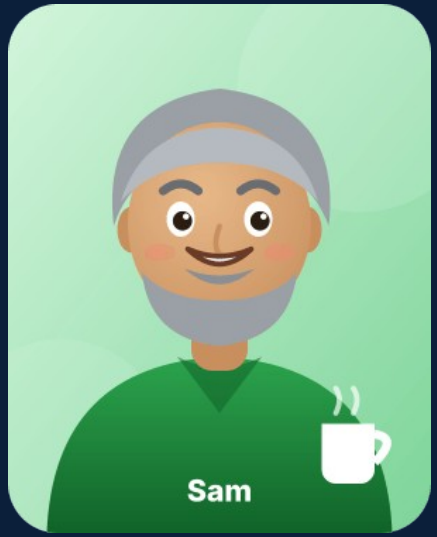
HEF-110 · MISSION 07 / 10

Doubles de test

Pour tester une pièce sans dépendre du monde entier, on la branche sur des doublures. Bien choisies, elles libèrent. Mal choisies, elles mentent.





Badge : Maître des doublures

100 min · intermédiaire

L'histoire

Kodi veut tester l'envoi d'un e-mail de rappel d'inscription. Mais ses tests envoient de vrais e-mails (aux formateurs !), dépendent du réseau, et échouent quand le serveur de messagerie est en panne. Il remplace donc l'envoi par une fausse fonction… et sa suite devient verte, y compris le jour où le vrai envoi est cassé depuis une semaine.



> *« Une doublure bien choisie rend ton test rapide et fiable, dit Sam. Une doublure mal choisie lui fait raconter n'importe quoi. Il faut connaître les cinq sortes, et savoir quand arrêter. »*

Tu apprends dummy, stub, spy, mock et fake, l'injection de dépendances, `unittest.mock`, et comment ne pas trop doubler.

Ta mission

À la fin, je serai capable de…

1. distinguer dummy, stub, spy, mock et fake
2. injecter des dépendances pour pouvoir les remplacer
3. utiliser `unittest.mock` (`Mock`, `patch`, `side_effect`, vérifications d'appels)
4. tester le temps et le hasard grâce à une horloge ou un générateur injecté
5. reconnaître le sur-usage de mocks et le couplage à l'implémentation

Défi d'échauffement

Défi d'échauffement

Pour s'entraîner à l'atterrissage, un pilote utilise un simulateur de vol plutôt qu'un vrai avion. Quelles choses le simulateur reproduit-il bien ? Lesquelles risque-t-il de ne pas montrer (turbulences réelles, panne mécanique inattendue) ? Fais le parallèle avec un faux serveur d'e-mails.

Le concept

Pourquoi des doublures

Un code dépend souvent de choses **lentes, imprévisibles ou dangereuses** : réseau, e-mail, paiement, horloge, hasard, base distante. Dans un test unitaire, on les remplace par des **doubles de test** que l'on contrôle.

Les cinq sortes (Meszaros)

Double	Rôle	Exemple
Dummy	remplit un paramètre obligatoire, jamais utilisé vraiment	un logger qui ne fait rien
Stub	renvoie des réponses **préparées**	`taux_de_change()` renvoie toujours 1,10
Spy	enregistre les appels reçus pour qu'on les inspecte ensuite	liste des e-mails « envoyés »

Le concept

Les cinq sortes (Meszaros) (suite)

Double	Rôle	Exemple
Mock	spy dont les **attentes** sont vérifiées (appelé 1 fois avec tel argument)	`envoyer.assert_called_once_with(...)`
Fake	implémentation **simplifiée mais réellement fonctionnelle**	dépôt en mémoire à la place de la base

Retiens : le **stub** fournit des **entrées**, le **spy/mock** observe les **sorties** (effets de bord), le **fake** remplace une dépendance entière.

L'injection de dépendances

Pour pouvoir remplacer une dépendance, le code la reçoit **de l'extérieur** plutôt que de la créer :

Le concept

L'injection de dépendances (suite)

```
# difficile à tester
class Rappel:
    def envoyer(self): smtp = SMTP("mail.exemple.org"); smtp.send(...)

# facile à tester
class Rappel:
    def __init__(self, expediteur): self.expediteur = expediteur
    def envoyer(self): self.expediteur.envoyer(...)
```

Même idée pour **l'horloge** (`maintenant` en paramètre) et le **hasard** (générateur passé en paramètre).

