Master Your “Shift Left” Moves

THIS CHAPTER WAS CONTRIBUTED BY MANISH MATHURIA, FOUNDER AND CHIEF TECHNOLOGY OFFICER AT INFOSTRETCH

MANISH MATHURIA has more than 14 years of professional and management experience in consulting services. With his vast knowledge and considerable experience in technology, Mr. Mathuria leads the Infostretch team in developing architecture, scalability, performance, and IT strategies for Infostretch customers. He has been a senior technology consultant in Quality, SaaS, and Mobile areas for the executive management in Fortune 500 firms and high-tech Silicon Valley startups. He speaks regularly at leading quality and software engineering conferences worldwide. Mr. Mathuria holds an Executive MBA from Santa Clara University and a Bachelor of Engineering in Computer Science degree.

INTRODUCTION

Let us examine the QA organizations of two imaginary companies — Amazing
Auto Intelligence (AAI) & Bricks ‘n’ Mortar Depot (BMD). They are leaders in their industries. AAI makes next generation technology for self-driving vehicles and BMD makes designer bricks. Both companies have significantly large software organizations. Both have imperatives to deliver high-quality software, as their business and competitive environment demands rapid delivery of software releases in order to remain competitive.

However, AAI and BMD write software using very different methodologies.

AAI has coordinated software releases across the company. They have divided their development teams into small scrum teams, about eight people each. As far as QA teams are concerned, AAI has no specialized QA in any scrum team. They follow the Behavior Driven Development (BDD) methodology where the scrum team develops acceptance tests as Behavior Driven Scenarios with relevant test data before they write code to implement the features.

Through the sprint, their objective is to make these tests pass. Through the sprint, their product owners and developers continuously review the scenarios and data through peer reviews and use these scenarios to discuss their progress. They develop the scenario code while they are building the feature code. Their Continuous Integration (CI) system builds the code several times a day, checks the code quality and runs the scenarios, and while initially during the sprint most scenarios fail, towards the end they begin to pass. Their sprint is not done until all scenarios pass. In addition to the BDD scenarios, developers write unit tests which also run through their CI builds.

Once the CI system successfully passes the “quality gate (the acceptable no passing tests)”, it promotes the software to an integration test environment that it builds “on the fly” in the Cloud. This environment is torn down after tests are run. In the integration test environment, more tests are run which verify integration among different deliverables of the scrum teams involved. They have a small end-to-end test team which finally verifies the integration manually after integration test quality gate has passed successfully as well. After the end-to-end test phase, the software gets promoted to production.

AAI does have a QA tools team which is responsible for building frameworks, reports, and tools. This team ensures that repeatable tasks which each scrum team does are eliminated and done by the tools team.
BMD builds software very differently. Their development team is in New York. Their QA team works from Dallas. They follow an iterative development process which releases builds to QA a couple times a month. The first few days of a release to QA are frustrating because of manual installation procedures that often fail due to several setup issues. The interaction between dev and QA often turns hostile and at best is tense. There are several rejected releases to QA that cause a lot of development rework. The QA team focuses on testing the prior released iteration and they automate tests once the graphical user interface (GUI) is complete. This automation is largely used for regression testing during the release.

AAI can release software to production every sprint. BMD can barely release software once a quarter. With the same number of people AAI delivers much more featured and complex software with far more acceptance on the marketplace than BMD.

While stereotypical and extreme, these examples are not unique. We have several AAIIs and BMDs among us. This chapter examines which are the traits that AAI has that BDD does not? What are the success patterns of high-performing QE function?

“SHIFT LEFT” EXPLAINED

“Shift Left” is a practice that provides an effective means to perform testing along with or in parallel to sprint and development activities.

- An entire Agile team comprising of development, test, and operations engineers works together to plan, manage, and execute automated and continuous testing to accelerate feedback, such that it can be used to optimize the development artifacts and process.

- Technology is often used to automate the software development lifecycle. Typical candidates for automation are builds, unit, integration and acceptance tests, and code quality inspection. This process of continuous integration is often staged, with each stage having a quality gate of acceptable criteria for promotion to the next one.

- An essential tenet of “shift left” is to write testable code that is unit verified
and build quality is guaranteed from component up to integration level. Essentially, this enables efficient localization of a problem. It also ensures that individual components are working before a large amount of software has been integrated.

