Module 1 Day 16

Test-Driven Development

What makes an application?

- Program Data
  - ✓ Variables & .NET Data Types
  - ✓ Arrays
  - ✓ More Collections (list, dictionary, stack, queue)
  - ✓ Classes and objects (OOP)
- Program Logic
  - ✓ Statements and expressions
  - ✓ Conditional logic (if)
  - ✓ Repeating logic (for, foreach, do, while)
  - ✓ Methods (functions / procedures)
  - ✓ Classes and objects (OOP)
  - ❑ Frameworks (MVC)
- Input / Output
  - User
    - ✓ Console read / write
    - ❑ HTML / CSS
    - ❑ Front-end frameworks (HTML / CSS / JavaScript)
  - Storage
    - ❑ File I/O
    - ❑ Relational database
    - ❑ APIs

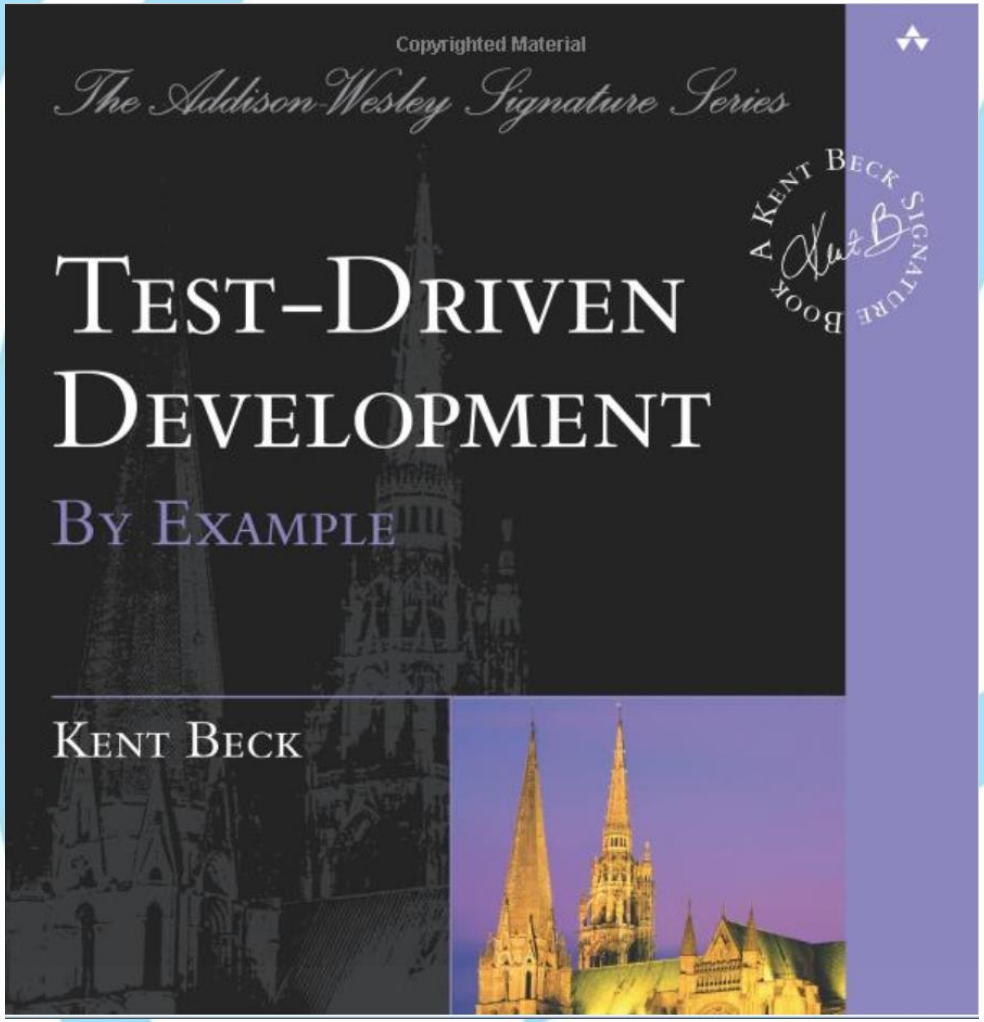
What is Test-Driven Development?

