SolutionsIQ

# Building Quality In

## And How to Deal With the Mess that Scrum Exposes

By Monica Yap

## Abstract

In traditional development, the tendency is toward longer development life cycles. Agile software development surfaces the many hidden problems inherent in long development cycles executed without end user feedback. In particular, Scrum and Extreme Programming make impediments to delivering high-quality software visible and also equip teams and organizations with tools and practices for removing these impediments. This white paper covers the four “messes” that Scrum exposes and how organizations can use XP practices like automated unit testing and test-driven development (TDD) to address them.

# Introduction

Scrum is great: it limits team size to a reasonable number of members, and it has specialized roles that focus on team health (ScrumMaster) and end user satisfaction (Product Owner). But what Scrum does best, whether for good or for bad, is it exposes a lot of the mess that normally hides in the cracks of long development cycles. In traditional development, the tendency is toward longer development life cycles: long design and planning periods yield huge requirements docs to be executed against until release, which is far off with wildly optimistic milestones along the way. And, for a good chunk of time, this approach seemed to work. But today the business and development terrain is just too uncertain for this to be sustainable.

Agile is much better suited to development in today's volatile, uncertain, complex and ambiguous (VUCA) landscape. Agile software development surfaces the many hidden problems inherent in long development cycles executed without end user feedback. In particular, Scrum and Extreme Programming make visible the impediments to delivering high-quality software and also equip teams and organizations with tools and practices for removing these impediments. This white paper covers the four "messes" that Scrum exposes and how organizations can use XP practices like automated unit testing and test-driven development (TDD) to address them. The four messes are:

1. Integration is hell.
2. Quality is not built in from the beginning.
3. Code bases are fragile and unstable.
4. There are too many specialized roles.

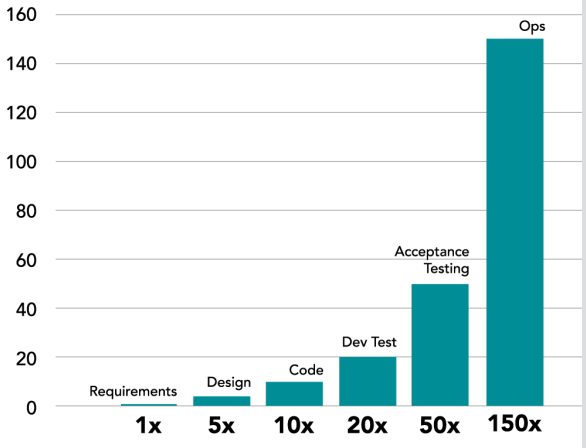
"Scrum and Extreme Programming make visible the impediments to delivering high-quality software and also equip teams and organizations with tools and practices for removing these impediments."

## Mess #1: integration is Hell

In a much longer development lifecycle of months or even years, we wait to integrate until the end right before release. This creates a huge integration risk, and it usually comes with big surprises that ultimately lead to a lot of rework that inevitably delays the release or causes certain already "built" features to be cut. Also, delaying the integration inflates the cost of fixing any defect. Scrum exposes this issue by delivering end-to-end, potentially shippable features by the end of every Sprint. This requires us to integrate the changes into the overall product every Sprint, which in turn lowers the cost to fix any defect found early in the development cycle.

The question is – how do we do this? Manual integration is too painful to do every Sprint. Some teams implement an additional "hardening" Sprint before each release to tie up loose ends and carry the product over the finish line, and this is where the integration happens. This antipattern presents much risk and surprises definitely arise, so now we are back to the original question: if integrating more frequently lowers the cost to fix defects, how do we integrate every Sprint in a sustainable way?

The answer is called Continuous Integration, a key XP practice to help us get to a sustainable integration pace. As small changes are checked into the common source code




True cost of a bug fix

### What is Continuous Integration?

Continuous Integration (CI) is a Extreme Programming practice where members of a delivery team frequently integrate their work (e.g., hourly, daily). Each integration is verified by an automated build, which also performs testing, to detect any integration errors quickly and automatically.

