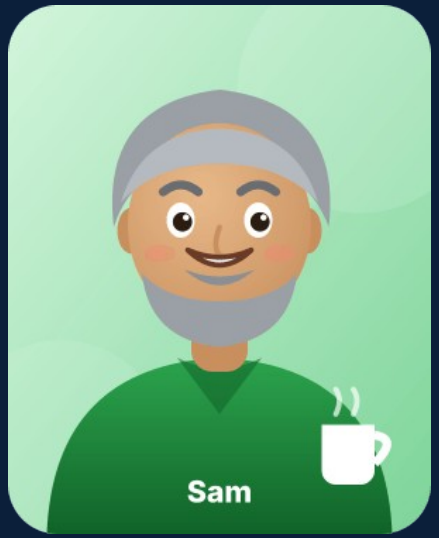
HEF-110 · MISSION 03 / 10

Tests unitaires

Un bon test unitaire est petit, rapide, indépendant, et il échoue pour une seule raison que l'on comprend tout de suite.





Badge : Orfèvre du test

100 min · intermédiaire

L'histoire

La suite de tests de HashBox compte 300 tests. Un matin, un seul changement fait échouer **47 tests**. Personne ne sait lequel regarder. Certains échouent seulement quand on les lance après d'autres. D'autres portent des noms comme `test1` ou `test_ok`.



> *« Une suite de tests est du code, dit Sam. Elle doit être aussi **lisible et maintenable** que le reste. Un bon test unitaire, c'est le contraire d'un cas particulier : c'est une phrase claire, une cause, une raison d'échouer. »*

Tu apprends à écrire des tests unitaires de qualité professionnelle, à mesurer ce qu'ils couvrent, et à reconnaître les mauvaises odeurs.

Ta mission

À la fin, je serai capable de…

1. structurer un test (préparer, agir, vérifier) et le nommer par comportement
2. utiliser `setUp`/`tearDown`, `assertRaises`, `assertAlmostEqual`
3. garantir l'indépendance et le déterminisme des tests
4. mesurer la couverture de lignes et en comprendre les limites
5. reconnaître et corriger les mauvaises odeurs de tests

Défi d'échauffement

Compare deux noms de test : `test1` et `test_un_panier_vide_a_un_total_de_zero`. Si l'un des deux échoue à 3 h du matin dans un pipeline, lequel te dit immédiatement ce qui ne va pas ? Écris trois autres noms de test pour un `Panier` selon ce modèle.

Le concept

Ce qu'est une « unité »

Une **unité** est la plus petite partie du code qui a un comportement observable : une fonction, une méthode, une petite classe. Un **test unitaire** la vérifie **isolée**, sans réseau, sans base de données, sans disque lent, sans horloge réelle.

Les qualités d'un bon test (F.I.R.S.T.)

Lettre	Qualité	Signification
F	*Fast*	rapide : des millisecondes
I	*Independent*	indépendant : l'ordre d'exécution ne change rien
R	*Repeatable*	reproductible : même résultat à chaque exécution et partout
S	*Self-validating*	auto-vérifiant : réussi ou échoué, sans interprétation humaine

Le concept

Les qualités d'un bon test (F.I.R.S.T.) (suite)

Lettre	Qualité	Signification
T	*Timely*	écrit au bon moment : avec (ou avant) le code

Nommer et structurer

- **Nom = comportement** : `test_un_code_expire_est_refuse`, pas `test_code_2`.
- **Une idée par test** : si le nom contient « et », ce sont deux tests.
- **Préparer / agir / vérifier** séparés par des lignes vides.
- Les valeurs du test sont **visibles dans le test** (pas cachées dans trois fichiers).

Fixtures : `setUp` et `tearDown`

`setUp` s'exécute **avant chaque test** pour préparer un état neuf ; `tearDown` nettoie après. Chaque test repart donc d'un état propre : c'est la clé de l'indépendance.

