

# Measuring Object Oriented Software with Predictive Object Points

Arlene F. Minkiewicz  
Staff Operations Research Engineer  
PRICE Systems, L.L.C.  
609-866-6576  
[Arlene.minkiewicz@PRICESystems.com](mailto:Arlene.minkiewicz@PRICESystems.com)

## Abstract

Recognizing that traditional software metrics are inadequate for object oriented software when used for productivity tracking and effort prediction, PRICE Systems has developed a new metric, Predictive Object Points. Predictive Object Points were designed specifically for Object Oriented software and result from measurement of the object-oriented properties of the system. This paper discusses the problems with traditional metrics, the evolution of Predictive Object Points, with the theory and data behind it, and an example measuring Predictive Object Points.

## **Introduction**

In recent years Object Oriented (OO) technologies have emerged as a dominant software engineering practice. As happens with many new technologies, the growth of OO practices has required software developers and their managers to rethink the way they have been estimating the size of their development projects. Traditional software measurement techniques have proven unsatisfactory for measuring productivity and predicting effort. The Source Lines of Code (SLOC) metric and the Function Point metric were both conceived in an era when programming required dividing the solution space into data and procedures. This notion conflicts with the object-oriented paradigm. Traditional design techniques separate data and procedures while object-oriented designs combine them. There are multiple dimensions that an OO metric must have if it is to provide accurate effort prediction or productivity tracking. It is important to measure the amount of raw functionality the software delivers, but it is equally important to include information about communication between objects and reuse through inheritance in the 'size' as well.

This paper discusses Predictive Object Points (POPs), a metric which incorporates the three dimensions of OO software mentioned above. Unlike traditional metrics, which are based on the data and procedure model of structured analysis, POPs are based on the objects and their characteristics. The POPs metric combines several measures popular in the literature to establish a metric suitable for predicting effort and tracking productivity. The metric at the heart of the POPs calculation is Weighted Methods per Class (WMC). This metric examines each top-level class (or each distinct object from the user's perspective) and assigns a weight to the behaviors (methods) of that class. Once a value for WMC has been calculated, the POPs counter combines this with information about the groupings of objects into classes and the relationships between these classes of objects to assign the POPs count. Following a discussion of the evolution of the POPs metric and the results of correlation studies with effort, an application of Predictive Object Points will be presented. Using an OO project from a public domain source, the reader will be guided through the mechanics of counting POPs.

## **What's wrong with traditional metrics?**

Traditional metrics still hold a place in any measurement program. For years software developers and estimators have been using Source Line of Code (SLOC) and/or Function Points to estimate the

effort required to build object-oriented software projects. This is done not because the chosen metric is the best but because it's the best available. This is often adequate while the organization's OO processes are immature. Ironically, as organizations become better at designing and developing object-oriented software, these traditional metrics get less and less useful because each individual line of code becomes less likely to be like the others. The reason for this becomes obvious if you stop to think how a line of code becomes a line of code, following the path from structured analysis to the object-oriented approach.

Think first about the traditional programming approach and how it evolved. The earliest software developers were forced to decompose their software solutions completely because the earliest programming languages required a painstaking step by step description of the actions required to implement each solution. As programming evolved, more and more steps were combined into a single compiler command, yet the same functional decomposition took place. Structured analysis requires the developer to think and program sequentially, making each line of code, each function and each subsystem a product a singular thinking. In the context of this singular thinking about lines of code and functions it is easy to see the appeal of measures like SLOC and Function Points.

Object-oriented techniques require a complete shift in thinking. A solution is not decomposed based on the steps required to implement it. Rather, it is decomposed into the actors involved in the solution and the actions they must take to effect that solution. The solution space is divided into actors and their behaviors. Since actors and their behaviors can be influenced from within or through their surroundings, the thinking that goes into each line of code requires understanding the many possible paths that lead to it. A measurement based on the sequential model of program development will miss the complexity introduced by these interactions.

