Follow the money

An Introduction to Cost of Quality for Software

Robert Bentall, 2014

Motivation

- Software is expensive to produce:
  - How do you know that your money is being well spent?
  - Cost of Quality helps you understand where your money is going.

If we want to make ourselves and teams more productive, it helps to know where we are wasting money.

Cost of Quality is a powerful technique for tackling this problem, and it's been around for a very long time.

What we will cover:

- Introduction to COQ
- Some examples
- How to start applying it for yourself

This talk will provide an introduction to the concepts, provide you with some examples of how it can be used, and will provide some suggestions on how to capture and analyse the data for your own project.

Cost of Quality defined

- *COQ is the sum of all costs that would disappear if there were no quality problems – Joseph Juran*

- If a production process is perfect:
  - No inspections or testing
  - No rework
  - COQ = 0

It represents the costs of activities associated with trying to protect the customer from our mistakes.

If we could guarantee that our production was defect free, we would not need to test, inspect etc.

COQ is really a consequence of the fact that we are human.

The concept was first discussed by Joseph Juran and Armand Feigenbaum in the 1950s. Other contributors to the subject are authors such as Philip Crosby and W Edwards Deming.

It has had a long history in manufacturing, and has helped organisations improve profitability.

At present it is not as widely used in software as it should be.

Cost of Quality basics

Costs of control *(Costs of conformance)*	Prevention costs
	Appraisal costs
Costs of failure of control *(Costs of non-conformance)*	Internal failure costs
	External failure costs

"Cost of quality" is the set of costs we incur because our service delivery or production process is imperfect.

These categories were originally defined by Armand Feigenbaum in the 1950s.

Prevention costs include things such as training, investment in infrastructure etc., and other interventions which are typically amortised over multiple projects.

Appraisal costs include things such as testing, formal technical reviews (FTRs), code reviews etc.

Internal failure costs are incurred when a defect is seen before the product reaches the customer (eg my new feature breaks another part of the system)

External failure costs are incurred when a defect reaches the customer (SWATS raised by the customer)

Note that this is the simplest possible model, and in reality one requires many more categories to describe the whole process of delivering a software product or service (see for example Capers Jones, Software Engineering Best Practices)

COQ categories – Costs of control

- Prevention:
  - Training/coaching
  - Effective project management
  - Effective requirements gathering, design etc.
  - Investment in tooling
- Appraisal:
  - Reviews/inspections (at all stages)
  - Static analysis tools
  - Testing

Probably the most powerful techniques for preventing problems defects in software are training and coaching. They give individual engineers and managers the tools they need to create high-quality software.

Good project management creates the space for high-quality work to take place – reasonable expectations can be set, and engineers are less likely to cut corners to give the appearance of progress.

And of course effective requirements gathering and design enable us to build the right thing in the most efficient/effective way.

Investment in tooling helps – but is arguably only a second-order effect.

You need a range of appraisal techniques to catch different types of defect:
Top of the list for effective appraisal techniques are reviews or inspections – their effectiveness has been known for a long time.
A wide range of static analysis tools are available to help identify some types of defect – but they can sometimes have poor precision.
The most widely used technique for appraisal is arguably testing.

COQ categories – Costs of failure of control

- Internal failure:
  - Bug fixes
  - Regression testing
  - Project delay costs
- External failure:
  - Bug fixes/regression testing
  - Tech support calls
  - Customer downtime
  - Lost sales
  - Lost goodwill

When we think about failure costs, most engineers will think about bug fixes.

However there are many other – sometimes less obvious – costs that are also incurred.

And once defects get out to a customer, the costs escalate drastically.

Relative costs of fixing defects depending on where they are introduced (McConnell)

		Time detected				
		Requirements	**Architecture**	**Construction**	**System Test**	**Post-Release**
Time introduced	**Requirements**	1	3	5-10	10	10-100
	Architecture	-	1	10	15	25-100
	Construction	-	-	1	10	10-25

Various estimates exist for the relative cost of defect removal, but the important point is that removal closest to the point of insertion is cheapest.