Le concept

Assertions utiles

Assertion	Usage
`assertEqual(a, b)`	égalité (message clair avec les deux valeurs)
`assertAlmostEqual(a, b, places=2)`	flottants
`assertIn(x, conteneur)`	appartenance
`assertRaises(Erreur)` + `assertRaisesRegex`	exception attendue
`assertIsNone`, `assertIsInstance`	état et type
`assertCountEqual(a, b)`	mêmes éléments, ordre indifférent

Le concept

Tester le comportement, pas l'implémentation

Test fragile (implémentation)	Test robuste (comportement)
vérifie qu'une fonction privée a été appelée 3 fois	vérifie le résultat final observable
lit les attributs internes `_items`	passe par l'interface publique

Si tu peux **réorganiser le code sans changer le comportement** et que des tests cassent, ils testent l'implémentation.

La couverture

La **couverture de lignes** indique la part des lignes exécutées par les tests.

- Elle **détecte ce qui n'est jamais exécuté** (utile).

Le concept

La couverture (suite)

- Elle **ne dit pas** si le code est bien vérifié : une ligne exécutée sans assertion est « couverte ».
- Viser 100 % par principe produit des tests creux ; on vise la **couverture des comportements importants**, avec un seuil raisonnable (par exemple 80 %) comme alerte, pas comme but.

On peut la mesurer avec l'outil `coverage` ; en Python pur, `sys.settrace` suffit à comprendre le principe (voir l'exemple).

Mauvaises odeurs de tests

Odeur	Problème	Remède
Test géant	fait vingt choses	un comportement par test
Dépendance d'ordre	passe seul, échoue en suite	`setUp` neuf, pas d'état global
Aléatoire/horloge	échecs intermittents	graine fixée, horloge injectée

Le concept

Mauvaises odeurs de tests (suite)

Odeur	Problème	Remède
Assertion vague (`assertTrue(x)`)	échec peu informatif	`assertEqual` avec la valeur attendue
Logique dans le test (boucles, `if`)	le test peut être faux	valeurs en dur, tables
Copier-coller massif	maintenance pénible	table, helpers lisibles

🧠 ***Astuce*** *: un test qui échoue doit répondre en 10 secondes à : quel comportement est cassé, avec quelle entrée, quelle sortie attendue et obtenue ?*

Exemple pas à pas

Problème : tester un `Panier` proprement, vérifier l'indépendance des tests, et mesurer la couverture de lignes avec un mini-outil.

```
import io, sys, unittest

class Panier:
    def __init__(self):
        self._lignes = {}                                   # code -> (prix_cents,
    quantite)

    def ajouter(self, code, prix_cents, quantite=1):
        if prix_cents < 0 or quantite <= 0:
            raise ValueError("prix ou quantité invalide")
        ancien = self._lignes.get(code, (prix_cents, 0))
        self._lignes[code] = (prix_cents, ancien[1] + quantite)

    def retirer(self, code):
        if code not in self._lignes:
```

Exemple pas à pas

Exemple pas à pas (suite)

```
            raise KeyError(code)
        del self._lignes[code]

    def total(self):
        return sum(prix * qte for prix, qte in self._lignes.values())

    def nombre_articles(self):
        return sum(qte for _, qte in self._lignes.values())

class TestPanier(unittest.TestCase):
    def setUp(self):
        self.panier = Panier()                              # état neuf avant CHAQ
   UE test

    def test_un_panier_vide_a_un_total_de_zero(self):
        self.assertEqual(self.panier.total(), 0)

    def test_le_total_additionne_prix_fois_quantite(self):
        self.panier.ajouter("HEF-010", 4000, 2)
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        self.panier.ajouter("HEF-020", 1500)
        self.assertEqual(self.panier.total(), 9500)

    def test_ajouter_deux_fois_le_meme_cours_cumule_les_quantites(self):
        self.panier.ajouter("HEF-010", 4000)
        self.panier.ajouter("HEF-010", 4000)
        self.assertEqual(self.panier.nombre_articles(), 2)

    def test_retirer_supprime_la_ligne(self):
        self.panier.ajouter("HEF-010", 4000)
        self.panier.retirer("HEF-010")
        self.assertEqual(self.panier.total(), 0)

    def test_retirer_un_cours_absent_leve_keyerror(self):
        with self.assertRaises(KeyError):
            self.panier.retirer("INCONNU")

    def test_un_prix_negatif_est_refuse(self):
        with self.assertRaisesRegex(ValueError, "invalide"):
```

