Notions de test unitaire et fonctionnel

Table des matières

Introduction

Dans le cadre de projets professionnels, un développeur est amené à contribuer à des programmes très volumineux dont peu des membres du projet, voire aucun, connaissent l'ensemble du code. Lors d'une modification, il peut arriver que des fonctions déjà existantes soient affectées sans que cela soit voulu. Un tel effet de bord pourrait mettre à mal certaines fonctionnalités, voire le programme complet. Pour éviter ce cas de figure, les développeurs créent des tests automatisés qui vérifieront l'intégrité du code à chaque modification ou ajout.

I Définitions

Objectifs

- Comprendre ce qu'est un test automatisé ;
- Connaître les deux principaux types de tests.

Mise en situation

La notion de test est souvent sous-estimée par les développeurs. Ils se lancent dans des projets qui ont pour vocation à grandir et négligent l'atout que représente les tests pour maintenir leur développement. Pour comprendre leur nécessité, il faut d'abord comprendre ce que sont ces tests.

Test automatisé

Az Définition

Un test automatisé est un programme chargé d'en tester un autre.

Pour réaliser ces tests, les tests comparent la **valeur retournée** par le programme testé avec une **valeur prévue** fournie lors de l'écriture du test. Si les deux valeurs ne sont pas égales, le test échouera.

Remarque

Il existe de nombreux types de tests : unitaires, fonctionnels, d'intégrité, d'intégration, etc.

Nous nous concentrerons sur les deux types les plus communs : unitaires et fonctionnels.

Maintenir son code

Fondamental

Ces tests sont utilisés afin de maintenir le code.

Lorsqu'un développeur créer ou modifie une fonction, il arrive que des **effets de bord** apparaissent. Un effet de bord est un impact indirect et imprévu sur le fonctionnement attendu d'un morceau de code. Par effet de cascade, le fonctionnement complet du programme pourrait être compromis.

Lorsqu'un projet grandit, il possède un nombre de fonctionnalités si important qu'il n'est plus possible des les tester manuellement et de s'assurer que les évolutions ne provoquent pas d'effets de bord. Les tests automatisés deviennent stratégiques afin de vérifier l'**intégrité** du projet à chaque modification majeure.

Régression

Az Définition

L'introduction d'un bug sur une fonctionnalité existante lors d'une mise à jour s'appelle une régression : les tests servent également à éviter les régressions.

Test unitaire

Az Définition

Le but d'un test unitaire est de vérifier le bon fonctionnement d'une composante d'un programme, le plus souvent une fonction.

Le test unitaire essaiera d'être exhaustif en testant tous les types de scénarios possibles pour une fonction. Par exemple, si une fonction contient une condition, il faudra au moins deux tests : un dans lequel la condition est vraie et l'autre dans lequel la condition est fausse.

Une telle exhaustivité permet aussi de vérifier que chaque partie du code est utile et atteignable dans des scénarios réalistes.

Test fonctionnel

Az Définition

Le but d'un test fonctionnel est de vérifier le bon fonctionnement d'une fonctionnalité complète.

Le test sera effectué sur une **boîte noire**, c'est-à-dire sans accès aux détails du code.

Ainsi, contrairement au test unitaire qui étudie une composante de code isolée, le test fonctionnel étudie le comportement d'un programme complet. Il s'agit donc d'un test plus global qui ne sera pas aussi exhaustif que le test unitaire, mais validera que les composantes du programme sont correctement assemblées.

Comprendre la différence entre fonctionnel et unitaire

Exemple

Pour comprendre la différence entre test unitaire et test fonctionnel, prenons un exemple simple : un bouton d'achat sur un site de e-commerce. Lors d'un clic sur la bouton *Acheter*, un message est envoyé au serveur pour lui dire d'acheter le produit. Le serveur se charge de l'opération et renvoie un message de réussite ou d'erreur. Voici ce que seraient les différents tests dans ce cas :

- **Test unitaire** : ce test consistera à vérifier qu'un clic sur le bouton résulte bien sur un message envoyé au serveur.
- **Test fonctionnel** : ce test consistera à vérifier l'ensemble de la chaîne en testant la nature de la réponse du serveur suite au clic sur le bouton.

Souvent, les tests unitaires sont développés en premier. La validation des tests unitaires est en général requise avant de dérouler les tests fonctionnels, car les fonctionnalités globales s'appuient sur les composantes du code.

Quand développer un test ?

Méthode

Il est recommandé de développer des tests à chaque fois qu'on livre une nouvelle fonctionnalité. Ces tests permettront d'une part de faire comprendre le fonctionnement de la fonctionnalité à un autre développeur et d'autre part d'assurer que la fonctionnalité ne sera pas compromise par des modifications futures.

Les frameworks de test ⊕ Complément

Il existe des *frameworks* de test qui encadrent et facilitent l'écriture et l'utilisation de tests. Ils complexifient l'écriture des programmes, mais dans le cadre de projets professionnels, il est vivement recommandé d'utiliser ces *frameworks*.

- En Python, les deux *frameworks* principaux sont `unittest` et `pytest`. Le premier est le *framework* historique inclus par défaut à l'installation de Python. Le second est le plus récent et commence à faire l'unanimité dans la communauté Python.
- En JavaScript, il existe de nombreux frameworks mais les plus populaires sont : `Mocha` et `Jasmine`.

À retenir

- Les tests automatisés sont des outils très puissants pour maintenir son code et éviter les régressions.
- Les tests unitaires se concentrent sur le fonctionnement d'un morceau de code isolé
- Les tests fonctionnels se concentrent les fonctionnalités attendues du programme global

Exercice : Appliquer la notion

[solution n°1 p. 27]

Voici une page Wikipédia à lire afin de pouvoir répondre à ce quiz :

https://fr.wikipedia.org/wiki/Test_driven_development

Exercice

Qu'est-ce que le Test-Driven Development ?

- **A** Une méthode de développement
- **B** Un langage de programmation
- **C** Un *framework* pour développer des tests automatisés
- **D** Un standard de développement obligatoire

Exercice

Ce cycle de développement possèdent trois lois, quelles sont-elles ?

