This paper addresses Object-Oriented (OO) system complexity at the application, object, method, and variable levels. At each level measures are proposed to account for the cohesion and coupling aspects of the system. OO system complexity at each level is presented as a function of the measurable characteristics such as fan-in, fan-out, number of I/O variables, fan-up, fan-down, and polymorphism. Each measure is defined with adherence to the principles that measures must be intuitive and that they must be applicable to all phases of the OO development life cycle.

I. INTRODUCTION

Complexity is one of the major factors in the cost of developing and maintaining software. Measurement of software complexity has been of great interest to researchers in software engineering for some time. One of the key reasons for this interest is the potential to use the measurements in procedures to control costs of a system over its lifetime. According to Coad and Yourdon (1991),

A good design is one that balances trade-offs to minimize the total cost of the system over its entire lifetime.

We propose that software complexity measures are necessary to accomplish two goals necessary to make trade-offs and to reduce costs over the lifetime of a system. These goals are:
1) To provide a quantitative method for predicting how difficult it will be to maintain, change and understand the system.

2) To provide a basis for making the trade-offs necessary to reduce costs over the lifetime of the system.

In this paper we propose measures at the variable, method, object, and application levels to allow the overall complexity of an Object-Oriented (OO) system to be controlled by trading off complexity between the levels. The proposed measures account for both process and data characteristics of an OO system. They also provide analysts, designers, and programmers of OO systems with the ability to identify which components of an OO system are the most complex, and may therefore require additional analysis, design, or testing. The methodology for making the trade-offs between the levels and between the measures within the levels are areas for future research.

The second section of the paper provides the background for the creation of our proposed measures. The third section defines a set of measurable characteristics of OO systems that provide a basis for the calculation of OO system complexity. The final section contains a summary and future research directions.

II. Background

This section of the paper provides an operational definition of software complexity, a set of desirable properties of software complexity measures, presentation of some specific aspects of OO systems that are different from structured systems, and a set of characteristics of good OO systems design.

Definition of Software Complexity

Software complexity is an area of software engineering concerned with the measurement of factors that affect the cost of developing and maintaining software. Zuse (1990) argues that the term "software complexity" is poorly defined and that software complexity measurement is a misnomer. He offers a definition of software complexity that is consistent with our approach:

The true meaning of software complexity is the difficulty to maintain, change and understand software. It deals with the psychological complexity of programs. (Zuse, 1990)

Three specific types of psychological complexity that affect programmer ability to comprehend software have been identified (Card and Glass, 1990; Conte et al, 1986) as problem complexity, system
design complexity, and procedural complexity.

Problem complexity is a function of the problem domain. Simply stated, it is assumed that complex problem spaces are more difficult for a programmer to comprehend than simple problem spaces. Since this type of complexity is impossible to control, it generally is ignored in software engineering.

System design complexity addresses the complexity associated with mapping a problem space into a given representation. An overall rating of the complexity of a system (System Complexity) consists of the sum of the individual module complexities associated with the module's connections to other modules (Structural Complexity) and the amount of work the module performs (Data Complexity) (Card and Glass, 1990). Structural complexity and data complexity are the two types of system design complexity defined for structured systems (Card and Glass, 1990). Structural complexity addresses the concept of coupling. Coupling measures the interdependence of modules of source code, e.g., C functions calling other C functions. It is assumed that the higher the coupling between modules, the more difficult it is for a programmer to comprehend a given module. Data complexity addresses the concept of cohesion. Cohesion measures the intradependence of modules. In this case, it is assumed that the higher the measure of cohesiveness, the easier it is for a programmer to comprehend a given module (Stevens, Myers and Constantine, 1976). The structural and data complexity measures are based on the module's fan-in, fan-out, and number of input/output variables. These metrics address information system complexity at the system and module levels.

Procedural complexity is associated with the complexity of the logical structure of a program. This approach to complexity measurement assumes that the length of the program [number of tokens or the lines of code] or the number of logical constructs [sequences, decisions, or loops] that a program contains determines the complexity of the program (Card and Glass, 1990).

This paper addresses only system design complexity for OO systems.

Desirable Properties of Software Measures

Many authors propose desirable properties of measures for effective evaluation of software complexity (McCabe, 1976; Tsai et al, 1986; Weyuker, 1988). We believe that two essential properties should be used to create a set of measures of the characteristics of software and designs that are correlated
with the concept of psychological software complexity. First, the measures should be applicable at all phases of the development process. Measures, such as lines of source code, that have an inherent dependence on completing significant portions of the development effort before the measure can be applied ignore potential opportunities to control psychological software complexity in early phases of the development process. Given that a majority of effort and cost associated with software projects occurs in the maintenance phase (Bohem, 1976), it is critical that measures be applicable to existing systems. Second, the measures must be intuitive in nature. By intuitive, we mean, that analysts, designers, and programmers can agree on both aggregate and component measures. Widely researched measures, such as those proposed as software science (Halstead, 1977) and cyclomatic complexity (McCabe, 1976), demonstrate this property.

