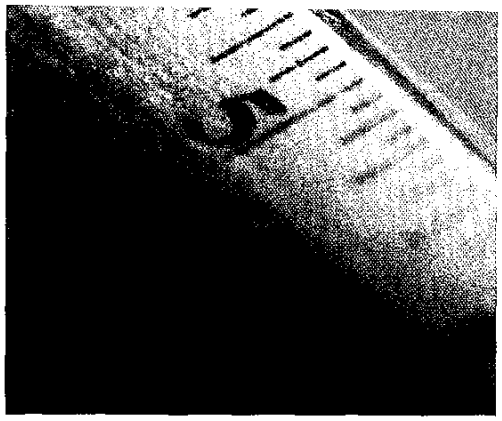
Metrics can help quantify your previous work in a way that can directly guide future efforts.

Mark Schroeder



A Practical Guide to Object-Oriented Metrics

Developers have a tough charter: Given the limited amount of resources and time—especially in the Internet milieu—it is absolutely critical to choose projects carefully and run them efficiently. The key to doing this is to leverage the experience you've gained on past projects. By understanding the development processes that worked best for you and using that understanding to refine your project management approach, you can go a long way toward reducing the risks your team faces.

While there are many ways you can capture your development experiences, metrics can help quantify previous work in a way that can directly guide future efforts. For example, projects of different sizes can require vastly different levels of effort, organizational structure, and management discipline. If you let experience be your guide and understand how a newly proposed system compares to projects you've already completed, you have a much better chance of finishing on time and under budget.

A wide range of metrics can aid you in managing projects, but here I focus on a particular set of product metrics that highlight and quantify a system's object-oriented (OO) properties.

I've drawn many of the results I mention here from an analysis of 18 production-level applications built in PowerBuilder—a common GUI tool used for developing client-server database applications on a variety of platforms. (The PowerBuilder metrics analyzer is available free from American Management Systems—http://www.amsinc.com.) While I've derived these results mainly from PowerBuilder applications, they should still provide practical guidance for development in most OO languages.

Inside

OO Metrics

CATEGORIES OF METRICS

To get a handle on how metrics can be useful to you and your team, you can begin by dividing the field into two broad categories: process metrics and product metrics.

Process metrics

Process metrics—also known as management metrics—relate to the process used to build a system. Developers generally use process metrics for

- helping predict the size of a final system,
- deriving the level of effort a project will require, and
- determining whether a project is on schedule.

Process metrics do not address system quality; they only assess size and status. In other words, process metrics describe your project without prescribing corrective actions. From an overall project perspective, however, process metrics are more important than product metrics.

Product metrics

Product metrics—also known as quality metrics—measure system quality. You can of course describe quality in many different ways, most

1520-9202/99/$10.00 © 1999 IEEE

popularly through the so-called "ilities." In *Object-Oriented Metrics* (Prentice Hall, 1994), Brian Henderson-Sellers describes a number of such categories: reliability, availability, maintainability, understandability, modifiability, testability, and usability.

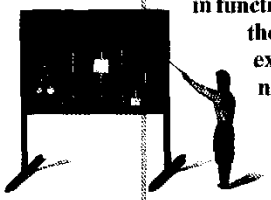
We generally use product metrics for providing

- guidelines that suggest local and specific prescriptive action for improving the quality of different system components,
- comparisons between existing systems, and
- comparisons between new systems and other known systems.

Keep in mind that quality metrics do not correlate well to a project's overall size or status measurements. System quality is a critical concern, however, and quality metrics do provide valuable insight into specific ways that enhance system quality.

Distinguishing between process and product

How do you distinguish between process and product metrics? Are defects part of the product—since that is where you find them—or are they a characteristic of the process that created them? As you might have already guessed, the distinction between the two isn't always clear. One rule of thumb is to think of product metrics as measuring the intended features or components of the system.



You can also think of both process and product metrics as either

- point values at a particular time or
- cumulative or rate measures over a span of time.

Point measures of quality metrics have several uses, which I'll describe later. *Cumulative* or *rate measures* of process metrics can provide early warning about unusual or harmful process trends.

