Applying Object-Oriented Metrics to Ada 95

William W. Pritchett IV
CACI, Inc.-FEDERAL
3930 Pender Drive
Fairfax, Va 22030
bpritchett@std.caci.com

"I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, your knowledge is of a meagre and unsatisfactory kind"

Lord Kelvin

Abstract

Ada 95 adds many new and notable features to the Ada 83 standard. The additions include such aspects as object-oriented programming, hierarchical libraries, and protected objects. The enhancements to the language may have a profound impact on the way developers design software in Ada. Consequently, the way in which the new software designs are assessed needs to be addressed. Recent studies suggest traditional functionally-oriented metrics are not applicable to object-oriented software. As a result, new measures are being proposed that may be applicable to object-oriented design. Some of these metrics have been validated on small to medium sized projects written in C++ and Smalltalk. This paper demonstrates how to apply these metrics to Ada 95.

INTRODUCTION

In February 1995, Ada 95, the revision to the Ada programming language, became the first internationally standardized object-oriented programming language. The new standard, officially ISO/IEC 8652:1995 adds many new and important features to the language [ISO 95]. The most notable of the new features include support for objectoriented programming, hierarchical library units, and protected objects.

Support for object-oriented programming is probably the single most important feature added to Ada. Ada 95 includes object-oriented facilities giving it the ability to implement programming by extension (inheritance) and dynamic binding (polymorphism).

The insertion of any new technology into a software process has an impact not only on the process, but on the products produced. This presents a challenge to organizations using established metrics to monitor, control, and improve the way they develop and maintain software [Basili 95]. New metrics may be needed i11 order to effectively assess the specifics of the new technology. These metrics must be empirically validated in order *to* be used credibly in practice.

SOFTWARE METRICS

Software metrics are necessary for any organization serious about assessing and improving its development process as well as the quality of its products. There are two fundamental reasons to measure software: to *predict* and to *assess.* Managers need to predict how long a project will take or how much money the project will require. Developers need to assess the "ilities" of the system such as reliability, maintainability, reusability, etc. These attributes cannot be evaluated without first being measured.

A measure, in general, is the assignment of a number to an entity for the purpose of characterizing a specific attribute of the entity. In software, that are three categories of entities in which all measurable attributes fall: processes, products, and resources. The measures described in this paper fall into the category of product metrics, that is, metric,, that measure an attribute of a specific software artifact like a design or code.

Before moving on, a general word of caution regarding metrics is in order. While developing software, it is very easy to "go overboard" with software measures. Many software attributes are easy to measure and readily obtainable. The problem is, once collected, what next? Metrics are only useful if they serve a specific goal such as improving the quality of your software or reducing the time it takes to deliver a product. Metrics collected for metrics' sake serves no useful purpose. Thorough descriptions of how metrics should be derived, collected, and validated have been done by Basili et. al. [Basili n.d, 95][Briand 94] and Fenton[Fenton 94a, 94b].

OBJECT-ORIENTED METRICS

The topic of object-oriented metrics is relatively new. Although many metrics have been proposed, few have been based on sound measurement theory or, further, have beer empirically validated. One of the first attempts to do this was by Chidamber and Kernerer (C&K). They have proposed six new 00 metrics based largely on theoretical concepts Chid 94]. These metrics are:

- weighted methods per class
- depth of inheritance
- number of children
- coupling between object classes
- response for a class
- lack of cohesion between methods.

Their metrics have been criticized, however, for being too ambiguous for practical applications and for not being language independent [Churcher 95].

Basili, Briand and Melo have assessed the C&K metrics in the context of being predictors of fault-prone classes (classes more likey to have problems) in medium-sized C.++ applications [Basili 95]. They discovered that the majority of metrics under study can be used to predict fault-prone C++ classes. This study, although performed following rigorous software experimentation principles, used the classroom setting as the environment. This approach has been criticized for not being scaleable to "real world" situations since most students are not software professionals and most of the applications are too small to be of any interest. This study does, however, provide the basis for further research in a more industrial setting.

