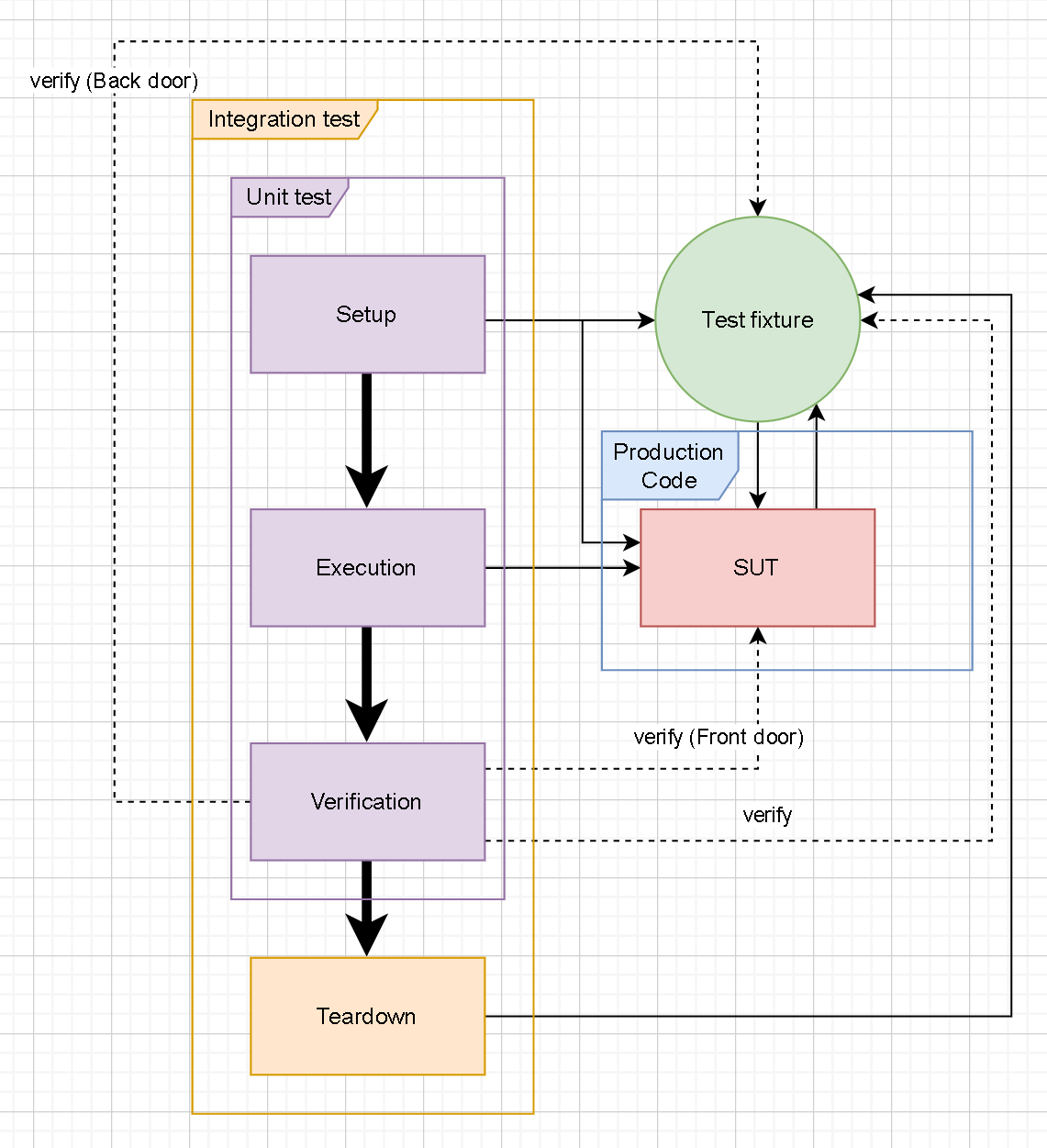
**Four phases of Automated Test**

****

**Terminology:**

**SUT**: System Under Test

**Test fixture**: SUT dependencies (environment)

**Phases:**

**Setup:** create/find SUT or Test fixture

**Execution**: execute something on SUT.

**Verification**: verify state or behavior of SUT/Test fixture.

**Teardown**: clean up after test.

**Unit Test Tools**

**Test Framework**: xUnit (<https://xunit.net/>)

Install-package xUnit

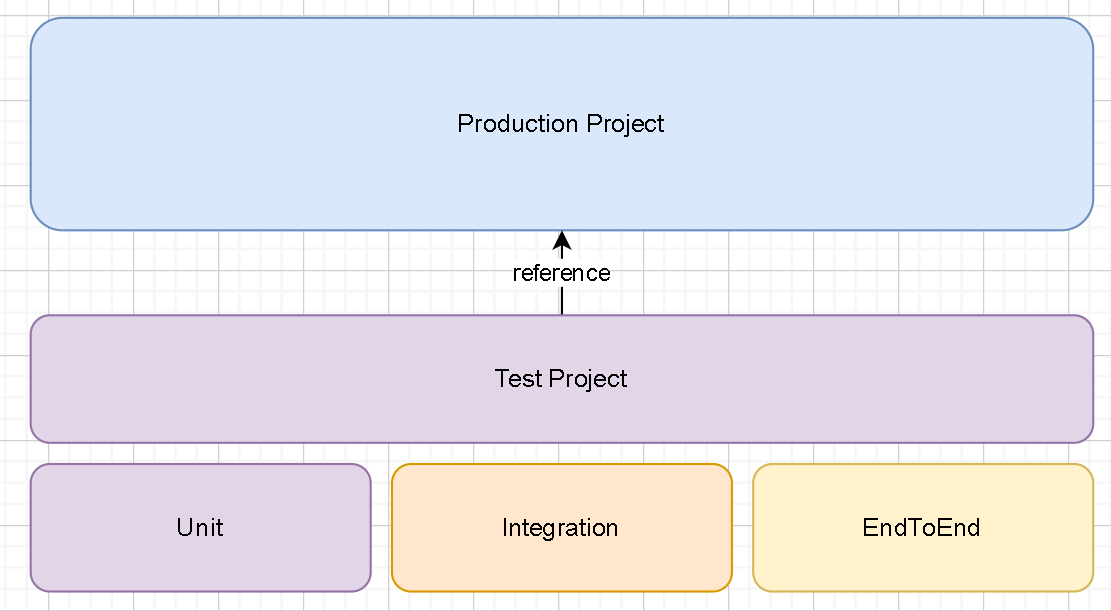
**Test Fixture**: Moq (<https://github.com/devlooped/moq>)

**Test Verification**: Fluent Assertions (<https://fluentassertions.com/>)

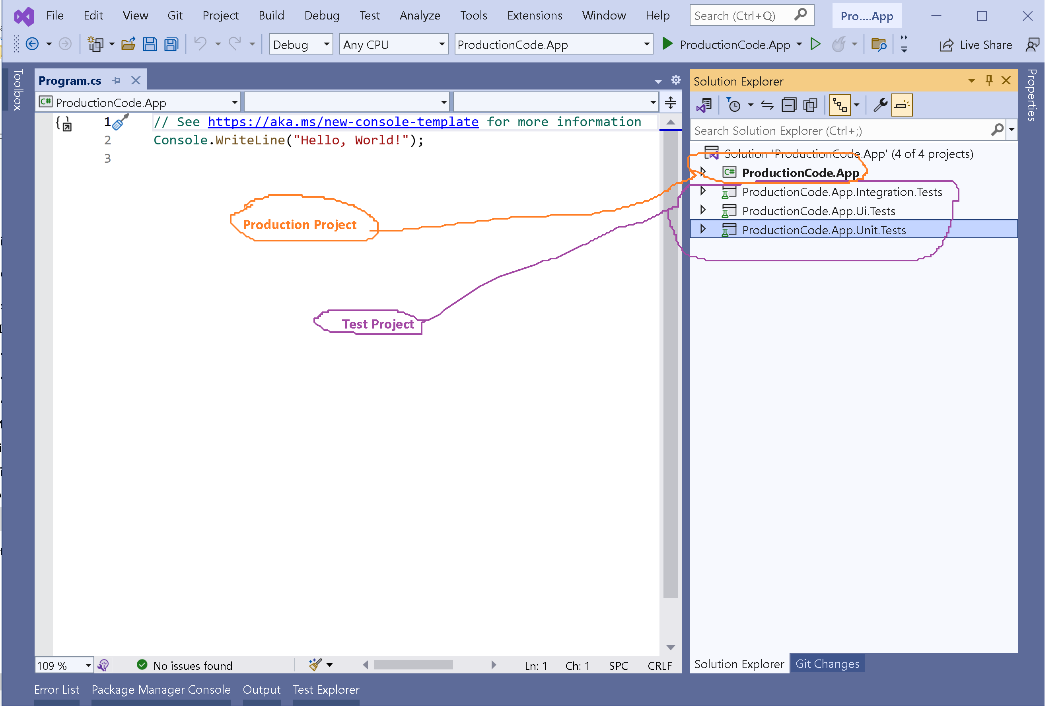
**Test Runner**: Visual Studio Test Explorer, SonarQube Build Pipeline

