Editor: Gerard J. Holzmann
Nimble Research
gholzmann@acm.org

Ends and Means

Gerard J. Holzmann

YOU'VE LIKELY SEEN graphs similar to the one in Figure 1 before. The graph is compelling enough. It matches our suspicion that it's more expensive to fix a bug after it has reached the customer than earlier during development. Often the graph, like Figure 1, has no units on the vertical axis, which should in itself be a red flag that there probably isn't any real data behind it. Sometimes the units are shown, but they seem to vary. Sometimes they range from a modest 1 to 10; sometimes they range more ambitiously from 1 to 100. And sometimes, instead of a linear increase of relative cost, the graph shows an exponential effect.

Since I first saw the graph, I've tried to find the original source for the data behind it, but I have yet to succeed. The problem is that intuition tells us that something like this is likely true and that the details don't matter much. But is that really the case?

If you reflect on the data just a little bit, it's not too hard to poke some holes. Clearly, even the smallest coding mistake can cause huge problems when it slips by testing. Finding it can be difficult, and retesting the fixed system can be expensive, but this certainly isn't true for every type of problem. A defect found and fixed during coding is a fairly routine occurrence and not costlier than a defect found and fixed during design. Quite the opposite is usually true. So what's going on?

Phases

In one way or another, any nontrivial software development project goes through the five phases shown on the horizontal axis in Figure 1. The artifacts built in each phase logically connect and build on each other.

Inevitably, every project must start with the designers figuring out what to build. This is settled in the requirements phase, which should lead to a set of requirements the product must eventually satisfy. Only after you've answered the "what" questions can you move on to the "how" questions. There are normally many ways to achieve a particular goal, and you now must choose one. The design phase should settle questions about the basic system architecture; it should identify the main modules and how they connect and interact. Only then are you ready to start building something, which you can then test to see whether it reliably meets the requirements and complies with the design.

In practice, some iteration is bound to occur over the various phases, as the design space is explored more fully and the real constraints faced in the implementation emerge. At the end of the run, though, it should be possible to pin any relevant activity to one of the five phases.

Defects in the development of a product can be both introduced and removed in any one of these phases. For example, the software requirements will generally be incomplete at the start of the project. They might also be ambiguous and unintentionally contradictory. Similarly, the design might not meet the requirements under off-nominal conditions, and the code might deviate from the design or misinterpret some of the requirements.

Some, but not all, defects are found and removed in the same phase in which they were introduced. Others are found and removed in later phases. The ones that slip through all phases without detection can eventually show up as residual defects in the final product. Figure 2 illustrates this basic flow of defect insertion, removal, and propagation. I first saw such a chart in "Estimating Software Fault Content before Coding."[1]

Figure 2 shows typical defect insertion and removal rates for an industrial software development process. The figure also shows the number of defects per KLOC that are eventually delivered to the customer. It illustrates how defects that are introduced but not caught in a particular development phase are propagated to the next phases.

 0740-7459/17/$33.00 © 2018 IEEE

Clearly, the number that counts the most is the last one: the number of residual defects discovered in the delivered product.

Control Points

I'll get back to the issue of the relative cost of fixing defects shortly. First, let's consider how you can reduce the number of residual defects on the basis of an analysis like this. The process in Figure 2 gives two possible control points. The first aims to reduce the defect insertion rates in each development phase. The second aims to increase the defect removal rates, again in each phase, not just the last one.

To reduce defect insertion rates for a specific process, you must first study precisely what types of issues lead to confusion and error in each phase. For instance, you can make many types of requirements less confusing by formally expressing them in tables or logic formulas. Similarly, you can make designs more robust by formalizing them as simulation models or as executable state machines that can be checked mechanically for different types of properties.

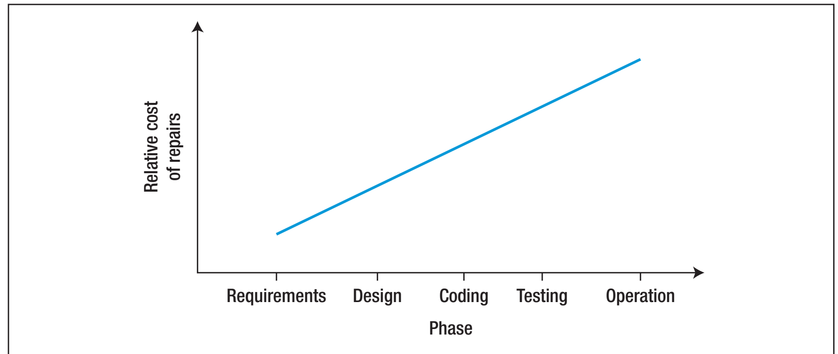

FIGURE 1. A popular graph suggesting that the relative cost of fixing defects increases throughout software development. The fact that there are no units on the vertical axis should be a red flag that there probably isn't any real data behind the graph.