Li, et. al. have also empirically evaluated C&K's metrics on C++ projects as being predictors of maintenance effort [Li 95]. In addition, Li, et. al. proposed new metrics that were used in their study including:

- message passing coupling,
- data abstraction coupling, and
- number of methods.

where message passing coupling and data abstraction coupling refine C&K's coupling between objects metric. They found a significant correlation between many of the metrics and the number of lines changed per class during maintenance.

The metrics proposed by C&K and Li, et. al. seem to offer the greatest potential as being valid metrics for object-oriented design. The metrics have been properly derived and arse well on their way to being empirically validated. Most of their metrics are also applicable to Ada 95 software developed using an object-oriented methodology. To summarize, the relevant metrics are:

- weighted methods per class,
- depth of inheritance tree,
- number of children,
- response for a class,
- message passing coupling,
- data abstraction coupling, and
- number of methods.

The definition of each of the proposed metrics, with examples in Ada 95, are given below.

Metrics Definition

To better define and demonstrate how these metrics are calculated using Ada 95, an example is used. The three listings of source code: Location, Point, and Circle are Ada 95 versions of the classes used by Li, et. al. [Li 95] to demonstrate the metrics using C++.

Note that the majority of the 00 metrics presented are based on the notion of a class. While there is no single construct for a class in Ada 95, Ada 95 packages can be constructed to provide the analogous behavior as shown in the code below:

```
package Class is type Instance' is tagged private; -- Type for instance
  variables -- of the class -- public methods procedure P1(The Class : in
  Instance);
private
  type Instance is tagged
  record
    Attribute' : This_Type;
  Attribute2 : That_Type; end
record; end Class;
```

All mention of the term class will refer to this type of package organization. Also note that the names of the metrics for Ada 95 may differ from the names originally proposed to accommodate accepted Ada terminology.

Listing 1 - Class Location

```
--Class Location
----------------------------------------------------------------------
package Location is type Instance is
  tagged private;
  procedure Initialize(The_Location : in out Instance;
                          Init X : in                  Integer;
        Init Y                   : in                                    Integer);
   function X Coordinate of(The_Location : in              Instance) return Integer;
   function Y_Coordinate_of(The_Location : in              Instance) return Integer;
private


  type Instance is tagged
  record
    X : Integer := 0;
  Y : Integer := 0; end
record; end Location;


--I Body of Location
package body Location is


  procedure Initialize(The_Location : in out Instance;
        Init_X                                              : inInteger;
        Init_Y                   : in                    Integer) is
  begin
  The Location.X :- Init_X;
  The_Location.Y := Init_Y;
  end Initialize;


  function X Coordinate of(The Location : in Instance) return Integer is
```

' **By convention, the term Instance will be used to denote all instance variables of a class. This was proposed by Rosen [95], but is by no means universally excepted.**

```
  begin return
     The_Location.X; end
     X_Coordinate_ Of;
 function Y Coordinate Of(The_ Location : in Instance) return Integer is
 begin
  return The_Location.Y;
 end Y_Coordinate_ Of;
end Location;
```

