

Can Clean Pipes Produce Dirty Water?

J. Voas

Reliable Software Technologies Corporation

Suite 250

21515 Ridgetop Circle

Sterling, VA 20166

phone: (703) 404-9293

fax: (703) 404-9295

Of course they can. Clean pipes can break, they can be attached to the wrong source, or the original water source may infuse dirty water into the pipeline. The complimentary question is “whether dirty pipes can produce clean water.” Once again the answer is “yes”, but that is less likely.

The analogy between water flowing from pipes and software flowing out from process standards is quite simple. Consider the original set of requirements to be the original water source, and each successive process applied to the developing software is the next link in the pipe. In this analogy, each link in the pipe is a process, and each process is either geared toward developing software or validating software. Certain processes may be reapplied during development, and hence our pipeline may contain distributed, redundant links. Eventually something will exit from the pipeline, and with a little luck, that will be quality software. The quality of the pipeline does impact the quality of the code, but to what degree and at what cost can vary. There is no guarantee that even the most superb software development processes will manufacture correct code.

“Software quality” has become a catch-all buzzword for a huge family of various methods geared either toward **achieving** better software or toward **assessing** how good software is. Each of these methods represents a different pipe link, and there are various ways of integrating these links that make

sense; other combinations are nonsense. For example, putting the system-level testing link in front of the processes for checking the requirements for ambiguity is foolish.

Today's software quality methods can be generally labeled as either **process-oriented** or **product-oriented**. Process-oriented methods are focused more towards achieving quality than assessing it. Process-oriented standards would include entities such as CMM, DO178-B, and ISO-9003, whereas product-oriented standards would include software metrics and system-level testing. It is not uncommon for process-oriented standards to call for certain product assessment methods to be performed, but these standards are more concerned with team infrastructure and interpersonnel communication than rigorous product measures. In our analogy, product-oriented methods would be valves at different locations along the pipe that allow the developing software, whether in a design language or specification format, to be sampled in order to assess its correctness. Those product-oriented methods that are specifically for software assessment would be clustered near the exit from the pipeline, since in the earlier phases, code will not be available. The process-oriented methods would attempt to ensure that the original requirements were clean, and that every transformation through the development cycle did not inject dirt. Clearly if the water is dirty at point A, then at a later point B, it will still be that dirty and possibly even dirtier, unless some filter is employed between A and B. Dirty water doesn't clean itself any more than does incorrect software corrects itself.

Today's software process "movement" is a logical, evolutionary advancement that evolved out from the unstructured, *ad hoc* software engineering methods used in the 1970s and early 1980s. The cry for systematic, repeatable methods that engendered more reliable software was "a natural" given the state-of-the-practice at that time. This call has been answered by dozens of different software development standards, even manufacturing standards that have been suppositively refurbished to account for software idiosyncracies. Even organizations such as the ECC have adopted such standards, not because the standards guarantee higher quality software imports, but rather as protectionist legislation to reduce software imports. The unfortunate effect of the process movement has been a perception that would refute the claim that clean pipes could produce dirty water. Software development processes are heavily based on manual effort and fallible software tools. Believing that clean pipes cannot produce dirty water amounts to accepting the position

that flawed software tools and flawed human efforts will somehow offset each other and produce good software. Still, this myth lives on in the minds of both novice and expert software engineers, and software assessment continues to take a backseat to process improvement models.

State-of-the-Practice in Software Assessment

Why is it that our industry seems so preoccupied with quality processes instead of quality products? Well for a start, exhaustive testing of software is generally infeasible, testing software to high levels of reliability is intractable, and software assurance models are often viewed suspiciously. Furthermore, the idea of doing things right from the outset has an intuitive flavor that is simply too alluring to challenge. We are facing what would appear to be insurmountable obstacles in accurately assessing software quality. And because of this, most of us are either clinging to the outdated techniques of yesterday or have decided that software assessment is hopeless and have embraced the myth.

You might believe that to assess software quality, all you need is accurate reliability estimation. Because we can almost never know the true reliability of a piece of software, most software reliability models employ error history information to predict future failures. However different error-history-based reliability models often compute different reliability predictions for the same data, making it impossible to determine which model will be the most accurate for a specific system. This implies that today's state-of-the-practice in software reliability assessment is deficient.

