Test Driven Development

.NET CORE

Test-driven development (TDD) *is a software development process that relies on the repetition of a very short development cycle.*
Requirements are turned into very specific test cases. Then the code is improved so that the tests pass. The cycle repeats.

HTTPS://EN.WIKIPEDIA.ORG/WIKI/TEST-DRIVEN_DEVELOPMENT

WHY WE FIGHT
TEST
A SERIES OF SEVEN
INFORMATION FILMS

Testing – Traditional Model

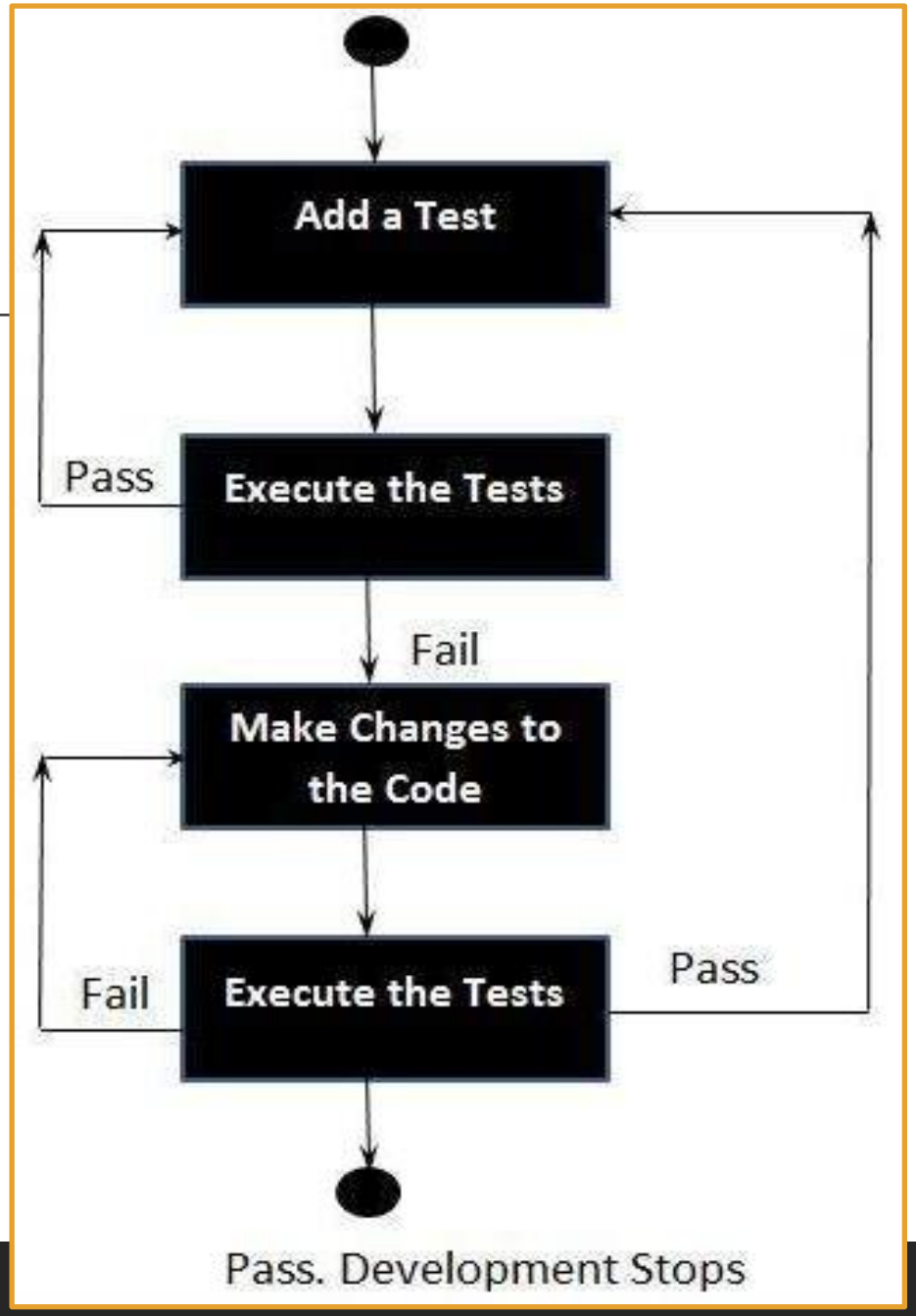
https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/debug-test/test-driven-development?view=toolsforcordova-2017#test-driven-development
https://www.tutorialspoint.com/software_testing_dictionary/code_driven_testing.htm

Developers who define their role as “writing code” usually jump right into writing methods that handle different kinds of data that might get thrown at them.