Exemple pas à pas (suite)

```
            self.panier.ajouter("HEF-010", -1)

def suite():
    return unittest.defaultTestLoader.loadTestsFromTestCase(TestPanier)

# ---- indépendance : mélangeons l'ordre des tests ----
import random
tests = list(suite())
random.Random(1).shuffle(tests)
resultat = unittest.TextTestRunner(stream=io.StringIO()).run(unittest.TestSuite(te
   sts))
assert resultat.wasSuccessful(), "les tests dépendent de leur ordre !"
print("ordre mélangé : toujours vert")

# ---- mini-couverture de lignes avec sys.settrace ----
def mesurer(fonction):
    code_cible = {Panier.ajouter.__code__, Panier.retirer.__code__, Panier.total._
   _code__, Panier.nombre_articles.__code__}
    vues = set()
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    def traceur(frame, evenement, arg):
        if frame.f_code in code_cible:
            if evenement == "line":
                vues.add((frame.f_code.co_name, frame.f_lineno))
            return traceur
    sys.settrace(traceur)
    try:
        fonction()
    finally:
        sys.settrace(None)
    toutes = set()
    for c in code_cible:
        toutes |= {(c.co_name, ligne) for _, _, ligne in c.co_lines() if ligne is
   not None and ligne != c.co_firstlineno}
    return vues, toutes

vues, toutes = mesurer(lambda: unittest.TextTestRunner(stream=io.StringIO()).run(s
   uite()))
print(f"couverture des lignes de Panier : {len(vues)}/{len(toutes)} = {len(vues) /
```

Exemple pas à pas (suite)

```
    len(toutes):.0%}")
assert len(vues) / len(toutes) >= 0.95
```

Lecture :

1. Chaque test porte un **nom de comportement** ; une panne se lit comme une phrase.
2. `setUp` crée un panier **neuf** : mélanger l'ordre ne change rien (indépendance).
3. Les tests passent par l'**interface publique**, jamais par `_lignes`.
4. Le mini-outil de couverture compte les lignes exécutées par la suite. Les vrais outils (`coverage`) font la même chose avec plus de confort et des rapports.

La couverture de 100 % ici ne prouve pas l'absence de bug : elle dit seulement que **chaque ligne a été exécutée**. Si on supprimait toutes les assertions, elle resterait à 100 %.

À toi, avec aide

Voici une suite de tests de mauvaise qualité pour la classe `Panier` de l'exemple :

```
def test1(self):
    p = Panier()
    p.ajouter("A", 100, 2)
    p.ajouter("B", 50)
    p.retirer("B")
    assert p.total() >= 0
    self.assertTrue(p._lignes)
    for c in ["A", "B"]:
        if c in p._lignes:
            print(c)
```

Fais la liste des **mauvaises odeurs**, puis réécris en trois bons tests.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Cherche : nom vague, plusieurs comportements, assertion vague (`>= 0`), accès à `_lignes`, logique et `print` dans le test.

Indice 2

Un test par comportement : total après ajout, retrait, quantité.

Indice 3

Utilise des valeurs précises pour `assertEqual`.

À toi, avec aide · réponse

Réponse

Mauvaises odeurs : nom `test1` sans sens ; plusieurs comportements dans un seul test ; `assert p.total() >= 0` ne détecterait presque aucun bug ; `assertTrue(p._lignes)` teste l'implémentation interne ; une boucle avec `if` et un `print` n'ont rien à faire dans un test.

```
import io, unittest

class Panier:
    def __init__(self):
        self._lignes = {}
    def ajouter(self, code, prix_cents, quantite=1):
        ancien = self._lignes.get(code, (prix_cents, 0))
        self._lignes[code] = (prix_cents, ancien[1] + quantite)
    def retirer(self, code):
        del self._lignes[code]
    def total(self):
        return sum(p * q for p, q in self._lignes.values())

class TestPanierPropre(unittest.TestCase):
```

À toi, avec aide · réponse

Réponse (suite)

```
    def setUp(self):
        self.p = Panier()

    def test_le_total_tient_compte_des_quantites(self):
        self.p.ajouter("A", 100, 2)
        self.p.ajouter("B", 50)
        self.assertEqual(self.p.total(), 250)

    def test_retirer_une_ligne_baisse_le_total_de_son_montant(self):
        self.p.ajouter("A", 100, 2)
        self.p.ajouter("B", 50)
        self.p.retirer("B")
        self.assertEqual(self.p.total(), 200)

    def test_retirer_toutes_les_lignes_donne_zero(self):
        self.p.ajouter("A", 100)
        self.p.retirer("A")
        self.assertEqual(self.p.total(), 0)
```

À toi, avec aide · réponse

Réponse (suite)

```
resultat = unittest.TextTestRunner(stream=io.StringIO()).run(unittest.defaultTestL
   oader.loadTestsFromTestCase(TestPanierPropre))
assert resultat.wasSuccessful()
print(resultat.testsRun, "tests propres")
```

Chaque test a un nom qui est une phrase, une seule raison d'échouer, des valeurs exactes.