- **A** ☐ Vous devez écrire un test qui échoue avant de pouvoir écrire le code de production correspondant.
- **B** ☐ Vous devez écrire une seule assertion à la fois, qui fait échouer le test ou qui échoue à la compilation.
- **C** ☐ Vous devez écrire systématiquement un test après avoir écrit une fonction
- **D** ☐ Vous devez écrire le minimum de code de production pour que l'assertion du test actuellement en échec soit satisfaite.
- **E** ☐ Vous devez écrire un test qui réussit avant de pouvoir écrire le code de production correspondant.
- **F** ☐ Vous devez documenter chaque fonction en s'appuyant sur ses tests si cela est pertinent.

Exercice

Parmi ces propositions, lesquelles expliquent le gain de productivité dû à l'utilisation du TDD ?

A

☐ TDD permet de livrer une nouvelle version d'un logiciel avec un haut niveau de confiance dans la qualité des livrables, confiance justifiée par la couverture et la pertinence des tests à sa construction.

B

☐ TDD permet de produire une documentation plus riche grâce à des tests qui remplacent des paragraphes longs et parfois peu compréhensibles. Cela permet également de ne pas avoir à traduire la documentation car le code est identique quel que soit la langue ou la culture du programmeur.

C

☐ TDD permet de rendre plus efficaces les fonctionnalités grâce une conception plus consciencieuse du code et de ses modifications futures.

D

☐ TDD permet d'éviter les accidents de parcours, où des tests échouent sans qu'on puisse identifier le changement qui en est à l'origine, ce qui aurait pour effet d'allonger la durée d'un cycle de développement.

Exercice

Qu'est-ce que la programmation binomiale ?

A

○ Une méthode de programmation permettant de trouver des bugs en s'appuyant sur la loi binomiale (en Probabilités).

B

○ Une méthode de programmation qui implique de développer chaque fonctionnalité dans un binôme de langage et de toujours retenir l'implémentation la plus claire.

C

○ Une méthode de développement dans laquelle deux programmeurs développent ensemble. Le premier écrit un test en échec puis le second développe la fonctionnalité. Pour la fonctionnalité suivante, les rôles sont inversés.

III Test unitaire

Objectif.

- Savoir écrire un test unitaire simple.

Mise en situation

Une première approche des tests est celle du test unitaire. Il s'agit de tester chaque composant, chaque fonction de notre programme de manière isolée.

Imaginez un programme de gestion d'une association, et qui est donc en charge de plusieurs fonctionnalités : gérer les cotisations, la comptabilité, les dons effectués, etc. Le programme est composé de très nombreuses fonctions, comme par exemple une fonction qui calcule un abattement fiscal sur un don. L'idée du test unitaire sera d'écrire un test qui va se charger de lancer cette fonction plusieurs paramètres différents, et de s'assurer que le résultat est toujours celui que l'on attend.

Test unitaire Rappel

Le but d'un test unitaire est de vérifier le bon fonctionnement d'une composante d'un programme, le plus souvent une fonction. Le test unitaire essaiera d'être exhaustif en vérifiant que chaque ligne de code est viable et ne provoque pas d'erreur. Par exemple, si une fonction contient une condition, il faudra au moins deux tests : un dans lequel la condition est vraie et l'autre dans lequel la condition est fausse.

Fondamental

On appellera test unitaire une fonction qui appelle de multiples fois une autre fonction afin de tester sa validité selon différents paramètres.

Couverture de code Méthode

- Une ligne de code est considérée comme couverte par un test si elle est exécutée durant le test.
- Pour couvrir complètement le code, il faut tester la fonction avec différentes valeurs permettant de vérifier toutes les conditions.
- Les conditions doivent être couvertes car elles sont souvent à l'origine de bugs.

Les *frameworks* de test permettent de calculer facilement la couverture de son code.

Tester les valeurs retournées

Exemple

Voici un exemple de test sur une fonction simple. Si le résultat attendu n'est pas identique à celui retourné par la fonction, le test échoue et affiche un message d'erreur.

```javascript
/** JavaScript : teste unitairement la fonction backetPrice. */
function basketPrice (listItemPrices) {
  let res = 0
  for (let i = 0; i < listItemPrices.length; i++) {
    res = res + listItemPrices[i]
  }
  return res
}

function testBasketPrice () {
  if (basketPrice([2, 5, 29]) !== 36) {
    console.log('Test échoué totalPanier pour [2, 5, 29]')
    return false
  }
  if (basketPrice([]) !== 0) {
    console.log('Test échoué totalPanier pour []')
    return false
  }
  console.log('Test totalPanier réussi')
  return true
}

testBasketPrice()

```

```python
"""Python : teste unitairement la fonction backet_price. """
def basket_price(list_item_prices):
  res = 0
  for i in range(len(list_item_prices)):
    res = res + list_item_prices[i]
  return res

def test_basket_price():
  if basket_price([2, 5, 29]) != 36:
    print("Test échoué basket_price pour [2, 5, 29]")
    return False
  if basket_price([]) != 0:
    print("Test échoué basket_price pour []")
    return False
  print("Test basket_price réussi")
  return True


test_basket_price()
```

Si lors d'une modification du code, une erreur est introduite, par exemple :

```javascript
for (let i = 0; i < listItemPrices.length - 1; i++)
```

Le test échouera et affichera :

```
Test échoué totalPanier pour [2, 5, 29]
```

Ainsi, l'erreur est détectée et corrigeable avant que les utilisateurs n'expérimentent le bug directement.