Then they write tests. After writing a few tests, they’ll see that some of those tests still fail because of certain code cases that their methods can’t handle. They improve those methods to handle those cases, and write a few more tests, which then reveal additional issues in the unit code.

This puts them into a pattern of bouncing back and forth between thinking about coding and thinking about data for test cases.

This results in missed test cases and faulty code.



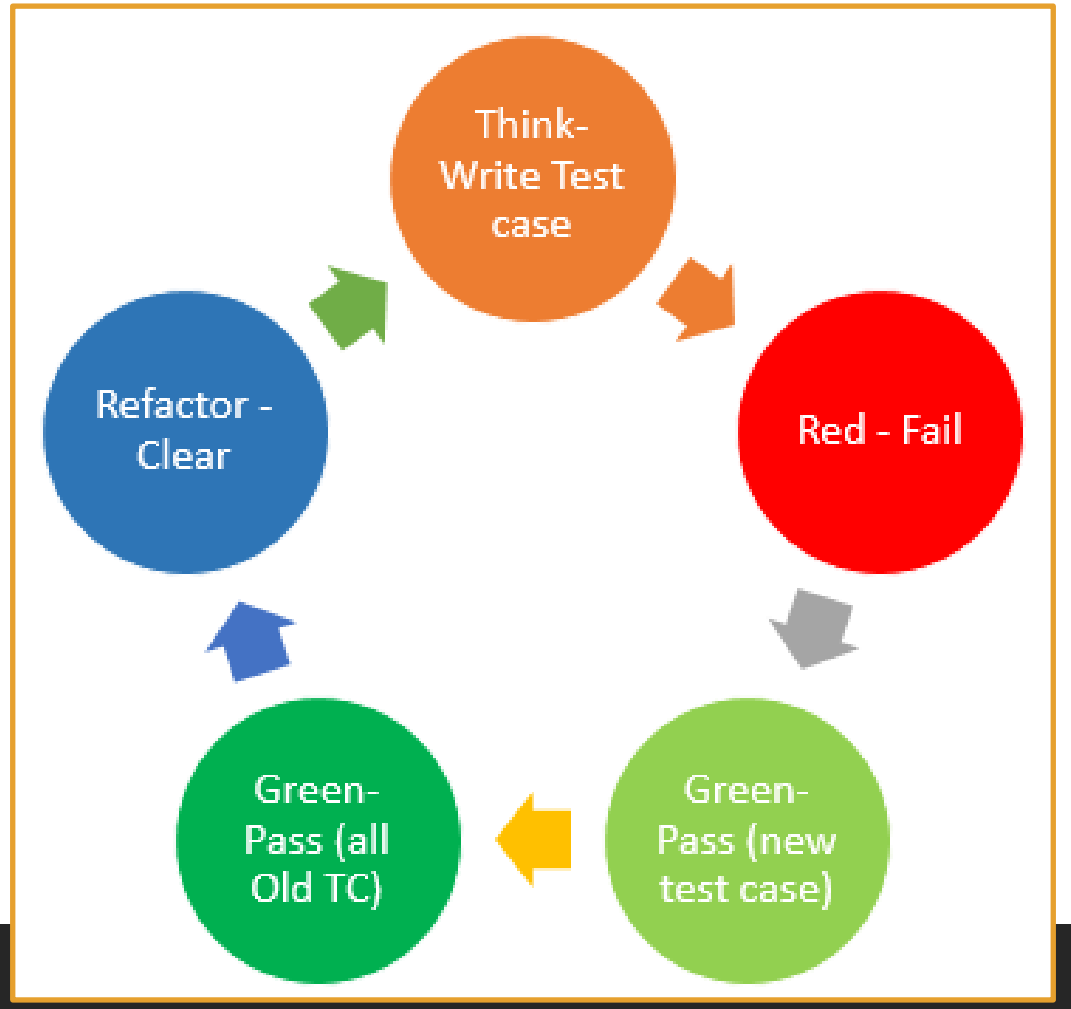
TDD – Test Driven Development

https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/debug-test/test-driven-development?view=toolsforcordova-2017#test-driven-development

Testing is just as important as coding, if not more so.

