TP3 - Doublures de test et isolation

Objectifs

- Ecrire des **doublures manuelles** (fake, spy, stub) pour isoler un service
- Comprendre pourquoi les doublures manuelles sont **preferables** aux frameworks de mocking
- Constater les **problemes** des mocks (couplage a l'implementation, fragilite au refactoring)
- **Refactorer du code couple** pour le rendre testable
- Utiliser le pattern **Test Data Builder** pour simplifier le setup
- Partager le setup avec **IClassFixture**

Mise en place

Clonez le projet de démarrage et installez les dépendances :

```
git clone https://gitlab.com/edouard.mangel/tests-validation-tp3.git
cd clean-code-tp3
dotnet restore
dotnet test
```

Vous verrez des tests échouer — c'est votre travail de les faire passer au vert.

Exercice 1 - Tester NotificationService avec des doublures (55 min)

Contexte

Ouvrez `TestingTP3/` et parcourez les fichiers du projet de production :

- **`Interfaces.cs`** — quatre interfaces : `IUserRepository`, `IEmailSender`, `ISmsSender`, `IClock`. Chacune représente une dépendance externe que le service recevra par injection.
- **`User.cs`** — le modèle : `Id`, `Name`, `Email`, `Phone`, `HasOptedOut`.
- **`NotificationService.cs`** — le service à tester. Repérez ses trois méthodes publiques (`NotifyUser`, `SendUrgentNotification`, `NotifyAllUsers`) et notez comment chacune gère l'opt-out.

Vous ne touchez pas à ces fichiers. Votre travail se passe entièrement dans `TestingTP3.Tests/`.

Etape 1 - Explorer le FakeUserRepository (10 min)

Ouvrez `TestDoubles/FakeUserRepository.cs`.

Un **fake** est une implémentation simplifiée mais fonctionnelle. Remarquez :

- Il stocke les utilisateurs dans une `List<User>` en mémoire — pas de vraie base de données.
- La méthode `Add(User)` n'existe pas dans `IUserRepository` : c'est une **extension du fake** qui permet au test de peupler les données sans passer par l'interface du domaine.
- `GetAll()` retourne `_users.ToList()` — une copie, pour éviter que le test modifie la liste interne.

Ce fake remplace une vraie base de données : rapide, isolé, entièrement contrôlable.

Etape 2 - Explorer les Spy et le premier test (10 min)

Ouvrez `TestDoubles/SpyEmailSender.cs`.

Un **spy** enregistre les appels pour qu'on puisse vérifier ce qui s'est passé. Remarquez que `SentEmails` est une liste de tuples nommés `(To, Subject, Body)` — cela permet des assertions précises sur chaque champ sans avoir à créer une classe dédiée.

Ouvrez aussi `TestDoubles/SpySmsSender.cs` et `TestDoubles/StubClock.cs`. Le **stub** d'horloge retourne toujours la même date : c'est ce qui rend les assertions sur les timestamps déterministes.

Ouvrez maintenant `Exercice1/NotificationServiceTests.cs`. Le premier test `NotifyUser_ExistingUser_SendsEmail` est déjà écrit et compilé. Lisez-le attentivement :

- Comment les doublures sont câblées ensemble via le constructeur de `NotificationService`.
- Pourquoi l'assertion porte sur `Body` et non sur `Subject`.

Lancez `dotnet test` — ce test doit passer au vert.

Etape 3 - Tester les autres cas (10 min)

Les quatre tests suivants sont dans `Exercice1/NotificationServiceTests.cs`, squelettisés avec un `// Arrange / Act / Assert` et un `throw NotImplementedException`. Complétez le corps de chaque test :

- `NotifyUser_UserNotFound_DoesNotSendEmail` — repo vide, NotifyUser(999, ...), vérifiez que `SentEmails` est vide.
- `NotifyUser_UserOptedOut_DoesNotSendEmail` — utilisateur avec `HasOptedOut = true`, vérifiez l'absence d'envoi.
- `SendUrgentNotification_ExistingUser_SendsEmailAndSms` — vérifiez que les deux spies reçoivent un message.
- `SendUrgentNotification_UserOptedOut_SendsAnyway` — les urgentes ignorent l'opt-out.

Etape 4 - Tester NotifyAllUsers avec le spy (10 min)

Le test `NotifyAllUsers_SendsEmailOnlyToActiveUsers` est aussi dans `NotificationServiceTests.cs`.

Peuplez le `FakeUserRepository` avec 3 utilisateurs : 2 actifs et 1 avec opt-out. Appelez `NotifyAllUsers`, puis vérifiez avec le spy que `SentEmails` contient exactement 2 emails.

Remarquez comme le spy rend l'assertion naturelle : on inspecte les emails **réellement capturés** et on vérifie leur contenu.

Etape 5 - Le meme test avec NSubstitute : comparer les approches (15 min)

Ouvrez `Exercice1/NotificationServiceNSubTests.cs`. Les deux méthodes squelettisées y sont prêtes.