**Test Quality**: Code Coverage SonarQube

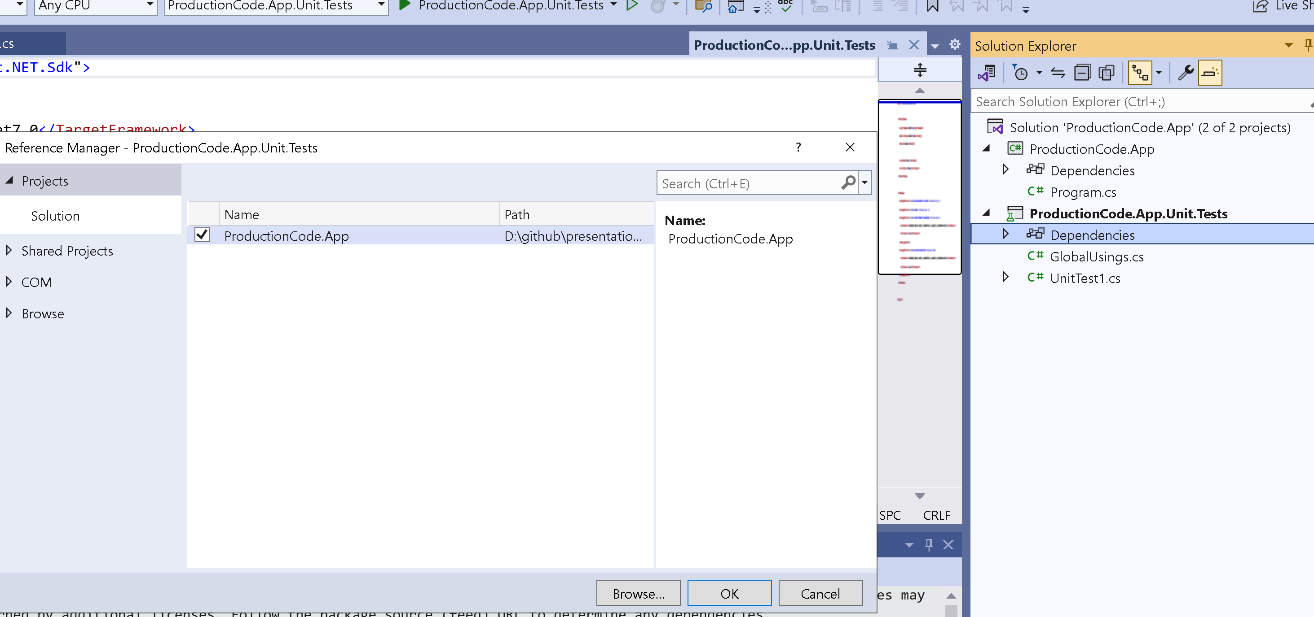
**Unit Test Project Hierarchy**

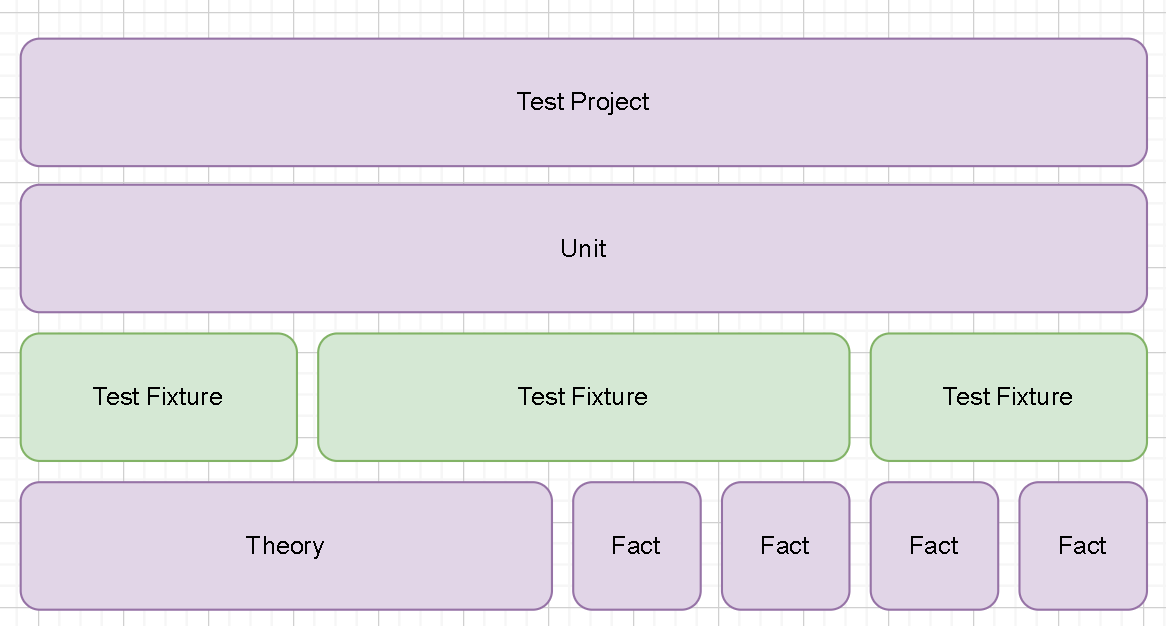
****

**Code**

****

Add product project to unit test project.

****

****

**Code**

Test project

**Unit Test structure**

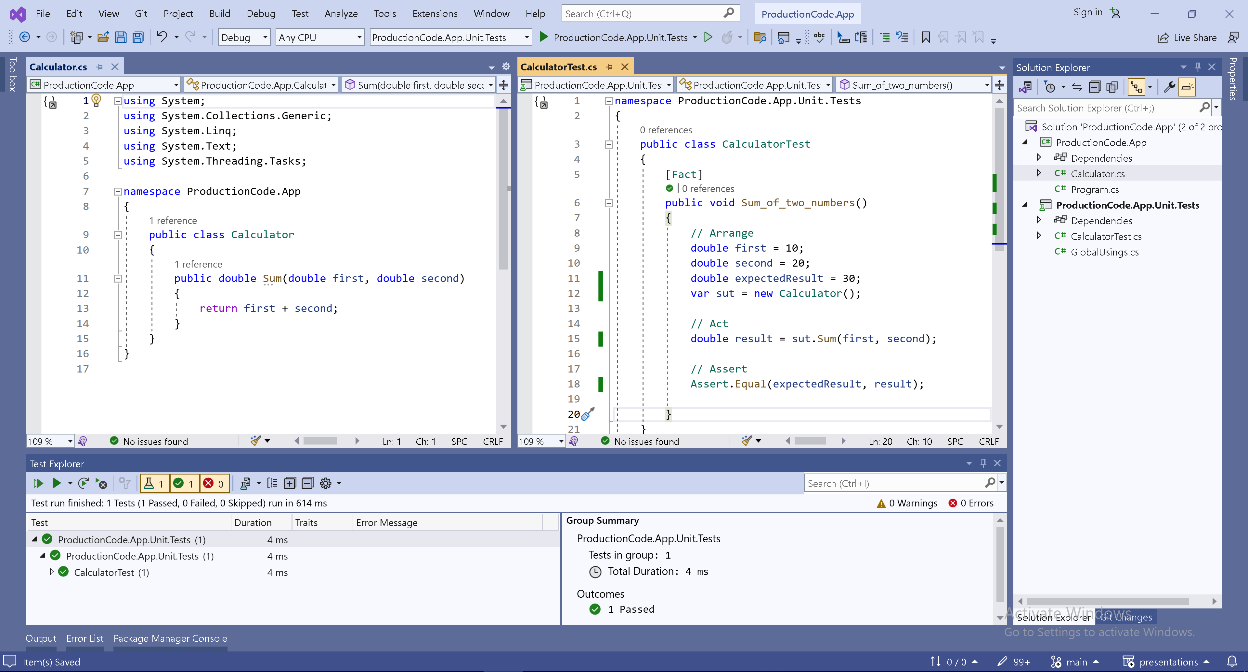
Use Arrange, Action, Assert (AAA) pattern for implement automated test phases