Thinking through variations of good and bad data is a different mental process than thinking about how to handle those variations in code.

- TTD asks, “How do I challenge the unit under test to fail?”
- “Coding” asks “How do I write this method to work properly?”

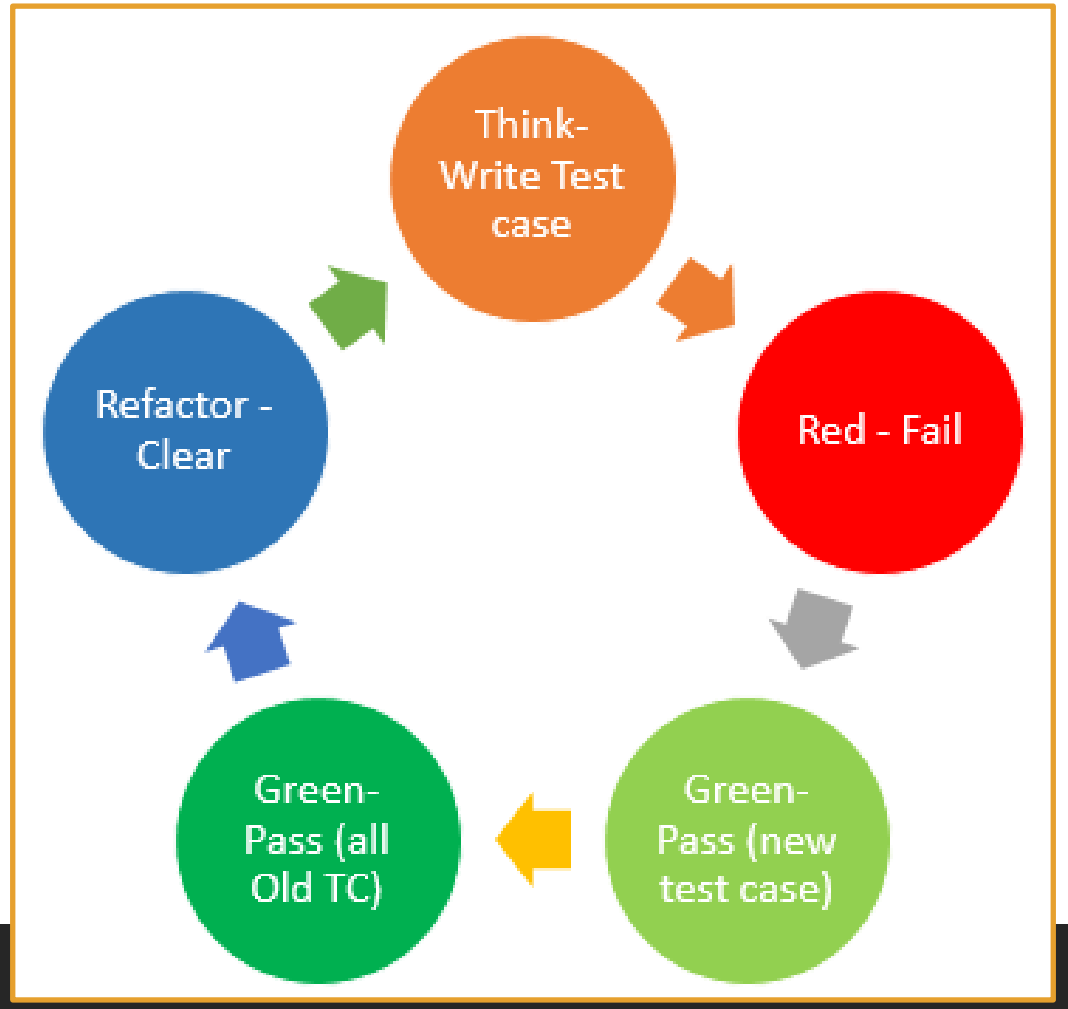


TDD – Test Driven Development

https://www.whizlabs.com/blog/what-is-tdd-and-its-phases/

1. Create a test case - Write a test case before writing any code. This ensures you write the test to a methods expected functionality and the test case is not biased to show code merely works.
2. Red Failure of test case - There's no code. You get a compile error.
3. Green – Write code so the new test case to passes. Write only the minimum required to pass the test case. The "just enough" concept helps ensure that no extra bit of code goes in.
4. Green - Ensure all old test cases still pass.
5. Refactor the code to clean it - Ensure that all functionality is intact and the code is refined.