`unittest.mock`

Outil	Usage
`Mock()` / `MagicMock()`	objet complaisant qui accepte tout appel et les enregistre
`mock.return_value = x`	stub : valeur renvoyée

Le concept

`unittest.mock` (suite)

Outil	Usage
`mock.side_effect = Erreur ou liste`	lever une exception, ou renvoyer des valeurs successives
`mock.assert_called_once_with(...)`	vérification d'appel
`mock.call_args_list`	historique des appels
`patch("module.nom")`	remplace temporairement un objet **là où il est utilisé**
`Mock(spec=Classe)`	refuse les attributs qui n'existent pas dans la vraie classe

Piège de `patch` : on remplace le nom **là où il est cherché**, pas là où il est défini (`patch("mon_module.requests")`, pas `patch("requests")` si `mon_module` a fait `import requests` / `from x import y`).

Quand utiliser quoi

- **Entrée externe** (heure, hasard, réponse d'API) → **stub**.

Le concept

Quand utiliser quoi (suite)

- **Effet de bord à vérifier** (e-mail envoyé, paiement déclenché) → **spy/mock**.
- **Dépendance riche** (base, système de fichiers) → **fake**, si possible vérifié contre la vraie.
- **Paramètre inutile** → **dummy**.

Le danger des doublures

Risque	Conséquence	Remède
Sur-usage de mocks	tests couplés à l'implémentation : cassent à chaque refactoring	doubler aux **frontières** (réseau, horloge), pas entre classes internes
Double qui diverge du réel	tests verts, production cassée	spec=, tests de contrat sur le fake et le réel
Tout est doublé	le test ne teste plus rien	un test d'intégration sans double pour les coutures

Le concept

Le danger des doublures (suite)

Risque	Conséquence	Remède
Vérifier le « comment »	refactoring impossible	vérifier l'**effet observable** (e-mail envoyé), pas la séquence d'appels internes

Règle utile : **ne double pas ce que tu ne possèdes pas sans l'envelopper** — crée un petit adaptateur (`ExpediteurMail`) que tu possèdes, double *celui-là*, et teste l'adaptateur en intégration.

🧠 ***Astuce** : si un test contient plus de lignes de configuration de mocks que de vérification, c'est un signal : le code testé a probablement trop de dépendances.*

Exemple pas à pas

Problème : tester un service de rappel d'échéances avec les cinq sortes de doubles, plus `unittest.mock`, une horloge injectée, et un test de contrat qui garde le fake honnête.

```
import io, sqlite3, unittest
from datetime import date, timedelta
from unittest import mock

# ---------- code de production ----------
class ServiceRappel:
    def __init__(self, depot, expediteur, journal):
        self.depot, self.expediteur, self.journal = depot, expediteur, journal

    def rappeler(self, aujourdhui):
        envoyes = 0
        for inscription in self.depot.echeances_avant(aujourdhui + timedelta(days=
   3)):
            if inscription["rappele"]:
                continue
```

Exemple pas à pas (suite)

```
            try:
                self.expediteur.envoyer(inscription["email"], f"Rappel : {inscript
   ion['cours']} le {inscription['date']}")
            except ConnectionError:
                self.journal.avertir(f"échec d'envoi à {inscription['email']}")
                continue
            self.depot.marquer_rappele(inscription["id"])
            envoyes += 1
        return envoyes

# ---------- doubles écrits à la main ----------
class JournalDummy:                                  # DUMMY : on ne s'en sert p
   as dans ce test
    def avertir(self, message): pass

class JournalSpy:                                    # SPY : enregistre les mess
   ages
    def __init__(self): self.messages = []
    def avertir(self, message): self.messages.append(message)
```

