

A Primer On Object-Oriented Measurement

Khaled El Emam
National Research Council, Canada

1 Introduction

Many object-oriented metrics have been proposed over the last decade. A few of these metrics have undergone some form of empirical validation, and some are actually being used by a number of organizations as part of an effort to manage quality.

There is evidence that using object-oriented metrics can be beneficial to conducting business and to the bottom line. For example, one study recently showed that prediction models using design metrics had an error rate of only 9% when estimating the proportion of classes with post-release defects for a commercial Java application. This is encouraging as such estimates can be used to allocate maintenance resources, and for obtaining assurances about software quality. Another study estimated corrective maintenance cost savings of 42% by using object-oriented metrics. Here, classes containing defects were predicted early in the project and were targeted for inspection. This particular study made some rather conservative assumptions, hence the savings may be larger.

The purpose of this talk is to present an overview of object-oriented metrics, their use, and their (empirical) validation.

2. Object-Oriented Metrics

Object-oriented metrics can be considered from a number of perspectives. We present three common ones below.

First, there are *static* and *dynamic* metrics. Static metrics can be computed by the analysis of the source code or design document. These characterize the static structure of an object-oriented system. Their premise is that they somehow measure *cognitive complexity*. Examples are metrics of coupling between classes and depth of a class in the inheritance hierarchy. Dynamic metrics require the execution of the program in order to be collected. These characterize the dynamic behavior of the system. Examples include metrics of test

coverage and extent of class usage (say an execution profile).

Second, object-oriented metrics can also be measured at different levels. At the *system* level, you can have metrics that capture some structural characteristic of the whole system. For example, some companies define a "maintainability index", which uses metrics to characterize the overall maintainability of a system. At the *file* level, you can capture information about the code in a particular file. This may be, for example, the size or the complexity of the data defined in that file. A file may have multiple classes and multiple methods. At the *class* level, you can measure structural features of the class, such as its coupling and inheritance depth. If in a particular environment a file contains all the information about a single class, then some of the file and class level metrics may have the same values. At the *method* level, metrics can be defined in a similar way to metrics for procedural systems.

Finally, you can characterize metrics by the development phase when they can initially be collected and when they become stable. Ideally, we would like to collect metrics at the earliest phase so that the metrics information can be used for the most effective decision making. In an iterative or spiral development environment, the metrics will become stable for only short periods.

The focus here is on static class-level metrics since these have been studied the most.

3. Utility of Object-Oriented Metrics

There are at a minimum two ways in which object-oriented metrics can be used: quality estimation and risk management. Both of these assume that the metrics are empirically valid.

3.1 Quality Estimation

To perform a quality estimate, it is necessary to have historical data. The historical data is then used to build an estimation model. Ideally, the historical data should come from a previous release of the

same product within the same organization, or a similar product also within the same organization. The reason for this preference is that quality estimation models usually do not work well across organizations.

In practice, quality estimation means either estimating reliability or maintainability. Both of these are discussed below.

Reliability is typically (in the context of o-o metrics work) measured as the number of defects. These can be pre-release (e.g., testing defects) or post-release (i.e., those found by customers). The estimated number of defects can also be normalized by a size measure to obtain a defect density estimate.

Maintainability is typically measured as change effort. Change effort can mean either: (a) the average effort to make a change to a class, or (b) the total effort spent on changing a class. The latter accounts for the number of changes that a class undergoes, whereas the former is specific to a single change.

Changes can be classified into change types and maintainability estimates made for different types of changes, for example, estimating maintenance effort for corrective changes only or for functionality enhancement changes only. In addition, for corrective changes, changes to fix different types of defects can be distinguished.

Depending on the objectives of estimation, one may be interested in estimating reliability or maintainability for each class in a system individually, or for the whole system in aggregate. For example, one can estimate the number of post-release defects for each individual class in an application. Alternatively, one can sum up all of these individual estimates to come up with the estimated number of defects for the whole system.

Another strategy is to classify the classes into "high" and "low" risk. A "high" risk class is one where the intention is to take action once it is identified. For example, a "high" risk class may have more than five pre-release defects or takes more than 40 person-hours on average to make a single change to. In such a case, one can estimate the proportion (or number) of classes in a system that are "high" risk. This can serve as another quality measure for individual classes (i.e., is it "high" risk) or for the whole system (i.e., how many "high" risk classes do we have).

Generally, the scope of the quality estimate should make clear what parts of the system are under consideration. If an external library is used, for example, it must be clear whether the library is within the scope of the estimate or not.