Utiliser la syntaxe assert

Complément

Pour simplifier les tests, il est possible d'utiliser un nouvel élément de syntaxe : l'assertion. Une assertion ne fera rien de particulier si l'expression qui la suit est vraie mais une erreur sera émise si l'expression est fausse (`AssertionError` pour être précis).

```javascript
/** JavaScript : teste unitairement la fonction backetPrice en utilisant des assert. */
function basketPrice (listItemPrices) {
  let res = 0
  for (let i = 0; i < listItemPrices.length; i++) {
    res = res + listItemPrices[i]
  }
  return res
}

function testBasketPrice () {
  const assert = require('assert')
  assert(basketPrice([2, 5, 29]) === 36)
  assert(basketPrice([]) === 0)
  console.log('Test totalPanier réussi')
  return true
}

testBasketPrice()

```

```python
"""Python : teste unitairement la fonction total_panier en utilisant des assert.
"""
def basket_price(list_item_prices):
  res = 0
  for i in range(len(list_item_prices)):
    res = res + list_item_prices[i]
  return res

def test_basket_price():
  assert basket_price([2, 5, 29]) == 36
  assert basket_price([]) == 0
  print("Test basket_price réussi")
  return True


test_basket_price()
```

Exemple

Voici un programme qui teste une fonction renvoyant la liste des prix TTC à partir des prix HT :

```javascript
/** JavaScript : teste unitairement la fonction htToTtc. */
function htToTtc (htPriceList) {
  const ttcPriceList = []
  for (let i = 0; i < htPriceList.length; i++) {
    const ttcPrice = htPriceList[i] * 1.2
    ttcPriceList.push(ttcPrice)
  }
  return ttcPriceList
}

function testHtToTtc () {
  if (htToTtc([2, 5, 10]).toString() !== [2.4, 6, 12].toString()) {
```

```
    console.log('Test htToTtc échoue pour [2, 5, 10]')
    return false
  }
  if (htToTtc([]).toString() !== [].toString()) {
    // On teste également les valeurs extrêmes
    console.log('Test htToTtc échoue pour [2, 5, 10]')
    return false
  }
  console.log('Test réussi')
  return true
}

testHtToTtc()

```

```
"""Python : teste unitairement la fonction HT_to_TTC. """
def HT_to_TTC(ht_price_list):
  ttc_price_list = []
  for i in range(len(ht_price_list)):
    ttc_price = ht_price_list[i] * 1.2
    ttc_price_list.append(ttc_price)
  return ttc_price_list

def test_HT_to_TTC():
  if HT_to_TTC([2, 5, 10]) != [2.4, 6, 12]:
    print("Test HT_to_TTC échoue pour [2, 5, 10]")
    return False
  if HT_to_TTC([]) != []:  # On teste également les valeurs extrêmes
    print("Test HT_to_TTC échoue pour []")
    return False
  print("Test réussi")
  return True

test_HT_to_TTC()
```

Mocker des variables

⊕ Complément

Un développeur Web écrira des fonctions faisant des requêtes web ou vers des bases de données.

Dans le cadre d'un test, ces appels ne seront pas possibles car on ne peut se permettre d'interroger les systèmes en production. Pour pallier ce problème, il est possible d'utiliser des *mocks* qui remplacent certaines variables par une valeur pré-définie.

À retenir

- Un test unitaire vérifie automatiquement le comportement d'une fonction, avec des valeurs simples et des valeurs extrêmes.
- Un bon test unitaire doit prévoir tous les cas d'utilisation de la fonction testée.

IV Exercice : Appliquer la notion

Voici une fonction qui trie dans l'ordre croissant la liste qui lui est passée en paramètre :

```
function bubbleSort (unsortedList) {
  const intList = unsortedList.slice()
  for (let i = intList.length - 1; i > 0; i--) {
    for (let j = 0; j < i; j++) {
      if (intList[j + 1] < intList[j]) {
        // Echanger les deux valeurs
        const temp = intList[j + 1]
        intList[j + 1] = intList[j]
        intList[j] = temp
      }
    }
  }
  return intList
}

```

Question 1

[solution n°2 p. 29]

Écrire une fonction `testBubbleSort` qui teste le résultat de la fonction pour [3,2,1], [1,2,3], [1,3,2].

Cette fonction doit afficher une erreur si le résultat attendu ne correspond pas au résultat renvoyé.

Indice :

Il est possible de comparer deux tableaux « *a* » et « *b* », en prenant l'ordre en compte, avec le test :

```
a.toString() !== b.toString()
```

Question 2

[solution n°3 p. 29]

Quel test de valeur extrême faudrait-il rajouter pour s'assurer que la fonction est correcte ?

Ajouter ce test.

Indice :

Les valeurs extrêmes sont en général des **limites**. Pour un entier positif, on testerait le comportement si la valeur d'entrée vaut 0.

V Test fonctionnel

Objectif

- Savoir écrire un test fonctionnel.

Mise en situation

Les tests fonctionnels testent directement les **fonctionnalités** d'un programme du point de vue de l'**utilisateur**. Pour cela, au lieu de tester une fonction du code source, on va tester une fonctionnalité du programme, par exemple que l'inscription fonctionne correctement sur un site web. Cette manière de tester est moins ancrée dans la technique que les tests unitaires, et plus orientée vers l'expérience utilisateur. On veut s'assurer ici que l'on n'introduit pas de régressions dans le fonctionnement de notre logiciel.

Test fonctionnel

Rappel

Le but d'un test fonctionnel est de vérifier le bon fonctionnement d'une fonctionnalité complète. Le test sera effectué sur une boite noire, c'est-à-dire que les opérations intermédiaires effectuées par la fonctionnalité sont ignorées. Ainsi, contrairement au test unitaire qui étudie le déroulé d'une composante, le test fonctionnel étudie le comportement d'un programme. Il s'agit donc d'un test plus global qui ne sera pas aussi exhaustif que le test unitaire.

Comment tester fonctionnellement ?

Méthode

Pour tester fonctionnellement, il y a deux étapes : le fonctionnement normal et le fonctionnement accidentel.

- Le **fonctionnement normal** assure que la fonctionnalité se comporte correctement dans les grandes lignes sans essayer tous les cas de figures (la combinatoire est en général trop importante).
- Le **fonctionnement accidentel** essaie des scénarios anormaux qui pourraient faire planter le programme.

Programme résistant aux accidents

Fondamental

Un programme que les tests de fonctionnement accidentel ne font pas planter, sera considéré comme un programme résistant aux accidents.

Il arrive également de croiser une terminologie anglaise plus négativement connotée : `idiot-proof program` qui fait référence à l'idiotie supposée des entrées testées.

Loi de Murphy

⚠ Attention

Selon la *Loi de Murphy*, Tout ce qui est susceptible d'aller mal, ira mal. Elle est également vraie en programmation car les utilisateurs se trompent parfois dans l'utilisation d'une application. Cette recherche passera très souvent par un usage totalement illogique de la fonctionnalité. Ainsi, tester des cas de figures **ridicules** permettra d'éviter ce type d'erreurs pouvant parfois être utilisés pour attaquer une application. Par exemple, un utilisateur pourrait entrer *« -18 »* dans la case *« Age »* pour voir si le programme plantera à cause de l'âge inférieur à 0.