**Set Up** -> **Arrange**: Initialize Test fixture, SUT

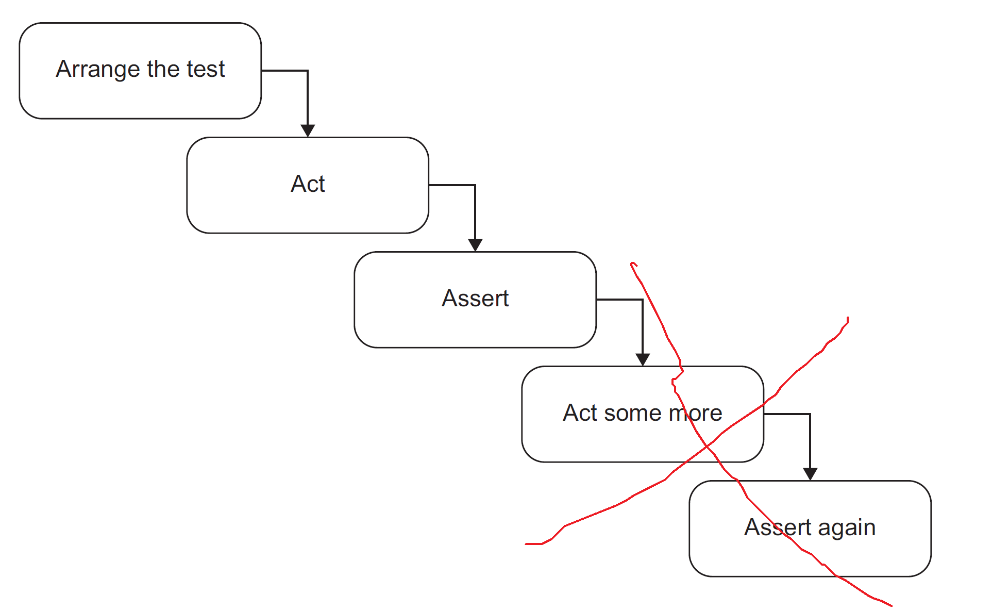
**Execute** -> **Action**: Execute SUT

**Verify** -> **Assert**: Verify result, behavior , state SUT

*Teardown -> Not implement teardown: It for integration test*



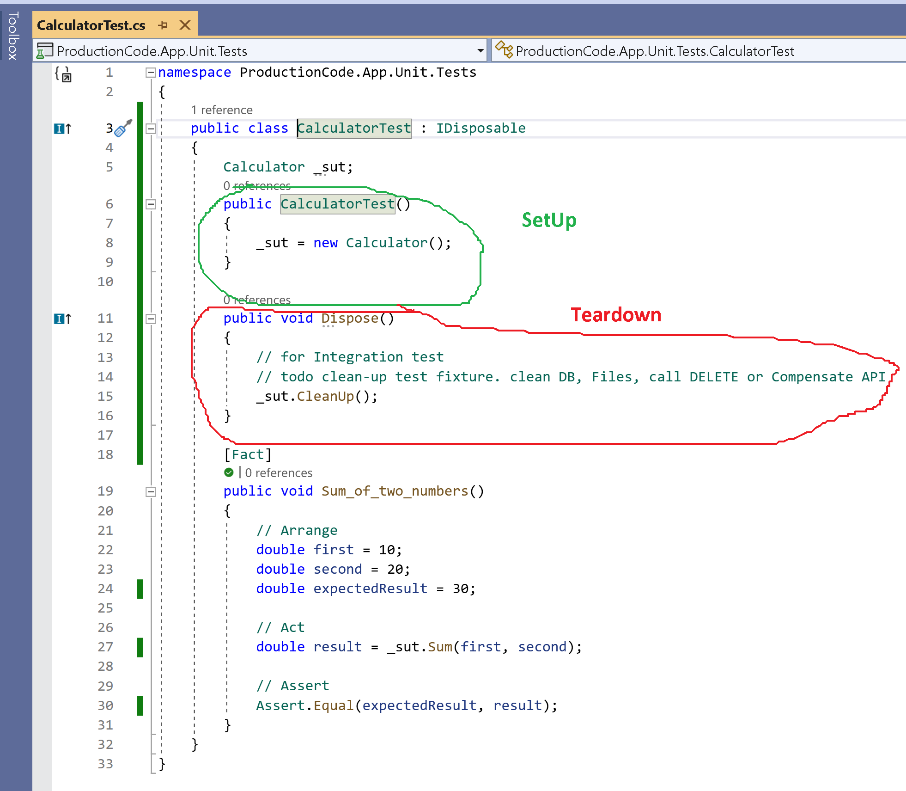
Avoid multiple arrange, act, and assert sections

****

Avoid if statements in tests

A test whether a unit test or an integration test—should be a simple sequence of steps with no branching.

xUnit framework use constructure class for SetUp and implement IDisposable for Teardown



**Reusing test fixtures between tests**

****

****

**Naming a unit test**

[MethodUnderTest]\_[Scenario]\_[ExpectedResult]

Sum\_of\_two\_numbers

Use format

Sum\_TwoNumbers\_ReturensSum

The initial name written in plain English is much simpler to read. It is a down-to-earth

description of the behavior under test.

**Unit test naming guidelines**

* Don’t follow a rigid naming policy.
* Name the test as if you were describing the scenario to a non-programmer who is familiar with the problem domain.
* Separate words with underscores.

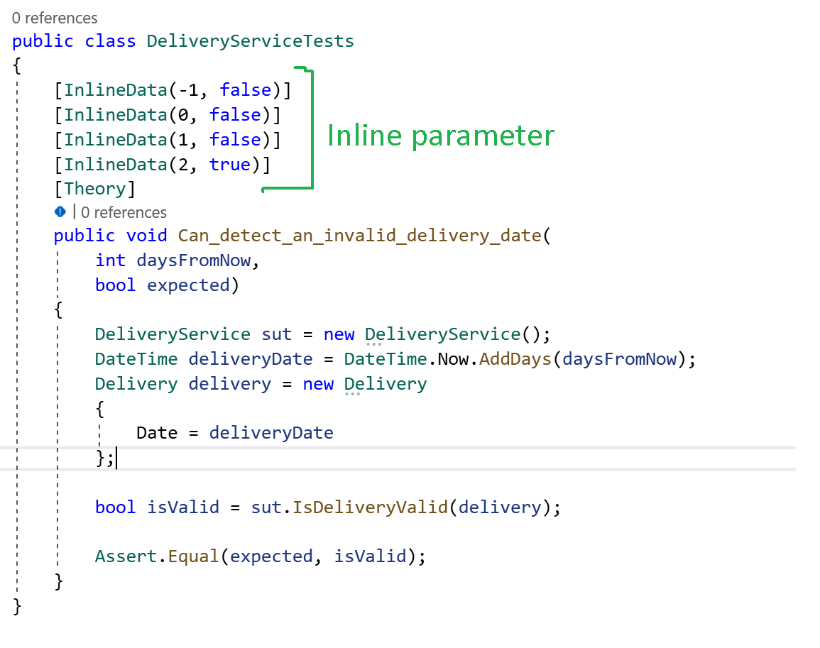
Example:

**IsDeliveryValid\_InvalidDate\_ReturnsFalse**

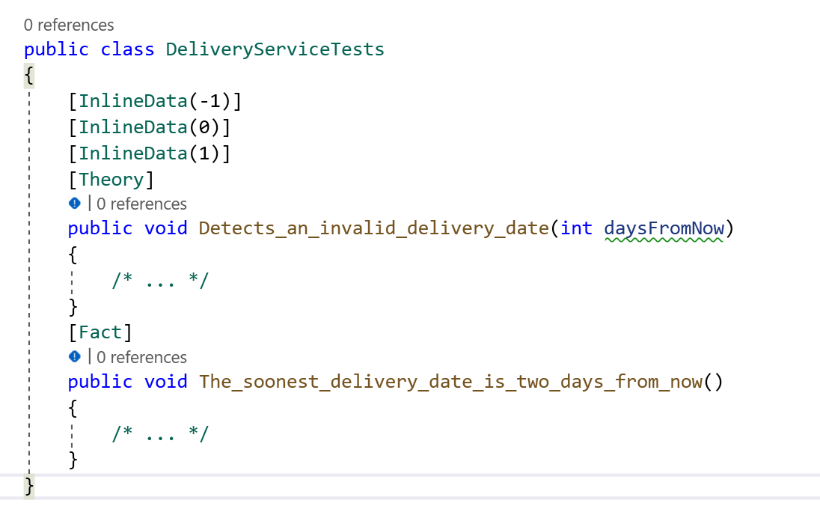
Rename use guideline

**Delivery\_with\_invalid\_date\_should\_be\_considered\_invalid**

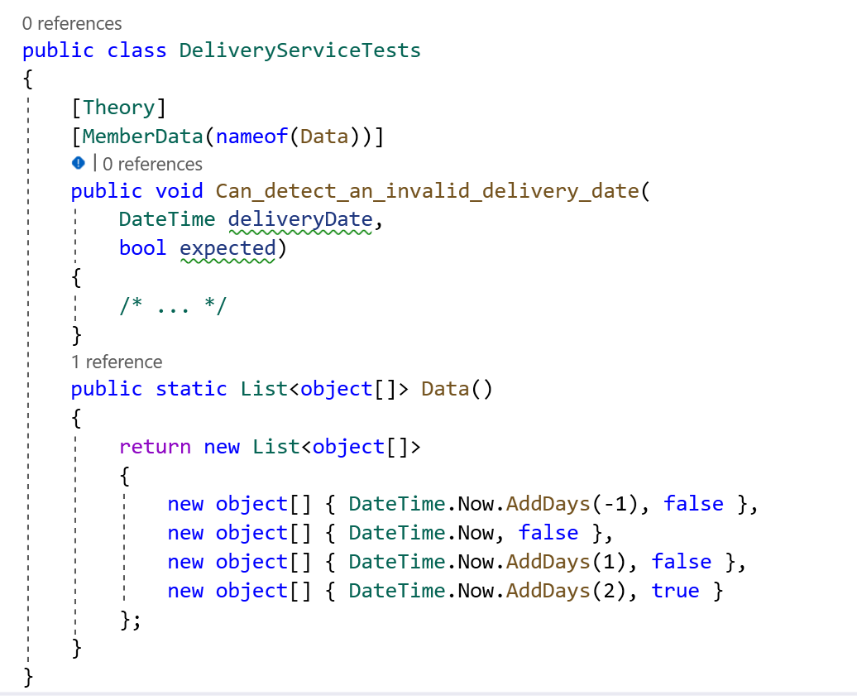
**Parameterized Tests**



**Refactor two test verify the positive and negative scenarios**

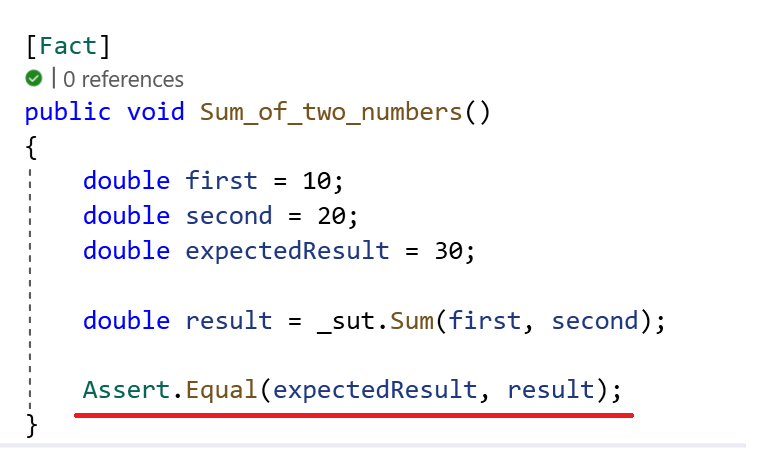


**Generating data for parameterized tests**

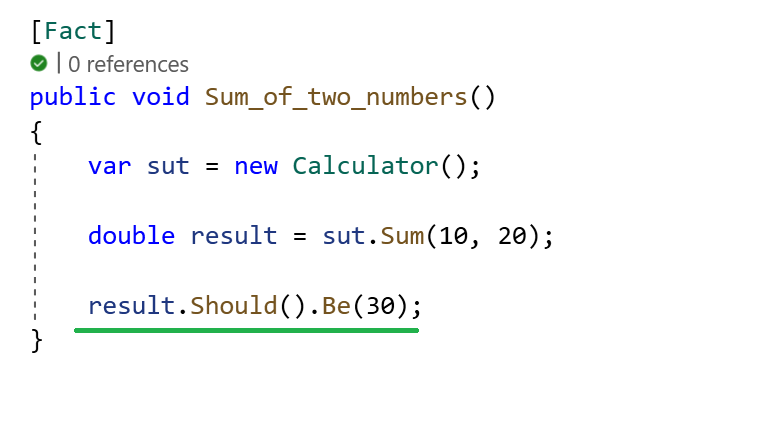


**Using an assertion library to further improve test readability**

*install-package FluentAssertions*

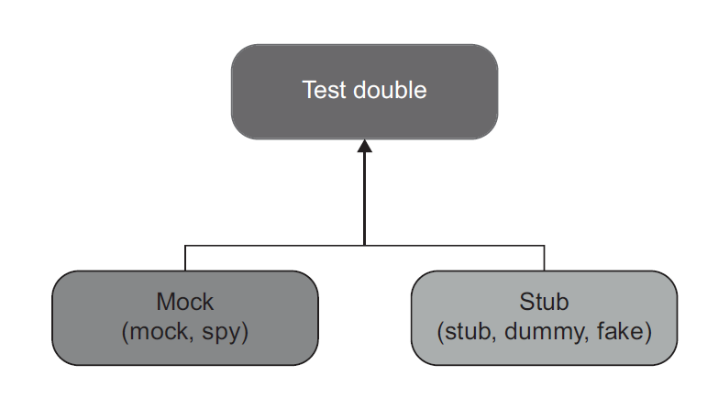


Refactor to apply *FluentAssertions*



**Isolation Unit Test with Test doubles**

Test doubles: *dummy, stub, spy, mock and fake* they can all grouped together into just two types: mocks and stubs.

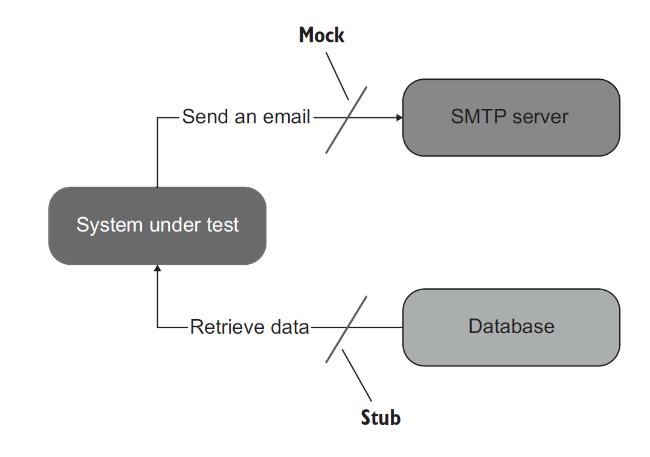


* **Mocks** help to emulate and examine outcoming interactions. These interactions

are calls the SUT makes to its dependencies to change their state.

* **Stubs** help to emulate incoming interactions. These interactions are calls the

SUT makes to its dependencies to get input data

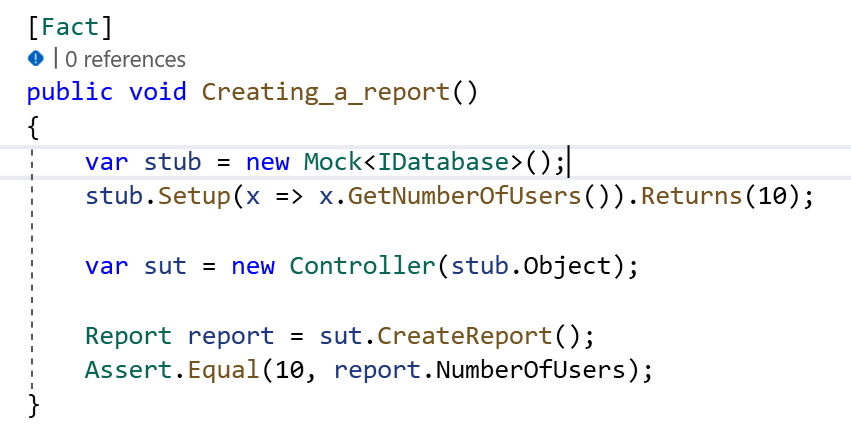
****

*install-package Moq*

Use package Moq mock class for create Mock



Use package Moq mock class for create Stub

****

**How mocks and stubs relate to commands and queries**

****

**Mocking best practices**

* Applying mocks to unmanaged dependencies only
* Verifying the interactions with those dependencies at the very edges of your

System \*

* Using mocks in integration tests only, not in unit tests
* Always verifying the number of calls made to the mock
* Only mock types that you own \*\*
  + The guideline states that you should always write your own adapters on top of third-party libraries and mock those adapters instead of the underlying types.