In addition to the essential properties of measures identified above, the following properties are also highly desirable (Tsai, et. al., 1986; Zuse 1990):

1) A measure of both the structure of the data and process must be included;

2) The measures must demonstrate consistency (i.e., if a structure is contained within another structure, the containing structure is computed to be at least as complex as the contained structure);

3) Measures must be capable of representing at least a weak order (i.e., the measures must be at least on an ordinal scale);

4) Measures must be additive (i.e., if two independent structures are put into sequence then the total complexity of the combined structures is simply the sum of the complexities of the independent structures); and

5) It must possible to automate the measures.

Measures defined for structured systems that are consistent with the characteristics of good OO systems and the desirable properties of software measures presented above are those based on information flow and data bindings (Henry and Kafura, 1981; Card and Agresti, 1988; Belady and Evangelisti, 1981; Hutchens and Basili, 1985). These measures focus on the coupling and cohesion characteristics of structured systems.

**Structured Systems Versus OO Systems**

Polymorphism and inheritance are two aspects unique to OO systems that tend too reduce procedural software complexity (Tegarden, Sheetz, and Monarchi, 1992) of an OO system as measured
by the traditional software measures of software science (Halstead, 1977) and cyclomatic complexity
(McCabe, 1976). Encapsulation is not a new concept pertaining to only OO systems. But the ability to
exploit advantages of using this concept to control the coupling and cohesion of a system are inherent to
OO approaches. Encapsulation, polymorphism, and inheritance are described below.

Encapsulation

OO systems integrate both the structural [data] and behavioral [operations] aspects of a program,
while structured systems force an artificial separation of the structure from behavior. According to Brodie
and Ridjanovic (1984),

The separate treatment of structure and behavior complicates design, specification,
modification, and semantic integrity analysis.

Encapsulation, inherent in OO approaches, allows programmers to focus on the current object description
and avoid creating unwanted side effects in other objects.

Polymorphism

Polymorphism means having the ability to take several forms. For OO systems, polymorphism
allows the implementation of a given operation to be dependent on the object that contains the operation.
For example, there can be different operations to pay employees based on the employee's object type, e.g.,
part-time, hourly, and salaried. Each type of employee object can have its own customized compute pay
operation. When a new type of employee is created, (e.g., student), the programmer simply creates a new
type of employee object and a new operation to pay the new type of employee in the new object. When
the new object receives the message to compute pay, it uses the operation defined in the new object to
perform the calculation. The compute pay operations of the other types of employee are not affected by
the payment operations required for the new type of employee. In contrast, structured systems often have
all pay operations contained in one program. The program must be capable of differentiating between the
different types of employees and applying the appropriate operation. Modifications to add a new type of
employee may require existing code to be changed.

Inheritance

If there is one prototypical characteristic of OO systems, it is the existence of an inheritance
mechanism. There have been many different types of inheritance mechanisms associated with OO systems
Inheritance is a reuse mechanism that allows programmers to define objects incrementally by reusing previously defined objects as the basis for new objects. For example, when defining a new type of employee, e.g., student, the new employee type can inherit the common characteristics of being an employee from a generic type of employee. In this approach, the programmer needs only to be concerned with the difference between student employees and generic employees. Structured systems do not have inheritance mechanism as part of their formal specification.

**Characteristics of Good OO Systems Design**

A set of criteria for a good OO design have been identified using the concepts of coupling, cohesion, reuse, clarity, depth of the generalization-specialization hierarchy, simplicity, and size (Coad and Yourdon, 1991). In addition, the creation of measures for the evaluation of a specific object has been suggested to include the concepts of coupling, cohesion, sufficiency, completeness, and primitiveness (Booch, 1991). The criteria of clarity, simplicity, sufficiency, completeness, and primitiveness appear to be tied to the ability of the OO system to support problem domain requirements and therefore are associated with problem complexity (as defined above). As such, it is not clear how they can be calculated from the available static representations (source code or designs). In this paper we propose measures of the coupling and cohesion of OO systems at the variable, method, object, and application levels. In addition, we propose measures of the depth of the generalization-specialization hierarchy and size at the object and application levels. These all can be calculated from the available static representations of an OO system. Cohesion and coupling of OO systems have been defined to include service cohesion, class cohesion, interaction coupling and inheritance coupling (Coad and Yourdon, 1991). The measures defined below provide a basis for quantifying each of these concepts. In this paper we do not present a measure for the reuse of OO systems.

