Object-Oriented Architecture Measures

Brian Keith Miller
Raytheon Systems Company
Millerbk@gvl.esys.com

Dr. Pei Hsia
The University of Texas at Arlington
Hsia@cse.uta.edu

Dr. Chenho Kung
The University of Texas at Arlington
Kung@cse.uta.edu

Abstract

Early elimination of poor design architecture allows software to be constructed that is more extendible, flexible, maintainable, and thus less expensive. Currently, there exist few object-oriented design architecture measures to aid in this task. Many contemporary object-oriented software measures fail to address the intrinsic components of the object-oriented paradigm: hierarchy, inheritance, identity, polymorphism, and encapsulation. By focusing upon the intrinsic components of the object-oriented paradigm, it is hoped a better set of measurements can be created which help to distinguish between those architectures that follow object-oriented principles and those that do not. This research proposes four new software measures that attempt to measure the strength of these intrinsic concepts in an object-oriented design. These measurements are derived from available principles of object-oriented design which are relatively new to research.

1. Introduction

As the object-oriented paradigm gains wider acceptance in the software community, powerful software measures will be required as they are in the functional paradigm. The plethora of existing functional-based measures is not suitable for application with object-oriented designs. New measures that are especially tailored for the object-oriented community are required. The new measures should be powerful enough to find poor design architecture constructs yet be easy to implement so that businesses and the commercial community will not find them to be a burden to maintain. They should have guidelines that help software engineers interpret the results for a particular context. Finally, these measurements should be validated for application in the object-oriented domain.

This research presents a suite of four measurements. Each measurement has a theoretical base rooted in one or more principles of object-oriented design and has been specifically designed to measure an intrinsic component of the object-oriented paradigm. It is hoped that the application of these measurements individually and in tandem will allow engineers to more easily identify good and bad design architectures. The early identification of bad design architectures is an essential component to the software development process. By locating and correcting poor design architectures in the early phases of software development, expensive rework can be avoided while improving the overall maintainability of the software.

The key contributions of this paper are the development and validation of a set of object-oriented design architecture measurements that are theoretically grounded in the principles of object-oriented design. The remainder of this paper is organized as follows. Section 2 describes the past research in object-oriented metrics. Section 3 describes the underlying theory of the approach taken to develop the measurements presented in this paper. Section 4 presents the measurements, their analytical evaluation, and their application in an attempt to locate poor design architectures. Section 5 presents some notes on the application of the measures and experimental validation. Concluding remarks and future directions are presented in Section 6.

2. Past Work in Object-Oriented Measurements

Several object-oriented measurements have been presented in past research. Perhaps the most notable achievement has been the suite of class-level measurements presented in [Chidamber94a]. This paper presents six measurements based upon the ontological principles of objects. A software size estimate metric for object-oriented systems is presented in [Larajeira90a]. An equation that is based upon the decomposition and the levels of abstraction for each object under measurement is provided. A similar metric for estimating programming effort is presented in [Jenson91a]. A metric that predicts maintainability of object-oriented systems is presented in [Li93a]. The paper presents a measurement that is based

0-7695-0001-3/99 $10.00 (c) 1999 IEEE

upon several other existing measurements including those from [Chidamber94a]. A framework for classifying object-oriented metrics is presented in [Brito94a]. This work examines and classifies several existing object-oriented and functional metrics that are applicable in the object-oriented domain. Additionally, a few new counting-type object-oriented metrics are introduced. A suite of product and process metrics for the object-oriented domain is presented in [Lorenz94a]. The metrics introduced cover areas such as management, application size, staffing and scheduling. There is a comprehensive set of design metrics that cover both class and hierarchy scope. The metrics are presented in a standard format which provides guidelines for usage. A suite of testability metrics covering encapsulation, inheritance, and polymorphism is presented in [Binder94a]. Many of the metrics are borrowed from other research such as [Chidamber94a]. A suite of cognitive complexity metrics for object-oriented programs is presented in [Cant94a]. This work relies on the ways that a software engineer uses chunking and tracing to understand program code.

Many of the above software measurements are attempting to describe various attributes of software such as reuse, maintainability, complexity, size, productivity, etc. These attributes are often overlapping in scope, ambiguously defined, or exclusive of each other in purpose. Furthermore, there is general disagreement in attempts to classify these measurements. Following the lead of [Chidamber94a] that presented measurements based on ontological principles, it is best to create design measurements that capture the essentials of the design domain. We assert that it is much less complex to look for elements of bad design architectures and remove them than to look for the many ambiguous and overlapping components of good design architectures (reuse, maintainability, etc.). Furthermore, the benefits of good design architectures extend far beyond the few measurable qualities already defined over it.

If the principles of the object-oriented paradigm are indeed correct, then designs which adhere to them should have the many qualities that the object-oriented paradigm promises: maintainability, reuse, less complexity, etc. In other words, the overall quality should be inherent in the final product because the principles were upheld. The landmark research presented in [Chidamber94a] proposed measurements based upon the fundamental theories of objects but not specifically the object-oriented paradigm. It is hoped that the measurements presented in this research may be useful because they are based directly upon principles of the object-oriented domain.

3. Measurement Theory Base

This section briefly presents an overview of the object-oriented design principles upon which the new measurements are founded. Also included are the necessary object-oriented constructs used during the definitions of the measurements.

3.1 Principles of Object-Oriented Design

There are many heuristics and rules-of-thumb to help achieve a good object-oriented design. Undoubtedly, many excellent designs and final software products have been produced using nothing more than simple rules and the designer's intuition on what constitutes a good and proper design architecture. However, if software measurement is to flourish as a true engineering endeavor, then the underlying axioms, principles and theories must be understood and explored. This section presents a list of the principles that are applied to object-oriented design efforts. Table 1 summarizes a list of principles used for this research. The reader is encouraged to seek out the original work in each case for more detailed information.

3.2 Representation of Operations

A class operation is represented as a set containing a set of argument types, a Boolean value which indicates the presence or absence of an operation body, a return value type, if any, and the type classification. Every class operation may be classified according to one of the entries in Table 2 that defines a set of classifications for describing meaningful attributes for class operations. The classifications have been taken from [Harrold92a] and extended to cover more general cases where operation bodies may not be defined.

It is useful to be able to distinguish subsets of operations for a given class by their scope type. Using the notations from Table 2, an alphabet Σ may be defined over a language μ using the '?' symbol from UNIX-style regular expressions. The '?' symbol indicates that the preceding symbol is optional. The alphabet creates a concise language of patterns for denoting particular subsets of operations for a given class C: $\text{Operation}_f(C)$. The representation of f is composed of a permutation from the regular language of Figure 1. Table 3 presents some examples using the operation classification language to describe a set of operations $\text{Operation}_f(C)$ for a class C. Note that the usage of the language may include the '*' symbol which represents the "don't care" condition.