Listing 2 - Class Point

```
--class Point
-------------------------------------------------------------------
package Location.Point is type Instance is new
   Location.Instance with private;
   procedure Initialize(The_Point : in out Instance;
                            Init X                        : in   Integer;
                            Init Y                         : in  Integer);
   procedure Show (The_Point :                                   in out Instance);
   procedure Hide                              (The-Point :  in out Instance);
   procedure Drag (The_Point : in out Instance;
                      By                             : in       Integer);
    function Is Visible             (The_Point : Instance) return Boolean;
   procedure Move (The_Point : in out Instance;
                      New -X                        : in        Integer;
                      New -Y                         : in       Integer);
private
   type Instance is new Location.Instance with
   record
   visible : Boolean := False; end
record; end Location.Point;
--I Body of Point
with Graphics; package body
Location.Point is
   procedure Initialize(The_Point : in out Instance;
                            Init X                        : in   Integer;
                            Init Y                            : in       Integer) is
   begin
      Initialize(Location.Instance(The-Point), Init X, Init Y);
     The_Point.Visible := False;
    end initialize;

   procedure Show (The_Point : in out Instance) is begin
     The_Point.Visible := True;
     Graphics.Put_Pixel(The_Point.X, The_Point.Y, Graphics.Current_ Color); end Show;

   procedure Hide (The_Point : in out Instance) is begin
     The_Point.Visible := False;
     Graphics.Put_Pixel(The_Point.X, The_Point.Y, Graphics.Background_Color); end Hide;
   procedure Drag (The Point : in out Instance;
```

```
--BY., in Integer) is
Delta_X, Delta _Y : Integer;
Figure X, Figure _Y : Integer;
begin
 Show(The Point);
 Figure_X := X Coordinate_Of(Location(The_Point));
 Figure_Y := Y Coordinate_Of(Location(The_Point)l;
 while Graphics.Delta Of(Delta_X, Delta Y) loop
  Figure -X :- Figure .X + (Delta _X * By);
  Figure_Y :- Figure_Y + (Delta Y * By);
  Move(The_Point, Figure X, Figure Y);
 end loop;
end Drag;
function Is Visible (The_Point : Instance) return Boolean is begin return The_Point.Visible; end Is Visible;
procedure Move (The-Point : in out Instance;
New _X : in Integer;
New_Y : in Integer) is
 begin
  Hide(The_Point);
  The_Point.X :- New X;
  The_Point.Y :- New Y;
  Show(The_Point);
 end Move;
end Location.Point;
```

Listing 3 - Class Circle

```
-- Class Circle
---------------------------------------------------------------------------
package Location.Point.Circle is
 type Instance is new Location.Point.Instance with private;
 procedure Initialize(The_Circle : in out Instance;
          Init X : in Integer;
          Init Y : in Integer;
          Init Radius: in Integer);
 procedure Show(The Circle : in out Instance);
 procedure Hide(The Circle : in out Instance);

    procedure Contract(The Circle : in out Instance;
                    By : in Integer);

    procedure Expand (The Circle : in out Instance;
                    By : in Integer);
  private

    type Instance is new Location.Point.Instance with record
    Radius : Integer; end record; end
  Location.Point.Circle; --I Body for Circle

  with Graphics; package body Location.Point.Circle is

    procedure Initialize(The_Circle : in out Instance;
          Init_X : in Integer;
          Init_Y : in Integer;
            Init Radius: in Integer) is
```

```
begin
  Initialize(Point.Instance(The_ Circle), Init X, Init Y);
  The _Circle.Radius := Init Radius;
end Initialize;
procedure Show(The_ Circle : in out Instance) is begin
  The-Circle.Visible :- True;
  Graphics.Draw Circle(The_ Circle.X, The Circle.Y, The Circle.Radius); end
Show;
procedure Hide(The_ Circle : in out Instance) is
  Temp Color : Integer;
begin
    Temp Color := Graphics.Current_Color;
    Graphics.Set Color(Graphics.Background Color);
    The-Circle.Visible := False;
    Graphics.Draw Circle(The Circle.X, The Circle.Y, The Circle.Radius);
  Graphics.Set_Color(Temp Color);
end Hide;
procedure Expand(The Circle : in out Instance;
                 By                              : in       Integer) is
begin
  Hide(The_Circle);
  The _Circle.Radius := The_Circle.Radius + By;
  if The_Circle.Radius < 0 then
    The_Circle.Radius :- 0;
  end if;
  Show(The_Circle);
end Expand;
procedure Contract(The Circle : in out Instance;
                   By                              : in    Integer) is
begin
Expand(The Circle, -(By));
end Contract;
end Location.Point.Circle;
```

Weighted Subunits per Class (WSC)

The weighted subunits per class is defined as being the sum of the "complexities" (i.e. Halstead, McCabe, etc.) of the individual subunits in a class. The assumption behind this metric is that:

1. classes with a large number of functions and procedures are more likely to be application specific, thus limiting the potential for reuse;
2. classes with a large number of functions and procedures are harder to maintain and have a greater impact on any classes that inherit from them [Chidam 94], and
3. classes with a small number of "complex" functions and procedures are equally hard to maintain.

