DevOps Fundamentals

Continuous Testing

Agenda

Unit Testing

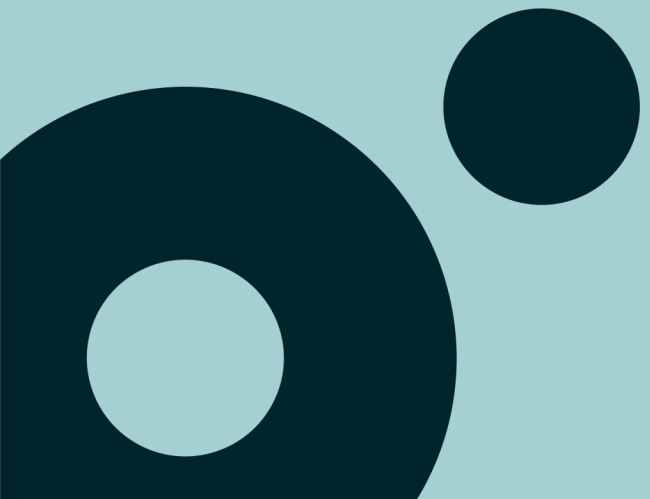
TDD & BDD

Test Automation

End-To-End Testing

moOngy.

Unit Testing



What is Unit Testing?

A unit test is a test of a functional **unit-of-work** in the system

A good unit test must be **fast**, **in-memory**, **consistent**, **automated**, **independent** and **repeatable.**

A **unit of work** is any functional scenario in the system that contains logic.

It can be as short as a function, or it can span multiple classes and functions, and it provides internal or business value to the system under test

Test cases must be done by developer team and most of the cases by the same developer that created the main code

What is NOT Unit Testing?

Does not require human intervention to run (fully automated)

Does not utilize external dependencies, such as a database or file system

Not a substitute for Testing/Quality Assurance

Not a guarantee that the code is bug free, only that it passes the criteria given

Why should I do this?

Ensuring quality finding bugs early (if you write tests early ☺)

Unit test proves that the code does what you write not what you wanted

Testing quickly and frequently makes refactoring easier

Unit tests provide documentation, like a spec in code that stays in-sync with your code

Helps on regression testing

Less time performing functional tests

Less coupled code

Why unit testing is hard?

Tight coupling with other units

- Interfaces, composition over inheritance, polymorphism over conditionals

Dependency on complex state

- Stubs, spies, state verification

Conflation of responsibilities

- Separation of responsibilities

Object creation

- Dependency injection. Factories

Closed objects

- Self-calls to non-private virtual methods & properties

Global state, singletons, static methods

moOngy.

Why testing is hard for Developers?

The goals of testing run counter to the goals of other development activities

Testing can never prove the absence of errors– only their presence

Testing by itself does not improve software quality. To improve the quality of the software the root causes of problems must be found and fixed

It is difficult to critique or review one's own work

Testing requires you to assume you will find errors in the code

Characteristics of a good unit test

Fast. It is not uncommon for mature projects to have thousands of unit tests. Unit tests should take very little time to run. Milliseconds.

Readable. Clear pre and post conditions (Arrange, Act, Assert) and execute on a single unit of work

Isolated. Unit tests are standalone and have no dependencies on any outside factors such as a file system or database.

Repeatable. Running a unit test should be consistent returning always the same result

Self-Checking. The test should be able to automatically detect if it passed or failed without any human interaction.

Timely. A unit test should not take a disproportionately long time to write compared to the code being tested

Unit Test Maintenance

Unit test code becomes part of the code base and needs to be maintained

Can become fragile

Must be always updated along side the production code

Code Coverage

High code coverage percentage is often associated with a higher quality of code

However, the measurement itself cannot determine the quality of code

Setting an overly ambitious code coverage percentage goal can be counterproductive

- A complex project with thousands of conditional branches having 90% code coverage
- The amount of time it takes to account all edge cases to reach 95% could be a massive undertaking without much benefits

High code coverage percentage is not an indicator of success, nor does it imply high code quality. It just represents the amount of code that is covered by unit tests

