Quality Management

High quality software is dependable, durable, fit for purpose, well supported, portable, easily integrated with other tools.

There are numerous routes to improvement of software quality: through the product (direct) and through the process (indirect).

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

Not Just Absence of Defects

We have looked a lot at defect checking (*e.g.* testing) but these are 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**.

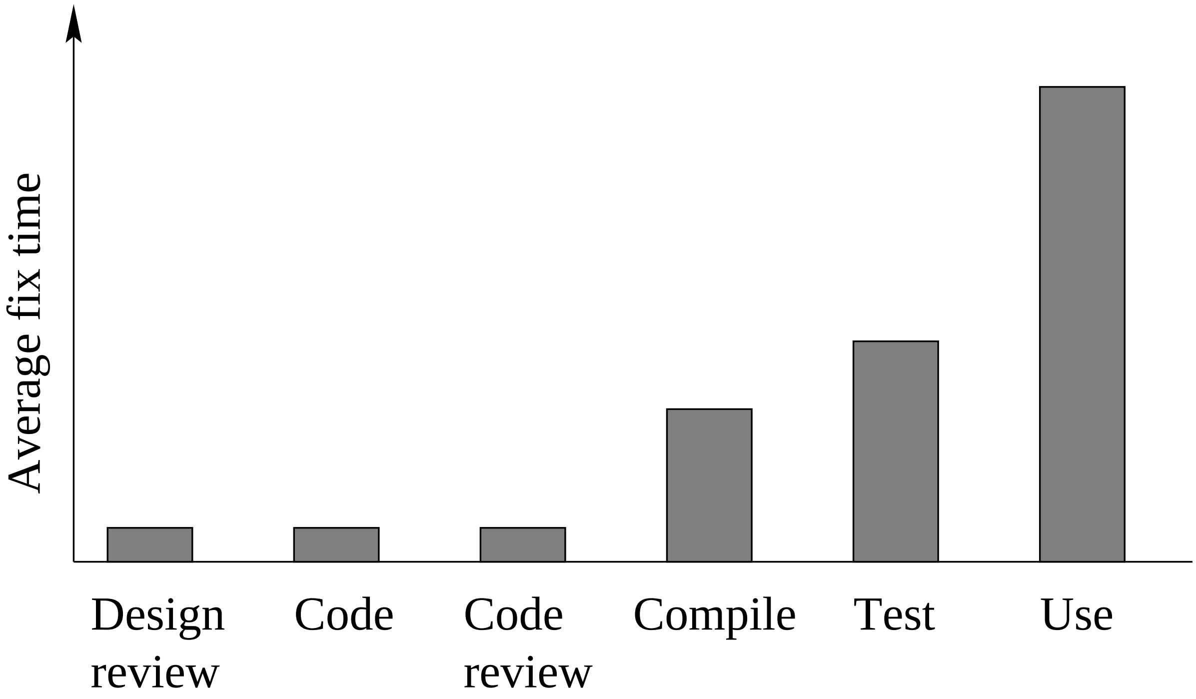
The Cost of Quality

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

For quality to improve, the cost of this must be less than the penalty for not fixing the problem *plus* the revenue lost by delaying release of the product.

Quality Delays are Expensive



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.

It Matters How you Look

Humphrey estimates that experienced software engineers normally inject 100 or more defects per 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 - **bad news**.

But inspection may be able to find up to (say) 70% of these defects in 0.5 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 - **better news**.

Cumulative Quality Improvement

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

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 $e(1), \ldots, e(N-1)$.

Notice that $e(N)$ can increase if defects are injected by changes.

Effect of Change in Performance

Suppose you have 1000 KLOC with an average of 100 defects per KLOC. That's 100000 defects to find.

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 caused a doubling in the number of defects.

Sensitivity to Defect Injection

Assuming we start with no defects:

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

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.

$$0.904 = (1 - 0.01)^{10}$$
$$0.4057 = 0.5 * (1 - 0.01)^9$$

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.

Sensitivity to Defect Removal

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

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.

$$32 = 100000 * (1 - 0.8)^5$$
$$96 = 100000 * (1 - 0.4) * (1 - 0.8)^4$$

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 previous slide) is why it is better to be defect-free than to rely on fixes.

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 manage for maximum return for minimum cost and, if in doubt, attempt to maximise on early design stages.

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).