À toi, en solo

Écris la suite complète d'une classe `Pile` bornée (capacité fixée : `empiler`, `depiler`, `sommet`, `est_vide`, `est_pleine`, `taille`) en respectant F.I.R.S.T. :

1. au moins 12 tests, nommés par comportement, avec `setUp` ;
2. couvre : pile vide, pleine, dépassement (`OverflowError`), dépilement d'une pile vide (`IndexError`), ordre LIFO, capacité 1, capacité invalide (≤ 0) ;
3. prouve l'indépendance en mélangeant l'ordre des tests ;
4. mesure la couverture de lignes avec `sys.settrace` ;
5. injecte 3 bugs (mauvaise borne de capacité, ordre FIFO au lieu de LIFO, `taille` fausse après dépilement) et montre que la suite les détecte.

À toi, en solo · réponse

Pistes de correction

```
import io, random, sys, unittest

class Pile:
    def __init__(self, capacite):
        if capacite <= 0:
            raise ValueError("capacité invalide")
        self._capacite, self._elements = capacite, []
    def empiler(self, x):
        if self.est_pleine():
            raise OverflowError("pile pleine")
        self._elements.append(x)
    def depiler(self):
        if self.est_vide():
            raise IndexError("pile vide")
        return self._elements.pop()
    def sommet(self):
        if self.est_vide():
            raise IndexError("pile vide")
        return self._elements[-1]
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    def est_vide(self):
        return not self._elements
    def est_pleine(self):
        return len(self._elements) == self._capacite
    def taille(self):
        return len(self._elements)

class TestPile(unittest.TestCase):
    def setUp(self):
        self.p = Pile(3)
    def test_une_pile_neuve_est_vide(self):
        self.assertTrue(self.p.est_vide()); self.assertEqual(self.p.taille(), 0)
    def test_empiler_augmente_la_taille(self):
        self.p.empiler(1); self.assertEqual(self.p.taille(), 1); self.assertFalse(
   self.p.est_vide())
    def test_le_dernier_empile_est_le_premier_depile(self):
        for x in (1, 2, 3): self.p.empiler(x)
        self.assertEqual([self.p.depiler() for _ in range(3)], [3, 2, 1])
    def test_le_sommet_ne_retire_rien(self):
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        self.p.empiler(9); self.assertEqual(self.p.sommet(), 9); self.assertEqual(
    self.p.taille(), 1)
     def test_une_pile_pleine_refuse_un_element_de_plus(self):
        for x in range(3): self.p.empiler(x)
        self.assertTrue(self.p.est_pleine())
        with self.assertRaises(OverflowError): self.p.empiler(99)
     def test_depiler_une_pile_vide_leve_indexerror(self):
        with self.assertRaises(IndexError): self.p.depiler()
     def test_sommet_d_une_pile_vide_leve_indexerror(self):
        with self.assertRaises(IndexError): self.p.sommet()
     def test_la_taille_baisse_apres_depilement(self):
        self.p.empiler(1); self.p.empiler(2); self.p.depiler()
        self.assertEqual(self.p.taille(), 1)
     def test_capacite_un(self):
        p = Pile(1); p.empiler("x")
        self.assertTrue(p.est_pleine())
        with self.assertRaises(OverflowError): p.empiler("y")
     def test_capacite_invalide(self):
        for c in (0, -1):
```