👁 Exemple

Voici un exemple simplifié de test fonctionnel de transfert d'argent d'une application bancaire.

Ici, deux cas sont essayés :

1. transférer avec assez de solde,
2. et transférer sans avoir assez de solde.

Le test fonctionnel sert ici à valider que le transfert échoue si le solde est insuffisant.

```
1  /** JavaScript : teste fonctionnellement le transfert d'argent. */
2  function checkAccount (account, amount) {
3    if (account.balance >= amount) {
4      return true
5    }
6    return false
7  }
8
9  function transfer (srcAccount, tgtAccount, amount) {
10   // Copies des variables
11   const newSrcAccount = { ...srcAccount }
12   const newTgtAccount = { ...tgtAccount }
13   if (checkAccount(newSrcAccount, amount)) {
14     newSrcAccount.balance = newSrcAccount.balance - amount
15     newTgtAccount.balance = newTgtAccount.balance + amount
16     console.log('Transfert réussi')
17   } else {
18     console.log('Echec du transfert')
19   }
20   return [newSrcAccount, newTgtAccount]
21 }
22
23 function testTransfer () {
24   let srcAcc = { owner: 'Jean Dupont', balance: 100 }
25   let tgtAcc = { owner: 'Anne Martin', balance: 20 }
26
27   let transferRes = transfer(srcAcc, tgtAcc, 60)
28   srcAcc = transferRes[0]
29   tgtAcc = transferRes[1]
30   // Transfert réussi
31   if (srcAcc.balance !== 40 || tgtAcc.balance !== 80) {
32     console.log('Test transfer échoué lors du premier transfert.')
33     return false
34   }
35
36   transferRes = transfer(srcAcc, tgtAcc, 60)
37   srcAcc = transferRes[0]
38   tgtAcc = transferRes[1]
39   // Transfert échoué donc les comptes doivent être inchangés
```

```
  if (srcAcc.balance !== 40 || tgtAcc.balance !== 80) {
    console.log('Test transfer échoué lors du second transfert.')
    return false
  }
  console.log('Test réussi')
  return true
}

testTransfer()

```

```
"""Python : teste fonctionnellement le transfert d'argent. """
def check_account(account, amount):
  if account["balance"] >= amount:
    return True
  else:
    return False

def transfer(src_account, tgt_account, amount):
  new_src_account = src_account.copy()
  new_tgt_account = tgt_account.copy()
  if check_account(new_src_account, amount):
    new_src_account["balance"] = new_src_account["balance"] - amount
    new_tgt_account["balance"] = new_tgt_account["balance"] + amount
    print("Transfert réussi")
  else:
    print("Echec du transfert")
  return new_src_account, new_tgt_account


def test_transfer():
  src_acc = {"owner": "Jean Dupont", "balance": 100}
  tgt_acc = {"owner": "Anne Martin", "balance": 20}
  src_acc, tgt_acc = transfer(src_acc, tgt_acc, 60)
  # Transfert réussi
  if src_acc["balance"] != 40 or tgt_acc["balance"] != 80:
    print("Test transfer échoué lors du premier transfert.")
    return False

  src_acc, tgt_acc = transfer(src_acc, tgt_acc, 60)
  # Transfert échoué donc les comptes doivent être inchangés
  if src_acc["balance"] != 40 or tgt_acc["balance"] != 80:
    print("Test transfer échoué lors du premier transfert.")
    return False

  print("Test réussi")
  return True

test_transfer()
```

Tester les entrées/sorties

Rappel

Tester les entrées/sorties d'un programme est souvent une tâche complexe.

- Ces tests sont facilités par les *frameworks* de test.
- Les développeurs ont recours à des *mocks*, qui simulent des réponses de bases de données ou de serveurs.

Test d'intégration ⊕ Complément

Un test d'intégration vérifie que la communication de plusieurs fonctions se fait comme prévue. Il contrôle les **assemblages de fonctions**.

À retenir

- Les tests fonctionnels sont des tests globaux qui testent le comportement d'une fonctionnalité.
- Il est nécessaire de tester des valeurs correctes ainsi que des valeurs intentionnellement fausses afin que les fonctionnalités soient résistantes aux usages imprévus.
- Pour développer des tests pour des projets de grande envergure il est nécessaire d'utiliser un *framework* de test.

VI Exercice : Appliquer la notion

Une application bancaire permet d'acheter des produits grâce à la fonction buy que voici :

```
function checkAccount (account, amount) {
  if (account.balance >= amount) {
    return true
  }
  return false
}

function buy (account, product) {
  // Copie de la variable account
  const newAccount = { ...account }
  if (checkAccount(newAccount, product.price)) {
    newAccount.balance = newAccount.balance - product.price
    console.log('Produit acheté:', product.name)
  } else {
    console.log('Echec de la transaction')
  }
  return newAccount
}

```

Les enregistrements gérant un compte sont de la forme suivante :

```
let account = {
  owner: 'Pierre',
  balance: 10
}
```

Les produits sont de la forme suivante :

```
const product = {
  name: 'Traces',
  price: 19
}
```

Question 1

[solution n°4 p. 29]

Écrire le code permettant de créer un compte avec une balance de 100 €, ainsi qu'un produit d'une valeur de 60 €.

Question 2

[solution n°5 p. 30]

Écrire le test fonctionnel `testBuy` de la fonctionnalité de paiement, qui vérifie qu'un solde suffisant valide l'achat et qu'un solde insuffisant le faire échouer.

Tenter d'acheter deux fois le produit avec le compte créé précédemment.

Vérifier que le test échoue.

VII Quiz

Exercice 1 : Quiz - Culture

[solution n°6 p. 30]

Exercice

Quel est l'intérêt de développer des tests automatisés ?

- **A** Accélérer le développement
- **B** Faciliter l'évolution et la maintenance de son code
- **C** Réduire la taille de son code
- **D** Améliorer la lisibilité du code
- **E** Détecter des erreurs dans son code

Exercice

Mocha est un *framework* de tests. Quel langage concerne-t-il ?