Frameworks

xUnit

Nunit

MSTest

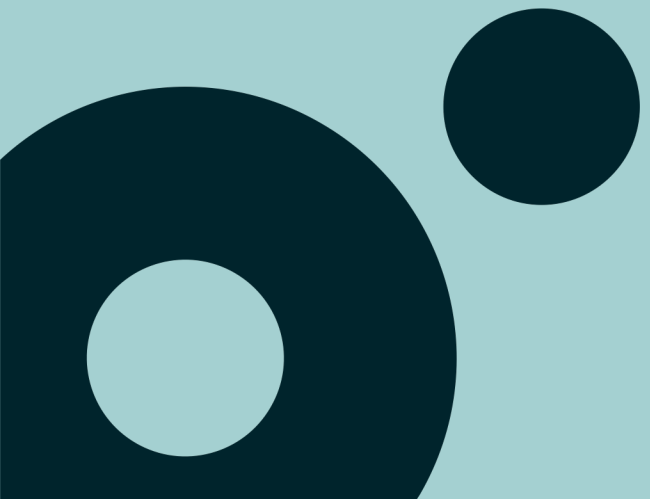
Junit

Mocha

.....

moOngy.

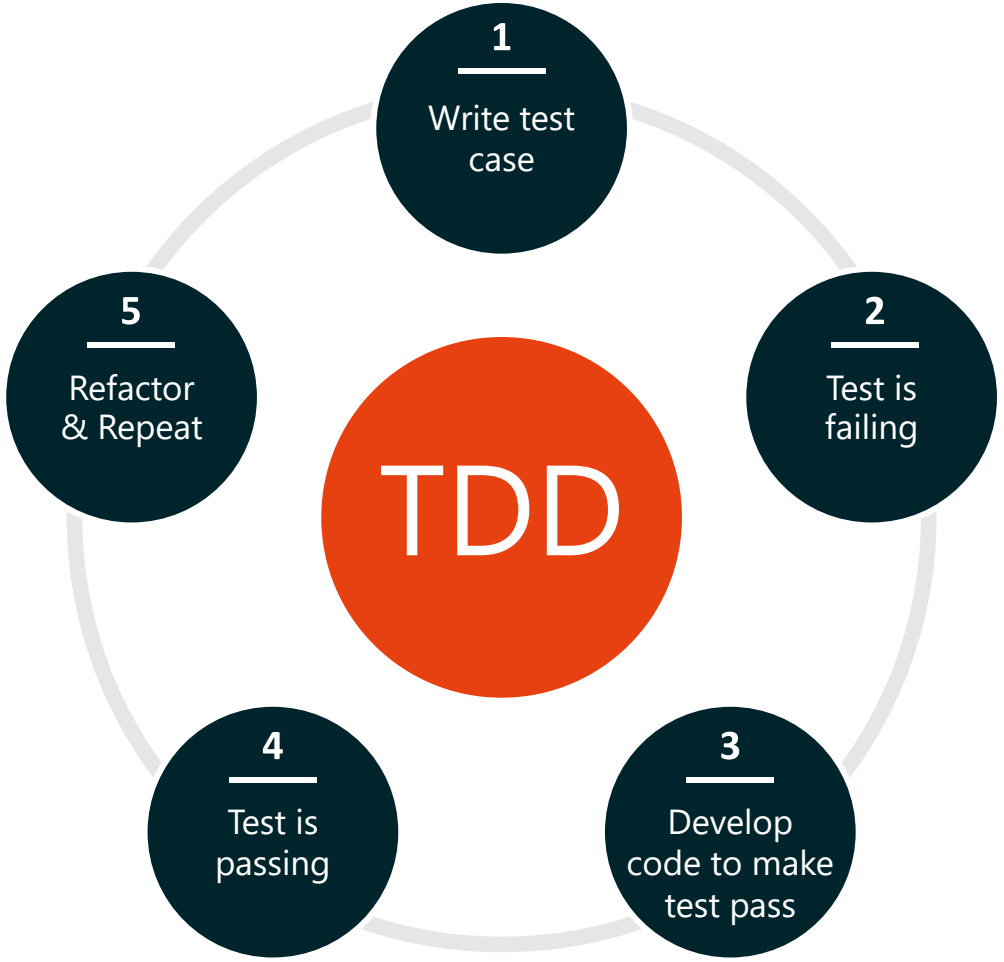
TDD & BDD



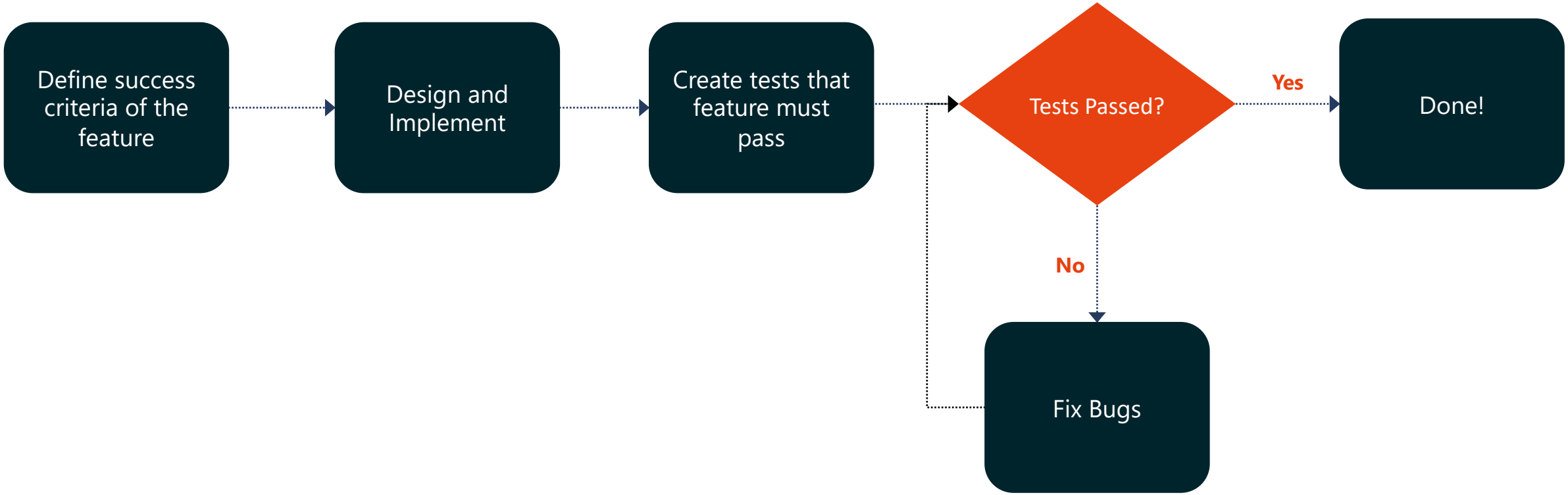
Test Driven Development (TDD)

Test-driven development (TDD) is a development practice that includes the creation of automated unit tests prior to implementation of the code.