Suppose you determine that your current development project has only 10 percent of the defects of a comparably sized system. Time to give the team a raise and pass out the champagne? Perhaps. If, however, you notice that your cumulative measure of defects over time flattened out the day you reassigned your test team, then passing out aspirin may be a better choice.

APPLYING METRICS

While there are differences between process metrics and product metrics, you use both of them, generally in conjunction, to assess the project as a whole. Specifically, metrics help you make determinations about three areas of your project's life cycle:

- cost and time frame,
- project status and effort, and
- quality.

Producing an accurate estimate for proposed work is critical to successful projects, in terms of defining realistic project costs and time frames. Indeed, accurate estimates can sometimes identify projects that are best not undertaken. Accurate estimates can also help balance quality with schedule and cost.

OO Metrics

Using product metrics to assess an OO system can help illuminate the object model's properties. But let's step back for a minute and review some object model basics.

The building blocks of an OO system are *classes*, which are groups of state and behavior. An object, or *class member*, is one instance of a class. The world outside the class cares about the class's behavior, not about how the class implements that behavior. In fact, the class hides its workings in functions or *methods* that implement the behavior with the aid of certain methods called *attributes*. Again, exactly how those attributes provide behavior is nobody's business but the class's itself.

Classes can pass on behavior to other classes through inheritance—sometimes called *specializing*. In this way a class can take on most characteristics of another class—called the *parent class*—yet make slight changes to implement a new construct. For example, a circle class may specialize a shape class by redefining what it means to draw and by adding new properties, such as radius. It may continue to use many properties and behaviors of the existing shape class, such as location, color, and so on.

Classes can also help other classes provide behavior—a process called *delegation*—which is one of the ways in which classes work together to implement a system. For example, to provide error-reporting behavior, a class may outsource the responsibility to another class. In this way, many classes can take advantage of the error-reporting behavior without building it themselves.

Given such powerful techniques, you can design systems in nearly countless ways. Some of these implementations will have desirable properties, such as easy maintainability, and some will not. The goal of OO metrics is to help guide the construction of a "good" system and to identify parts of the object model that might benefit from close examination and remedial action.

Rules of thumb, like how much of the budget remains, aren't good measures of progress. Worse, the report that development is 95 percent finished—which usually means that the easy 95 percent is finished but the hard 50 percent remains—isn't a good measure of progress either. Metrics can provide an excellent, quantified view of your current project status and a good understanding of the remaining effort you'll need to expend to complete the project. You'll need this information to identify problem areas early, which is the only time when you can easily make adjustments to scope or schedule.

You use both product and process metrics, generally in conjunction, to assess the project as a whole.

Throughout the project, quality metrics help guide the production of robust, high-quality systems. Low-quality work can sometimes lead to delays in acceptance or to expensive remedial work after implementation. Interim measures can highlight quality issues and spur corrective action before problems become significant. This is an area where product metrics provide much valuable information.

What follows are not just descriptions of metrics, but benchmark values against which you can measure your projects. Understanding what normal measures to expect helps identify when applications deviate significantly.

CATEGORIES OF OO METRICS

In addition to specifying process and product metrics, it is useful to group OO metrics into four categories:

- *System size.* Knowing, for example, how many function calls and objects to anticipate using in a system can help make more accurate estimates.
- *Class or method size.* Though measured in various ways, small, simple classes and methods are typically better than large, complex ones.
- *Coupling and inheritance.* The number and types of these relationships indicate the interdependence of classes. Clear, simple relationships are preferable to numerous, complex ones.
- *Class or method internals.* This metric reveals how complex classes and methods are and how well you've documented them in your code comments.

Unfortunately, system size metrics have no standard values against which you might compare your own system. Size depends entirely on the amount of functionality you build into your system.

Other metrics, however, do have standard values. For instance, a method's size is fairly consistent across systems. So you might want to provide some guidance to your team about how large the methods should be. You also might want to have in place an upper limit for method size, above which you would inspect methods to determine whether and how they might be shortened.

