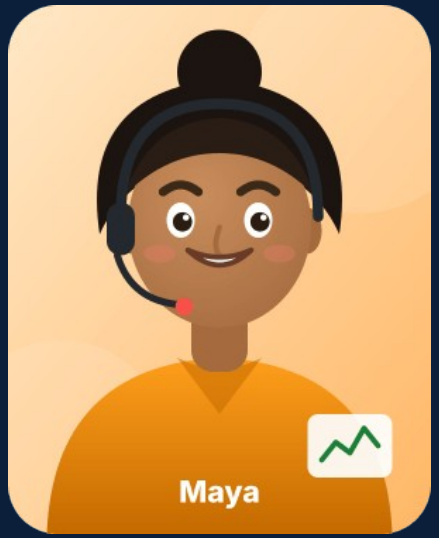
HEF-110 · MISSION 02 / 10

Spécifications et comportements

On ne teste pas du code : on teste des comportements promis. Il faut donc d'abord savoir les décrire.



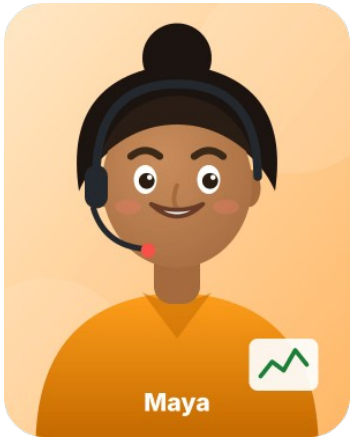


Badge : Traducteur de besoins

90 min · intermédiaire

L'histoire

Léo, formateur, demande : « Je voudrais que la livraison des supports papier soit gratuite à partir d'un certain montant. » Kodi code, teste… et Léo répond : « Ah non, pas pour les colis lourds ! Et c'est *à partir de* 50 €, c'est-à-dire 50 € compris ? »



> *« Le bug n'est pas dans le code, dit Maya. Il est dans la phrase. Avant de tester, il faut rendre la règle **précise** : quelles entrées, quelles sorties, quelles exceptions, et quels cas aux limites. »*

Tu apprends à transformer une demande floue en comportements précis, à en dériver des cas de test méthodiquement, et à penser aux frontières.

Ta mission

À la fin, je serai capable de…

1. transformer une exigence en exemples concrets (étant donné / quand / alors)
2. dériver des cas de test par classes d'équivalence et valeurs limites
3. écrire des tests pilotés par table avec `subTest`
4. tester les erreurs et cas interdits
5. vérifier des propriétés générales avec des entrées aléatoires

Défi d'échauffement

Défi d'échauffement

« Les enfants paient moins cher. » Écris trois questions qui montrent que cette phrase est ambiguë (À partir de quel âge ? 12 ans pile ? Compte-t-on l'âge à la date du billet ?). Puis réécris la règle en une table précise de 5 lignes.

Le concept

De la phrase au comportement

Une exigence floue (« livraison gratuite pour les gros paniers ») doit devenir une **spécification testable**. Un bon format : **Étant donné / Quand / Alors** (*Given / When / Then*).

```
Étant donné un panier de 60 € et un colis de 1 kg
Quand on calcule les frais de livraison
Alors les frais sont de 0 €
```

Cette phrase est à la fois une **exigence**, un **exemple**, et — bientôt — un **test**.

Exemples, pas adjectifs

Remplace « rapide », « petit », « correct » par des **nombres** et des **exemples**. Les exemples révèlent les ambiguïtés (50 € inclus ou non ?).

Le concept

Classes d'équivalence

On ne peut pas tester toutes les entrées. On les range en **classes** dont tous les membres sont traités de la même manière, et on teste **un représentant par classe**.

Règle	Classes
frais selon le montant (seuil 50)	montant < 50 / montant ≥ 50
frais selon le poids (seuil 5 kg)	≤ 5 kg / > 5 kg
entrée invalide	montant négatif, poids ≤ 0, type incorrect

Valeurs limites

Les bugs se cachent aux **frontières** (< contre <=). Pour chaque seuil, teste : **juste en dessous**, **pile sur** le seuil, **juste au-dessus**.

Le concept

Valeurs limites (suite)

Seuil 50 €	Valeurs à tester
	49,99 · 50,00 · 50,01

Ajoute les cas **extrêmes** : zéro, vide, un seul élément, très grand.