3.2 Risk Management

Risk management here means identifying potentially problematic classes as early on in a project as possible and taking some preventative action. There are three common techniques with which this can be done: classification models, rank models, and thresholds. Since risk management entails predicting what will happen in the future, and planning for it, all of the above approaches are predictive, but differ in the way the predictions are made.

3.2.1 Classification Models

Classification models are built using historical data, and classify each class as either "high" or "low" risk. A "high" risk class can be one that has more defects, is more likely to have a defect, or that is expected to have high costs to change.

For example, if we are interested in post-release defects within the first 6 months, then we can identify classes with a high probability of a post-release defect. Once these are identified, then appropriate action can be taken. Appropriate action can mean, for example, putting more or better resources for inspecting and testing these classes, redesigning these classes or even the part of the system where these classes reside. The logic of such actions is that fixing the problems early will result in an overall cost reduction.

3.2.2 Rank Models

Similar to classification models, rank models are typically built using historical data. However, instead of classifying each class as "high" or "low" risk, the model ranks all the classes by their risk level. For example, risk may be defined as the probability of a post-release defect. Classes with a higher predicted post-release defect are at a higher risk. The advantage of ranking over classification is that it allows the organization to tailor its efforts to its resources.

For example, if a future release has 100 classes, and say a classification model predicts that 15 of these are "high" risk in the sense that they will have a customer found defect within the first 6 months

after release. Since it is not known which of these 15 classes has the highest risk, it is prudent to take preventative action for all 15. Now, let us assume that the previous release of this product had only 5% of the classes with customer found defects. It would be reasonable to expect similar performance for this release (assuming there are no reasons to believe otherwise, such as a turnover of almost all staff from the previous release). If a rank model was available, this information can be used: preventative action need to be taken on the top 5% of classes only rather on all 15.

3.2.3 Thresholds

With thresholds, one attempts to identify the range of values on the object-oriented metrics that are acceptable or unacceptable, and take action for the components with unacceptable values. This means identifying thresholds that delineate between *acceptable* and *unacceptable*.

Thresholds can be defined in at least three ways. The first is an experiential way. This means experienced developers or designers define the thresholds based on their perceptions. One would consider this approach as a last resort if other approaches do not work. The second approach is to use percentiles. For example, data is collected from a representative sample of historical projects and the threshold is drawn at say the 80th percentile on a particular metric. Classes above the percentile threshold are then considered to be unacceptable. The third approach is the most systematic and involves building threshold statistical models. These relate the metrics with the criterion of interest (e.g., defects). However, recent studies using this third approach have revealed that there are no valid thresholds for many popular object-oriented metrics. This still means that thresholds may still be useful, but they are sub-optimal.

4. Interpreting Validation Studies

Empirical validation studies examine the relationship between metrics and the criterion of interest, for example, the incidence of defects. In the recent past there has been a growth in the number of such studies.

While the growth of studies is encouraging, from a consumer's perspective (i.e., the perspective of one wishing to use object-oriented metrics) it is important to be able to interpret the results of such studies properly. This means making a sound judgement

about the strength of evidence supporting the efficacy, or otherwise, of a particular metric.

The following basic heuristics can be used to evaluate study quality:

- A descriptive summary of object-oriented metrics is not a validation of the metrics.
- Studies that look only at bivariate relationships (e.g., simple correlation coefficients or so-called univariate regressions) ignore obvious (and easy to control) confounding effects and therefore cannot be trusted.
- Studies that have as the dependent variable measures such as *defect density* or *productivity* suffer from a ratio correlation statistical artifact and therefore cannot be trusted.
- Studies that evaluate the accuracy of quality models using measures such as proportion correct, correctness, and kappa may be hiding a prevalence bias and therefore must be interpreted with caution.
- Studies that use automatic selection algorithms such as ordinary stepwise selection should be looked at with caution since these algorithms are notorious for capitalizing on chance (i.e., often noise variables will be selected, and not the same variables will be selected repeatedly).
- Studies that validate a quality model on the same data set that was used to construct the model (without a k-fold cross-validation or a bootstrap) cannot be trusted since they inflate the accuracy of the quality model.
- Caution should be exercised with studies where the dependent variable is the count of defects found since measuring testing and post-release defect counts can be unreliable.
- Caution should be exercised when surrogate measures are used (especially for the dependent variable). For example, using code churn as a surrogate for maintenance effort.

The above heuristics are based on observations from the current object-oriented metrics validation literature. The adverse impact of ignoring these heuristics can be easily illustrated through examples and/or Monte Carlo simulations.

5. References

Much of the material referenced in this talk, plus some publicly available tools, can be found at:

<http://www.object-oriented.org>