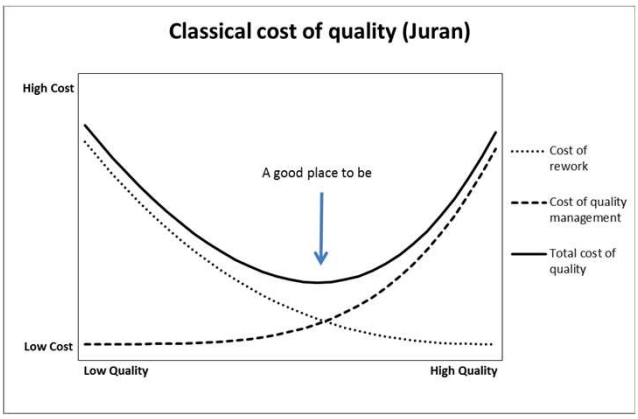
This table is from McConnell (Code Complete 2nd edition, Chapter 3 – Measure Twice, Cut Once, p29).

It is adapted from a number of sources, but the key point is that it pays to catch stuff early.

To bring home the point a little, think about the bottom line – a defect injected in construction takes 10-25 times as long to fix when the product is out in the wild.

Or to put it another way, it's the difference between:
- I can fix it this morning before my 11 O'clock meeting, and
- It will take me most of this week to fix.

If we get it right:



This is the classical COQ model borrowed from manufacturing.

It captures the intuition that both prevention and rework are subject to laws of diminishing returns.

At the left hand end of the graph, where we don't invest in any prevention or appraisal, our rework cost is high.
As we invest in prevention and appraisal, rework costs drop up to the point where it becomes cheaper to fix problems rather than prevent them.

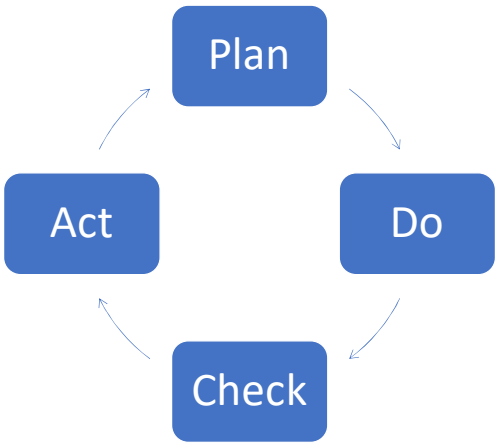
We want to ensure that we minimise the total quality costs (prevention + appraisal + failure) for a given level of quality.

That implies investing in defect prevention + effective appraisal so we catch stuff as early as possible.

Later on, Juran described an alternative version of this graph for systems that had a minimum COQ at the far right hand end of the graph, when approaching zero defects.

I haven't seen one of these drawn for software products yet, although there is quite a lot of data about the relative effectiveness of different defect prevention and removal strategies.

How do you apply the principle?



So how do you apply the principle?

This is sometimes called the Shewhart or Deming cycle.

It captures the notion that performance improvement is an iterative process:
- First we decide what to focus on,
- Then we carry out the activity – recording data as we go,
- Then we analyse the data we've captured to understand what's going on,
- Then we can act to change the process.

This doesn't have to be an onerous activity – done correctly, it will be a small investment with a significant return.

Remember, COQ is the sum of costs that would disappear if our production process was perfect.

Tracking and analysing your COQ allows you to figure out how much of your development cost is due to the fact that we make mistakes.

And the distribution of costs helps you understand the root causes of failure were.

It is another example of a powerful technique that started life in large-scale manufacturing plants which can be easily adapted for use by individual software developers or software development teams.

And because it was developed in the 50s, it doesn't require vast quantities of data to be collected in order to be applied.

So – if you want to apply this for yourself, or for your team:
- the tooling costs are minimal – a stopwatch and a spreadsheet.
- the analysis is simple – basic charts and calculations that can be done with a spreadsheet.

Plan - Identify scope

- Organisational:
  - Individual – tasks you've picked up?
  - Individual or team – repetitive operations?
  - Team – all tasks to deliver a user story?
- Temporal:
  - Only stuff that happens on my user story?
  - Up to end of beta test?
  - 1 year after release?