3.3 Representation of Class Distance in a Hierarchy

The distance between two classes in a class hierarchy is defined as the absolute value of the difference in the hierarchy nesting level of the two classes with respect to the same root class. If two classes do not participate in the same hierarchy, then their distance is defined to be infinity. In cases of multiple inheritance, the two classes must be in the same sub-hierarchy, that is one must be related to the other as its ancestor. The distance between two classes C and D is denoted by Distance(C, D). For

example, using the classes Figure 4. then Distance(Geometry, Ellipse) = Distance(Ellipse, Geometry) = 2.

4. Results

The measurements presented in this section will be validated by direct comparison with similar existing metrics, if any. The measurements used in the comparison must evaluate similar architecture components of design. The design problem used for these comparisons is in Figure 4.

4.1 Class Abstraction and Hierarchy Brittleness

The class is the fundamental unit of abstraction in the object-oriented paradigm. The formation of classes and the relationships between them is a critical endeavor when constructing object-oriented software. During the construction of class hierarchies, parent classes must be extended in order to form more specialized child classes. The extension of a class involves the addition of new behaviors and the optional modification of existing behaviors. While any class may be arbitrarily extended to build a new subclass, it may not be an action that is in the best interest of the hierarchy or system architecture. Object-oriented principles and heuristics exist that provide conditions for appropriate class extension in the context of a class hierarchy.

The abstractness of a class is the degree to which the class can be extended via inheritance. Abstract classes are considered to be extendible while concrete classes are not. A class is composed of operations and data. Class member data exists only to provide service to operations and thus does not affect the abstractness of a class. Class operations directly contribute to the abstractness of a class and must be further examined.

Non-virtual-defined operations are those operations classified as new, recursive, and redefined ($Operation_{N}(C)$, $Operation_{Rv}(C)$, and $Operation_{Rd}(C)$). These operations include an interface and associated behavior and cannot be overridden by subclasses but instead must be inherited without change from the parent class. Since these operations cannot be modified in any way by subclasses, they are not considered to be extendible and thus do not contribute to the extendibility of a class.

Virtual-defined operations are those operations classified as virtual-new, virtual-recursive, and virtual-redefined ($Operation_{VN}(C)$, $Operation_{VRv}(C)$, and $Operation_{VRd}(C)$). These operations include an interface and associated behavior. Unlike their non-virtual counterparts, these operations may be overridden in behavior by subclasses. Because these operations may be extended by overriding, they are considered to be extendible to some degree and thus contribute to class extendibility.

Virtual-undefined operations are those operations classified as virtual-new-undefined, virtual-recursive-undefined, and virtual-redefined-undefined ($Operation_{VNU}(C)$, $Operation_{VRvU}(C)$, and $Operation_{VRdU}(C)$). These operations include an interface but do not include an associated behavior. They are very similar to their virtual-defined counterparts except that it is required that some child class of the defining parent class must provide the operation behavior. Since these operations must be extended by overriding, they are considered to be the most extendible of the operation types and thus contribute to class extendibility.

Through conceptual argument it has been established that operations without bodies are more extendible (and thus more abstract) than operations with bodies. It has also been established that non-overridable are the least extendible of all. These facts are presented in Table 4, along with a simple numeric mapping that describes their relative abstractness to each other. Using the concepts provided in Table 4, Equation 1 is created which provides a measure of abstractness or extendibility of a class.

In Table 4, all possible operation types are mapped with abstraction constants that indicate the amount of abstractness that a given operation contributes to its constituent class. It is desired to measure the overall abstractness of an individual class based upon the abstractness of its member operations. Class attributes do not contribute to the abstractness of a class. In order to achieve the desired mapping between 0.0 (most abstract) and 1.0 (least abstract), the abstractness constants from Table 4 are used as weightings Equation 1. The numerator uses only operations that contribute to the concreteness of a class (non-zero abstractness constant). The magnitude of the numerator will grow proportionally to the number and type of operations that contribute concreteness. The denominator is used as a normalization factor that constrains the resultant values between 0.0 and 1.0.

Just as classes can be described in terms of abstractness, so can class hierarchies. The abstractness of a class hierarchy is directly contributed from its member classes. The term brittleness is applied to class hierarchies to describe the degree of inflexibility inherent due to the abstractness of the constituent classes. Based upon object-oriented principles, deep hierarchies must retain some degree of abstractness to be useful. The deeper a hierarchy is, the more abstract the classes at or near the root should be.

On an abstraction vs. hierarchy graph such as the one presented Figure 2, the area under the curve for a given hierarchy chain represents an accumulated amount of concreteness for the chain. The amount of concreteness within a hierarchy is termed “brittleness.” The concept of brittleness can be captured as an equation. Brittleness is defined as the area under the curve for a hierarchy chain over the depth of the chain. The region under the curve is

easily approximated by employing the trapezoidal rule from calculus integral theory.

The normalized result for Brittleness(H) approaches 1.0 for brittle hierarchies and 0.0 for non-brittle (flexible) hierarchies. The deeper the hierarchy is, the more flexible it should be to allow for proper subclassing opportunities. Shallow and concrete hierarchies are much more maintainable and extendible than their deeper concrete counterparts because the complexity due to the depth of classes is avoided. Therefore, the brittleness of a hierarchy should be tuned according to its depth.

At the midpoint on the abstractness scale (Y-axis), a horizontal line is drawn called the "virtual-undefined" line. This line marks that maximum abstractness value that can be obtained if no virtual-undefined operations are present in a class hierarchy. Due to the nature of the Abstractness(C) equation, the result will always be 0.5 or greater if no virtual-undefined operations exist for a given class. In well-formed hierarchies, root and other classes that dictate hierarchy protocol should have abstractness values below the virtual-undefined line. Classes that fall under this line are populated with a majority of virtual-undefined operations as opposed to virtual-defined and other non-virtual operations.

Brittleness(H) is a measurement of abstraction for a class hierarchy. As such, it may only be compared with measurements that have hierarchy scope. The candidate measurements are from [Lorenz94a]: Class Hierarchy Nesting Level (NL) and Number of Abstract Classes (NAC). The results using the example Figure 4 are presented in Table 5.

Although both of the Lorenz measurements do attempt to measure quality attributes of a class hierarchy, neither adequately captures the concept of hierarchy brittleness. The indicated NL values do indicate that further subclassing will be possible and remain within the recommended Lorenz guideline of six. However, even though each example class receives the same NL measurement result, they are not all equally extendible as indicated by the brittleness results. The NAC measurement offers no meaningful information concerning the extendibility of a leaf class or the overall abstractness of a hierarchy. Lorenz purports that 10-15% of classes in a project should be abstract. If each hierarchy of Figure 4 is taken in isolation the NAC percentage is 33. If the entire example is considered, the NAC percentage falls to approximately 17%. Therefore, the NAC indicates that too few abstract classes have been used, but at an application level. It says nothing about the distribution of abstract classes throughout any given hierarchy chain.

