A tester’s appreciation of unit tests

Katie L. Fox

katiefoot9@gmail.com

Abstract

Unit tests are the foundation of the test pyramid but are commonly seen as a developer’s obligation and not as a tester’s secret weapon for risk-based testing. During recent interviews with Quality Engineer candidates, I heard a recurring theme that unit tests are “tests developers write,” and it had me questioning my own contrary opinion. I questioned how I had arrived at those thoughts, and how I would encourage others to view them differently: as simple, powerful tests that increase in value with more visibility and more attention to detail during their creation.

Well-written unit tests provide a way to concisely understand the changes for a story without reading every line of source code. Applying risk-based principles and factoring in the areas of change indicated by unit tests could allow either for a reduction in testing scope, or it could reveal new areas that were not initially considered. It can also inform where test scenarios have already been partially or fully automated and thereby prevent duplicated effort and test suite bloat. If the “whole team owns quality,” and “everyone contributes to automation,” then developers and test engineers should be familiar with all levels of automated tests that ensure such quality. As part of a team’s effort to shift-left, everyone needs to include focusing on creating and improving automation in areas that will gain the most benefit.

This paper will discuss the benefits of shift-left testing via tester-developer collaboration around unit tests and show its successful applications in practice within unique team structures. It will provide frameworks to help developers to write readable, atomic unit tests and testers to decide which tests to automate at each level. Additionally, this framework provides less-technical testers with conversation starters to conduct code and test inspection verbally.

Biography

Katie Fox is a senior software test engineer at e-Builder.

She spent her post-college years learning and growing within the field of software testing at Ultimate Software (now Ultimate Kronos Group) and has been navigating remote work since 2016. She graduated from the University of Central Florida with a B.S. in Computer Science and no idea that her future career existed. Fast-forward to now, she loves balancing strong domain knowledge of the user experience with a thorough understanding of the technical implementation to determine how to layer exploratory and automated testing. She takes joy in catalyzing collaborative solutions to persistent team-wide problems... if two or more people have complained about something, it is time to fix it.

When not working, she likes to run, spend days in the mountains, cook, snuggle with her cat, and cultivate plants both indoors and in the patio vegetable garden.

Copyright Katie L. Fox, June 15th, 2021
1 Personal career foundation

My testing career at Ultimate Software (now Ultimate Kronos Group, or UKG) began once I passed two prerequisites: an internal training on testing techniques and a technical engineering interview. The training covered functional testing techniques to design exploratory test cases without seeing the code implementation. I left the training feeling capable of writing and executing manual test cases. The technical interview covered fundamental questions on OOP and emphasized that good object-oriented design was necessary for writing testable code. After the interview, I felt responsible for knowing enough about dependency injection and inheritance to hold developers accountable for using them so that my team could write unit tests. It took a year before I put this knowledge into practice, but it instantly changed my perspective on test engineering from “testing after the code had been written” to “influencing quality within the design on the software.”

In my early career, I was exposed to testing buzzwords, including agile, cross-functional teams, test pyramid, software development life cycle, and shift left, and learned the basics of what they meant.

- An agile team was responsive to changes in priority, learned details along the way instead of upfront, and broke down features into smaller pieces of work to deliver incrementally.
- A cross-functional team consisted of members that were business analysts, developers, and testers.
- A test pyramid (Fig 1) is a visual tool to represent the ideal distribution of test automation across unit, integration, and functional tests. Many unit tests form the base, fewer integration tests in the middle, and the smallest number of functional tests on top.
- Software development life cycle (SDLC) was the process by which a feature was delivered from the first requirements to functionality in production.
- Shift-left (Fig 2) meant ensuring that test automation was created early in the SDLC, so developers wrote unit tests and teams had automated tests to catch regression bugs and relieve the tester from repetitive manual test cases.

Despite understanding these techniques in isolation, it took several years of experience before I began to understand the interactions between techniques and how I could use them most effectively in my role. Once I did, I found that focusing too much on one technique can lead to gaps in quality, while focusing on using them all together can increase their effectiveness.

![Figure 1: The Test Pyramid](image)

*Figure 1: The Test Pyramid*. Vertical arrows indicate trade-offs between fast execution in unit tests for high confidence against regression defects in functional tests.
Across the company, development teams at UKG use the Kanban agile methodology to manage their deliverables for the company’s human capital management software products. This agile methodology emphasizes striving for consistency of work throughput to generate confident predictions of future delivery dates. Development teams were cross-functional and organized to work on features around particular product components. However, due to distinct/diverse technologies and testing challenges across all teams, uniformly applying agile principles and shift-left guidelines for testers was impractical, and team-specific/team-directed solutions were implemented. The flexibility afforded by team-specific testing guidelines resulted in both the challenge and freedom to figure out what worked best on an individual team.

