Software Metrics and the Object-Oriented Paradigm

Wei Li, Charles Cooley, John Lewis
and Sallie Henry

TR 90-55

Software Metrics and the Object-Oriented Paradigm

by

Wei Li
Charles Cooley
John Lewis
and
Sallie Henry

Computer Science Department
Virginia Tech
Blacksburg, Virginia 24061

Internet: henry@vtopus.cs.vt.edu

Software Metrics and the Object-Oriented Paradigm

abstract

Software metrics are in use to guide current software development practices. As commercial organizations make use of the benefits of the object-oriented paradigm, the desire to apply metrics to that paradigm has logically followed. However, standard procedural metrics are limited in their ability to describe true object-oriented designs and code, and in some aspects fail outright. This paper describes the difficulties in applying standard metrics to object-oriented code and defines a set of metrics which are specifically geared toward the features which make the object-oriented approach unique.

1. Introduction

The object-oriented paradigm has established itself in mainstream computer science practice and can no longer be considered just another "buzzword" of the discipline. Many applications of the techniques promoted by an object-oriented approach are used throughout the software process life cycle in commercial systems development. Research in areas which compliment the successful application of object-oriented concepts is essential to make the most of this powerful paradigm.

The use of software complexity metrics is one area which has not been extensively explored in conjunction with the object-oriented approach. Software metrics have been successfully used in the past to predict future maintenance needs, guide testing efforts, and enhance reliability of newly developed software systems, but have been only applied to a great extent in standard procedural-based development models. Software complexity metrics evaluate source code and/or designs, establishing the level of complexity of an item on a static scale. This allows a critical evaluation of software products and identifies those areas which need attention due to excess complexity.

Many organizations use metrics as standard practice during software development, but are having difficulties in applying them to their object-oriented development. Numerous metrics are defined in the literature, but are fundamentally based in a procedural approach. Some of these metrics can be modified or extended to apply to the object-oriented paradigm, but none specifically address its particular characteristics. Specific features in an object-oriented language change the perspective of the developer, therefore the evaluating metrics must change prospective as well.

Procedural metrics can be divided into two categories: 1) code metrics, which concentrate their definitions of complexity on the internal aspects of a routine, and 2) structure metrics, which focus on system structure and communication details to determine complexity. Examples of code metrics are Lines of Code, Halstead's Software Science Indicators [HALM77], and McCabe's Cyclomatic Complexity [MCCT76]. Structure metrics include Henry and Kafura's Information Flow [HENS81] and McClure's Invocation Complexity . Hybrid metrics can also be defined which combine aspects of both code and structure evaluation.

While these metrics apply in limited scope to certain aspects of the object-oriented paradigm, none of them take into account the properties of class structure, inheritance, polymorphism, etc. As more and more organizations avail themselves to the benefits of the object-oriented approach, metric evaluation is sure to follow. This paper describes the object-oriented characteristics which make it a unique development paradigm and define some initial metrics which critically evaluate various aspects of complexity. Furthermore, metrics are defined which critique the degree to which designs and code make use of object-oriented concepts.

The next section discusses the basic principles of the object-oriented paradigm and establishes the need for additional metrics. Section 3 defines a set of metrics geared toward the object-oriented approach and discusses their applicability to various object concepts. Finally, Section 4 summarizes our current applications of object-oriented metrics and discusses future directions.

2. Principles of the Object-Oriented Paradigm

Some standard metrics are still meaningful when applied to object-oriented programs, but the inherent differences between the procedural and object-oriented paradigms require that new metrics be constructed. This section examines the features of the object-oriented approach before discussing the new metrics proposed in Section 3.

Various classification of object-oriented programming concepts have been presented in the literature, including [KORT90] [WARP87] [WEGP87]. While there are some differences, especially in terminology, the primary concepts have been consistent. The foundation of object-oriented programming is the *encapsulation* of information within objects. Traditional programs consist of passive data which is manipulated by a set of functions, which call each other and communicate by exchanging data. A *class*, in an object-oriented program, is a collection of data, *members*, and all functions, *methods*, which can access that data. An *object* is a particular instance of a class, and each object has its own set of members and methods. Objects are not passive data that can be manipulated, but are entities which are capable of performing actions. Any access to the data within an object is accomplished by sending a message to the object specifying the action to be taken by some method in that object. Thus *message passing* between objects is analogous, but conceptually different, to function calls in procedural programs.

Additionally, methods within an object may also call other methods within that same object by sending the appropriate message.

Standard code metrics can be applied to the methods of an object as they would be applied to functions. Standard structure metrics which monitor function calls and data flow can also be applied to message passing, although the information gathered takes on a slightly different meaning. Structure data is associated with the object involved not the method. This reflects the fact that all interaction is performed at the object level. The change of perspective from function to object implies a need for some unique interpretations of the results of applying standard metrics to objects.

