# Software Quality Management

High quality software is dependable, durable, fit for purpose, well supported, portable and easily integrated with other tools. There are numerous routes to improvement of software quality: through the product (direct) and through the process (indirect).

## 0.1 Quality and the Pursuit of Defects

Quality assurance is not simple pursuit of defects. Defect checking (through testing, for example) is part of the story but is a means to an end, not the end itself. If you are focused on product quality then:

- You tend to produce components with fewer defects
- so you have more time in your schedule for things like usability and compatibility checking.

If you don't focus on product quality then:

- You tend to produce components with more (hidden) defects
- so you have to spend more time fixing these (late)
- so you have little time for anything else
- so you produce minimum quality software **even though you put huge amounts of effort into defect checking**.

## 0.2 The Cost of Quality

The reason we can't have uniformly high quality software is that it takes time and money to do the following things:

- Spot a problem.
- Isolate its source.
- Connect it to the real cause.
- Fixing the requirements, design and code.
- Inspecting the fixes.
- Testing the fix for this problem.

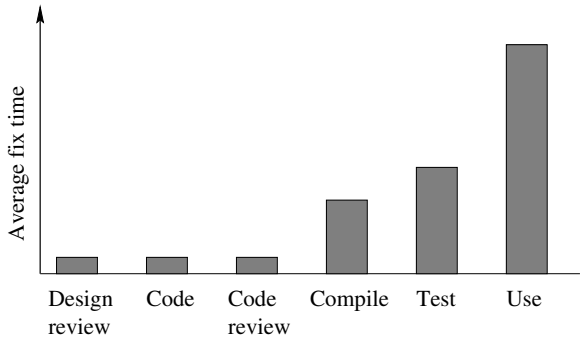


Figure 1: Cost of quality delays

- Testing the fix hasn't caused new problems.
- Changing the documentation.

How much time we spend on these activities may be influenced by when we discover that we have a problem. Figure 1 gives a stereotypical example of this, where the average fix time for defects discovered in early design stages (such as design review, coding and code review) may be orders of magnitude less than that for defects discovered in later design stages (such as testing and use). This observation does not imply that all individual defects would take more time to fix the later they are discovered (since the defects found early are by definition not those found later) but it does accord with the intuition that the longer a defect persists the more it becomes entwined with the rest of the design and the harder it is to disentangle.

So we would like to have reasonably high quality components before testing and catch difficult defects early, but there is not necessarily a correlation between ease of identification of a defect and its ease of correction. Usually we are not able to know whether we have found all the "significant" defects so the decision about whether to dig deeper at any given design stage is determined by experience and available resources. For quality to improve, the cost of all of this must be less than the penalty for not fixing the problem *plus* the revenue lost by delaying release of the product. This is why software sometimes is released before known defects are removed.

## 0.3 Cumulative Quality Improvement

To appreciate why we need to worry about the timing of quality improvements and their cumulative effect, consider the following example. Humphrey estimates that experienced software engineers normally inject 100 or more defects per thousand lines of code (KLOC). Perhaps half of these are detected automatically (*e.g.* by the compiler), so a 50 KLOC program probably contains around 2500 defects to find (semi-)manually. Suppose we need about five hours to find each of these defects by testing. That's over 20000 person-hours for the whole

program, which is prohibitively expensive. Now suppose that we could find up to (say) 70% of these defects by first doing code inspection at a cost of 0.5 person-hours per defect so the first 1750 defects could take 875 hours; then we only have 750 to find in testing at (say) 8 hours each. That's less than 7000 hours in total - about a third of our earlier cost estimate.

Generalising from this example, the equation below is a simple model of this sort of sequential defect removal:

$$y(N) = \frac{r(N)}{r(N) + e(N)}$$

where:

- $y(N)$ is fraction of defects removed in step $N$
- $r(N)$ is the number of defects removed at step $N$.
- $e(N)$ is the number of defects escaping at step $N$.