The following procedure describes the high-level process for a development team to produce a new “feature.”

1) The team business analyst would split the feature work into “stories” that represented pieces of deliverable business value.
2) Each story would undergo a mini SDLC of requirements analysis, development, and testing to ensure the delivered code meets the business need.
3) Developers would write unit tests and review code changes for each story, and then testers would conduct exploratory testing and write functional automation.
4) Once story testing was completed, the tester would perform a live demonstration of the story’s business functionality to the business analyst before merging changes to the main repository.

Testers on each team managed feature test planning, functional automation strategy, and exploratory test sessions. Testing for individual stories included risk-based exploratory testing, ensuring automated coverage with new or updated tests and data, and reviewing risks with the analyst before merging the code changes. Testers also served as a source of knowledge for setting up scenarios needed to demonstrate new story functionality or reproduce defects. This consequently caused testers to frequently context-switch between roles as an individual contributor and a team-knowledge-resource. Contributing effectively as a tester, therefore, required a comprehensive understanding of their business use-cases, the underlying business rule functionality, how users interacted with the product, and how data influenced different test scenarios.

Testers cannot test everything. Automated tests provide increased confidence that code changes did not break expected behavior. In a traditional shift-left effort, developers write unit tests and testers write functional tests, but writing layers of the test pyramid in silo between roles results in a lack of cohesion within the test pyramid. Redundant scenarios or gaps in coverage occur but are difficult to identify.

Figure 2: Attention to Quality by Role and SDLC Phase as Shift-Left Testing changes are implemented. I learned shift-left as every individual role considering quality or writing tests earlier, but without the collaborative approach to testing that I found allowed us to achieve quality more efficiently.
The goal of this paper is to highlight the importance of collaboration between testers and developers when writing unit tests to increase the quality of team deliverables.

- Section 2 describes the structure and challenges of four of my prior teams at UKG, each with distinct team size, experience levels, and technologies. I will also describe team-level strategies for shifting left to address these challenges.
- Section 3 details specific strategies for shifting left employed across teams.
- Section 4 provides concrete examples of collaborative strategies to improve unit tests to further shift left.
- Section 5 describes ways that tester workload was reduced through collaborative unit test creation.
- Section 6 summarizes my experiences with unit tests and their potential benefits reducing the effort to achieve quality while maintaining or increasing the quality of deliverables.

2 Team Experiences

Described below are four teams at UKG Payment Services that exhibited unique structures and quality challenges which were addressed through collaborative shift-left testing, and my role in promoting this strategy to help each team deliver better quality faster.

2.1 Team 1

The team started small with four experienced developers and two novice testers with basic programming experience. We built a full-stack, event-driven web application with technology that was selected based on substantial use case research. During the months before the UI was created, testers wrote unit and integration tests for the backend services. We taught ourselves by reading existing tests for similar code, then copying and modifying them to fit the new methods. This allowed us to learn the data models and core architecture while the developers focused on writing the code and building the deployment pipeline. In the short term, a solid and coherent foundation of unit tests using consistent patterns was created and documented for future knowledge sharing. As the team doubled in size with both entry-level and established developers and testers, that foundation provided training resources in a clean code approach to aid the onboarding process. With this strong start, our primary quality challenge was to maintain cohesive testing practices as the team and codebase grew.

As part of one of the first shift-left strategies, the developers began conducting “desk checks” (post-coding demo to analyst and tester) to ensure the basic requirements satisfied the analyst’s expectation and to give the tester a head start on more complex test planning. Once the user interface development began, testers shifted unit test responsibility over to developers so that we could focus on our UI testing infrastructure. During this shift, testers collaborated with developers after the desk check to plan or review unit tests so that we knew what was covered before testing the story. Understanding where developers implemented unit tests in their code enabled testers to design more efficient integration and functional tests and ensured that each layer of tests added value.