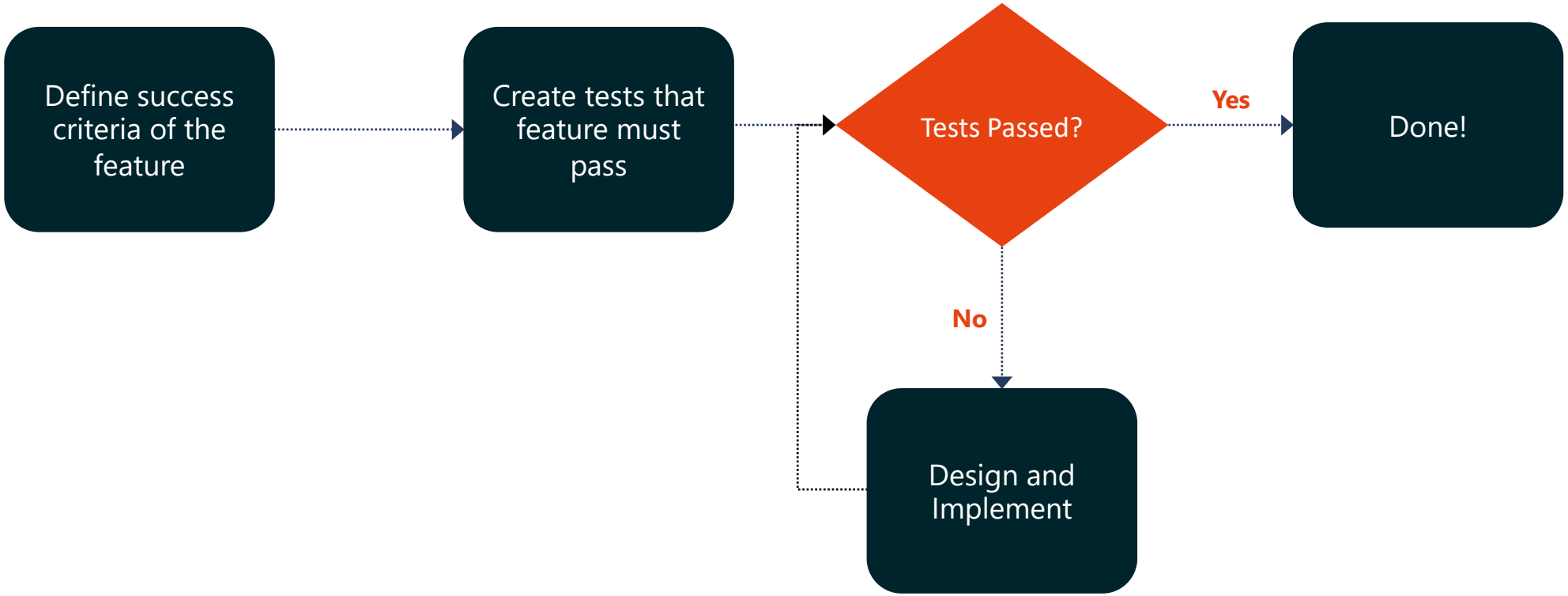
The TDD practice was created to address enhanced simples designs and build confidence on the code, not to create a suite of tests.



“Common” Development Flow



TDD Development Flow



Behaviour Driven Development (TDD)

Behavior-driven development (BDD) is an agile software development process

Encourages collaboration among developers, QA and non-technical or business participants

Behavior-driven development is an extension of test-driven development (TDD) that makes use of a simple, domain-specific scripting language (DSL).

DSLs convert structured natural language statements into executable tests

The result is a closer relationship to acceptance criteria for a given function and the tests used to validate that functionality

As such it is a natural extension of TDD testing in general

Behaviour Driven Development (TDD)

Requires using a framework (like SpecFlow, Cucumber, etc.) that translates the DSL statements into executable instructions