The above discussion assumes that total encapsulation of data within the object is achieved. Some languages, such as C++, allow the programmer to access members of an object directly without message passing. In this case, a new metric must be used which measures the degree of encapsulation of a class. The lack of encapsulation is similar to the use of global variables in procedural programs.

An issue closely associated with message passing is *dynamic binding*. Many object oriented languages support runtime binding of messages being passed between objects. This dynamic feature of object-oriented languages is not measured by existing metrics. Metrics to measure the dynamic behavior of a program are still in their infancy and have not been applied to an object-oriented situation.

Perhaps the most influencial feature of the object-oriented paradigm is the concept of *inheritance*, which allows one class to be defined in terms of another. A *derived class* is a class defined in the context of some other class with possible substitutions and additions of methods and members. An object oriented program will contain at least one *class hierarchy*, consisting of a *base class* and any classes which are directly or indirectly derived from that base class.

Class hierarchies form trees, where a base class forms the root from which other classes are derived. These classes can then be used as base classes themselves. Typically, a complete object-oriented system will have an overall class hierarchy whose root is considered a generic object and all new classes are derived from it. Each derived class is a more refined version of its predecessor. Some object-oriented languages allow a class to inherit from more than one class. This *multiple inheritance* results in more complex class

hierarchies, whose form is now a directed graph and no longer a tree, but may make the representation of the problem easier.

As an example of a simple class hierarchy, assume a base class vehicle. Classes car and plane can be derived from vehicle and class jet fighter can be derived from plane. The common features of objects are represented by classes which are inherited. A jet fighter is a plane but not a car, however, all three of these are vehicles and so have some common traits. Any message which can be sent to a vehicle can also be sent to a jet fighter. This ability to define one thing in terms of something else is one of the great strengths of object-oriented programming. The logical structure of the program can in some cases be expressed directly as a class hierarchy, greatly simplifying the program in other aspects. The ability to express the relationship between objects with inheritance and the class hierarchy is unique to object-oriented programming and there are currently no metrics to measure this new interconnection structure. Existing metrics can be applied to each individual class but not to the hierarchy as a whole.

The next section discusses new metrics geared specifically toward the object-oriented paradigm.

3. Object-Oriented Metrics

Given the differences between the object-oriented and procedural paradigms, new metrics must be established to adequately reflect the object-oriented approach. This section defines such metrics and discusses how they apply to six fundamental concepts in the paradigm. The six concepts are: classes, objects, inheritance, polymorphism, message passing, and dynamic binding. The metrics are not intended to be comprehensive, but serve as a starting point for the specific evaluation of object-oriented concepts.

Throughout our discussion, we will use the object-oriented programming language called Classic Ada as a running example. Classic Ada is a superset of Ada with the object-oriented features listed above [CAUM89].

3.1 Classes

A class is the basic element in an object-oriented design. A class defines a unique interface for a set of possible objects. It defines a set of data and a set of operations to manipulate the data. The set of operations (methods) provides the only interface to the outside world to access the data. This mechanism enforces the information hiding and data abstraction principles.

In Classic-Ada, a class is refereed to as a "class object". A Classic-Ada class description consists of two parts: a class specification that describes its visible interface and a class body that describes its implementation . Since classes are the primitives in the inheritance hierarchy, it is important to know where a class stands in the inheritance hierarchy and what kind of relations it has with other classes. A Classic-Ada class specification includes the declarations of numbers, types, subtypes, subprograms, exceptions, generic instantiations, instance variables, class methods, and instance methods. Some quantitative primitive measurements are defined for classes as follows:

By Class

Inheritance Hierarchy

1. depth in the inheritance hierarchy.
2. number of children.
3. number of descendants.

Within A Class

Local Declaration

1. number of numbers declared.
2. number of types declared.
3. number of subtypes declared.
4. number of subprograms declared.
5. number of exceptions declared.
6. number of generic instantiations declared.
7. number of instance variables declared.
8. number of class methods declared.
9. number of instance methods declared.
10. for each locally declared variable:
    A. number of methods which reference it.
    B. number of methods which modify it.
11. for each method:

A. number of locally declared variables which are referenced.
B. number of locally declared variables which are modified.
C. number of inherited instance variables which are referenced.
D. number of inherited instance variables which are modified.

3.2 Objects

An object is the basic element in object-oriented programming. It is also the basic run-time entity in an object-oriented system [KORT90]. An object is an instance of a class. Each instance is a specific occurrence of a class. The objects of the same class have the same interface but different states (values of instance variables, etc). Consequently, no distinction is made between the measurement primitives for an object and for the class which defines it. Using the above class metrics should measure the objects as well.