2.2 Team 2

This second team began with an experienced group of five developers and two testers, then doubled in size within four months. The team inherited part of a full-stack, monolithic web application and was tasked with refactoring and extending its functionality to integrate with existing systems and process time-sensitive financial records. Because we did not design this solution from scratch, we spent the first months simultaneously learning its architecture, modifying code to meet business needs, and backfilling initially nonexistent test automation. Testers focused on manual exploratory testing as we familiarized ourselves with the new domain and set up UI testing infrastructure. Developers implemented necessary
features and attempted to reduce the deficit of unit and integration test coverage, but this invariably required extra effort to make the code testable. During the first year, the team focused more on keeping up with feature deliverables than improving and paying down technical debt to improve testability. The primary quality challenge was to efficiently design manual tests for each story to catch regression bugs without wasting time on irrelevant scenarios.

The first big shift-left type effort was increased collaboration in test case planning during the desk checks. This checkpoint provided an opportunity for the tester, analyst, and developer to brainstorm creative scenarios based on the story's requirements, code changes, and related process flows. After identifying what was most important to test, the developer and tester discussed whether all or part of those cases could be covered by the unit and integration tests; or, based on the code changes and existing automation coverage, if test cases could be eliminated. This process gave testers more confidence in their planning so that they could execute or automate the tests quickly and confidently without second-guessing if the coverage was sufficient.

Over time, the team caught up enough on feature deliverables to begin to shift-left in test automation. Many early unit tests had problematic design spread through copy-paste and rushed implementation. Similarly, integration tests contained repetitive hard-coded setup data that was substandard for testing code that was heavily data and state-driven. Improvements included:

1) supplying missing coverage
2) brittle tests that made refactoring difficult
3) unclear test failures that resulted in bugs when the test was deleted and re-written instead of tediously debugged
4) unreliable tests in CI that slowed down deployments

Developers and testers also regularly evaluated if the confidence gained by writing exhaustive test coverage was worth the time investment in cases where the underlying code was expected to be temporary and removed eventually. Because of the large team, frequent design improvements, and evolving test strategies, the team became challenged to keep a quality approach in sync. As individuals discovered and made improvements, they shared findings in its monthly team technical discussions so that everyone could uphold best practices in writing, maintaining, and reviewing its test suite.

2.3 Team 3

My third team started with five established developers and one novice tester. I joined them for the first four months to mentor their dedicated tester and establish shift-left testing practices so they could deliver quality quickly. We created a new microservice that compared input data from two sources and produced adjustment records to be used in a transaction reconciliation process. I started by collaborating with the developers on every story to plan or review automation. Using my time and experiences from working with the two previous teams, I established a strong foundation and corrected problematic practices early. I also emphasized whole team involvement to evaluate and improve quality practices as the codebase grew larger and more complex. By the end of my involvement, the developers worked independently on low-level automation and reviewed each other's tests, and only pulled me in when they had new questions or ideas to discuss. This allowed the other tester to continue learning the business domain and generate robust test data to expand coverage in the integration tests and exploratory manual testing.

With clean code and tests to start, the team focused our shift-left efforts on quickly catching and paying down technical debt before it could accumulate. When writing or modifying a unit test was especially complicated, we questioned if the underlying code had strayed away from SOLID[3] design principles and prioritized refactoring appropriately. If explicit data values in the test setup were not necessary for its flow, those fields were removed or assigned random values using a helper library AutoFixture[4]. For tests
asserting against List results, we leveraged a test framework FluentAssertions\(^5\) that would produce more descriptive failure messages.

After several months, the team realized its biggest risk came from a part of our calculation code after I discovered a bug in amount rounding. The code was implemented to mimic calculation rules owned by another team because the results of those calculations in an earlier part of the process were not stored for us to simply query. When inspecting the unit tests for my team, I decided to reference unit tests from the other team to verify results in my testing and found that some of our results for a given input were different by one cent. While fixing the calculation was very simple, this bug pointed to a risk that the two calculations could easily be out of sync. After discussions with the other team, we learned a third team was also impacted because they maintained yet another copy of the rules. While the effort to centralize this code would be substantial, the risk from this technical debt discovery earned the refactoring a prioritized spot on the teams' backlogs.

2.4 Team 4

This last team was an established one of four developers that had been without a dedicated tester for a year when I joined. They formed as a DevOps subset of Team 1 and owned both the CI/CD infrastructure and a service dependency for multiple teams in the domain. This service transferred financial records from a monolithic database into our domain for additional downstream processing. It handled heavy loads during peak processing but had timing issues requiring manual intervention that we wanted to automate. While its test coverage was adequate from initial implementation, infrequent code changes resulted in those tests being overlooked as quality practices evolved over the following years. Since I had participated in development and testing during this service's creation, I recognized many patterns that I would handle differently with my increased experience. Our quality challenge was finding ways to improve these tests while juggling necessary infrastructure changes and responding to urgent production issues.