- **A** Python
- **B** JavaScript
- **C** Java
- **D** Pytest

Exercice

Quels sont les critères qui font qu'un test peut être **automatisé** ? Aidez-vous de cette page : https://fr.wikipedia.org/wiki/Automatisation_de_test

- **A** Répétitivité
- **B** Simplicité
- **C** Systématique
- **D** Rapidité

Exercice

Comment appelle-t-on une variable ou une fonction dont la valeur a été **simulée** dans le cadre d'un test ? Ceci est réalisé grâce à des *frameworks* pour tester des fonctions qui appellent des éléments externes comme des ressources Web ou une base de données. Voici une page Web qui pourrait aider : https://fr.wikipedia.org/wiki/Mock_(programmation_orient%C3%A9e_objet)

- **A** Joker
- **B** Mock
- **C** Remplaçante
- **D** Dummy

Exercice

Quels rôles peuvent remplir les tests fonctionnels ?

- **A** Tester le bon affichage d'une interface web
- **B** Tester le bon comportement d'une fonctionnalité
- **C** Tester la résistance aux valeurs invalides

Exercice

Qu'est-ce qu'un programme robuste aux accidents ?

- **A** Un programme qui résistera même si l'utilisateur entre des valeurs absurdes
- **B** Un programme qui corrige automatiquement les erreurs (accidents) du développeur
- **C** Un programme d'intelligence artificielle
- **D** Un programme de signalisation routière

Exercice 8 : Quiz - Code

[solution n°7 p. 32]

Exercice

La fonction `isFriend()` est-elle couverte à 100 % par `testIsFriend()` ?

```
1 function isFriend (friendList, user) {
2   for (let i = 0; i < friendList.length; i++) {
3     if (JSON.stringify(user) === JSON.stringify(friendList[i])) {
4       return true
5     }
6   }
7   return false
```

```
}

function testIsFriend () {
  const friends = [
    { lastName: 'Jean', firstName: 'Zay' },
    { lastName: 'Marie', firstName: 'Curie' },
    { lastName: 'Veil', firstName: 'Simone' }
  ]
  const user = { lastName: 'Veil', firstName: 'Simone' }
  if (!isFriend(friends, user)) {
    console.log('Test estAmi a échoué')
    return false
  }
  console.log('Test estAmi a réussi')
  return true
}

testIsFriend()

```

A Oui

B Non

Exercice

Le test ci-dessous affiche : `Test estAmi a réussi`

```
function isFriend (friendList, user) {
  for (let i = 0; i < friendList.length; i++) {
    if (JSON.stringify(user) === JSON.stringify(friendList[i])) {
      return true
    }
  }
  return false
}

function testIsFriend () {
  const friends = [
    { lastName: 'Jean', firstName: 'Zay' },
    { lastName: 'Marie', firstName: 'Curie' },
    { lastName: 'Veil', firstName: 'Simone' }
  ]
  const user = { lastName: 'Veil', firstName: 'Simone' }
  if (!isFriend(friends, user)) {
    console.log('Test estAmi a échoué')
    return false
  }
  if (isFriend(friends, {})) {
    console.log('Test estAmi a échoué')
    return false
  }
  console.log('Test estAmi a réussi')
  return true
}

testIsFriend()

```

A Vrai

B Faux

VIII Exercice : Défi

Nois développons une application d'e-commerce. Il est possible d'acheter les produits grâce à un porte-monnaie virtuel inclus dans l'application. Voici les fonctions de l'application dans un programme montrant comment elles peuvent être utilisées :

```
// On définit une classe représentant un panier
class Basket {
  constructor (items = [], totalPrice = 0) {
    this.items = items
    this.totalPrice = totalPrice
  }
}

function addToBasket (basket, item) {
  basket.items.push(item)
  basket.totalPrice = basket.totalPrice + item.price
}

function removeFromBasket (basket, item) {
  for (let i = 0; i < basket.items.length; i++) {
    if (JSON.stringify(item) === JSON.stringify(basket.items[i])) {
      basket.items.splice(i, 1)
      basket.totalPrice = basket.totalPrice - item.price
      break
    }
  }
}

function transactionAllowed (userAccount, priceToPay) {
  if (userAccount.balance >= priceToPay) {
    return true
  }
  return false
}

function payBasket (userAccount, basket) {
  if (transactionAllowed(userAccount, basket.totalPrice)) {
    userAccount.balance = userAccount.balance - basket.totalPrice
    console.log('Paiement du panier réussi')
  } else {
    console.log('Paiement du panier échoué')
  }
}

const currentBasket = new Basket()

const item1 = { name: 'Carte mère', price: 100 }
const item2 = { name: 'Carte graphique', price: 300 }
const user = { name: 'Perceval', balance: 500 }
addToBasket(currentBasket, item1)
addToBasket(currentBasket, item2)

// Plus qu'un produit dans le panier
removeFromBasket(currentBasket, item1)
console.log(currentBasket)
payBasket(user, currentBasket)
console.log(user)
// Perceval n'a plus que 200 euros
```

Question 1

[solution n°8 p. 34]

Développer un test unitaire qui ajoute un produit au panier et vérifie que le montant est bien celui prévu. Ce test sera nommé `testAdd()`.

Indice :

Par exemple :

- Ajouter un produit dans le panier, comme :

```
1 const item = {name: "Carte mère", price: 100}
```

- Vérifier que `basket.totalPrice` vaut 100 :

Question 2

[solution n°9 p. 34]

Développer un test unitaire qui supprime un produit du panier et vérifie que le montant est bien celui prévu. Ce test sera nommé `testRemove()`.

Indice :

Il faudra au préalable ajouter un produit dans un panier vide avant de pouvoir retirer celui-ci pour tester le retrait.

Par exemple :

- Ajouter un objet au panier ;
- Le retirer immédiatement ;
- Vérifier que le montant total du panier est nul.

Question 3

[solution n°10 p. 34]

On constate qu'il est possible de factoriser les tests unitaires des fonctions d'ajout et de retrait précédentes.

Donner le test factorisé nommé `testAddRemove()`.

Indice :

Le test de retrait se base sur l'ajout d'un produit. Il est donc possible de tester d'abord l'ajout d'un produit, puis son retrait, au sein de la même fonction.

