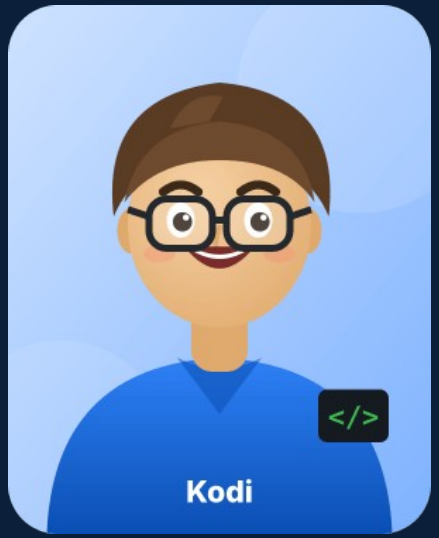
HEF-110 · MISSION 09 / 10

Flakiness et qualité des suites

Un test qui échoue parfois sans raison est pire qu'un test absent : il apprend à toute l'équipe à ignorer le rouge.



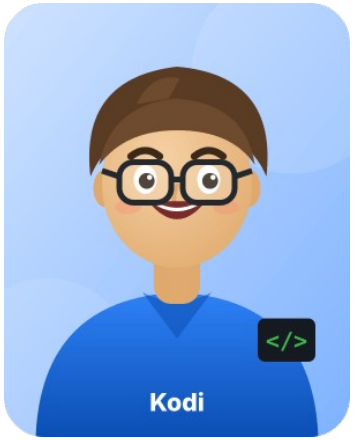


Badge : Chasseur de tests instables

100 min · intermédiaire

L'histoire

Chez HashBox, le pipeline est rouge un jour sur cinq. Les développeurs ont pris l'habitude de cliquer « relancer ». Un mardi, le pipeline est rouge pour de bon : un vrai bug. Personne ne le remarque pendant deux jours, tout le monde pensant à « un test instable ».



> *« C'est la pire conséquence, dit Maya. Quand le rouge ne veut plus rien dire, la suite de tests ne protège plus personne. Et ce n'est pas une fatalité : l'instabilité a des causes précises, qu'on sait chasser. »*

Tu apprends à reconnaître les causes d'instabilité, à les détecter par la méthode, à les corriger, et à mesurer la vraie qualité d'une suite avec le test de mutation.

Ta mission

À la fin, je serai capable de…

1. lister les causes classiques d'instabilité (temps, hasard, ordre, concurrence, réseau, état partagé)
2. détecter un test instable en le rejouant dans des ordres et conditions variés
3. corriger une instabilité par injection, graine fixée, isolation et attente explicite
4. mesurer la qualité d'une suite avec le test de mutation
5. gérer les tests instables (quarantaine) sans les laisser s'installer

Défi d'échauffement

Défi d'échauffement

Une balance de cuisine affiche 250 g, puis 247 g, puis 253 g pour le même sac de farine. Pourrais-tu t'en servir pour une recette précise ? Compare avec un test qui passe 9 fois sur 10 : qu'apprend-il à ceux qui voient le rouge ?

Le concept

Un test instable (*flaky*)

Un test est **instable** s'il peut passer **ou** échouer sur le **même code**, sans modification. Conséquences : confiance perdue, relances inutiles, vrais bugs masqués, pipeline ralenti.

Les causes classiques

Cause	Exemple	Remède
Temps	« il est entre 9 h et 18 h » ; durée mesurée ; expiration	horloge injectée, durées larges ou simulées
Hasard	`random` sans graine	générateur injecté, graine fixée, afficher la graine
Ordre	un test suppose que le précédent a créé des données	isolation complète (mission 3)

Le concept

Les causes classiques (suite)

Cause	Exemple	Remède
État partagé	variable globale, fichier, base commune	créer/détruire dans chaque test
Concurrence	threads dont l'ordre varie	synchronisation explicite, attentes sur conditions
Attentes fixes	`sleep(1)` en espérant que ce soit prêt	attente explicite (mission 6)
Environnement	port déjà pris, fuseau horaire, locale, système de fichiers	ports éphémères, configuration explicite
Dépendance externe	réseau, API tierce	double, ou environnement contrôlé
Ordre des collections	dictionnaires/ensembles non ordonnés comparés par texte	comparer des ensembles ou trier

Le concept

Détecter : rejouer pour révéler

Un test instable se révèle en **variant** les conditions :

1. **Rejouer N fois** le même test ;
2. **Mélanger l'ordre** des tests ;
3. **Varier la graine** du hasard ;
4. **Exécuter en parallèle** ou sous charge ;
5. **Changer l'environnement** (fuseau, locale, vitesse).