4.2 Class Abstraction Cohesion

The proper abstraction of domain concepts into classes is an essential activity that can have a grave impact on design architectures if performed improperly. A class should represent a single, clearly defined domain concept with crisp boundaries. Improper class abstractions will generally contain more than a single, clearly defined concept within the class definition. Generally, the logic and intuition of the designer is the only guard against improper class abstractions. However, classes have measurable qualities that can be used to identify the strength of their abstractions. The strength of abstraction in a class definition can be defined as a form of cohesiveness of identity for the class. If the class is based upon a single, clearly defined concept, then the class representation is highly cohesive which means that the bound operations and data are closely related to each other in order to represent the abstraction. We define this as class identity cohesiveness. For a class that is based on an improper abstraction, the operations and data will not be cohesive. The measurement of this cohesive quality is called Class Abstraction Cohesion (CAC) defined Equation 2. This measurement borrows heavily from concepts pioneered by [Bieman94a].

In order to measure the strength of abstraction in a class, the various bindings between the class operations and class attributes must be examined. Generally, class attributes are found only within the class definition. Class attributes and operations in a class have two possible sources: they have been locally defined as new in the class itself or they have been inherited from one or more ancestor classes. It is rare that parent classes make local class attributes available to child classes. Such instances are considered to be very poor design and are always a cause of concern. However, class operations are almost always inherited, even in simple classes. Therefore, for the purposes of this measurement, only locally available (i.e., not inherited) attributes and operations that may access those attributes will be considered.

All operations that have access to the private sections of the enclosing class have access to the local class attributes. These operations are the ones that will be used during the measurement of class abstraction identity. First, all operations that have no body of code can be summarily eliminated. The second set of operations which can be eliminated are those which are inherited without modification from ancestor classes. These operations are inherited with their existing body of code intact. Since these operations are defined in ancestor classes, they have no direct access to the local class attributes. The remaining operation categories have the ability to access local class data. These operations are new operations defined within the class itself and redefined operations whose behavior has been redefined in the local class. The results are summarized in Table 6.

Bieman and Ott [Bieman94a] define the concept of adhesiveness of functions to their internal data members in an attempt to measure functional cohesion. This concept may be translated to the object-oriented paradigm and used to measure the cohesiveness of classes by examining

the cohesion between member operations and member data. Thus, in translating Bieman and Ott's [Bieman94a] measurement from the functional paradigm to the object-oriented paradigm, Bieman's slice abstractions are represented as class operations and Bieman's data tokens are represented as class attributes. The results are summarized in Equation 2.

Class abstraction cohesion for a class C measures the internal cohesion of the class. The only candidate measurement is Lack of Cohesion in Methods (LCOM) from [Chidamber94a]. The results using the example from Figure 4 are presented in Table 7.

Both measurements correctly identified the CompositeGeometry class as perfectly cohesive. That is, all member functions use all available data members. Both measurements also provide indications that the LinearGeometry and Ellipse classes are not completely cohesive. The CAC measurement seems to provide more intuitive results concerning the difference in cohesiveness between LinearGeometry and Ellipse. However, neither measurement satisfies the monotonicity property of Weyuker's evaluation criteria and therefore such intuitiveness is not justified. Based upon these results, neither measurement seems clearly superior.

4.3 Pure Inheritance Index

The inheritance relationship between two classes is one of the most powerful concepts in the object-oriented domain. Through inheritance, class behavior is extended to fit new requirements and new environments. Inheritance also provides the mechanics that makes code and protocol reuse possible on a broad scale and sets the stage for generic programming. It is a widely accepted fact in the object-oriented community that inheritance should take place strictly for true specialization of a concept. Inheritance used only to acquire certain behaviors or for code reuse is considered improper and detrimental to good design architecture. The Liskov Substitution Principle (LSP) provides theoretical support for these widely held assertions. The LSP states informally that subclasses must be able to act as their ancestor classes without the clients of the instantiated objects knowing any different. In Table 8, a comprehensive set of outcomes for inheritance in a subclass [Lorenz94a] are identified. For each outcome, an alphabetic code has been assigned and the determination has been made whether or not the outcome adheres to the LSP. All but two outcomes are known to support the LSP.

It is useful to create a mapping from the inheritance outcomes Table 8 to the known operation classifications. Such a mapping would make it possible to determine the manner of inheritance present in a subclass. Table 9 presents such a mapping. In Table 9, the adherence of each operation to the LSP is specified.

Specializing by pure inheritance is considered to be ideal. The pure inheritance index (PII) measure should be defined to provide and detect deviations from pure inheritance. In practice, pure inheritance is characterized by subclasses adding totally new behaviors with few or no method overrides and deleted methods. In cases of pure inheritance where methods must be overridden, the new methods are expected to uphold the "child is a parent" tenant relationship between the subclass and its parent class by adhering to the LSP. A measurement for pure inheritance should involve the following three components:

- Operations that are overridden/deleted O(C),
- Operations that do not preserve inherited behaviors P(C),
- Operations that are newly added for a class at a given hierarchy level N(C).

Assuming that each of the above properties will contribute equally to the overall measure of pure inheritance, Equation 3 (pure inheritance index) is easily derived as a normalized (0.0 to 1.0) measure. In this equation, P represents the component due to preserved behaviors, O represents the component due to overridden/deleted behaviors, and N represents the component due to newly added behaviors. Each component P(C), O(C), and N(C) has a range of 0.0 (ideal) to 1.0 (worst). Therefore, the aggregated function is easily normalized between 0.0 and 1.0.

Proper inheritance is characterized by a high amount of preserved behaviors between a subclass and its parent class. It has already been established which operation types preserve behaviors and support the LSP. If a measurement goal is to focus upon possible deviations to the LSP, then only operations that possibly support or do not support the LSP must be analyzed. Virtual-redefined and (non-virtual) redefined operations may or may not adhere to the LSP. For a given class, Equation 4 measures its ability to preserve inherited, redefined behaviors. The equation P(C) contains the number of overridden operations that do not preserve their inherited behaviors in the numerator. The denominator holds the number of all redefined operations for the class, even if the behavior has been preserved. Thus, as the number of preserved operations increases relative to the number that could have been preserved, the result will approach 0.0. The value P(C) results in an ideal value of 0.0 if there are no overridden operations and a worst-case value of 1.0 if none of the operations preserve their parent's behaviors.

