#06
MAY 2019

SQ_mag

THE VOICE OF SOFTWARE QUALITY

AUTOMATIZATION IN A LABORATORY ENVIRONMENT

Christopher P. Rüger on developments in research

ARTICLE

ABOUT CLOUDS AND SERVERLESS TECHNOLOGIES

Abby Kearns on function as a service

INTERVIEW

THE WORLD OF SOFTWARE QUALITY

THE REGRESSION MANAGEMENT QUADRANTS

Pieter Withaar and Johan van Berkel on code and test quality

ARTICLE

DESIGN THINKING IS A "TEAM SPORT."

Mark Tannian on what it means to be a Design Thinker

INTERVIEW

TESTING APPLICATIONS FOR CHILDREN

Nadia Soledad on the difference of testing apps for adulds or children

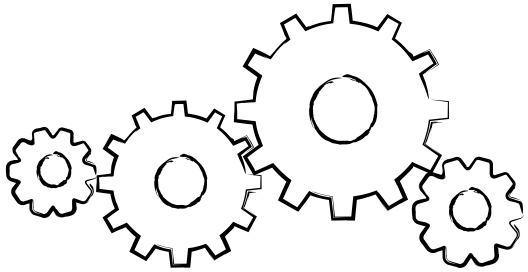
ARTICLE

THE REGRESSION MANAGEMENT QUADRANTS

C

Capable of detecting regression at low execution costs as well as saving engineers from death by boredom, automated checks are often considered to be the silver bullet to regression. But when applied in the real world, the promise seems to be more of a fairy tale: after all the investments are done to automate the regression tests, the amount of regression doesn't seem to decrease. And even though the execution is cheap, maintenance sure isn't.

> ANY CHANGES MADE TO THE CODEBASE FROM THE VANTAGE POINT OF ONE USER NEED CAN YIELD UNEXPECTED CHANGES IN ANY NUMBER OF OTHER USER NEEDS.



> KEEPING THE IMPACT OF REGRESSION IN CHECK IS A NEVER-ENDING STORY THAT REQUIRES CONSTANT EFFORT TO MAINTAIN.

The burden of regression

When a product is changed for whatever purpose, developers will primarily focus on changing the codebase to support the new user need. Assuming that the user need is implemented in an existing product, this means building new code as well as refactoring existing code to integrate the user need in the product. The resulting cycle of development, test, and fix will ensure that –eventually – the new user need is met.

Unfortunately, user needs are never implemented in perfect isolation. Instead, the code is often reused to fulfill multiple user needs. As a result, any changes made to the codebase from the vantage point of one user need can yield unexpected changes in any number of other user needs. Whenever such an unexpected change occurs, we speak of regression. Although regression, technically, doesn't include a specific impact on the business value within its definition, positive regression rarely ever occurs. More likely, the impact is either negligible or detrimental (in the form of increased maintenance costs and/or decreased product performance).

With the risk that regression poses towards the business value of the product, the need to prevent regression from reaching the customer is obvious. Roughly speaking, the level of detrimental regression that reaches the customer can be lowered by:

- Preventing the occurrence of regression during development
- Detecting any (residual) detrimental regression before reaching the customer

Keeping the impact of regression in check is a never-ending story that requires constant effort to maintain. But since it prevents potential issues from occurring, it is virtually impossible to quantify the economic benefits that come from the effort spent on managing regression. As a result, organizations are inclined to spend as little effort on the topic as possible.

Following this line of reasoning, extensive automated regression testing is often seen as the silver bullet to regression. By writing tests that focus on verifying the business value of all user needs, only detrimental regression is caught and fixed. Only code that causes detrimental regression is reworked, and since automated tests are cheap and fast to execute, the cost-benefit ratio of this approach is supreme. Or so it seems… Because take a step back and you will see that this approach only results in endless drudging through the swamps of maintenance hell while chasing a unicorn that doesn't exist.

If that last bit doesn't make a lick of sense at the moment, then you probably haven't been introduced to the Regression Management Quadrants. Luckily, that's exactly what this article is about.

Introducing: The Regression Management Quadrants

The Regression Management Quadrants take the two methods of managing (detrimental) regression (i.e. prevention and detection) and plot the relationship between them into four quadrants that can help organizations determine on how they should (and shouldn't) manage regression. But before we start talking about the quadrants, let's start by explaining what we feel are the most impactful factors when it comes to the prevention and detection of regression:

Preventing regression by increasing code quality

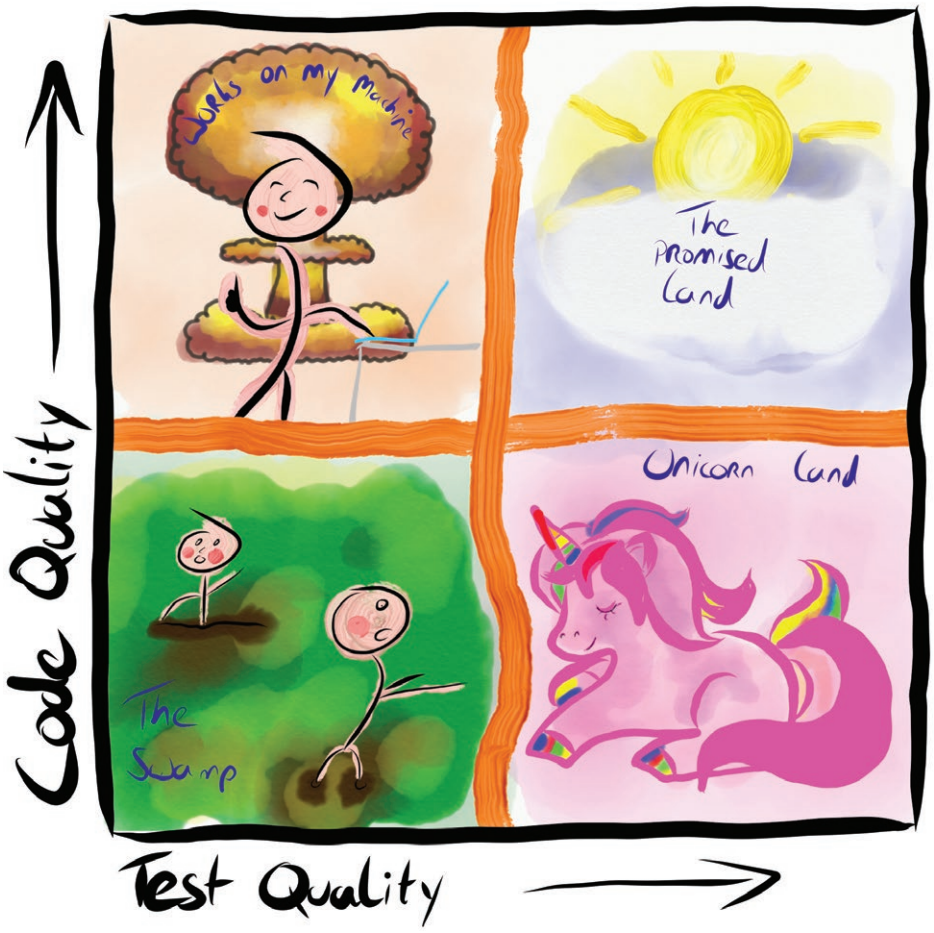
The first method to limit detrimental regression is to prevent regression from occurring in the first place. Its success rate is mostly determined by the cognitive effort it costs to correctly assess the impact of any change in code on the whole codebase. Although experience plays a role, code quality is vastly more impactful. When talking about code quality, we mean any characteristic that impacts the effort it takes to translate the code into a relevant and correct mental model. This can be anything ranging from readable naming conventions to cyclomatic complexity, code cohesion, and code coupling.

Detecting regression by high-quality tests

The second method to limit detrimental regression is to detect regression before it reaches the customer. Detection is done by testing; therefore, the most impactful factor is the quality of testing. When talking about test quality, we mean any characteristic that impacts either the effectiveness or efficiency at which tests can detect detrimental regression. Effectiveness is determined by the amount of detrimental regression still reaching the customer (the less regression goes through, the more effective the tests are). Efficiency is determined by the time and effort it takes to sustain the regression tests, which can be characterized by, for example, test redundancy and test maintainability.

Characterizations as result

Now that we have defined the most impactful factors for successfully preventing or detecting regression, we can plot both of them, resulting in the four quadrants mentioned earlier:



Each quadrant in the RMQ represents a potential strategy for managing regression and comes with its own reasons for why organizations end up using the strategy, and why – as is the case with any self-respecting Four Quadrant Matrix – the top right quadrant is the only correct answer to managing regression. Let's dive into the quadrants, shall we?

↗
Pieter Withaar
Test automation enthusiast, consultant and trainer at Improve Quality Services.

↗
Johan van Berkel
Test consultant and technical writer at Improve Quality Services.

Low code quality, low test quality: Welcome to the swamps of maintenance hell

Most product developments are extensions or improvements to an existing product. Whether it is the result of prioritizing new features over maintenance activities in the earlier days of the product life cycle or legacy code inherited from days long gone that no one dares touch, most organizations are stuck with a product that contains (a lot of) technical debt. Since assessing the impact of a change in a product with low code quality is nearly impossible, the product will yield high levels of regression. Initially, this might result in a storm of complaints from customers. This quickly backfires into a reflex response by the organization: development needs to stop regression bleeding through yesterday. Since the fastest way to stop bleeding is to apply a Band-Aid, the effort to detect regression is intensifying, but with low code quality, regression doesn't decrease; it merely shifts to other areas of the product. With each failure found, the automated checks are expanded, gradually growing to unmanageable proportions. Welcome to the swamp of maintenance hell.

