Tester les interactions

Tester en isolation Tester une classe *en isolation* signifie la tester *indépendamment des classes dont elle dépend*. C'est le sens strict du *test unitaire*. Exemple : pour une classe `Caisse` (de paiement) qui utilise une classe `Carte` (de crédit), on souhaite tester les fonctionnalités de `Caisse` sans du tout utiliser la classe `Carte`. Si on teste `Caisse` en utilisant directement `Carte`, c'est plutôt du *test d'intégration*.

Pourquoi tester en isolation ?

- Si un test de `Caisse` échoue, doit-on chercher l'erreur dans `Caisse` ou dans `Carte` ?
- Les dépendances ne sont peut-être pas encore développées.
- Les dépendances sont parfois *difficilement contrôlables* : comment simuler une panne de réseau ? que faire si la dépendance a un comportement aléatoire ?
- Les dépendances sont parfois lourdes à mettre en place : base de données.
- Si le code à tester dépend de beaucoup d'autres classes (ce qui est mauvais signe côté conception !), le « arrange » des tests sera très gros. Cela introduit un fort couplage entre les tests et le reste du système, avec de gros problèmes de maintenance à venir, ce qui incite à laisser tomber les tests.
- On veut parfois tester qu'une dépendance a reçu un appel donné, sans le faire.

Mocks Un *mock* est une *doublure* (au sens cinématographique) appelé aussi *bouchon*. C'est un *objet du test* qui vient remplacer pour la durée du test un objet du domaine applicatif. Le mock reçoit les appels envoyés à l'objet qu'il remplace et est configuré pour le simuler.

Tester les interactions Le principe est d'écrire le test en :

- remplaçant dans le « arrange » les implantations des dépendances par des *mocks* ;
- utiliser si besoin un *oracle* qui vérifie que le « act « appelle correctement les mocks = vérifier que les *interactions* entre le SUT (System Under Test) et les mocks sont correctes.
- **attention** : 1) on mocke les dépendances, **on ne mocke pas la classe à tester** 2) un mock n'a d'autre état que celui qu'on lui donne, donc ne pas lui appliquer d'assertions testant son état.

Dépendances explicites Pour pouvoir remplacer les implantations par des doublures, il faut aussi que les dépendances apparaissent explicitement (par exemple en argument du constructeur ou d'une méthode). Ce pattern s'appelle *injection des dépendances* : il permet un *couplage faible* entre les classes et favorise les *architectures testables*.

Dépendances à des abstractions Quand on utilise un langage statiquement typé comme Java, il faut que la classe à tester dépende d'*interfaces* (abstractions) et non directement des implantations. On peut ainsi manipuler une caisse utilisant un vrai objet `Carte`, ou un mock. La classe `Caisse` dépend d'une interface `CarteI`, dont `Carte` et les mocks seront un sous-type.

Quand on utilise un langage à typage dynamique comme Python, il n'y a pas de notion d'interface au sens de Java. On passe à l'objet une implantation dans l'application effective, et un mock lors des tests. Il faut être rigoureux :

- Si on mocke une dépendance qui n'a pas encore été développée, on garde au fur et à mesure la trace de son interface (au sens du contrat qu'elle est censée respecter) par ex dans une doctest, et on s'y réfère systématiquement (par ex pour ne pas se retrouver avec 2 méthodes `debiter` et `debiter_porte_monnaie` qui devraient n'en faire qu'une, en l'absence des vérifications statiques d'un compilateur)
- Si on mocke une dépendance déjà développée, on prend garde à ce que mocks et implantation aient la même interface, pour ne pas découvrir lors de des tests d'intégration qu'on s'est trompé.

Test des interactions / Test d'état *Test d'état* : test portant sur l'état interne de l'objet, en utilisant les assertions d'une bibliothèque de test unitaire standard (`unittest`, `JUnit`). *Test d'interaction* : test portant sur les interactions entre 2 types d'objets différents. Le test d'interaction demande obligatoirement un mock. On n'a pas systématiquement besoin de l'un ou de l'autre, on a parfois besoin des deux.

Description des mocks Les mocks peuvent être codés à la main. Ils peuvent avantageusement être générés automatiquement à partir d'une description de leur comportement (*stubbing*). Les vérifications à faire en interrogeant les mocks peuvent aussi être décrites. Voir `mockito` pour Python et Java.