A low number of operation overrides is an indicator of pure inheritance. The inverse situation is that a high number of overrides may indicate impure inheritance since subclasses are probably not taking advantage of inherited behaviors. This is true even if the overridden operations preserve the parent's behaviors. For a given class Equation 5 measures the component of pure inheritance

that involves the number of overridden operations. The equation O(C) contains the number of overridden operations in the numerator. The denominator holds the number of all inherited operations for the class. Thus, as the number of overridden operations decreases relative to the number of operations that could have been overridden, the result will approach 0.0. The value O(C) results in an ideal value of 0.0 if there are no overridden operations and a worst-case value of 1.0 if all of the inherited operations have been overridden. Note that the difference between O(C) and P(C) is subtle, but important.

Proper inheritance is characterized by a decreasing number of new operations as the class hierarchy level increases. Classes that are low in the inheritance hierarchy can and should take advantage of the many inherited operations made available from ancestor classes. Therefore a function is desired which captures these attributes concerning class hierarchy level and the number of newly added operations. We require a function that returns 0.0 in the ideal case when the number of newly added operations is decreasing as the hierarchy level increases. Likewise, the function should return a 1.0 for the worst case when the number of newly added operations is increasing as the hierarchy level decreases. For a given class C Equation 6 measures the component of pure inheritance that involves the number of newly added operations at a given hierarchy level. A constant 1 has been added to the denominators of the function to avoid division by zero.

Very few measurements address the validity of class inheritance at the class scope. The only reasonable candidate found in research is the Specialization Index (SI) measurement from [Lorenz94a]. The results from using the example in Figure 4 are presented in Table 10.

Both measurements correctly evaluate the base class Geometry for perfect inheritance. However, this example does not show a relative weakness of SI: while SI will never evaluate a base class to a non-zero value, the PII measurement will return a non-zero value for a base class which has valid member functions. Such a result would indicate the less than ideal use of a base class as actually defining behavior instead of just providing protocol. Lorenz uses a SI threshold value of 15% to locate anomalies. Based upon this fact, the SI measurement flags both SurfaceGeometry and Polygon as improperly subclassed classes. This is due primarily to its sensitivity to the raw number of overridden methods. The class Polygon is in fact properly inherited and takes great advantage of its ancestor classes and only requires two new additional operations, one of which is an override. While the SI measurement flagged Polygon as a potential problem, the PII measurement provides an indication that the inheritance is indeed acceptable due to the near-zero result.

4.4 Relationship Abstraction Index

Class relationships form the basis for object communication and interaction by providing visibility between related classes. Relationships are derived early in the design phase usually out of convenience or strict modeling of the problem domain in the solution domain. However, there are no automatic mechanisms to ensure that the relationships are well formed. Naive designs will often make use of class interfaces lower in the hierarchy than necessary, ignoring more abstract class interfaces that provide the same ability with more generalism. To achieve the maximum generic use of classes, client classes should use the most abstract class interface possible. Doing so insulates the client against the addition of new classes to the hierarchy and changes to existing classes. If the client has made use of a high-level generic class interface in a hierarchy, new subclasses that support the interface may be defined and used without changes to the client in most cases. The use of high-level interfaces by client classes is addressed by the Dependency Inversion Principle (DIP). Relationship abstraction can be measured relative to several different scopes: class, category, internal category and external category.

For a single class, the RAI measurement is calculated using all classes related through aggregation or association such that the measured class has visibility. Since dependency inversion depends upon the abstractness of the related classes, the abstractness measurement presented in Equation 1 will be utilized as a component of this measurement. For a single class C, the RAI(C) is calculated in Equation 7 by summing the abstractness values of all appropriately related classes and dividing by the total to obtain an average.

Measures for category, internal category, and external category scope are derived in a similar fashion. If a class category Ca is treated as an single entity, RAI(Ca) is easily calculated. RAI(Ca) would be given an indication of how abstractly the entire category makes use of itself and other any external classes. Additionally, RAI(Ca) may be further refined by including only internal and external classes giving $RAI_{INT}(ca)$ and $RAI_{EXT}(ca)$, respectively. These measurements indicate how abstractly the class category makes use of internal and external classes respectively.

No measurements were located in research that attempted to evaluate the abstract usage of one class by another. A measurement called “Bounce-C” found in [Binder94a] was found to be similar in purpose, but no details of its definition or usage were provided. Therefore no comparison for validation purposes is available at this time. Some examples using Figure 4 are provided in Figure 3.

5. Experimental Validation

Each of the four proposed measurements was validated against a large code set from the aerospace industry. The code set was written in the C++ language and was subject to code reviews. The total code set scanned was in excess of 250,000 non-commented source lines of code (NCSLOC). However, only eight subsets for approximately 43,000 NCSLOC (291 classes) were chosen for a detailed analysis. The details on the eight code bases are in Table 14.

5.1 Hierarchy Brittleness (HB)

Each code set was subjected to the HB measurement. The results are summarized in Table 15. Anomalies were identified by any measurement reading greater than 0.700. The measure was able to successfully locate the following problems:

Improper Inheritance

Two instances were located in code set 7 where a class had been subclassed improperly. The original design had intended for these subclasses to each be instances of the same class. Instead of instantiation, inheritance was used that created a very short yet brittle hierarchy indicating that the parent class contained most of the desired functionality needed in the subclasses. Instances of inheritance for the sake of code reuse were located in code set 4. The classes of code set 4 do pick up desired behaviors but will always fail the "is-a" test of pure inheritance.

Missing levels of abstraction

Code set 3 was the site of missing levels of class abstraction. The HB measurement shows undeniable problems with the hierarchy that dominates this code set. The base classes (including) the root are much too concrete (greater than 0.600) to serve as effective protocol providers. Maintenance change logs indicate that this hierarchy has been expanded several times. The HB measurement indicates that more expansion of levels is needed near the root of the hierarchy.

Inflexible root class

The hierarchy of code set 3 also is the site of a very inflexible (concrete) root class. The root class of this hierarchy is much too concrete (0.696 abstractness). This concreteness makes it and any of its ancestors difficult to subclass as well. Change records indicate that this is due to the proliferation of member functions in and near the root class. The class long ago lost its ability to simply define protocol for subclasses. Instead, the desire for a more homomorphic class hierarchy has constantly provided for an upward migration for member functions (and associated data).

5.2 Class Abstraction Cohesion (CAC)

Each code set was subjected to the class abstraction cohesion measurement. Anomalies were identified by a value less than 0.200. The results are summarized in Table 16. The measure was able to successfully locate the following problems:

Combined Interface and state

Many classes were discovered which do not separate interface from state as the model-view-controller design pattern dictates. Such classes were located in code sets 2, 6, 8, and 7. Similar problems with other classes were located in code sets 3 and 7.