**III. Measures for OO Systems**

The complexity of OO systems can be represented by a set of measures defined at the variable, method, object, and application levels of the system. We present a set of measures at each level that consist of counts of characteristics of the OO system at that level and the effects of those characteristics
on OO system complexity. The combined set of all measures from all the levels results in a representation of OO system complexity applicable to the design and implementation phases of OO system development from both macro and micro views. The following presents for each level the counted characteristics, the counting rules for each characteristic, and a simple technique for making the counts.

**Variable Level Measures for OO Systems**

At the variable level, the characteristics of data type, fan-in, fan-out, and variable polymorphism contribute to the complexity of a variable.

*Definition of Variable Data Type (vdt):*

This a value representing whether the variable is an atomic or composite variable. The value is zero if it is atomic (e.g., an integer, string, or other primitive data type), one if it is composite (e.g., some kind of structure).

*Definition of Variable Fan-in (vfi):*

This is the number of methods that access a variable, ignoring how many times a method accesses the variable.

These measures can be calculated through the creation of the variable method connection matrix (VMCM). In the VMCM the rows are identified by the combination of object name and method name and the columns are identified by the combination of object name and variable name, with additional columns identified by object name to allow for each object to be returned as the result of a message. Each cell of the matrix contains a one if the method of the row accesses the variable of the column and a zero if it does not. The sum of the cell values down each column results in the total number of methods that access a variable or object.

*Definition of Variable Fan-out (vfo):*

The number of different kinds of objects or pointers to objects that can be placed in the variable.

A simple technique for calculating vfo is through the creation of a variable connection matrix (VCM). In a VCM, the rows are identified by the combination of object name and variable name. The columns of the VCM are identified by the objects that make up the application. Each cell of the matrix contains a one if the variable of the row can be assigned a value from the domain of the object of the column, otherwise the cell contains a zero. The sum of the cell values across each row results in the total
number of domains, objects, from which the variable can be assigned a value.

*Definition of Variable Polymorphism (vp):*

The number of times a variable name is defined.

If a variable is defined in more than one object then the variable is polymorphic.

A simple technique for counting the variable polymorphism is to create a variable polymorphism matrix (VPM). In a VPM the rows are identified by each unique variable name without the object name prefix and the columns are identified by the object name. Each cell contains a one if the variable name of the row is defined in the object of the column, otherwise it contains a zero. The sum of the cells in the row is the variable polymorphism. If the variable polymorphism is greater than one, the variable of the row is polymorphic.

The concept of fan-down is associated with the object hierarchy. We assume that when a variable will be inherited and used by other objects that a programmer must consider those objects when developing or maintaining the method(s) that access the variable. This assumption results in the following definition of variable fan-down.

*Definition of Variable Fan-down (vfd):*

The number of objects below the object in which the variable is defined in the object hierarchy. In other words, the count of the subclasses that inherit the variable.

Inheritance is measured with fan-down. Variable fan-down can be measured through the creation of a variable inheritance matrix (VIM). A VIM is a measure of the inheritance lattice associated with variables in the application. Each row of the matrix is identified by the combination of object name and variable name, and each column of the matrix is identified by object name. Each cell of the matrix contains a one if the variable of the row is inherited by the object of the column, otherwise it contains a zero. Variable fan-down for each variable is defined as the sum of the cells across the row of the variable.

The concepts of cohesiveness and coupling are not normally defined for variables in structured systems. We propose that measures of this type are important for variables in OO systems. These measures combine to provide measures of how cohesive a variable is, intra-variable complexity, and how much coupling of variables exists in the system, inter-variable complexity.
Intra-variable Complexity

For variables, cohesiveness means the degree to which the variable is used in the same context as compared to the amount it is used in other contexts. We propose that intra-variable complexity is a function of data type, variable fan-out, and variable polymorphism. Furthermore, we believe that all three characteristics contribute to increase the intra-variable complexity and therefore reduce the amount of cohesiveness of the variable. In addition, we propose that the variable polymorphism does not contribute as much as the data type and that fan-down is the strongest contributor to intra-variable complexity.

Inter-variable Complexity

If two variables are accessed by the same method(s), they are connected. This is a measure of interaction coupling (Coad and Yourdon, 1991). Coupling for variables is associated with the number of methods that can use the variable, the number of domains from which the variable can take values, and variable polymorphism. We propose that variable fan-in and fan-out contribute to increase the inter-variable complexity and variable polymorphism contributes to decrease inter-variable complexity. In addition, variable polymorphism does not contribute as much as the fan-in and fan-out, and that fan-in and fan-out contribute in similar magnitudes.

Method Level Measures for OO Systems

At the method level the number of input/output variables, fan-in, fan-out, fan-down, and method polymorphism contribute to the complexity of a method. Each of these method level characteristics is defined by the counting rules used to compute the measure for each method in an OO system.