Un test est instable si, sur le **même code**, il réussit parfois et échoue parfois. Son **taux d'échec** (par exemple 3/30) le classe par urgence.

Le concept

Gérer : la quarantaine

Un test instable ne doit pas bloquer toute l'équipe, ni être ignoré :

1. **le sortir** de la voie bloquante (quarantaine), avec une étiquette, un responsable et une échéance ;
2. **continuer à l'exécuter** (pour mesurer) dans une voie non bloquante ;
3. **le corriger ou le supprimer** — un test en quarantaine qui traîne est une dette.

Jamais de « relancer automatiquement jusqu'au vert » sans trace : ça cache le problème.

La qualité d'une suite : au-delà de la couverture

La couverture dit ce qui est **exécuté**. Le **test de mutation** mesure ce qui est **vérifié** :

Le concept

La qualité d'une suite : au-delà de la couverture (suite)

1. on crée des **mutants** : copies du code avec une petite altération (`>=` devient `>`, `*` devient `/`, `3` devient `4`) ;
2. on lance la suite sur chaque mutant ;
3. un mutant **tué** = au moins un test échoue ; un mutant **survivant** = la suite n'a rien remarqué ;
4. **score de mutation** = mutants tués / mutants total.

Un mutant survivant désigne un test manquant ou trop vague. Certains survivants sont des **mutants équivalents** (la modification ne change pas le comportement) : on les écarte à la main.

Mesure	Ce qu'elle dit
Couverture de lignes	quelles lignes sont exécutées
Couverture des branches	quelles décisions ont pris les deux chemins
Score de mutation	si les tests détectent réellement des défauts

Le concept

La qualité d'une suite : au-delà de la couverture (suite)

Mesure	Ce qu'elle dit
Taux d'instabilité	à quel point on peut leur faire confiance
Durée de la suite	à quelle vitesse on a un retour

Entretenir la suite

- **Supprimer** les tests redondants ou sans valeur ;
- **Accélérer** : repérer les 10 tests les plus lents ;
- **Lire les échecs** : message clair, valeurs attendues et obtenues ;
- **Revoir les tests comme du code** : une suite est un produit.

🧠 ***Astuce*** *: un test instable est un bug* ***dans le test ou dans le code*** *(courses, états cachés). Ne conclus jamais « c'est juste instable » sans avoir cherché laquelle des deux causes c'est.*

Exemple pas à pas

Problème : (1) détecter des tests instables en variant ordre et graine, (2) les corriger, (3) montrer avec un mini test de mutation qu'une couverture de 100 % peut cacher une suite faible.

```
import ast, io, random, sys, unittest
from datetime import datetime

# ================ 1. une classe de tests FRAGILE ================
COMPTEUR = {"n": 0}

def prochain_id():
    COMPTEUR["n"] += 1
    return COMPTEUR["n"]

def tirage():
    return random.randint(1, 6)

def est_heure_ouvrable():
    return 9 <= datetime.now().hour < 18              # dépend de l'heure réelle
```

Exemple pas à pas

Exemple pas à pas (suite)

```
class TestFragile(unittest.TestCase):
    def test_premier_identifiant_vaut_un(self):
        self.assertEqual(prochain_id(), 1)                # ❌ dépend de l'ordre (
   état global)
    def test_deux_identifiants_successifs(self):
        a = prochain_id(); b = prochain_id()
        self.assertEqual(b, a + 1)
    def test_le_de_ne_donne_pas_six(self):
        self.assertLess(tirage(), 6)                       # ❌ dépend du hasard (
   échoue quand on tire 6)
    def test_stable(self):
        self.assertEqual(sorted([3, 1, 2]), [1, 2, 3])

def detecter(classe, essais=40):
    echecs = {}
    for essai in range(essais):
        random.seed(essai)                                 # on VARIE la graine…
        COMPTEUR["n"] = 0
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        tests = list(unittest.defaultTestLoader.loadTestsFromTestCase(classe))
        random.Random(essai).shuffle(tests)                # …et l'ordre d'exécut
   ion
        resultat = unittest.TextTestRunner(stream=io.StringIO()).run(unittest.Test
   Suite(tests))
        for test, _ in resultat.failures + resultat.errors:
            echecs[test._testMethodName] = echecs.get(test._testMethodName, 0) + 1
    return echecs

taux = detecter(TestFragile)
instables = {nom: n for nom, n in taux.items() if 0 < n < 40}
print("tests instables détectés :", {nom: f"{n}/40" for nom, n in sorted(instables
   .items())})
assert "test_premier_identifiant_vaut_un" in instables and "test_le_de_ne_donne_pa
   s_six" in instables
assert "test_stable" not in taux

# ================ 2. la version CORRIGÉE ================
class Generateur:
```