Using aggregation instead of inheritance

Code set 3 was found to contain an instance of using aggregation where inheritance was needed. Instead of using an inheritance hierarchy to distribute behaviors to a set of subclasses, behaviors were divided into several single-level classes. These classes (merely packets of similar functionality) were then promptly aggregated at certain points in the hierarchy. The encapsulation class was also required to provide "pass-thru" interfaces for all the embedded functionalities.

Other Concerns

The CAC measure also showed some sensitivity to valid situations by returning false positive results. Some classes were identified as anomalous that had not outstanding design flaw. For example, using both static and non-static member functions in a class (a limitation of the language) often resulted in false CAC detection, such as in code set 3. It also showed sensitivity to passive classes that have mostly read-only interfaces. Code set 5 is a geometric class hierarchy where many of the classes have a passive interface. These interfaces usually were falsely detected. Code set 7 includes a class that forms an interface (and barrier) to a large library of cartographic functions. Because a single class encapsulates the many disparate functionalities, it also was shown not to be cohesive. On this last point, it is unclear if this represents a design problem or not. Instead of a single class encapsulating the library, it may be possible to use a hierarchy of classes. The CAC needs to be reevaluated for the possibility of making it less sensitive to the identified cases above.

5.3 Pure Inheritance Index (PII)

Each code set was subjected to the pure inheritance index measurement. Anomalies were identified by any values greater than 0.700. The results are summarized in Table 17. The measure was able to successfully locate the following problems:

Coarse Hierarchies

Code sets 3, 6 and 7 include small class hierarchies that set the stage for future reuse and expansion. This results in a situation where there is a hierarchy of two classes and the child overrides and defines the members (large in number in these cases) of the parent class. The PII measure flags this "compressed" hierarchy state as a problem. A better design would probably spread the members among a set of intermediate ancestor classes allowing for more finely grained packets of functionality and separation of concerns which in turn leads to more opportunities for reuse.

Inverse Abstraction Hierarchies

An "inverse" abstraction problem was located in code set 7. This situation occurs when a child class is more abstract than an ancestor class. This indicates an inheritance problem between the parent and child class. If a child is more abstract than a parent class, the child is essentially creating his own protocol over that of the parent. The new protocol will probably fail the "is-a" test of pure inheritance.

Other Concerns

The PII measurement showed extreme sensitivity to the number of overridden operations in a class. While the theoretical basis does appear to be sound, the level of sensitivity to this one type of operation is probably not desired. This pervasiveness of the number of overridden methods in this measure needs to be reevaluated.

5.4 Class Relationship Abstraction Index (RAI)

Each code set was subjected to the class relationship abstraction measurement. Anomalies were identified by values greater than .700. The results are summarized in Table 18. The measure was able to successfully locate the following problems:

Unnecessary use of subclasses in a hierarchy

Many instances concerning the use of child classes when one or more ancestors would have sufficed were located in most code sets. Many times the designers had provided generic base class interfaces only to have them ignored during implementation. However, in many other cases, the direct use of a class was necessary because no base classes were provided.

Cyclic class relationships

This measurement was also helped to locate classes with improper or questionable relationships to other classes. Code set 8 was found to have two classes which was symbiotic. This type of codependence is often problematic during maintenance.

Forward associations in a hierarchy

Another questionable relationship was located in code set 5 where a root class has a direct (forward) association to a leaf class. A better abstraction of the problem domain could have avoided the need to have the forward reference. This was an easy situation to spot since root classes should have very abstract relationships to other classes.

Other Concerns

The RAI measurement shows a sensitivity to aggregated components. Currently, the measurement includes both aggregation and association relationships when calculating the abstraction of a set of relationships. This situation unfairly penalizes classes with aggregated components since a component cannot be strongly aggregated in an abstract sense. The use of aggregated relationships need to be reevaluated for this measurement.

5.5 Comparison to other measurements

Four other measurements were also used against the code sets of Table 14. The results are summarized in Table 19. These measurements were chosen because they provide the closest attempts at measuring similar object-oriented attributes as the four measurements provided in this research. The four measurements are briefly described along with the results obtain against the experimental code sets. In general these measurements ignore the richness of the object-oriented domain, but instead focus on very high-level indicators.

Nesting Level (NL)

The nesting level is the depth of the deepest hierarchy of a code set. Generally, any value over 6 is considered to be a sign of a hierarchy that has grown too deep. Only code set 3 provided a hierarchy that was flagged by this measure. While the hierarchy presented in code set 3 is deep and complex, it maps to a deep and complex set of domain objects and still requires more levels for better mapping. The HB measure takes into account other factors (such as abstractness) that may be used to offset the complexity involved with a deep hierarchy.

Number of Abstract Classes (NAC)

The number of abstract classes is used as an indication of the successful application of inheritance. While the existence of these classes is essential to good class hierarchies, this measurement also fails to capture the essentials of how the inheritance occurs and says nothing about the inheritance between any two particular classes.

Specialization Index (SI)

The specialization index is an attempt to examine how two classes share an inheritance relationship. This measure involves the number of overridden methods and the depth of the class hierarchy similar to the hierarchy brittleness and pure inheritance index measures. There appears to be no correlation between SI and the measurements presented in this research. The SI measure is very sensitive to the nesting level of a class which results in heavy penalties for classes deep in a hierarchy. The PII measurement gauges the inheritance relationship between two classes in isolation. The depth of a class in a hierarchy is offset by other attributes of inheritance.

Lack of Cohesion in Methods (LCOM)

The LCOM measurement shows a strong correlation with the CAC measurement. Both attempt to measure cohesiveness by examining the usage of attributes by member functions. At this time, there is not enough evidence to support the assertion that one is any more accurate than the other.

6. Concluding Remarks

This research has developed and implemented a new suite of software measurements for object-oriented design architectures. Each measurement attempts to gauge the strength of an intrinsic attribute of the object-oriented paradigm. It is hoped that by measurement of intrinsic attributes, more powerful and accurate measures will have been produced. Each measurement has been validated, if possible, by direct comparison with other object-oriented metrics that evaluate similar software attributes. Two of the measurements evaluated as superior, one was indeterminate, and one had no located peer in research to be compared against. The results of experimental validation on code sets from industry have been presented. The measurements were successful in locating poor design constructs that were not always visible to other measurements.

This paper represents research in progress. Therefore, definitive conclusions are yet to be reached. It is hoped that the proposed new measurements of object-oriented design architecture will prove to be useful in identifying design constructs that do not conform to the commonly accepted principles of the object-oriented domain. Furthermore, it is hoped that the measurements presented will prove valuable to identifying overall quality in object-oriented software products.

References

[Basili92a] Basili, V. *Software Modeling and Measurement: The Goal/Question/Metric Paradigm*. University of Maryland, Dept. of Computer Science, Technical Report CS-TR-2956, 1992.