Recommend Adapters pattern in effect, act as an anti-corruption layer between your code and the external world

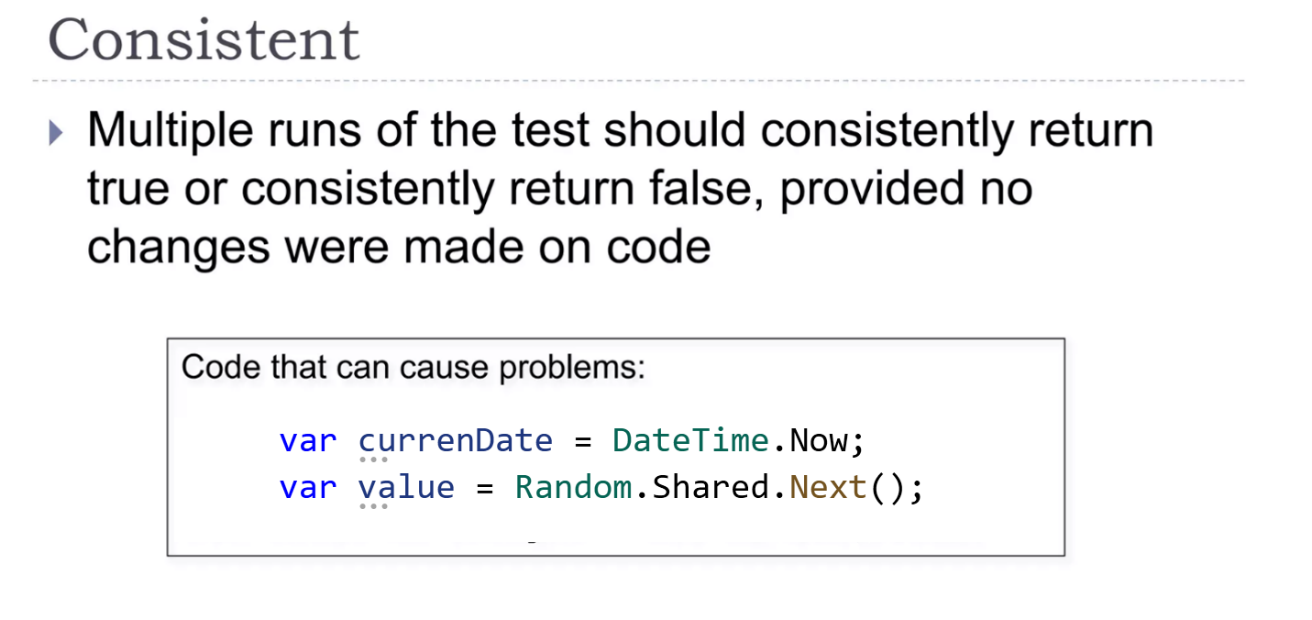
* + - Abstract the underlying library’s complexity
    - Only expose features you need from the library
    - Do that using your project’s domain language

\* "Verifying the interactions with those dependencies at the very edges of your system" เป็นภาษาไทยได้เป็น "การตรวจสอบการสื่อสารกับพวกขึ้นตรงบริบทของความขึ้นตรงของระบบของคุณ" ซึ่งหมายถึงการตรวจสอบหรือการยืนยันการสื่อสารกับพวกขึ้นตรง (dependencies) ที่ต่อระบบของคุณที่ขอบของระบบ เราพูดถึงการตรวจสอบการสื่อสารกับความขึ้นตรงเพราะว่า "the very edges of your system" หมายถึงส่วนที่อยู่ในตำแหน่งที่ขอบของระบบ ซึ่งอาจเป็นส่วนที่เชื่อมต่อกับระบบภายนอกหรือพวกขึ้นตรงที่รับอินพุตหรือส่งอินพุตออกไปจากระบบของคุณ เราต้องการตรวจสอบว่าการสื่อสารนี้ทำงานอย่างถูกต้องและเหมาะสมที่ขอบของระบบนั้น ๆ และไม่เกิดข้อผิดพลาดในส่วนนี้ที่อาจมีผลกระทบต่อระบบโดยรวมของคุณ

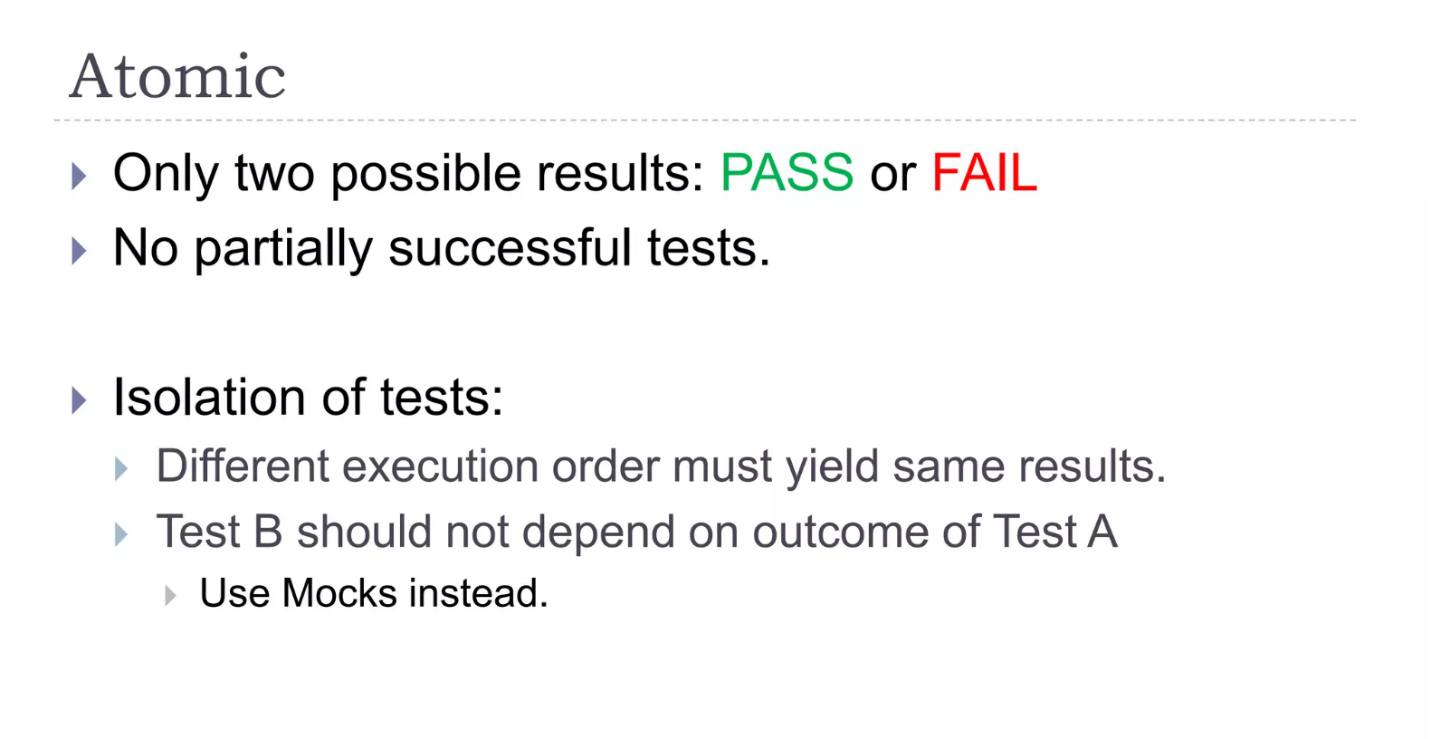
\*\* "Mocking only types that you own" เป็นภาษาไทยได้เป็น "การสร้างม็อคเพียงแค่ชนิดข้อมูลที่คุณครอบครอง" ซึ่งหมายถึงการใช้เทคนิคการสร้างม็อค (mocking) ในการทดสอบโปรแกรมหรือระบบเพื่อสร้างวัตถุที่จำลองขึ้น (mock object) เฉพาะสำหรับชนิดข้อมูลที่คุณเป็นเจ้าของ หรือมีความรับผิดชอบ นั่นหมายความว่าคุณสร้างม็อคเฉพาะส่วนของระบบหรือโครงสร้างข้อมูลที่คุณควบคุมและไม่สร้างม็อคสำหรับส่วนของระบบที่อยู่ในความรับผิดชอบของผู้อื่น นี้เป็นแนวทางที่ควรปฏิบัติในการทดสอบแบบม็อคเพื่อรักษาความเป็นอิสระและความเป็นความรับผิดชอบในการทดสอบของแต่ละชิ้นส่วนของระบบของคุณ