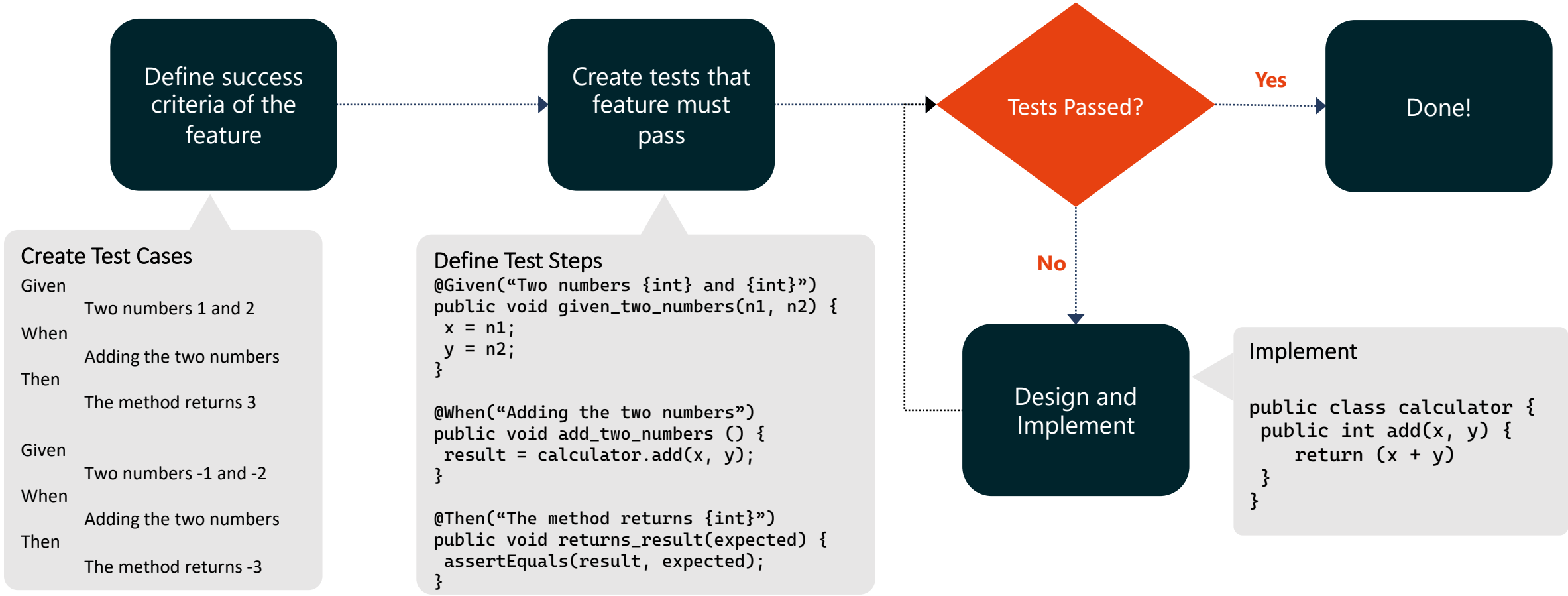
These DSL languages flow a Given-When-Then approach and transform it to Gherkin format that is standard for every BDD Testing Framework

Then, a final step is to transform Gherkin to a technical representation (meaning in code language you chose) to allow developer to create the code to implement the tests

The job of the engineer is no longer to code the tests but to code the DSL translation actions into executable code

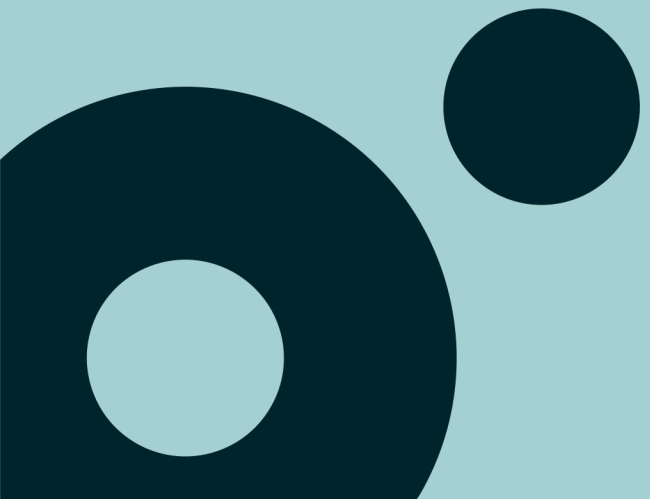
This allows that not technical people can create the tests

BDD Development Flow



moOngy.

Test Automation



Shift-Left Testing

Shifting left is a reference to moving testing to the left on a timeline

Approach used to speed software testing and facilitate development by moving the testing process to an earlier point in the development cycle

Shift left testing is designed to be an improved model for shift left (fast lane) development because traditional testing models that wait until later in the development cycle can bottleneck development

With this approach you get quality on code sooner on the development cycle

Practices: TDD & BDD, Automated Unit Test, CI/CD Integration, Telemetry

Shift-Right Testing

There's only one environment really similar with Production… and id Production!