## **Related Research**

The study of object-oriented metrics is not, by any means, uncharted territory. Chidamber et al. present a suite of object-oriented metrics, including some which measure 'size', which are well founded in measurement theory[4]. Lorenz published a table of 11 'design metrics' which include some well suited to size [16]. Other authors have contributed collections of metrics which measure the various dimensions of the software development effort. Henderson-Sellers [2] contains a comprehensive list of these metrics. None of these measures alone qualifies as a good size predictor because each addresses only one dimension of object-oriented software size. Whitmire [19] proposed 3-D Function Points, an extension of function points that

measure data, control and functionality of a software project. While 3D Function Points address many of the problems encountered using traditional Function Points and they can measure all three dimensions, they only accomplish this at a class level. This makes 3D Function Points a good metric for productivity analysis of completed software but less valuable for effort prediction.

Table 1 lists some of the more popular metrics suitable for object-oriented software and the characteristic(s) of a system that they are intended to measure.

<b>Metric</b>	<b>Suitable for Measuring</b>
Number of use cases/scenario scripts	Size
Weighted Methods per Class (WMC)	Size,Complexity
Methods per Class	Size
Number of Children (NOC)	Size,Complexity
Depth in Inheritance Tree (DIT)	Size,Complexity
Method Size(LOC)	Size
3D Function Points	Size,Complexity
Coupling Between Objects (CBO)	Coupling
Number of Instance Variables per class	Complexity
Number of unique messages sent	Coupling/Complexity
Number of classes inherited(derived classes)	Complexity
Number of classes inherited from (base classes)	Complexity
Reuse Ration	Quality,Complexity

*Table 1 – Some popular OO Metrics*

## **The Tree Dimensions of Object-Oriented Software**

The problem with traditional measures, as they apply to OO solutions, is that they tend to measure one dimension of the software, the functionality. Without a measure of the complexity of object-object communications and the amount of reuse through inheritance, the functionality metric ignores crucial aspects of the software size. The functionality (behavior of the objects) is an important piece of information when you want to predict effort, but to consider just this aspect could prove a mistake, particularly in a well-designed OO solution. In addition to functionality, there is also a level of complexity added to the software that depends on the amount of communication between the objects in the system. This complexity can add substantially to the

size of the project. Increased inter-object communication requires more intensive design and test of the objects that are providing increased services to the system.

The third dimension of object-oriented size is reuse through inheritance. Part of a good object-oriented analysis involves identifying groups of objects (actors) whose behaviors are similar enough to warrant them belonging to the same class or the same family of classes. A class is a description of the properties and behaviors that a particular object will have when instantiated. A group of objects with many similar behaviors is often designed as a base class (or parent class) containing the common behaviors with several derived classes (or children classes) which inherit behavior from the parent, add new behaviors that the parent does not provide and override those behaviors require different functionality from the parents. Inheritance is a powerful feature of an object-oriented software system and has the potential for a substantial reduction of effort in certain software projects.

### **Measuring All Three Dimensions**

Having established that traditional metrics lack two of the three dimensions important to an accurate assessment of software size, the next step is to identify a metric or set of metrics that will incorporate all three dimensions. Predictive Object Points, the subject of this paper, combine several of the metrics presented in the current literature to measure OO software. They are all metrics focused on an object-oriented analysis and design of a software system. These metrics measure the important OO aspects of that project – the classes developed, the behaviors of these classes and the effects of these behaviors on the rest of the system. They also incorporate measures of the breadth and depth of the intended class structure. The metrics included in a POPs count are :

- Number of Top Level Classes (TLC)
- Average Number of Weighted Methods per Class (WMC)
- Average Depth of Inheritance Tree (DIT)
- Average Number of Children per Base Class(NOC)

What follows is a description of each of these metrics along with an explanation as to why it is important to the POPs metric

**Number of Top Level Classes (TLC)** This metric is a count of the classes that are roots in the class diagram, from which all other classes are derived. Because the POPs count is intended for predictive purposes, it is

important to be able to analyze the system with top level information. The number of top level classes, along with the NOC and DIT provide a basis for an estimate of the breadth and depth of the OO system being described with this count. This adds the reuse through inheritance dimension to the POPs count.

**Weighted Methods Per Class (WMC)** This metric is an average of the number of methods per class, where each method is weighted by a complexity based on the type of method, the number of properties the method effects and the number of services this method provides to the system. The details of this weighting will be covered in more detail in later sections. This metric is the heart of the POPs count. Research indicates there are two prominent schools of thought in the determination of object-oriented metrics suitable for size estimation (remember this is size as it relates to effort and productivity). One uses a count of the total number of distinct objects [10], [11]. The other uses a count of the Weighted Methods Per Class of objects [4], [12], [9], [16]. While the number of objects has shown promise as a useful effort estimator, we favor using a WMC count for several reasons:

- Methods relate to behavior and in so doing provide a metric that has meaning to non-software savvy individuals
- Intended behaviors of the system are known early in the analysis, making it easier to develop a credible estimate early in the software lifecycle
- WMC counting methods can be established to impose some rigor on the counting process.