À toi, en solo · réponse

Pistes de correction (suite)

```
            with self.assertRaises(ValueError): Pile(c)
    def test_une_pile_pleine_depilee_n_est_plus_pleine(self):
        for x in range(3): self.p.empiler(x)
        self.p.depiler(); self.assertFalse(self.p.est_pleine())
    def test_on_peut_reutiliser_la_pile_apres_l_avoir_videe(self):
        self.p.empiler(1); self.p.depiler(); self.p.empiler(2)
        self.assertEqual(self.p.sommet(), 2)

def lancer(ordre=None):
    tests = list(unittest.defaultTestLoader.loadTestsFromTestCase(TestPile))
    if ordre is not None:
        random.Random(ordre).shuffle(tests)
    return unittest.TextTestRunner(stream=io.StringIO()).run(unittest.TestSuite(te
   sts))

assert lancer().wasSuccessful() and lancer(5).wasSuccessful()
print("12 tests verts, indépendants de l'ordre")

# couverture de lignes
```

À toi, en solo · réponse

Pistes de correction (suite)

```
cibles = {m.__code__ for m in (Pile.empiler, Pile.depiler, Pile.sommet, Pile.est_v
   ide, Pile.est_pleine, Pile.taille, Pile.__init__)}
vues = set()
def traceur(frame, ev, arg):
    if frame.f_code in cibles:
        if ev == "line": vues.add((frame.f_code.co_name, frame.f_lineno))
        return traceur
sys.settrace(traceur); lancer(); sys.settrace(None)
toutes = {(c.co_name, l) for c in cibles for _, _, l in c.co_lines() if l is not N
   one and l != c.co_firstlineno}
print(f"couverture {len(vues)}/{len(toutes)}")
assert len(vues) == len(toutes)

# bugs injectés
originaux = (Pile.est_pleine, Pile.depiler, Pile.taille)
Pile.est_pleine = lambda self: len(self._elements) > self._capacite
assert not lancer().wasSuccessful(); Pile.est_pleine = originaux[0]
Pile.depiler = lambda self: self._elements.pop(0)
assert not lancer().wasSuccessful(); Pile.depiler = originaux[1]
```

À toi, en solo · réponse

Pistes de correction (suite)

```
Pile.taille = lambda self: len(self._elements) + 1
assert not lancer().wasSuccessful(); Pile.taille = originaux[2]
assert lancer().wasSuccessful()
print("3 bugs injectés, 3 détectés")
```

Cette suite est courte, rapide et chaque nom dit un comportement : en cas d'échec, le diagnostic est immédiat.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Test géant	Impossible de savoir ce qui casse	Un comportement par test, nom explicite
Dépendances entre tests	Passe seul, échoue en suite	`setUp` neuf, aucun état partagé
Viser 100 % de couverture par principe	Tests creux sans assertion	Viser les comportements importants ; la couverture sert d'alerte
Tester les détails internes	Casse à chaque refactoring	Passer par l'interface publique

Mini-boss

Mini-boss

Assainir une suite malade. On te donne une classe `Inscription` (inscrire un membre à un cours avec capacité maximale, liste d'attente, désinscription qui promeut le premier en attente) et une suite de 6 tests « malades » : noms `test_a`, dépendance d'ordre via une variable globale, assertions `assertTrue(x)`, test de 40 lignes, heure réelle, accès à `_attente`.

1. Écris la classe et la suite malade ;
2. liste les mauvaises odeurs de chacune (tableau) ;
3. réécris une suite saine de 12+ tests selon F.I.R.S.T. ;
4. mesure la couverture avant/après ;
5. injecte 5 bugs (promotion dans le mauvais ordre, dépassement de capacité, désinscription d'un absent, doublon, attente non vidée) : la suite malade en détecte combien ? la saine ?

Mini-boss · réponse

Pistes de correction

Un bon travail : tableau d'odeurs (nom, ordre, assertions vagues, test géant, horloge, détails internes) ; suite saine avec `setUp`, valeurs en dur, noms comme `test_desinscrire_promeut_le_premier_de_la_liste_d_attente` ; comparaison chiffrée (par exemple malade 2/5 bugs détectés, saine 5/5). Le point de réflexion attendu : la couverture de lignes de la suite malade pouvait être proche de la saine, alors que sa **capacité de détection** est bien plus faible — la couverture mesure l'exécution, pas la vérification.