Our first shift-left change was adding me to pull requests to review both code and test changes. At first, I familiarized myself with and evaluated test updates on my own without communicating them outward. I quickly realized I could not maintain this with new stories perpetuating the patterns I was removing. Next, I facilitated a team discussion for the developers to call out areas they had identified for inconsistent test behavior, lack of coverage, or difficult-to-maintain tests. This exercise centered quality as a full-team effort, not my agenda and sole responsibility. We met again to review easy ways to improve unit tests so the developers could use those strategies from the beginning rather than me keeping up with corrections. As my rapport on the team grew, the developers started bringing test questions proactively rather than waiting for me to add code review comments. They brought their own ideas for how to make tests better.

3 Collaborative shift-left strategies

Each of these teams had different engineering strengths and varied technical challenges. But they adopted similar strategies to shift-left in their efforts to find bugs earlier in the software development lifecycle. A dominant theme was to add checkpoints to ensure the analyst, developer, and tester were in sync and to quickly resolve any issues that surfaced since the last one. The following subsections describe the various strategies implemented and the benefits gained from each.

3.1 Story kick-offs

Teams conducted a pre-coding "kick-off" in which a business analyst would review a story's requirements with a developer and a tester. The developer identified ambiguous requirements for the analyst to clarify in real-time, which reduced the likelihood of incorrect implementation. The tester identified preliminary edge-case scenarios for the developer to account for while coding, which avoided the increased cost of finding and fixing related bugs at a later stage in the SDLC.
3.2 Testers in code reviews

Developers ensured that a tester had reviewed and approved the code review before moving a story to coding-finished status. Testers primarily reviewed unit and integration tests for consistency and adherence to test patterns that would help reduce future maintenance. Testers with less experience reading code and unit tests focused on logical test names and actionable log messages. Those with more code familiarity ensured that the conditions and assertions of the tests were appropriate for the method under test. Because tester bandwidth was limited and not every code review would receive the same level of attention, the most important outcome was that developers learn why improvements were necessary so that they could self-correct in future work.

3.3 Developer + Tester automation discussions

When automation decisions required a judgment call, such as how to handle a dependency or which test data to select, developers brought in a tester to review their options. They evaluated the pros and cons of mocking a dependency in different ways, and whether unit or integration tests were more appropriate for certain cases. They discussed expanding test coverage through more assertions, new test paths, or better test data. This collaborative approach both resolved the developer's dilemma and provided the tester with a familiarity of low-level tests that allowed them to refine exploratory testing and high-level automation scope.

3.4 Story desk-checks

Developers conducted a post-coding "desk-check" in which they would demo their working code changes to the business analyst and tester to ensure they satisfied the story requirements. They addressed edge cases that were called out during the kick-off and executed additional test scenarios that the tester identified since then. If those scenarios revealed bugs, they would be fixed before the story moved to test. Developers also used this time to alert the tester to implementation details that required a specific setup to verify in testing (e.g., processing large quantities of data in batches) so that the tester could verify and review with the analyst before completing the story.

4 Unit test improvements

My early mentors emphasized writing descriptive tests at all levels rather than documenting features in plain English. Passing tests had to reflect the current state of the code they covered, while written documentation could quickly become stale if not updated alongside every code change. Tests, therefore, should be written in a way that their failure raises questions about code changes. "Did you mean to change this behavior and forget to update the documentation (tests), or did the changes unexpectedly affect other flow?" Tests must be easily readable to aid in debugging, or they risk being ignored, deleted, or rewritten in a way that changes their intended purpose. These sections are unit test improvements I found particularly effective in speeding up my testing process as well as reducing developers' maintenance of them over time.

4.1 Name tests descriptively

Teams found it useful to name tests according to their behavior. They followed the GivenWhenThen structure to choose behavior-driven test names with information that shortened time to identify the execution path of a failed test.

Using the method UpdateRecord() as an example, consider the following test names. Initially, they seem intuitive, but they do not tell the difference between triggering an Error versus a Failure, and how those results differ. They could be mistakenly evaluated as duplicate tests and one of them incorrectly deleted.
- UpdateRecordSuccess
- UpdateRecordError
- UpdateRecordIgnore
- UpdateRecordFailure

Consider these revised names instead.

