Frequently Forgotten Fundamental Facts about Software Engineering

Robert L. Glass

This month's column is simply a collection of what I consider to be facts—truths, if you will—about software engineering. I'm presenting this software engineering laundry list because far too many people who call themselves software engineers, or computer scientists, or programmers, or whatever nom du jour you prefer, either aren't familiar with these facts or have forgotten them.

I don't expect you to agree with all these facts; some of them might even upset you. Great! Then we can begin a dialog about which facts really are facts and which are merely figments of my vivid loyal opposition imagination! Enough preliminaries. Here are the most frequently forgotten fundamental facts about software engineering. Some are of vital importance—we forget them at considerable risk.

Complexity

C1. For every 10-percent increase in problem complexity, there is a 100-percent increase in the software solution's complexity. That's not a condition to try to change (even though reducing complexity is always desirable); that's just the way it is. (For one explanation of why this is so, see RD2 in the section "Requirements and design.")

People

P1. The most important factor in attacking complexity is not the tools and techniques that programmers use but rather the quality of the programmers themselves.

P2. Good programmers are up to 30 times better than mediocre programmers, according to "individual differences" research. Given that their pay is never commensurate, they are the biggest bargains in the software field.

Tools and techniques

T1. Most software tool and technique improvements account for about a 5- to 30-percent increase in productivity and quality. But at one time or another, most of these improvements have been claimed by someone to have "order of magnitude" (factor of 10) benefits. Hype is the plague on the house of software.

T2. Learning a new tool or technique actually lowers programmer productivity and product quality initially. You achieve the eventual benefit only after overcoming this learning curve.

T3. Therefore, adopting new tools and techniques is worthwhile, but only if you (a) realistically view their value and (b) use patience in measuring their benefits.

Quality

Q1. Quality is a collection of attributes. Various people define those attributes differently, but a commonly accepted collection is portability, reliability, efficiency, human engineering, testability, understandability, and modifiability.

Q2. Quality is not the same as satisfying users, meeting requirements, or meeting cost and schedule targets. However, all these things have an interesting relationship: User satisfaction = quality product + meets requirements + delivered when needed + appropriate cost.

Q3. Because quality is not simply reliability, it is about much more than software defects.

Q4. Trying to improve one quality attribute often degrades another. For example, attempts to improve efficiency often degrade modifiability.

Reliability

RE1. Error detection and removal accounts for roughly 40 percent of development costs. Thus it is the most important phase of the development life cycle.

RE2. There are certain kinds of software errors that most programmers make frequently. These include off-by-one indexing, definition or reference inconsistency, and omitting deep design details. That is why, for example, *N*-version programming, which attempts to create multiple diverse solutions through multiple programmers, can never completely achieve its promise.

RE3. Software that a typical programmer believes to be thoroughly tested has often had only about 55 to 60 percent of its logic paths executed. Automated support, such as coverage analyzers, can raise that to roughly 85 to 90 percent. Testing at the 100-percent level is nearly impossible.

RE4. Even if 100-percent test coverage (see RE3) were possible, that criteria would be insufficient for testing. Roughly 35 percent of software defects emerge from missing logic paths, and another 40 percent are from the execution of a unique combination of logic paths. They will not be caught by 100-percent coverage (100-percent coverage can, therefore, potentially detect only about 25 percent of the errors!).

RE5. There is no single best approach to software error removal. A combination of several approaches, such as inspections and several kinds of testing and fault tolerance, is necessary.

RE6. (corollary to RE5) Software will always contain residual defects, after even the most rigorous error removal. The goal is to minimize the number and especially the severity of those defects.

Efficiency

EF1. Efficiency is more often a matter of good design than of good coding. So, if a project requires efficiency, efficiency must be considered early in the life cycle.

EF2. High-order language (HOL) code, with appropriate compiler optimizations, can be made about 90 percent as efficient as the comparable assembler code. But that statement is highly task dependent; some tasks are much harder than others to code efficiently in HOL.

EF3. There are trade-offs between size and time optimization. Often, improving one degrades

the other.

Maintenance

M1. Quality and maintenance have an interesting relationship (see Q3 and Q4).

M2. Maintenance typically consumes about 40 to 80 percent (60 percent average) of software costs. Therefore, it is probably the most important life cycle phase.