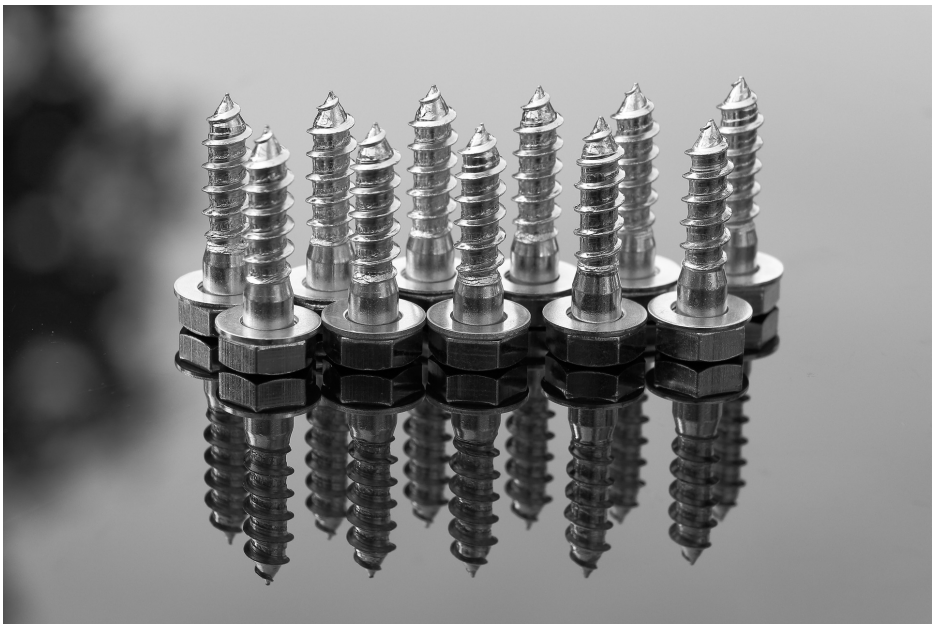
The principle benefits of CI are significantly reduced integration problems and better team coordination leading to higher-quality software being released more rapidly than in traditional software development environments.

repository along with tests, we have an automated engine to perform a build, run all the tests, and spit out the errors when it fails. Then whoever checks in the code has to fix the errors and rebuild it, and no one else is allowed to check in until the build is fixed. Note that this is not just a technology or magical server; it's a practice carried out rigorously by all team members and all teams. In time, using this approach, you have a code base that is always integrated and tested.

## Mess #2: Quality is not built in from the beginning

Screw manufacturers today have a screw-building machine that automatically validates that the screw size and length is up to specification as each screw is cut. By building the specs validation into the individual manufacture of each screw, the manufacturer is building quality into each unit. This is what we call "building quality in from the beginning."

How does this compare to the software world? When developers execute within a traditional development cycle, we get used to the design-code-test phases. Because the test phase is at the end, it's most likely performed by a different group of people (e.g., testers). Although we spot-check our code from time to time, true testing is usually not top of mind until the end when we have to fix bugs. Furthermore, testing is not in our job spec. So we don't really know if the code satisfies the functional, security, and performance requirements until the very end. This creates a problem if the true quality is not baked into



Screw manufacturers now have machines that automatically validates the screw size and length. That is an example of building quality in.

## What is TDD?

Test-Driven Development (TDD) is a software development process invented by Ken Beck in the 12 original Extreme Programming practices. The process is as follows:

- The developer writes a failing automated test case that defines a desired improvement or new function.
- They then produce code to pass that test.
- They refactor the new code to acceptable standards.

the code from the beginning.

Again, Scrum exposes this by requiring shippable code from each Sprint (which means it has to work functionally), and it makes the whole team accountable for the results, not just the testers. This means the quality has to be built in every Sprint. The most effective way to do this is by ensuring each unit in the code works as it was designed to.

So how do we create a process where code is tested as it's produced to ensure quality is built in? Or, going back to our opening analogy: how do we design a screw machine that will measure each screw as it is produced?

The first thing we need to do is learn how to do automated unit testing, where we test each method or function. This process is called test-driven development (TDD). TDD is not just unit testing; it's a practice that shifts our way of designing code to yield more loosely coupled and easily testable units.

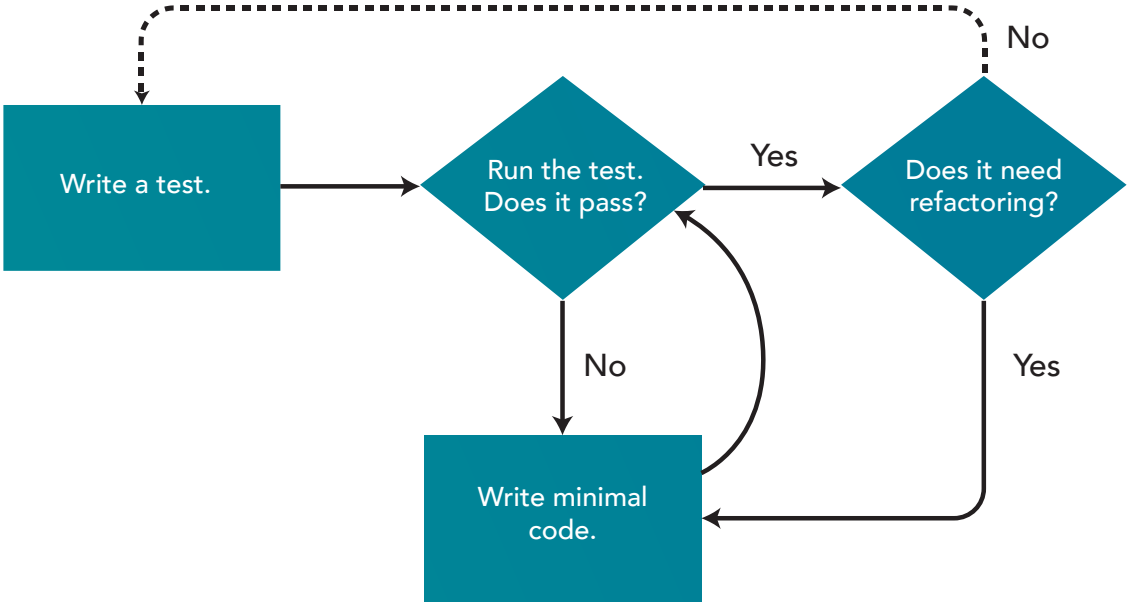
Now we have each unit and foundational block in our code, which has been thoroughly tested. Next we need to start building automated story tests each Sprint, so the code for the new features is tested as it is developed, and all existing features are regression tested during each Sprint. From this scenario, it should be clear that testers cannot be separate from the team; testers are part of the Scrum team. More importantly, they can help write story tests from the beginning.