Repeat the cycle.



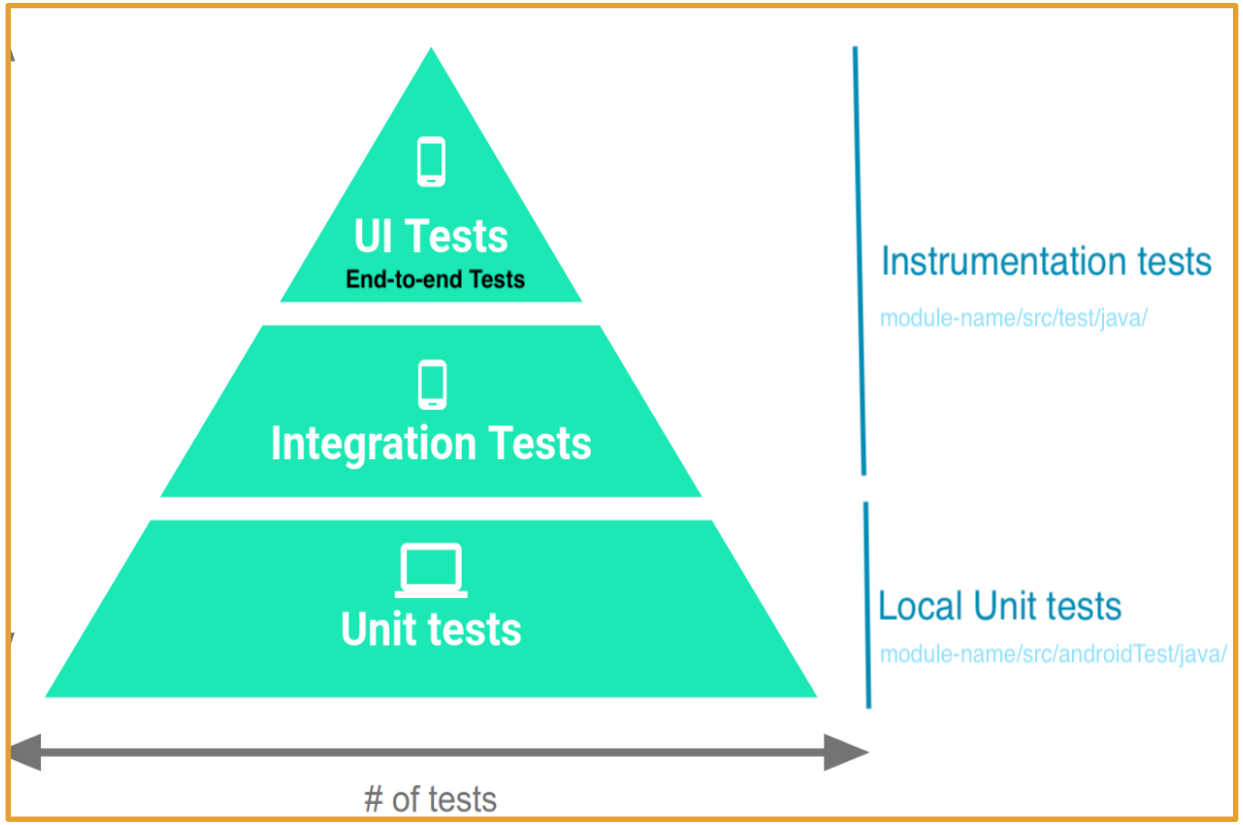
Unit Testing

https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2019

It's called ***unit*** testing because you break down the functionality of your program into discrete testable behaviors that you can test as individual ***units***.

Use a ***unit*** testing framework to create ***unit*** tests, run them, and report the results of these tests.