From the example code, the WSC for *Location is* 3 since there are 3 methods, each with a McCabe complexity of 1. *Point* has 5 local methods with a complexity of 1, and one method *(Drag)*, with a McCabe complexity of 2 for a WSC of 7. The WSC for *Circle*, however, is 6. *Circle* which has 4 local subunits with a complexity of 1 and one procedure, *expand*, with a complexity of 2.

Depth of Inheritance Tree (DIT)

Inheritance has also been called programming by extension. The key idea of programming by extension is the ability to declare a new type that refines an existing type by inheriting, modifying or adding to both the existing components and the operations of the parent type [Barnes 93]. The depth of inheritance is defined to be the level of the class in the inheritance hierarchy, with the root class being zero. The rationale behind this metric is that changes in the parent classes can potentially ripple down to the child classes, thus deep class hierarchies are potentially harder to maintain. This metric may need additional refinement to take into account Ada 95's ability to hide inheritance by declaring the tagged type in the private part of the specification.

As seen in the example code, the class *Location* inherits from no other class so the DIT is *0. Point,* however, inherits from Location so DIT is 1. In turn, *Circle* inherits from Location, making it two levels deep in the inheritance hierarchy so DIT for *Circle is 2.* Note that the definition of this metric is dependent on the notion of a class as previously defined. This metric does not apply to packages with more than one tagged type.

Number of Children (NOC)

The number of children is the number of direct descendants for a class. That is, in Ada 9`>, the number of packages that extend the type of the package being measured. The rationale behind this metric is that classes with a large number of children are difficult to modify because changes to the parent can adversely impact the children. Again, the impact of this metric is subject to the type of inheritance used. Also, classes with large numbers of children are usually very general, requiring a greater number of contexts [Basili 95].

Examining the example code, *Location* has an immediate child, *Point so NOC* for *Location is* 1. The same holds for *Point* which has one child - Circle. *Circle* has no children so its NOC is 0.

Response for a Class (RFC)

The response for a class is the total of the number of functions or procedures that can potentially be executed in a class. Specifically, this is the number of operations directly invoked by member operations in a class plus the member operations themselves. The assumption is that classes with a large response set are harder to understand and are more fault-prone.

As seen in the example code, the RFC for *Location is 3* since it has 3 local methods *(Initialize, Get* X, and *Get* Y) and makes no calls to external classes. *Point* has 6 local methods, plus makes calls to *Location.Initialize, Graphics.Put Pixel*, *Graphics.Current_Color*, *Point.X_ Coordinate Of* and *Point. Y Coordinate Of* for an RFC total of *11. Circle* has 5 local methods plus calls to *Graphics.Draw_ Circle, Graphics.Current_Color*, *Graphics.Background_Color,* and *Graphics.Set_ Color* for an RFC of 9.

Message Passing Coupling (MPC)

Message passing coupling is a count of the total number of function and procedure calls made to external units (different calls to the same routine are counted separately). The assumption behind this metric is that classes interacting with many other classes are harder to understand and maintain. **Measuring the example** code reveals the a MPC for the class *Location* to be 0. There are no calls to external subprograms. The MPC for class Point is 5. Point calls the external units *Graphics.Put_Pixel* (twice), *Graphics. Current - Color,* and *Graphics.Background_Color* and *Graphics.Delta Of.* The MPC for *Circle is 6* because *Circle* makes calls to *Graphics.Set_Color* (twice), *Graphics.Background_Color, Graphics.Draw Circle* (twice), and *Graphics.CurrentColor.*

Data Abstraction Coupling (DAC)

Data abstraction coupling is a count of the total number of instances of other classes within a given class. It is a count of the number of external classes the given class uses. Again, the rationale is that classes using the services of many other classes are harder :o understand and maintain. The value of DAC for each of the example classes is 0 beca .Use neither *Location, Point* nor *Circle* declares variables whose type is an instance of another class.