Quiz éclair

Quiz éclair

1. F.I.R.S.T. désigne les qualités d'un test :

A. fort, intuitif, riche, simple, testable
B. rapide, indépendant, reproductible, auto-vérifiant, écrit à temps
C. fiable, interne, résilient, sûr, tracé

2. `setUp` sert à :

A. arrêter les tests
B. mesurer la couverture
C. préparer un état neuf avant chaque test

3. Une couverture de 100 % signifie :

A. que chaque ligne a été exécutée, pas forcément bien vérifiée

Quiz éclair

Quiz éclair (suite)

B. que le code n'a aucun bug
C. que les tests sont rapides

4. 🔁 *Révision mission 02* : Une valeur limite est :

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

5. 🔁 *Révision mission 02* : Une propriété de test est :

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

Quiz éclair · réponse

Réponses

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

Je réfléchis

Je réfléchis

1. Dans ton code actuel, quel test aurait besoin d'un meilleur nom ?
2. Pourquoi une couverture élevée peut-elle donner une fausse confiance ?

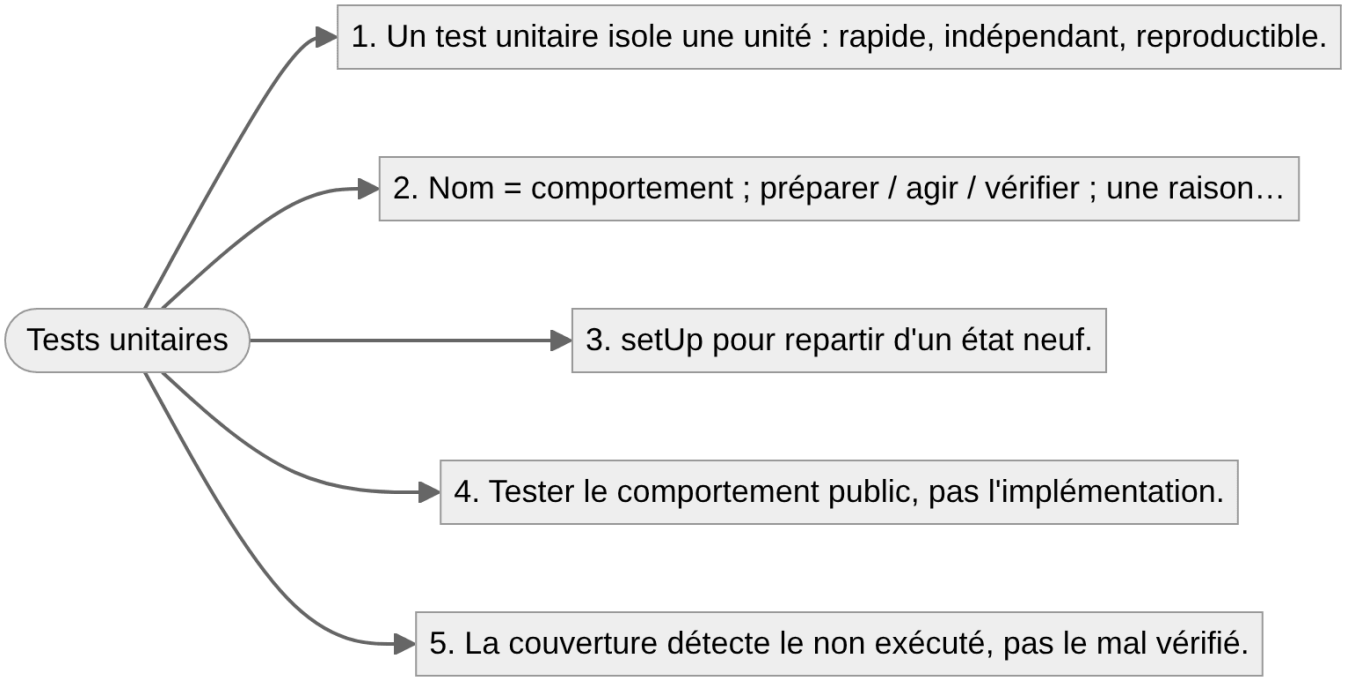
À retenir

À retenir

1. Un test unitaire isole une unité : rapide, indépendant, reproductible.
2. Nom = comportement ; préparer / agir / vérifier ; une raison d'échouer.
3. `setUp` pour repartir d'un état neuf.
4. Tester le comportement public, pas l'implémentation.
5. La couverture détecte le non exécuté, pas le mal vérifié.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Orfèvre du test

Prochaine mission : Tests d'intégration