Exemple pas à pas (suite)

```
    def __init__(self): self.n = 0
    def prochain(self):
        self.n += 1; return self.n

def tirage_corrige(rng):
    return rng.randint(1, 6)

def est_heure_ouvrable_corrigee(maintenant):
    return 9 <= maintenant.hour < 18

class TestStable(unittest.TestCase):
    def setUp(self):
        self.g = Generateur()                                  # état neuf à chaque t
   est
    def test_premier_identifiant_vaut_un(self):
        self.assertEqual(self.g.prochain(), 1)
    def test_deux_identifiants_successifs(self):
        a = self.g.prochain(); b = self.g.prochain()
        self.assertEqual(b, a + 1)
```

Exemple pas à pas (suite)

```
    def test_le_de_reste_entre_un_et_six_quelle_que_soit_la_graine(self):
        for graine in range(200):
            self.assertIn(tirage_corrige(random.Random(graine)), range(1, 7))
    def test_heures_ouvrables_avec_horloge_injectee(self):
        self.assertTrue(est_heure_ouvrable_corrigee(datetime(2025, 6, 2, 9, 0)))
        self.assertFalse(est_heure_ouvrable_corrigee(datetime(2025, 6, 2, 18, 0)))

assert detecter(TestStable) == {}
print("version corrigée : aucun échec sur 40 variations d'ordre et de graine")

# ================ 3. test de mutation : couverture ≠ qualité ================
SOURCE = '''
def remise(total, nb_cours):
    if nb_cours >= 3:
        return total * 0.8
    if nb_cours == 2:
        return total * 0.9
    return total
'''
```

Exemple pas à pas (suite)

```
def sites(arbre):
    return [n for n in ast.walk(arbre)
            if isinstance(n, (ast.Compare, ast.BinOp)) or (isinstance(n, ast.Const
   ant) and isinstance(n.value, (int, float)) and not isinstance(n.value, bool))]

PERMUTATION = {ast.GtE: ast.Gt, ast.Gt: ast.GtE, ast.Eq: ast.NotEq, ast.NotEq: ast
   .Eq, ast.Lt: ast.LtE, ast.LtE: ast.Lt}

def muter(source, k):
    arbre = ast.parse(source)
    n = sites(arbre)[k]
    if isinstance(n, ast.Compare):
        n.ops[0] = PERMUTATION[type(n.ops[0])]()
    elif isinstance(n, ast.BinOp):
        n.op = ast.Div() if isinstance(n.op, ast.Mult) else ast.Sub() if isinstanc
   e(n.op, ast.Add) else n.op
    else:
        n.value = n.value + 1
```

Exemple pas à pas (suite)

```
    ast.fix_missing_locations(arbre)
    return ast.unparse(arbre)

def suite_faible(remise):                                  # exécute TOUTES les l
   ignes… mais vérifie mal
    assert remise(100, 5) == 80
    assert remise(100, 2) > 0                              # assertion vague
    assert remise(100, 1) == 100

def suite_forte(remise):
    assert remise(100, 1) == 100
    assert remise(100, 2) == 90
    assert remise(100, 3) == 80
    assert remise(100, 4) == 80

def score(suite):
    total = len(sites(ast.parse(SOURCE)))
    survivants = []
    for k in range(total):
```

Exemple pas à pas (suite)

```
        espace = {}
        exec(muter(SOURCE, k), espace)
        try:
            suite(espace["remise"]); survivants.append(k)
        except AssertionError:
            pass
        except Exception:
            pass                                          # un plantage compte a
   ussi comme détecté
    return total, survivants

def couverture(suite):                                    # lignes de `remise` e
   xécutées
    espace = {}
    exec(compile(SOURCE, "<remise>", "exec"), espace)
    vues = set()
    def traceur(frame, ev, arg):
        if frame.f_code.co_filename == "<remise>":
            if ev == "line": vues.add(frame.f_lineno)
```