Question 4

[solution n°11 p. 35]

Donner maintenant un test unitaire qui teste entièrement la fonction `transactionAllowed()`. La fonction de test s'appellera `testTransactionAllowed()`.

Indice :

Il faut que chacune des branches de la condition soit vérifiée, donc il faudra appeler deux fois la fonction testée : une fois pour un solde suffisant, une fois pour un solde insuffisant.

Par exemple :

- Ajouter un utilisateur avec un solde de 500 € ;
- Vérifier qu'une transaction de 400 € est autorisée ;
- Vérifier qu'une transaction de 600 € est interdite.

Question 5

[solution n°12 p. 35]

Écrire un test fonctionnel pour le règlement d'un panier qu'on nommera `testPayBasket()`. Le test vérifiera que le solde de l'utilisateur est bien mis à jour après règlement.

Indice :

Pour réaliser ce test fonctionnel, il faut créer un panier puis essayer de régler deux fois le panier. La première fois, la transaction doit réussir. La seconde fois, la transaction doit échouer car l'utilisateur n'a plus assez d'argent, donc son solde ne doit pas être mis à jour.

Par exemple :

- Ajouter un utilisateur avec un solde de 500 € ;
- Ajouter un produit à son panier valant 300 € ;
- Effectuer le règlement et vérifier que son solde est passé à 200 € ;
- Essayer d'effectuer un deuxième règlement et vérifier que son solde reste à 200 €.

Question 6

[solution n°13 p. 35]

Écrire une fonction `testAppEcommerce()` qui lance successivement tous les tests. Cette fonction affichera *« OK »* si tous les tests sont passés, et *« ERREUR »* sinon.

Écrire le programme complet permettant de réaliser tous les tests.

Indice :

Tous les tests retournent `true` ou `false`.

Indice :

Il est possible de faire des opérations booléennes :

```
1 const success = true && false // Ici, on utilise l'opérateur logique ET
```

On pourra alors vérifier que l'intégralité des tests a renvoyé `true`, en chaînant les &&.

Indice :

```
1  function testAppEcommerce () {
2    let success = testAddRemove()
3    success = success && testTransactionAllowed()
4    success = success && testPayBasket()
5    if (success) {
6      console.log('OK')
7    } else {
8      console.log('ERREUR')
9    }
10 }
```

Conclusion

Écrire des tests pour son code est une tâche qui fait partie du métier de développeur et qu'il ne faut pas négliger. Les tests permettent d'apporter plus de fiabilité, et donc plus de sérénité lorsque l'on déploie une mise à jour de son code. Il est important de ne pas remettre les tests à plus tard, et de commencer leur écriture dès le début du développement d'un programme.

Nous avons vu qu'il existe deux types de tests, unitaire et fonctionnel. Leur approche n'est pas concurrente mais complémentaire, et ces deux types de tests sont généralement mis en place conjointement.

Solutions des exercices

Solution n°1

[exercice p. 7]

Exercice

Qu'est-ce que le Test-Driven Development ?

- (A) Une méthode de développement
- **B** Un langage de programmation
- **C** Un *framework* pour développer des tests automatisés
- **D** Un standard de développement obligatoire — Il ne s'agit pas d'un standard à proprement parler. Chaque développeur a sa propre méthode de développement. Il existe des méthodologies formalisées avec un but précis, comme le TDD, que les développeurs sont libres d'adopter et d'adapter.

Exercice

Ce cycle de développement possèdent trois lois, quelles sont-elles ?

- **A** ☐ Vous devez écrire un test qui échoue avant de pouvoir écrire le code de production correspondant.
- **B** ☐ Vous devez écrire une seule assertion à la fois, qui fait échouer le test ou qui échoue à la compilation.
- **C** Vous devez écrire systématiquement un test après avoir écrit une fonction
- **D** ☐ Vous devez écrire le minimum de code de production pour que l'assertion du test actuellement en échec soit satisfaite.
- **E** Vous devez écrire un test qui réussit avant de pouvoir écrire le code de production correspondant.
- **F** Vous devez documenter chaque fonction en s'appuyant sur ses tests si cela est pertinent. — Ceci est une très bonne pratique mais n'est pas une loi du TDD.

Le TDD se base sur l'idée générale suivante : écrire les tests qui échoue avant d'implémenter le code qui résout le problème. De cette manière, on est certain que le test est correct et que la fonction répond bien aux attentes.

Exercice

Parmi ces propositions, lesquelles expliquent le gain de productivité dû à l'utilisation du TDD ?

A

☐ TDD permet de livrer une nouvelle version d'un logiciel avec un haut niveau de confiance dans la qualité des livrables, confiance justifiée par la couverture et la pertinence des tests à sa construction.

B

TDD permet de produire une documentation plus riche grâce à des tests qui remplacent des paragraphes longs et parfois peu compréhensibles. Cela permet également de ne pas avoir à traduire la documentation car le code est identique quel que soit la langue ou la culture du programmeur.

La documentation reste nécessaire car un code reste peu intelligible sans explication. Cela n'est pas affaire de langue ou de culture mais de manière de penser.

C

TDD permet de rendre plus efficaces les fonctionnalités grâce une conception plus consciencieuse du code et de ses modifications futures.

TDD n'exige rien sur la maintenabilité du code en lui-même.

D

☐ TDD permet d'éviter les accidents de parcours, où des tests échouent sans qu'on puisse identifier le changement qui en est à l'origine, ce qui aurait pour effet d'allonger la durée d'un cycle de développement.

Exercice

Qu'est-ce que la programmation binomiale ?

A

Une méthode de programmation permettant de trouver des bugs en s'appuyant sur la loi binomiale (en Probabilités).

B

Une méthode de programmation qui implique de développer chaque fonctionnalité dans un binôme de langage et de toujours retenir l'implémentation la plus claire.

C

◯ Une méthode de développement dans laquelle deux programmeurs développent ensemble. Le premier écrit un test en échec puis le second développe la fonctionnalité. Pour la fonctionnalité suivante, les rôles sont inversés.