- Testing may be performed as a part of the development process or as a service running in parallel to development activities. In either case, shifting left accelerates feedback to developers and improves the quality of code delivered for testing.

“Shift left” is an essential paradigm for Agile teams. It may be successfully leveraged by all development teams.

**IS IT NEW?**

No, not really. The ability to test a component and assess its validity has always been practiced at hardware level. For even basic hardware, every component is designed to self-test on power on and declare itself functioning or not. In case of an issue, you clearly know where the fault is. Before you assemble the whole solution and integration test it, you know that components themselves are working.

Shifting QA left helps achieve some of the same benefits and outcomes this component level testing has enabled in hardware assembly.

Why is it that this seemingly obvious practice that has helped hardware development has not been adopted by software teams? There could be several reasons.

a. Software is easy/easier to fix after production through releasing a patch, for example. Hosted software, such as in Software as a Service (SaaS), makes it even easier. You do not have to ship anything, you just roll out the next daily update and the case is closed.

b. Software requirements, development, and releases are more complex and volatile. This has caused software teams to test things in more of an ad-hoc way in the name of completeness and quality.

Demand for Agility is putting a different level of pressure on QA teams. Agile teams are expected to deliver working software in a sprints lifetime which typically lasts for a week or two. What used to be tested in weeks or months now
needs to be performed in a matter of days, while final certification is sometimes even done in minutes. Unless teams adopt automation and component level testing, this is an impossible task.

“Shift left” is not just about automation. It is a state of mind.

KEY BENEFITS OF “SHIFT LEFT”

Enable Agile

- Enable in-sprint testing and automation with a clear definition of done (DoD).
- Release software in production incrementally with every sprint.

Find defects early

- With testing happening sooner in the life cycle, you find defects sooner. It is a well-known truth: the earlier in the life cycle you find a defect, the cheaper it is to fix it.

Improve communication and feedback

- With QA engaging earlier in the life cycle, developers and product owners get early feedback. This helps teams identify considerably faster feature ambiguities, inconsistencies among sub-systems, integration problems, and data incompatibilities.
- This early feedback increases feature velocity and reduces cost.

Reduce number of production defects

- Continuous quality maturing all the way to production.
- Dramatic quality improvements earlier in pipeline to reduce risk to production.
- Clear quality gates to migrate software closer to production reduces ambiguities and increases quality.

GETTING READY TO “SHIFT LEFT”

It is important to know how Quality organizations should evolve to the stage of being efficient and shifted-left. There are specific dimensions that an organization should look at to enhance their adoption of “shift left”. This section examines the maturity of QA organizations through the lens of these dimensions.
QUALITY ENGINEERING MATURITY

“Shift left” is not a trivial change. It has far reaching impact on how QA teams are structured and skilled; also on their operating processes and best practices. The sketch in Figure 89 classifies the maturity of QA teams according to various “shift left” dimensions in three categories “basic”, “intermediate”, and “expert”. To conquer “shift left”, QA teams need to rise to at least an “intermediate”, if not an “expert” level.

![Figure 89: Maturity of QA Organizations According to “Shift Left” Dimensions](image)

QUALITY ENGINEERING PROCESS

Quality process engrained in the overall Agile software development process is a major factor. This includes structure of teams, approach to, and integration of quality into the software process.
AUTOMATION METHODOLOGY

The ability to automate scenarios in a structured, maintainable, and usable methodology is also a significant factor.

SCALABILITY OF TEST INFRASTRUCTURE

The modern agile QA teams can fire up their automation assets on on-demand infrastructure in such an automated fashion that a large number of automated tests can be executed in a massively parallel way.

CI/CD AND DEVOPS INTEGRATION

Automation is ineffective if it is leveraged manually. The process of Continuous Integration and DevOps is the consumer of automated assets. At the end of the day, it is that process flow that actually manages and consequently automates the end-to-end test execution and the reporting based on it!

TEST ENVIRONMENT AND TEST DATA

Finally, if you have far reaching challenges with test environments and test data that prevent seamless and reliable automation runs, no matter what or how much you automate, execution will be bottlenecked and the desired goal of certifying a build automatically will remain underachieved.