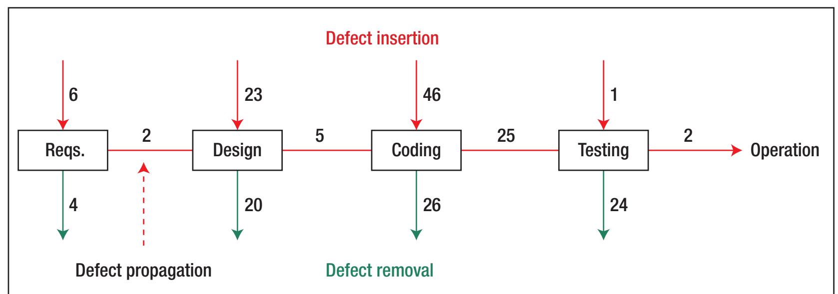

FIGURE 2. Software defects can be introduced and removed in any development phase. (The numbers indicate the number of defects per KLOC eventually produced.) Defects introduced but not removed in a given phase are propagated downstream. The final number of propagated defects determines the residual-defect rate, which is typically between one and 10 defects per KLOC.[2]

One key step in this process is to replace the ubiquitous informal development processes that are aptly described as "PowerPoint engineering" with more robust tool-based processes that can help you create machine-readable, machine-checkable artifacts in each development phase. Requirements stated in logic can be checked for consistency and completeness and can be used to generate test suites. None of this is possible when requirements remain imprisoned in PowerPoint slides, spreadsheets, or Word documents. Similarly, design models that are formalized as automata can be rigorously checked against requirements expressed in logic before any code is written, preventing unpleasant surprises much later in the process.

In the coding phase, you can reduce defect insertion rates by using a sensible coding standard with machine-checkable rules, and you can increase defect removal rates by introducing strong static-source-code-analysis tools. For example, before committing code, you can ask developers to complete two routine checks. The first ensures that the code can be compiled without any compilation warnings, with warnings enabled at the highest possible level the compiler supports. The second check is that the code can be scanned by a good source code analyzer without generating warnings, not even invalid ones. This might seem like a hassle, but the top developers in a team will usually quickly embrace this routine as part of the creation of a culture of craftsmanship in software development.

The Cost of Defects

Collecting the metrics that are needed to populate Figure 2 requires tracking a few more things than you might otherwise do, but it's not a

Table 1. Root-cause analysis of defects.*

Phase in which a defect was introduced	Phase in which a defect was detected					
	Requirements	Design	Coding	Testing	Operation	Total
Requirements	4	1	—	1	—	6
Design	—	19	1	3	—	23
Coding	—	—	25	20	1	46
Testing	—	—	—	—	1	1
Operation	—	—	—	—	—	—
Total	4	20	26	24	2	76

** The numbers indicate the number of defects per KLOC.*

big deviation from the normal process. This can be different for the next step, but it's especially this next step that can help reduce costs in the long run.

Once you find a defect, it's not enough to fix it and call it a day. For each major defect you find, you should understand in which phase the defect was introduced and why it wasn't detected sooner. After all, every defect in the code is also a defect in the process that produced that code. A defect found in testing could originate in a poorly worded requirement, in a design flaw, in a coding mistake, or even from a bug introduced in the testing process itself. It's important to know what the cause is so that you can fine-tune the process to prevent recurrences.

With the data you can gather from such a root cause analysis, you can construct something like Table 1. The cells in the lower-left triangle of the table are empty because those cells correspond to defects that would be found before they're introduced—a technique we have yet to master. The numbers in each row add up to the defect insertion numbers in Figure 1. The numbers in each column add up to the defect removal totals.

The Key Metrics

A table like Table 1 lets you compute two key metrics. The first is the distance a defect travels from its insertion point to its detection point. The longer that distance is, the more harm the defect can do and, yes, the greater the repair cost can be, as hinted at in the mythical graph from Figure 1. In Table 1, the greatest distance spans three phases: a requirements defect that wasn't caught until testing. The worst case would be a requirements defect that wasn't found until the software was delivered to the customer. The longer the span, the more rework that might be needed to fix it.

The second key metric is the ratio of the defect removal and insertion rates for each phase. A low ratio indicates a poorly performing process that might need stricter controls. For Table 1, the ratios are 4/6 (0.67) for requirements, 20/23 (0.87) for design, 26/46 (0.57) for coding, and 24/1 (24.00) for testing. This points the finger at the coding phase as performing the worst. Possibly a stronger coding standard, with mechanically verifiable compliance, could help turn things around. Similarly, increased use of assertions to create more robust self-checking code[3] and targeted training aimed at avoiding software risk can help achieve a better ratio.

The intent of the well-known graph in Figure 1 is a good one, although the details can be a bit more subtle than what it purports to show. The basic lesson from the slightly more detailed analysis I gave in Figure 2 and Table 1 remains the same, though: Software defects should be caught as early as possible, and unless you're routinely producing zero-defect code, you can almost certainly do a better job of that.

References

1. S.G. Eick et al., "Estimating Software Fault Content before Coding," *Proc. 15th Int'l Conf. Software Eng.* (ICSE 92), 1992, pp. 59–65.

2. C. Jones, *Applied Software Measurement*, McGraw-Hill, 1991, p. 177.
3. G. Kudrjavets, N. Nagappan, and T. Ball, "Assessing the Relationship between Software Assertions and Faults: An Empirical Investigation," *Proc. 2006 Int'l Symp. Software Reliability Eng.*, 2006, pp. 204–212.

ABOUT THE AUTHOR

GERARD J. HOLZMANN works on developing stronger methods for the design and analysis of safety-critical software as a consultant and researcher at Nimble Research. Contact him at gholzmann@acm.org.