Card and Agresti (1988) identified the number of input/output variables as a component of the intramodule complexity of a module in a structured system. The number of input/output variables for OO systems at the method level consists of five parts: the number of input parameters in the message from the sending method, the number of parameters returned to the sending method, the number of object variable accesses, the number of parameters that the method must pass to other methods as part of sending messages, and the number of parameters returned to the method containing the results of messages sent by the method. Each of these parts is defined below.
Definition of Method Input Parameters (mip):

The number of parameters passed to the method as part of the message. These parameters contain the data from the sending method that the receiving method needs to perform the operation for which it was defined. Each of these parameters becomes a local variable within the method providing the basis required for calculation and flow of control.

Most methods also return a parameter, usually a value or a pointer to an object. The creation of this parameter is the purpose for which the method is defined.

Definition of Method Parameters Returned (mpr):

The number of parameters returned to the sending method.

In addition to the data passed to the method as part of the message, additional data may be required to complete the operations defined in the method. This data may be brought into the method by accessing object variables or by sending a message to another object.

Definition of Object Variables Accessed (ova):

The number of object variables accessed is the count of the object variables that are used plus the count of the object variables assigned.

The object variables accessed can be either inputs or outputs of the method. With the assumption that the programmer must understand the entire object to modify the methods contained in the object, there is no need to differentiate between inputs and outputs for object variables as the complexity associated with the variable must be overcome only the first time the variable is used or assigned.

Definition of Parameters Returned to the Method (prm):

The number of parameters returned to the method contain the results of the operations performed by other methods in response to a message.

Similar to the mip, the parameters returned to the method are inputs to the method that are used for calculation and flow of control.

To obtain the correct response from the messages sent, the messages must include the parameters required by the receiving methods. The method sending the message must provide these parameters.
**Definition of Method Parameters Passed (mpp):**

The number of unique parameters passed to other methods by the method as part of a message are outputs required of the method to defer part of the work it must do to other methods. If a parameter is passed as part of more than one message the parameter is counted only once.

Together these measures represent the flow of information through the method.

These measures can be calculated using the VMCM defined above. The sum of the cell values across each row results in the total number of variables accessed by the method. This value can also be calculated with the following formula:

\[ miov_i = mip_i + mpr_i + ova_i + prm_i + mpp_i \]

where \( miov \), is the total number of input/output variables for method \( i \) and the other variables are as described above for method \( i \). This is a measure of the amount of information flow associated with the method.

In addition to the previous measures associated with the data accessed by the method, measures representing the use of the method and its use of other methods also are required. These measures consist of the fan-in and fan-out of the method.

**Definition of Method Fan-in (mfi):**

The number of methods that send messages to the method, ignoring how many times a method accesses the method.

**Definition of Method Fan-out (mfo):**

The number of methods that receive messages from the current method. This includes the count of different messages used to access object variables, to invoke other methods defined in the object, and to obtain data from other objects. What is recorded is that the method uses another method not the number of times the other method is used.

A simple technique for calculating these measures is through creation of the method connection matrix (MCM). In the MCM the rows and columns are identified by the combination of the object name and method name. Each cell of the matrix contains a one if the method of the row sends a message to the method of the column, otherwise it contains a zero. The sum of the cell values across each row results in the total number of messages sent by the method or the fan-out of the method. The sum of the values of cells from a column is the number of messages received by the method or the fan-in of the method.

In addition, we assume it is important to differentiate between the messages sent to the object in
which the method is defined and those messages sent to other objects. This differentiation is important
due to the increased amount of effort required by a programmer to use methods that are defined in objects
that are not part of the current object. At the minimum, the programmer must obtain the public interface
of the object that contains the method, select a message from the interface, and determine what available
variables are appropriate parameters for the message. Therefore, it is more difficult to use a method
defined outside the current object. This is similar to differentiating between the use of local functions
versus shared library functions in traditional systems. This differentiation is not common among measures
of structured systems. However, given the extensive nature of and the reuse intended for most OO class
libraries, a distinction at this level is intuitively appealing. To account for this increase in complexity we
rely on the idea that the sending a message to another object requires a pointer to that object. The value
of this pointer must be contained in a variable, therefore, variable fan-out as defined above also captures
some of the complexity involved with sending messages to other objects.

Method polymorphism is defined and calculated in that same manner as variable polymorphism.
This results in the following definition of method polymorphism.

*Definition of Method Polymorphism (mp):*

The number of objects in which a method name is defined.

A method polymorphism matrix (MPM) is used to calculate method polymorphism. In an MPM the rows
are identified by method name and the columns are identified by object name.