Even direct measurement of software quality is less than practical. Consider the fact that the testing effort required to establish a certain mean-time-to-failure (MTTF) with 90% confidence is at least twice that MTTF. Even ignoring all problems of test generation, test oracles, test-administration overhead, and the use of overspeed execution and parallel hardware for testing, it is difficult to see how more than about 10 tests/second for complex software could be performed. Current state-of-the-practice is perhaps three orders of magnitude less quick.

Many of the remaining software assessment approaches focus on metrics. Over 100 software metrics are in widespread use today that mainly measure structural or "static" properties. However only a handful of approaches attempt to measure behavior dynamically. The key problem with structural

metrics is that they do not capture the essence of software. Software defines a process by which an input is transformed into an output by a series of instructions. This transformation is the essential characteristic of the software. A program execution is a series of state transitions, where the final state contains the output. Exactly what effect a particular instruction has on the mapping between program inputs and program outputs is determined by the program's input distribution and the program's instructions. Structural metrics cannot capture this dynamic aspect of software behavior. This lack of connection with the behavior of the software makes structural metrics especially poor as software quality assessors.

Our research group's key interest for the last three years has focused on a family of validation assessment processes termed software *fault-injection* methods. Software fault-injection purposely "messes up" your software in some manner during execution, and checks to see how that affects your software's output. Since fault-injection operates on software, it is a pipe that lies near the exit of the pipeline, and in some developments, will be the last pipe.

Software fault-injection is not without limitations. The combination of problems that most information systems will experience during their lifetimes is intractable. Software fault-injection simulates some of these events, and by doing so, it predicts how the software will behave in the future if these anomalous events were to occur. Also, you gain a rough feeling for how your software will behave when confronted with the remaining anomalies that were not simulated, since some of the behaviors will be similar. This entire process of fault-injection involves a lot of statistics and pseudo-random fault-injection methods. Also, the actual process of instrumenting for the anomalies is quite complex. But the results can be most informative, particularly when you learn that your code doesn't handle problems quite as well as you thought it would. This information provides an immediate quality improvement opportunity.

Interestingly, software fault-injection methods, while they directly assess the behavior (i.e., goodness) of your code, also indirectly assesses the goodness of your pipes. By this, we mean that if software fault-injection methods find that software is intolerant to either faults within itself or anomalies that can attack the software from external sources (such as human factor errors, failed external hardware, or failed external software), then we learn that the other software development processes failed to build in the necessary water

filtering mechanisms. Code must be designed with the proper water filtering systems to ensure that the end result from the pipe is crystal-clear water. If software is incapable of producing the outputs we want under even anomalous circumstances, then we cannot claim that the pipes are clean. And if over time it can be demonstrated (via the results of fault-injection) that the set of pipes you have used over and over again do result in clean water, then you at least have anecdotal evidence that correlates your processes with the quality of your software.

Conclusions

The quality of your pipes is only one part of the formula for determining the purity of your water. It is myth to believe differently. Sadly enough, persons at the highest levels in governments and corporations have all swallowed this lure, hook, line, and sinker.

Quality software does not fail often, and never fails in hazardous ways. Software development processes do not define software quality; software behavior does. Behavior is an intrinsic characteristic of software, and it can be viewed without regard to the software's developmental history. Before we can have faith in software standards and process models, it must be demonstrated that they have a quantifiable relationship to the behavior of the software produced according to them. Parnas once said: "It seems clear to me that not only is a 'mature' process not sufficient, it may not even be necessary."

The current popularity of process-oriented assessment techniques is, in part, a reaction to the intractability of performing adequate software assessment. Unfortunately, the relationship between development processes and the attainment of some desired degree of product quality is not well-established. This problem is particularly acute for formal methods: performing the required processes does not give a quantifiable confidence that the software, when released, will have the required reliability. In short, testing measures the right thing — reliability — but it often cannot measure it to the desired precision. Process measurements are more tractable, but they measure the wrong thing.