Weighted methods per class encompass both the functionality and the inter-object communication in the POPs count.

**Average Depth of Class in Hierarchy Tree (DIT)** Each class described can be characterized as either a base class or a derived class. Those classes that are derived classes, fall somewhere in the class hierarchy other than the root. The DIT for a class indicates it's depth in the inheritance tree i.e. it is the length (in number of levels) from the root of the tree to that particular class. For example, in Figure 2, the DIT for Class C is 3 because there are three levels between the root, A, and class C. The average DIT, along with TLC and NOC, is used to help establish the reuse through inheritance dimension and the overall system size.

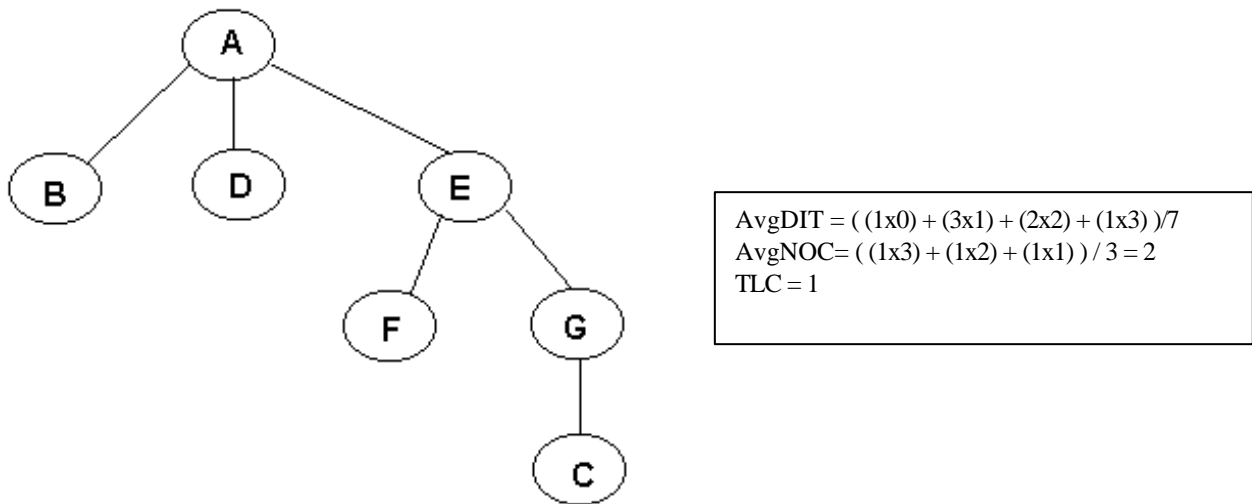


Figure 2 – Sample Inheritance Tree

**Average Number of Children per Class (NOC)** Each class has zero or more direct descendants (derived classes). The NOC is a count of these derived classes. In Figure 2, the NOC for class A is 3 because it has three children – B, D, and E. The average of the NOC across the parent classes is used to help determine the effects of reuse through inheritance and overall system size.

### The Evolution of the POP metric

The next step in the process was to collect data. We started with a data set consisting of over 20 projects. Data was collected from various software development venues including military, financial and commercial software tool vendors. The data was normalized across common application types and operating specifications. Although not all of the data sets were as complete as we would have liked, we were able to study in some detail over 775 classes and 5900 methods. Most of the software was written in either C++ or Smalltalk.

Armed with some data and an idea of the right metrics to collect to cover all three dimensions, the task was to determine the best way to combine these metrics into a quantity that correlates meaningfully with effort. The first

step in this process was the weighting of methods. The challenge in this case was to provide a framework for classifying methods that will work early in the analysis the software solution yet provide enough granularity to be meaningful. Booch [1] suggest that method can be divided into 5 classifications:

- Constructors – methods which instantiate an object.
- Destructors – methods which destroy an object.
- Modifiers – methods which change the state of an object. A modifier will contain references to one or more of the properties of it's own class or another class.
- Selectors – methods that access the state of an object but make no changes to this state. These are the methods used to provide public access to the data that is encapsulated by the object.
- Iterators – methods which access all parts of an object in a well-defined order. These may be used to visit each member in a collection of objects, performing the same operation on each member.

