endava

Test-Driven Development By Example

.NET

Agenda

1. TEST DRIVEN DEVELOPMENT (TDD)
2. UNIT TESTING
3. DEMO 1
4. DEMO 2
5. DEMO 3
6. RESOURCES

1

Test Driven Development (TDD)

Clean Code and TDD

Many forces drive us away from clean code, and even from code that works.

Clean code that works - is the goal of TDD.

1. First we'll solve the "that works" part of the problem.
2. Then we'll solve the "clean code" part.

This is opposite of architecture-driven development, where you solve "clean code" first, then scramble around trying to integrate into the design the things you learn as you solve the "that works" problem.
If you don't drive development with tests, what do you drive it with? Speculation? Specifications?

Why We Need TDD?

- TDD creates tests that serve as documentation. That documentation shortens developer onboarding or hand-offs of codebases. It also increases resiliency to losing knowledgeable people.

- TDD increases developer retention because it makes their jobs easier and more satisfying.

- TDD increases developer ownership and concern for quality.

- TDD helps developers focus on solving the right problems which can make them more efficient, and help ship software that better satisfies a customer's need.

- TDD process allows tests to guide your implementation, resulting in more maintainable and higher-quality code.

TDD History

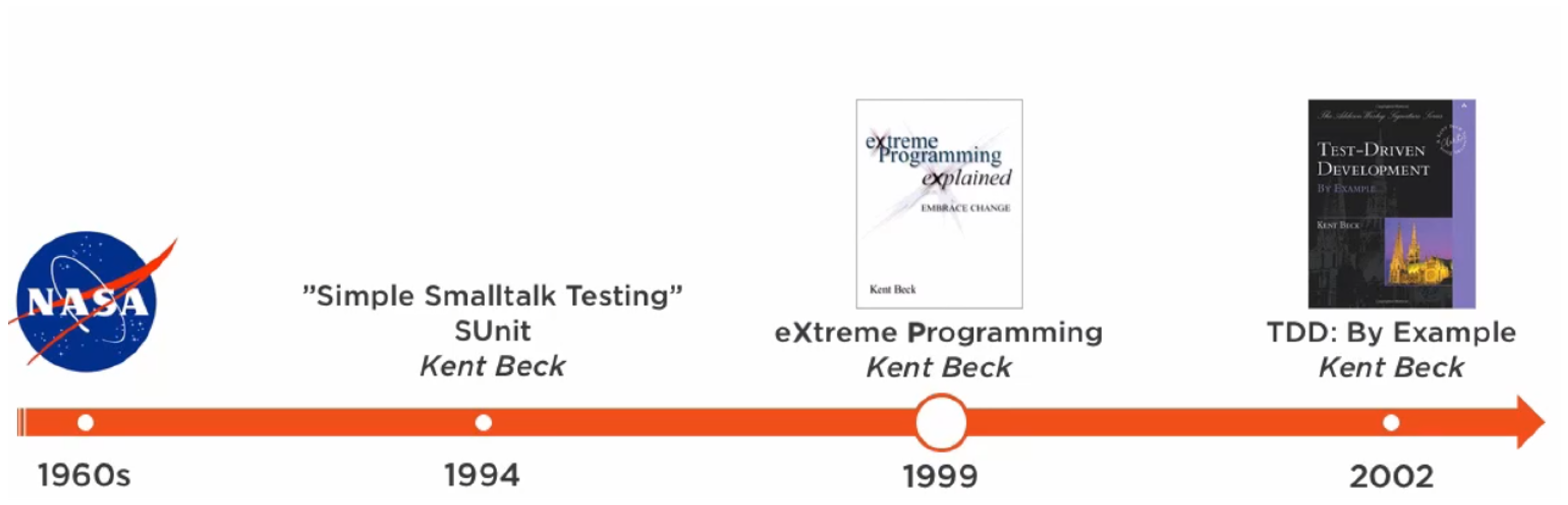
1960: Nasa - Project Mercury -> but not the purest TDD as we know today (test focus approach, more like test-first)

1989-1994: Smalltalk community discussions and Kent Beck smalltalk testing book

1999: Kent Beck - eXtreme Programming

2002: Kent Beck - TDD by example

2000+: Uncle Bob, Martin Fowler, etc...