Tests pilotés par table

Un même comportement, plusieurs exemples : une **table** (tuples) parcourue par une boucle. `unittest` fournit `subTest` : chaque ligne est rapportée séparément en cas d'échec.

```
CAS = [(49.99, 4.90), (50.00, 0.0), (50.01, 0.0)]
for montant, attendu in CAS:
    with self.subTest(montant=montant):
        self.assertEqual(frais(montant), attendu)
```

Le concept

Cas d'erreur

Une spécification dit aussi **ce qui est refusé** : exception levée (laquelle ?), message clair, rien d'écrit à moitié.

Propriétés

Au lieu d'exemples, on énonce une **propriété** vraie pour **toute** entrée, et on la vérifie sur beaucoup d'entrées aléatoires (graine fixée pour la reproductibilité) :

Fonction	Propriété
`trier`	le résultat est ordonné, de même taille, mêmes éléments
`encoder/decoder`	`decoder(encoder(x)) == x`
`prix_total`	croissant avec le nombre d'articles ; jamais négatif

Le concept

Propriétés (suite)

Cette approche (*property-based testing*) trouve des cas auxquels on n'avait pas pensé.

Matrice de comportements

Pour garder une vue d'ensemble, liste les comportements et le test qui les couvre :

Comportement	Test
Livraison gratuite dès 50 € inclus	`test_seuil`
Surcharge au-delà de 5 kg	`test_poids`
Entrée invalide refusée	`test_erreurs`

🧠 ***Astuce** : quand deux personnes ne sont pas d'accord sur une règle, demande-leur un **exemple chiffré**. Le désaccord apparaît en trente secondes.*

Exemple pas à pas

Problème : spécifier et tester la règle de livraison de HashBox : base 4,90 € ; gratuite dès 50 € de panier **inclus** ; supplément de 3 € si le colis dépasse 5 kg (même si la livraison est gratuite le supplément s'applique) ; entrées invalides refusées.

```
import io, random, unittest

def lancer(*classes):
    suite = unittest.TestSuite(unittest.defaultTestLoader.loadTestsFromTestCase(c)
    for c in classes)
    return unittest.TextTestRunner(stream=io.StringIO(), verbosity=0).run(suite)

def frais_livraison(montant, poids_kg):
    if not isinstance(montant, (int, float)) or isinstance(montant, bool) or monta
   nt < 0:
        raise ValueError("montant invalide")
    if not isinstance(poids_kg, (int, float)) or isinstance(poids_kg, bool) or poi
   ds_kg <= 0:
        raise ValueError("poids invalide")
```

Exemple pas à pas (suite)

```
    base = 0.0 if montant >= 50 else 4.90
    supplement = 3.0 if poids_kg > 5 else 0.0
    return round(base + supplement, 2)

class TestFraisLivraison(unittest.TestCase):
    def test_seuil_de_gratuite_valeurs_limites(self):
        for montant, attendu in [(0, 4.90), (49.99, 4.90), (50.00, 0.0), (50.01, 0
   .0), (500, 0.0)]:
            with self.subTest(montant=montant):
                self.assertEqual(frais_livraison(montant, 1), attendu)

    def test_supplement_poids_valeurs_limites(self):
        for poids, attendu in [(0.1, 4.90), (5.0, 4.90), (5.01, 7.90)]:
            with self.subTest(poids=poids):
                self.assertEqual(frais_livraison(30, poids), attendu)

    def test_supplement_meme_si_livraison_gratuite(self):
        self.assertEqual(frais_livraison(80, 6), 3.0)
```

Exemple pas à pas (suite)

```
    def test_entrees_invalides(self):
        for montant, poids in [(-1, 1), (10, 0), (10, -2), ("10", 1), (True, 1), (
    10, None)]:
            with self.subTest(montant=montant, poids=poids):
                with self.assertRaises(ValueError):
                    frais_livraison(montant, poids)

    def test_propriete_jamais_negatif_et_borne(self):
        rng = random.Random(42)                                  # graine fixée :
    reproductible
        for _ in range(1000):
            m, p = rng.uniform(0, 200), rng.uniform(0.01, 20)
            f = frais_livraison(m, p)
            self.assertGreaterEqual(f, 0)
            self.assertLessEqual(f, 7.90)

    def test_propriete_monotone_en_montant(self):
        rng = random.Random(7)
        for _ in range(1000):
```