Low code quality, high test quality: unicorns are still mythical

The most common approach to dealing with maintenance is to prioritize testing, which is the reflex response to issues in the field for any organization. However, sustaining regression testing results in high costs, and eventually, people will repeat the logic of this article's introduction: we need good tests, and since we can't predict where regression occurs, we have to test everything all the time. From there, test automation is only a step away: "If all regression testing is done extensively and automated", they reason, "we can limit the code intervention to fixing parts that cause actual issues, and automated tests are cheap top execute, so it's a no-brainer!".

In practice, this reasoning neglects a critical component of the cost of testing, in general, and test automation, specifically: maintenance. Even when the test suite is effective at detecting regression, its application is inherently inefficient. Remember: regression is not always detrimental. In some cases, its impact is trivial. But a trivial change in behavior is a change nonetheless, and automated tests are binary in their result. This means that with any trivial regression in the product, the effectiveness of the tests will decrease as some tests will fail while not detecting detrimental regression.

> REGRESSION IS NOT ALWAYS DETRIMENTAL. IN SOME CASES, ITS IMPACT IS TRIVIAL. BUT A TRIVIAL CHANGE IN BEHAVIOR IS A CHANGE NONETHELESS...

THE REGRESSION MANAGEMENT QUADRANTS

> ADDITIONALLY, THE QUADRANTS CAN HELP TO DETERMINE WHICH QUADRANT YOUR ORGANIZATION IS AT...

To make the regression tests effective again, all regression tests failing due to trivial changes need to be adjusted to account for the change. This, basically, moves the maintenance burden from code to test, and since test only detects regression and does not prevent it, this maintenance effort is endless and ever-changing. Where the organization thought it found the magical unicorn, they actually end up running circles in the swamps of maintenance hell.

High code quality, low test quality: well, that escalated quickly…

It may be that the organization has spent considerable effort on code quality in the initial version of the product, understanding that doing so would lower the burden of maintenance. Or it could be that the organization stuck in the swamps of maintenance hell didn't fall for the unicorn and understood the value of preventing regression over detecting it. Regardless of their reason, the organization could explicitly strategize to focus their attention on code quality. Although this is, arguably, the "lesser evil", it still poses some challenges on its own.

The problem with good code quality is that – on the surface – it devaluates regression testing. If regression rarely ever occurs, then why spend all this effort on building extensive automated regression tests that rarely ever detect detrimental regression? Alternatively, there might be a lot of effort spent on test automation, but the resulting tests are actually ineffective in detecting detrimental regression. How can you know whether they are effective or not if nothing ever fails? Regardless of whether the automated tests are deprioritized or ineffective, the result is the same: all tests are "green" and no complaints from customers, so all is well, right?

The answer is "yes, for now". The primary purpose of regression tests should not be to find practical failures but to help identify patterns that emerge from finding those failures. Consider this: degradation in code quality should cause regressions tests to fail regularly and erratically. But with automated tests being ineffective at detecting regression, this pattern doesn't occur. Instead, the regression caused by the gradual decline in code quality builds until it reaches critical mass and blows up in their faces. Customer complaints start pouring in on a regular basis until the damage is too much to ignore and the strategy is revisited. Since the issues were not detected, the first inclination is to analyze the automated tests, which will uncover that the quality of the automated tests is, in fact, insufficient. Suddenly, the unicorn in the bottom-right starts to look very real as it winks seductively. And before we know it, we're back in the swamps of maintenance hell.

High code quality, high test quality: at long last, we reach the Promised Land

With all the wrong ways of going about managing regression explained, we end our story in the quarter where everyone wants to be: the Promised Land. It should no longer come as a surprise that proper regression management requires investing in both high-quality code and regression tests. Hopefully, we helped you realize that the true purpose of regression testing shouldn't be to detect failures but to prevent them from occurring in the first place. So even though the initial costs of setting up your product and tests for proper regression management can be costly, the upkeep of such a strategy is significantly lower than the costs your organization would incur from buying all those silver bullets.

Three of the four quadrants turned out to be dead ends, but they still serve a purpose: providing arguments that prevention and detection both have their own merit. High code quality is needed to achieve low regression rates, whereas high-quality regression tests are required to retain it. Additionally, the quadrants can help to determine which quadrant your organization is at, which – as it turns out – requires a broader perspective than just analyzing the current list of incidents. Instead, focus on patterns that occur over time: do the automated tests fail frequently? Better double-check your code quality. Structurally reworking failed test cases that shouldn't have failed in the first place? You might be chasing that illusive unicorn. Perfect scores on static code analysis with all builds green? No time for complacency, but remain critical about your test coverage to prevent the need to "duck and cover" from the sudden influx of customer complaints. And regardless of the characterization applicable to your organization, we hope we helped in the process of turning the maintenance hell, into a maintenance "swell"…

Ok, we'll show ourselves out now.