Software Quality Data  
Part 1: Basic and Derived Metrics

Abstract

We measure, quantify and report on software quality. But can we control it? Can we actually assure quality (as opposed to just measuring it)? This is the first of three papers in which we will learn how we can go beyond just quantifying the quality of the software after it has been built.

In this “Part 1 Basic and Derived Metrics”, we will discuss:

• The three basic metrics from which the most important quality measures can be derived (Defects, Effort, and Size).
• Several important metrics that can be derived from those basic three:
  • Defect Injection Rate
  • Defect Removal Yield
  • Defect Removal Efficiency
  • Review Rate

In “Part 2: Quantitative Quality Planning”, we will look at how to use this information to produce a Quality Plan that we can use to understand our quality performance before the project is complete.

Finally, in “Part 3: Quality Control Using In-Process Data” we will discuss how to use our Quality Plan and the data we are collecting to actually control the quality of the software we are producing.

Introduction

Those of us who are charged with assuring the quality of the software that our organizations produce have quite a challenge. We are being asked to manage something that is difficult to quantify, and even harder to control. We are familiar with a variety of activities we could engage in that would be likely to assure the quality of our software, and we do what we can to be sure that those activities are done.

But activity is not sufficient. Indeed, there are far more activities that we could engage in than there is time to do so. So we must choose how we will spend our available time and effort; what we will do, and to what extent. Making these choices requires that we have a clear picture of what we are trying to achieve. Not the nebulous objective of “good quality”, or even “better” than last time. We need to set specific quality goals for each project: goals that we can use to direct our activities. And finally, we need ways of understanding how we are performing against those goals early enough in the project to allow us to take corrective action.

The key to all of these things; the key to our ability to actually manage quality as we are working, is data. By collecting a few carefully focused measures, we can build the understanding we need to be able to set quantitative quality goals, plan to achieve those goals, and take corrective action during the project to make our success more likely. We need to use data to understand:
Software Quality Data
Part 1: Basic and Derived Metrics

• How we are spending our time,
• How the activities we engage in are affecting the quality of our software,
• What our past performance has been, and
• What performance we can reasonably expect to achieve on the next project.

Although the methods we will explore are rarely used in software projects, they are by no means new. These methods have been used with great success for many decades by many engineering disciplines, as well as manufacturers. Luminaries such as Walter Shewhart and W. Edwards Demming made these topics the subject of their lives’ work, and taught the world how to apply them. The Software Engineering Institute (SEI) at Carnegie Mellon University (CMU) has been building our understanding of how these principles apply to software with their work on the Capability Maturity Model for Software (CMM), the Personal Software Process (PSP), and the Team Software Process (TSP). Much of the content of this paper is application of these principles from the SEI.

Basic Quality Measures

There are many things we could measure in our software projects, and each measurement will give us specific information that will be useful for certain purposes. If your company has a measurement program, then by all means, use the data that are being collected in any way that you can. But, if the data we will discuss here are not included, then you must collect them as well, because each one provides information that is important to your ability to manage quality.

Basic Measure #1: Defects

Log every defect, beginning as early as possible in the development process. Almost every organization logs defects that are found in their products; the only question is at what point this logging begins. For many, defect logging does not begin until system test or delivery. This is too late to manage quality during development.

In order to manage quality during development, you must have defect information during development. The best case would be for engineers to begin logging defects during their detailed design step, and to continue logging them through coding, reviews, compile, unit test and integration. Until your engineers have been trained to use the Personal Software Process (PSP), they are unlikely to faithfully log defects before peer reviews. This is unfortunate, and your insight into the quality of what is being built will be constrained. But the principles we will discuss can still be used if formal defect logging begins with peer reviews and unit testing.

The specific information we will need about each defect is:

• The artifact that contained the defect.

This is not a list of every artifact that required changes; only the artifact that was the root cause of the problem. If your design was flawed and caused you to have to rework not only the design, but also code and test plans, you would log it as a design defect. (Code that faithfully implements a defective design is not defective; the design was the defective item.)
Software Quality Data
Part 1: Basic and Derived Metrics

• The **type** of defect that it is. That is, is it a logic problem, and interface problem, a syntax error, an erroneous comment, etc.

Your organization will need to build up its list of defect types based on the defects that actually occur; and you will need to carefully define each of those types so everyone knows how to use them.