Table 6 explains the tenets or characteristics of QA teams at different maturity stages. Effectively, Agile teams are between an “intermediate” and “expert” level.

In the next sections, we will learn how to achieve the “expert” level by using the “shift left” techniques.
Table 6: Characteristics of Different “Shift Left” Maturity Stages Explained

**“SHIFT LEFT” — PATTERNS AND ANTI-PATTERNS**

Traditionally, in the Waterfall project management model, software is conceived of (in the form of some requirements), developed, tests are conceived of, executed and software is fixed in a cyclical order. When you try to do this cycle in a shorter (say a sprint) span, it becomes very painful for the following reasons:

- Unlike Waterfall, a sprint’s time frame in agile does not give you enough time to start thinking about the tests after development is done. QA teams are forced to write tests and automate them while code is being developed.
- This cycle time also puts pressure on the entire team for effective reviews. There is no easy way to pull in the entire team to purposefully discuss test approach, test cases, and test data.

The above restrictions evolved into various test-driven patterns described below:
TEST-DRIVEN DEVELOPMENT (TDD)

This turns the aforementioned traditional software development process upside down. Instead of writing code to a spec, you first write the test, think about its data inputs and various boundary conditions under which it would fail the spec. As the code comes alive, you also try to automate these tests. Also, the emphasis is to continuously execute these tests. Clearly, before the code that implements a feature exists, the tests will fail. This is expected, but as you come close to functioning code, the tests start to pass. You perform the above process in small incremental steps.

ACCEPTANCE TEST-DRIVEN DEVELOPMENT (ATDD) AND BEHAVIOR-DRIVEN DEVELOPMENT (BDD)

These are special cases of TDD. ATDD and BDD\(^1\) are mostly synonymous. Teams practicing ATDD would write an acceptance test describing the behavior of the user story or functionality that needs to be developed. BDD enables the team to describe the test and hence the software behavior in a Domain Specific Language (DSL) called Gherkin which reads very much like English.

The BDD scenario is made executable as a test using frameworks like Cucumber (Ruby) and behat (PHP). An executable BDD Scenario is a powerful tool to automate tests, as everyone can see exactly what the automated test does with its data by simply reading the test. It hides the automation details within its implementation. Compare this approach to the one which translates a manual test case in a programming language, such that eventually breaks the link between manual test scenarios (their intent) and the corresponding implementation. Now you can better appreciate how AAI and BMD above are different.

Following is an example of a BDD and its implementation:

---

\(^1\) Behavior-driven development — https://en.wikipedia.org/wiki/Behavior-driven_development)
Story: Hotel rooms when cancelled go back to inventory

As a hotel manager,

In order to keep track of available hotel rooms,

I want to add rooms back to inventory when they are cancelled.

Scenario 1: Cancelled hotel room should be returned to inventory

Given a customer previously booked <no_of_rooms> rooms of <type> rooms,
And I currently have <Inventory_no> in inventory,
When she cancels the rooms,
Then I should have <upd_inventory> of <type> in inventory,
And the customer gets her money back in her account.

<table>
<thead>
<tr>
<th>no_of_rooms</th>
<th>type</th>
<th>inventory_no</th>
<th>upd_inventory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Queen</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>King</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

//Implementation

@Given("a customer previously booked $no_of_rooms rooms of $type rooms ")
public void theRoomBaseCondition(int no_of_rooms, String type) {
    ...
}

Figure 90: An Example of a BDD Scenario and its Implementation

SPECIFICATIONS BY EXAMPLE AND EXECUTABLE SPECIFICATIONS

This is a state reached when BDD scenarios adequately describe the acceptance criteria in Gherkin, so that the entire software test scenarios/behavior specification is documented. These test scenarios are further annotated with the required data that describe the software behavior under certain conditions. Combined with the data and scenarios, the specification reaches a state that lends itself to be an effective test review tool, one that is crucial to bringing everyone in the team to the same page.

In the example depicted in Figure 90, the clarity of the intent is quite evident. This is a point to remember for test engineers.

To wrap it up, it is easy to see that TDD and its flavors help you “shift left”.

186
Fundamentally, by forcing one to write tests before you code, you are naturally enforcing “shift left”. BDD helps you automate what you see by creating executable specifications and executable test cases. Teams which embrace TDD/BDD practices commit to early testing in their software development lifecycle. This means that they also embrace the philosophy that testing is a problem for the whole team to solve.