Exemple pas à pas

Exemple pas à pas (suite)

```
            return traceur
    sys.settrace(traceur); suite(espace["remise"]); sys.settrace(None)
    corps = {ligne for c in [espace["remise"].__code__] for _, _, ligne in c.co_li
   nes() if ligne and ligne != c.co_firstlineno}
    return len(vues & corps) / len(corps)

for nom, suite in (("faible", suite_faible), ("forte", suite_forte)):
    total, survivants = score(suite)
    print(f"suite {nom} : couverture {couverture(suite):.0%}, mutants tués {total
   - len(survivants)}/{total} (score {1 - len(survivants) / total:.0%})")

assert couverture(suite_faible) == 1.0
assert len(score(suite_faible)[1]) > 0 and score(suite_forte)[1] == []
```

Lecture :

Exemple pas à pas (suite)

1. `detecter()` rejoue la suite **40 fois** en variant la graine et l'ordre. Les tests qui échouent seulement *parfois* sont identifiés avec leur taux d'échec. Les causes sont visibles : état global (`COMPTEUR`) et hasard non maîtrisé.
2. La version corrigée applique les remèdes : état créé dans `setUp`, hasard injecté (on teste *toutes* les graines), horloge passée en paramètre.
3. Le test de mutation montre qu'une suite peut avoir **100 % de couverture** (toutes les lignes sont exécutées) et laisser **survivre** des mutants (`> 0` est une assertion trop vague). La suite forte tue tous les mutants.

À toi, avec aide

À toi, avec aide

Corrige ces trois tests instables en écrivant pour chacun la **cause** et la **version stable** :

```
# A
def test_expiration():
    token = creer_token()
    time.sleep(1)
    assert token.est_valide()           # valide pendant 1 s seulement

# B
def test_liste_utilisateurs():
    assert str(set(["ada", "bob", "cleo"])) == "{'ada', 'bob', 'cleo'}"

# C
def test_resultat_asynchrone():
    lancer_calcul()
    time.sleep(0.5)
    assert lire_resultat() == 42
```

À toi, avec aide · réponse

Indices progressifs

Indice 1

A : le temps réel. Injecte une horloge ou fixe une durée de validité large avec horloge simulée.

Indice 2

B : l'ordre des éléments d'un ensemble n'est pas garanti ; compare des ensembles.

Indice 3

C : remplace `sleep` par une attente explicite avec délai maximal.

À toi, avec aide · réponse

Réponse

Test	Cause	Correction
A	dépendance au **temps réel** (`sleep` près de l'expiration)	`creer_token(maintenant)` avec horloge injectée ; avancer l'horloge simulée de 0,9 s puis de 1,1 s
B	**ordre non garanti** d'un `set` converti en texte	`assert set(lister()) == {"ada", "bob", "cleo"}` ou `sorted(...)`
C	**attente fixe**	`attendre(lambda: lire_resultat() is not None, delai=5)` puis `assert lire_resultat() == 42`

```
from datetime import datetime, timedelta

class Token:
    def __init__(self, maintenant, duree=timedelta(seconds=1)):
        self.cree, self.duree = maintenant, duree
    def est_valide(self, maintenant):
        return maintenant - self.cree < self.duree
```

À toi, avec aide · réponse

Réponse (suite)

```
t0 = datetime(2025, 6, 1, 12, 0, 0)
token = Token(t0)
assert token.est_valide(t0 + timedelta(seconds=0.9))
assert not token.est_valide(t0 + timedelta(seconds=1.1))
assert {"ada", "bob", "cleo"} == set(["cleo", "ada", "bob"])
print("test A déterministe : l'horloge est simulée")
```

Le test A ne dépend plus de la vitesse de la machine, ne dure plus 1 seconde, et couvre les deux côtés de la frontière.

À toi, en solo

Mini outil de gestion d'instabilité. Écris `Suivi` qui :

1. enregistre, pour chaque exécution de suite, le résultat (réussi/échoué) de chaque test, avec le numéro de validation (*commit*) ;
2. calcule le **taux d'instabilité** d'un test : parmi les exécutions **sur le même commit**, la proportion de résultats qui diffèrent du résultat majoritaire ;
3. met en **quarantaine** (liste) les tests avec taux > 5 % sur au moins 5 exécutions ;
4. sort un rapport : tests en quarantaine, depuis quand, taux ;
5. teste avec 3 commits simulés : un test stable, un test cassé réellement (échoue toujours après le commit 2), un test instable (échoue 30 % du temps, graine fixée). Montre que le test réellement cassé n'est **pas** mis en quarantaine.

