Best Practices for Expanding Quality into the Build Cycle

What's Inside

01 | Introduction

03 | Improving Quality Through Functional Testing in Continuous Integration

05 | Shifting Quality Left Is Hard Work

06 | Continuous Testing Costs vs. Delayed Feedback Costs

08 | Improving Quality One Step at a Time

12 | Balancing WIP: Stop Starting, Start Finishing

13 | Taking Advantage of Bottlenecks

14 | Beyond the Build and into a Culture of Quality

15 | Using Test First Approach to Build Automation into Development

17 | A New Kind of Testing Professional

18 | Conclusion

Introduction:

Agile and Fast Feedback

Building quality into development is now an imperative for all software-intensive companies. The Agile approach for software delivery is finding its way into more of these organizations as the pace of mobile demands faster and more efficient releases. Yet "Agile" is not a free lunch.

It's fairly common for organizations to focus on the project management aspect of Agile and implement frameworks such as Scrum, Scaled Agile, or Kanban. Some teams just switch their Application Lifecycle Management (ALM) tool to an "Agile" tool or add an Agile template to their current tool. But these same organizations end up frustrated with the mediocre results they get from their Agile investment.

Other companies go beyond just the process and implement some key technical principles. One of the most useful principles in Agile is "Working Software over Comprehensive Documentation."

The intent behind this principle is that as uncertainty grows we need to tighten our feedback loops to make sure our assumptions are correct and we're headed in the right direction. Whether those are business assumptions or technical ones, the principal of "working software" is a much better way to close the feedback loop to see if your software is achieving the right outcomes, rather than just presenting what you plan to do. What's missing is the importance of testing "working TESTED software," which goes beyond cursory demonstration to real test coverage of the new functionality and the system as a whole.

To go faster, we must make sure that software is indeed "working TESTED software." By not testing often and as early as possible, you risk failing to deliver products at the frequency and quality your business demands.

How early should you be testing? The short answer is you should shift testing left into the build phase of the SDLC. But such a transition requires changes in culture, tools and strategy. This report will explore these challenges and offer advice on what it takes to expand quality testing into the build cycle.

By not testing often and as early as possible, you risk failing to deliver products at the frequency and quality your business demands.

Shifting Quality Left Through Functional Testing in Continuous Integration

You can't discuss Agile development without mentioning CI (continuous integration), a well-established practice that calls for building, integrating, and testing the system you're developing upon delivery of even the smallest change.

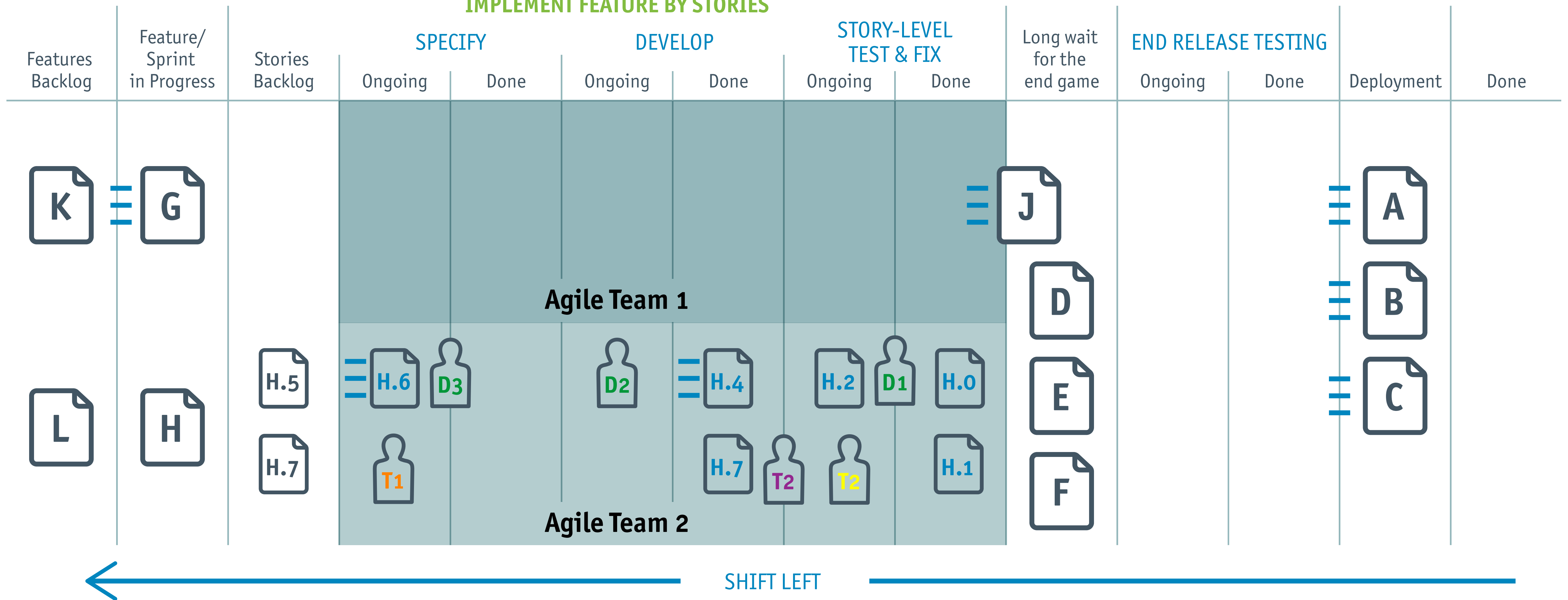
Many organizations invested in CI before even thinking about adopting Agile development approaches. This helps explain why CI is one of the most popular technical practices associated with Agile. Fifty percent of all Agile teams use CI, according to VersionOne's 2015 State of Agile Report.