The concept of fan-down is associated with the object hierarchy. Method fan-down is defined and
calculated in the same manner as variable fan-down. We assume that when a method will be inherited
and used by other objects, a programmer must consider those objects when developing or maintaining the
method. This assumption results in the following definition of method fan-down.

*Definition of Method Fan-down (mfd):*

The number of objects below the object in which the method is defined in the object
hierarchy. In other words, the count of the subclasses that inherit the method.

Method inheritance matrix (MIM) is similar to the VIM except that the rows identify methods rather than
variables. The calculation of method fan-down on the MIM is the same as that for variable fan-down on
the VIM.
Intra-method Complexity

The set of measures defined for intra-method complexity apply to the concept of service cohesion as defined by Coad and Yourdon (1991). These measures include an extension of data complexity (Card and Glass, 1990) to OO systems and other measures that can be derived using the counts defined above.

We believe that method polymorphism, method fan-down, and the number of input/output variables increase the intra-method complexity of the method. This is due to the requirement that the programmer must understand how to maintain consistent semantics across the application if polymorphism is used, understand what the effects of changing the current method are on objects lower in the object hierarchy, and that input/output variables increase intra-method complexity. Furthermore, we believe that method fan-out decreases intra-method complexity. The method fan-out reduces intra-method complexity by deferring work to other methods through message passing. Intra-method complexity incorporates the polymorphic characteristics, data access requirements, and the amount of work deferred for a method to create a value for the internal complexity of each method.

Inter-method Complexity

Connections between methods are established through message-passing. A measure of inter-method complexity based on the message-passing and polymorphism characteristics of OO systems is proposed using fan-in, fan-out, the number of input/output variables, and method polymorphism. We believe that the number of input/output variables, fan-in, fan-out, and method polymorphism increase the inter-method complexity. Furthermore, that fan-in and fan-out have a greater effect than does the number of input/output variables and that method polymorphism has the least effect. Inter-method complexity is another measure of interaction coupling as defined by Coad and Yourdon (1991).

Object Level Measures for OO Systems

At the object level characteristics are defined to account for psychological complexity. These characteristics are grouped together based on the concepts that they measure. This results in six groups: input/output variables, property definitions, message passing, inheritance structure, inheritance conflicts, and object polymorphism. Each of these object level groupings is defined by the characteristics included
in the group and the counting rules used to create the measure for each characteristic for each object in an OO system.

The number of input/output variables for OO systems at the object level consists of several components. These components are the same as those defined at the method level, except that the number of object variables accessed is not included. Each component at the object level is computed in a manner similar to the corresponding component at the method level, except that the object is the focus. Each of these components is defined below. The number of input/output variables for OO systems at the method level consists of four parts: the number of input parameters in the message from the sending method, the number of parameters returned to the sending method, the number of parameters that the method must pass to other methods as part of sending messages, and the number of parameters returned to the method containing the results of messages sent by the method. Each of these parts is defined below.

Definition of Object Input Parameters (oip):

The number of parameters passed to the object as part of the messages sent from outside the object.

These parameters contain the data from the sending object that the receiving object needs to perform the requested operation. This does not include the parameters associated with methods defined within the object sending messages to other methods defined in the object.

Definition of Object Parameters Returned (opr):

The number of unique parameters returned to sending object(s).

Most messages require that a value or a pointer to an object be returned in response to the message. For most objects this should be the number of public methods in the object protocol.

In addition to the data passed to the object as part of the message, additional data may be required to complete the operations defined for the object. This data may be brought into the method by sending a message(s) to other object(s). With the assumption that the programmer must understand the entire object to modify the methods contained in the object, there is no need to count the number of messages passed between methods defined within the object.
Definition of Parameters Returned to the Object (pro):

The number of parameters returned to the object contain the results of the operations performed by other objects in response to a message.

Similar to the $oip$, the parameters returned to the object are inputs to the method that are used for calculation and flow of control.

To obtain the correct response from the messages sent, the messages must include the parameters required by the receiving object(s). The object sending the message must provide these parameters.

Definition of Object Parameters Passed (opp):

The number of unique parameters passed to other objects by the object as part of a message are outputs required of the object to defer part of the work it must do to other methods. If a parameter is passed as part of more than one message, the parameter is counted only once.

Together these measures represent the flow of information through the object.

These measures can be calculated through the creation of a object variable connection matrix (OVCM). In a OVCM the rows are identified by the object name and the columns are identified by the combination of object name and variable name. The sum of the total cells for a row are the total of number of variables accessed by the object. This value can also be calculated with the following formula:

$$oiov_i = opr_i + pro_i + opp_i$$

where $oiov$ is the total number of input/output variables for object $i$ and the other variables are as described above for object $i$. This is a measure of the amount of information flow associated with the object.

