THE DEV:QA RATIO
And its importance in Software Development

Effective software development needs a correct balance — that is, a functioning *ratio* — of development resource (Developers) to quality assurance (Tester) resource.

Without a conducive Dev : QA ratio the organisation is literally paying money for wasted effort. A ratio that's too low means the ROI on testing disappears. A ratio that's too high puts your product at risk by introducing a volume of dodgy code. Interestingly (and, when you start thinking in these terms, obviously) the amount of that risk is in direct proportion to the inaccuracy of the Dev : QA ratio.

So what's the optimal Dev : QA ratio? The sweet spot in any software development organisation is determined by many factors, including the relationship between automated testing vs manual testing (and manual oversight of that automation). Suffice to say that if the work your Testers are getting through matches up closely to the volume of code changes being made by Developers then you're in the optimal zone. If, however, Testers are routinely doing overtime and/or have a sizeable and growing backlog of untested code changes to get through, then your ratio is too low. Likewise, if Testers are getting through everything Development is putting out, without cutting corners, and without significant time to spare, then your ratio is probably too low. A low ration gives you QA Testers who have time to explore and implement test automation and general up-skilling. A high ratio, however, is a lot more insidious.

Rather than covering all factors involved in identifying the perfect ratio (many of which may be particular to individual organisations) this article focuses instead on the importance of thinking in terms of the Dev : QA ratio, and provides some strategies for mitigating the effects of a ratio that's too high.

In my career I have worked with both near-optimal ratios, as well as appallingly skewed ones. The latter, whilst the most frustrating, have offered the most opportunities to learn. Most instructive was the Dev : QA ratio that sat consistently at about 9:1, and that's with a stark minimum of automated testing to bolster test protocols. I'm sure there were genuine business reasons contributing to that situation, naturally, and this article isn't an accusatory finger pointed at any particular organisation. Instead, my aim is to describe the effects of that situation based on my first-hand observations, to explore the difficult decision mandated by that situation, and to introduce some of the strategies I used to address them. Whether your interests centre on business management, software development, quality assurance, service delivery, or some other business function, I believe and hope your organisation can benefit from ratio-centric thinking. Ultimately I'm sure the principles central to the Development : Quality Assurance ratio extend beyond software development to any product or service offering.

By definition, any organisation that has more capacity for making code changes than it has capacity to test those changes has a Dev : QA ratio that is too high. Such organisations, at release planning time, face this question:

"Do we add it to the QA backlog, or release it untested?"

Let's examine the impacts of both these options to understand why that decision is such a doozy.

Option 1 — Add indefinitely to the QA backlog

The QA backlog in this scenario is a black hole. I say this for the simple reason that if QA can't test something now, then they never will. There will always be something more important for them to test, and will therefore never get around to the screeds of old stuff that's piling up in the QA backlog. These are code changes of relatively low importance that have not been labelled "must haves" by the business for the upcoming delivery — and that means they are extremely unlikely to be picked up again by QA, who will always be focussed on high-importance and high-risk current changes. This is a tough decision to make in good conscience, because the deduction to be made by the person making the decision is that *you are throwing away perfectly good development simply because you don't have anyone to test it.*

By making the choice to take code changes out of the release and add them to the black hole that is the QA backlog, essentially you're deciding that the efforts of your developer are going to waste.

Financially speaking, in this situation **your organisation would actually be better off laying off some developers** so that you're not paying people to make code changes that will never see a release. What a dreadful concept – but its a viable solution to the core problem, because in doing so you would establish a Dev : QA ratio that actually *works*. You'd get less output in that situation, but you'd avoid the very significant problem of development effort being wasted.

Option 2 – Release untested code

Releasing those same code changes untested, on the other hand, represents a more immediate risk.

QA hasn't even looked at these items: the release date is upon you, and if the fix or feature doesn't work there's going to be trouble and fires to put out, and negatively impacted customers.

In this event it's key to note here that while you may have sufficient development resource to quickly jump on resulting issues, **it will also take your test focus off present matters to address problems caused by not having enough QA in the first place.** This means that whatever QA tests, if there's more development in the release than can be tested, the problem can't be addressed simply by changing QA focus. This is quite literally a way of planning for disaster.

Obviously some code is simply too high risk to go out untested, and presumably that's what QA is actually testing. But what about all the rest? All code changes contain some risk until they're test passed, so when it's release time and you've got to make the call on what goes in and what goes out, what do you do? How should your process handle this situation when it rears its ugly head at the time of every release?