[Bieman94a] Bieman, J.M. and L. Ott. *Measuring Functional Cohesion.* IEEE Transactions on Software Engineering, Vol. 20, No. 8, August 1994, pp. 644-57.

[Binder94a] Binder, R.V. *Design for Testability in Object-Oriented Systems.* Communications of the ACM, Vol. 37, No. 9, September 1994, pp. 87-101.

[Booch91a] Booch, G. Object-Oriented Design with Applications. Benjamin/Cummings, 1991.

[Briand94b] Briand, L., et al. *Goal-Driven Definition of Product Metrics Based on Properties.* University of Maryland, Dept. of Computer Science, Technical Report CS-TR (unknown), UMIACS-TR-94-75.

[Briand94d] Briand, L., et al. *Characterizing and Assessing a Large-Scale Software Maintenance Organization.* University of Maryland, Dept. of Computer Science, Experience Report.

[Cant94a] Cant, S.N., et al. *Application of Cognitive Complexity Metrics to Object-Oriented Programs.* Journal of Object-Oriented Programming, July/August 1994, pp. 52-63.

[Chidamber94a] Chidamber, S.R. and Kemerer, C.F. *A Metrics Suite for Object-Oriented Design.* IEEE Transactions on Software Engineering, Vol. 20, No. 6 June 1994, pp. 476-93.

[Harrold92a] Harrold, et al. *Incremental Testing of Object-Oriented Class Structures.* ACM, 1992.

[Jenson91a] Jensen, R. and J. Barteley. *Parametric Estimation of Programming Effort: An Object-oriented Model.* Journal of Systems and Software, Vol. 15, 1991, pp. 107-14.

[Kennedy92a] Kennedy, B.M. *Design for object-oriented reuse in the OATH library.* Journal of Object-Oriented Programming, July/August 1992, pp. 51-57.

[Laranjeira90a] Laranjeira, L. *Software Size Estimation of Object-Oriented Systems.* IEEE Transactions on Software Engineering, May 1990, pp. 510-22.

[Li93a] Li, W. and S. Henry. *Object-Oriented Metrics that Predict Maintainability.* Journal of Systems and Software, Vol. 23, No. 2, November 1993, pp. 111-122.

[Liskov86a] Liskov, B. and Guttag J. Abstraction and Specification in Program Development. MIT Press, 1986.

[Lorenz94a] Lorenz, M. and J. Kidd. Object-Oriented Software Metrics. Prentice-Hall, Object-Oriented Series, 1994.

[Martin96c] Martin, R. The Dependency Inversion Principle. C++ Report, Vol. 8, No. 6, June 1996, pp. 61-5.

[Martin96d] Martin, R. The Dependency Inversion Principle. C++ Report, Special Issue, August 1996,

pp. 30-7.

[Meyer88a] Meyer, B. Object Oriented Software Construction, Prentice-Hall, 1988, p 23.

[Weyuker88a] Weyuker, E.J. *Evaluating Software Complexity Measures.* IEEE Transactions on Software Engineering, Vol. 14, No. 9, September 1988, pp. 1357-65.

[Wirfs90a] Wirfs-Brock, R.J. and R.E. Johnson. *Surveying Current Research in Object-Oriented Design.* Communications of the ACM, September 1990, Vol. 33, No. 9, pp. 104-24.

Table 1. Object-Oriented Design Principles Summary

Principle	Source	Summary
Open-Closed	Meyer88a	Classes should be open for extension and closed for modification
Liskov Substitution	Liskov86a	Child is a parent
Dependency Inversion	Martin96c	Details should depend upon abstractions
Interface Segregation	Martin96d	Classes should only depend upon interfaces they use
Reuse/Release Equivalency	Martin96d	The granule of reuse is the same as the granule of release
Common Closure	Martin96d	Classes within a class category should share common closure
Common Reuse	Martin96d	Classes within a class category should be reused together
Stable Abstractions	Martin96d	The more stable a class category is, the more it must consist of abstract classes
Least Astonishment	Booch91a	Solution domain should consist of minimal and necessary classes from the problem domain
Deep Abstract Hierarchies	Kennedy92a	Class hierarchies should be deep and abstract
Demeter	Wirfs90a	Collaboration between classes should respect class boundaries

Table 2. Class Operation Type Summary

Operation Type (T)	Symbol	Summary Explanation
New	N	Newly defined in the class with no matching operation in ancestor classes
Recursive	Rv	Defined in an ancestor class and inherited with no new definition
Redefined	Rd	Defined in an ancestor class and redefined with a new definition
Virtual-New	VN	New + Allows overriding by child classes
Virtual-Recursive	VRv	Recursive + Allows overriding by child classes
Virtual-Redefined	VRd	Redefined + Allows overriding by child classes
Virtual-New-Undefined	VNU	New + Virtual + Operation has no body implementation (interface only)
Virtual-Recursive-Undefined	VRvU	Recursive + Virtual + Operation has no body implementation (interface only)
Virtual-Redefined-Undefined	VRdU	Redefined + Virtual + Operation has no body implementation (interface only)

$$\Sigma = \{V, N, Rd, Rv, U\}$$
$$\mu = (V\ ?\ (N \mid Rd \mid Rv\ )\ U\ ?)$$

Figure 1. Operation Classification Language

Table 4. Operation Type Expression Examples

Regular Expression Expansion	Meaning for $Operation_i(C)$
*	All operations for C
V or V*	All virtual operations for C
*NU	All New-Undefined operations for C
*U	All Undefined operations for C
V*U	All Virtual-Undefined operations for C
VrvU	All Virtual-Recursive-Undefined operations for C

Table 3. Effect of Operation Type on Class Absractness

Operation Type	Extendibility	Abstractness/Concreteness Constant
New	None	2
Recursive	None	2
Redefined	None	2
Virtual-New	Some	1
Virtual-Recursive	Some	1
Virtual-Redefined	Some	1
Virtual-New-Undefined	Very	0
Virtual-Recursive-Undefined	Very	0
Virtual-Redefined-Undefined	Very	0

$$\text{Abstractness}(C) = \frac{\left|\text{Operations}_{V(N|\,Rv|\,Rd)}(C)\right| + \left(\left|\text{Operations}_{(N|\,Rv|\,Rd)}(C)\right| \cdot 2\right)}{\left|\text{Operations}_{*}(C)\right| \cdot 2}$$

