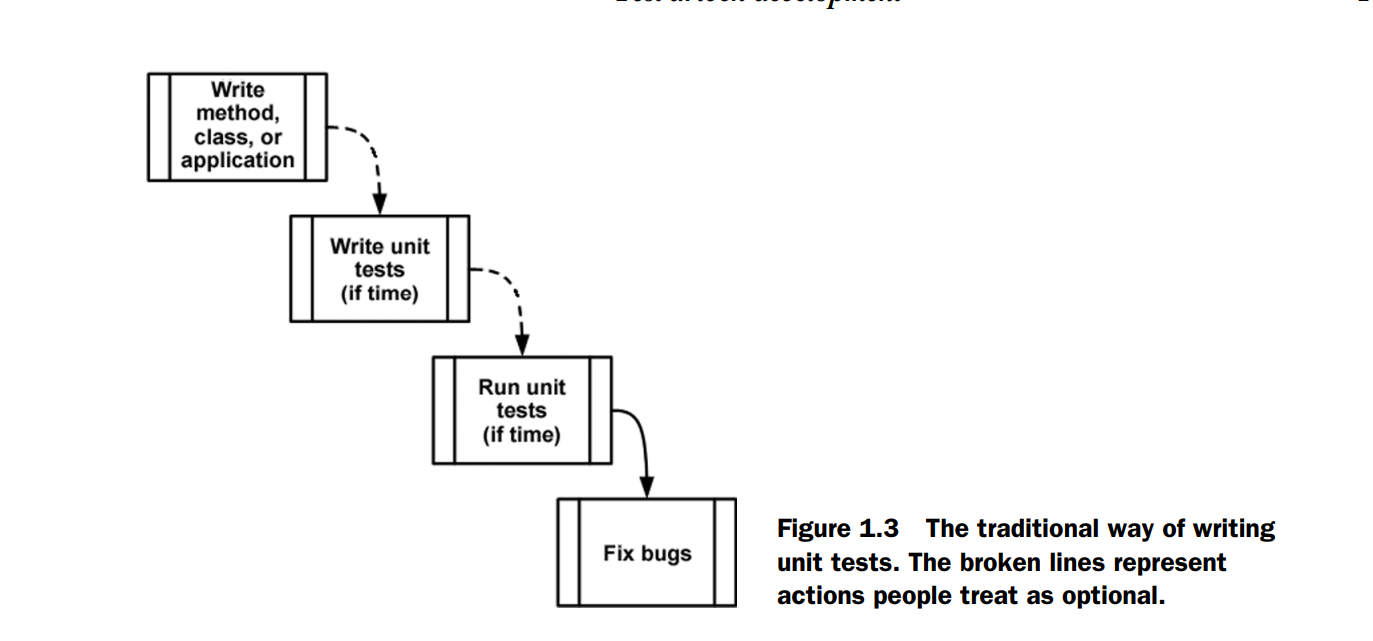
Định nghĩa Unit test:

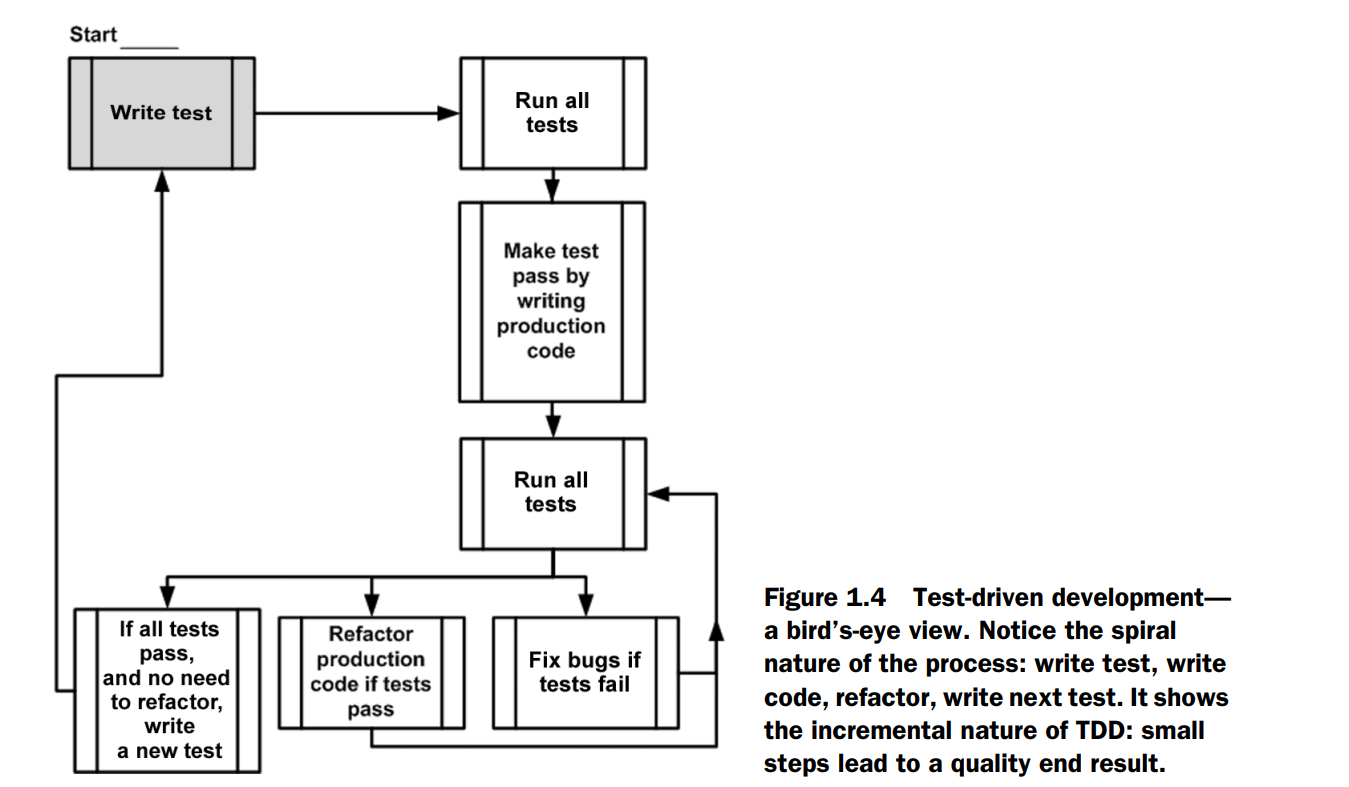
* Đoạn mã tự động gọi một đơn vị, sau đó kiểm tra một số giả định về kết quả cuối cùng của đơn vị đó.
* Hầu như được viết bằng unit test framework.
* Phải được viết dễ dàng và chạy nhanh chóng.
* Đáng tin cậy, dễ đọc và có thể bảo trì.
* Nhất quán, miễn là code không thay đổi.

Test-driven development

* Khi nào thì viết unit test? Nhiều người cảm thấy rằng thời điểm tốt nhất viết test là sau khi code được viết xong. Nhưng ngày càng nhiều người thích viết test trước khi viết code, cách tiếp cận này gọi là phát triển theo hướng thử nghiệm (test-first) hay gọi là TDD.
* Quy trình coding truyền thống:



Chú thích: đường nét đứng là tùy chọn.

* Quy trình coding theo TDD:
  + Viết một test fail
  + Chuyển sang viết code
  + Test đạt => tiếp tục refactor code hoặc tạo 1 test fail khác.
* 

Quy tắc đặt tên Func trong UT:

[UnitOfWorkName]\_[ScenarioUnderTest]\_[ExpectedBehavior]

UnitOfWorkName: Tên của phương thức hoặc nhóm phương thức hoặc các class để test.

ScenarioUnderTest: Các điều kiện mà đơn vị có thể kiểm tra.

ExpectedBehavior: Kết quả mong muốn là gì? Nên là 1 trong 3 khả năng: trả về 1 giá trị, thay đổi giá trị, gọi bên thứ 3.

NUnit

* Sử dụng các attr để nhận ra và load các bài kiểm tra.
* Nunit Runner cần ít nhất 2 attr để hiểu và chạy test:
  + [TestFixture] :
  + [Test]
* Các method phải để public
* 3 phần chính để test code:
  + Arrange: sắp xếp các đối tượng, tạo và thiết lập chúng khi cần thiết.
  + Act: hành động trên mỗi đối tượng.
  + Assert: khẳng định 1 cái gì đó như mong đợi

Lớp Assert

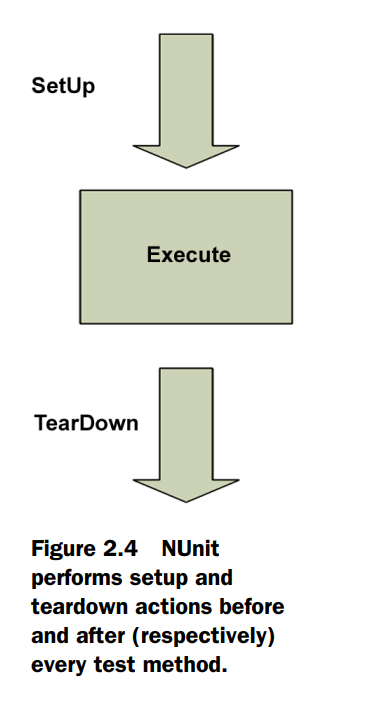
* Tuyên bố 1 giả định nào đó tồn tại.
* Nếu các đối số được truyền vào lớp Assert khác với giả định => NUnit sẽ biết là test không thành công và cảnh báo.
* “Red-Green-Refactor,” : làm fail test, sau đó pass nó, và làm code dễ đọc và dễ bảo trì hơn.

Test code styling:

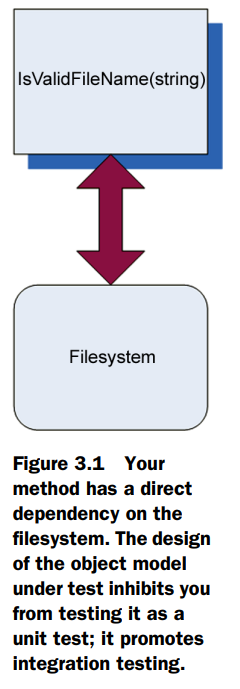
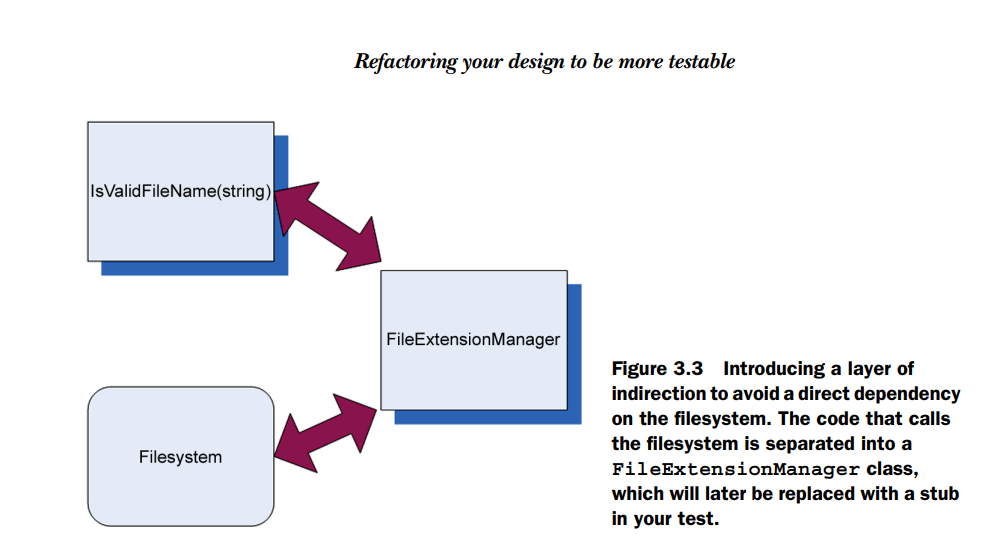
* Nên tách assert khỏi act càng nhiều càng tốt.

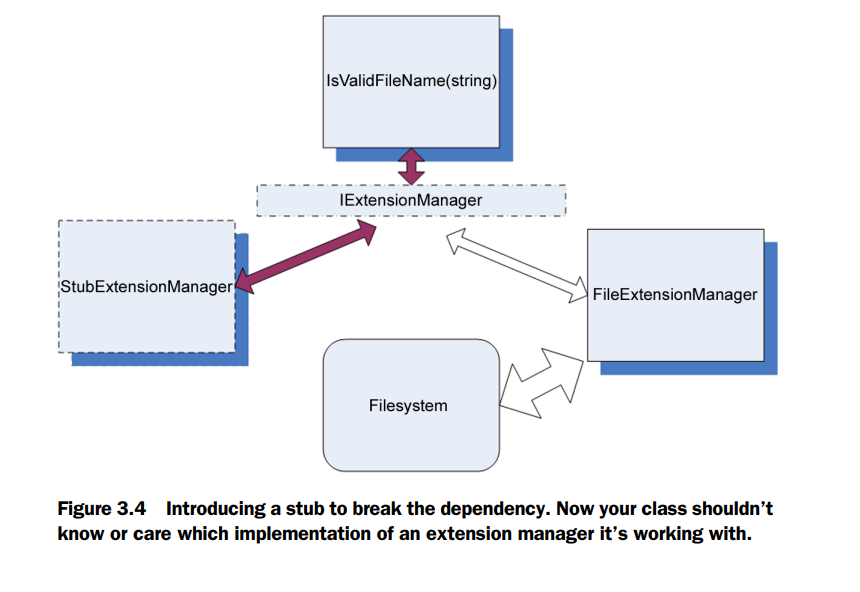
Refactoring to parameterized tests

Setup and teardown

* Với unit tests: điều quan trọng là mọi dữ liệu hoặc instance từ test trước còn xót lại phải được destroyed và test mới được tạo lại như thể chưa có test nào chạy trước đó.
* 
* [SetUp]— có thể đặt trên 1 phương thức, giống như Test. Và nó khiến NUnit chạy phương thức thiết lập đó khi chạy mỗi bài test trong class test.
* [TearDown]—
* Checking for expected exceptions:
  + Assert.Catch(delegate)
* Setting test categories
* system state change:
* rename
  + ByDefault: sử dụng khi không có 1 hành động nào làm thay đổi giá trị.
  + WhenCalled or Always

Core techniques

* Using stubs to break dependencies:
  + Sử dụng khi nào:
    - Trong trường hợp 1 đối tượng phụ thuộc vào 1 đối tượng khác mà không có quyền control.
    - Có thể là web service, time of day, threading,, filesystems, memory….
    - Mock cũng giống stub nhưng mock object cần assert lại còn stub thì ko.
  + Break dependency:
    - Tìm “interface” or API mà đối tượng thử nghiệm đang dựa vào.
    - Nếu I kết nối trực tiếp với đối tượng thử nghiệm => Thêm 1 mức chuyển hướng ẩn giao diện.
    - Thay thế implement với 1 thứ có thể kiểm soát.
    - Before:
      * 
    - After
      * 



* There are two types of dependency-breaking refactorings:
  + Type A—Abstracting concrete objects into interfaces or delegates
  + Type B—Refactoring to allow injection of fake implementations of those delegates or interfaces
  + Eg:
  + Type A—Extract an interface to allow replacing underlying implementation.
  + ■ Type B—Inject stub implementation into a class under test.
  + ■ Type B—Inject a fake at the constructor level. www.it-ebooks.info Refactoring your design to be more testable
  + ■ Type B—Inject a fake as a property get or set.
  + ■ Type B—Inject a fake just before a method call.
* a parameter object refactoring

word

* simulate: mô phỏng
* sake of: lợi ích