For OO metrics other than system size, it also makes sense to talk about system-level averages. For example, ask yourself whether your methods are, on average, larger than those of comparable systems. Averages provide an indication of the overall system quality and can signal trends that may affect system quality.

Because several metrics within these four categories are well known in the industry, I'm focusing here on the lesser known metrics and on any peculiarities I experienced when applying metrics to PowerBuilder.

System size

System size metrics correlate to the total effort required for system construction. In most OO systems, GUI components represent a significant amount of the total development work, so, in addition to the other size metrics, I've also noted one of the most popular ways to measure the windowing interface. Size metrics include

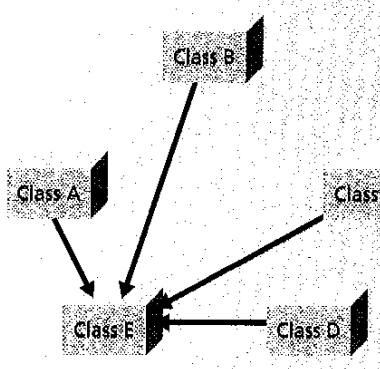
- *Total lines of code.* The LOC metric tallies all lines of executable code within the system, class, or method. Developers widely use the LOC metric as a measure of system size. Since you can determine the total LOC automatically, you can use it as a benchmark to compare unlike systems.
- *Total function calls.* TFC tallies the number of calls to methods and system functions within the system, class, or method. This metric measures size in a way that is more independent of coding style than the LOC metric.
- *Number of classes.* The NOC metric counts all classes within the system, including both nonvisual classes and GUI classes like controls and windows.
- *Number of windows.* The NOW metric counts the number of visible windows within the system, a number that simply indicates the size of the user interface.

Class and method size

You can usually consider class or method size metrics to be equivalent to design or quality metrics because unusually large classes or methods may indicate ill-conceived abstractions or overly complex implementations. Measurements of class or method size measures that differ substantially from average values are generally good candidates for inspection or rework. Class and method size metrics include

- *LOC* and *function calls per class/method.* These metrics are similar to the LOC system size metrics but focus on individual classes and methods.

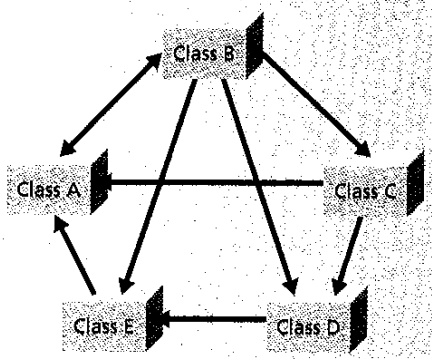
Figure 1. A type of class dependency known as fan-in.



In the illustration, each arrow represents a dependency of one class on another. In this case, all objects depend on class E.

By using the fan-in technique, you can reuse any of the related classes in another project simply by including class E.

Figure 2. A type of class dependency known as fan-out.



With this technique, you have to include many objects whenever you want to reuse one of them. You also will have a more difficult time maintaining objects since changes to objects may ripple through the system. Reusing class E would require including all the other classes because of the interdependencies.

- *Number of methods per class* and *public method count per class*. The number of methods per class indicates the total level of functionality implemented by a class. The number of public methods indicates the amount of behavior exposed to the outside world and provides a view of class size and how complex the class might be to use.
- *Number of attributes per class* and *number of instance attributes per class*. The number of attributes in a class indicates the amount of data the class must maintain in order to carry out its responsibilities. Attributes can either be instance attributes, which are unique to each instance of an object, or class variables, which have the same value for all members of the class.

Coupling and inheritance

Coupling and inheritance metrics help measure the quality of an object model. More specifically, they help reveal the degree to which interobject dependencies exist. Ideally, objects should be independent, which makes it easy to transplant an object from one environment to another and reuse existing objects when you build new systems. The reality, of course, is that objects have interdependencies, and reusing them is rarely as simple as cutting and pasting. All too often, the number of dependencies is so large that understanding and moving the entire group of objects is more expensive than rewriting objects from scratch.