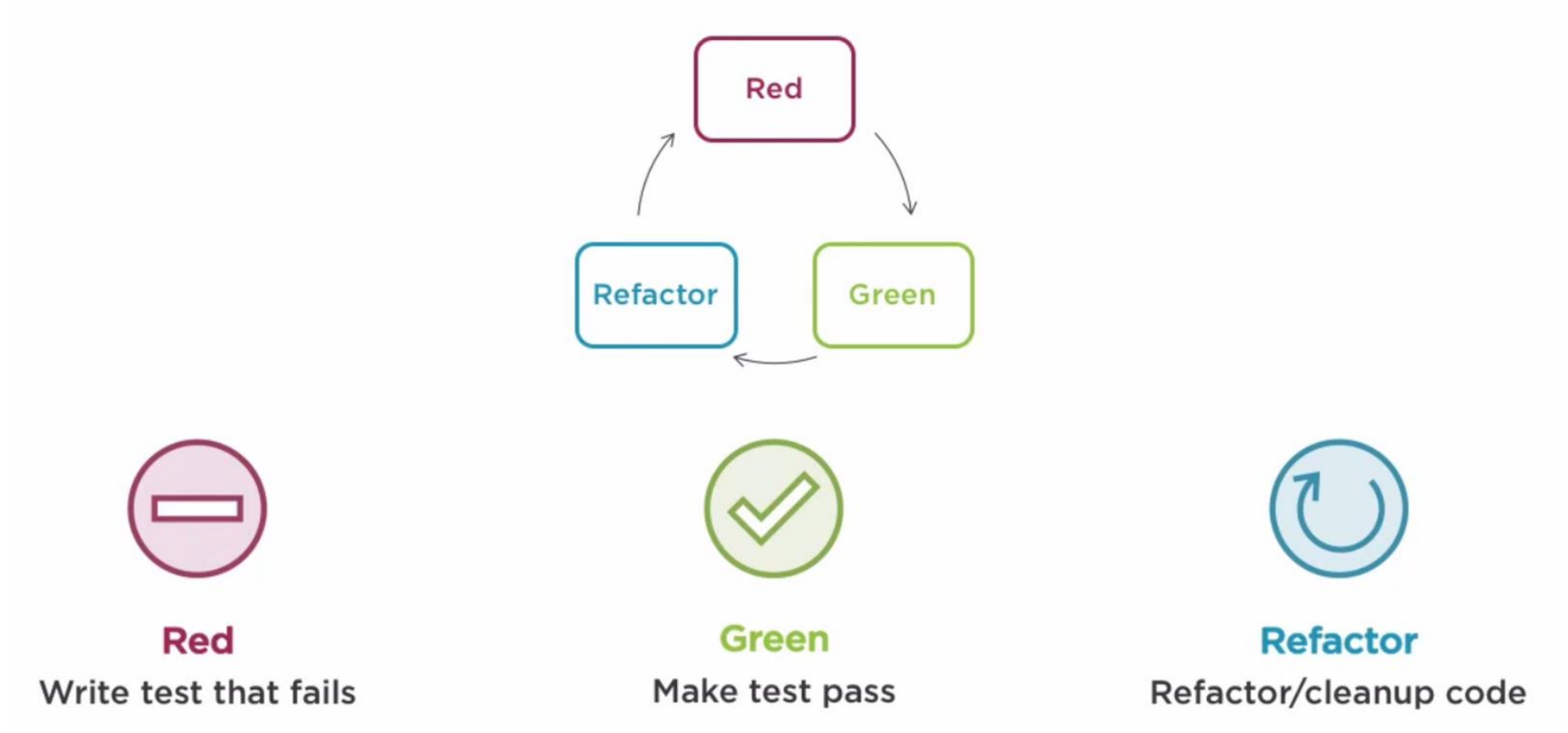


Red-Green-Refactor

1. Red - Write a little test that doesn't work, and perhaps doesn't even compile at first.

2. Green - Make the test work quickly, committing whatever sins necessary in the process.

3. Refactor - Eliminate all of the duplication created in merely getting the test to work.



Rhythm of TDD

1. Quickly add a test.
2. Run all tests and see the new one fail.
3. Make a little change.
4. Run all tests and see them all succeed.
5. Refactor to remove duplications.