Another practice coming from the Agile world is "Definition of Done" combined with "Whole Team." Effective Agile teams implementing a framework like Scrum share these common traits:

1. They consider it the responsibility of the "Whole Team" to get an item to "Done."
2. They don't consider a work item "Done" until it has been tested and cleaned up.
3. When testers are struggling to finish testing, developers give them a hand by taking on some test automation and test preparation responsibilities, running regression tests, and even testing new functionality that others on the team delivered.

Teams are discovering that these practices help them become more efficient. Defects are found earlier among a much smaller set of changes and therefore are easier to fix. Convergence on a working and reliable system is dramatically accelerated. If you look at the development flow as going from left to right, these practices will shift testing to the left. The increasingly popular term "Shift Left" was invented to describe this process of building in quality earlier.

50% of all Agile teams use CI, according to VersionOne's 2015 State of Agile Report.



Key takeaways:

- Continuous integration is a key enabler of Agile; without CI, productivity improvements will be limited.
- A "Definition of Done" that doesn't include testing and cleanup delays both developers and testers in the end.
- The more integrated testing is into the development phase, the fewer escaped defects and delays.

Faster Feedback: The Human Perspective

In addition to the cost savings that come with fixing defects earlier in the lifecycle, there's a human nature reason for shifting left.

Agile practitioners notice that quality goes up as time to feedback goes down. As described in Sterling-Mortensen's HP Laserjet Firmware Development Case Study: *"People are self motivated to improve quality if they can see quickly that what they're doing has problems. But if there is a long delay their motivation drops significantly. Going to small batch iterative development reduces defects. Every time the process went faster, the quality got even better."*

Shifting Quality Left Is Hard Work

With all that said, one would expect widespread adoption of practices that instill quality into the build cycle. But while unicorns like Google, Facebook, Amazon and Netflix may be shifting left with relative ease, IT organizations in "work horse" industries such as banking, insurance, and media are struggling to shift left both technically and organizationally.

On the technical side, while continuously building a system is now a mainstream practice, integrating significant test coverage as part of the continuous build is much more challenging. As crucial as test coverage is to quality, it's surprisingly difficult to integrate into CI without help. Statistics from the State of Agile report show that while CI adoption is at 50% among the survey respondents, automated acceptance testing is only at 28%.

For many IT organizations, this isn't a surprise. Implementing a comprehensive test automation suite that covers acceptance tests and integration tests is hard. This is especially true when there are legacy test cases and old systems to deal with, or when a mobile application with a lot of UI elements is involved.

In many cases, continuous integration is just "continuous build" managed by developers and quality is handled by testers later. Sometimes we see a hybrid structure where some quality is built into the build cycle through the testing of new functionality by a tester on the Agile team and regression testing happens toward the end of the release by a dedicated testing team.

Test automation in and of itself is another challenge for these organizations. We still hear statements like, "There's no ROI for test automation." In other cases, it's still handled by a special automation team removed from the build cycle. As the time between coding and test automation grows, it becomes harder to drive an automation-friendly design because it's not on developers' minds when they're building the software. It then becomes too late to change course when automation finally comes into the picture.

Key takeaways:

- Even as "continuous build" grows in popularity, achieving real continuous integration that includes robust test automation eludes many organizations.
- The "separate silo" approach to test automation is mediocre at best because it's too far removed from the development cycle.

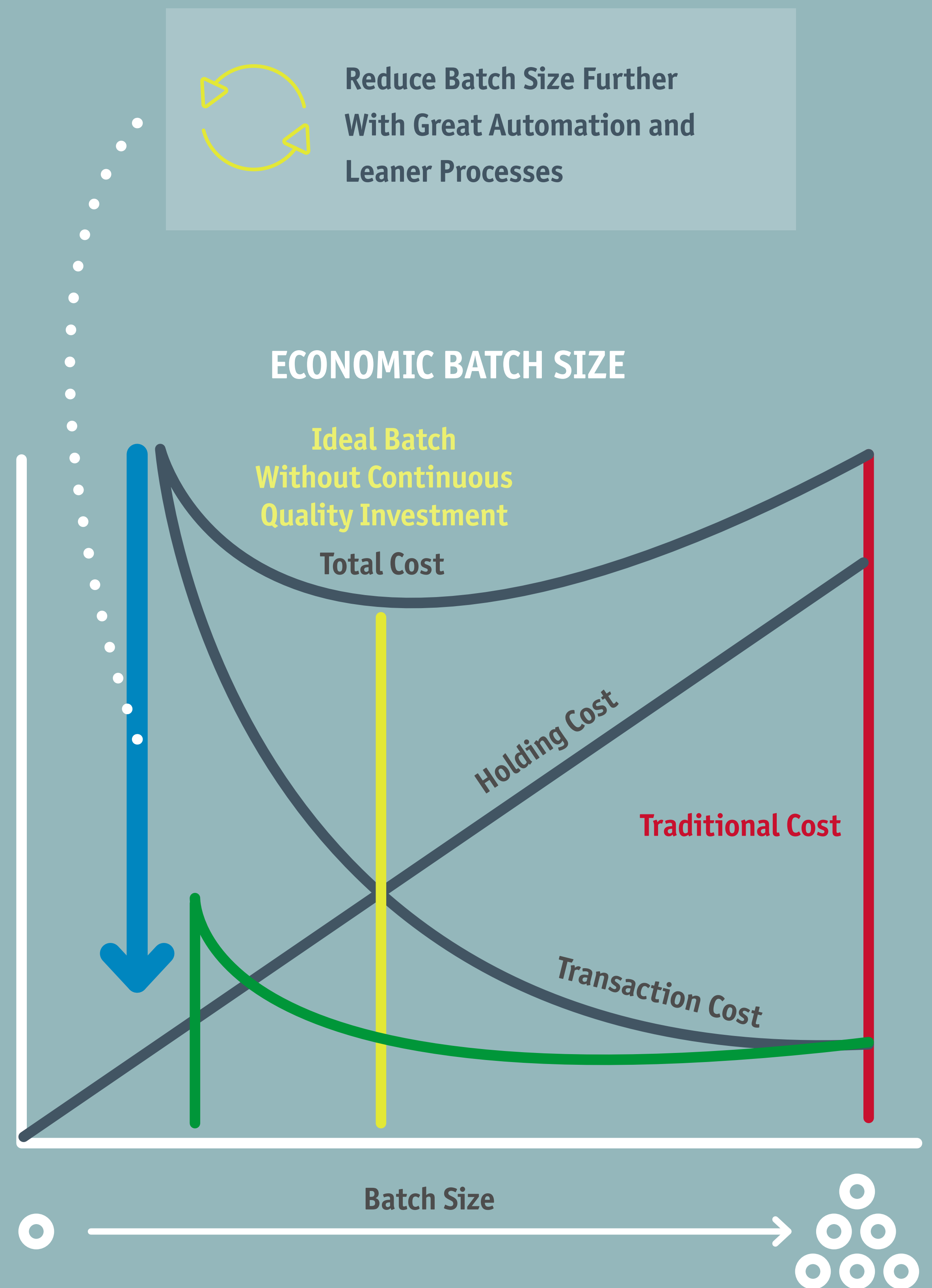
In the State of Agile report, CI adoption is at 50% among survey respondents, but automated acceptance testing is only at 28%.