Exemple pas à pas

Exemple pas à pas (suite)

```
class ExpediteurSpy:                                    # SPY : enregistre les e-ma
   ils « envoyés »
    def __init__(self): self.envoyes = []
    def envoyer(self, destinataire, texte): self.envoyes.append((destinataire, tex
   te))

class ExpediteurEnPanne:                                # STUB qui simule une panne
    def envoyer(self, destinataire, texte): raise ConnectionError("serveur de mess
   agerie injoignable")

class DepotFake:                                        # FAKE : vraie logique, sto
   ckage en mémoire
    def __init__(self, lignes): self.lignes = {l["id"]: dict(l) for l in lignes}
    def echeances_avant(self, limite): return [dict(l) for l in sorted(self.lignes
   .values(), key=lambda l: l["id"]) if l["date"] <= limite]
    def marquer_rappele(self, id_): self.lignes[id_]["rappele"] = True

AUJOURDHUI = date(2025, 6, 1)                           # l'horloge est un PARAMÈTR
```

Exemple pas à pas

Exemple pas à pas (suite)

```
   E : aucun test ne dépend de la date réelle
LIGNES = [
    {"id": 1, "email": "ada@exemple.fr", "cours": "HEF-010", "date": date(2025, 6,
    3), "rappele": False},
    {"id": 2, "email": "bob@exemple.fr", "cours": "HEF-020", "date": date(2025, 6,
    20), "rappele": False},
    {"id": 3, "email": "cleo@exemple.fr", "cours": "HEF-030", "date": date(2025, 6
   , 2), "rappele": True},
]

class TestServiceRappel(unittest.TestCase):
    def test_spy_seuls_les_cours_proches_non_rappeles_sont_envoyes(self):
        exp = ExpediteurSpy()
        n = ServiceRappel(DepotFake(LIGNES), exp, JournalDummy()).rappeler(AUJOURD
   HUI)
        self.assertEqual(n, 1)
        self.assertEqual(exp.envoyes, [("ada@exemple.fr", "Rappel : HEF-010 le 202
   5-06-03")])
```

Exemple pas à pas (suite)

```
    def test_un_rappel_n_est_envoye_qu_une_fois(self):
        depot, exp = DepotFake(LIGNES), ExpediteurSpy()
        service = ServiceRappel(depot, exp, JournalDummy())
        service.rappeler(AUJOURDHUI); service.rappeler(AUJOURDHUI)
        self.assertEqual(len(exp.envoyes), 1)

    def test_stub_en_panne_journalise_et_ne_marque_pas_comme_rappele(self):
        depot, journal = DepotFake(LIGNES), JournalSpy()
        n = ServiceRappel(depot, ExpediteurEnPanne(), journal).rappeler(AUJOURDHUI
    )
        self.assertEqual(n, 0)
        self.assertEqual(journal.messages, ["échec d'envoi à ada@exemple.fr"])
        self.assertFalse(depot.lignes[1]["rappele"])           # on pourra réessay
    er au prochain passage

    def test_mock_verifie_l_appel_exact(self):
        expediteur = mock.Mock(spec=["envoyer"])               # spec : refuse tou
    t autre attribut
        ServiceRappel(DepotFake(LIGNES), expediteur, JournalDummy()).rappeler(AUJO
```

Exemple pas à pas (suite)

```
    URDHUI)
         expediteur.envoyer.assert_called_once_with("ada@exemple.fr", "Rappel : HEF
    -010 le 2025-06-03")

     def test_mock_side_effect_premier_echec_puis_succes(self):
         lignes = [dict(LIGNES[0]), dict(LIGNES[0], id=4, email="dan@exemple.fr")]
         expediteur = mock.Mock(spec=["envoyer"])
         expediteur.envoyer.side_effect = [ConnectionError("panne"), None]
         n = ServiceRappel(DepotFake(lignes), expediteur, JournalSpy()).rappeler(AU
    JOURDHUI)
         self.assertEqual(n, 1)                                  # le premier a écho
    ué, le second est passé

     def test_patch_remplace_temporairement_une_fonction_utilisee(self):
         import types
         module = types.SimpleNamespace(hasard=lambda: 0.99)
         def tirage(): return "gagné" if module.hasard() > 0.9 else "perdu"
         with mock.patch.object(module, "hasard", return_value=0.1):
             self.assertEqual(tirage(), "perdu")
```