COQ as a technique can be applied at any level of scale, from the macro to the micro.

Originally, it was an accounting technique – quality experts would review a company's general ledger and identify which category a cost should be put into.

One can still do that, but it can also be applied:
- at the level of an individual team working on a user story (which will comprise multiple tasks)
- at the level of an individual who wants to understand their own performance.
- at the level of an operation that you have to carry out repeatedly.

Most of the software literature that deals with COQ for software looks at it at the macro level (program or project). The only author I know of that treats it at the level of the individual developer or team was Watts Humphrey (PSP, TSP)

There are really two dimensions of scope that you need to consider, organisational and temporal.

- The smaller the scope, the easier it is to manage the data.
- However the analysis is then less powerful

You need to be aware of the "pillow problem" – just moving the problem around the system.

Once you've identified the scope, that defines the task list you need to consider.

Keeping scope small is helpful – once you get into trying to track costs of testing and rework across multiple teams and after release, it gets more complex.

You can gain a lot of insight from simple models.

Do - Record effort as you go:

- Most important thing is for your measurements to be internally consistent
- Measurements don't have to be perfect (to nearest second, net of all interrupts etc.)
- They provide an *estimate* of the cost of delivering a feature/service etc.
- *Don't expect them to be the same as the numbers on a timesheet*

Once you've completed the plan, you can move onto capturing data on your performance.

As you complete tasks, record your effort (generally don't want to use total elapsed time, but "task time" as a proxy for cost).

The most important thing is for your measurements to be internally consistent.

Obviously you need to be careful when combining sets of data from different teams or individuals without checking that all parties are using the same definitions.

Check – Categorise your data

- Assign each task to a category:
  - Prevention
  - Appraisal
  - Internal failure
  - External failure

Once you've gathered some data, you can start to analyse.

First of all, you assign each task to a category.

Some tasks will be a combination of categories
If your tasks are "small" this isn't a problem
Need a "good enough" approximation
Can identify your own categories to suit your needs.

Check – Analyse your data

- Is there a stand-out category?
- Failure costs due to mistakes made earlier in the project?
- Failure costs due to mistakes made years ago?
- Which defects made it out to my customers?
- Why did the failures occur?
- How do I stop it happening in the future?

Here is where it gets interesting.

You can start to look for patterns in the data.

And causes for those patterns.

Act

- What small change can I make *now*?
- What could we have done differently?
- What can my team change for the next user story?
- What can the project or program change?
- How do we build a business case?

ACT – you can start to change stuff.

There are many ways to make small incremental improvements to the way you or your team is working.

And using cost data is enormously powerful – money is the language of management (Phil Crosby).

Worked example – Small feature

Increment	Task	Plan Effort/hrs	COQ category
Increment 1	Construction	10	N/A
	Code Review	5	Appraisal
	Post-review Rework	5	Internal Failure
Increment 2	Construction	12	N/A
	Code Review	6	Appraisal
	Post-review Rework	6	Internal Failure
Increment 3	Construction	6	N/A
	Code Review	3	Appraisal
	Post-review Rework	3	Internal Failure
	Total effort	**56**	

Total appraisal = 14 hrs (25 %), total internal failure cost = 14 hrs (25 %), so COQ = 28 hrs or 50 %

So we are going to start with a worked example. Imagine a small user story that comprises three “feature increments”.

We write a short plan, and decide that each feature increment will comprise three tasks:
- construction.
- code review.
- rework after code review.

After a little more thinking, we put some estimates together for each feature increment:
- we estimate construction effort,
- we assume that code reviews will take c. 50 % of construction cost,
- we assume that rework will take c. 50 % of construction cost.

This gives the following task list, with a planned cost of quality at 50 %.

Actuals after completing all 3 increments.

