Build in Quality at Scale with Test-Driven Development

"Built-in Quality" is a core value of SAFe®. It allows quicker delivery of business value. One of the key practices in realizing built-in quality is Test-Driven Development (TDD). This course teaches developers how to write behavior-focused unit tests and to develop incrementally using these tests. It emphasizes how these tests reflect the system requirements. Refactoring legacy code and adding unit tests to that legacy code is also explored. The course emphasizes hands-on practice in writing unit tests, creating mocks, and refactoring.

TARGET AUDIENCE:

Software developers in Java, C#, or C++ who want to learn to create quality code.

LEARNING OBJECTIVES:

- How to be "test-driven"
- How to write behavior focused unit tests
- When and how to use mock objects (test doubles)
- How to refactor legacy code

INSTRUCTOR

Ken Pugh, author of *Lean-Agile Acceptance Test-Driven Development: Better Software Through Collaboration*.

SAFe's DESCRIPTION OF TDD

Built-in Quality is one of the four Core Values of SAFe. The Enterprise's ability to deliver new functionality with the fastest sustainable lead time and to be able to react to rapidly changing business environments is dependent on Solution quality. But built-in quality is not unique to SAFe. Rather, it is a core principle of the Lean-Agile Mindset, where it helps avoid the cost of delays associated with recall, rework, and defect fixing.

Test-First is a philosophy that encourages teams to reason deeply about system behavior prior to implementing the code. This increases the productivity of coding and improves its fitness for use. It also creates a more comprehensive test strategy, whereby the understanding of system requirements is converted into a series of tests, typically prior to developing the code itself.

In Test-Driven Development (TDD), developers write an automated unit test first, run the test to observe the failure, and then write the minimum code necessary to pass the test. Primarily applicable at the code method or unit (technical) level, this ensures that a test exists and tends to prevent gold plating and feature creep of writing code that is more complex than necessary to meet the stated test case. This also helps requirements persist as automated tests and remain up to date.

The focus is on writing the unit test before writing the code, as described below:

1. Write the test first. Writing the test first ensures that the developer understands the required behavior of the new code.

2. Run the test, and watch it fail. Because there is yet no code to be tested, this may seem silly initially, but this accomplishes two useful objectives: It tests the test itself and any test harnesses that hold the test in place, and it illustrates how the system will fail if the code is incorrect.

3. Write the minimum amount of code that is necessary to pass the test. If the test fails, rework the code or the test as necessary until a module is created that routinely passes the test.

In XP, this practice was primarily designed to operate in the context of unit tests, which are developer-written tests (also code) that test the classes and methods that are used. These are a form of "white-box testing" because they test the internals of the system and the various code paths that may be executed. Pair work is used extensively as well; when two sets of eyes have seen the code and the tests, it's probable that the module is of high quality. Even when not pairing, the test is "the first other set of eyes" that see the code. Developers note that they often refactor the code in order to pass the test as simply and elegantly as possible. This is quality at the source—one of the main reasons that SAFe relies on TDD.

SAFe References to Built-in Quality and Test-First
http://www.scaledagileframework.com/built-in-quality/
http://www.scaledagileframework.com/test-first/

OUTLINE:

- Introduction
  - How TDD helps with quality
  - Creating tests first
  - Running the framework
- Unit test introduction
  - Naming tests
  - Trying out the framework
- TDD Exercise
  - Organizing unit tests
  - An TDD exercise
- Mocking (Test doubles)
  - When to use
  - How to create
  - A mocking exercise
  - Ways to minimize mocking
- Refactoring
  - Common refactorings
  - A legacy code refactoring exercise
- Your TDD
  - Large exercise in TDD

ATTENDEE MATERIALS

Workshop materials are provided at the start of the class

ROOM SETUP AND EQUIPMENT

One computer for every two students, loaded with necessary software.
Flip chart and whiteboard for the instructor.
A projector with screen.

PREREQUISITES

Experience in C#, Java, or C++

COURSE LENGTH

3 days

MAXIMUM

24 students

NOTES

This course does not provide any certification related to Scaled Agile, Inc. or SAFe.

PROVIDER

Ken Pugh, Inc. 732 Ninth Street #695, Durham, NC 27705.

CONTACT

Leslie Killeen 919-490-6335

leslie@kenpugh.com