Continuous Testing Costs vs. Delayed Feedback Costs

The main economic benefit of expanding quality into the build cycle is dramatic reduction in release costs. If we can tighten the feedback loop, then fixing what we find earlier will be easier and cheaper. The ROI of automated testing is the fast feedback about defects that ultimately saves developers hours of time. So why do so many of us still lag behind with a longer feedback loop that depends on testing for quality late in the cycle?

The main culprit is transaction costs. What are transaction costs? Whenever we go through a build/integrate/test cycle, there's certain overhead (backlog prioritization, story lock, retrospectives) regardless of how many new features have entered the build. In many cases, we look at these fixed costs and find it hard to justify wasting time and effort when we could just wait a bit and run the process for a bigger batch. Organizations end up asking, "Why run a test cycle every two weeks if we could run it every four weeks and save some testing costs?"

The transaction cost curve diagram from Donald G. Reinertsen's book "Principles of Product Development Flow" (on the right) visualizes this desire to save costs by going to bigger batch sizes.



From the book "The Principles of Product Development Flow" by Donald G. Reinertsen

What the diagram also shows is those costs we tend to ignore: "Holding costs" or "costs of delay." These are the costs of finding and fixing quality problems further from the point they occurred.

By looking at the total costs—combining the transaction costs with the holding costs—we get what is called a tradeoff curve. We can use this to find the ideal batch size for a certain process in a certain context. When applying this model in the field, we frequently see that neither the Agilists insisting on continuous quality in the build cycle nor the people who wait for "just one more feature" before running their tests are optimizing their economic result.

With all this data at hand, we could calculate the batch size that achieves the perfect balance between transaction and holding costs, but we don't need to get this perfect. Because this is a U-shaped tradeoff curve, there's a big area in the middle where the economic outcome is similar for a range of batch sizes. Reinertsen advises that if you're currently running an economy of scale, simply reduce your batch size by half and start from there. It's important to note that this is the right approach even without making any process improvements that reduce the transaction costs.

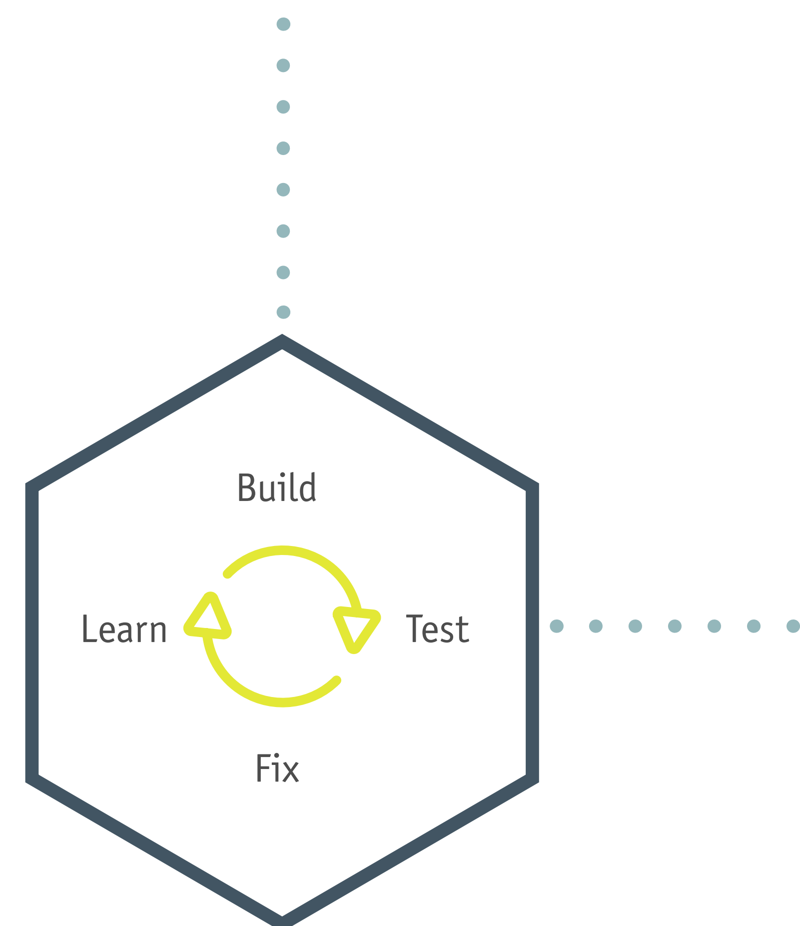
In many cases, each test type (performance tests, regression tests, security testing) bears different costs. This means we need to apply the tradeoff curve model and find the right batch size for each test type. The ideal batch size for regression testing may be daily, while security testing is more effective on a weekly basis.