Voici les opérations clés NSubstitute :

Operation	Syntaxe NSubstitute
Creer une doublure	`Substitute.For<IEmailSender>()`
Configurer un retour	`repo.GetById(1).Returns(new User { ... })`
Verifier un appel	`emailSender.Received(1).Send("alice@test.com", ...)`
Verifier l'absence d'appel	`emailSender.DidNotReceive().Send(...)`
Accepter n'importe quel argument	`Arg.Any<string>()`

Implémentez les deux tests :

1. `NotifyUser_ExistingUser_SendsEmail` — verifiez avec `.Received(1).Send(...)` au lieu du spy
2. `NotifyAllUsers_SendsEmailOnlyToActiveUsers` — configurez `GetAll().Returns(...)`, puis verifiez avec `.Received()` et `.DidNotReceive()` pour chaque utilisateur

Les deux versions (spy et NSubstitute) doivent etre au vert avant de continuer.

5a - Refactoring : changer le format du sujet

Un collegue modifie le format du sujet des emails. Dans `NotificationService.NotifyUser`, changez :

```
// Avant
$"Notification du {_clock.Now:dd/MM/yyyy}"

// Apres
$"[{_clock.Now:dd/MM/yyyy}] Notification"
```

Lancez **tous** les tests. Constatez :

- Quels tests de la version **spy** cassent ? Pourquoi ?
- Quels tests de la version **NSubstitute** cassent ? Pourquoi ?

Le spy capture des **données** qu'on peut inspecter librement — vos assertions portaient sur le **body**, pas sur le subject. `.Received()` verifie des **appels exacts** — si vous avez verifie le sujet, le test casse pour un changement cosmétique qui ne change pas le comportement.

Corrigez les tests NSubstitute qui ont casse, puis gardez le nouveau format — c'est un refactoring legitime.

5b - Evolution : ajouter une formule de politesse

Le product owner demande que les emails de notification soient plus professionnels. Modifiez `NotifyUser` pour formater le body avec une salutation :

```
var body = $"Bonjour {user.Name},\n\nCordialement,\nL'equipe";
_emailSender.Send(user.Email, subject, body);
```

C'est une evolution raisonnable. Lancez **tous** les tests (spy et NSubstitute).

Un des deux suites detecte un probleme. Lequel, et pourquoi l'autre ne le detecte pas ?

Corrigez le body pour inclure le `message` dans la formule de politesse, puis repassez tous les tests au vert.

Exercice 2 - Refactorer du code couple (40 min)

Contexte

Ouvrez `TestingTP3/ReportGenerator.cs`.

Le code original — mis en commentaire dans le fichier — ressemble à ce qu'écrit un collègue pressé : `SqlConnection` créée en interne, `DateTime.Now` appelé

directement, `SmtpClient` instancié sur place. Tout est câblé en dur. Il est **impossible a tester** car toutes les dependances sont creees en interne.

Votre mission : refactorer `GenerateAndSend` pour qu'il reçoive ses dépendances par injection.

Etape 1 - Identifier les dependances (5 min)

Lisez les commentaires dans `ReportGenerator.cs` et répondez à ces questions :

1. Combien de dependances externes ce code a-t-il ?
2. Lesquelles empechent les tests unitaires ?
3. Quels sont les risques de ce code en production ? (indice : regardez la requete SQL)

Etape 2 - Extraire les interfaces et refactorer (10 min)

Les interfaces sont déjà déclarées dans `TestingTP3/Interfaces.cs` : `IReportRepository`, `IReportSender` et `IClock` (partagée avec l'exercice 1). `ReportData` est dans `ReportData.cs`.

Complétez `ReportGenerator.cs` :

- Ajoutez les champs privés et un constructeur qui reçoit `IReportRepository`, `IReportSender` et `IClock`
- Implémentez `GenerateAndSend` :
  - Appeler `_repository.GetByName(reportName)` pour obtenir les donnees
  - Utiliser `_clock.Now` pour l'horodatage
  - Appeler `_sender.Send(recipientEmail, reportName, body)` pour envoyer le rapport

Etape 3 - Ecrire les tests avec des doublures manuelles (15 min)

Les doublures sont déjà dans `TestDoubles/` :

- **`FakeReportRepository`** — stocke les rapports dans un `Dictionary`. Remarquez qu'il lève une `InvalidOperationException` si le rapport est introuvable : c'est un comportement fonctionnel, pas juste un retour vide.

- `SpyReportSender` — enregistre les rapports envoyés dans une liste de tuples `(To, Subject, Body)`.
- `StubClock` — déjà utilisé en exercice 1, réutilisez-le ici.

Ouvrez `Exercice2/ReportGeneratorTests.cs`. Les trois tests sont squelettisés. Complétez le corps de chacun en utilisant ces doublures :

- `GenerateAndSend_ValidReport_SendsEmailToRecipient` — vérifiez que le rapport est envoyé au bon destinataire
- `GenerateAndSend_ValidReport_IncludesTimestampInBody` — vérifiez que le corps du mail contient la date formatée
- `GenerateAndSend_ValidReport_IncludesRowCountInBody` — vérifiez que le corps du mail contient le nombre de lignes

Etape 4 - Tester le cas d'erreur (10 min)

Objectif : tester ce qui se passe quand le repository leve une exception.