À toi, en solo · réponse

Pistes de correction

```
import random
from collections import defaultdict

class Suivi:
    def __init__(self):
        self.executions = defaultdict(lambda: defaultdict(list))        # test ->
   commit -> [bool, ...]
    def enregistrer(self, commit, resultats):
        for test, ok in resultats.items():
            self.executions[test][commit].append(ok)
    def taux_instabilite(self, test):
        diff = total = 0
        for resultats in self.executions[test].values():
            if len(resultats) < 2:
                continue
            majoritaire = max(set(resultats), key=resultats.count)
            diff += sum(1 for r in resultats if r != majoritaire)
            total += len(resultats)
        return diff / total if total else 0.0
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    def nombre_executions(self, test):
        return sum(len(r) for r in self.executions[test].values())
    def quarantaine(self, seuil=0.05, minimum=5):
        return sorted(t for t in self.executions if self.nombre_executions(t) >= m
   inimum and self.taux_instabilite(t) > seuil)

rng = random.Random(21)
suivi = Suivi()
for commit in (1, 2, 3):
    for _ in range(10):                                          # 10 ex
   écutions par commit
        suivi.enregistrer(commit, {
            "test_stable": True,
            "test_casse": commit < 2,                            # cassé
    à partir du commit 2, TOUJOURS
            "test_instable": rng.random() > 0.3,                  # écho
   ue ~30 % du temps, même code
        })
quarantaine = suivi.quarantaine()
```

À toi, en solo · réponse

Pistes de correction (suite)

```
for t in ("test_stable", "test_casse", "test_instable"):
    print(f"{t:14s} instabilité {suivi.taux_instabilite(t):.0%}")
assert quarantaine == ["test_instable"]
assert suivi.taux_instabilite("test_casse") == 0.0
```

La mesure se fait **à code identique** (même commit) : un test réellement cassé échoue de façon **constante** et ne doit pas être caché en quarantaine — il doit être corrigé ou le code réparé. Seule la **variabilité sans changement de code** désigne l'instabilité.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
« Relancer jusqu'au vert »	Cache les vrais bugs et l'instabilité	Tracer, mesurer le taux, corriger
`sleep` fixes pour attendre	Lents et instables à la fois	Attentes explicites sur une condition
Hasard, horloge, réseau non maîtrisés	Échecs non reproductibles	Injection, graines fixées, doubles
Se fier uniquement à la couverture	Une suite peut tout exécuter sans rien vérifier	Compléter par le test de mutation

Mini-boss

Mini-boss

Audit de santé d'une suite de tests. On te fournit (ou tu écris) un petit projet de 4 fonctions et une suite de 15 tests dont 4 sont instables (heure, hasard, ordre, état global) et 3 sont faibles (assertions vagues).

1. écris le détecteur d'instabilité (ordre mélangé, 50 graines, mesure du taux) et classe les tests ;
2. corrige les 4 tests instables ;
3. écris un test de mutation pour les 4 fonctions (au moins 25 mutants) et calcule le score avant/après renforcement des tests faibles ;
4. liste les mutants survivants **équivalents** que tu écartes, avec justification ;
5. produis un rapport de santé : couverture, taux d'instabilité, score de mutation, tests les plus lents.

Mini-boss · réponse

Pistes de correction

Un bon travail : `detecter()` qui renvoie des taux ; corrections par injection (horloge, rng), `setUp` pour l'état ; un mutateur AST paramétrable (opérateurs de comparaison, arithmétiques, constantes, `and`/`or`, `return` de constantes) ; score avant (par exemple 60 %) et après (par exemple 95 %) ; justification des équivalents (par exemple `x >= 0` → `x > 0` quand `x` est toujours un entier strictement positif par construction) ; rapport en tableau. Le point important : le mutant équivalent doit être démontré équivalent, pas simplement « gênant ».