- UpdateRecord_WhenRecordFound_AndReadyToUpdate_ReturnsSuccess
- UpdateRecord_WhenRecordNotNotFound_ReturnsError
- UpdateRecord_WhenRecordFound_AndAlreadyUpdated_ReturnsSuccess
- UpdateRecord_WhenRecordFound_AndUpdateFails_ReturnsWarning

Now it is clear how Error and Failure are different in code path and result produced. If a test fails after refactoring, the developer immediately knows which behavior (rather than methods) was impacted and can evaluate fixes with a better understanding of the use cases that are affected.

Additionally, test names better document the behavior of the underlying method so that a new developer or tester could read through them to gain a high-level understanding rather than tracing through the method line-by-line.

4.2 Keep setup clean

Redundant or unnecessary lines of setup in the test method made it difficult to understand its purpose. While descriptive naming helped, the test body needed to be equally comprehensible, and only contain the necessities. Common issues I found were:

1) assigning unnecessary record properties or configuring dependency fakes that were not referenced in the method under test

2) assigning hard-coded values when the exact value was unnecessary for the test purpose and simply needed to be not null.

These issues typically resulted from copying code from another test fixture without removing irrelevant lines or considering if the assigned values were integral to the test flow. Over time, these small issues added up to result in tests that were very difficult to understand without running them in debug line-by-line.

A pattern that worked for my teams in our nUnit tests (since xUnit behaves differently) was to configure the system-under-test to default success conditions in the Setup methods and then use the AAA pattern (Arrange, Act, Assert) to outline each test method. Happy path tests then required minimal Arrange sections, while error paths configured only the dependency changes that triggered their specific flow. If this did not significantly reduce the Arrange section for error paths, it was a sign that the method might have too much responsibility and should be evaluated for refactoring.

Additionally, teams found value in using a library to provide random test data when the values did not drive the flow or outcome of the test. Typically, fields needed to be not-null and not-empty, and their values would be asserted in the result, but the exact values did not matter. We used AutoFixture for our .NET application, and other libraries such as Bogus or Faker exist for different languages.

Using both these strategies, to test Record.Validate(), the happy path status of "Valid" and a random value for amount are assigned in the SetUp method. Specific alternate test paths overrode status in the Arrange section to Failed or Processing before calling Record.Validate() in the Act section. The Assert section verified the result and checked that the amount field was passed through the method calls and populated in the result details, but not assert on the exact value.
4.3 Remove branching logic

Developers introduced branching logic in unit tests to maximize reuse of test code by combining similar scenarios that deviated by a single arrange or assert statement. These tests typically used a boolean parameter to trigger extra statements, which allowed two or three tests to be “simplified” down to one. This strategy reduced lines of test code in exchange for nondeterminism and future maintenance headaches, which was not desirable.

```csharp
[TestCase(true)]
[TestCase(false)]

public void TestErrors(bool isFatal) {
  //arrange
  if (isFatal) {
    //setup fatal case
  }
  ...
  ...
  //act
  ...
  ...
  //assert
  if (isFatal) {
    //logger was called
  }
  ...
}
```

The code snippet above shows an example where the same code is used for both a true and false test case that share identical arrange, act, assert, but then additional lines are conditional to the false case. This immediately makes the test more difficult to understand, and only worsens over time

We addressed this problem less with creativity and more with vigilance and education around the need to avoid this anti-pattern in our test suite. We used online code reviews to call out branching logic within the test and have it removed. If the developer was not yet familiar with why, a tester or another developer explained how it makes tests more difficult to read and understand and suggested appropriate ways to split the branched logic into another test.