M3. Enhancement is responsible for roughly 60 percent of software maintenance costs. Error correction is roughly 17 percent. So, software maintenance is largely about adding new capability to old software, not about fixing it.

M4. The previous two facts constitute what you could call the “60/60” rule of software.

M5. Most software development tasks and software maintenance tasks are the same—except for the additional maintenance task of “understanding the existing product.” This task is the dominant maintenance activity, consuming roughly 30 percent of maintenance time. So, you could claim that maintenance is more difficult than development.

Requirements and design

RD1. One of the two most common causes of runaway projects is unstable requirements. (For the other, see ES1.)

RD2. When a project moves from requirements to design, the solution process’s complexity causes an explosion of “derived requirements.” The list of requirements for the design phase is often 50 times longer than the list of original requirements.

RD3. This requirements explosion is partly why it is difficult to implement requirements traceability (tracing the original requirements through the artifacts of the succeeding lifecycle phases), even though everyone agrees this is desirable.

RD4. A software problem seldom has one best design solution. (Bill Curtis has said that in a room full of expert software designers, if any two agree, that’s a majority!) That’s why, for example, trying to provide reusable design solutions has so long resisted significant progress.

Reviews and inspections

RI1. Rigorous reviews commonly remove up to 90 percent of errors from a software product before the first test case is run. (Many research findings support this; of course, it’s extremely difficult to know when you’ve found 100 percent of a software product’s errors!)

RI2. Rigorous reviews are more effective, and more cost effective, than any other error-removal strategy, including testing. But they cannot and should not replace testing (see RE5).

RI3. Rigorous reviews are extremely challenging to do well, and most organizations do not do them, at least not for 100 percent of their software artifacts.

RI4. Post-delivery reviews are generally acknowledged to be important, both for determining customer satisfaction and for process improvement, but most organizations do not perform them. By the time such reviews should be held (three to 12 months after delivery), potential review participants have generally scattered to other projects.

Reuse

REU1. Reuse-in-the-small (libraries of subroutines) began nearly 50 years ago and is a well-solved problem.

REU2. Reuse-in-the-large (components) remains largely unsolved, even though everyone agrees it is important and desirable.

REU3. Disagreement exists about why reuse-in-the-large is unsolved, although most agree that it is a management, not technology, problem (will, not skill). (Others say that finding sufficiently common subproblems across programming tasks is difficult. This would make reuse-in-the-large a problem inherent in the nature of software and the problems it solves, and thus relatively unsolvable).

REU4. Reuse-in-the-large works best in families of related systems, and thus is domain dependent. This narrows its potential applicability.

REU5. Pattern reuse is one solution to the problems inherent in code reuse.

Estimation

ES1. One of the two most common causes of runaway projects is optimistic estimation. (For the other, see RD1.)

ES2. Most software estimates are performed at the beginning of the life cycle. This makes sense until we realize that this occurs before the requirements phase and thus before the problem is understood. Estimation therefore usually occurs at the wrong time.

ES3. Most software estimates are made, according to several researchers, by either upper management or marketing, not by the people who will build the software or by their managers. Therefore, the wrong people are doing estimation.

ES4. Software estimates are rarely adjusted as the project proceeds. So, those estimates done at the wrong time by the wrong people are usually not corrected.

ES5. Because estimates are so faulty, there is little reason to be concerned when software projects do not meet cost or schedule targets. But everyone is concerned anyway!

ES6. In one study of a project that failed to meet its estimates, the management saw the project as a failure, but the technical participants saw it as the most successful project they had ever worked on! This illustrates the disconnect regarding the role of estimation, and project success, between management and technologists. Given the previous facts, that is hardly surprising.

ES7. Pressure to achieve estimation targets is common and tends to cause programmers to skip good software process. This constitutes an absurd result done for an absurd reason.

Research

RES1. Many software researchers advocate rather than investigate. As a result, (a) some advocated concepts are worth less than their advocates believe and (b) there is a shortage of evaluative research to help determine the actual value of new tools and techniques.

There, that’s my two cents’ worth of software engineering fundamental facts. What are yours? I expect, if we can get a dialog going here, that there are a lot of similar facts that I have forgotten—or am not aware of. I’m especially eager to hear what additional facts you can contribute.

And, of course, I realize that some will disagree (perhaps even violently!) with some of the facts I’ve presented. I want to hear about that as well.