Solution n°2

[exercice p. 13]

```
function testBubbleSort () {
  if (bubbleSort([3, 2, 1]).toString() !== [1, 2, 3].toString()) {
    console.log('Test échoué pour [3, 2, 1]')
    return false
  }
  if (bubbleSort([1, 2, 3]).toString() !== [1, 2, 3].toString()) {
    console.log('Test échoué pour [1, 2, 3]')
    return false
  }
  if (bubbleSort([1, 3, 2]).toString() !== [1, 2, 3].toString()) {
    console.log('Test échoué pour [1, 3, 2]')
    return false
  }
  console.log('Test réussi')
  return true
}

testBubbleSort()
```

Solution n°3

[exercice p. 13]

Il serait pertinent d'ajouter un test pour un tableau vide : [ ]

```
if (bubbleSort([]).toString() !== [].toString()) {
  console.log('Test échoué pour []')
  return false
}
```

Complément

Les tests fonctionnent. Mais en revanche, pour être vraiment exhaustifs, il faudrait aussi tester des valeurs **aberrantes** : envoyer un nombre, une valeur `undefined`, etc.

Avec le code actuel, la fonction renverra une erreur. Cette erreur doit être maîtrisée, et il faut le vérifier.

Solution n°4

[exercice p. 18]

```
let acc = {
  owner: 'Jean Dupont',
  balance: 100
}

const prod = {
  name: 'Carte graphique',
  price: 60
}
```

Solution n°5

[exercice p. 18]

Une implémentation possible est la suivante :

```
function testBuy () {
  let acc = {
    owner: 'Jean Dupont',
    balance: 100
  }
  const prod = {
    name: 'Carte graphique',
    price: 60
  }
  acc = buy(acc, prod)
  // Paiement réussi
  if (acc.balance !== 40) {
    console.log('Test buy échoué sur le premier paiement')
    return false
  }

  acc = buy(acc, prod)
  // Echec du paiement
  if (acc.balance !== 40) {
    console.log('Test buy échoué sur le second paiement')
    return false
  }
  console.log('Test réussi')
  return true
}

testBuy()
```

Son exécution renvoie :

```
Produit acheté: Carte graphique
Echec de la transaction
Test réussi
```

Ce qui valide que la fonctionnalité se comporte comme attendue.

Solution n°6

[exercice p. 19]

Exercice

Quel est l'intérêt de développer des tests automatisés ?

A	Accélérer le développement	Le développement n'est pas accéléré. Au contraire, il pourrait être allongé lorsque les développeurs ne sont pas habitués à cette pratique.
B	Faciliter l'évolution et la maintenance de son code	
C	Réduire la taille de son code	Au contraire, le code sera plus volumineux car il inclura les tests.

D

Améliorer la lisibilité du code — La lisibilité n'est pas améliorée. Au mieux, les tests permettent à un relecteur de comprendre comme chaque fonction devrait être utilisée mais la lisibilité intrinsèque de la fonction elle-même est inchangée.

E Détecter des erreurs dans son code

Les tests permettent de valider le bon fonctionnement du code **et** d'éviter l'introduction de bug lors des évolutions du code, donc de faciliter sa maintenance.

Exercice

Mocha est un *framework* de tests. Quel langage concerne-t-il ?

A Python

B JavaScript

C Java

D Pytest

Exercice

Quels sont les critères qui font qu'un test peut être **automatisé** ? Aidez-vous de cette page : https://fr.wikipedia.org/wiki/Automatisation_de_test

A

Répétitivité — Plus un test est répétitif, plus ce sera une tâche qui devrait être fait par une machine.

B

Simplicité — Il est évidemment préférable que le test soit simple, mais un test complexe n'est pas nécessairement une barrière à son automatisation.

C

Systématique — Dans le meilleur des cas, ce test doit être utilisé à chaque nouvelle version. Plus un test est utilisé, plus l'intérêt de son automatisation est grand.

D Rapidité

Exercice

Comment appelle-t-on une variable ou une fonction dont la valeur a été **simulée** dans le cadre d'un test ? Ceci est réalisé grâce à des *frameworks* pour tester des fonctions qui appellent des éléments externes comme des ressources Web ou une base de données. Voici une page Web qui pourrait aider : https://fr.wikipedia.org/wiki/Mock_(programmation_orient%C3%A9e_objet)

- **A** Joker
- **B** Mock
- **C** Remplaçante
- **D** Dummy

Exercice

Quels rôles peuvent remplir les tests fonctionnels ?

- [x] **A** Tester le bon affichage d'une interface web
- [x] **B** Tester le bon comportement d'une fonctionnalité
- [ ] **C** Tester la résistance aux valeurs invalides

  Un test fonctionnel peut réaliser sa tâche en utilisant des valeurs invalides notamment pour vérifier que des entrées accidentelles ne fassent pas planter le programme.

Exercice

Qu'est-ce qu'un programme robuste aux accidents ?

- **A** Un programme qui résistera même si l'utilisateur entre des valeurs absurdes
- **B** Un programme qui corrige automatiquement les erreurs (accidents) du développeur
- **C** Un programme d'intelligence artificielle
- **D** Un programme de signalisation routière

Solution n°7

[exercice p. 20]

Exercice

La fonction `isFriend()` est-elle couverte à 100 % par `testIsFriend()` ?

```
1  function isFriend (friendList, user) {
2    for (let i = 0; i < friendList.length; i++) {
3      if (JSON.stringify(user) === JSON.stringify(friendList[i])) {
4        return true
5      }
6    }
7    return false
8  }
9
10 function testIsFriend () {
```

```
  const friends = [
    { lastName: 'Jean', firstName: 'Zay' },
    { lastName: 'Marie', firstName: 'Curie' },
    { lastName: 'Veil', firstName: 'Simone' }
  ]
  const user = { lastName: 'Veil', firstName: 'Simone' }
  if (!isFriend(friends, user)) {
    console.log('Test estAmi a échoué')
    return false
  }
  console.log('Test estAmi a réussi')
  return true
}

testIsFriend()

```

A Oui

B Non

La fonction n'est pas couverte à 100 % car le test ne vérifie pas que la fonction renvoie faux lorsqu'on lui fournit une personne qui ne fait pas partie de la liste d'amis.