To answer that, I offer three process-orientated strategies for mitigating (not resolving) this scenario. All of them have some merit, and all of them contain flaws, since they're symptomatic measures only. Some battles can't be won, but with these strategies I offer you a way to contain the damage in a way that hopefully aligns with your organisation's values, strategy, and vision, with a long-term view to win the war by restoring your Dev : QA ratio to one that benefits the business with no wasted effort and no unnecessary risk.

Strategy 1 – The release-branch-centric approach

I have had the most success using this approach when managing a weekly and fortnightly release schedule. This strategy focuses QA efforts on the release branch, rather than those in the code repository trunk. Yes, it makes your QA resource entirely reactive and ad-hoc, but the benefit is that stabilisation releases actually contain a degree of stabilisation, and the changes going out untested are kept to an absolute minimum.

Longer term, because those changes that are tested are only tested in the release branch and not in trunk, this approach does set you up for significant trouble when next you branch for that upcoming major version, at which point your new release branch will pick up such items from trunk in the new branch. There's little you can do about this, except handle the exhaustive emergency-response that becomes necessary to find and fix the innocuous problems that were overlooked during the period all those code changes were committed, and which have now snowballed into bigger issues without anybody knowing it (i.e. because QA has barely touched trunk code for the entire stabilisation cycle of your previous major release – which may be a quarter of a year!).

Using this model (which is obviously far from perfect, however much I happen to favour it) the typical outcome is that stabilisation releases are done in a timely manner, and major versions are not – due to an overabundance of unforeseen issues.

Strategy 2 – The trunk-comes-first approach

With this method, the value emphasis is on keeping the code trunk in the best shape it can be in, by focussing QA effort there (instead of in release branches).

To the long-term thinker this is simply common sense. But to the Release Manager concerned with meeting deliveries in the immediate-term it is practically unworkable, since it means almost everything being released isn't tested in the release branch — a gross set of assumptions is used instead: "if it works in trunk, it'll probably work in the release branch too." That "probably" is a disaster waiting to happen whereby issues are first discovered by **the customer, instead of by QA.**

This method is also arguably the least efficient, since Developers have to commit code to trunk, wait for it to be QA tested (if it ever is), and then merge it to the release branch. This requires incredible diligence and ownership from Developers, QA, and other business functions (which in itself is a good thing), but this drawn out sequence of events results in the most effort occurring immediately before release. When QA has to test code changes right up to the moment of deployment, and then smoke test the release too, your delivery deadline is in danger.

Strategy 3 —The high-risk-items-only approach

With this method, your limited QA resource focuses on testing high-risk items only, but testing them in BOTH trunk and release branches. Sounds great, but the key thing to remember here is that this basically halves the number of code changes that actually go through QA, adding to the volume of code changes being released untested. There are obvious drawbacks to this.

The main advantage of this approach, though, is that releases tend to be of better quality (due as that may be to wasted development effort when commits are rolled back out of release branches and trunk alike, to enter the "QA backlog").

The main downside of this approach is the duplication of effort for QA, and the limitations it imposes: larger volumes of code being released untested, and the questionable approach of basing QA workloads on what only someone's best guess has deemed "high risk".

Although this approach is awkward, limiting, and often frustrating, it is the strategy best positioned to effect cultural change in the business, even if only for the reason that it makes Dev : QA imbalances so much more visible than the other approaches do. From a Release Management viewpoint, it's certainly the most complex and unforgiving of these three strategies.

At the end of the day, the strategies above only address the symptoms of a Dev : QA ratio that is too high. To fix the root problem, the organisation needs to acknowledge that the ratio is at fault, and actually commit to acquiring more QA testing resource! The key here is to find ways to quantify wasted effort and the costs of re-work emerging from code changes that have been released untested in monetary terms, to contribute to a business case. We'll explore methods for doing this in a separate article.

If any of the scenarios described in this article are familiar to you, my advice is to first and foremost identify your Dev : QA ratio in numerical terms. If it's 9:1 you've got a major problem. Once you've identified if your ratio is out of kilter, ask whether additional QA resource will actually help.

Once you have an understanding If you believe it will, even if the numbers don't add up, give it a trial: intangible factors like customer confidence in your product, greater internal pride in each release, and a gradual change from putting out fires to proactively developing a quality product are only the beginning.

July 2014

Michael Pritchard