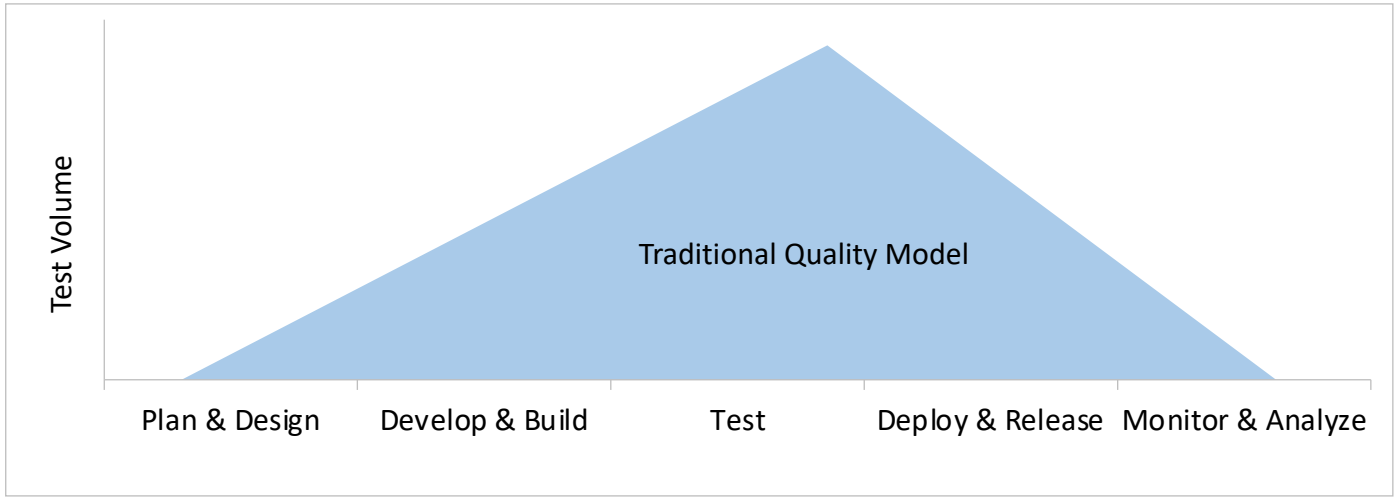
A common DevOps practice is to not only perform testing early in the development cycle and more often application behavior is also tested in production ("shift right" testing).

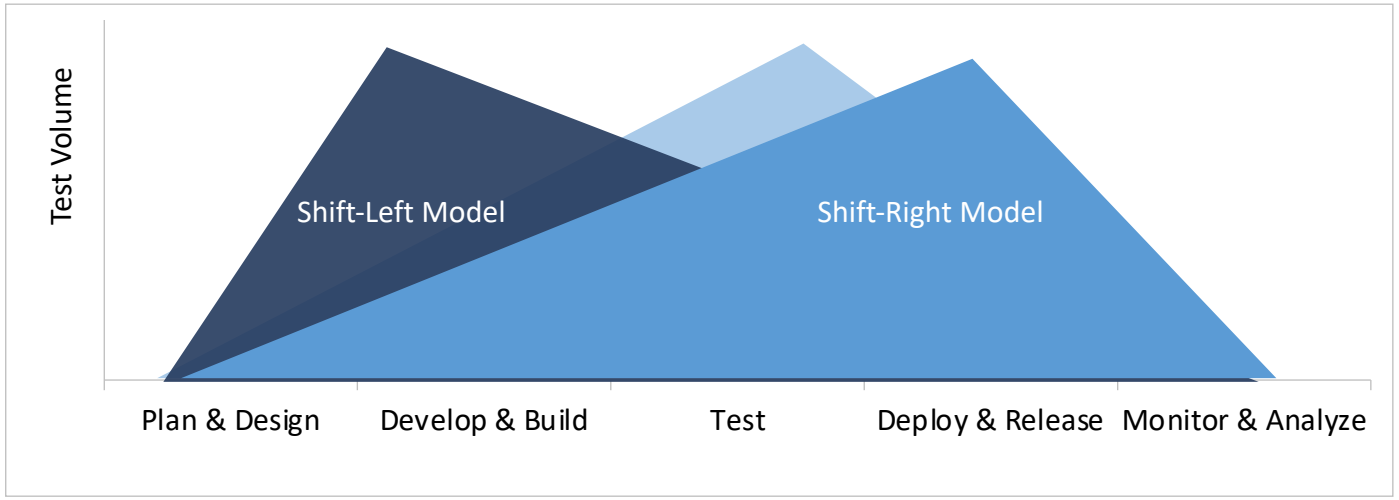
Application stability and resiliency can be improved by performing this "shift right" testing, combined with using new capabilities to monitor applications when under stress and/or unexpected (new) elements are introduced.