Once we decide this, we need to reduce batch sizes. How do we do that? We'll look at several steps we need to take in the next sections.

Key takeaways:

- Ideal testing frequency is a function of testing costs as well as fixed costs.
- There isn't one best practice for testing frequency. You need to identify the ideal frequency per testing type.
- Each test type might have a different ideal batch size. Apply the tradeoff curve model for each test type separately.

"The ROI of automated testing is the fast feedback about defects that ultimately saves developers hours of time."



Add Feature/Epic/
Iteration level testing

STEP 1: INTRODUCE AGILE TESTING

IMPLEMENT FEATURE BY STORIES

Features Backlog | Feature/ Sprint in Progress | Stories Backlog | SPECIFY: Ongoing, Done | DEVELOP: Ongoing, Done | STORY-LEVEL TEST & FIX: Ongoing, Done | Ready For Feature Test | TEST FEATURE LEVEL: Ongoing, Done | END RELEASE TESTING: Ongoing, Done | Deployment | Done

K G J A
D B
L H.5 H.6 D3 D2 H.4 H.2 D1 H.0 E C
H
M H.7 T1 H.7 T2 T2 H.1 F

Agile Team 1

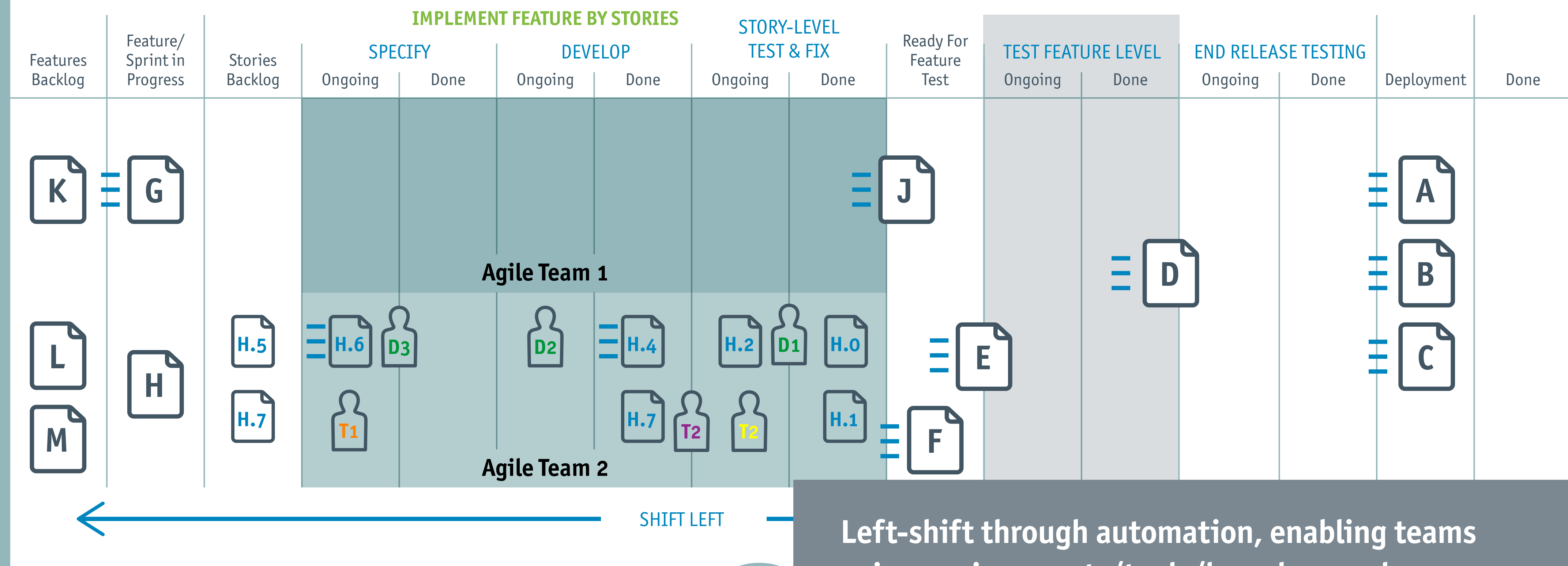
Agile Team 2

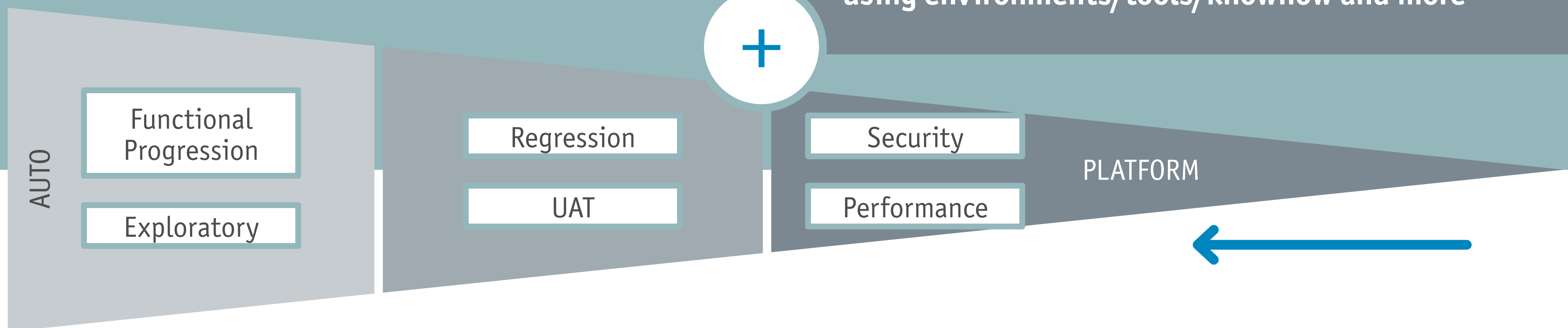
SHIFT LEFT

Improving Quality One Step at a Time

Once we decide to reduce the batch size, the first step is to establish a way to visualize your workflow. You can use a Kanban board (shown above) to help you see and improve the flow. You can use a physical board or an electronic one if you have a distributed team. Some well-known Kanban tools include LeanKit and Trello. Most Agile ALM tools like JIRA, CA Agile Central, and VersionOne now provide Kanban boards as well.

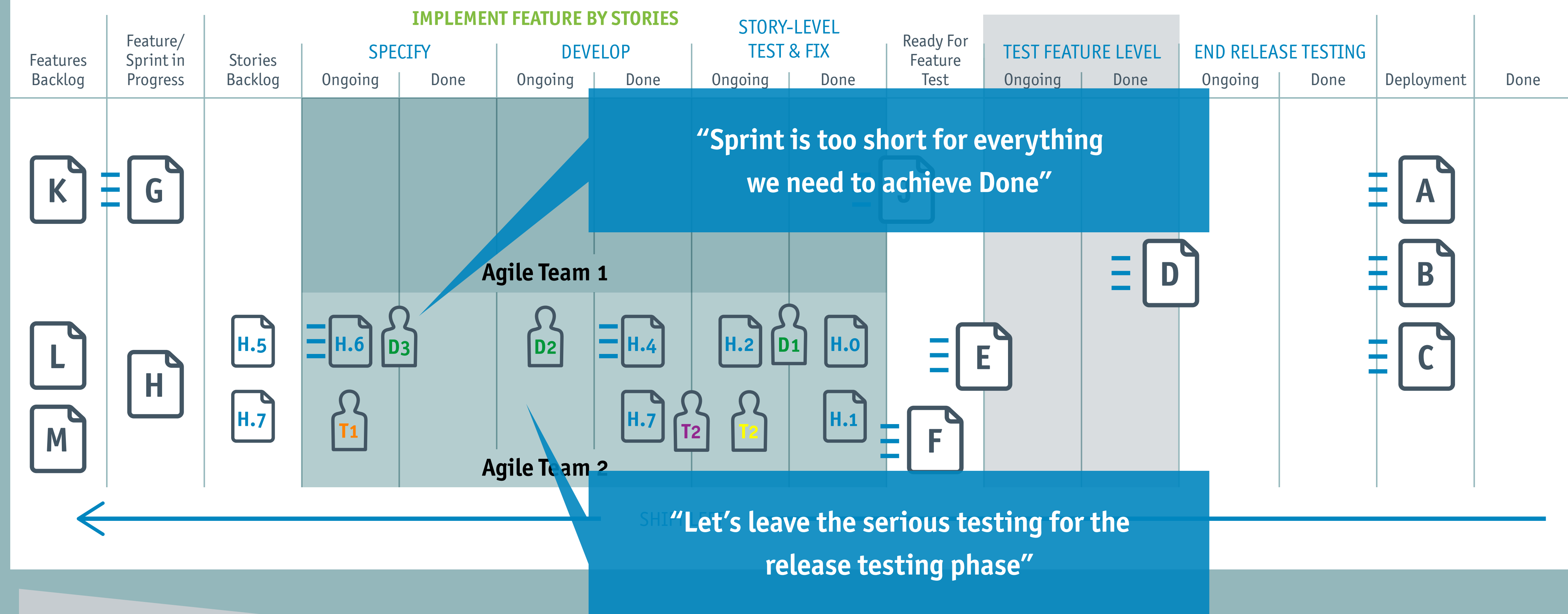
You can also look at your features and apply various Agile techniques to slice them into smaller features (also known as Minimum Viable Products) that can flow faster through the development pipeline to the point where testing takes place.