• The development **phase** when the defect was **injected**. Was the defect injected in Design, Code, Testing? This will usually be obvious, but not always. Where it is not clear, make your best guess.

It is particularly important to track defects that are injected in testing or other phases that should be removing rather than injecting defects. Although this will naturally happen from time to time, it should be the exception rather than the rule. If you find that it is a common occurrence, then you have identified an improvement opportunity.

• The development **phase** when the defect was **removed**. In almost all cases, this will be the same phase in which it was detected; but if not, log the phase when the defect was actually removed.

• The total amount of **effort** that the defect cost the organization. People tend to undercount this effort, so be sure that all of these things are counted:
  - **Diagnosis**. For example, when you are testing, some amount of effort will be required to understand the symptom and figure out the cause. Sometimes we spend hours in this diagnosis phase: adding debug code and running tests over and over until we can isolate the problem and understand it. *All* of the diagnosis time must be counted. (When a problem is found during a review, there is often no diagnosis time. The review time itself is accounted for separately, and is not counted as diagnosis time.)
  - **Rework**. All of the time that is spent reworking artifacts to remove the defect must be counted. This includes collateral damage. For example, when a design defect is found during test, it often requires rework, not only in the design, but also in the code, and possibly in the test plans. When this is the case, you must count all of the time involved, including recompiling and re-building executables.
  - **Re-testing**. When a defect has been corrected during testing, you will generally have to re-run the affected tests. This is also part of the effort caused by the defect. (If not for the defect, you would have run the test only once.)
Basic Measure #2: Effort

Log all “Cost of Quality” effort. All of the costs of quality must be logged. For each quality-related activity, you will include all of the effort to do the activity, including the effort spent removing defects during that activity. This does not result in double-counting the defect removal effort, because you will not be adding the defect removal effort to the effort in these activities. (The effort logged for each defect will be used in a different way, as we will see later.)

You will have to examine your development process to identify all of these quality-related activities, but they generally fall into these categories:

- **Quality Planning.** Many people count quality planning as part of project planning, and do not count it as a cost of quality. Others do count it. You will need to decide how you will account for quality planning and account for it consistently in your organization.

- **Test Planning.** All time spent in developing test plans and test cases must be counted. Best practice is for the developers to create formal Unit and Integration test plans. If formal test planning is done for these activities, then it should also be counted.

Some organizations embed test planning into other activities. For example, they do system test planning during the requirements phase, integration test planning during system design, or unit test planning during detailed design. If your organization does these things, then you may choose not to account for the test planning as a separate activity. Whatever choice is made, make sure that you account for test planning consistently in your organization.

- **Reviews.** Account for all time involved in each kind of review you do. For example, for a formal Fagan-style inspection, you must count the preparation time, each attendee’s time at the pre-review meeting, all time people spend desk-checking, each attendee’s time in the inspection meeting, and all time in post-inspection activities (including, as we mentioned above, time spent fixing the defects the review uncovered).

- **Compile.** The vast majority of the time a programmer spends running the compiler is actually defect-removal time. PSP-trained engineers will log all of this time as well as the defects they remove while compiling. Until your engineers have been PSP-trained, you will be unlikely to be able to get this sort of information from them.

- **Testing.** For each testing phase (unit, integration, system, acceptance, field, etc.) the effort spent by all people in the organization should be logged. This includes the testers themselves, the developers (diagnosing and fixing problems), and for customer tests, all of the people who support the customer during the test.

It is important to note that a comparison of the time in a testing phase with the total time required to fix the defects in that phase should show most of the time being spent fixing defects. Recall that the defect fix time should include all diagnosis, fixing and re-testing. The only test time not accounted for by the defect time should be the time to set up the test environment and run each test case once.


Basic Measure #3: Size

Log all artifact sizes. All of the quality-related activities that we will engage in are affected by the size of the artifacts involved. The larger is a document or a program, the more time it will require to review or test, and the more defects we can expect it to contain. Therefore, managing quality requires that we know the size of the artifacts we are dealing with.

For all documents including user guides, and technical specifications, pages of text is generally a useful size measure.

When code is actually written by a person, lines of code (LOC) is an easily countable and very useful size measure. There is much discussion of this in the software engineering community, with different people advocating different size measures for code. The bottom line is a particular measure’s usefulness for a certain purpose. For the purposes to which we will put it, LOC is a useful measure.