Feature	Task	Plan	Actual	COQ category
Increment 1	Construction	10	12	N/A
	Code Review	5	4	Appraisal
	Post-review Rework	5	7	Internal Failure
	Fix broken build	0	3	Internal Failure
Increment 2	Construction	12	11	N/A
	Code Review	6	7	Appraisal
	Found bugs in third-party library	0	2	Appraisal
	Bug from increment 1	0	2	Internal Failure
	Post-review Rework	6	10	Internal Failure
Increment 3	Construction	6	8	N/A
	Code Review	3	2	Appraisal
	Fix broken build	0	3	Internal Failure
	Bug from increment 1	0	6	Internal Failure
	Post-review Rework	3	4	Internal Failure
	Total	**56**	**81**	

Now imagine we've completed the work – here's the same task list with two changes:
- actuals filled in for each line.
- new items in red are unplanned work – stuff we realised that we needed to do as we went along,.
- in this case, these are all defect-related – broken builds, bugs in third-party libraries, bugs from previous increments.

Cost of quality breakdown – plan vs actuals

COQ Category	Plan	Actual	Plan vs actual %
Appraisal	14	15	7%
Internal Failure	14	35	-150%
Total COQ	28	50	-79%

We expected COQ to be 28 hrs. In actual fact, it was 50 hours, driven by failure costs.

The first point to note that our actual COQ is a best-case number. Once released, the only thing that will happen is that COQ goes up – as customers use the software and find defects.

Comparing our plan vs actual COQ – our appraisal costs were in line with expectations, but our failure costs were way higher than we planned.

Why?

Failure costs – ordered by size

Task	Plan	Actual
Post-review Rework (increment 2)	6	10
Post-review Rework (increment 1)	5	7
Bug from increment 1 (increment 3)	0	6
Post-review Rework (increment 3)	3	4
Fix broken build (increment 1)	0	3
Fix broken build	0	3
Found bug in third-party library (increment 2)	0	2
Bug from increment 1 (increment 2)	0	2

To figure out what's going on, we need to drill into the failure costs in more detail.

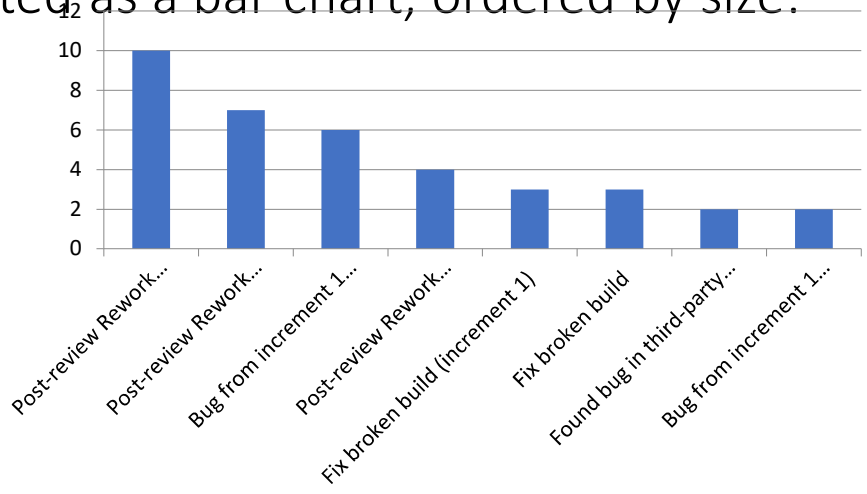
As a table, ordered by actual costs descending, you can see that the most expensive item is rework from increment 2. It's also twice what we expected.

Why? What happened in increment 2 that meant we had so much rework?

Looking at the bottom half of the table, you can see that most of the "unplanned" failures were relatively small in size, although they still accounted for about 43 % of the total failure cost.

Although these data are fictional, the numbers aren't unrealistic.

Plotted as a bar chart, ordered by size:



Plotting the data can also help – it makes it much more obvious where the significant costs are.

What can you change?

- Analysing your data will help identify significant cost drivers
- The real challenge is identifying things you can change
- What is the strongest factor?
  - Engineer skill?
  - Reliability of infrastructure?
  - Third-party dependencies?
  - Etc...

This was artificial data to give a flavour of how you do the analysis.

However the important point is that we had reasonably granular data – so we can actually drill into the costs.

If you don't have much detail, the cost of the project is just a big black box.

Three examples...

- How much time should I budget for code reviews?
- How much is legacy technical debt costing the project?
- How much are upstream defects costing my team?