4.4 Write better assertions

FluentAssertions[5] made a huge difference in our tests for asserting on IEnumerable objects. In the example below, I show lines 7-10 simplified to the more descriptive lines 16-17 by using the FluentAssertions library. The line 13 assertion shows a valid conversion to FluentAssertion, but one that does not take advantage of better failure messaging because it would simply produce “Expected 1 but found ___." The FluentAssertions at the end also communicate more meaning with their structure. In this example, the importance is not that any single list element has Item1 with value 1, but that all list elements have Item1 with value 1.

```csharp
public void TestListResult() {
  //arrange
  ...
  //act
  ...
  //assert
  Assert.AreEqual(3, list.Count);
  Assert.AreEqual(1, list[0].Item1);
  Assert.AreEqual(1, list[1].Item1);
  Assert.AreEqual(1, list[2].Item1);
}  ```
The failure message for FluentAssertions will also aid in debugging. Instead of “Expected 1 but was ___” which then requires debugging to determine if one assertion or all were affected, the error would read like “Expected all elements to have Item1 but {object details} do(es) not match.” This message would contain all objects that failed the condition, immediately providing more detail on the scope of the failure to aid in identifying cause and solution.

5 Using unit tests as a tester

Easy-to-read unit tests allow for better leverage of them to logically reduce exploratory or automated testing efforts. Heavy user interaction with our front end required that we consider intuitive design and usability in addition to correct logic. This section outlines the most frequent or memorable ways that unit tests influenced testing work on a story.

5.1 Simplifying webpage testing

Many elements of webpages are enabled or disabled based on the state of records shown or selected. For example, enabling an Action button required the record to be correct status(es), and options in a dropdown list depended on the record type. This state-driven behavior was dictated by methods in the front-end code. Once we verified that all combinations for a page state result were covered in unit tests, we could apply boundary and equivalence partitioning techniques during exploratory testing to look for edge cases and usability of the front end.

5.2 State transition rules

Our backend code also had many rules governing state transition. The most common were status transitions and verifying actions could be applied, and well-written methods and tests provided peace of mind when these areas changed. Like with page state testing, unit test coverage for all combinations of input/output allowed testers to focus on looking for edge cases not accounted for by the requirements.

5.3 Known risk - batching

My teams worked on code that processed thousands of records and depended on events being raised and consumed correctly for all the records to progress as expected. To avoid excessive memory consumption, the code was designed to run queries, process records, or raise events in batches of 100 or 1000 at a time. All batches needed to run without overlap or gaps between the contents, and my teams encountered several production issues because we failed to account for new records entering the system during processing that disrupted the batch order. Batching became a known risk to make sure that developers and testers agreed on how it was both implemented and tested.

While ideally, the developer would call out this risk in one of our checkpoint meetings, unit tests gave testers a clue that this risk existed in the story so that we could initiate a testing discussion with the developer. In the FakeItEasy testing library, the method ReturnsNextFromSequence() is used to set up unique returns for multiple calls to the same dependency, and my teams used to test that a batched process occurred multiple times until it reached its typical end condition that the number of records found...
was less than the batch size. Finding references to this setup provided a clue that batching was used and that the developer had considered it enough to test it as a unit, but testers would still learn more about it to determine if additional manual tests were needed.

5.4 Known risk - idempotency

Process flows triggered by an event had to be idempotent, meaning that multiple attempts to process the event should only result in one complete pass through the flow and ignore (exit out without more changes) any subsequent repetitions. Idempotent design was a pattern that testers needed to ensure but usually did not need to test edge cases except in special circumstances. Verifying the existence of unit tests for idempotent scenarios was a fast way to ensure it had been designed. I looked for tests that arranged data as if the action had already happened and then called the method again. For example, Test 3 in section 4.1 is an idempotency test I would want to find, or else ask the developer to add them as appropriate.

6 Conclusion

Shift-left means paying attention to quality earlier, finding bugs earlier, and fixing them more easily. However, simply doing quality and test steps as early as possible results in shifting fractured, siloed quality responsibility to other roles without reducing the sum effort required by the team. Taking an agile approach to shift-left, valuing "Individuals and interactions over processes and tools," results in a collaborative approach to shift-left. It allows teams to spend smaller amounts of time earlier in the software development life cycle in order to reduce both the later testing effort as well as the total quality effort across the team. Unit tests provide a collaboration point on the intersection of developer responsibility and tester specialization.

Shift-left was not accomplished by doing the exact same thing on every team. Technologies, individual skills, and preferences created different environments requiring unique approaches. On each team, I paired experience from previous teams with my growing knowledge of shift-left test principles to draw attention to the unit test foundation of our test pyramid. I strived to promote quality as a whole team effort, so process changes were discussed and decided upon by everyone. The specific changes varied by team, but on each one we streamlined our test efforts by pursuing a cohesive test pyramid, leveraging unit and integration tests to reduce the effort of manual testing and keep our functional UI tests to a minimum. The common realization on each of these teams was that, as we worked to shift-left, both testers and developers benefitted from collaborating to improve unit test strategy. The test pyramids for each team were worked on by multiple people in two different stages, and so strategies and improvements required two-way conversations. Testers helped drive that conversation, but the goal was to figure it out as a group. Unit tests provided an underrated bridge for sparking those conversations.
References