Distinguishing between locally defined properties and inherited properties is also possible at the object level. The number of properties and the number of methods that reference the properties are included in Table 1. Several different useful summations can be made from the numbers of methods and variables and their referencers definitions are included in Table 2. Together these characteristics represent the degree to which the operations performed by the object are carried out through properties defined in the object versus through the use of inherited properties.
<table>
<thead>
<tr>
<th>Object Characteristics</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Variables (olv):</td>
<td>The number of locally defined variables in the object.</td>
</tr>
<tr>
<td>Local Variables Local Referencers (olvlr):</td>
<td>The number of locally defined methods in the object that reference locally defined variables.</td>
</tr>
<tr>
<td>Local Variables Inherited Referencers (olvir):</td>
<td>The number of locally defined methods in the object that reference locally defined variables.</td>
</tr>
<tr>
<td>Inherited Variables (oiv):</td>
<td>The number of inherited variables in the object.</td>
</tr>
<tr>
<td>Inherited Variables Local Referencers (oivlr):</td>
<td>The number of locally defined methods in the object that reference inherited variables.</td>
</tr>
<tr>
<td>Inherited Variables Inherited Referencers (oivir):</td>
<td>The number of inherited methods in the object that reference inherited variables.</td>
</tr>
<tr>
<td>Local Methods (olm):</td>
<td>The number of locally defined methods in the object.</td>
</tr>
<tr>
<td>Local Methods Local Referencers (olmlr):</td>
<td>The number of locally defined methods that reference locally defined methods.</td>
</tr>
<tr>
<td>Local Methods Inherited Referencers (olmir):</td>
<td>The number of inherited methods that reference locally defined methods.</td>
</tr>
<tr>
<td>Inherited Methods (oim):</td>
<td>The number of inherited methods in the object.</td>
</tr>
<tr>
<td>Inherited Methods Local Referencers (oimlr):</td>
<td>The number of locally defined methods that reference inherited methods.</td>
</tr>
<tr>
<td>Inherited Methods Inherited Referencers (oimir):</td>
<td>The number of inherited methods that reference inherited methods.</td>
</tr>
</tbody>
</table>
### Table 2: Useful Summations of Object Properties

<table>
<thead>
<tr>
<th>Total Object Summation Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Properties:</strong></td>
<td>$top = oim + olm + oiv + oim$</td>
</tr>
<tr>
<td><strong>Referencers:</strong></td>
<td>$topr = olvlr + olvir + oivlr + oimlr + olmir + oimlr + oimir$</td>
</tr>
<tr>
<td><strong>Inherited Properties:</strong></td>
<td>$tiop = oim + oiv$</td>
</tr>
<tr>
<td><strong>Inherited Properties Referencers:</strong></td>
<td>$tiopr = oimlr + oivlr + oimir + oimir$</td>
</tr>
<tr>
<td><strong>Local Properties:</strong></td>
<td>$tlop = olm + olv$</td>
</tr>
<tr>
<td><strong>Local Properties Referencers:</strong></td>
<td>$tlopr = olmlr + olvlr + olmir + olvir$</td>
</tr>
<tr>
<td><strong>Local Properties:</strong></td>
<td>$tlop = olm + olv$</td>
</tr>
<tr>
<td><strong>Local Properties Referencers:</strong></td>
<td>$tlopr = olmlr + olvlr + olmir + olvir$</td>
</tr>
<tr>
<td><strong>Variables:</strong></td>
<td>$tov = olv + oiv$</td>
</tr>
<tr>
<td><strong>Variables References:</strong></td>
<td>$tovr = olvlr + olvir + oivlr + oivir$</td>
</tr>
<tr>
<td><strong>Methods:</strong></td>
<td>$tom = olm + oim$</td>
</tr>
<tr>
<td><strong>Methods References:</strong></td>
<td>$tomr = olmlr + oimlr + oimlr + oimir$</td>
</tr>
</tbody>
</table>

In addition to the previous measures associated with the data accessed by the object, measures representing the use of the object and the object’s use of other objects also are required. These measures consist of the fan-in and fan-out of the object.

**Definition of Object Fan-in (ofi):**

The number of objects that send messages to the object, ignoring how many times an object sends messages to the object.

**Definition of Object Fan-out (ofo):**

The number of unique messages sent by an object.

A simple technique for calculating these measures is through creation of a object connection matrix (OCM). In an OCM, the rows and columns are identified by object name. Each cell of the matrix contains a one if the object of the row sends a message to the object of the column. The sum of the cell values across each row results in the total number of unique messages sent by the object or the fan-out of the object. The sum of the values of cells from a column is the number of messages received by the
object or the fan-in of the object. This matrix is similar to the matrix derived from the message graph proposed by Moreau and Dominick (1990).