Exemple pas à pas

Exemple pas à pas (suite)

```
            p = rng.uniform(0.01, 20)
            m1 = rng.uniform(0, 100)
            m2 = m1 + rng.uniform(0, 100)
            self.assertGreaterEqual(frais_livraison(m1, p), frais_livraison(m2, p)
    )       # payer plus ne coûte jamais plus de frais

resultat = lancer(TestFraisLivraison)
print("tests :", resultat.testsRun, "| réussite :", resultat.wasSuccessful())
assert resultat.wasSuccessful(), resultat.failures + resultat.errors

# --- le même ensemble détecte-t-il un bug de frontière ? ---
def frais_livraison(montant, poids_kg):           # bug : '>' au lieu de '>='
    base = 0.0 if montant > 50 else 4.90
    supplement = 3.0 if poids_kg > 5 else 0.0
    return round(base + supplement, 2)
resultat = lancer(TestFraisLivraison)
print("avec la frontière cassée :", len(resultat.failures), "échec(s)")
assert len(resultat.failures) >= 1
```

Exemple pas à pas (suite)

Lecture : la **table de valeurs limites** (49,99 / 50,00 / 50,01 ; 5,0 / 5,01) est ce qui attrape le bug `>` contre `>=`. Les deux **propriétés** couvrent 2 000 entrées aléatoires reproductibles. Les entrées invalides sont une partie **explicite** de la spécification.

À toi, avec aide

Voici la règle de tarification d'un billet : moins de 4 ans = gratuit ; de 4 à 11 ans inclus = 8 € ; de 12 à 64 ans inclus = 15 € ; 65 ans et plus = 10 €. Âge négatif ou supérieur à 130 = erreur.

Écris :

1. la table des **classes d'équivalence** et des **valeurs limites** ;
2. la fonction `prix_billet(age)` ;
3. les tests avec `subTest` couvrant toutes les limites (3, 4, 11, 12, 64, 65, 0, 130) et les erreurs (−1, 131).

Indices progressifs

Indice 1

Les limites sont 3/4, 11/12, 64/65 et les extrêmes 0/130.

Indice 2

Utilise une liste de tuples `(age, attendu)`.

Indice 3

Les erreurs sont testées à part avec `assertRaises`.

À toi, avec aide · réponse

Réponse

```
import io, unittest

def prix_billet(age):
    if age < 0 or age > 130:
        raise ValueError("âge hors plage")
    if age < 4:
        return 0
    if age <= 11:
        return 8
    if age <= 64:
        return 15
    return 10

class TestBillet(unittest.TestCase):
    def test_tarifs_et_limites(self):
        table = [(0, 0), (3, 0), (4, 8), (11, 8), (12, 15), (64, 15), (65, 10), (1
   30, 10)]
        for age, attendu in table:
            with self.subTest(age=age):
```

À toi, avec aide · réponse

Réponse (suite)

```
                self.assertEqual(prix_billet(age), attendu)

    def test_ages_invalides(self):
        for age in (-1, 131):
            with self.subTest(age=age):
                with self.assertRaises(ValueError):
                    prix_billet(age)

suite = unittest.defaultTestLoader.loadTestsFromTestCase(TestBillet)
resultat = unittest.TextTestRunner(stream=io.StringIO()).run(suite)
assert resultat.wasSuccessful()
print("classes : [0-3] [4-11] [12-64] [65-130] + invalides")
```

Les **limites** (3/4, 11/12, 64/65) sont exactement celles où < et <= se confondent.

À toi, en solo

Tests de propriétés sur un tri. Écris ton propre `fusionner(a, b)` qui fusionne deux listes **déjà triées** en une liste triée (sans utiliser `sorted`). Puis, sans aucun exemple chiffré, vérifie sur 2 000 paires aléatoires (graine fixée) ces propriétés :

1. le résultat est trié ;
2. sa longueur est `len(a) + len(b)` ;
3. il contient exactement les mêmes éléments (comparaison de multiensembles) ;
4. fusionner avec la liste vide renvoie l'autre liste ;
5. `fusionner(a, b) == fusionner(b, a)`.

Ajoute un bug (par exemple oublier le reste d'une liste) et montre que les propriétés le détectent.