I want to emphasise that COQ as a technique is enormously versatile, and doesn't need to be expensive to apply.

So the examples we are going to consider have been selected because they are small scale and demonstrate different ways in which the technique can be applied.

The first of these considers how a developer and reviewer might use COQ to plan their work more effectively.

Effective code reviews are enormously powerful, but you need to ensure you spend enough time to produce a high yield review.

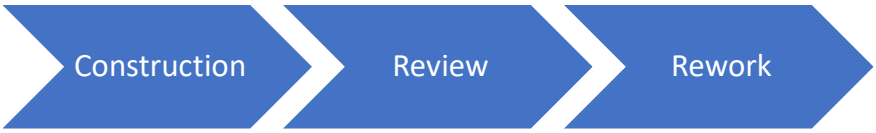
The second of these looks at how we can quantify the effect of previous decisions on our current project – legacy technical debt.

This allows us to start to factor this into our plans, and to construct a business case for re-engineering or retiring the most expensive components in the system.

The third example is really a variation on the second.

We all consume components from other teams within the system. How much do defects from upstream components cost you when developing features?

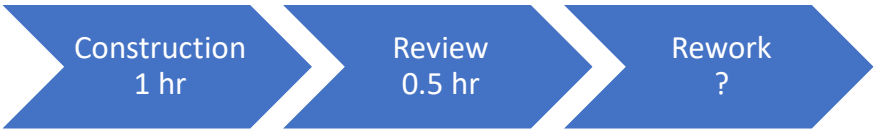
How much time should I budget for code reviews?



Here's the workflow that we had in the worked example a few slides earlier.

- Code reviews are an appraisal cost.
- Imagine your colleague asks you to do a review of some work they've submitted.
- How long should you budget?
- How much rework would you expect to find?

Solution 1 - Phase ratios



The simplest solution to this problem is to use phase ratios.

This is how the Personal Software Process apportions effort between different types of activity for a small piece of work.

For each hour of “construction activity,” you would look to spend about half an hour on review activity.

How much rework would you expect?

Answer – it varies... Potentially by quite a lot.

With practice (and careful data collection) you can understand the volume of rework you are likely to see.

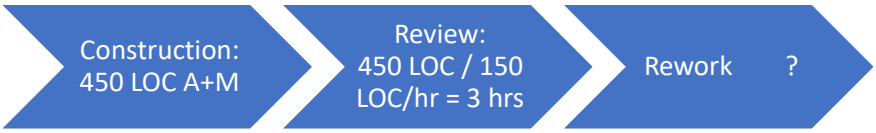
Analysing the type of rework generated by construction and caught by review can help you figure out how to prevent it in the first place.

And tweaking how you work can make it easier to catch defects before they escape to subsequent steps in the process (in this case, the reviewer).

Doing this helps make your work more predictable (and so easier to manage).

Don’t be surprised if (at first) rework is the same order of magnitude as construction.

Solution 2 – Volume of code added/modified



An alternative solution is to use the volume of code added/modified.

This works well if you don't know (or can't find out) how long your colleague spent on construction.

Assume you your colleague has created or modified 450 LOC (for some sensible measure of LOC)
- Estimate review effort as review size of code created or modified / rate
- Industry benchmarks suggest 150-200 LOC / hr is slow enough to produce a high yield review.
- So you would plan to spend 2-3 hours reviewing their work

The same remarks on rework apply as in the previous slide.

Summary – estimating review effort

- A single “increment of functionality” (small)
- Short timescales (less than a week)
- Relatively easy to measure construction, appraisal and failure costs.
- Analysing the defects found can provide interesting insights
- Hence – a very good place to start.

Example – how much is legacy technical debt costing the project?

I analysed the data for a project I worked on in 2013.

