Can Adopting a Shift-Left Approach to Testing Shift Left too far?

David D. Winfield

David_Winfield@vanguard.com

Abstract

Adopting a Shift-Left Testing approach is seen as a positive trend in software development. Integrating quality deeper in the software development lifecycle helps reduce costs by finding and fixing bugs earlier in development; however, the notion of shifting left can be misleading, moving the act of testing left instead of the process of testing. This puts more dependence on developers and may lead to less involvement from the tester, potentially weakening the testing program. Shift-Left Testing may be seen as a test automation-only approach, which could increase this dependency. What elements of the testing lifecycle might be missed, overlooked, or thought of as process “dead weight” if an organization adopts an automation-only approach?

Through use of the V-Model and Testing Pyramid, this paper will attempt to illustrate some parts of Shift-Left Testing that may make a testing program vulnerable. Finding defects is as much about analyzing the software with professional skepticism, subject matter expertise, and understanding the practice of software development as it is finding the defects fast and cheap. Incorporating developers in the testing lifecycle is an improvement in quality and a way to increase collaboration, and this combination is key in developing a more comprehensive testing program.

Biography

Dave Winfield is a Scrum Master and Technical Manager at The Vanguard Group. He has been working in software development and delivery for over 30 years in development, project management, and team management.
Introduction

Software development organizations are always looking for ways to improve the software development process. Over the past 20 years, Agile methods have helped improve software development by getting developers closer to the customer, but another process improvement—Shift-Left Testing—was introduced at about the same time to get testers closer to the developers.

Shift-Left Testing was introduced as a concept\(^{(1)}\) to show how moving testing from a reporting function to a cooperative effort with development could improve development outcomes. While the definition of Shift-Left Testing was centered on cooperation with developers, a later model continued testing’s shift left to user requirements \(^{(2)}\).

It can be argued that Shift-Left Testing has had the additional benefit of making the developers more responsible for the quality of the code; however, moving the testing left may have the unintended (or, in some situations, intended) consequence of reducing the role of the tester.

Shift-Left Testing

Shift-Left Testing, as introduced in an article by Larry Smith in 2001\(^{(1)}\), describes “linking” the two functions of testing and development\(^{i}\). He wrote that this would reduce costs by finding bugs earlier. In his experience, Quality Assurance (QA) was more a reporting function, producing detailed bug reports for management as a way in gauging quality. His idea was to find the bugs earlier in the process in a cooperative arrangement that would be less about pitting development against testing and more about finding bugs earlier in the code where they would be cheaper to fix.

Smith continues that automation should be a key element of the testing plan. This is vital to his idea that QA should not be running scripts manually but instead these scripts should be automated. Not only was manually running scripts costly to the organization, it was also not a good use of the tester’s capabilities—two concepts that continue to be key components in a good testing program. While not all testers would (or could) build automation (“developers are good at developing, testers are good at testing”\(^{(1)}\)), they can contribute ideas to automation. This article resonates true today where organizations continue to use testers to manually run scripts.

Finally, Smith writes of building “testable code” and automating reporting, and he describes requesting certain log messages to be included in the code he would later analyze. Testable code includes other good software engineering techniques such as making the code easy to maintain and loosely coupled\(^{ii}\). By building testable code, it’s easier and therefore cheaper to build the automation.
The V-Model and Four Phases of Shift Left Testing

One way to visualize the software development lifecycle is through the V-Model\(^3\), a graphical representation of the software development lifecycle. This model shows the software development lifecycle in increasing granularity from requirements to coding down the left side of the “V,” while each corresponding testing phase is on the opposite side going up the right side. Donald Firesmith, of CMU’s Software Engineering Institute, described the implementation of Shift-Left Testing in four distinct models: Traditional, Incremental, Agile/DevOps, and Model-Based\(^2\). The first three “concentrated on beginning testing of the software earlier in the development cycle” \(^2\) as described in Shift Left Testing by Larry Smith; however, Firesmith’s fourth phase, Model-Based, tries to address testing before software is developed.

![V-Model Diagram](image)

Source: V-Model from Donald Firesmith’s Four Types of Shift Left Testing \(^2\)

Model-Based Shift-Left Testing looks more deeply at requirements, architecture, and design. Firesmith sees this fourth stage as a goal for those on the Shift-Left Testing journey. In this stage, he says, there is testing applied at all levels of the left side of the V as soon as that stage is complete and before code is written.

The Testing Pyramid

Smith’s Shift-Left Testing, as well as the model-based extension proposed by Firesmith, clearly and respectively articulate the need for testing to be early—and even earlier—in the software development lifecycle and to include automation as much as possible. Unfortunately, in the early days of Agile (in this case, Scrum) testing followed the development of executable code and therefore followed in subsequent sprints. Scrum specifies the delivery of completed software, so testing is critical if development is to be complete; however, if it’s follow-on testing (i.e., testing the “completed code” in the next Sprint), how can the delivered software be considered complete? Mike Cohn popularized the Testing Pyramid in “Succeeding in Agile” (2009) \(^3\) to help Agile development teams conceptualize the quantity of automated...
tests they should be applying at each level: Unit, Service, and UI. While there has been some controversy regarding the Testing Pyramid, it follows to put more effort into tests that are easier to write and maintain.

