3_Deceler les éléments à tester

Supposons que nous sommes dans une entreprise qui doit mettre en place une super-calculatrice.

Mettre en place des tests sur un tel projet peut sembler futile. En effet, on peut rapidement vérifier si l'addition fait bel et bien une addition. Mais imaginez un projet bien plus gros. Des classes comme celle-ci peuvent se compter par centaines, et vérifier que chacune d'elle fait ce que l'on souhaite serait extrêmement long et fastidieux.

Pire encore, imaginez une fonctionnalité qui ne lève pas d'erreur, mais ne retourne pas ce que l'on souhaite pour autant. Ce bug pourrait passer entre les mailles du filet. Mais si mon code est correctement couvert par des tests, une erreur sera précisément **repérée** et donc **plus rapidement corrigée**.

Déterminez ce qu'il faut tester

La première question à se poser lorsque l'on doit ajouter des tests à un projet est :

> **Que souhaitons-nous tester ?**

Est-ce le résultat de notre programme ? Dans notre cas, un affichage sans exception. Ou est-ce l'intégralité du code, fonction par fonction ?

Première stratégie

Tester uniquement la dernière étape de notre programme n'est pas tout à fait une bonne idée car c'est **trop vaste**. Adoptons donc le point de vue inverse et créons des tests unitaires qui vont vérifier que chaque méthode

de notre code produit bien le résultat que nous souhaitons. Au moins nous serons sûrs que notre programme est **parfaitement** sécurisé !

Notre code contient 32 fonctions et variables de classes que nous pourrions tester. La première stratégie serait donc de **créer autant de tests que de fonctions**...

Mais faire cela est loin d'être idéal. Réfléchissons un instant.

> **Pourquoi voulons-nous écrire des tests ?**

Pour ajouter une fonctionnalité après, et être donc certains que cette dernière ne va pas avoir un effet pervers sur les anciennes.

Cela ne sert donc à rien de tester l'implémentation d'un programme, c'est-à-dire tout le détail. Pourquoi ? Car si je dois modifier mon code pour l'améliorer, cela va casser plusieurs tests alors même que mon programme fonctionne encore. Simplement car **j'aurai écrit plusieurs tests qui dépendent de mon implémentation**.

Rappels sur l'Objet

Afin de comprendre ce qu'il est préférable de tester, revenons un peu en arrière sur des notions de programmation orientée objet.

Chaque objet peut être vu comme une **navette spatiale en orbite**. Cette dernière n'a aucune idée de ce qui se passe dans les autres navettes. Elle sait uniquement ce qu'elle a besoin de savoir pour son bon fonctionnement à elle.

La communication avec les autres navettes se fait via des **messages entrants** et des **messages sortants**. Elle peut également envoyer des **messages internes** au personnel de la navette.

> **Les messages entrants sont gérés par les méthodes publiques d'un objet. Par exemple, un objet `Apollo` peut avoir une méthode publique `equipage` qui renvoie le nombre de personnes à bord. Les messages internes, eux, seront les méthodes privées d'un objet. Vous ne pouvez les utiliser qu'à l'intérieur d'une classe et non en dehors. Par exemple, `Apollo` peut avoir une méthode privée `reparer_moteur`.**

Tester une interface

Une bonne pratique consiste à tester uniquement les **messages entrants**, c'est-à-dire les méthodes publiques, et non les méthodes privées.

En effet, nous considérons qu'**un objet est une boîte noire** qui contient tout ce dont il a besoin pour son bon fonctionnement. Nous, externes, n'avons pas à connaître la manière dont il nous envoie les informations. Nous vérifions simplement que ces informations correspondent à celles que nous attendons.

Ceci veut dire que nous allons tester **l'interface** publique d'un objet et non son fonctionnement interne. Offrant ainsi une plus grande **flexibilité** à la fois dans nos tests et dans la configuration de nos objets.

Nous pouvons alors facilement changer le code d'un objet sans avoir à modifier nos tests. Par exemple, si un jour nous changeons la manière de calculer les données, mais que le résultat renvoyé est toujours conforme à nos attentes, le test sera toujours valide et nous n'aurons pas à le réécrire.