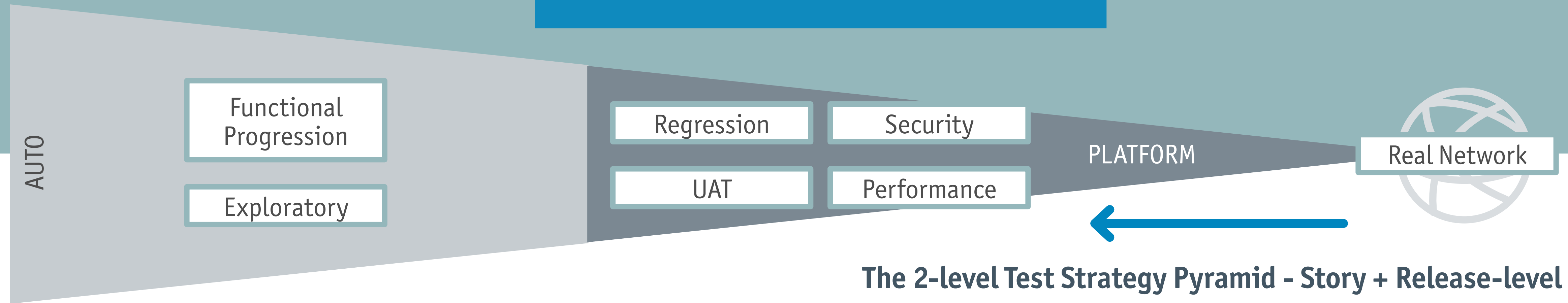




However, smaller batch sizes are not enough. No matter what the batch size, quality can only be built in if it depends on a Definition of Done that includes full test coverage and resolution of defects. So mark a feature or story “Done” only if it has passed all of its test coverage and if all the defects that need to be fixed pass release-grade criteria. (For project management purposes, track which features are done and which are still in progress.) The important point to note is that even if stories have been completed by an Agile team, the feature itself will be considered just a work in process.

Quality can only be built in if it depends on a Definition of Done that includes full test coverage and resolution of defects.





The 2-level Test Strategy Pyramid - Story + Release-level

Here's where most people say: "That's way too tough! We cannot get to 'Done Done', including all the coverage, every two weeks!"

They'd be right: it's not very pragmatic to reach "Done Done" in a two-week sprint. However, this is exactly where we should recall the batch size tradeoff curve we discussed earlier.

Theoretically, Done Done would mean full coverage — the one that you run before release and that covers the following testing types: progression, regression, exploratory, usability, performance, other non-functional testing, full user acceptance testing, and full compatibility matrix.

Now try to make a separate tradeoff curve for each testing type. As mentioned earlier, the curve for performance testing can be different than for regression testing. Balance the type of testing with what aspects are a priority for the work at hand.

One of the key levers to improve your results is to identify cases where it makes the most sense to shift to a smaller quality batch size and not try to "boil the ocean" by forcing a tight feedback loop for everything. This is a modern version of risk-based testing.

To apply the risk-based method to holding costs, you need to:

1. Consider how much more expensive it becomes to fix defects you find in each of the test types the further you are from the time the defect was introduced.
2. For each of the testing types, consider what the transaction costs are each time you run it. Look for cases where the risk/cost of delayed feedback is high and the transaction costs are manageable and try to shift those testing types left into development cycle.
3. Ideally, you want to shift those tests all the way into the CI system, but if that's not practical then you can decide to run them every time a story is done (a matter of days) or every time a feature is done (a matter of weeks).
4. Look for cases where the risk/cost of delayed feedback is high but the transaction costs simply don't make it economically viable to shift the testing left. For these cases, work on ways to reduce the transaction costs by introducing more automation, training more people to run tests, or by creating a minimally viable test that uncovers the most costly types of defects.
5. This test coverage activity isn't a one-time event. Repeat it every couple of months.

Key takeaways:

- It's important to establish a way to visualize your flow of work (i.e. Kanban).
- It's realistic to think you can't do all the things in each sprint.
- Small batches still depend on a "Definition of Done" that includes test coverage and resolution of defects.
- Pay attention to the different costs associated with quality, such as the cost of testing more frequently (i.e. transaction costs) and the costs of delayed feedback.
- Balance the type of testing that you shift left with what aspects of quality are a priority.

> "Pay attention to the costs of testing more frequently (transaction costs) and the costs of delayed feedback."

Balancing WIP: Stop Starting, Start Finishing

After enabling faster feedback by building more quality into the build cycle, we need to ensure a healthy flow of features/ stories through this pipeline.

You frequently hear: *"We have nothing to test yet! The build isn't meaningful. Everything is coming in on the last couple of days of the release."* The antidote to this is to adopt a "stop starting, start finishing" mindset. Start to continuously manage the flow of features and avoid having "too many features in progress." The Kanban boards discussed earlier are the classic way to achieve this flow.

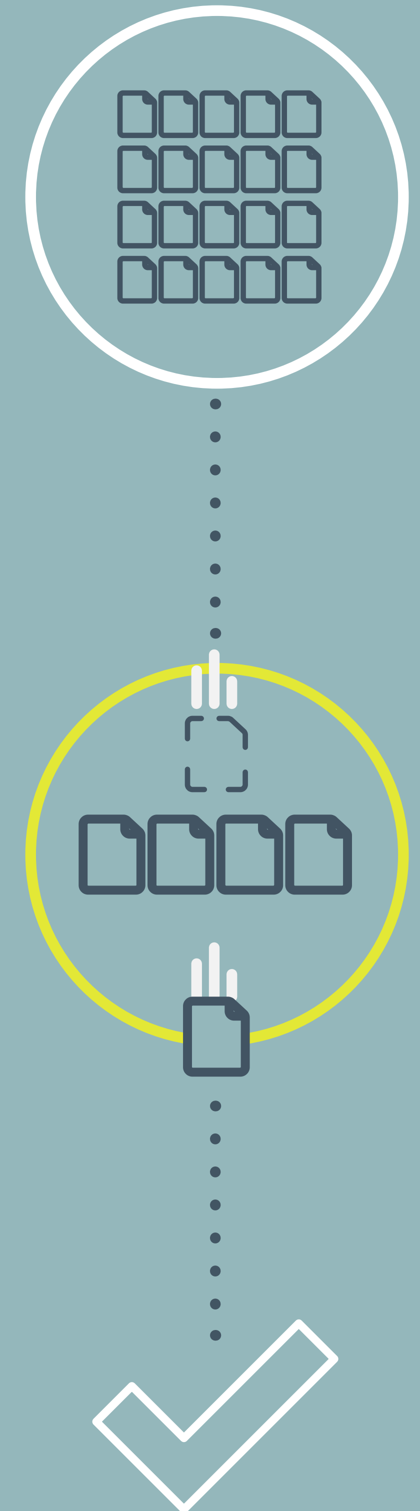
Another proactive antidote is to limit the amount of features that are allowed to be "in process" at each stage in the lifecycle — known as the WIP (work in process) Limit. If we decided the Development WIP should be four, once there are four features in development, we cannot start developing a new feature until one is pulled from development into testing. You should set your WIP Limit to what your team says they can actually do, and then seek to optimize it.