The characteristics of fan-down and fan-up are associated with the object hierarchy and are used as a measure of inheritance. Fan-up and fan-down are measures of inheritance coupling as defined by Coad and Yourdon (1991).

*Definition of Object Fan-down (ofd):*

The number of objects below the object in which the method is defined in the object hierarchy. In other words, the count of the subclasses that inherit from the object.

*Definition of Object Fan-up (ofu):*

The number of objects above the object in which the object hierarchy. In other words, the count of the superclasses that the object inherits from.

Other measures of inheritance are the number of levels of the hierarchy that are above the object and the number of levels of the object hierarchy that are below the object. These are known as the object to root depth and object to leaf depth and are defined as follows:

*Definition of Object to Root Depth (ord):*

The maximum number of levels in the object hierarchy that are above the object.

*Definition of Object to Leaf Depth (old):*

The maximum number of levels in the object hierarchy that are below the object.

The object inheritance matrix (OIM) is similar to the VIM and MIM except that it is for objects rather than variables or methods. Each row and column of the matrix is identified by object name. Each cell of the matrix contains a one if the object of the row is a subclass of the object of the column otherwise it contains a zero. The calculation for object fan-down is exactly as defined for method fan-down except that it operates on the OIM. The calculation of fan-up is the sum of the cell values from the column identified by the object. The OIM matrix is similar to the matrix derived from the inheritance graph proposed by Moreau and Dominick (1990).

The effects of inheritance conflicts are also of interest as they represent a clear opportunity for the programmer to become confused. An inheritance conflict occurs when a method or variable in one object has the same name as a method or variable in a super object. This is called a conflict because the
programmer must decide which of the methods or variables to use. This requires additional effort as compared to using methods or variables that have no conflicts. We propose that there are two types of inheritance conflicts, local to parent and parent to parent. In a local to parent inheritance conflict, a property has been defined in the local object that has the same name as a property that was inherited. These are defined as:

**Definition of Local to Parent Inheritance Conflicts (lpic):**

The number of properties defined in the object which have the same name as an inherited property.

**Definition of Parent to Parent Inheritance Conflicts (ppic):**

The number of properties defined in multiple parents of the object which have the same name.

Object polymorphism is a measure of the amount of polymorphism that has been used within the object. This results in the following definition.

**Definition of Object Polymorphism (op):**

The sum of $mp$ and $vp$ for the methods and variables in the object.

These values can be calculated from the VPM and MPM by adding the column totals from each matrix for the current object.

**Intra-object Complexity**

Intra-object complexity is a function of the object level characteristics presented in Table 3. The table shows for each characteristic whether the characteristic increases (+) or decreases (-) intra-object complexity. A formula to integrate these characteristics into an overall measure of intra-object complexity is a focus of future research.

Two proposed measures of intra-object complexity are average inter-method complexity and object property coupling. The average inter-method complexity is defined as the sum of the inter-method complexity of the individual methods defined within the object divided by the number of methods contained in the object. Object property coupling is defined as the relationship between the actual connections of object properties by the potential connections of these properties. The formula for object property coupling is:
where $OPC_i$, is object property coupling of object $i$, the other variables are as defined above for object $i$. This is similar to the ratio of used data bindings to potential data bindings as defined for structured systems (Hutchens and Basili, 1985). These are measures of class cohesion as described by Coad and Yourdon (1991).

**Table 3: Object Characteristic Effect on Intra-object Complexity**

<table>
<thead>
<tr>
<th>Object Characteristic</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Parameters</td>
<td>+</td>
</tr>
<tr>
<td>Parameters Returned by the Object</td>
<td>+</td>
</tr>
<tr>
<td>Parameters Returned to the Object</td>
<td>+</td>
</tr>
<tr>
<td>Parameters Passed to Other Objects</td>
<td>-</td>
</tr>
<tr>
<td>Local Methods</td>
<td>+</td>
</tr>
<tr>
<td>Referencers of Local Methods</td>
<td>+</td>
</tr>
<tr>
<td>Inherited Methods</td>
<td>-</td>
</tr>
<tr>
<td>Referencers of Inherited Methods</td>
<td>+</td>
</tr>
<tr>
<td>Local Variables</td>
<td>+</td>
</tr>
<tr>
<td>Referencers of Local Variables</td>
<td>+</td>
</tr>
<tr>
<td>Inherited Variables</td>
<td>-</td>
</tr>
<tr>
<td>Referencers of Inherited Variables</td>
<td>+</td>
</tr>
<tr>
<td>Fan-out</td>
<td>-</td>
</tr>
<tr>
<td>Fan-up</td>
<td>+</td>
</tr>
<tr>
<td>Fan-down</td>
<td>+</td>
</tr>
<tr>
<td>Object to Root Depth</td>
<td>+</td>
</tr>
<tr>
<td>Object to Leaf Depth</td>
<td>+</td>
</tr>
<tr>
<td>Local to Parent Inheritance Conflicts</td>
<td>+</td>
</tr>
<tr>
<td>Parent to Parent Inheritance Conflicts</td>
<td>+</td>
</tr>
<tr>
<td>Object Polymorphism</td>
<td>+</td>
</tr>
</tbody>
</table>