SOFTWARE TESTING PYRAMID AND “ICE-CREAM CONE”

The term “Software Testing Pyramid” was first conceived by Mike Cohen in his book “Succeeding with Agile”\(^2\). The anti-pattern “Ice-cream Cone” was introduced by Alister Scott in his blog.\(^3\) Figure 91 below visualizes both.

![Figure 91: Software Testing Pyramid and “Ice-cream Cone” — Side by Side](image)

Before Agile is embraced, most teams are structured like an ice-cream cone, including our friends at BMD above. They are heavy on finished product testing with an emphasis on manual testing. While that works, it does not align with the goals of Agile and “shift left”. The time taken and the cost to perform testing manually is high. Note that their efforts in automation are also lopsided towards GUI automation. There is nothing wrong with the GUI automation. However, only automating the GUI is not sufficient. It is done late (GUIs get built last), it is fragile (GUIs change a lot), it is slow to execute and, at the end of the day,

---

2 Cohn, M. (2009), Succeeding with Agile: Software Development Using Scrum, Addison-Wesley Professional

3 Scott, Al., Ice-cream Cone, [https://watirmelon.blog/2012/01/31/introducing-the-software-testing-ice-cream-cone/](https://watirmelon.blog/2012/01/31/introducing-the-software-testing-ice-cream-cone/)
it is expensive.

Overall, the ice-cream cone anti-pattern does not go well together with Agile. The “software testing pyramid” pattern, however, does. It simply states that testing should be structured in a bottom-up way with the heaviest emphasis on automating unit tests to increasingly lighter on automated API/Services/Components, automated GUI, and lightest on manual tests. The pyramid also provides recommendations on the amount of testing that is supposed to be done at each layer in %.

Note that neither model eliminates manual testing. Often, manual ad-hoc testing covers answering the question “Are we building the right system?” as opposed to the question that testers are typically tasked to answer, “Are we building the system right?” Manual testing should be used for non-automatable corner case scenarios and visual deformities. For a further discussion on visual automated testing, please refer to the respective section “Automated Visual Testing”.

**PRACTICAL TEST APPROACH AND TEST TOPOLOGY**

Often, teams migrating into Agile and automation struggle with the right test topology. They know that they do not want an ice-cream cone, but they cannot find their way to the pyramid. Also, QA organizations cannot or do not control developers and therefore they lack a say on how much and how effective unit testing is.

Below, you will find a list of practical rules with good track record in the industry, that is, they work. The diagram in *Figure 92* maps them to the paradigm of the “Software Testing Pyramid”.

1. If you do not have control on unit testing, plan your testing approach treating code as a “black box”. This may not be practical all the time and you may find yourself depending on developers. Do not miss the section on “Inserting Testability in Software”.

2. Assuming you have access to the service layer and it is well-defined (with the advent of micro services patterns, most modern applications are developed with a well-defined services layer), plan to test this extensively.
Services layers are typically exposed as Web services, APIs or SDKs. A huge number of functional tests with relevant data combinations and under certain boundary conditions should be done here. If done right, this could form 50–65% of the functional coverage.

3. Leverage GUI automation to facilitate the execution of more end-to-end business scenarios, so that good coverage on GUI workflows, error handling, translation, and many more can be ensured. These tests could add up to 20–30% of your functional coverage.

4. The rest can be done manually.

**Figure 92: A Practical Proposal for Concrete Test Topology within the “Software Testing Pyramid”**

**INSERTING TESTABILITY IN SOFTWARE**

Following are some common challenges or observations in the process of writing test automation solutions:

1. Often, testers struggle with high maintenance of testing activities and test code. Tests fail frequently and unexplainably. Upon investigation, you find that developers changed some code or some configuration parameter.

2. The only way to validate that a test fulfills its desired function is hidden inside the “black box” of the code. It is not accessible to you as a tester. Therefore, you need access to the code.
3. Your test tools require a certain level of a build instrumenting the code.

4. Your test tool is unable to access GUI widgets or the widget library used by developers.

5. Developers encode the identifiers of GUI widgets in a way that they change rapidly. You are dependent on locating and accessing the GUI widgets to run your automated test.