À toi, en solo · réponse

Pistes de correction

```
import random
from collections import Counter

def fusionner(a, b):
    i = j = 0
    sortie = []
    while i < len(a) and j < len(b):
        if a[i] <= b[j]:
            sortie.append(a[i]); i += 1
        else:
            sortie.append(b[j]); j += 1
    sortie.extend(a[i:])
    sortie.extend(b[j:])
    return sortie

def proprietes(f, essais=2000, graine=3):
    rng = random.Random(graine)
    for _ in range(essais):
        a = sorted(rng.randint(-20, 20) for _ in range(rng.randint(0, 8)))
```

Pistes de correction (suite)

```
        b = sorted(rng.randint(-20, 20) for _ in range(rng.randint(0, 8)))
        r = f(a, b)
        assert r == sorted(r), (a, b, r)
        assert len(r) == len(a) + len(b), (a, b, r)
        assert Counter(r) == Counter(a) + Counter(b), (a, b, r)
        assert f(a, []) == a and f([], b) == b
        assert f(a, b) == f(b, a)

proprietes(fusionner)
print("5 propriétés vérifiées sur 2000 paires")

def fusionner_bug(a, b):                          # oublie les restes
    i = j = 0
    sortie = []
    while i < len(a) and j < len(b):
        if a[i] <= b[j]:
            sortie.append(a[i]); i += 1
        else:
            sortie.append(b[j]); j += 1
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    return sortie

try:
    proprietes(fusionner_bug)
    raise SystemExit("le bug n'a pas été détecté !")
except AssertionError as e:
    print("bug détecté sur :", e.args[0][:2])
```

Les propriétés trouvent le bug sans que tu aies eu à imaginer le cas qui le révèle.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Spécification floue (« petit », « rapide »)	Chacun comprend autre chose	Remplacer les adjectifs par des nombres et des exemples
Oublier les valeurs limites	Les bugs vivent aux frontières	Tester juste en dessous, pile, juste au-dessus
Ne tester que les cas heureux	Les erreurs sont aussi un comportement	Spécifier et tester les entrées invalides
Aléatoire sans graine	Test qui échoue « parfois »	Fixer la graine, afficher le cas fautif

Mini-boss

Mini-boss

Spécification complète d'un code promo. Règles : le code est composé de 3 à 12 caractères alphanumériques majuscules ; `HASH10` = −10 % ; `BIENVENUE` = −5 € ; un code expiré (date) est refusé ; un code ne se cumule pas avec un autre ; sur un panier de 0 €, aucun code n'est appliqué ; la remise ne rend jamais le total négatif.

1. Écris la **matrice de comportements** (au moins 12 lignes, avec exemples chiffrés) ;
2. liste les classes d'équivalence et les limites (longueur 2/3/12/13, date d'expiration veille/jour/lendemain) ;
3. implémente `appliquer_code(total_cents, code, aujourdhui, expirations)` en passant la date en paramètre ;
4. écris les tests pilotés par table et au moins 2 propriétés ;
5. détecte 4 bugs injectés.

Mini-boss · réponse

Pistes de correction

Un bon travail : une table de 12+ comportements avec valeurs chiffrées (par exemple `("HASH10", 10000) → 9000`, `("BIENVENUE", 300) → 0`), la date injectée (jamais `date.today()` dans la fonction : mission 9), les limites de longueur (« AB » refusé, « ABC » accepté, 12 caractères accepté, 13 refusé), l'expiration (`expiration < aujourdhui` refusé, égale acceptée — à décider **avec un exemple** puis noter la décision). Propriétés : le total après remise est toujours entre 0 et le total initial ; appliquer un code invalide ne change jamais le total.

Quiz éclair

Quiz éclair

1. Une valeur limite est :

A. une valeur extrêmement grande seulement
B. une valeur choisie au hasard
C. une valeur située juste autour d'un seuil de la règle

2. Un test piloté par table avec `subTest` permet de :

A. rapporter séparément chaque ligne qui échoue
B. accélérer le programme
C. éviter d'écrire des assertions

3. Une propriété de test est :

A. un exemple unique

Quiz éclair (suite)

B. un énoncé vrai pour toute entrée, vérifié sur beaucoup d'entrées
C. une variable privée

4. 🔁 *Révision mission 01* : Un test automatisé sert surtout à :

A. rendre le code plus rapide
B. remplacer la documentation du projet
C. rejouer une vérification à l'identique après chaque modification

5. 🔁 *Révision mission 01* : Un test qui n'a jamais été rouge :

A. n'a pas prouvé qu'il sait détecter un bug
B. est forcément parfait
C. est inutile à exécuter

Quiz éclair · réponse

Réponses

1. **C**.
2. **A**.
3. **B**.
4. **C**.
5. **A**.