- A software development process
- Very short development cycle
- Tests are written ***before*** the code-under-test
- Code is then written to make the test pass
  - As little code as necessary
- Code is re-factored as needed, and re-tested
- More tests are added, which will "strengthen" the code
- And so on…highly iterative

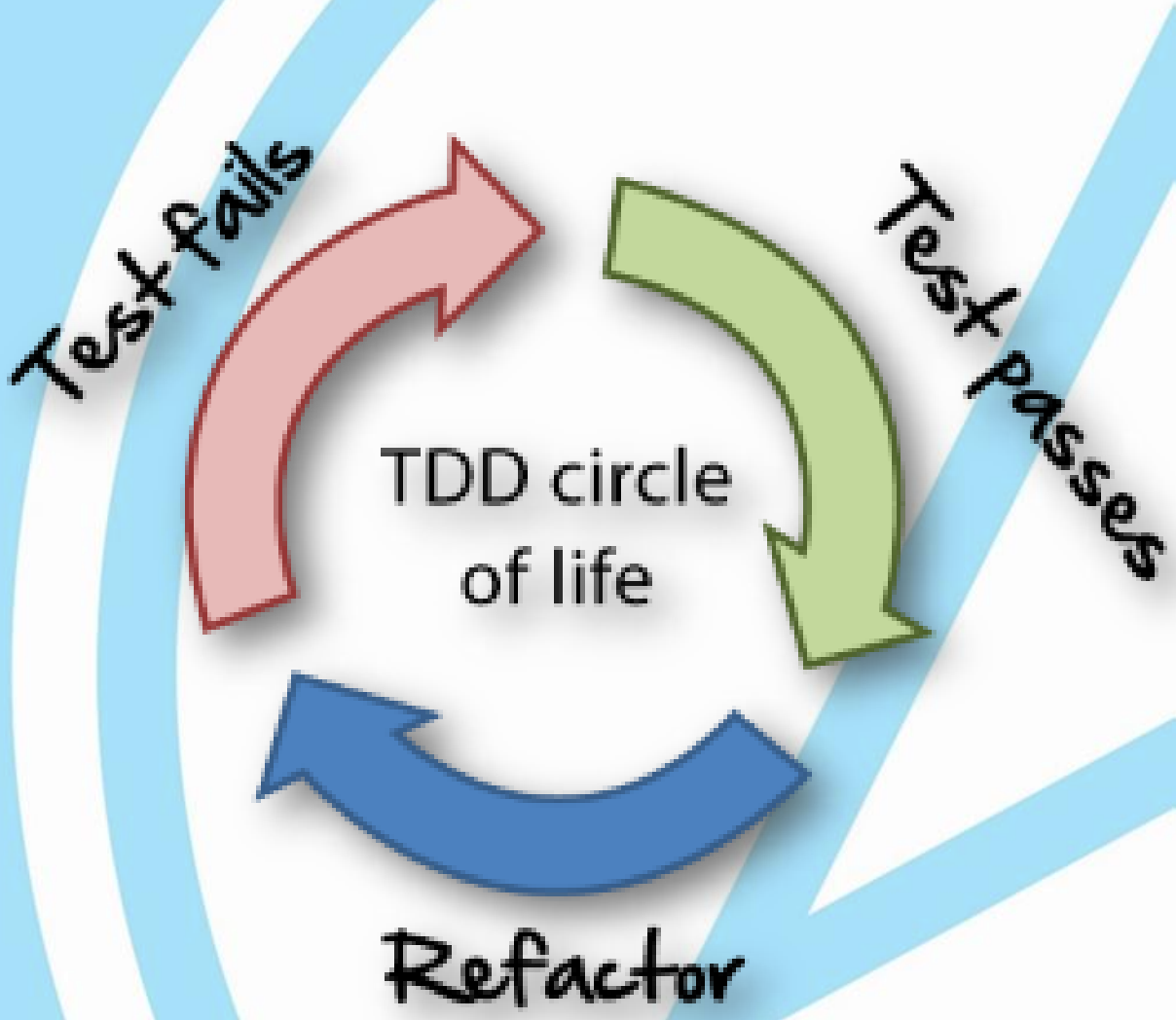
https://en.wikipedia.org/wiki/Test-driven_development

