TP3 - Doublures de test et isolation (TypeScript)

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 `beforeAll` / `beforeEach`

Mise en place

Clonez le projet de demarrage et installez les dependances :

```
git clone https://gitlab.com/edouard.mangel/tests-validation-tp3-ts.git
cd tests-validation-tp3-ts
npm install
npm test
```

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

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

Contexte

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

- **`interfaces.ts`** — quatre interfaces : `UserRepository`, `EmailSender`, `SmsSender`, `Clock`. Chacune represente une dependance externe que le service recevra par injection.
- **`User.ts`** — le modele : `id`, `name`, `email`, `phone`, `hasOptedOut`.
- **`NotificationService.ts`** — le service a tester. Repérez ses trois methodes publiques (`notifyUser`, `sendUrgentNotification`, `notifyAllUsers`) et notez comment chacune gere l'opt-out.

Vous ne touchez pas a ces fichiers. Votre travail se passe entierement dans `tests/`.

Voici les interfaces et le modele pour reference :

```
// src/interfaces.ts
export interface UserRepository {
    getById(id: number): User | null;
    getAll(): User[];
}

export interface EmailSender {
    send(to: string, subject: string, body: string): void;
}

export interface SmsSender {
    send(to: string, message: string): void;
}

export interface Clock {
    now(): Date;
}
```

```
// src/User.ts
export interface User {
    id: number;
    name: string;
    email: string;
    phone: string;
    hasOptedOut: boolean;
}
```

Etape 1 - Explorer le FakeUserRepository (10 min)

Ouvrez `tests/doubles/FakeUserRepository.ts`.

Un **fake** est une implementation simplifiee mais fonctionnelle. Remarquez :

- Il stocke les utilisateurs dans un tableau `User[]` en memoire — pas de vraie base de donnees.
- La methode `add(user: User)` n'existe pas dans `UserRepository` : c'est une **extension du fake** qui permet au test de peupler les donnees sans passer par l'interface du domaine.
- `getAll()` retourne `[...this.users]` — une copie, pour eviter que le test modifie le tableau interne.

Ce fake remplace une vraie base de donnees : rapide, isole, entierement controlable.

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

Ouvrez `tests/doubles/SpyEmailSender.ts`.

Un **spy** enregistre les appels pour qu'on puisse verifier ce qui s'est passe. Remarquez que `sentEmails` est un tableau d'objets `{ to, subject, body }` — cela permet des assertions precises sur chaque champ.

Ouvrez aussi `tests/doubles/SpySmsSender.ts` et `tests/doubles/StubClock.ts`. Le **stub** d'horloge retourne toujours la meme date : c'est ce qui rend les assertions sur les timestamps deterministes.

Ouvrez maintenant `tests/exercice1/NotificationService.test.ts`. Le premier test `notifyUser - utilisateur existant - envoie un email` est deja ecrit et compile. Lisez-le attentivement :

- Comment les doublures sont cablees ensemble via le constructeur de `NotificationService`.
- Pourquoi l'assertion porte sur `body` et non sur `subject`.

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

Etape 3 - Tester les autres cas (10 min)

Les quatre tests suivants sont dans `tests/exercice1/NotificationService.test.ts`, squelettises avec un commentaire `// TODO`. Completez le corps de chaque test :

- `notifyUser - utilisateur introuvable - n'envoie pas d'email` — repo vide, `notifyUser(999, ...)`, verifiez que `sentEmails` est vide.
- `notifyUser - utilisateur a opted out - n'envoie pas d'email` — utilisateur avec `hasOptedOut: true`, verifiez l'absence d'envoi.
- `sendUrgentNotification - utilisateur existant - envoie email et SMS` — verifiez que les deux spies recoivent un message.
- `sendUrgentNotification - utilisateur opted out - envoie quand meme` — les urgentes ignorent l'opt-out.

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

Le test `notifyAllUsers - envoie seulement aux utilisateurs actifs` est aussi dans le fichier.

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

Remarquez comme le spy rend l'assertion naturelle : on inspecte les emails **reellement captures** et on verifie leur contenu.

Etape 5 - Le meme test avec vi.fn() : comparer les approches (15 min)

Ouvrez `tests/exercice1/NotificationServiceViMock.test.ts`. Les deux methodes squelettisees y sont pretes.