At the team level, you can continue to use this Limited WIP approach but this time at the level of more granular work items that flow from development to testing every couple of days. Agile teams typically use "user stories," which are small, testable, valuable slices of functionality.

Another popular alternative is working in "timeboxes." In Scrum, a team including developers and testers will look at a backlog of work items (typically these are user stories) and carve a list of items they will focus on for the next timebox (timeboxes are usually two weeks). The criteria for deciding how many items to focus on in a timebox is: "How many items can we get to Done?" It's not, how many items can be developed, or how many items can be tested. It's how many items we can design-develop-test-fix. Once teams create this list (often called the Sprint-Backlog or SBL), they should stay laser-focused on that list throughout the timebox.

Key takeaways:

- To ensure a healthy flow through the quality cycles, avoid having too much WIP (work in process).
- Teams shouldn't commit to more than they can realistically finish in the sprint. Collaborate to finish the high priority work you've already started.
- Scrum sprint planning is effective only when your "Definition of Done" for each story includes all the activity required for a quality result.
- Limiting WIP will result in some pains. It's crucial to deal with them to improve flow.



Taking Advantage of Bottlenecks

Regardless of the approach towards reducing the amount of WIP, there will be both positive impact (healthier workflow) as well as potentially painful adjustments to working in a more collaborative workflow. This is natural. However, you must recognize the impediments to leaner, more collaborative flow and work on removing them. If you don't do this, healthy workflow will be unachievable.

One example of a hardship is developers having to follow through to a complete "Definition of Done" rather than just "Coded." In many cases, this results in a testing bottleneck that prevents developers from starting new features. To alleviate a testing bottleneck, create a backlog of engineering investments that will improve testing capacity, usually by reducing the amount of work needed per feature.

The best example is, of course, test automation that can be developed by developers. This solves the slowdown problem. Involve your teams in building the Engineering Investment backlogs to increase their commitment to this approach.

Experience also shows that there's a higher level of commitment when working to alleviate rather than locally dealing with a problem. This is exactly where the transaction cost analysis you performed earlier becomes useful. The work you do to help reduce transaction costs and enable shifting left to smaller batch sizes is exactly the kind of work you want the team to take on when they see a backlog in testing.

Key takeaways:

- Limiting WIP is not easy, but it is rewarding if teams work to reduce the amount of work needed per feature (usually through test automation).
- WIP limits throttle development pace and align it with the testing pace. This often creates slack that can be redirected to improve the testing pace by forcing developers to write test automation code.
- To help developers deliver better quality code, have them focus on getting features to "Done" rather than "Code Complete."

To alleviate a testing bottleneck, create a backlog of engineering investments that will improve testing capacity, usually by reducing the amount of work needed per feature.

Beyond the Build and into a Culture of Quality

Bringing quality further into development is the main goal. But why stop there? The test scenarios we run provide useful information. Why not get this information before we do the technical design?

This is what Acceptance Test Driven Development (ATDD) is all about. It is an application of Test First, an approach that encourages discussing and defining test coverage before implementing code. Other Test First approaches include Behavior-Driven Development (BDD) and Specification by Example (SbE).

A culture of quality includes acceptance tests as part of the design and early specification stages. Having the discipline to specify concrete acceptance test scenarios in a collaborative discussion between product/business teams and developers and testers improves teamwork. Figuring out expectations up front will help teams reduce the amount of defects they have to deal with.

If continuous integration is building quality into the development cycle, ATDD/BDD/SbE is building quality into the design cycle. Both developers and testers benefit from the collaborative effort to define acceptance test scenarios from the get-go. Testers will have influence on the choice of acceptance tests and can focus on preparing to test the right scenarios with the right data. Developers can now code for the expected functionality rather than for the technical spec only. Developers will know what acceptance tests are expected to pass and testers can trust developers to deliver a higher quality build to them.

Some teams use ATDD/BDD tools to specify the acceptance test scenarios. The emphasis though should be less on tools and more on communication about what the acceptance criteria should be. Tools can support this process but not replace it. Typically at this point, the acceptance tests are specified at the "highlights" level. We use these acceptance test highlights to make sure our UI matches the business requirements.

Key takeaways:

- A culture of quality includes acceptance tests as part of the design stages with the aim of informing design rather than just validating it.
- Both developers and testers benefit from practices that encourage up front collaboration.

Both developers and testers benefit from the collaborative effort to define acceptance test scenarios from the get-go.

Discover more resources about Test-first approaches like ATDD, BDD and Specification by Example by visiting agilesparks.com

Using Test First Approach to Build Automation into Development

As described earlier, one of our key recommendations is to take a "Whole Team Automates as part of Done" approach to include more test coverage as part of CI testing. Yet this is challenging because it shifts ownership of automation from a testing silo to the "Whole Team." This will require a leap of faith to reduce the reliance on UI testing. This also requires a major shift in automation tools and skills that will challenge the comfort zone of developers, testers and automation experts.

But the "Whole Team Automates" approach has proven to be a strong enabler for building quality into the development cycle.