Rerun ***unit*** tests when you make changes to test that your code is still working correctly. ***Unit*** testing has the greatest effect on the quality of your code when it's an integral part of your software development workflow.



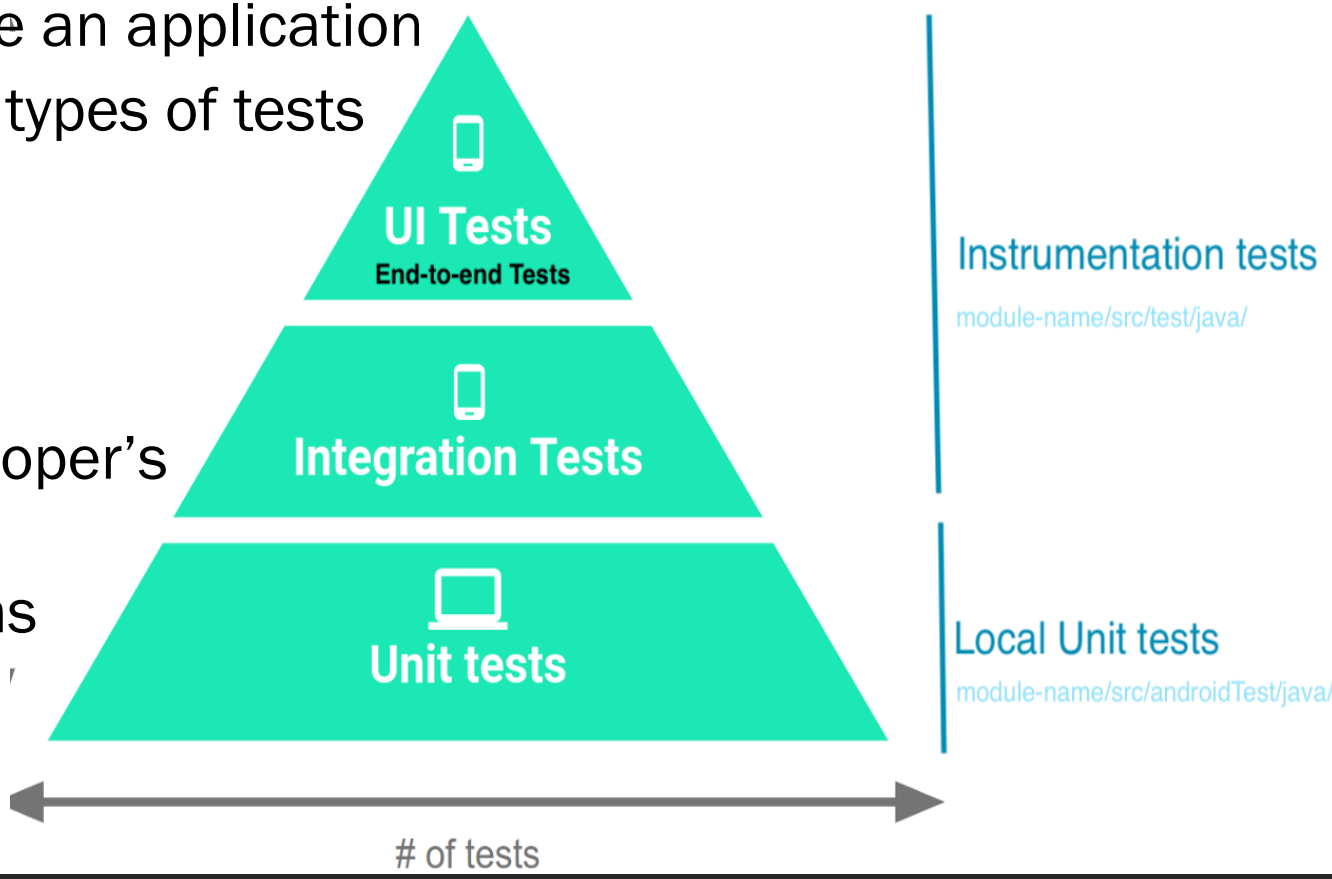
Unit Testing

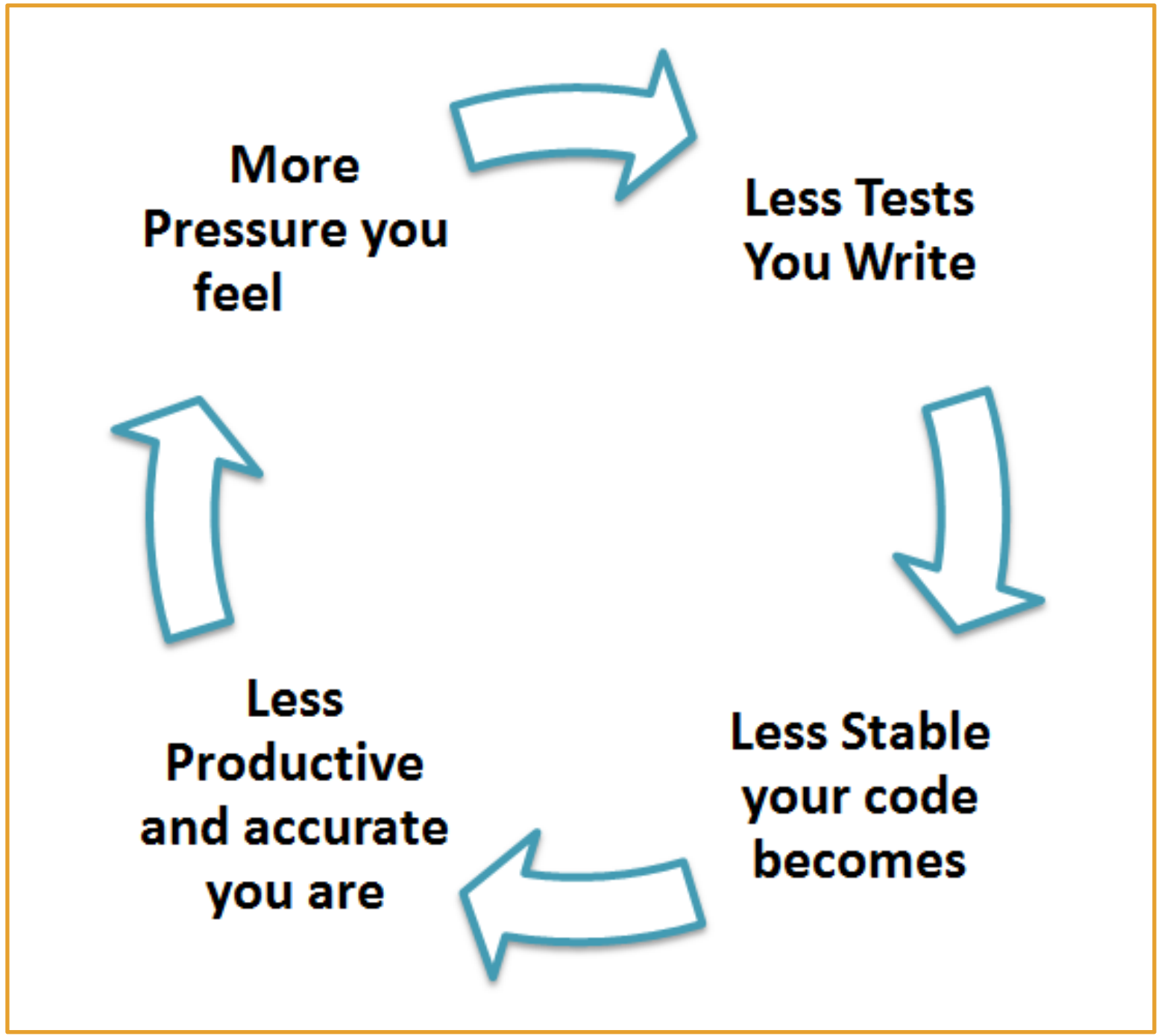
https://docs.microsoft.com/en-us/dotnet/core/testing/
https://docs.microsoft.com/en-us/dotnet/core/testing/#what-are-unit-tests
https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2019

Automated tests are a great way to ensure an application does what's intended. There are multiple types of tests (***integration*** tests, ***web*** tests, ***load*** tests).

Unit tests:

- primarily test methods.
- should only test code within the developer's control.
- should not test infrastructure concerns (databases, file systems, network resources).




More Pressure you feel
Less Tests You Write
Less Stable your code becomes
Less Productive and accurate you are