Basic COCOMO Model Examples

We ignore the multiplier, M, so $E = C * P^s$

Then we estimate C and s for different types of project:

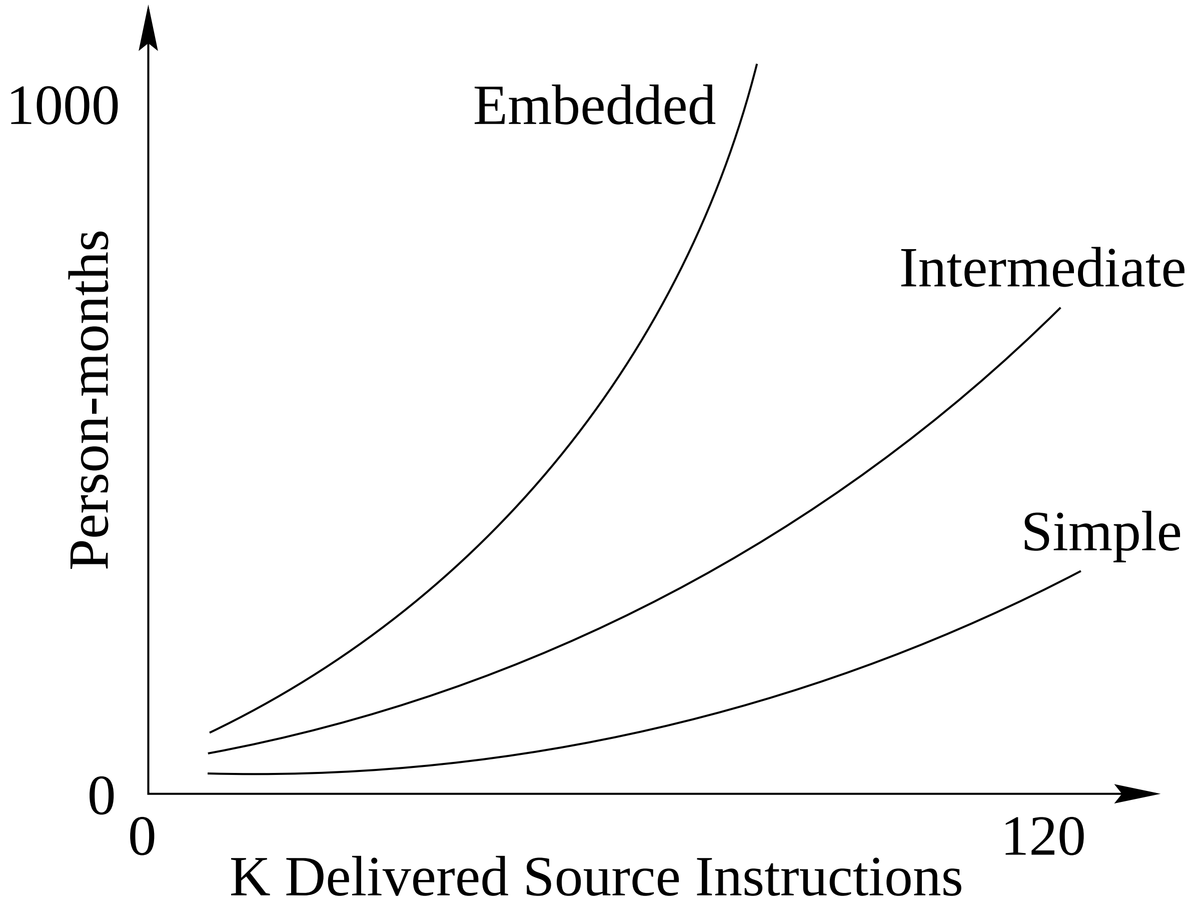
Simple $(E = 2.4 * P^{1.05})$: A well understood application developed by a small team.

Intermediate $(E = 3.0 * P^{1.12})$: A more complex project for which tem 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.

We are looking for rough (order of magnitude) estimates of effort.

Behaviour of the Basic Examples



Extending the COCOMO Model

The basic examples didn'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 is to modify the basic estimate.

Where do the attributes come from?

- 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).

COCOMO Multiplier Example

If the basic estimate is 1216 person-months.

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

Now $1216 * 1.4 * 1.3 * 1.2 * 1.1 * 1.23 = 3593$

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 $1216 * 0.75 * 0.7 * 1 * 0.9 * 1 = 575$

Model with Care

The predictions of models are approximate and sensitive to small changes in parameters so perform a sensitivity analysis to changes in parameter values.

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.