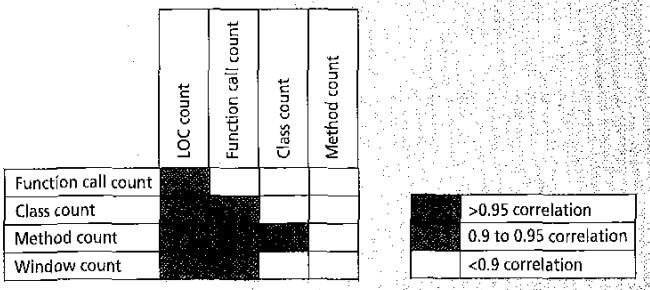
- *Class fan-in*. Fan-in metrics measure the number of classes that depend on a given object. If you have to couple your objects, you'll want to use fan-in, since it centralizes dependencies, as illustrated in Figure 1.
- *Class fan-out*. Fan-out metrics measure the number of classes on which a given class depends. You should avoid using the fan-out technique, since it represents a situation in which you spread dependencies across the system, as illustrated in Figure 2.
- *Class inheritance level*. The inheritance depth of a class is the number of its direct ancestors. An unnecessarily deep class hierarchy adds to complexity and can represent a poor use of the inheritance mechanism.
- *Number of children per class*. This metric measures the number of direct descendants of a particular class, which can indicate unnecessarily complex hierarchies.

Class and method internals

Internals metrics provide various measures of quality for both classes and methods. They can be used to identify the areas where remedial action might prove useful. At a systemwide level, you can also use these metrics to compare the overall quality of one system with another. These metrics include

- *Number of global/shared references per class*. This metric indicates the number of global variable references found within a class. Global references tend to break encapsulation and inhibit reuse. While global references may be difficult to eliminate entirely, they should be used as sparingly as possible.
- *Method complexity*. The method complexity metric

Figure 3. Sizing metrics compared to traditional LOC counts.



measures *cyclomatic complexity*, the number of different execution paths within a block of code. Code with many paths will be harder to understand and more likely to contain errors.

- *Number of public attributes per class.* This metric measures all attributes not specifically marked as protected or private. These attributes can expose an object's implementation to the outside world, which essentially violates good encapsulation principles.
- *Lack of cohesion among methods.* This metric measures the extent to which methods reference class instance data. In general, a high-quality design will have low coupling (interaction among objects) and high cohesiveness (objects that cannot be split apart).
- *Class specialization index.* The specialization index metric measures the extent to which subclasses override (replace) the behavior of their ancestor classes. Numerous overridden methods indicate that the abstraction may not have been appropriate, since (if much behavior needs to be replaced) the child and ancestors may not have much in common. A more appropriate form of subclassing involves extending class behavior with new methods rather than replacing or deleting behavior through overrides.
- *Percent of commented methods.* This metric measures the extent to which you've internally documented your code. The metric counts the number of separate comment blocks rather than the number of comment lines.
- *Number of parameters per method.* This metric measures the average number of parameters involved in invoking a method. A high number of parameters indicates a complex interface to calling objects, and should be avoided.

APPLYING OO METRICS

There are almost countless ways of applying metrics to your projects. This section describes only one perspective among many. As I've mentioned, the data I provide here is based on analysis of 18 PowerBuilder projects. The systems from which I derive this data span a variety of markets, with most of the systems being either in production or in beta. The size of each system ranges from several thousand lines of code to nearly 200,000 lines.

Sizing with a variety of metrics

Good project planning of course requires not only determining a system's current size but also estimating the size of future changes. How do the OO sizing metrics I mentioned above compare to traditional LOC counts? The data from the analyzed systems shows high correspondence between types of measurements, as illustrated in Figure 3.

The correlation of measures, such as LOC and function calls, means that they tend to move together. A system that has a high LOC value is highly likely to also have a high number of function calls, and vice versa. Another way to look at it is that their ratios are nearly constant from one system to another.