*The first three phases need to go by quickly, so we get to a known state with the new functionality.

* Write tests until fear is transformed into boredom.

The Three Laws of TDD – Uncle Bob

1. You are not allowed to write any production code unless it is to make a failing unit test pass.

2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.

3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Writing a Test (Failing Test)

When we write a test, we imagine the perfect interface for our operation. We are telling ourselves a story about how the operation will look from the outside.

Think about how you would like the operation in your mind to appear in your code. You're writing a story. Invent the interface you wish you had.

Make it Run (Green Bar Patterns)

Fake it (‘Till you make it)
Return a constant. Once you have the test running, gradually transform the constant into an expression using variable (replace constants with variables until you have the real code)

Triangulate
Only generalize code when we have two examples or more. (Abstract the implementation when have two or more tests)

Obvious implementation
Type in the real implementation.

If you know what to type -> Obvious implementation.
If you don't know what to type -> Fake it.
If the right design still isn't clear -> Triangulate.

Make it Right (Refactoring, Duplication Removal)

- Two loop structures are similar. By making them identical, you can merge them.

- Two branches of a conditional are similar. By making them identical, you can eliminate the conditional.

- Two methods are similar. By making them identical, you can eliminate one.

- Two classes are similar. By making them identical, you can eliminate one.

Test List (To-Do List)

Before you begin, write a list of all the tests you know you will have to write (on paper, a text file or as TODO comments in tests).

To remind us what we need to do, to keep us focused, and to tell us when we are finished.

As you make the tests run, the implementation will imply new tests. This means that the list will change.

Pros and Cons

Pros

1. Testing the tests
    - First, the test fails. After completing the implementation they pass.
2. Bigger test coverage
    - Fewer bugs
    - Less debugging
3. Better design
    - avoids over-engineering
    - improves the quality of the code
4. Gaining more confidence
    - enables changing the code without breaking it (refactoring without tests is the craziest thing)
5. Not writing the tests having in mind the implementation (not handling only the cases that are already seen in the code)