Quiz éclair

1. Un test instable est un test qui :

A. est très lent
B. passe ou échoue sur le même code sans modification
C. n'a pas d'assertion

2. Pour détecter l'instabilité, on peut :

A. supprimer le test
B. lancer une seule fois
C. rejouer en variant l'ordre, la graine et l'environnement

3. Un mutant survivant indique :

A. que la suite n'a pas détecté une altération du code

Quiz éclair

Quiz éclair (suite)

B. que le code est parfait
C. que le mutant est mort

4. 🔁 *Révision mission 08* : Le cycle du TDD est :

A. vert, rouge, déployer
B. rouge, vert, refactor
C. refactor, rouge, vert

5. 🔁 *Révision mission 08* : Un test de caractérisation sert à :

A. spécifier un nouveau comportement
B. accélérer le code
C. figer le comportement actuel d'un code existant avant de le refactorer

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quel test « qui échoue parfois » existe dans ton équipe ? Que s'est-il passé la dernière fois qu'un vrai bug s'y est caché ?
2. Pourquoi mesurer l'instabilité sur le même code (même commit) est-il essentiel ?

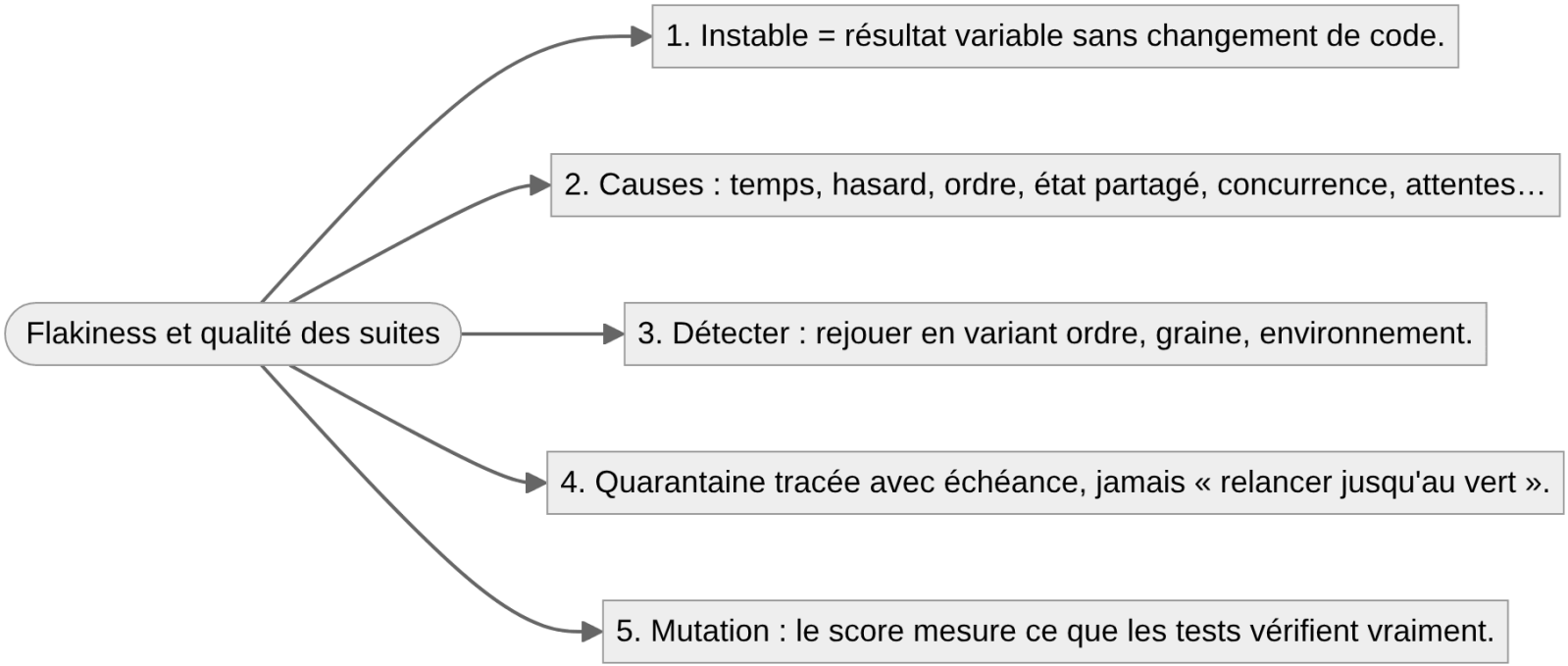
À retenir

À retenir

1. Instable = résultat variable sans changement de code.
2. Causes : temps, hasard, ordre, état partagé, concurrence, attentes fixes, réseau.
3. Détecter : rejouer en variant ordre, graine, environnement.
4. Quarantaine tracée avec échéance, jamais « relancer jusqu'au vert ».
5. Mutation : le score mesure ce que les tests vérifient vraiment.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Chasseur de tests instables

Prochaine mission : Tests dans CI