Equation 1. Class Abstractness

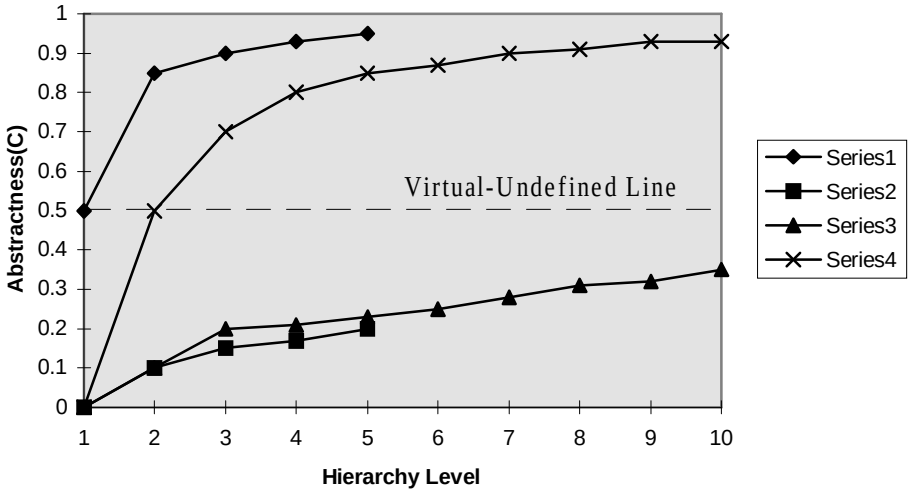

Figure 2. Deep/Shallow vs. Abstract/Concrete

Table 5. Brittleness Measurement Comparison

Hierarchy Chain(H)	NL(H)	NAC(H)	Brittleness(H)
LineSegment	2	1	0.5250
Rectangle	2	1	0.5078
Sphere	2	1	0.4205
Ellipse	2	1	0.5125
Polygon	2	1	0.4950

Table 8. Effect of Operation Type on Class Identity

Operation Category	Identity Cohesion Contribution
New	Yes
Recursive	No
Redefined	Yes
Virtual-New	Yes
Virtual-Recursive	No
Virtual-Redefined	Yes
Virtual-New-Undefined	No
Virtual-Recursive-Undefined	No
Virtual-Redefined-Undefined	No

$$CAC(C) = \frac{\sum_{t \in Operation_{(*N|*Rd)}(C))} \#\text{operations in C containing t}}{|Attribute(C)| \cdot |Operation_{(*N|*Rd)}(C)|}$$

Equation 2. Class Abstraction Cohesion

Table 6. CAC Measurement Comparison Example

Class C	LCOM(C)	CAC(C)
LinearGeometry	3	0.500
Ellipse	70	0.667
CompositeGeometry	0	1.000

Table 7. Operation Type and the Liskov Substitution Principle

Code	Results of Inheritance in a Subclass	LSP
A	Adding new operations	Yes
B	Overriding old behavior with the old behavior plus some new behavior (augmentation)	Yes
C	Overriding operations with entirely new behavior (replacement)	Maybe
D	Overriding operations with no new or old behavior (deleting)	No
E	Overriding virtual operations which have no body	Yes
F	Inheriting an operation with no changes	Yes

Table 9. Operation Classification and the LSP

Subclass Operation Type	Code Reference	LSP
New	A	Yes
Recursive	F	Yes
Redefined	B,C	Maybe
Virtual-New	A	Yes
Virtual-Recursive	F	Yes
Virtual-Redefined	B,C,E	Maybe
Virtual-New-Undefined	A	Yes
Virtual-Recursive-Undefined	F	Yes
Virtual-Redefined-Undefined	D	No

$$PII(C) = \frac{P(C) + O(C) + N(C)}{3}$$

Equation 3. Pure Inheritance Index

$$P(C) = \frac{|Operation^{*}{}_{Rd}{}^{*}(C)| - |Preserved(Operation^{*}{}_{Rd}{}^{*}(C))|}{|Operation^{*}{}_{Rd}{}^{*}(C)|}$$

Equation 4. Specialization and Preserved Behaviors

$$O(C) = \frac{|Operation^{*}{}_{Rd}(C)|}{|Operation_{(*Rd|*Rv)}(C)|}$$

Equation 5. Specialization and Overridden Behaviors

$$N(C) = \frac{HLevel(C)}{|Operation^{*}{}_{N}{}^{*}(C)| + 1} \bullet \frac{|Operation^{*}{}_{N}{}^{*}(C)|}{HLevel(C) + 1}$$

Equation 6. Specialization and Decreasing Operations

Table 10. PII Measurement Comparison

Class C	SI(C)	PII(C)
Geometry	0.000	0.000
SurfaceGeometry	0.600	0.600
Polygon	1.000	0.144

$$RAI(C) = \frac{\sum_{Cx} Abstractness(Cx) | Cx \in (Relation_{As}(C) U Relation_{Ag}(C))}{|Relation_{As}(C) U Relation_{Ag}(C)|}$$

Equation 5. Relationship Abstractionn Index for Class Scope

$$RAI(CompositeGeometry) = UNDEFINED$$

$$RAI(Rectangle) = Abstractness(Rectangle) = 0.731$$

$$RAI(Geometry) = \frac{\sum_{Cx} Abstractness(Cx)}{10} = \frac{6.02}{10} = 0.60$$

where Cx = {Geometry, LinearGeometry, SurfaceGeometry, VolumeGeometry, CompositeGeometry, LineSegment, Rectangle, Ellipse, Polygon, Sphere}

$$RAI_{INT}(Geometry) = RAI(Geomtr y) = 0.602$$

$$RAI_{EXT}(Geometry) = UNDEFINED$$

Figure 3. RAI Calculation

Table 11. Overview of measurements

Measurement Name	Dependent Relationship			Class Members		Measurement Scope			Object-Oriented Attributes			
	In	Ag	As	Ops	Attrs	Class	Hier	Cat	ID	Poly	In	En
Class Abstractness	x			x		x					x	
Hierarchy Chain Brittleness	x			x			x				x	
Class Abstraction Cohesion				x	x	x			x			
Pure Inheritance Index	x			x		x					x	
Relationship Abstraction Index		x	x	x		x		x		x		

Table 12. Measurement to OO Principle Mapping

Measurement Name	Object-Oriented Principles							
	OCP	LSP	DIP	REP	CCP	CRP	LAP	DAHP
Class Abstractness	x	x						
Hierarchy Chain Brittleness							x	x
Class Abstraction Cohesion								
Pure Inheritance Index		x						
Relationship Abstraction Index		x	x					x

Table 13. Measurement Results Interpretation

Measurement Name	Range		Result	
	Min	Max	Best	Worst
Class Abstractness	0.0	1.0	-	-
Hierarchy Chain Brittleness	0.0	1.0	-	-
Class Abstraction Cohesion	0.0	1.0	1.0	0.0
Pure Inheritance Index	0.0	1.0	0.0	1.0
Relationship Abstraction Index	0.0	1.0	0.0	1.0