Exemple pas à pas (suite)

```
        self.assertEqual(tirage(), "gagné")                    # le patch est annu
   lé à la sortie du bloc

# ---------- le fake est-il honnête ? même contrat pour le fake et le vrai dépôt -
   ---------
class DepotSQL:
    def __init__(self, lignes):
        self.bd = sqlite3.connect(":memory:")
        self.bd.execute("CREATE TABLE r (id INTEGER PRIMARY KEY, email TEXT, cours
    TEXT, date TEXT, rappele INTEGER)")
        for l in lignes:
            self.bd.execute("INSERT INTO r VALUES (?,?,?,?,?)", (l["id"], l["email
   "], l["cours"], l["date"].isoformat(), int(l["rappele"])))
    def echeances_avant(self, limite):
        lignes = self.bd.execute("SELECT id, email, cours, date, rappele FROM r WH
   ERE date <= ? ORDER BY id", (limite.isoformat(),))
        return [{"id": i, "email": e, "cours": c, "date": date.fromisoformat(d), "
   rappele": bool(r)} for i, e, c, d, r in lignes]
    def marquer_rappele(self, id_):
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        self.bd.execute("UPDATE r SET rappele = 1 WHERE id = ?", (id_,))

class ContratDepot:                                    # les MÊMES tests pour
   le fake et le vrai
    def fabriquer(self, lignes): raise NotImplementedError
    def test_echeances_triees_et_filtrees(self):
        depot = self.fabriquer(LIGNES)
        self.assertEqual([l["id"] for l in depot.echeances_avant(date(2025, 6, 3))
   ], [1, 3])
    def test_marquer_rappele_est_visible(self):
        depot = self.fabriquer(LIGNES)
        depot.marquer_rappele(1)
        self.assertTrue(next(l for l in depot.echeances_avant(date(2025, 6, 3)) if
    l["id"] == 1)["rappele"])

class TestContratFake(ContratDepot, unittest.TestCase):
    def fabriquer(self, lignes): return DepotFake(lignes)
class TestContratSQL(ContratDepot, unittest.TestCase):
    def fabriquer(self, lignes): return DepotSQL(lignes)
```

Exemple pas à pas

Exemple pas à pas (suite)

```
classes = [TestServiceRappel, TestContratFake, TestContratSQL]
suite = unittest.TestSuite(unittest.defaultTestLoader.loadTestsFromTestCase(c) for
    c in classes)
resultat = unittest.TextTestRunner(stream=io.StringIO()).run(suite)
print("tests :", resultat.testsRun, "| réussite :", resultat.wasSuccessful())
assert resultat.wasSuccessful(), resultat.failures + resultat.errors
```

Lecture :

Double	Où
Dummy	`JournalDummy` (obligatoire mais inutile dans ce test)
Stub	`ExpediteurEnPanne` (prépare une réponse : une panne) ; `side_effect`
Spy	`ExpediteurSpy`, `JournalSpy` : on inspecte après coup
Mock	`Mock(spec=["envoyer"])` + `assert_called_once_with`

Exemple pas à pas (suite)

Double	Où
Fake	`DepotFake` (vraie logique, mémoire)

L'**horloge** est un paramètre (`AUJOURDHUI`) : aucun test ne dépend de la date réelle. Le **contrat partagé** exécute les mêmes tests contre le fake et le vrai dépôt SQL : si le fake divergeait, un test de contrat échouerait. C'est ce qui rend le fake digne de confiance.

À toi, avec aide

Voici une fonction qui dépend de l'heure et du hasard :

```
def message_du_jour():
    h = datetime.now().hour
    salut = "Bonjour" if h < 18 else "Bonsoir"
    astuce = random.choice(["Teste d'abord.", "Relis ton code.", "Fais une pause."
   ])
    return f"{salut} ! {astuce}"
```

Refactore-la pour qu'elle soit **testable** (injecte l'horloge et le générateur) et écris 3 tests déterministes : matin, soir, et le choix d'une astuce donnée par un générateur à graine.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Passe `maintenant` et `rng` en paramètres, avec des valeurs par défaut.