![Testing Pyramid Diagram](image)

Source: Mike Cohn’s Test Pyramid

Test automation was intended to eliminate manually running the test scripts, the most expensive part of testing. The Test Pyramid’s depiction was intended to show where the development team should focus their efforts, writing more tests at the Unit level and fewer UI tests (the top layer), as tests at the UI level are more expensive and complex to build and maintain.

The Testing Pyramid was introduced as part of Agile methods to help emphasize automating tests, but it’s important to remember that with this automation comes development and maintenance—keeping each layer of tests updated with the new requirements and code. Further developments, such as Test Driven Development, are making maintenance of these tests more integral to development which should be less costly.

Shift-Left Testing highlights the need for testers to be more closely involved with specifying the tests even if they don’t have the coding skills to implement them. Collaboration is a key component in Shift-Left Testing just as it is in Agile development, but the Test Pyramid does not specify interaction between developers and testers or the tester’s role in developing the tests.

**Exploring Gaps**

In summary, Larry Smith’s original article described shift-left testing as collaboration between testing and development, increased test automation, writing testable code, and automated reporting, all to find and fix bugs earlier where they are cheaper. He was clear that testers bring certain talents to software development that developers (not all) are not good at or may not be interested in:
“Development programmers are quite simply not good at finding bugs. Nor should they be. Consider the mindset of development programmers: They need to be good coders and good bug fixers—but if they are good at finding bugs, they should be migrating into QA. That is where they can do the most good with such a skill, after all. We should, therefore, not expect development teams to be good at finding bugs. If your management is doing its job, they won't be, almost by definition.”

As Agile methods became more prevalent in the industry, the Testing Pyramid guides teams to increase unit test automation with less emphasis on UI or end-to-end testing automation (while there is value in UI test automation, we value unit testing more). This emphasis, however, can lead to reducing or eliminating the role of the tester on development teams.

“The Four Types of Shift Left Testing”, in contrast, move testing even further left (based on the V-Model) to user requirements and user design, while advocating for continued test automation. Firesmith describes the tester and testing in the role of experimentation on the executable code to ensure it operates correctly and this should include executable requirements.

What happens if an organization thinks it can replace testers with automation at each level? What does an organization lose with this elimination?

Removing Exploratory Testing

A key role for a tester is exploratory testing. Even though automation is a key tenet in Shift-Left Testing, it does not provide for complete coverage of the system. The definition of testing is experimentation, implying an analysis that can’t be performed through scripts. James Bach explains that exploratory testing is “simultaneous learning, test design, and test execution”. As teams shift left, the importance of exploration becomes critical in the collaboration between developers and testers, as testers can provide additional tests to the developers for automation or input into Test Driven Development as they explore the stories in the Sprint. Removing the tester removes this unbiased critical thinking.

Test Driven Development (TDD) allows the developer to integrate testing in the process of writing code, which keeps the functioning code and the test code in the same workspace. In TDD, developers can’t write code without writing tests first; however, these tests are points-in-state and are very granular. As tests are developed, moving up the Testing Pyramid, the tests become less granular, harder to maintain, and represent states of development and functionality further apart. Maintenance of these tests becomes increasingly difficult, so the tester’s role is more important to fill in the gaps between test scripts.

Exploratory testing is not random or ad-hoc (as it might be called) but rather reactive to the software being tested. Testers can adjust; while they may not run scripts as precisely as a machine, the tester may notice bugs that would have previously been cast as script error.
Collaboration with Development

Who knows the software system better than the Product Owner? Generally, testers do; they are usually left on the periphery when requirements are written but have unique angles on the software from their experience testing it. New Agile approaches are geared toward automation and are described from the product development perspective more than from software development. A tester’s collaboration in product development allows for identification of missing or lacking requirements. Testing can adhere to company standards working with Product Owners to reduce requirements rather than increase them to meet the goals of the customer.

Conclusion

Shift-Left Testing is a key component to an organization’s approach to delivering not just bug-free software but also products that continue to delight the customer. Organizations should be aware of the potential pitfalls in removing key components of the testing infrastructure, including the testers themselves. Removing testers from the development teams removes an unbiased perspective and critical thinking. Using a “Model-Based”, from Firesmith’s paper, approach would incorporate testing at all levels of the SDLC (Software Development Lifecycle) and provide a collaborative relationship between testers, developers, and product owners. Finally, it is important to understand the difference between just running scripts and exploring the software in new ways.

Shift-Left Testing is about shifting the testing program to the left, collaborating with development and product design, and finding bugs before a single line of code is written. Taking Shift-Left Testing too far would be to focus solely on automation, potentially leaving out the tester’s unbiased, critical perspective.
References


2.) Donald Firesmith, 2015, SEI Blog, Four Types of Shift Left Testing (cmu.edu) (accessed July 2021)


---

i Mr. Smith’s paper uses QA and testing interchangeably.
ii Like a lot of terms used in software development, there does not seem to be a single definition of testable code but rather an agreement that code written to good engineering principles, like SOLID, will be easier to test and maintain.
iii Visually, the V-Model represents the SDLC but with the associated testing phase and distinction between verification and validation. (7)
iv Firesmith does a great deal more as he describes the difference between quality assurance and testing, and in fact suggests the use of the term shift-left quality assurance or shift-left verification, which just doesn’t have the same ring to it.