Je réfléchis

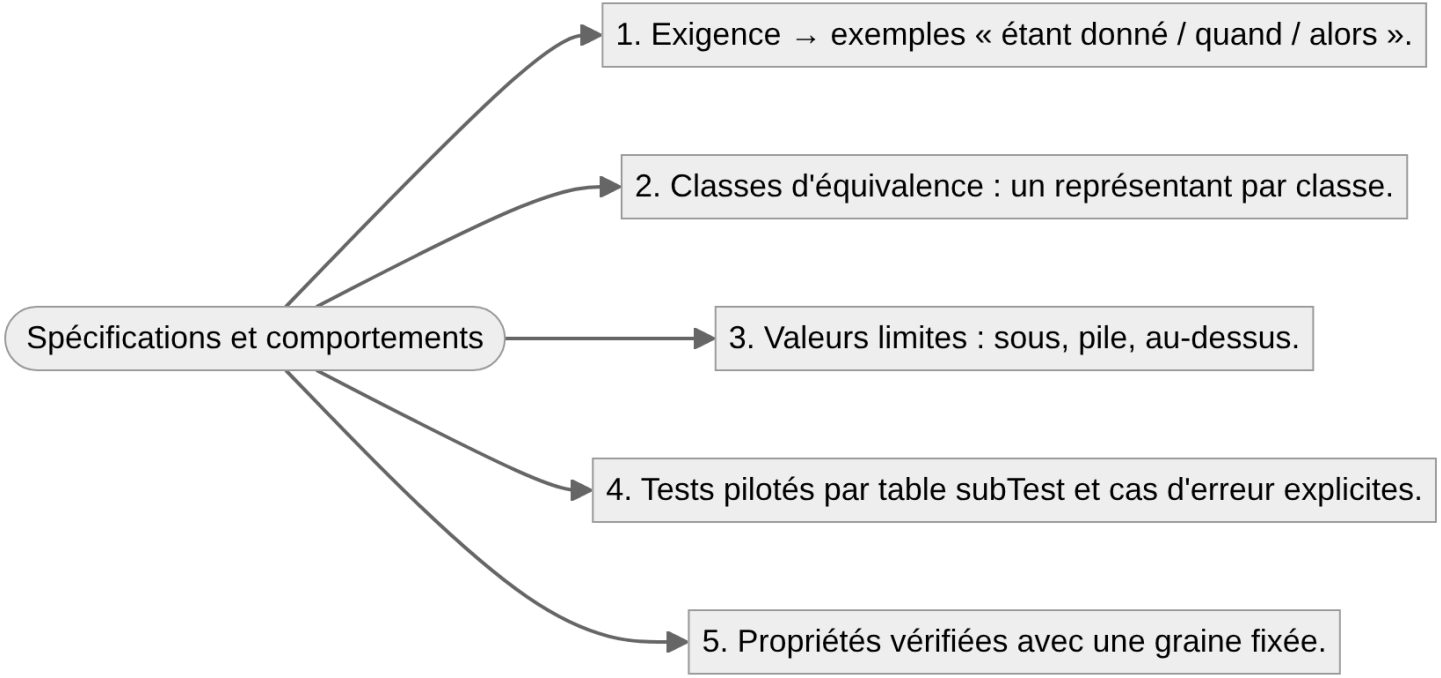
1. Quelle règle de ton quotidien professionnel est floue au point que deux collègues la comprennent différemment ?
2. Pourquoi un exemple chiffré lève-t-il les ambiguïtés plus vite qu'une définition ?

À retenir

1. Exigence → exemples « étant donné / quand / alors ».
2. Classes d'équivalence : un représentant par classe.
3. Valeurs limites : sous, pile, au-dessus.
4. Tests pilotés par table (`subTest`) et cas d'erreur explicites.
5. Propriétés vérifiées avec une graine fixée.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Traducteur de besoins

Prochaine mission : Tests unitaires