What this means is that the measures are essentially interchangeable; the high correlations mean that you have some freedom to use measures you are most comfortable with when you count and estimate. For example, if you have experience using LOC counts and have confidence in your LOC estimates, then by all means use those measurements. If class counts—or even windows counts—come more naturally, then use those measures instead.

Benchmarking

Experience derived from past systems is often a useful guide for future efforts. In particular, you can examine the metrics of the systems you've successfully implemented and treat those metrics as guidelines for future work. What worked in the past will likely continue to work in the future. Conversely, abnormal classes—or classes you haven't tried before—might mean trouble.

Table 1 illustrates a number of values taken from the sample of PowerBuilder systems. All of these values are 90th-percentile benchmarks, meaning that 90 percent of the observed systems, classes, or methods had values smaller than those shown in the table.

For example, about 90 percent of systems had an average method size of 21 or less. Systems with a higher average not only indicate you've written them in a procedural rather than OO fashion, but also run the risk of being difficult to understand. Individual classes generally had eight or fewer instance variables; methods rarely took more than two parameters.

You can use the numbers in the table to

- guide development,
- prioritize reviews and inspections, and
- review standards.

If your team ever asks you how large a method should be, this table can provide some help. Most of the methods in the PowerBuilder apps had fewer than 33 lines. The numbers in the table can help identify unusual classes and methods that might be good candidates for closer scrutiny.

Also, at some level above the 90th percentile, you may want to examine individual classes or methods to see if they make sense. I don't recommend any absolute rules, so use your own judgment. Suppose you find a method with 80 lines or a method with five arguments. Are they acceptable? Perhaps. But suppose you find a method with 800 lines or one with 20 arguments. These methods could almost certainly benefit from rework.

Table 1. Benchmarks derived from analysis of 18 PowerBuilder systems.

Metric	System average	90th-percentile benchmarks: Class	90th-percentile benchmarks: Method
Class and method size metrics			
LOC per class	145	286	N/A
LOC per method	21	N/A	33
Function calls per class	67	127	N/A
Function calls per method	9	N/A	14
Number of methods per class	10	21	N/A
Public method count per class	N/A	18	N/A
Number of attributes per class	4.40	8	N/A
Number of instance attributes per class	N/A	8	N/A
Coupling and inheritance metrics			
Class fan-in	N/A	4	N/A
Class fan-out	2.98	6	N/A
Class inheritance level	3.44	5	N/A
Number of children per class	0.88	1	N/A
Class and method internals			
Number of global/shared references per class	0.38	0	N/A
Method complexity	0.13[1]	N/A	7
Number of public attributes per class	2.75	4	N/A
Lack of cohesion among methods	0.89	1	N/A
Class specialization index	0.33	0.5	N/A
Percent commented methods[2]	63%	N/A	N/A
Number of parameters per method	N/A	2	N/A

[1] The system-level value is the percentage of methods with complexity of 10 or greater.
[2] This metric represents a lower bound.

Measuring method and class complexity

Method complexity almost directly relates to how likely errors will be. Attempting to reduce complexity is one of the ways developers in the procedural-language world employ the cyclomatic complexity metric, which is also useful for analyzing OO systems.

Figure 4 illustrates the results of using the cyclomatic complexity metric to analyze one of the PowerBuilder systems. The bars indicate the amount of code present in objects having different levels of complexity. It is easy to see that almost half of all the code exists in classes that have fairly simple methods. There is very little code—only about 11 percent—in classes that contain a very complex method.

The story for defects is exactly the opposite. The large fraction of simple classes contained only about 12 percent of all defects. However, the small amount of complex code contained more than 40 percent of the problems.

Even in OO systems you can measure complexity to screen methods for potential problems and to prioritize where your scarce review time is best spent.

Determining critical quality

To find out how much quality you have in your system, you can find how many of the metrics provide abnormal measures at the system level. Figure 5 correlates those abnormalities to the system size of the sample applications.