The difficulty with this equation is that we can only estimate $e(N)$ as a function of $e(1), \ldots, e(N-1)$. That is, we can learn more about the defects escaping as we gain experience with our software but we can't learn from experience we haven't had. We don't even know that $e(N+1)$ will necessarily be smaller than $e(N)$, since defects can be injected by changes made at step $N$, although an objective quality management is to make this so.

Imagine that we are skilled enough to be removing defects without introducing significant numbers of new defects. Then we need to consider the sensitivity of quality to the effectiveness of our defect removal over the series of quality improvement stages. Suppose you have 1000 KLOC with an average of 100 defects per KLOC. That's 100000 defects to find. Consider the following two strategies for defect removal:

**Scenario 1** :

- You have an inspection process which finds 75% of these, leaving 25000 to find in test.
- You then use 4 levels of test, each trapping 50% of remaining defects. That leaves 1562 defects in the final code.

**Scenario 2** :

- Your inspection process only finds 50% of defects, leaving 50000 to find in test.
- The same 4 levels of test each trap 50% of remaining defects. That leaves 3125 defects in the final code.

So a 33% drop in yield in inspection between scenarios 1 and 2 caused a doubling in the number of defects remaining in the code.

## 0.4 Prevention versus Detection

Is it better to spend money on preventing errors entering our code or on detecting those errors we have introduced? The answer in practice is likely to be a balancing of effort but the two simple equations below give an understanding of the basic forces which we must balance. Assuming we start with no defects then the following equation is a simple model of the probability that we will continue our code development without introducing errors.

$$P_i = (1 - p)^i$$

where:

- $p$ is the probability of injecting a defect at a stage.
- $i$ is the number of stages.
- $P$ is the probability of a defect-free product at stage $i$.

If we have ten design stages and the probability of introducing a defect is 0.01 at each stage then the probability of a defect-free product at the end (stage 10) is $0.904 = (1 - 0.01)^{10}$. But suppose that at one of the stages we have shoddy workmanship which raises the probability of injecting a defect at that stage to 0.5. The probability of a defect-free product at stage 10 is then $0.4057 = 0.5 * (1 - 0.01)^9$. So a high probability of fault injection in one step radically drops the overall probability of freedom from defects. This is why cleanrooms are so clean.

Now let us look at what happens if we had a lapse in performance when removing defects. The equation below is a simple model of defect removal over a series of design stages.

$$R_i = N * (1 - y)^i$$

where:

- $N$ is the initial number of defects.
- $y$ the fraction of defects removed per stage.
- $i$ is the number of stages.
- $R_i$ is the number of defects remaining at stage $i$.

If we have 100000 defects initially and remove 80% of them at each of five design stages then the number of defects remaining after stage 5 is $32 = 100000 * (1 - 0.8)^5$. Had one of our five defect removal operations been half as effective then the number of defects remaining becomes $96 = 100000 * (1 - 0.4) * (1 - 0.8)^4$.

So dropping a lot lower on one stage of a high quality defect removal process has a small effect on overall yield. This (combined with result from our first model) is why it is better to be defect-free than to rely on fixes.

A final word on yield management. If we had no resource limitations then an 80-40 test-inspection yield is no different from a 40-80 yield. But test defect correction typically involves more labour than inspection defect correction, so it costs more and the extra labour means more opportunities for defect injection. So a useful guiding principle is to manage for maximum return for minimum cost and, if in doubt, attempt to maximise on early design stages.

## 0.5 Algorithmic Cost Modelling

Various models exist for predicting cost of software production based on estimates of key parameters (*e.g.* number of programmers). A popular model the COCOMO model, which in its simplest form is:

$$E = C * P^s * M$$

where:

- $P$ is a measure of product size (*e.g. K Delivered Source Instructions*)
- $C$ is a complexity factor.
- $s$ is an exponent (usually close to 1).
- $M$ is a multiplier to account for project stages.
- $E$ is the estimated effort (*e.g.* in person-months).

For the purposes of example we shall ignore the multiplier, $M$, for now so $E = C * P^s$. Then we need to estimate $C$ and $s$ for different types of project. Here we are looking for rough (order of magnitude) estimates of effort - for example:

**Simple** ($E = 2.4 * P^{1.05}$) : A well understood application developed by a small team.

**Intermediate** ($E = 3.0 * P^{1.12}$) : A more complex project for which team members have limited experience of related systems.