Cons

1. Slow process?
    - The only way to go fast, is to go well. - Robert C. Martin
2. Don’t want to do it?

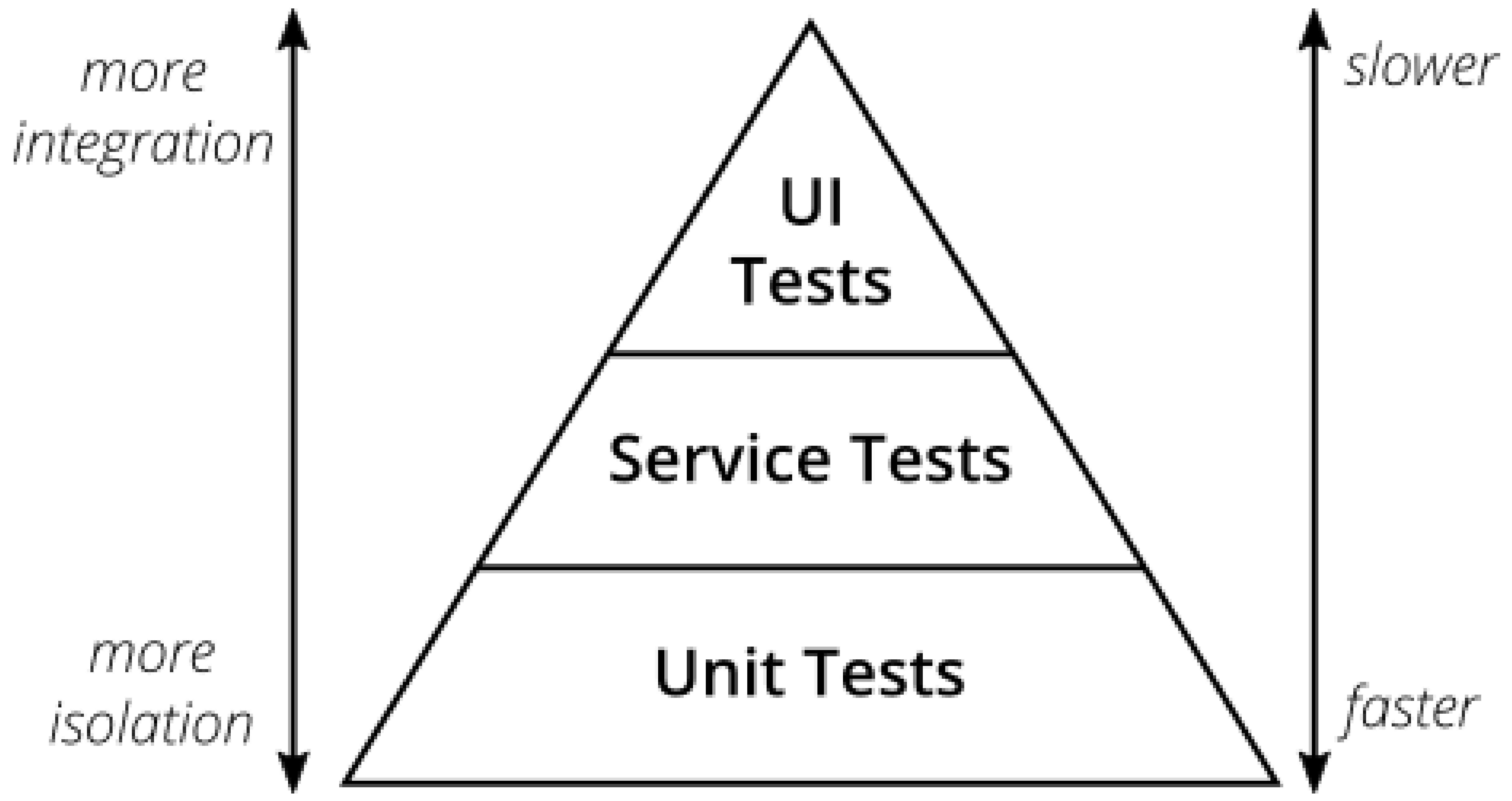
2

Unit Testing

What Is a Unit Test?

Unit test is an automated piece of code that invokes the unit of work being tested and checks some assumptions about a single end result of that unit.

It's consistent in its results as long as the production code hasn't changed.



Namings

Project - [ProjectUnderTest].UnitTests

Class - [ClassName]Tests

Method - [UnitOfWorkName]_[ScenarioUnderTest]_[ExpectedBehavior]

UnitOfWorkName - The name of the method or group of methods or class you're testing (usually a method).

ScenarioUnderTest - The condition under which the unit is tested (such as "bad login", "invalid user"...). You could describe the parameters being sent to the method. Or the initial state of the system when the unit of work is invoked (such "no user exist", "user already exists"...)

ExpectedBehavior - What you expect the tested method to do under the specified conditions (ScenarioUnderTest). This could be one of three possibilities:
- return a value (a real value, or exception)
- change the state of the system
- call a third-party system.

The method name explains the test like:
When I call method X with Y, then it should do Z.

AAA (Arrange, Act, Assert)

Known also as Build – Operate – Check (Uncle Bob - Clean Code book)
Also similar to GWT (Given – When – Then)

Arrange objects, creating and setting them up as necessary.

Act on an object/SUT.

Assert that something is as expected.

Notes:
SUT – System Under Test
When you test something, you refer to the thing you're testing as the SUT.
Some people use CUT (Class Under Test or Code Under Test)

External Dependencies

An external dependency is an object in your system that your code under test interact with and over which you have no control.

Fakes are used when solving external dependencies (Handwritten or Isolation frameworks)

Fakes are used to isolate the unit of work from the external dependencies.