Small systems seem to have a wide-ranging quality level. Bear in mind that these are all production systems. Does quality matter for a small system? Generally speaking, the level of discipline and rigor required of small teams is lower for smaller systems, since for a small system developers can—and tend to—keep everything in

Figure 4. Complex code contains more defects. Understanding complexity metrics can help you spend review time wisely.

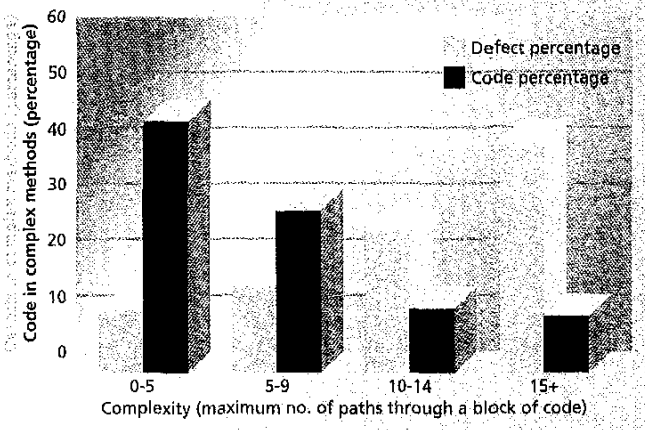
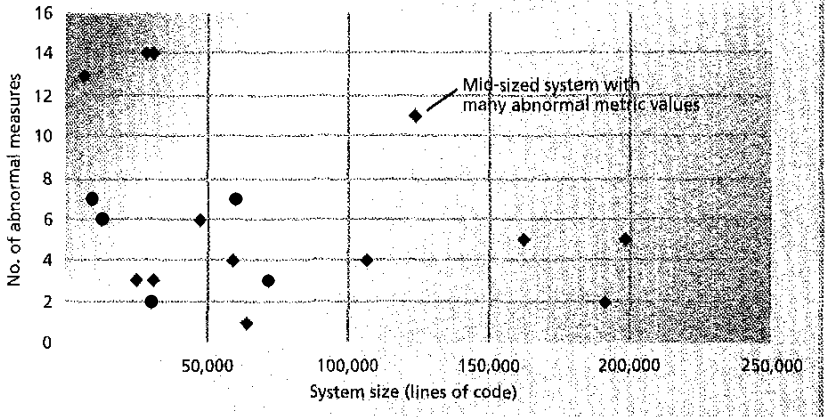


Figure 5. Smaller systems can tolerate more abnormal metric measurements, but larger systems should have fewer to be successful.



their heads. In these cases there is less need for formal documentation since a small team has less communication overhead.

With a small handful of developers, software can be more complex because good developers can more easily understand the whole system. Even so, the ability to deliver a working system with a great deal of complexity does not mean lower quality should be your preferred aim.

When it comes to large systems, this is definitely not the case. Communication and quality are critical. Lack of discipline can have a disastrous effect on the eventual outcome. For example, one mid-sized system (see Figure 5) has a great deal of abnormal metrics values. Despite the designers significantly reworking and then rereleasing this system, it is no longer in service. The lesson to be learned is that as system size increases, quality becomes critical.

Metrics can play a valuable role in system development. OO metrics, in particular, can be used to help

- size and estimate work,
- define review policies and development guidelines,
- prioritize review and inspection efforts, and
- assess overall system quality.

Incorporating OO metrics into your development plans is a simple step toward creating better systems.

In fact, now is a good time to begin using metrics because the barriers to implementing them are dropping. Development tools, even some that are freely available, often make it much easier to capture detailed measurements. Some metrics even work within CASE tools and analyze any language that the tool supports.

On the other hand, while the time is indeed right, you'll find that the pressure for rapid development will only increase, which will discourage disciplined development practices.

The solution? Keep it simple. If you do deploy a metrics program, you'll find it'll do just that. ■

Mark Schroeder *is a senior software engineer at folioTrade, an Internet startup in the financial services industry.* *Schroeder thanks American Management Systems for providing invaluable guidance and information, without which this article would not have been possible.* *Contact him at schroederm@foliotrade.com.*