**Embedded** ($E = 3.6 * P^{1.20}$) : A complex project in which the software is part of a complex of hardware, software, regulations and operational constraints.

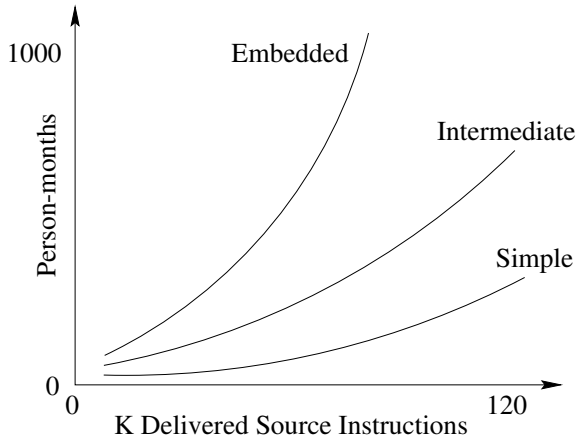


Figure 2: Variation of effort with product size in different models

Figure 2 shows the effect of these different models on our estimates of effort based on product size.

The basic examples above don't use the multiplier, $M$. If we want to use it then it often is estimated as a function of key attributes of the problem. The effect of this is to modify the basic estimate. Which key attributes are chosen, and how they modify the basic equation, depends on experience in an area of application. Typical types of attributes are:

- Product attributes (*e.g.* reliability).
- Computer attributes (*e.g.* memory constraints).
- Personnel attributes (*e.g.* programming language experience).
- Project attributes (*e.g.* project development schedule).

Let us consider a brief illustrative example of this sort of modification. Suppose we have a basic estimate is 1216 person-months. We then identify five key attributes and give multipliers for them according to their strength of influence. This particular project is demanding, requiring software of both high reliability and complexity; it places tight constraints on memory use; we can't make much use of tool support in development; and the project schedule is accelerated.

| Attribute | Magnitude | Multiplier |
|---|---|---|
| Reliability | V high | 1.4 |
| Complexity | V high | 1.3 |
| Memory constraint | High | 1.2 |
| Tool use | Low | 1.1 |
| Schedule | Accelerated | 1.23 |

This gives us the new, much higher, estimate: $1216 * 1.4 * 1.3 * 1.2 * 1.1 * 1.23 = 3593$

Suppose instead that we anticipate that the project will be less stressful, with low complexity and reliability requirements; no memory use constraints; a lot of work done using tool support; and a normal project schedule.

| Attribute | Magnitude | Multiplier |
|---|---|---|
| Reliability | V low | 0.75 |
| Complexity | V low | 0.7 |
| Memory constraint | None | 1 |
| Tool use | High | 0.9 |
| Schedule | Normal | 1 |

Now we can reduce our effort estimate to $1216 * 0.75 * 0.7 * 1 * 0.9 * 1 = 575$

Notice that the predictions of models like those above are approximate and sensitive to small changes in parameters so you should perform a sensitivity analysis to changes in parameter values. Also you must remember that your initial estimates are likely to be wrong. Check them when you have more experience.

Predictions may constrain what actually happens. If you base your project plan on an estimate of effort which is too low it may be difficult retrospectively to increase the budgeted amount of effort, so you will have to cut quality to stay within your effort target. Your prediction may turn out to be accurate but at the cost of bad engineering.

Much of this lecture taken from Watts Humphrey 1995 *A Discipline for Software Engineering* and from Sommerville 1996 *Software Engineering*.

## 0.6 Exercise

Cleanroom software development attempts to maintain an "ultra-clean" atmosphere by applying: formal specification; incremental development (perhaps partitioned by modules); structured programming and stepwise refinement (so both structural elements and design choices are constrained); static verification (*e.g.* using proof of correctness) and statistical testing of the integrated system. Discuss how this might work. You will need to chase up a few references (in the library or on the Web) - no pointers to these are supplied here because you should practice this sort of literature search.