Number of Subunits (NUS)

The number of subunits is the total number of functions and procedures defined for the: class. The rationale behind this metric is that classes with a large number of operations are harder to maintain and are more fault prone. Note that if the complexity for each operation is 1 then the NUS metric is the same as the WSC metric.

From the examples, the NUS for *Location is 3,* with the subunits being *Initialize, X Coordinate Of* and Y *Coordinate-Of.* The NUS for *Location is 6* with the subunits being *Initialize, Show, Hide, Drag, Is-Visible,* and *Move.* Finally, the NUS for *Circle is 5* counting the subunits *Initialize, Show, Hide, Conti-act* and *Expand.*

The values of each metric for the three example classes are summarized in the table below

Table I - Metric Values for Example

WSC	3	7	6
DIT	0	1	2
NOC	1	1	0
RFC	3	11	9
MPC	0	5	6
DAC	0	0	0
NUS	3	6	5

SUMMARY AND FUTURE RESEARCH

This paper identified several metrics which *may* be used to predict fault-prone classes in Ada 95 as well as predict various aspects of maintainability for these classes. While some of these metrics have been validated using languages such as C++ and Smalltalk, *none of these metrics have been proven for Ada* 95. Individual organizations need to experiment on their own to determine whether these metrics are applicable. Additional research needs to be done to validate these metrics using rigorous experimental procedures in a controlled setting. The use of industrial case studies can also be substituted for an experiment.

The metrics presented in this paper are, by no means, a complete set of object-oriented metrics for Ada 95. Ada 95 has several language constructs that are not present in other 00 languages, making it necessary to refine the current metrics and define additional metrics. This area is reserved for future research.

ACKNOWLEDGMENTS

I would like to thank my colleagues at CACI for providing comments, encouragement and inspiration.

REFERENCES

[Barnes 93] John Barnes, "Introducing Ada 9X," *Ada Letters,* Volume XIII, No. 6, November/December 1993, pp. 61-132.

[Basili n.d.] Victor R. Basili, Gianlugi Caldiera, and H. Dieter Rombach, "The Goal Question Metric Approach," Technical Report, University of Maryland, No Date.

[Basili 95] Victor R. Basili, Lionel Briand, and Walcelio L. Melo, A *Validation of Object-Oriented Design Metrics,* CS-TR-3443, University of Maryland,

[Briand 94] Lionel Briand, Sandro Morasca, and Victor R. Basili. *A Goal-Driven Definition Process for- Product Metrics Based on Properties.* Institute for Advanced Computer Studies, Dept. of Computer Science, Univ. of Maryland, September 1994.

[Chidam 94] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Transactions on Software Engineering,* Volume 20, Number 6, June 1994, pp. 476-493.

[Churcher 95] Neville I. Churcher and Martin J. Shepperd, "Comments on `A Metrics Suite for Object-Oriented Design,'" *IEEE Transactions On Software Engineering, Vol.* 21, No. 3, March 1995, pp. 263-265.

[Fenton 94a] Norman E. Fenton, *Software Metrics - A Rigorous Approach*, Chapmaa & Hall, London, 1991.

[Fenton 94b] Norman E. Fenton, "Software Measurement: A Necessary Basis," IEEE *Transactions of Software Engineering, Vol.* 20, No. 3, March 1994.

[Li 95] Wei Li, Sallie Henry, Dennis Kafura, and Robert Schulman, "Measuring Object-Oriented Design," *Journal of Object-Oriented Programming* Volume 8, No. 4, July/August 1995.

[ISO 95] International Standards Organization. Reference Manual for the Ada Programming Language, ISO/8652-1995, 1995.

[Rosen 95] J. P. Rosen, "A Naming Convention for Classes in Ada 95," *Ada Letters,* Volume XV, Number 2, March/April 1995, pp. 54-58.