Shift-right testing increases customer feedback, drives Hypothesis-Driven Testing and helps to achieve High Test Coverage.

Practices: Release Rings, Feature Flags, Hypothesis-Driven Testing, Fault Injection, Insights gained from Telemetry

Shift-Left & Shift-Right Testing





Test Automation

Test automation is a core activity in DevOps

Key aspect in driving the goal of faster delivery of value to customers with higher quality

Automation should consider both Functional and Non-functional testing covering as many layers as possible of the application/system to be tested

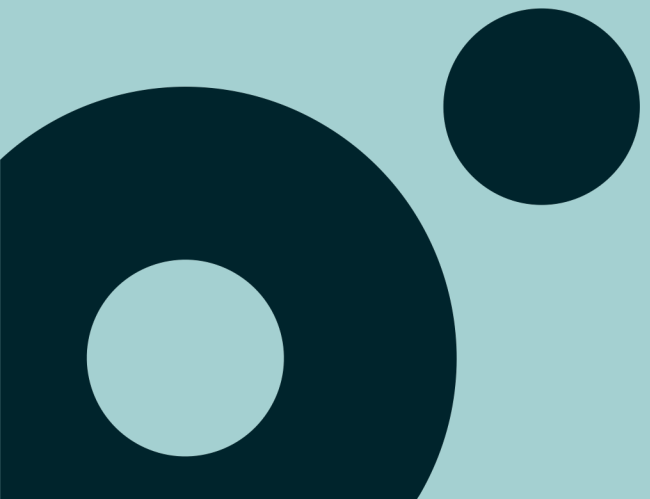
Different tests provide validation of different aspects of the system and contribute to provide a complete quality assessment of what is being delivered