Next would be automated load/performance testing, so that the unfortunate discovery that a database cannot

perform happens at the beginning, rather than at the end right before the release.

Now fast-forward to post-launch, when the next version needs to be developed. New teams are formed or new team members are added to develop the new version, as is usually the case. The code is handed over to them and all the unit, functional, and performance tests serve as "live" documentation rather than a static Word doc, which is generally instantly out of date. The team can change or add new features safely, and perform more tests to enhance the code.

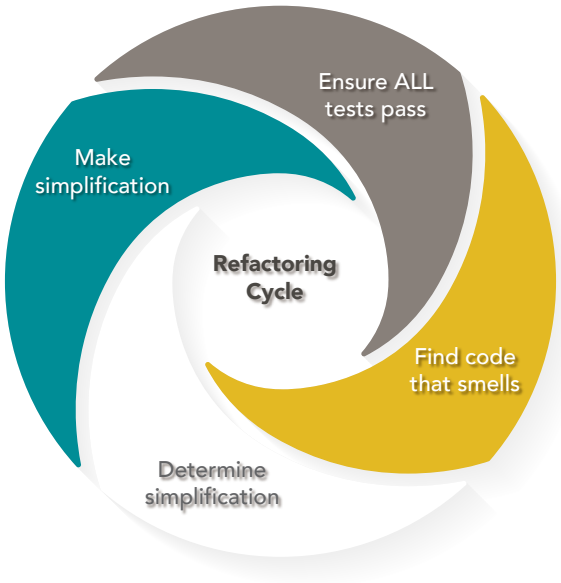
TDD and automated testing are not easy to start or maintain – not to mention the learning curves for all roles in the team. "Who's paying for this?" is another question to ask. In the end, everyone ultimately pays the cost of not doing these practices in a number of ways: long bug fix cycle, rework, delay of launch, missed early release window opportunities, etc. All of these affect the bottom line. Scrum paired with XP offer the business and development operations an option for addressing these problems early on when the cost is undoubtedly lower.




TDD Cycle (O'Reilly Media)

## Mess #3: Code bases are fragile and unstable

When we add new features, we are smart about not reinventing the wheel each time: we search for existing code that we can reuse, copy it, and then make changes to fit the new functionality. The next time, we do the same thing again. Eventually we have similar code in multiple places, because this seemed to be the faster way to get things done at the time. In reality, this makes adding or changing code more difficult in the future. We need a way to add code quickly while maintaining the ability to rapidly make changes in the future. It's much easier to clean up and simplify after small changes rather than after a whole system has been written. We need to refactor our code after we have successfully implemented each change, for example, cleaning up any mess we make before starting on the next change. In this case, we would extract the common pieces into a centralized unit for repeated reuse. We do this by adding the steps of identifying smells and simplifying into our standard daily development cycle.




### What is Refactoring?

Refactoring is the Extreme Programming practice of changing existing software code in order to improve the overall design. Refactoring normally doesn't change the observable behavior of the software; it improves its internal structure.

## Mess #4: There are too many specialized roles

In the traditional development process, work is divided up and assigned to each individual, usually based on each person's familiarity with the business domain and/or technical skill. Joe may be more familiar with the ordering part of the system, so we assign all work related to ordering to Joe. Meanwhile, Mary is more skilled at user interface development, so we assign all UI work to Mary. With such an arrangement, Joe and Mary get used to working by themselves and in their own areas. By optimizing based on our expertise and domain knowledge, gradually no one else can work beyond their own focus area because everyone is siloed. We are afraid of even letting others touch our area and vice versa. This is called local optimization.