Fake, Stub, Mock…

Fake denotes an object that looks like another object but can be used as a mock or stub! Used to describe mocks and stubs. Whether a fake is a stub or a mock depends on how it's used in the current test.

Stub

Stub is a controllable replacement for an existing dependency (or collaborator) in the system. A stub can never fail your test and is strictly there to simulate various situations.

Mock

Mock is a fake object in the system that DECIDES whether the unit test has passed or failed. It does so by verifying whether the object under test called the fake object as expected.

These are known also as *test doubles*. Test double is a generic term for any case where you replace a production object for testing purposes. Mentioned in Gerard Meszaros book - xUnit Test Patterns.
Test doubles according Gerard Meszaros: Dummy, Fake, Stubs, Spies, Mocks.

The Pillars of Good Unit Tests

1. Trustworthiness - Developers will want to run trustworthy tests, and they'll accept the test results with confidence. Trustworthy tests don't have bugs, and they test the right things.

2. Maintainability - Unmaintainable tests are nightmares because they can ruin project schedules, or they may be sidelined when the project is put on a more aggressive schedule. Developers will simply stop maintaining and fixing tests that take too long to change or that need to change very often on very minor production code changes.

3. Readability - This means not just being able to read a test but also figuring out the problem if the test seems to be wrong. Without readability, the other two pillars fail pretty quickly: Maintaining tests becomes harder, and you can't trust them anymore because you don't understand them.

F.I.R.S.T.

- Fast - Tests should be fast! They should run quickly. When tests run slow, you won't want to run them frequently.

- Independent - Tests should not depend on each other! One test should not set up the conditions for the next test. You should be able to run each test independently and run the tests in any order you like. Isolated!

- Repeatable - Tests should be repeatable in any environment! You should be able to run them in the production environment, in the QA environment, on your laptop while riding home on the train without a network.

- Self-Validating - The tests should have a boolean output, either they pass or fail!

- Timely - The tests need to be written in a timely fashion. Unit tests should be written JUST BEFORE the production code that make them pass!

Only One...

One test class per tested class
- You want to be able to quickly locate all tests for a specific class (or one test class per unit of work under test).
- If we are compelled to split unit tests for a class across multiple files, that may be an indication that the class itself is poorly designed (the class doesn't follow the SOLID principles).

One/single concept per test
- We don't want long test functions that go testing one miscellaneous thing after another.

One assert per test
- The first time an assert fails, it actually throws a special type of exception that is caught by the test runner. That also means no other lines below the line that just failed will be executed.
- Testing only one concern - A concern is a single end result from a unit of work: a return value, a change to system state, or a call to a third-party object.

One mock per test
- In a test where you test only one thing, there should be no more than one mock object. All other fake object will act as stubs.
- Having more than one mock per test usually means you're testing more than one thing.

More Notes

- Avoid logic in tests
- Test private methods through public methods
- Don't skip trivial tests
- Test boundary conditions
- Use a coverage tool
- Test should be easy to implement
- Test should be relevant tomorrow
- Test should be consistent in its results! (should always returns the same result if you don't change anything between runs)

3

Demo 1

Is Leap Year?

Simple implementation where we'll see the Red-Green-Refactor mantra using teeny-tiny steps.

Teeny-tiny steps

TDD is not about taking teeny-tiny steps, it's about BEING able to take teeny-tiny steps.

Would I code day-to-day with steps this small? NO!
But when things get the least bit weird, I'm glad I can!
If you can make steps too small, you can certainly make steps the right size.

4

Demo 2

Move Circle

Simple implementation where we'll focus on the Refactor (Make it right) part (from Red-Green-Refactor) mantra using not so teeny-tiny steps.