For code that is generated from a tool (as opposed to being written by a person), no measure of the code itself will be useful. You will want to measure the artifact that the person creates as input to the code-generator. For example, if a person uses a visual screen editor to create a screen, then a system converts that to code, a useful measure may be the number of objects the person defined in the screen.

The most complex situation is where a person uses automatically generated code as the starting point for significant coding work. In this case, the only reasonable method may be to measure the two steps separately. Count the inputs to the automatic generation and measure the resulting code (LOC), then measure the final code to see how much it has grown.

Whatever measures you choose to use, the key is that they be defined carefully so that everyone in the organization uses them consistently. Inconsistency in measurement will compromise the usefulness of the data more than any decision you make in defining your measures.
Derived Quality Measures

By measuring the three things described above (defects, effort and size), we can compute a variety of metrics that we can use to manage quality. Each will be described next.

**Derived Measure #1: Defect Injection Rates**

Every activity in a software development project can be thought of as a defect injection activity that injects defects into the resulting product at a certain rate. Our design activity injects a predictable number of design defects for each page of design specification we write. Our coding activity injects a predictable number of code defects into each 1000 lines of code (KLOC) we write. And even during testing, we will inject some number of defects for each 100 defects that we fix (though we hope our tests remove many more defects than they inject).

\[
\text{Defect Injection Rate for a Phase} = \frac{\# \text{ of Defects Injected in Phase}}{\text{Size of Objects produced in Phase}}
\]

\[
\text{Defect Injection Rate for Project} = \frac{\text{Total \# of Defects Injected}}{\text{Size of Software Produced (e.g. KLOC)}}
\]

In order to manage the quality of the software we are producing, we need to understand our defect injection rates for the various activities in our software development process. We can use these rates to plan for defect removal activities, as well as to understand how the current project compares to our past performance.

**Derived Measure #2: Defect Removal Yields**

Defect removal yield is the percentage of existing defects that any defect removal activity actually finds.

\[
\text{Defect Removal Yield for a Phase} = 100\% \times \frac{\# \text{ of Defects Removed in Phase}}{\# \text{ of Defects Existing at Beginning of Phase}}
\]

\[
\text{Defect Removal Yield for Project} = 100\% \times \frac{\# \text{ of Defects Removed before Unit Test}}{\text{Total \# of Defects Removed}}
\]

Yield is not useful for in-process management because you will not have a reasonable understanding of the total number of defects until development is complete. But it is an important historical measure that you can use in your quality planning, and it can highlight process improvement opportunities.
Derived Measure #3: Defect Removal Efficiencies

The defect removal efficiency of activities allows us to compare how efficiently different activities use our most critical resource (people’s effort) to remove defects.

\[
\text{Defect Removal Efficiency for a Phase} = \frac{\# \text{ of Defects Removed in Phase}}{\text{Hours of Effort Expended in Phase}}
\]

While we are unlikely to eliminate an activity from our project solely on the basis of its defect removal efficiency, we can use this information to plan for how we will spend our effort in a project in order to achieve our goals with the available resources. We can also compute this measure immediately after an activity is complete to see if its efficiency was as we expected, and take corrective action if not.

Derived Measure #4: Review Rates

Review Rate is simply the number of units of size reviewed per effort hour expended during a review. Although this metric does not look at defects, it turns out to be a particularly important one in planning for quality, as well as managing quality during the project.

\[
\text{Review Rate for a Review} = \frac{\text{Size of Objects Reviewed (e.g. KLOC)}}{\text{Hours of Effort Expended in Review}}
\]

This is so because doing a careful and thorough review of an artifact of a certain size requires a certain amount of time. If a review is rushed, it is almost certain to be ineffective at detecting defects. We can use this knowledge not only to plan sufficient time for our reviews, but also to check our reviews immediately after they are complete to see if they are likely to have been thorough.

Conclusion, Part 1: Basic and Derived Metrics

Data is the key to our ability to make actionable quality plans and use them to actually manage quality during a development project. With the necessary data, you can make quantitative plans that include targets for product size, defect content, and effort for defect-removal activities. The data you collect as you are working through the project can then be compared with the quantitative plans so you can see when things are going according to plan, and when corrective action is needed.

In “Part 2: Quantitative Quality Planning”, we will look at how to use this information to produce a Quality Plan that we can use to understand our quality performance before the project is complete.

Finally, in “Part 3: Quality Control Using In-Process Data” we will discuss how to use our Quality Plan and the data we are collecting to actually control the quality of the software we are producing.