**Unit Test Best Practices**

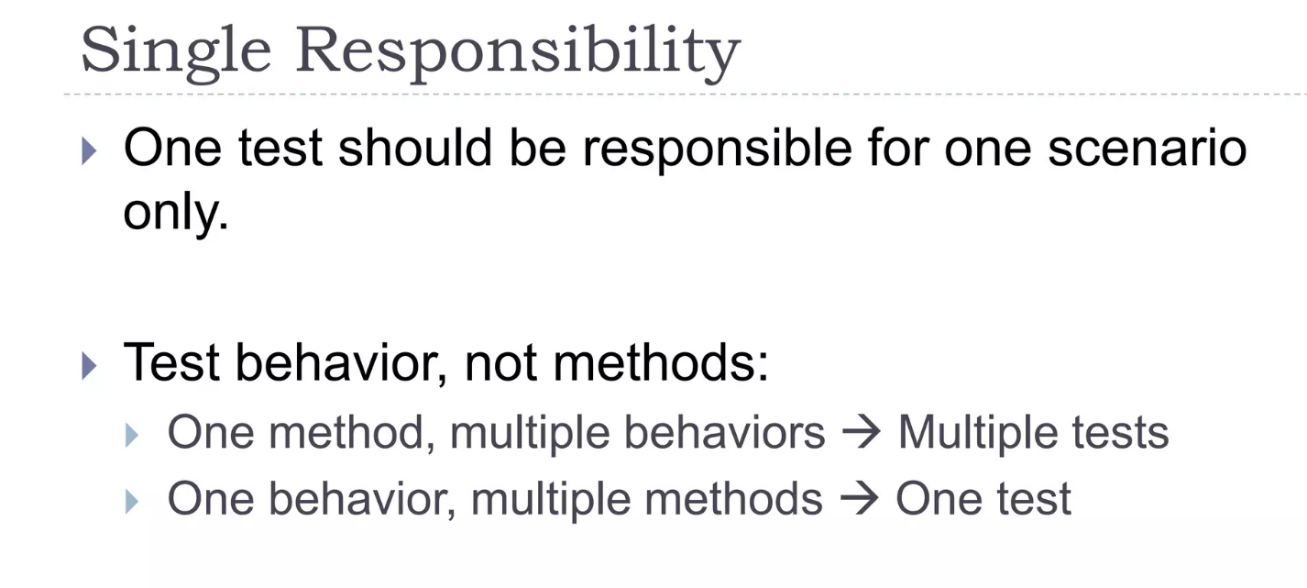
Consistent



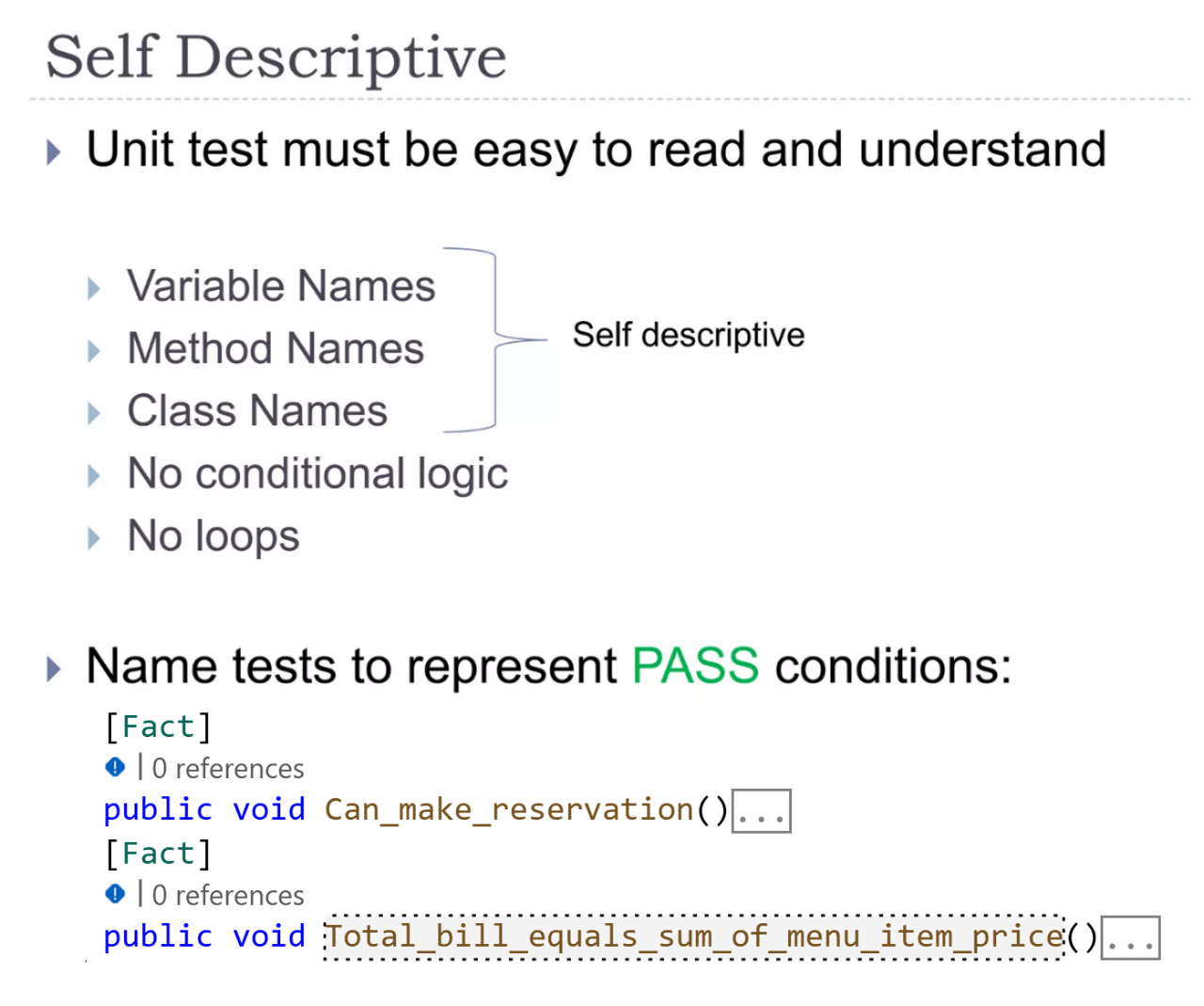
Atomic



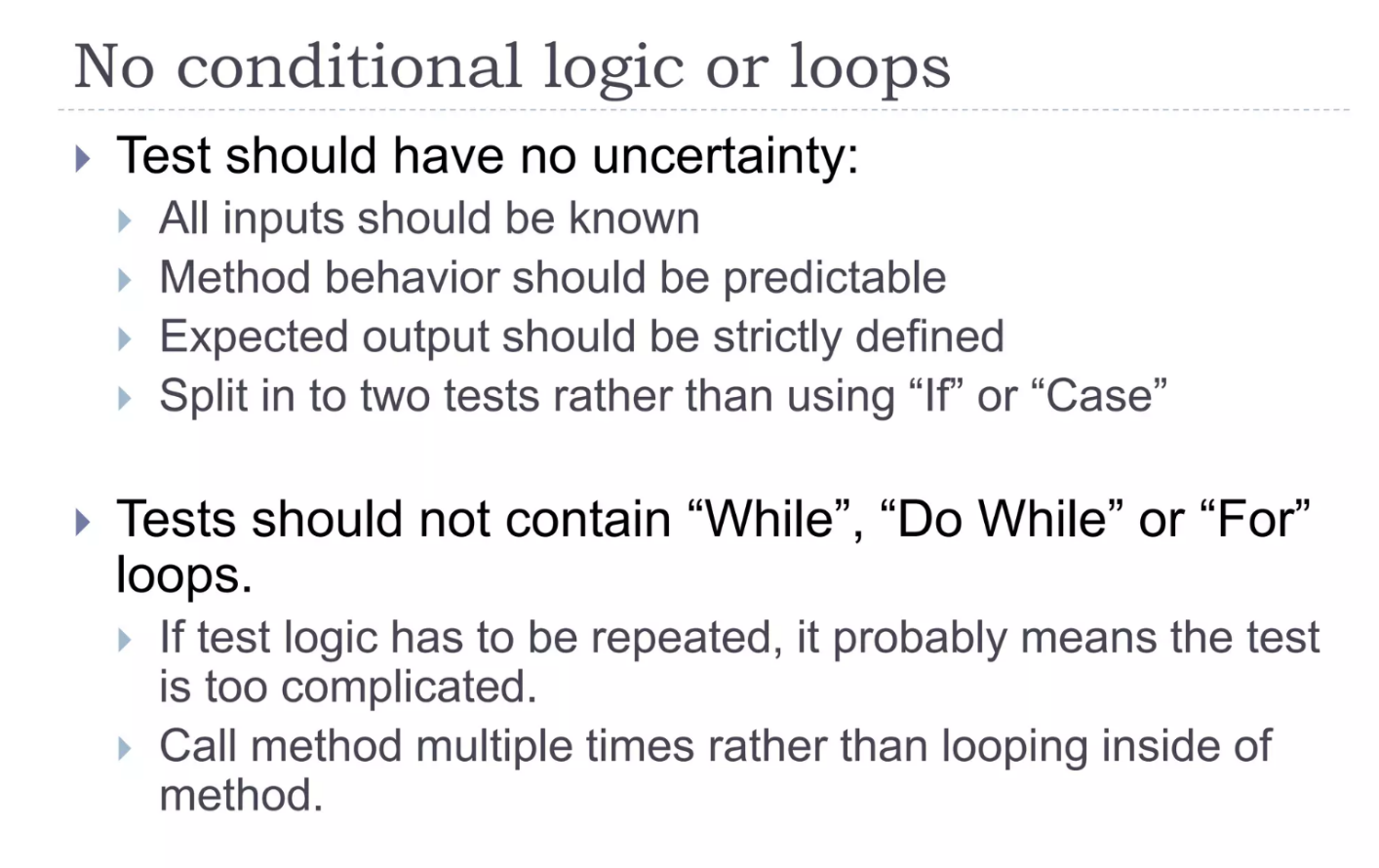
Single Responsibility

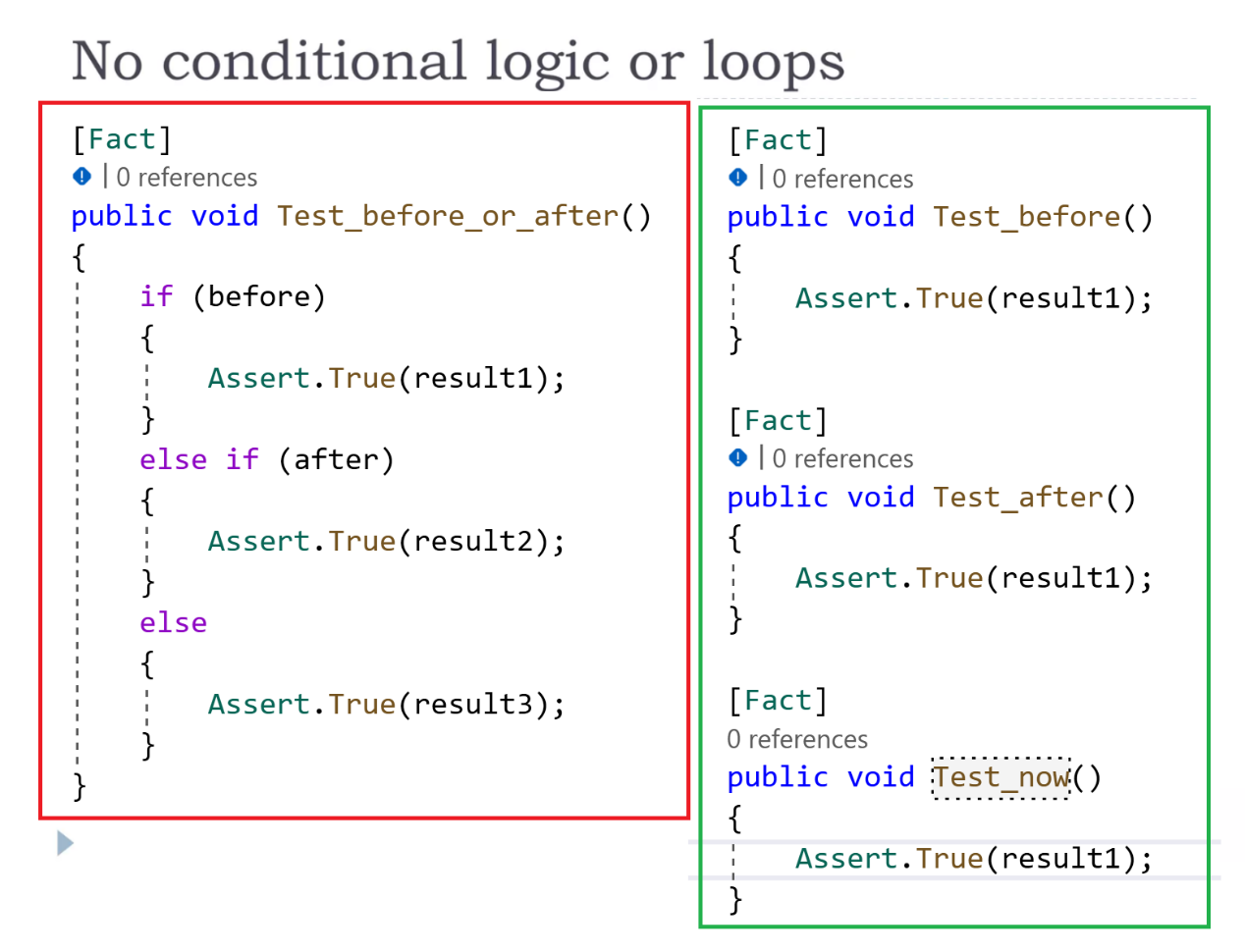


Self-descriptive

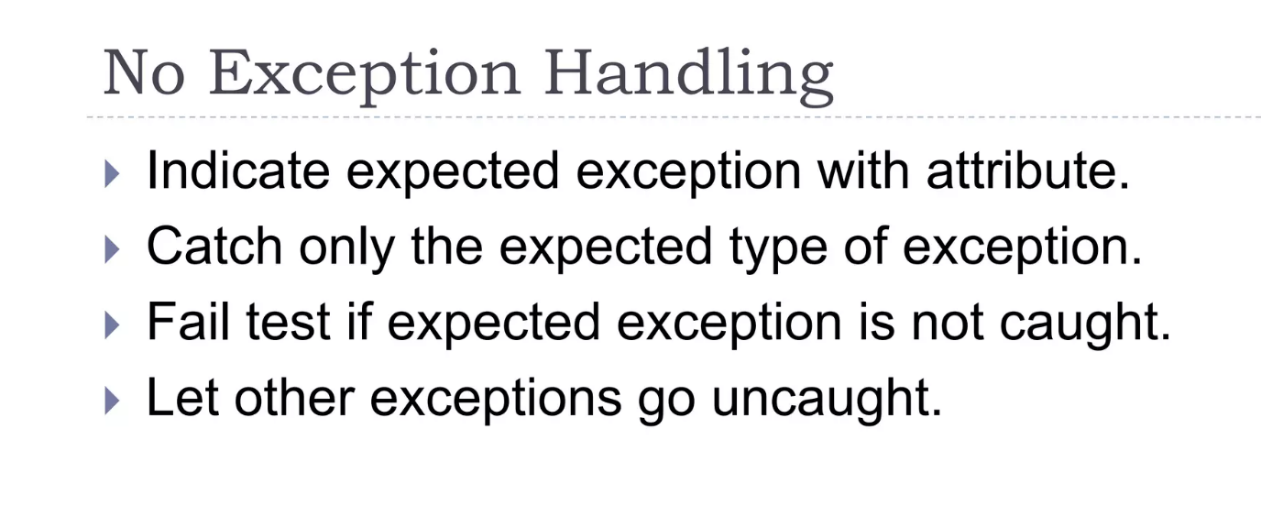


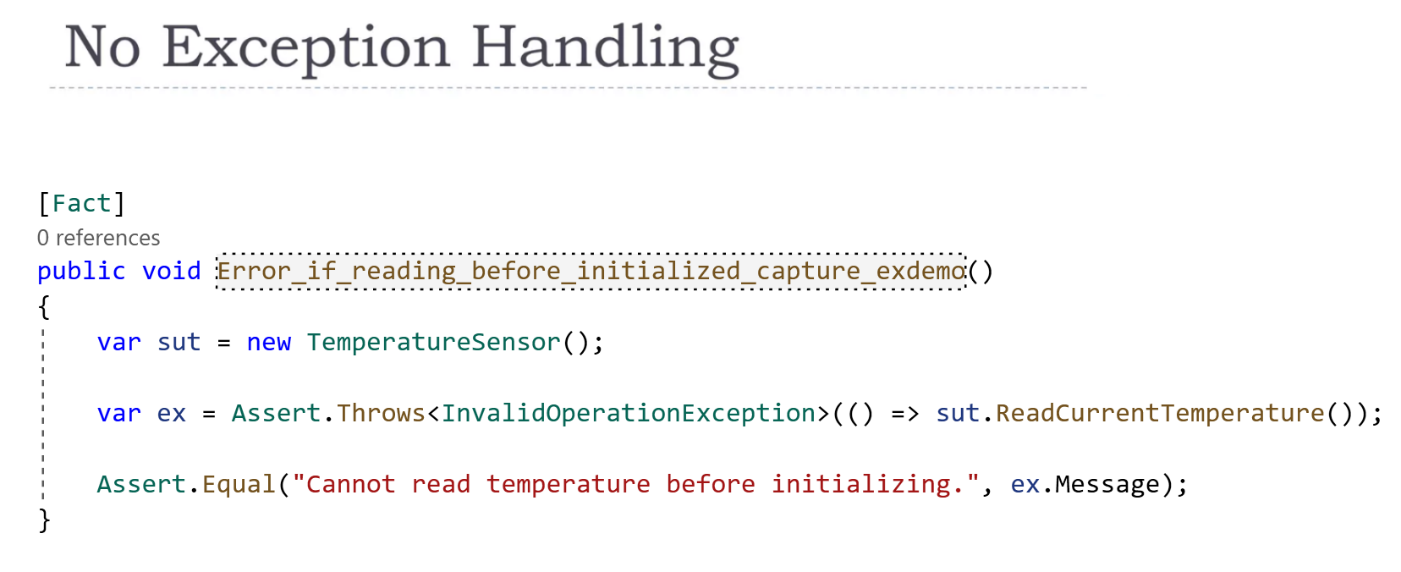
No conditional logic or loops



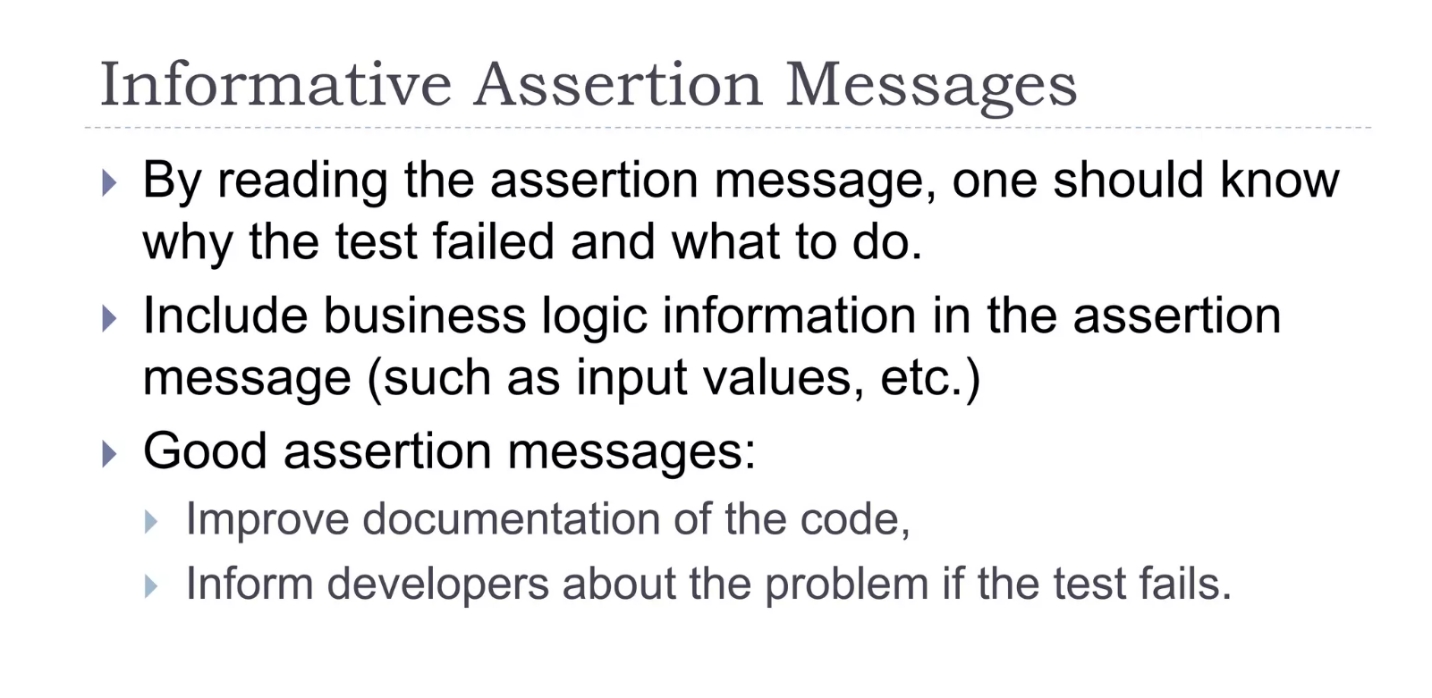


No exception handling