Le quatrième test `GenerateAndSend_UnknownReport_ThrowsException` est dans `Exercice2/ReportGeneratorTests.cs`. Utilisez un `FakeReportRepository` vide (sans rapport ajouté) et vérifiez avec `Assert.Throws<InvalidOperationException>(...)`.

Aucun besoin de framework de mocking : le fake gere naturellement le cas d'erreur.

Exercice 3 - Test Data Builders (15 min)

Contexte

Regardez les blocs `Arrange` de vos tests des exercices 1 et 2. Vous allez remarquer beaucoup de repetition dans la construction des objets `User` et `ReportData`.

Le pattern **Test Data Builder** encapsule cette construction avec des valeurs par defaut et une API fluente.

Etape 1 - Observer le pattern Builder (5 min)

Ouvrez `Builders/UserBuilder.cs`. Remarquez :

- Chaque champ a une valeur par défaut sensée (`_id = 1`, `_name = "Alice"`, etc.).
- Les méthodes fluentes retournent `this` — ce qui permet le chaînage.
- `AsOptedOut()` est sémantique : elle exprime l'intention du test, pas un détail d'implémentation.

Comparez ces deux blocs Arrange :

```
// Sans builder
var user = new User { Id = 1, Name = "Alice", Email = "alice@test.com",
                      Phone = "0600000001", HasOptedOut = false };

// Avec builder — seul ce qui est pertinent pour le test est visible
var user = new UserBuilder().AsOptedOut().Build();
```

Le builder met en avant **ce qui compte** pour le test et masque le bruit.

Etape 2 - Implementer un ReportDataBuilder (5 min)

Ouvrez `Builders/ReportDataBuilder.cs`. Le squelette est là, les `TODO` indiquent ce qu'il faut compléter.

L'usage cible :

```
var report = new ReportDataBuilder().WithName("Ventes Q1").WithRowCount(15
```

Implémentez les valeurs par défaut, les méthodes fluentes `WithName` et `WithRowCount`, et la méthode `Build`.

Etape 3 - Refactorer vos tests existants (5 min)

Choisissez 3 tests de l'exercice 1 ou 2 et remplacez la construction manuelle des objets par les builders.

Verifiez que tous les tests passent toujours.

Exercice 4 - IClassFixture : partager le setup (15 min)

Contexte

Dans vos tests de l'exercice 1, chaque methode de test recree les memes doublures et le meme service. Si on ajoute une dependance au `NotificationService`, il faudra modifier **chaque test**.

Etape 1 - Identifier le setup repete (3 min)

Regardez vos tests de l'exercice 1. Quel code est duplique dans chaque methode de test ?

Listez les lignes qui apparaissent dans au moins 3 tests.

Etape 2 - Utiliser la fixture partagee (7 min)

Ouvrez `Fixtures/NotificationTestFixture.cs`. La fixture est déjà écrite : elle instancie toutes les doublures et le service dans son constructeur, et expose chaque doublure via une propriété publique.

Modifiez `Exercice1/NotificationServiceTests.cs` pour utiliser `IClassFixture<NotificationTestFixture>` :

```
public class NotificationServiceTests : IClassFixture<NotificationTestFixt
{
    private readonly NotificationTestFixture _fixture;

    public NotificationServiceTests(NotificationTestFixture fixture)
    {
        _fixture = fixture;
    }

    [Fact]
    public void NotifyUser_ExistingUser_SendsEmail()
    {
        // Arrange — utiliser _fixture.UserRepo, _fixture.EmailSpy, etc.
        // ...
```

```
    }
}
```

Attention : avec `IClassFixture`, la fixture est partagee entre tous les tests de la classe. Vos tests doivent-ils etre adaptes pour rester isoles ? Reflechissez-y.

Etape 3 - Verifier que tout fonctionne (5 min)

Lancez tous vos tests. Ils doivent tous passer au vert.

Si certains tests echouent, c'est probablement un probleme d'isolation : un test modifie l'etat de la fixture et affecte un autre test. Identifiez le probleme et corrigez-le.

Recapitulatif

Concept	Ou l'avez-vous pratique ?
Fake (implementation simplifiee)	Exercice 1 etape 1, Exercice 2 etape 3
Spy (enregistre les appels)	Exercice 1 etapes 2-4, Exercice 2 etape 3
Stub (retourne des valeurs fixes)	Exercice 1 etape 2, Exercice 2 etape 3
NSubstitute et ses limites	Exercice 1, etape 5a (refactoring qui casse les mocks)
`Arg.Any<>()` masque les regressions	Exercice 1, etape 5b (evolution qui revele la faiblesse)
Refactoring pour testabilite	Exercice 2, etapes 1-2
Tester un cas d'erreur (exception)	Exercice 2, etape 4
Test Data Builder	Exercice 3
`IClassFixture<T>`	Exercice 4