Table 16. Experimental Code Set Summary

Code Set	Code Set Size			Type	Description
	NCLOC	Classes	Depth		
1	404	8	2	library	GUI + functionality
2	1947	16	3	library	GUI + functionality
3	12657	116	7	library	domain hierarchy
4	10710	52	4	library	GUI + functionality
5	1336	11	4	library	domain hierarchy
6	2225	16	3	library	GUI + functionality
7	12613	68	4	application	GUI + functionality + 3rd party library wrapper
8	836	4	2	library	GUI + functionality

Table 15. Hierarchy Brittleness Results

Code Set	Data Count	Average	Low	High	Detections
1	4	0.428	0.357	0.482	0
2	10	0.385	0.317	0.485	0
3	116	0.534	0.303	0.740	13
4	68	0.651	0.333	0.786	28
5	7	0.390	0.309	0.418	0
6	9	0.453	0.338	0.667	0
7	53	0.419	0.292	0.703	2
8	2	0.455	0.431	0.480	0

Table 14. Class Abstraction Cohesion Results

Code Set	Data Count	Average	Low	High	Detections
1	4	0.337	0.214	0.500	0
2	10	0.581	0.176	1.000	1
3	77	0.272	0.038	0.750	22
4	44	0.353	0.849	0.875	0
5	10	0.313	0.150	0.500	1
6	10	0.390	0.079	1.000	1
7	23	0.272	0.077	0.667	11
8	2	0.241	0.149	0.333	1

Table 17. Pure Inheritance Index Results

Code Set	Data Count	Average	Low	High	Detections
1	4	0.148	0.139	0.156	0
2	11	0.323	0.139	0.694	0
3	93	0.453	0.133	0.789	1
4	42	0.373	0.125	0.591	0
5	10	0.561	0.157	0.713	1
6	11	0.258	0.133	0.728	2
7	58	0.297	0.125	0.803	4
8	2	0.159	0.158	0.160	0

Table 18. Relationship Abstraction Index Results

Code Set	Data Count	Average	Low	High	Detections
1	3	0.904	0.850	0.960	3
2	8	0.739	0.292	0.902	7
3	40	0.808	0.302	1.000	30
4	19	0.752	0.430	0.969	13
5	1	0.706	0.706	0.706	1
6	2	0.596	0.314	0.878	1
7	12	0.835	0.500	1.000	6
8	2	0.911	0.861	0.960	2

Table 18. Other Measurements Results

Code Set	NL (max)	NAC	SI			LCOM		
			Avg.	Low	High	Avg.	Low	High
1	1	0	0	0	0	7.5	5	11
2	3	1	1.243	0.900	1.500	8.0	1	15
3	7	4	2.576	0.600	4.667	11.3	1	66
4	4	1	1.138	0.105	2.667	9.5	1	55
5	4	3	1.535	0.462	2.095	15.9	7	22
6	3	1	1.012	0.900	1.125	10.4	1	50
7	4	5	1.110	0.429	1.500	8.6	1	28
8	2	0	0	0	0	17.5	13	22

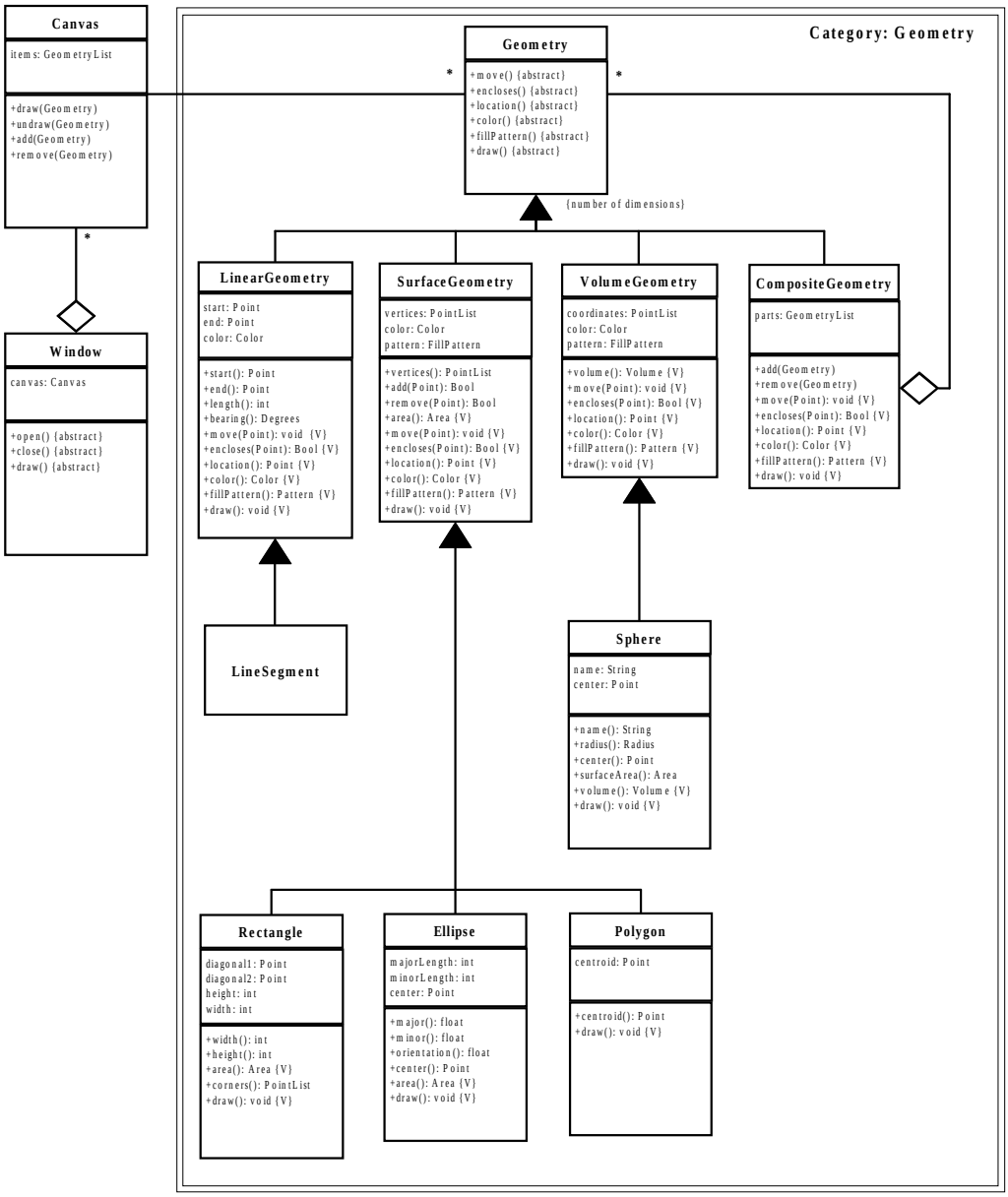

Figure 4. Measurement Example