Test-Driven Development By Example

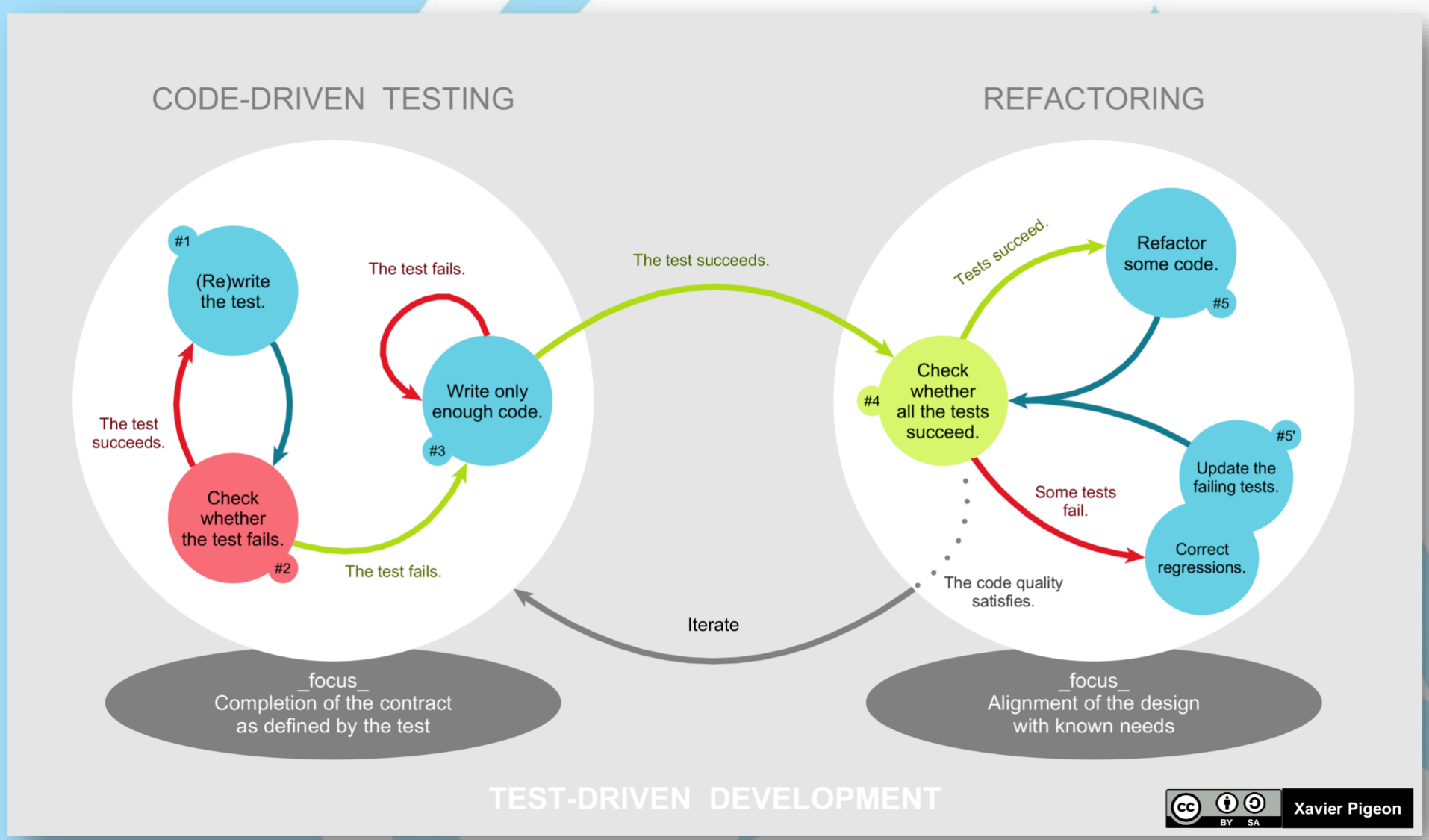
Kent Beck



TDD Circle of Life: Red-Green-Refactor



TDD Lifecycle



TDD Benefits

- Forces programmer to focus on requirements
- More tests are written
  - Uh, that is, tests are written
- Higher code coverage
- No more code is written than is needed (YAGNI)
  - You Ain't Gonna Need It
- In other ways it’s the same as traditional unit testing
  - Code developer is test developer
  - Still must think of edge cases
  - Same tools can apply
  - Same best practices (A-A-A, independent, isolated, targeted)

Mike’s slightly-informed opinion: It’s all about re-factoring with confidence

A Strategy for TDD

1. Think of a Requirement or User Scenario you need to build
2. Create a list of tests needed
3. Write a test (start with the simplest test)
4. **RED** Run the test to see it fail *in the way you expect*
5. Write enough code to make the test build
6. **GREEN** Write enough code to make that test pass (possibly by faking it)
7. **REFACTOR** Generalize the code if possible, by eliminating code duplication or reducing dependencies
8. Go back to step 3

Refactoring

- Eliminate duplicate code
- Extract a method by breaking down long difficult methods
- Extract complex operations to variables
- Introduce constants for magic numbers
- Simplify conditional expressions
- https://www.martinfowler.com/articles/workflowsOfRefactoring/
- https://martinfowler.com/books/refactoring.html
- https://www.refactoring.com