5

Demo 3

Desk Booking

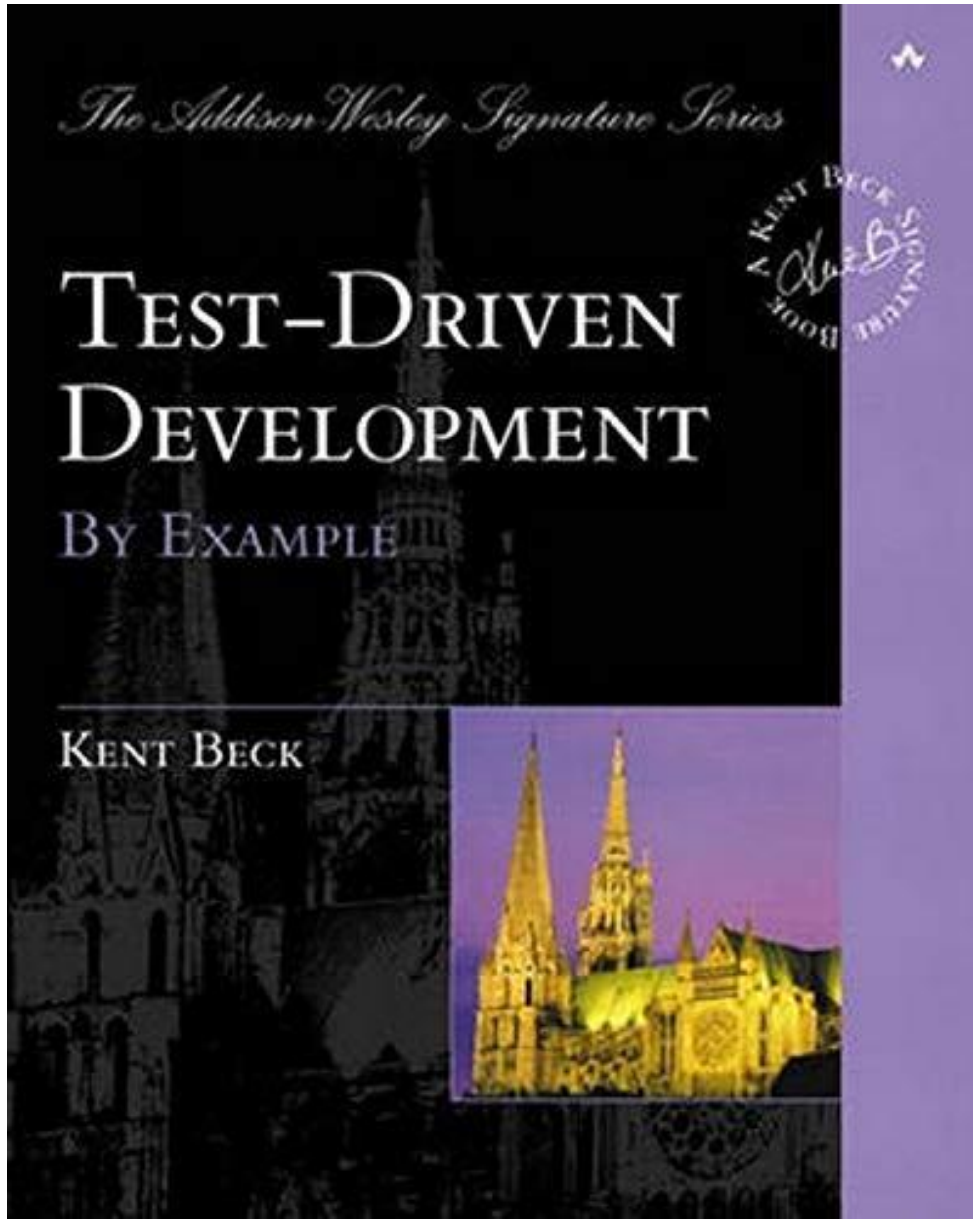
Trying TDD in a “real” application. Web API application, using domain centric architecture – onion, and entity framework for the database (Microsoft SQL Server).

Bigger steps will be used in this TDD, also fakes are needed to solve the many external dependencies.

