuristics used by the two evaluators, 12 and 13, showed that they did not include heuristics to incorporate the effect of granularity of the classes in the designs, i.e., the influence of the number of classes on the quality of the design. This is considered significant because all the designs and implementations evaluated address the same requirements. Typically, a solution with more classes is likely to be more easily extendible and reusable when compared with a solution that implements the requirements using a few classes. This was felt to account for the low correlation (below cutoff) with these two evaluators.

5 Conclusions and Future Research Directions

The hierarchical model, QMOOD, for assessment of high-level design quality attributes in object-oriented designs has been developed and validated using structural and functional information from object-oriented designs of several

TABLE 15
QMOOD and Evaluator Ranking Correlation's of COOL Projects

	1	2	3	4	5	6	7	8	9	10	11	12	13
$\sum d^2$	180	146	91	68	198	196	42	154	142	136	168	214	260
r_s	0.60	0.68	0.80	0.85	0.56	0.57	0.91	0.66	0.69	0.70	0.63	0.53	0.43
$r_s >$ 0.55	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	X

releases of two large and popular commercial framework systems. The model's ability to estimate overall design quality from design information has also been demonstrated using several functionally equivalent projects where the overall quality estimate computed by the model had statistically significant correlation with the assessment of overall project quality characteristics determined by independent evaluators. The use of the QMOOD++ tool to collect data for design metrics and compute quality attributes was found to be nonintrusive and easy to apply.

Even though the model used simple and straightforward assumptions, the correlation with a limited set of projects were high. This gives an indication that models of this type can be effectively used in monitoring the quality of software product. While the issues of quality assessment still remain somewhat elusive, the QMOOD model also offers a tool that will allow various models and approaches to be quickly evaluated on real-world projects, helping accelerate the convergence of differing quality views.

REFERENCES

[1] J. Bansiya, "A Hierarchical Model For Quality Assessment Of Object-Oriented Designs," *PhD Dissertation,* Univ. of Alabama in Huntsville, 1997.

[2] J. Bansiya and C. Davis, "Automated Metrics for Object-Oriented Development," *Dr. Dobb's J.,* vol. 272, pp. 42-48, Dec. 1997.

[3] J. Bansiya and C. Davis, "Class Cohesion Metric For Object-Oriented Designs," *J. Object-Oriented Programming,* vol. 11, no. 8, pp. 47-52, Jan. 1999.

[4] B. Basili, L. Briand, and W.L. Melo, "A Validation of Object-Oriented Metrics as Quality Indicators," *IEEE Trans. Software Eng.,* vol. 22, no. 10, pp. 751-761, Oct. 1996.

[5] B.W. Boehm, *Characteristic of Software Quality.* TRW Inc., 1978.

[6] G. Booch, *Object-Oriented Analysis and Design,* second ed. The Benjamin/Cummings Publishing, 1994.

[7] L.C. Briand, S. Morasca, and V.R. Basili, "Property Based Software Engineering Measurement," *IEEE Trans. Software Eng.,* vol. 22, no. 1, pp. 68-86, Jan. 1996.

[8] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite For Object-Oriented Design," *IEEE Trans. Software Eng.,* vol. 20, no. 6, pp. 476-493, June 1994.

[9] P. Coad and E. Yourdan, *Object-Oriented Analysis and Design,* New Jersey: Yourdan Press Computing Series, 1991.

[10] G.R. Dormey, "A Model for Software Product Quality," *IEEE Trans. Software Eng.,* vol. 21, no. 2, pp. 146-162, Feb. 1995.

[11] G.R. Dormey, "Cornering the Chimera," *IEEE Software,* vol. 13, no. 1, pp. 33-43, 1996.

[12] G.R. Dromey and A.D. McGettrick, "On Specifying Software Quality," *Software Quality J.,* vol. 1, no. 1, pp. 45-74, 1992.

[13] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach.* PWS Publishing, 1997.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns; Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[15] M. Hitz and B. Montazeri, "Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective," *IEEE Trans. Software Eng.,* vol. 22, no. 4, pp. 67-271, Apr. 1996.

[16] S.H. Kan, *Metrics and Models in Software Quality Engineering.* Addison Wesley, 1995.

[17] B. Kitchenham and S.L. Pfleeger, "Software Quality: The Elusive Target," *IEEE Software,* vol. 13, no. 1, pp. 12-21, 1996.

[18] B. Kitchenham, "Towards a Constructive Quality Model," *Software Eng. J.,* pp. 105-112, July 1987.

[19] B. Kitchenham, S.L. Pfleeger, and N. Fenton, "Towards a Framework for Software Validation Measures," *IEEE Trans. Software Eng.,* vol. 21, no. 12, pp. 929-944, Dec. 1995.

[20] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability," *The J. Systems and Software,* vol. 23, pp. 111-122, 1993.

[21] J.A. McCall, P.K. Richards, and G.F. Walters, "Factors in Software Quality," vols. 1, 2, and 3, *AD/A-049-014/015/055, Nat'l Tech. Information Service,* Springfield, Va., 1977.

[22] M. Paulk, B. Curtis, M. Chrissis, and C. Weber, "Capability Maturity Model, Version 1.1," *IEEE Software,* pp. 18-27, July 1993.

[23] S.L. Pfleeger, R. Jeffery, B. Curtis, and B. Kitchenham, "Status Report on Software Measurement," *IEEE Software,* vol. 14, no. 2, 1997.

[24] J. Rumbaugh, M. Bhala, W. Lorensen, F. Eddy, and W. Premerlani, *Object-Oriented Modeling and Design.* Englewood Cliffs, N.J.: Prentice Hall, 1991.

[25] A. Snyder, "Encapsulation and Inheritance in Object-Oriented Languages," *Proc. Object-Oriented Programming Systems, Languages, and Applications Conf.,* pp. 38-45, Nov. 1986.

[26] *Software Product Evaluation—Quality Characteristics and Guidelines for Their Use,* ISO/IEC Standard ISO-9126, 1991.

[27] J. Vincent, A. Walters, and J. Sinclair, *Software Quality Assurance,* vol. 1. Prentice Hall, 1988.

Jagdish Bansiya received the BTech degree in computer science from the University of Roorkee, India, in 1990, and the MS and PhD degrees in computer science from the University of Alabama in Huntsville, in 1992 and 1997, respectively. He is currently an assistant professor in the Department of Mathematics and Computer Science at the California State University, Hayward, where he also directs the research and development of an industry-funded Internet technology lab. His previous roles have included several industry positions and consulting activities. His research interests include software architectures, object-oriented design measurement, object-oriented enterprise frameworks, design patterns, and software reverse engineering. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Carl G. Davis received the PhD degree in engineering mechanics from the University of Alabama, Huntsville. He is a professor emeritus in the Computer Science Department at the University of Alabama in Huntsville, where he was chairman of the department from 1986 to 2000. Prior to joining the university, he worked for the federal government, where he was a senior executive in the strategic defense program, responsible for R&D programs in real-time hardware, and software development techniques. He was involved in the development of several early prototype software development support environments for large, complex, real time systems in the early 1970's. His research interests are in the areas of software engineering, with most recent emphases in the areas of measurement of object-oriented designs. He also directs a federally funded precollege teacher-training program teaching how to use computational science as a new teaching paradigm. He has published numerous papers in the areas of software requirements development, software reuse, and tools for real-time software development. He has chaired six international conferences on software engineering and helped to organize several others. He is a fellow of the IEEE.