When there is too much local optimization, the throughput suffers globally because the work will always bottleneck somewhere. The Scrum framework exposes this problem because teams need to deliver end-to-end features every few weeks. What if, for example, someone is sick during the Sprint and no one else is able to fill in for them? This is an all too common problem that affects the team's ability to deliver. How do we fix this?

Too much local optimization ensures there will always be a bottleneck somewhere. Pair programming can help.

We start by helping Scrum teams become cross-functional by adopting the XP practice of pair programming. In pair programming, two individuals with different expertise and experience can both contribute to the code base. And

## What is Pair Programming?

Pair programming is an Extreme Programming practice used in Agile software development. In pair programming, two programmers work together on the same code at one workstation. One types in code ("the driver") while the other reviews each line of code as it is typed in ("the navigator"). The two programmers switch roles frequently. Pair programming is one of the 12 original XP practices invented by Ken Beck. Counter-intuitively, pair programming is often more productive than two individuals working independently on separate tasks.

it isn't only developers who pair up with other developers to write code/unit tests; developers can be paired up with testers to write functional/performance tests. SolutionsIQ's own development shop has brand new recruits pairing with seasoned veterans within moments of formally joining a team! This kind of intellectual and technical cross-pollination ensures that no one on the team is a silo.

Traditional development teams operate with many individual siloes, or I-shaped developers: architect, developer, tester, business analyst, etc. I-shaped developers have very deep knowledge in one general area and may even be an expert that all work has to pass through. This problem is already a mess when the expert is available but even worse when he or she is not. And often no one wants to touch a spoken-for focus area for fear of sullying it. In contrast, cross-functional teams work best with what's called T-shaped developers or skill sets. Teams with T-shaped developers more evenly distribute knowledge across several individuals, even though each individual has his or her own subject matter domain(s). In cross-functional teams with T-shaped developers, productivity isn't affected whenever someone gets sick, or goes on vacation, or wins the lottery.

As with the previous mess, cost is often an obstacle implementing pair programming. Some ask, "Wouldn't it be too costly for two people to perform one person's work?" The flaw in this question is the assumption that the pair will somehow provide only the same amount of value as one individual. Further, the questioning party doesn't say, "I guess two people working on the same problem would

resolve it twice as fast." This is because the naysayers are convinced that development is an activity best done by one person in the isolation of their silo.

However, according to multiple studies done on pairing, while there is increased effort in pairing, there are also reductions in time to deliver and increases in product quality. What's interesting is that the more important benefits are rarely considered: over time, everyone builds a common knowledge of all areas, code ownership is shared by the collective, and the team has more experts to tap into even if the main expert is not available. Note that we are not advocating that everyone be an expert in every area. Instead, we're encouraging team members to gain a working level of understanding of the areas that they are not familiar with, so that when you have to fill in the gap, the work may take longer but it doesn't screech to a halt.

With pair programming, over time, everyone builds a common knowledge of all areas, code ownership is shared, and the team has more experts to tap into. That adds up to real business value.

# Putting it All Together

Some people vilify Scrum because of all of the messes it surfaces. People argue that, "Things were just fine before we started doing Agile and Scrum!" But the fact of the matter is that things were not "just fine": the dysfunction in the system was hidden by dysfunctions in the working patterns. The four messes covered here are antipatterns of traditional software development, and they have very real business implications. While it may be a hard truth, it is a truth nonetheless: in today's day and age, we can't afford to build shoddy code. You may get away with it for a while, but eventually users and workers will recognize the suboptimal quality of the product and look for a better alternative.

Taken as a whole, Scrum and XP work together to bring legacy code bases and programming practices into this millennium. Scrum teams are small and nimble, collaborating together to deliver end-user value quickly and responsively. The quality of the product is ensured by XP practices like Continuous Integration, TDD, refactoring, and pair programming. These practices complement each other: for CI to be effective, you need validation tests running at the many different levels of your code, which is simply too gargantuan a task for humans. TDD and Story Testing make these sustainable and manageable. Even so, delivering at high speed makes refactoring an absolute necessity. Pair programming makes all aspects of coding but especially refactoring more effective and, ultimately, more efficient. All of these practices put together mean that the enterprise spends less time, money and resources later down the line when the cost of even a small defect can balloon to astonishing proportions.

In today's day and age, we can't afford to build shoddy code. You may get away with it for a while, but eventually users will look for a better alternative.

SolutionsIQ

# Building Quality In

**And How to Deal With the Mess that Scrum Exposes**

## Want to Learn More?


**Visit** SolutionsIQ.com

**Email:** info@solutionsiq.com

**Toll-Free:** 1-800-235-4091

**Direct:** 1-425-451-2727


SOLUTIONS IQ