In order to verify that these categorizations offer substantial differences in complexity and to determine weightings based on these differences, we examined hundreds of C++ and Smalltalk methods. For each method we collected information about the type and number of properties. We organized the methods by type, assigning a weight to each method based on the amount of effort associated with that particular method. Because effort was tracked at the class level not at the method level, we relied on information from the submitters to determine by percent how much of the total class effort could be assigned to a particular method. In the absence of reliable information from the submitter we used source lines of code to apportion effort across the methods. This use of source lines of code does not violate the original hypothesis because what we were trying to accomplish with this part of the project was relative weightings of particular method types. We averaged the numbers for each method type and used these as the original weightings. During this analysis we determined that Constructors and Destructors did not differ significantly in complexity so these two method types were combined.

At this point we had determined average weightings for each type of method. These weightings were based primarily on functionality. The next step was to determine what the discriminators were between those methods

that were above average and those that were below average. Review of the data led to the conclusion that the amount of responsibility to the system (number of messages the method was expected to respond to) and the number of properties that were affected by the method were higher in methods that had above average weighting and lower in methods with below average weighting. It is important to note here that the amount of our data samples that provided enough detail for this analysis was not enough for this result to be based on pure analysis. There was some expert knowledge applied as well.

The results of this study led to the development of a process for determining weighted methods per class that includes an analysis of the functionality of the method along with an indication of low, average or high complexity based on the inter-object communication. Table 2 shows the method complexities by method type and Table 3 shows the counting rules (based on a count of messages responded to and properties effected) for determining the ranking of a method within the classifications.

Method Type	Method Complexity	Weight
Destructors/Constructors	Low	1
	Average	4
	High	7
Selectors	Low	1
	Average	5
	High	10
Modifiers	Low	12
	Average	16
	High	20
Iterators	Low	3
	Average	9
	High	15

*Table 2 – Method Weightings by type and complexity*

Message Responses	Number of Properties		
	0->1	2->6	7 or more
0->1	low	Low	avg
2->3	avg	Avg	avg
4 or more	avg	High	high

*Table 3 – Complexity assignments*

Once the values for the constants were determined, we used these values to calculate Weighted Methods Per Class (WMC). The values for Weighted Methods per class were combined with counts of the top level classes (TLC), average number of children (NOC), and average depth in inheritance tree (DIT). We performed regression on this data using various forms and settled on an equation with the following form. In this equation, f1 attempts to size the overall system while f2 applies the effects of reused through inheritance.

$$POPs(WMC, NOC, DIT, TLC) = \frac{WMC * f1(TLC, NOC, DIT)}{7.8} * f2(NOC, DIT)$$

Since we started with a rather small set of data points, we used a jack-knifed approach to the data analysis. We would hold out a small percent the data points for verification, perform regression with the rest, and then check the results against the samples held out. We did this repeatedly, holding out different samples until we found the set of equations that gave the best coefficient of variation. The tool we used to do this regression was SigmaPlot from Jandel. Our data indicates good correlation with Coefficients of Variation ranging from 5 to 19 percent across the samples we used. We recognize these results as preliminary given the amount of data studied to date. The research effort is on going and requires collection of data from other types of software applications using other development languages.

### ***A Sample POPs Calculation***

Armed with an understanding of what POPs are and how they came to be, the obvious next question is how can they be used as an estimating tool. Some of the information required to do a POPs count is information that is not fully known during early analysis. There are however, ways that we can use what is available to make reasonable estimates of the parameters required to calculate POPs. The following example was take from Lorenz [17]. Using the project information provided and the outputs from an OO metrics tool we demonstrate how to perform a count of POPs. The software developed is an OO Case Tool. It was developed using Smalltalk and has 36 classes and 1040 methods. The software development team had higher than average productivity and lots of experience developing software.