Voici les operations cles avec `vi.fn()` de Vitest :

Operation	Syntaxe
Creer une doublure	`const emailSender = { send: vi.fn() }`
Configurer un retour	`mockUserRepo.getById.mockReturnValue({ ... })`
Verifier un appel	`expect(emailSender.send).toHaveBeenCalledWith('alice@test.com', ...)`
Verifier l'absence	`expect(emailSender.send).not.toHaveBeenCalled()`

d'appel	
Accepter n'importe quel argument	`expect.any(String)`

Implementez les deux tests :

```
import { vi, expect, it, describe } from 'vitest';

describe('NotificationService avec vi.fn()', () => {
    it('notifyUser - utilisateur existant - envoie un email', () => {
        // Arrange
        const mockUserRepo = { getById: vi.fn(), getAll: vi.fn() };
        const mockEmailSender = { send: vi.fn() };
        const mockSmsSender = { send: vi.fn() };
        const mockClock = { now: vi.fn().mockReturnValue(new Date('2024-01-15

        mockUserRepo.getById.mockReturnValue({
            id: 1, name: 'Alice', email: 'alice@test.com',
            phone: '0600000001', hasOptedOut: false
        });

        const service = new NotificationService(
            mockUserRepo, mockEmailSender, mockSmsSender, mockClock
        );

        // Act
        service.notifyUser(1, 'Test message');

        // Assert
        expect(mockEmailSender.send).toHaveBeenCalledWith(
            'alice@test.com',
            expect.any(String),
            expect.any(String)
        );
    });

    it('notifyAllUsers - envoie seulement aux utilisateurs actifs', () => {
        // TODO : implementez ce test
    });
});
```

Les deux versions (spy et vi.fn()) 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
subject: `Notification du ${this.clock.now().toLocaleDateString('fr-FR')}`

// Apres
subject: `[${this.clock.now().toLocaleDateString('fr-FR')}] Notification`
```

Lancez **tous** les tests. Constatez :

- Quels tests de la version **spy** cassent ? Pourquoi ?
- Quels tests de la version **vi.fn()** cassent ? Pourquoi ?

Le spy capture des **donnees** qu'on peut inspecter librement — vos assertions portaient sur le **body**, pas sur le subject. `expect.any(String)` masque toute regression sur le format du sujet.

Corrigez les tests vi.fn() 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 :

```
const body = `Bonjour ${user.name},\n\nCordialement,\nL'equipe`;
this.emailSender.send(user.email, subject, body);
```

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

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 `src/ReportGenerator.ts`.

Le code original — mis en commentaire dans le fichier — ressemble a ce qu'ecrit un collegue presse : connexion SQL creee en interne, `new Date()` appele directement, client SMTP instancie sur place. Tout est cable en dur. Il est **impossible a tester** car toutes les dependances sont creees en interne.

Votre mission : refactorer `generateAndSend` pour qu'il recoive ses dependances par injection.

Etape 1 - Identifier les dependances (5 min)

Lisez les commentaires dans `ReportGenerator.ts` et repondez a 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 et l'absence de validation)

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

Les interfaces sont deja declarees dans `src/interfaces.ts` : `ReportRepository`, `ReportSender` et `Clock` (partagee avec l'exercice 1). `ReportData` est dans `src/ReportData.ts`.

Completez `ReportGenerator.ts` :

- Ajoutez les champs prives et un constructeur qui recoit `ReportRepository`, `ReportSender` et `Clock`
- Implementez `generateAndSend` :
  - Appeler `this.repository.getByName(reportName)` pour obtenir les donnees
  - Utiliser `this.clock.now()` pour l'horodatage
  - Appeler `this.sender.send(recipientEmail, reportName, body)` pour envoyer le rapport

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

Les doublures sont deja dans `tests/doubles/` :

- **`FakeReportRepository`** — stocke les rapports dans un `Map<string, ReportData>`. Remarquez qu'il leve une `Error` si le rapport est introuvable : c'est un comportement fonctionnel, pas juste un retour vide.
- **`SpyReportSender`** — enregistre les rapports envoyes dans un tableau d'objets `{ to, subject, body }`.
- **`StubClock`** — deja utilise en exercice 1, reutilisez-le ici.

Ouvrez `tests/exercice2/ReportGenerator.test.ts`. Les trois tests sont squelettises. Completez le corps de chacun en utilisant ces doublures :

- `generateAndSend - rapport valide - envoie l'email au bon destinataire` — verifiez que le rapport est envoye au bon destinataire
- `generateAndSend - rapport valide - inclut l'horodatage dans le body` — verifiez que le corps du mail contient la date formatee
- `generateAndSend - rapport valide - inclut le nombre de lignes dans le body` — verifiez 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 quatrieme test `generateAndSend - rapport introuvable - leve une exception` est dans `tests/exercice2/ReportGenerator.test.ts`. Utilisez un `FakeReportRepository` vide (sans rapport ajoute) et verifiez avec `expect(() => ...).toThrow()`.

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 `tests/builders/UserBuilder.ts`. Remarquez :

- Chaque champ a une valeur par defaut sensee (`id = 1`, `name = 'Alice'`, etc.).
- Les methodes fluentes retournent `this` — ce qui permet le chainage.
- `asOptedOut()` est semantique : elle exprime l'intention du test, pas un detail d'implementation.

Comparez ces deux blocs Arrange :