3.3 Inheritance

Inheritance is the most distinguishing feature in the object-oriented paradigm in comparison with other paradigms. It allows for the creation of one class to be based on that of existing classes. It is also the most promising concept of constructing software systems from reusable parts [KORT90].

In Classic-Ada, inheritance is class-based simple inheritance [CAUM89]. This means the inheritance is at the class level rather that at the instance level. A class can have only zero or one superclass, i.e., Classic Ada does not support multiple inheritance. A subclass inherits all the class methods, instance methods, and instance variables declared by the super class and all its ancestors [CAUM89]. Some quantitative inheritance measurement primitives are defined as follows:

Within A Class

Inherited Properties

1. number of numbers inherited.
2. number of types inherited.
3. number of subtypes inherited.
4. number of subprograms inherited.
5. number of exceptions inherited.
6. number of generic instantiations inherited.

7. number of instance variables inherited.
8. number of class methods inherited.
9. number of instance methods inherited.
10. for each instance variable inherited:
    A. number of methods which reference it.
    B. number of methods which modify it.

3.4 Polymorphism

Polymorphism, in general, means the ability to take more than one form. In an object-oriented language, a polymorphic reference is one that over time, refers to instances of more than one class [KORT90]. If the two classes are not on the same inheritance path (no ancestor-descendant relation), then the polymorphism is a compiler implementation issue which is transparent to users. However, if one of the classes is the subclass or the descendant class of the other one, then this means that the method in the subclass or the descendant class is overriding the inherited method with the same name in the superclass or the ancestor class. In this case, the polymorphism is visible to the designer, programmer, and maintainer. Some primitive measures of polymorphism in the second case are defined as follows:

Within A Class

Polymorphism

1. number of overriding methods.
2. for each overriding method:
    A. number of locally inherited variables which are referenced
    B. number of locally inherited variables which are modified
    C. number of inherited instance variables which are referenced.
    D. number of inherited instance variables which are modified.

3.5 Message Passing

In the procedural paradigm, a task is accomplished by calls to procedures and functions. In the object-oriented paradigm, a task is accomplished by sending messages to objects. The message passing can be measured at two levels: method level and object level. Some primitive measures of the message passing are defined as follows:

By Class

Message Passing

1. for each method declared within the object:
   A. number of messages passed in from within the same object (self).
   B. number of messages passed in from outside the object.
   C. number of messages passed out from the object.
2. for each object:
   A. number of messages passed in to the object.
   B. number of messages passed out from the object.

3.6 Dynamic Binding

In the object-oriented paradigm, dynamic binding means the binding of a message to a method in a specific instance of a class is determined at the run-time from the dynamic type of the reference [KORT90]. It is often associated with polymorphism and inheritance. The dynamic binding concept is a compiler implementation issue which is transparent to users. Since dynamic binding requires dynamic metrics and we are only dealing with static metrics, measurement of this characteristic is not considered.

4. Conclusion

We are currently in the implementation phases of the object-oriented metric analyzer. Hopefully versions for Classic Ada and C++ will be finished by the end of the academic year.

Plans for the Classic Ada tool include measuring object oriented constructs during design, to give testing guidelines and to predict maintenance activities. We will integrate this tool in an existing development environment to validate our goals.

The C++ effort involves metric measurement of a reusability experiment. We will try to answer such questions as "Does object oriented software promote reusability?" and "Do less complex software components increase or reduce reusability?" These object oriented metrics may give us an indication of the reusability characteristics of each software component.

References

[CAUM89] Software Productivity Solutions, Inc., "Classic-Ada User's Manual", 1989.

[HALM77] Halstead, M. Elements of Software Science, Elsevier North Holland, Inc., New York, NY.

[HENS81] Henry, S. M. and Kafura, D. G. . "Software Structure Metrics Based on Information Flow," IEEE Transactions on Software Engineering, September 1981, pp. 510-518.

[MCCT76] McCabe, T. "A Complexity Measure," IEEE Transactions on Software Engineering, December 1976, pp. 308-320.

[KORT90] Korson, Tim, and John D. McGregor, "Understanding Object-Oriented: A unifying Paradigm," Communication of the ACM, September 1990, Vol. 33, No. 9, pp41-60.

[WARP87] Ward, Paul T., "How to Integrate Object Orientation with Structured Analysis and Design,", Proceedings: OOPLSA '87, October, 1987.

[WEGP87] Wegner, Peter, "Dimensions of Object-Based Language Design," Proceedings: OOPLSA '87, October, 1987.