**Inter-object Complexity**

Inter-object complexity is a function of the object level characteristics presented in Table 4. The table shows for each characteristic whether the characteristic increases (+) or decreases (-) inter-object complexity. A formula to integrate these characteristics into an overall measure of inter-object complexity is a focus of future research. We expect this is another measure of interaction coupling as presented by Coad and Yourdon (1991).
Table 4: Object Characteristic Effects on Inter-object Complexity

<table>
<thead>
<tr>
<th>Object Characteristic</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Parameters</td>
<td>+</td>
</tr>
<tr>
<td>Parameters Returned by the Object</td>
<td>+</td>
</tr>
<tr>
<td>Parameters Returned to the Object</td>
<td>+</td>
</tr>
<tr>
<td>Parameters Passed to Other Objects</td>
<td>+</td>
</tr>
<tr>
<td>Inherited Methods</td>
<td>+</td>
</tr>
<tr>
<td>Referencers of Inherited Methods</td>
<td>+</td>
</tr>
<tr>
<td>Inherited Variables</td>
<td>+</td>
</tr>
<tr>
<td>Referencers of Inherited Variables</td>
<td>+</td>
</tr>
<tr>
<td>Fan-in</td>
<td>+</td>
</tr>
<tr>
<td>Fan-out</td>
<td>+</td>
</tr>
<tr>
<td>Fan-up</td>
<td>+</td>
</tr>
<tr>
<td>Fan-down</td>
<td>+</td>
</tr>
<tr>
<td>Object to Root Depth</td>
<td>+</td>
</tr>
<tr>
<td>Object to Leaf Depth</td>
<td>+</td>
</tr>
<tr>
<td>Local to Parent Inheritance Conflicts</td>
<td>+</td>
</tr>
<tr>
<td>Parent to Parent Inheritance Conflicts</td>
<td>+</td>
</tr>
<tr>
<td>Object Polymorphism</td>
<td>-</td>
</tr>
</tbody>
</table>

Application Level Measures for OO Systems

Characteristics defined at the application level include the number of abstract classes, the number of concrete classes, maximum depth of the object hierarchy, the maximum breadth of the object hierarchy, and the number of unique messages sent between objects.

Definition of Application Concrete Classes (acc):

The number of the instantiable classes in the application.

Definition of Application Abstract Classes (aac):

The total number of classes minus the number of the instantiable classes in the application.

Definition of Maximum Depth of the Object Hierarchy (amd):

The number of levels from the root(s) of the object hierarchy to the leaf that is the furthest away.

Definition of Maximum Breadth of the Object Hierarchy (amb):

The maximum number of classes at any one level in the object hierarchy.

The average of inter-method complexity for the individual methods defined within all objects, as described above, is a measure of application complexity. Similarly, the average of inter-object complexity...
for the individual objects defined within the application is also a measure of application complexity. Both of these are measures of interaction coupling as described by Coad and Yourdon (1991). These measures provide a basis for the comparison of different implementations of an application as well as of different applications.

Average object property coupling is another indicator of complexity at the application level. The formula for this measure is:

\[
AOPC = \frac{\sum_i^n OPC_i}{n}
\]

where \(AOPC\), is the average object property coupling the objects in the application, \(OPC_i\), the object property coupling of object i, and \(n\), is the number of objects in the application.

Finally, the sum of the total columns and rows of the OCM, OPM, and the OIM provide indications of the overall application complexity.

IV. Summary.

In this paper we propose a set of measures of the complexity of an OO system at the variable, method, object, and application levels. These measures are consistent with the characteristics of good OO design and the essential and desirable properties of software measures presented above. Based on the simplicity of calculation and comprehensiveness of these measures additional research activities are warranted.

Previously, we have examined the use of source lines of code, software science (Halstead, 1977) and cyclomatic complexity (McCabe, 1976) as indicators of OO system complexity. Currently, we are evaluating OO class libraries and working application to fully describe complexity at each level are the focus of this evaluation. Additionally, an experiment is being developed to validate these measures with the perceptions of professional OO analysts, designers and programmers. Future research includes the development of criteria to allow the tradeoff of OO system complexity between the levels and between measures within the levels. The goal of this research is to develop a theoretically based methodology to provide the minimum OO system complexity solution for a given application.
REFERENCES:


