7 astuces pour améliorer vos tests unitaires

Pascal Laurin

Novembre 2015

Microsoft .NET MVP
Développeur & Architect chez GSoft

@plaurin78
pascal.laurin@outlook.com
www.pascallaurin.com
http://fr.slideshare.net/PascalLaurin
https://bitbucket.org/pascallaurin

GSoft

Agenda

1. Trouver les bons noms
2. Avoir des messages d’erreur clairs
3. Être facile à comprendre
4. Être rapide
5. Être indépendant
6. Tester une seule chose
7. Éviter les valeurs hard-coder

GSoft

Principes des bons tests unitaires

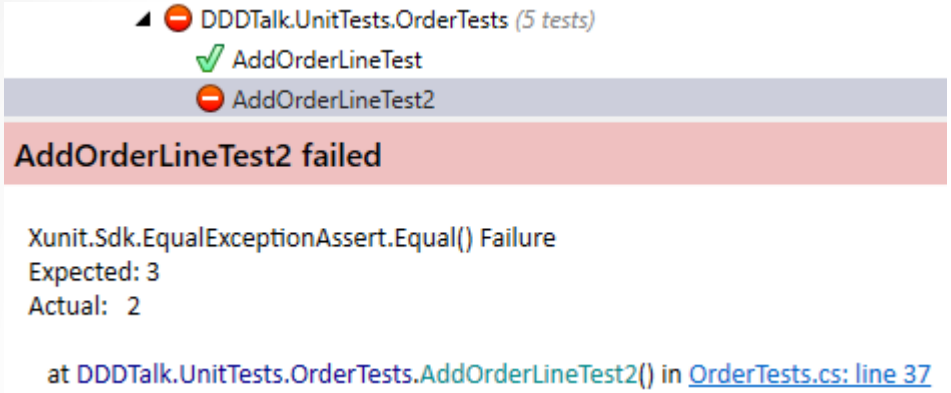
- Facile à comprendre
  - Facile à lire
  - On doit comprendre immédiatement ce qui se passe s'il échoue
- Indépendant
  - Les tests doivent pouvoir s'exécuter dans n'importe quel ordre
  - Ne doit pas avoir de dépendances externes
- Rapide
  - Exécution en moins de 0.01 secondes (ou 100 tests/seconde)
- Doit tester une seule chose
  - Sinon il y a plusieurs raisons d'échouer et le problème va être plus difficile à diagnostiquer

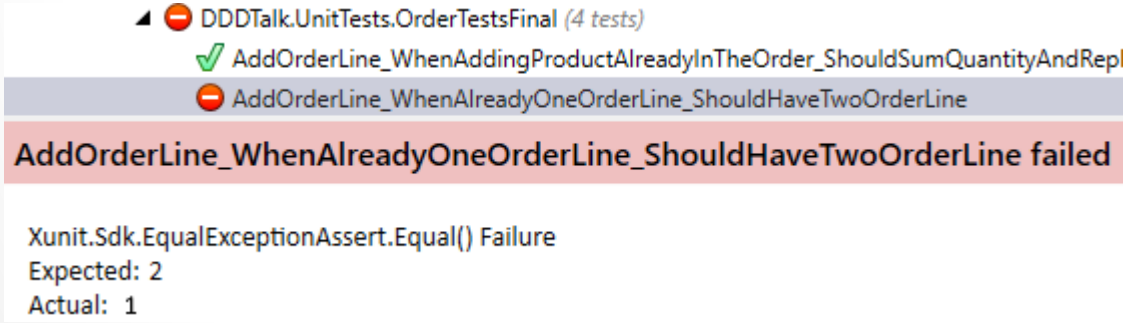
GSoft

1. Trouver les bons noms

- Nom de la classe de test
  - Comprendre sur quoi le test s'applique
  - Recherche rapide
- Nom de la méthode de test
  - Doit se lire comme le résumé du test
  - Surtout en cas d'échec du test
- Dans le code du test
  - Nom de variables significatives
  - Utilisation du langage du domaine d'affaire
- Objectif
  - Documentation du système

1. Trouver les bons noms



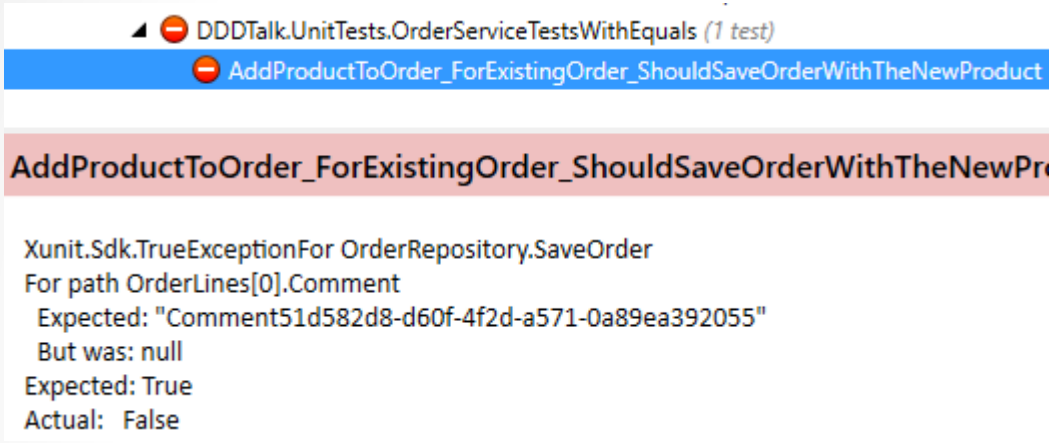


GSoft

2. Avoir des messages d'erreur clairs

- En cas d'échec le message doit être clair
  - Pour diagnostiquer le problème rapidement
  - Ajouter des traces au besoin
  - Toujours fournir une explication sur les Assert
  - Utiliser une libraires spécialiser (ie Fluent Assertions)
- Comparaison entre objets attendus et objets actuels
  - En implémentant ToString() (et/ou Equals() et GetHashCode())
  - Sérialisation Json pour comparer et afficher l'état des objets
- Objectif
  - Trouver la source du problème le plus rapidement possible

2. Avoir des messages d'erreur clairs



3. Être facile à comprendre

- Code du test
  - Structure avec Arrange, Act et Assert
- Utilisation de méthodes utilitaires
  - Méthodes de création, arrange et assert pour faciliter la lecture
  - Utilisation de classes utilitaires et classes de base pour la réutilisation des méthodes utilitaires
- Cacher ce qui n'est pas pertinent aux tests
  - Setup répétitif, plomberie d'architecture, les mocks, les valeurs littérales qui ne sont pas importantes, etc...
- Objectif
  - Garder le test pertinent longtemps (écrit une fois, lu plusieurs fois)

3. Être facile à comprendre

```
[Fact]
public void AddProductToOrder_ForExistingOrder_ShouldSaveOrderWithTheNewProduct()
{
    const int OrderId = 12;
    const int NumberOfLine = 2;

    // Arrange
    var existingOrder = CreateSomeOrder(OrderId, NumberOfLine);
    this.ArrangeOrderRepositoryLoadOrderReturns(existingOrder);

    // Act
    this.orderService.AddProductToOrder(OrderId, productId: 34, quantity: 56);

    // Assert
    this.AssertOrderRepositorySaveOrderWith(expectedCount: NumberOfLine + 1,
        expectedNewProductId: 34, expectedNewQuantity: 56);
}

private static Order CreateSomeOrder(int orderId, int numberOfLine) {...}
private void ArrangeOrderRepositoryLoadOrderReturns(Order existingOrder) {...}
private void AssertOrderRepositorySaveOrderWith(int expectedCount,
    int expectedNewProductId, int expectedNewQuantity) {...}
```

4. Être rapide

- Aucun appel externe
  - Utilisation d'Interfaces pour fournir des « tests doubles » dans les test unitaires
  - Utiliser le principe d'Inversion de Contrôle (IoC) et Injection de Dépendances (DI)
  - Utiliser des Mocks (et mocking framework) pour se faciliter la vie
- Objectif
  - Rouler les tests unitaires après chaque modification au système

4. Être rapide

GSoft

```
public class OrderService
{
    private readonly IOrderRepository orderRepository;
    public OrderService(IOrderRepository orderRepository)
    {
        this.orderRepository = orderRepository;
    }
    public void AddProductToOrder(int orderId, int productId, int quantity)
    {
        var order = this.orderRepository.LoadOrder(orderId);

        order.AddOrderLine(productId, quantity);

        this.orderRepository.SaveOrder(order);
    }
}

public OrderServiceTests()
{
    this.fakeOrderRepository = A.Fake<IOrderRepository>();
    this.orderService = new OrderService(this.fakeOrderRepository);
}
```

GSoft

5. Être indépendant

- Ne pas être basé sur l'état des tests précédents
  - Chaque test est responsable de l’état initial du système sous test
  - Éviter les bases de données et les services web externes
- Objectif
  - Pouvoir exécuter un test seul ou tous les tests sans manipulation ou initialisation quelconque: éviter les faux positifs.

5. Être indépendant

```
// Arrange
var existingOrder = CreateSomeOrder(OrderId, NumberOfLine);

...

private static Order CreateSomeOrder(int orderId, int numberOfLine)
{
    var existingOrder = new Order { Id = orderId };

    for (var i = 0; i < numberOfLine; i++)
    {
        existingOrder.AddOrderLine(productId: 123456 + i, quantity: 999);
    }

    return existingOrder;
}
```

GSoft

6. Tester une seule chose

- Séparer les tests d'intégrations/systèmes
  - Tests unitaires pour le domaine d'affaire
  - Tests d'intégration pour les adaptateurs vers les systèmes externes
- Séparer les tests unitaires
  - Créer deux ou plusieurs tests à partir d'un test qui en fait trop
  - Extraire les tests d'intégrations
- Objectif
  - En cas d'échec il n'y a qu'une seule cause: diagnostique rapide

GSoft

7. Éviter les valeurs hard-coder

- Utilisation de librairie de génération de valeurs aléatoire
  - i.e. AutoFixture
  - Ne garder que les valeurs pertinentes au test
- Objectif
  - Identifier clairement les valeurs qui vont influencer le test : aide la compréhension

7. Éviter les valeurs hard-coder

```
[Fact] // Avant
public void AddProductToOrder_ForExistingOrder_ShouldSaveOrderWithTheNewProduct()
{
    const int OrderId = 12;
    const int NumberOfLine = 2;

    // Arrange
    var existingOrder = CreateSomeOrder(OrderId, NumberOfLine);
    this.ArrangeOrderRepositoryLoadOrderReturns(existingOrder);

    // Act
    this.orderService.AddProductToOrder(OrderId, productId: 34, quantity: 56);

    // Assert
    this.AssertOrderRepositorySaveOrderWith(expectedCount: NumberOfLine + 1,
        expectedNewProductId: 34, expectedNewQuantity: 56);
}
```

7. Éviter les valeurs hard-coder

```
[Fact] // Après
public void AddProductToOrder_ForExistingOrder_ShouldSaveOrderWithTheNewProduct()
{
    var orderId = this.fixture.Create<int>();
    var productId = this.fixture.Create<int>();
    var quantity = this.fixture.Create<int>();
    const int NumberOfLine = 2;

    // Arrange
    var existingOrder = this.CreateSomeOrder(orderId, NumberOfLine);
    this.ArrangeOrderRepositoryLoadOrderReturns(existingOrder);

    // Act
    this.orderService.AddProductToOrder(orderId, productId: productId,
        quantity: quantity);

    // Assert
    this.AssertOrderRepositorySaveOrderWith(expectedCount: NumberOfLine + 1,
        expectedNewProductId: productId, expectedNewQuantity: quantity);
}
```

GSoft

Questions?

- Références
  - BitBucket pour le code
    - http://bit.ly/1ITLwca
  - FakeItEasy
    - http://fakeiteasy.github.io/
  - AutoFixture
    - https://github.com/AutoFixture/AutoFixture

@plaurin78
pascal.laurin@outlook.com
www.pascallaurin.com
http://fr.slideshare.net/PascalLaurin
https://bitbucket.org/pascallaurin