The first step is to Calculate Weighted methods per class. We know that on average there are 29 methods per class (1040/36). Since we do not know a spread across method types, we use percentages established during our study of methods as follows:

$$\text{Constructors/Destructors}(20\%) = 6$$

$$\text{Selectors}(30\%) = 9$$

$$\text{Modifiers}(45\%) = 13$$

$$\text{Iterators}(5\%) = 1$$

We use the charts in Figure 3 to determine method complexity. We are using number of messages sent and number of instance variables per class since this is the information available. From this we can approximate instance variables effects and messages received.

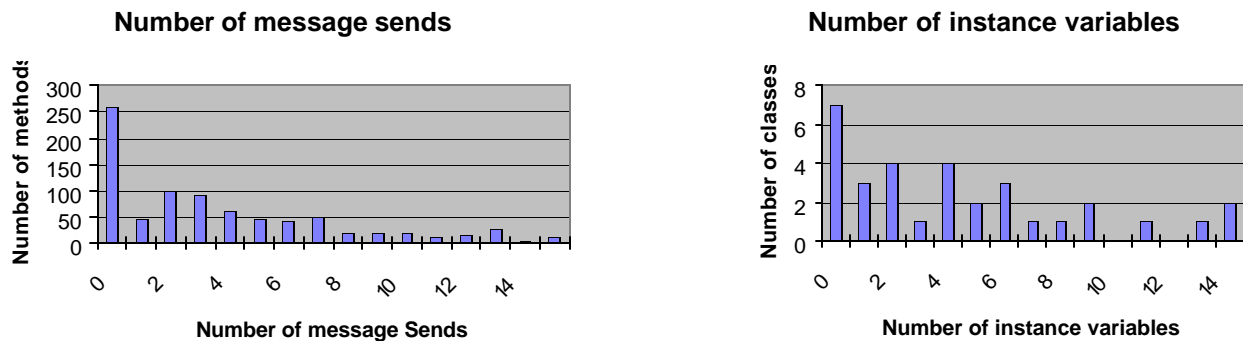


Figure 3 – Data used to determine method complexity

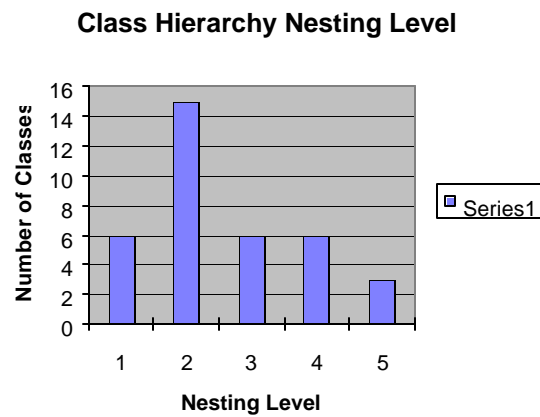
We determine complexities by counting messages and instance variables and mapping them into the complexity assignment table (Table 3). Figure 4 shows this analysis along with the resulting number of low, average and high complexity methods.

Messages sent (percent of total)	Properties effected (percent of total)		
	0->1 (31%)	2->6 (44%)	7+ (25%)
0->1 (29%)	9%	13%	7%
2->3 (23%)	7%	10%	6%

22% Low

*Figure 4 – Complexity assignments for example*

Finally we use the chart in Figure 5 to determine the average NOC and DIT counts for this example. From this we calculate that the Average NOC is 1.4 by taking the total number of children and dividing them by the number of classes that have children (30/21). We calculate the average DIT to be 1.6 The Number of Top Level Classes is 6 – the number of classes at level 0.



*Figure 5 – Class Hierarchy information*

Finally we take all of the inputs calculated and perform a POPs calculation as shown in Figure 6

**Predictive Object Point Builder**

Calculate Weighted Methods per Class (WMC)  
 Enter Weighted Methods per Class

Calculate Weighted Methods per Class (WMC)

Method Type	Method Complexity	Weighted Methods	WMC Totals
Constructor/ Destructor	1 Low x 1 =	1	= 27
	3 Avg x 4 =	12	
	2 High x 7 =	14	
Selector	2 Low x 1 =	2	= 52
	4 Avg x 5 =	20	
	3 High x 10 =	30	
Modifier	3 Low x 12 =	36	= 212
	6 Avg x 16 =	96	
	4 High x 20 =	80	
Iterators	0 Low x 3 =	0	= 9
	1 Avg x 9 =	9	
	0 High x 15 =	0	
Total Weighted Methods per Class =			300

Enter Weighted Methods per Class (WMC)

Average Weighted Methods per Class: 0

Class Data

Number of top level classes: 6

Avg. Depth (DIT): 1.6

Avg. Nbr of Children (NOC): 1.4

Predictive Object Points: 1,356

OK

Cancel

Figure 6 – Sample POPs calculation

Although this example uses a completed project, a similar analysis could be done using the artifacts from an early analysis. At the point when a use case analysis is done, the actors, their behaviors and a preliminary class structure should be available and from this other parameters can be approximated.