Examples: Unit, Load, Integration, Regression, Functional/UI, Smoke, …

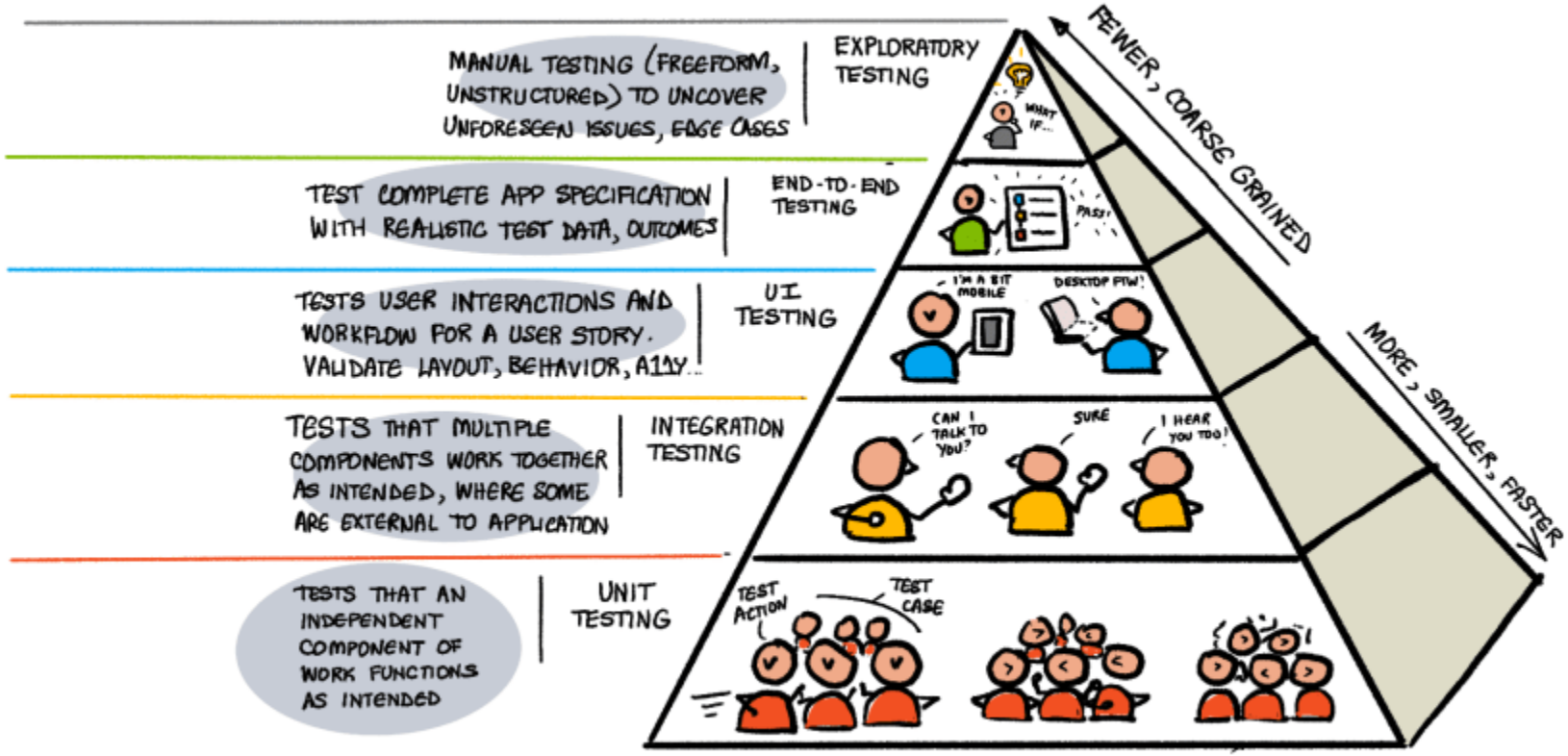
Automated vs. Manual Testing

Automated Testing	Test procedures that can be run in a consistent and repeated manner Increased ROI, test can be run at anytime Add new tests sustainably in a growing regression suite Self-documented test steps
Manual Testing	Humans are inherently bad at running the same set of steps reliably repeatedly Humans are less efficient, more error prone and may produce inconsistent outcome Humans can provide much more value in less procedural tasks like exploratory testing and bring innovation to quality assurance phase

End2End Testing



Test Pyramid



moOngy.

Benefits from E2E Testing Automation

Reduce risk, providing more comprehensive testing coverage

Enable faster and more reliable execution

Facilitate greater test coverage, supporting the execution of test scripts across all popular browsers and operating systems

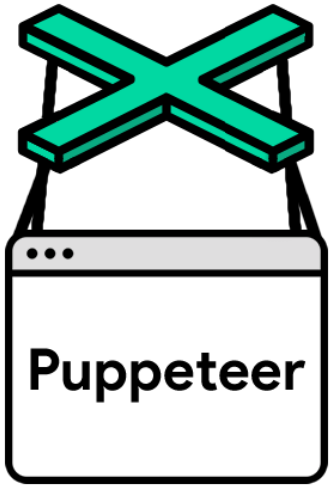
Enable regression testing of ever-changing applications and environments

Deliver higher test accuracy and find more defects earlier

Provide formalized and documented processes

Facilitate the re-use of tests

UI/E2E Testing Frameworks



Load Testing Frameworks

Demo: Test Automation

Lab

Lab 06 – Continuous Testing

Learning Objectives

- Add a new step to your pipeline to perform E2E testing
- Learn how to share data between jobs
- Brief introduction to Playwright

Markdown version: https://github.com/tasb/devops-with-github-training/blob/main/labs/lab06.md

HTML version: https://tasb.github.io/devops-with-github-training/labs/lab06.html

moOngy.
Minds on the move

Rua Sousa Martins, nº 10

1050-218 Lisboa | Portugal