```
// Sans builder
const user: 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
const 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 `tests/builders/ReportDataBuilder.ts`. Le squelette est la, les `// TODO` indiquent ce qu'il faut completer.

L'usage cible :

```
const report = new ReportDataBuilder().withName('Ventes Q1').withRowCount(150
```

Implementez les valeurs par defaut, les methodes fluentes `withName` et `withRowCount`, et la methode `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 - beforeAll / beforeEach : partager le setup (15 min)

Contexte

Dans vos tests de l'exercice 1, chaque 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 test ?

Listez les lignes qui apparaissent dans au moins 3 tests.

Etape 2 - Utiliser un setup partage (7 min)

Vitest propose deux hooks de setup :

- `beforeEach` — execute avant **chaque** test, recrée les objets a chaque fois
- `beforeAll` — execute une seule fois avant **tous** les tests du `describe`, partage l'instance

```
import { describe, it, expect, beforeEach } from 'vitest';

describe('NotificationService - setup partage', () => {
    let userRepo: FakeUserRepository;
    let emailSpy: SpyEmailSender;
    let smsSpy: SpySmsSender;
    let clock: StubClock;
    let service: NotificationService;

    beforeEach(() => {
        // Recrée les doublures avant chaque test → isolation garantie
        userRepo = new FakeUserRepository();
        emailSpy = new SpyEmailSender();
        smsSpy = new SpySmsSender();
        clock = new StubClock(new Date('2024-01-15'));
        service = new NotificationService(userRepo, emailSpy, smsSpy, clock);
    });
```

```
    it('notifyUser - utilisateur existant - envoie un email', () => {
        // Arrange — utiliser les variables du describe
        userRepo.add({ id: 1, name: 'Alice', email: 'alice@test.com',
                       phone: '0600000001', hasOptedOut: false });
        // Act + Assert...
    });
});
```

Refactorisez `tests/exercice1/NotificationService.test.ts` pour utiliser `beforeEach`.

Attention : avec `beforeAll`, le setup est partage entre tous les tests. 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 partage et affecte un autre test. Identifiez le probleme et corrigez-le.

Recapitulatif

Concept C# (.NET)	Equivalent TypeScript (Vitest)	Ou l'avez-vous pratique ?
Fake (implementation simplifiee)	Classe TypeScript implementant l'interface	Exercice 1 etape 1, Exercice 2 etape 3
Spy (enregistre les appels)	Tableau d'objets captures	Exercice 1 etapes 2-4, Exercice 2 etape 3
Stub (retourne des valeurs fixes)	Classe avec valeur fixe	Exercice 1 etape 2, Exercice 2 etape 3
NSubstitute	`vi.fn()` de Vitest	Exercice 1, etape 5a
`Arg.Any<string>()`	`expect.any(String)`	Exercice 1, etape 5b
Refactoring pour testabilite	Meme approche (injection de dependances)	Exercice 2, etapes 1-2

Tester un cas d'erreur	`expect(() => fn()).toThrow()`	Exercice 2, etape 4
Test Data Builder	Meme pattern, TypeScript fluent API	Exercice 3
`IClassFixture<T>`	`beforeAll` / `beforeEach`	Exercice 4