## The Future of POPs

Research to date has resulting in a metric and a counting process that, for the data studied, offers a correlation between object-oriented metrics and effort. There are still hurdles to be overcome. First, the data studied has been, by no means, exhaustive in coverage of types of software or implementation techniques. We need to collect additional data and continue to refine the original results. We also need to address the gap between artifacts available during early analysis and the metrics required for a POPs count. The next step is to begin looking for a direct relationship between use case artifacts and the POPs

metrics in order to simplify the estimation process. Finally, it is very important that we automate the procedure for counting POPs once a software project is complete. One of the biggest hurdles to the acceptance of any new size metric is whether automation is available to allow for calibration with an organization's existing metrics. This type of automation would ease the introduction of POPs into an environment that currently has historical data in terms of some more traditional metrics.

## References

- 1.Booch, G. 1994, *Object Oriented Analysis with Applications - 2<sup>nd</sup> Edition*, Benjamin/Cummings Publishing Co. Inc., Redwood City, CA.
- 2.Henderson-Sellers, B.,1996, *Object Oriented Metrics -Measures of Complexity*, Prentice Hall, Upper Saddle River, NJ.
- 3.Albrecht, A & Gaffney, J., 1983, Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation, *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 6,pp 639-648, November
- 4.Chidamber, S. R. & Kemerer C. F., 1994, A Metrics Suite for Object Oriented Design, *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp476-493, June
- 5.Churcher, N.I. & Shepperd, 1995, M. J., Comments on "A Metric Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, Vol. 21, No. 3, pp. 263-264, March
- 6.Banker, R.D, et. al. 1992, An Empirical Test of Object-based Output Measurement Metrics in a Computer Aided Software Engineering (CASE) Environment, *Journal of Management Information Systems*, Vol. 8, No. 3, pp. 127-150, Winter.
- 7.Boehm, B. et. al.,1995, Cost Models for Future Software Life Cycle Processes : COCOMO 2.0, *Annals of Software Engineering, Special Volume on Software Process and Product Measurement*.
- 8.Banker, R. D. et. al., 1994, Automating Output Size and Reuse Metrics in a Repository-Based Computer Aided Software Engineering (CASE) Environment, *IEEE Transactions on Software Engineering*, Vol. 20, No. 3, pp169-186, March.
- 9.Pittman, M., 1993, Lessons Learned in Managing Object-Oriented Development, *IEEE Software*, January
- 10.Laranjeira, L., 1990, Software Size Estimation of Object-Oriented Systems, *IEEE Transactions on Software Engineering*, Vol. 16, No. 5, pp510 - 522, May
- 11.Jenson, R. L. & Bartley, J. W., 1991, Parametric Estimation of Programming Effort: An Object-Oriented Model, *Journal of Systems and Software*, Vol. 15, pp. 107-114
- 12.Lockheed Martin, Advanced Concept Center training materials, 1994, Object Oriented Size and Cost Estimation.
- 13.Basili, V., 1980, Qualitative software complexity models: a summary, *Tutorial on Models and Methods for Software Management and Engineering*, IEEE Computer Society Press, Los Alamos, CA.
- 14.Weyuker, E., 1988, Evaluating software complexity measures, *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, September
15. McCabe T.J., 1976, A complexity measure, *IEEE Transactions on Software Engineering*, Vol. No. 4, April
- 16.Minkiewicz,A.F.,1997, 'Objective Measures', *Software Development*, June 1997, pp43-47.
17. Lorenz , M. 1993, *Object-Oriented Software Development: A Practical Guide*, Prentice Hall, Englewood, NJ. p227.
18. Jacobson, I., et al, 1992, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading MA

19. Whitmire, Scott, 1996, 3D Function Points: Applications for Object-Oriented Software, ASM '96 Conference Proceedings, San Diego CA