In all of these cases, streamlining and enhancing the code development practices, build, or release processes would significantly reduce the testing effort as well as spurious test code failures. Thus, the test code becomes more resilient. Making the code more testable or increasing the testability of the software is a real issue.

The testers should discuss such challenges openly with the developers. After all, less tested or testable code would impinge on the overall quality. The senior engineering management will often understand the impact of less testable code. Its representatives will work with the engineering team to increase testability. Following are some good practices that can help you increase testability.

1. Use configuration management, CI, and automated builds to reduce human errors in configuration and builds. This approach is often paired with a smoke test that acts as a quality gate for the build.

2. Write special functions accessible only to the tester either at the GUI or at the services layer. Exposing some validation properties, setting/resetting internal states of the application, querying state of components/objects... the examples can be many more.

3. Use consistent naming and widget identification. Developers and testers should align and agree upon a reliable GUI widget identification methodology, one that is amenable for test tools to work.

4. Do special test builds.

**SPRINT TESTING TO REGRESSION TESTING PROMOTION PROCESS**

QA teams have two main jobs when it comes to a typical Agile development process.
1. To ensure that the functional and non-functional testing for the features developed in the sprint has been conducted to the satisfaction of certifying the sprint.

2. To ensure that the development process progresses well with each consequent sprint, i.e. the current sprint has not broken the software developed in previous iterations.

Figure 93 describes the typical role that a QA team has in the sprint towards the objective of automating as much testing as possible during the sprint. The patterns described in this chapter help choose an automation methodology that does not get in the way of this objective.

**In-sprint Activities**

- 1. Review Stories
- 2. Assess in sprint BDD and write BDDs
- 3. Write tests that can not/should not be automated
- 4. Change Regression BDDs to cover last sprint BDDs
- 5. Assess reuse from existing automation code and refactor if needed
- 6. Do not execute regression

- 1. In Sprint BDD development as much as possible
- 2. Fix Regression tests if need be in last sprints BDDs
- 3. Nightly regression analysis
- 4. Manual/Automated execution of in-Sprint BDDs

**Figure 93: Migration from Sprint Testing to Regression Testing**

Sprint tests are focused tests that validate and assert a specific need. For example, whether a user story was about to change the brand color from pink to purple. Imagine you wrote a test that automated this kind of validation. Such a test may find little relevance in the broad scope of regression testing. It is probably a validation point in some broader regression test rather than a standalone test scenario as it was during the sprint period.

Sprint tests often need to be reviewed, refactored, rewritten or deprecated post sprint. A retrospective period is the best time to do this or a team can set some time aside. Whenever done, it is important for teams to realize the difference
between sprint and regression tests and address the migration process.

**AUTOMATED VISUAL TESTING**

Visual testing is the process of validating that the application is visually functional. It asserts that no part of the application has been deformed, lacks information, does not suit its purpose or is in general inadequate to appeal visually. It is usually assumed that visual testing will be done manually.

Again, in the spirit of the principle of “shifting left”, it is important to find avenues that allow you to automate as much as possible.

Following are some techniques that help automate visual testing activities.

1. Every screen can be manually baselined (verified in advance) and visually inspected in real time by comparing it to the baseline screen. There are both open-source and commercial tools that make this technique possible. Sikuli, for instance, is an open-source image comparison technology. Applitools Eyes is a commercial alternative.

2. For the sake of validating HTML and CSS, there are open-source frameworks like Galen that allow you to code visual validation specs as part of the code. They are executed when the tests run, so that rendering in web browsers is verified at run-time.

Whichever technique you use, it is important to be careful with bringing in visual verifications and assertions to virtually every functional test. This will make your tests execute slower and with considerable overhead. Instead, the best practice is to identify a few end-to-end scenario tests that have been specifically written to provide broad coverage of the GUI and, hence, inspect it visually.

**SUMMARY**

Within this chapter, you got to know what “shift left” really means. More importantly, you are now aware of the buttons QA and development organizations can push, so that this transition can be as smooth as possible. Keep in mind that it has the potential to be a rather painful and tiresome one.

As always, the right mix of people, processes, and tools will save the day. In
any case, practice makes perfect. Stick to the already discussed approaches, techniques, and industry practices and soon you will master the moves of left-shifting.