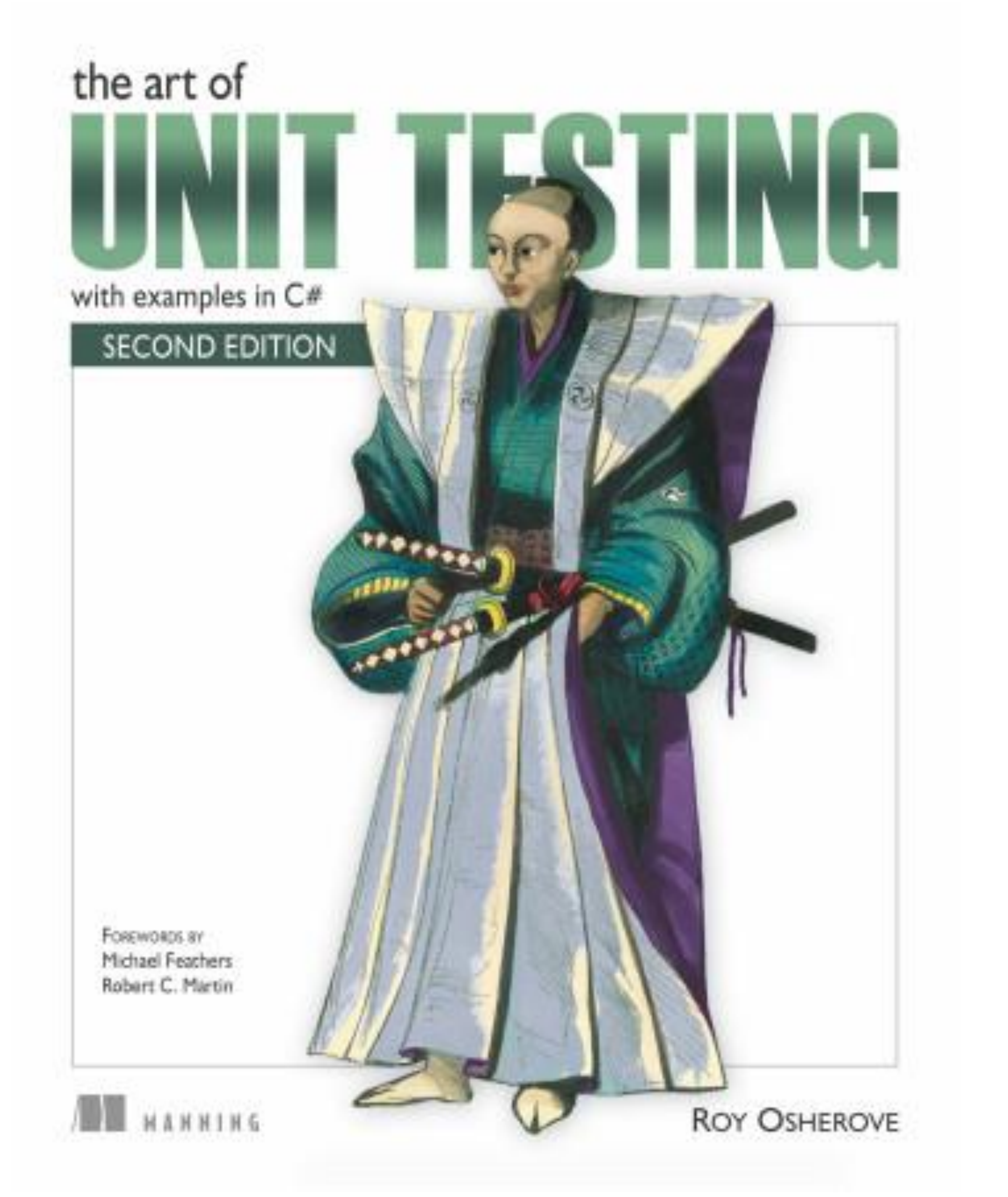
6

Resources

Test Driven Development By Example


The Addison Wesley Signature Series
A Kent Beck Signature Book
Test-Driven Development
By Example
Kent Beck

The Art of Unit Testing (The Unit Testing Bible)



TDD Researches

1. Microsoft - https://www.microsoft.com/en-us/research/wp-content/uploads/2009/10/Realizing-Quality-Improvement-Through-Test-Driven-Development-Results-and-Experiences-of-Four-Industrial-Teams-nagappan_tdd.pdf

2. https://arxiv.org/ftp/arxiv/papers/1711/1711.05082.pdf

endava