A key refrain we hear from testing organizations is: *"Testing is a profession. We're the experts on identifying the right coverage. If the developers are now taking on some of the testing and automation work, are we expected to trust that they'll think of the right scenarios? Won't quality be at risk?"*

One way the Test First approach helps here is that it provides a lightweight way for the testers to guide the developers' automation efforts. The risk is reduced because developers can focus not on prioritizing scenarios, but rather on the technical implementation of how to best test for those scenarios and how to come up with an ideal test automation system architecture that will enable fast, reliable, maintainable and extendable test automation.

At this point, many organizations choose to look at ATDD/BDD tools like jBehave, Cucumber, Robot, or FitNesse that are able to parse tests written in the business domain language of the acceptance test scenarios and execute calls to the system under test using test drivers like Selenium, SoapUI, Perfecto, or any other tool that provides an API.

"The 'whole team automates' approach has proven to be an enabler for building quality into the development cycle."

Teams using Test First approaches get much higher automated test coverage (some above 90%) than the typical test-last teams.

One big advantage of this ATDD/BDD automation approach is that it enables tests in a language everyone on the team (including the non-coders) can read, write and discuss. It also allows non-coding testers to participate in the automation effort. These testers specify scenarios. If they rely on existing domain language, they can directly execute the tests. If they need a new capability in the domain, they can request it and then ask a developer or automation engineer on the team to support this capability.

Teams using Test First approaches get much higher automated test coverage (some above 90%) than the typical test-last teams, both due to discipline and to developers' awareness of the acceptance tests.

Key takeaways:

- Team members need to get out of their individual comfort zones if they want to automate every part of the delivery chain.
- Testers can elevate their impact by using Test First approaches to help guide developers' automation efforts.
- Using the language of your domain through ATDD/BDD tools makes it easier for teams to collaborate on quality.
- Real world teams using Test First approaches get much higher automated test coverage (some above 90%) than the typical test-last teams.

A New Kind of Testing Professional

Test automation is clearly a major enabler for building quality into the build cycle. But what does that mean for the testing engineer role? One common answer is that all testers need to become proficient in test automation.

So now the question is what does test automation look like? If you look at the classic commercial test automation tool, you will see a record and play combined with a scripting language — designed for non-developers. But test automation suites that rely on such approaches are brittle and costly to maintain.

The situation worsens when you try to apply the "Whole team ownership of automation" principle. The typical developer has a distaste for commercial test automation tools. So more organizations are using a new breed of test automation tools that have a developer-friendly interface with integration into their IDE, APIs/SDKs in their favorite programming languages, and a better-architected structure.

The downside is that this makes life more challenging for the tester. Even scripting-level test automation skills are a non-trivial thing to ask of the typical test engineer. There aren't many super-testers who can shine a light on the right coverage and write effective test automation code. It's a lot to ask them to also develop in a full-fledged programming language such as Java, C# or Ruby.

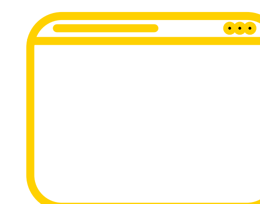
One of the advantages of the ATDD/BDD tools described earlier is they help conquer the divide between the actual automation code that talks to the system under test and the specification of the test scenarios. We can now ask the testing expert to focus on the high-level acceptance test specification. The test automation itself will be built by the developers or test automation engineers.

What is becoming clear is that the classic low-cost manual tester is struggling to fit into this new "build quality in" world. In most cases, manual testers are left to focus on those high-cost tests that are, at least for the time being, left out of the development cycle.

Key takeaways:

- Finding testing experts who are also automation aficionados is a struggle for many organizations.
- Don't expect testers to be adept at writing code.
- ATDD/BDD can act as a bridge between testing technologies and testing skills.

More organizations are using a new breed of test automation tools that have a developer-friendly interface.



Conclusion

If you're not working on shifting left, you risk escalating release costs and failing to keep up with business demands. However, this is easier said than done — we're fighting against strong economic and psychological forces.

The key to success is to take a set of evolutionary steps and shift more and more quality aspects into the build cycle and even upstream into the design cycle. This is a transformation. It requires thoughtful change management and leadership. It is not something the development and testing leads can undertake on their own. It requires a group of leaders to get into a room, chart the way and create a coalition to support the change.

About the author

Yuval Yeret is a senior enterprise agility coach and head of the US office for AgileSparks, an international Lean Agile consulting company with offices in Boston, Israel, and India. Yuval is leading several strategic long-term scaled Lean/Agile initiatives in large enterprises such as Siemens, HP, Amdocs, Informatica, Intel, and CyberArk, among others. Yuval is a big believer in pragmatic, best-of-breed solution design, taking the best from each approach. He is a recipient of the Brickell Key Award for Lean Kanban community excellence and is the author of "Holy Land Kanban" based on his thinking and writing at yuvalyeret.com.

About Perfecto

Perfecto enables exceptional digital experiences. We help you transform your business and strengthen every digital interaction with a quality-first approach to creating web and native apps, through a cloud-based test environment called the Continuous Quality Lab™. The CQ Lab is comprised of real devices and real end-user conditions, giving you the truest test environment available.

More than 1,500 customers, including 50% of the Fortune 500 across the banking, insurance, retail, telecommunications and media industries rely on Perfecto to deliver optimal mobile app functionality and end user experiences, ensuring their brand's reputation, establishing loyal customers, and continually attracting new users. For more information about Perfecto, visit www.perfectomobile.com, join our community follow us on Twitter at @PerfectoMobile.

Get content just like this delivered to your inbox!

www.perfectomobile.com
781.205.4111

© 2016 Perfecto Mobile Ltd.