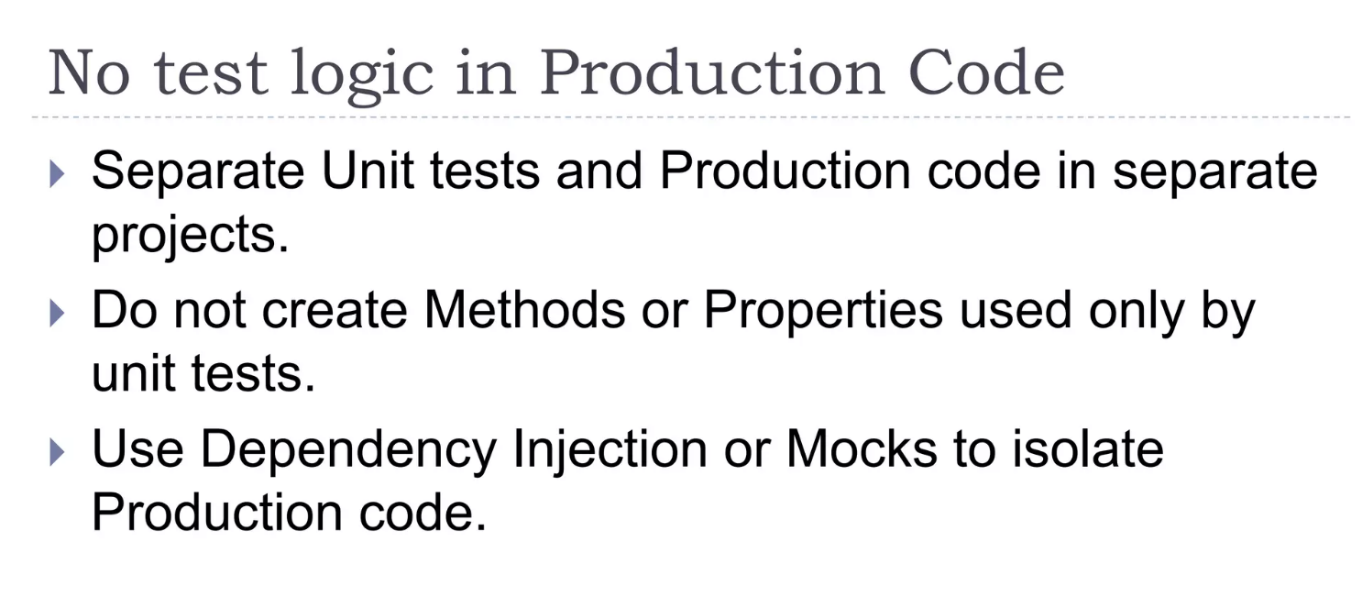




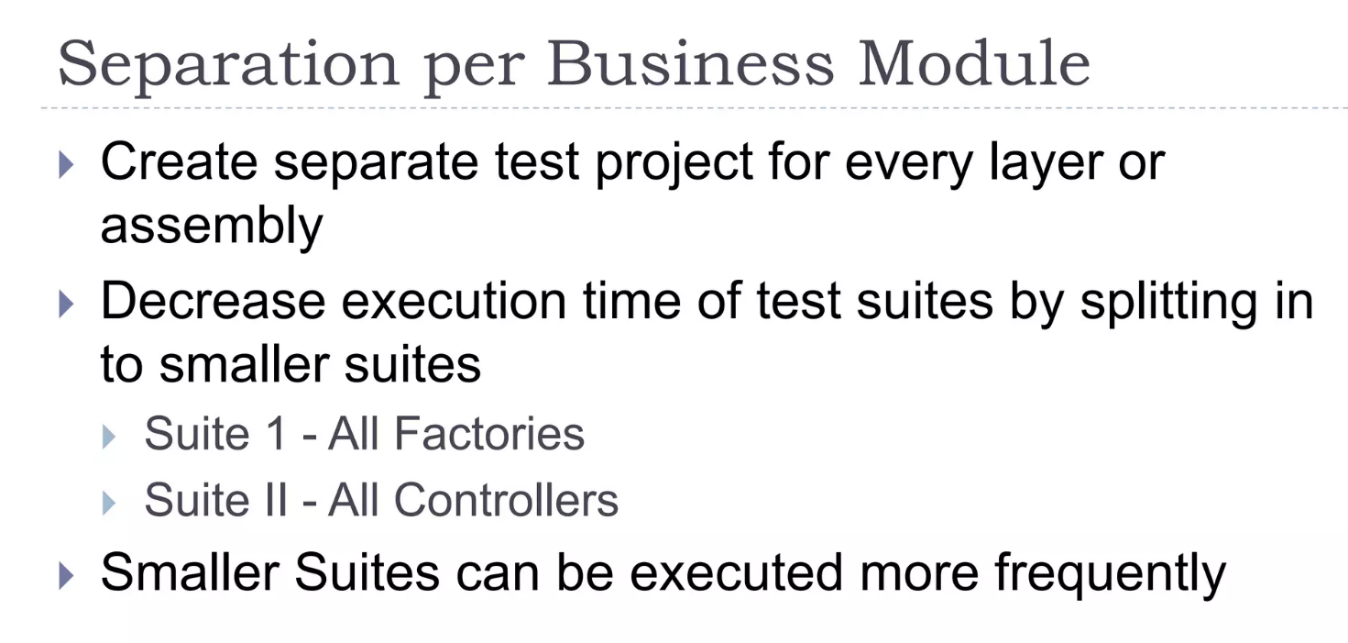
Informative Assertion messages



No test logic in production code



Separation per business module



Separation per type