Prévoir un plan de test

Nous avons pu constater qu'il n'est pas très facile de déceler l'ensemble des scénarios qu'il faut tester. C'est pourquoi il est intéressant de **prévoir un**

plan de test. Il permet d'identifier les éléments et les fonctionnalités à tester. De plus, il permet de mieux organiser et planifier l'implémentation des tests. Le gain de temps est donc bel et bien considérable avec un plan de test correctement spécifié.

Le plan de test peut être plus ou moins détaillé, mais il y a des points importants à spécifier pour chaque cas de test :

- le **type** de test (unitaire, d'intégration, fonctionnel…) ;
- l'**unité** de code à tester ;
- les **inputs** à tester ;
- les **outputs** attendus.

Prenons un exemple concret et essayons maintenant de déceler les différents **scénarios à tester**. Voici ci-dessous une classe qui permet d'effectuer des opérations sur deux valeurs (addition, soustraction, multiplication et division).

```
import re

class Calculator:
    def __init__(self):
        self.left_value = 0
        self.right_value = 0

    def calculate(self, operation):
        if self._check_and_set_value(operation):
            if "+" in operation:
                return self.left_value + self.right_value
            elif "-" in operation:
                return self.left_value - self.right_value
            elif "*" in operation:
                return self.left_value * self.right_value
            elif "/" in operation:
                try:
```

```
                    return self.left_value / self.right_value
                except ZeroDivisionError:
                    return "Invalid operation : Zero Division
Error"
            else:
                return "Invalid operation"
        else:
            return "Invalid operation"

    def _check_and_set_value(self, operation):
        operation = operation.replace(" ", "")
        values = re.split('\+|\-|\*|\/', operation)
        if len(values) == 2:
            try:
                self.left_value = float(values[0])
                self.right_value = float(values[1])
                return True
            except ValueError:
                return False
        else:
            return False
```

Nous devons tout d'abord identifier **les méthodes publiques** de notre classe `Calculator`.

D'une part, nous devons tester **les messages entrants** et, d'autre part, nous devons vérifier **les messages sortants** de notre boîte noire. Ainsi nous allons tester l'ensemble des méthodes publiques de la classe `Calculator`.

Néanmoins, il est important de noter que nous n'allons pas créer un test par méthode publique, mais autant de tests que de **possibilités de sortie** que fournira la méthode publique. Il sera donc normal d'avoir plusieurs tests pour une même méthode publique.

On va premièrement vérifier les raisons pour lesquelles *check_and_set_value* va retourner False

- Lorsque len(values) != 2 :
  - si on a un symbole différent de '+|-|*|/' : **Invalid operator**
  - Lorsqu'il y a une double opération. Exemple 2+2+2 : **More than one operation**
- Left values not a float
- Right value not a float

Ensuite on va vérifier **calculate**

- Si *check_and_set_value* retourne True :
  operator +
  operator *
  operator -
  operator /
  * operator / with Zero Division
  Pour notre méthode publique on a 9 tests à mettre en place :

✓ **Méthode publique**

- [x] ~~Invalid operator~~
- [x] ~~More than one operation~~
- [x] ~~Left values not a float~~
- [x] ~~right value not a float~~
- [x] ~~operator +~~
- [x] ~~operator -~~
- [x] ~~operator *~~
- [x] ~~operator /~~

- [x] ~~operator / with Zero Division~~

Exercice

L’entreprise a besoin d’un plan de test afin de prévoir l’ensemble des tests que les équipes devront mettre en place dans les jours à venir.

Ainsi, votre mission est de **mettre en place un plan de test détaillé** pour chaque fichier du [projet super-calculatrice] (uniquement pour les tests unitaires).

En résumé

- Il faut tester **l'interface** d'un objet et non son fonctionnement interne.
- Testez uniquement les **méthodes de l’interface publique**.
- Effectuez un **plan de test** afin de mieux organiser et planifier l’implémentation des tests.
- Créez autant de tests que de possibilités de sortie que fournira la méthode publique.