- In this case, the scope was:
  - tasks that I worked on
  - during the project
  - 27 discrete tasks
  - Total of 148 task-hours effort

Analysis

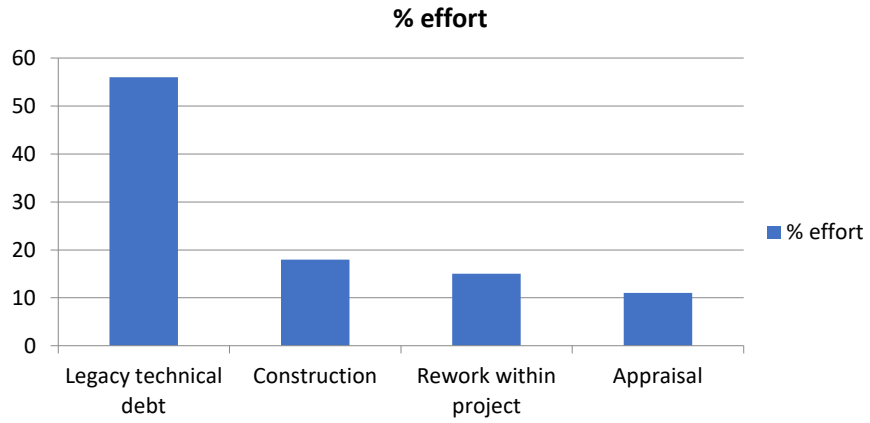
COQ Group	COQ Subgroup	Task Hours	%
Costs of construction	N/A	26	18
Costs of control	Appraisal	16	11
Costs of failure of control	Rework from project	23	15
	Rework from previous projects	83	56
	Totals	148	100

We need to be careful when assuming that this was a “typical” distribution.

In this case, I split rework into two categories:
- from this project (“we made a mistake last week”)
- from previous projects (“we made a mistake a few years ago”).

The top three rows are of the same order of magnitude – and comprise approximately half of the total cost.
Half of the cost was due to remedial action

Or to put it another way.....



A bar chart ordered by descending size makes the point even more forcefully.

The left-most bar constitutes 83 task hours - or approximately 4 weeks of engineer effort.

What effect does that have on your project planning?

Maybe you know that for each hour coding you need to spend half an hour on appraisal and an hour on rework. But do you expect to have to spend three hours fixing legacy problems in the product?

This is like driving your car with the handbrake on.

Summary – effect of technical debt

- Legacy issues can be like icebergs – 9/10ths hidden.
- Rework to fix these tends to be unpredictable and expensive.
- Tracking costs associated with this sort of activity is very valuable:
  - We can understand the impact of earlier decisions
  - and try to make better ones in the future.

Example - how much are upstream defects costing my team?

- Sometimes poor quality in components we consume can hamper us.
- And if we create poor quality components, we hurt other teams.

This is a variation on the legacy technical debt example.

The common theme is “other people have a (potentially big) impact on what we as a team can deliver.”

And of course the converse is true.

An example from a user story…

- Estimated c. 18 hours task effort.
- Simple process model:
  - Iterative construction & test
  - Personal review
  - Submit
- Tracked costs due to defects in third-party components.

Plan vs actuals?

Task	Plan /hrs	Actual /hrs
Construction	16	23
Third party defects	0	7
Review	2	2
Total effort	18	32

In this case, third-party defect costs accounted for about half of the estimation error, and 22 % of the total effort.

Caution: you need to track this over multiple user stories to see the trends.

Summary – effect of upstream defects

- Upstream defects can be expensive
- It isn't difficult to track these costs
- You can use this data to:
  - understand the bigger system
  - contribute towards improvement
- Don't use it to beat people up.

Putting these examples together…

- The technique is very flexible
- You don't need lots of data
- It isn't difficult to do
- Use the analysis to gain insight
- Then you can figure out what to change

– and can be applied to a wide variety of problem
- or complex systems to capture what is going on.s.

Some suggestions if you want to try this for yourself

- Start small and simple
- Minimize tooling:
  - Stopwatch
  - notepad
- Take care with operational definitions:
  - How we measure effort?
  - How we categorise?
- Resist the temptation to interfere
- Practice – lots.

Questions?

Further reading

- Joseph Juran – Quality Control Handbook
- Herb Krasner – Using the Cost of Quality Approach for Software
- Roger E Olson - Cost of Quality as a Driver for Continuous Improvement
- Capers Jones - Software Engineering Best Practices
- Watts Humphreys - PSP – A Self-Improvement Process for Software Engineers