Exercice

Le test ci-dessous affiche : `Test estAmi a réussi`

```
function isFriend (friendList, user) {
  for (let i = 0; i < friendList.length; i++) {
    if (JSON.stringify(user) === JSON.stringify(friendList[i])) {
      return true
    }
  }
  return false
}

function testIsFriend () {
  const friends = [
    { lastName: 'Jean', firstName: 'Zay' },
    { lastName: 'Marie', firstName: 'Curie' },
    { lastName: 'Veil', firstName: 'Simone' }
  ]
  const user = { lastName: 'Veil', firstName: 'Simone' }
  if (!isFriend(friends, user)) {
    console.log('Test estAmi a échoué')
    return false
  }
  if (isFriend(friends, {})) {
    console.log('Test estAmi a échoué')
    return false
  }
  console.log('Test estAmi a réussi')
  return true
}

testIsFriend()

```

A Vrai

B Faux

Solution n°8

[exercice p. 24]

```
function testAdd () {
  const testBasket = new Basket()
  const item = { name: 'Carte mère', price: 100 }
  addToBasket(testBasket, item)
  if (testBasket.totalPrice !== 100) {
    console.log('Test ajout échoué')
    return false
  }
  console.log('Test ajout réussi')
  return true
}

testAdd()
```

Solution n°9

[exercice p. 24]

```
function testRemove () {
  const testBasket = new Basket()
  const item = { name: 'Carte mère', price: 100 }
  addToBasket(testBasket, item)

  removeFromBasket(testBasket, item)
  if (testBasket.totalPrice !== 0) {
    console.log('Test retrait échoué lors du premier retrait')
    return false
  }
  console.log('Test retrait réussi')
  return true
}

testRemove()
```

Solution n°10

[exercice p. 24]

```
function testAddRemove () {
  const testBasket = new Basket()
  const item = { name: 'Carte mère', price: 100 }
  addToBasket(testBasket, item)
  if (testBasket.totalPrice !== 100) {
    console.log("Test ajout-retrait échoué lors de l'ajout")
    return false
  }

  removeFromBasket(testBasket, item)
  if (testBasket.totalPrice !== 0) {
    console.log('Test ajout-retrait échoué lors du premier retrait')
    return false
  }

```

```
  console.log('Test ajout-retrait réussi')
  return true
}

testAddRemove()
```

Solution n°11

[exercice p. 24]

```
function testTransactionAllowed () {
  const testUser = { name: 'Perceval', balance: 500 }
  if (!transactionAllowed(testUser, 400)) {
    console.log('Test transactionAllowed échoué pour 400')
    return false
  }
  if (transactionAllowed(testUser, 600)) {
    console.log('Test transactionAllowed échoué pour 600')
    return false
  }
  console.log('Test transactionAllowed réussi')
  return true
}

testTransactionAllowed()
```

Solution n°12

[exercice p. 25]

```
function testPayBasket () {
  const testUser = { name: 'Perceval', balance: 500 }
  let testBasket = new Basket()
  testBasket = addToBasket(testBasket, { name: 'Carte mère', price: 300 })

  payBasket(testUser, testBasket)
  // Paiement réussi
  if (testUser.balance !== 200) {
    console.log('Test régler panier échoué lors de la première transaction')
    return false
  }

  payBasket(testUser, testBasket)
  // Paiement échoué car le solde n'a pas changé
  if (testUser.balance !== 200) {
    console.log('Test régler panier échoué lors de la première transaction')
    return false
  }

  console.log('Test de la fonctionnalité de règlement du panier réussi')
  return true
}

testPayBasket()
```

Solution n°13

[exercice p. 25]

```
// On définit une classe représentant un panier
class Basket {
  constructor (items = [], totalPrice = 0) {
    this.items = items
    this.totalPrice = totalPrice
```

```
  }
}

function addToBasket (basket, item) {
  basket.items.push(item)
  basket.totalPrice = basket.totalPrice + item.price
}

function removeFromBasket (basket, item) {
  for (let i = 0; i < basket.items.length; i++) {
    if (JSON.stringify(item) === JSON.stringify(basket.items[i])) {
      basket.items.splice(i, 1)
      basket.totalPrice = basket.totalPrice - item.price
      break
    }
  }
}

function transactionAllowed (userAccount, priceToPay) {
  if (userAccount.balance >= priceToPay) {
    return true
  }
  return false
}

function payBasket (userAccount, basket) {
  if (transactionAllowed(userAccount, basket.totalPrice)) {
    userAccount.balance = userAccount.balance - basket.totalPrice
    console.log('Paiement du panier réussi')
  } else {
    console.log('Paiement du panier échoué')
  }
}

function testAddRemove () {
  const testBasket = new Basket()
  const item = { name: 'Carte mère', price: 100 }
  addToBasket(testBasket, item)
  if (testBasket.totalPrice !== 100) {
    console.log("Test ajout-retrait échoué lors de l'ajout")
    return false
  }

  removeFromBasket(testBasket, item)
  if (testBasket.totalPrice !== 0) {
    console.log('Test ajout-retrait échoué lors du premier retrait')
    return false
  }

  console.log('Test ajout-retrait réussi')
  return true
}

function testTransactionAllowed () {
  const testUser = { name: 'Perceval', balance: 500 }
  if (!transactionAllowed(testUser, 400)) {
    console.log('Test transactionAllowed échoué pour 400')
    return false
  }
  if (transactionAllowed(testUser, 600)) {
    console.log('Test transactionAllowed échoué pour 600')
    return false
  }
```

```
  console.log('Test transactionAllowed réussi')
  return true
}

function testPayBasket () {
  const testUser = { name: 'Perceval', balance: 500 }
  const testBasket = new Basket()
  addToBasket(testBasket, { name: 'Carte mère', price: 300 })

  payBasket(testUser, testBasket)
  // Paiement réussi
  if (testUser.balance !== 200) {
    console.log('Test régler panier échoué lors de la première transaction')
    return false
  }

  payBasket(testUser, testBasket)
  // Paiement échoué car le solde n'a pas changé
  if (testUser.balance !== 200) {
    console.log('Test régler panier échoué lors de la deuxième transaction')
    return false
  }

  console.log('Test de la fonctionnalité de règlement du panier réussi')
  return true
}

function testAppEcommerce () {
  let success = testAddRemove()
  success = success && testTransactionAllowed()
  success = success && testPayBasket()
  if (success) {
    console.log('OK')
  } else {
    console.log('ERREUR')
  }
}

testAppEcommerce()

```