Indice 2

`random.Random(0)` est reproductible ; ou fournis un stub dont `choice` renvoie toujours le premier élément.

Indice 3

`datetime(2025, 6, 1, 9)` pour le matin, `datetime(2025, 6, 1, 20)` pour le soir.

À toi, avec aide · réponse

Réponse

```
import random
from datetime import datetime

def message_du_jour(maintenant=None, rng=None):
    maintenant = maintenant or datetime.now()
    rng = rng or random
    salut = "Bonjour" if maintenant.hour < 18 else "Bonsoir"
    astuce = rng.choice(["Teste d'abord.", "Relis ton code.", "Fais une pause."])
    return f"{salut} ! {astuce}"

class RngPremier:                                    # stub : choisit toujours
   le premier élément
    def choice(self, seq): return seq[0]

assert message_du_jour(datetime(2025, 6, 1, 9), RngPremier()) == "Bonjour ! Teste
   d'abord."
assert message_du_jour(datetime(2025, 6, 1, 20), RngPremier()) == "Bonsoir ! Teste
    d'abord."
a = message_du_jour(datetime(2025, 6, 1, 9), random.Random(0))
```

À toi, avec aide · réponse

Réponse (suite)

```
b = message_du_jour(datetime(2025, 6, 1, 9), random.Random(0))
assert a == b and a.startswith("Bonjour ! ")
print("déterministe :", a)
```

Les valeurs par défaut (`datetime.now()`, `random`) gardent l'usage normal intact ; les tests passent leurs propres doublures.

À toi, en solo

Écris ton propre mini-mock. Implémente une classe `EspionAppels` qui :

1. enregistre chaque appel (arguments positionnels et nommés) ;
2. permet de programmer des retours (`retourne(valeur)`) ou une suite de retours, ou une exception ;
3. fournit `assert_appele_une_fois_avec(...)`, `nombre_appels` et `assert_jamais_appele()` ;
4. teste-le sur un service de paiement : le service doit appeler `passerelle.debiter(montant, devise)` **une fois**, ne **pas** débiter un montant nul, et gérer une exception `PaiementRefuse` en renvoyant `False`.

À toi, en solo · réponse

Pistes de correction

```
class PaiementRefuse(Exception):
    pass

class EspionAppels:
    def __init__(self):
        self.appels, self._sorties = [], []
    def retourne(self, *valeurs):
        self._sorties = list(valeurs); return self
    def leve(self, exception):
        self._sorties = [exception]; return self
    def __call__(self, *args, **kwargs):
        self.appels.append((args, kwargs))
        if self._sorties:
            s = self._sorties.pop(0) if len(self._sorties) > 1 else self._sorties[
   0]
            if isinstance(s, Exception):
                raise s
            return s
    @property
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    def nombre_appels(self): return len(self.appels)
    def assert_appele_une_fois_avec(self, *args, **kwargs):
        assert self.nombre_appels == 1, f"{self.nombre_appels} appel(s)"
        assert self.appels[0] == (args, kwargs), f"appelé avec {self.appels[0]}"
    def assert_jamais_appele(self):
        assert self.nombre_appels == 0, f"{self.nombre_appels} appel(s)"

class Passerelle:
    def __init__(self, debiter): self.debiter = debiter

def payer(passerelle, montant, devise="EUR"):
    if montant <= 0:
        return False
    try:
        passerelle.debiter(montant, devise)
        return True
    except PaiementRefuse:
        return False
```

À toi, en solo · réponse

Pistes de correction (suite)

```
# 1 appel exact
esp = EspionAppels().retourne("ok")
assert payer(Passerelle(esp), 50) is True
esp.assert_appele_une_fois_avec(50, "EUR")

# montant nul : aucun débit
esp = EspionAppels()
assert payer(Passerelle(esp), 0) is False
esp.assert_jamais_appele()

# paiement refusé
esp = EspionAppels().leve(PaiementRefuse("carte refusée"))
assert payer(Passerelle(esp), 30, "USD") is False
esp.assert_appele_une_fois_avec(30, "USD")
print("mini-mock validé")
```

En construisant l'outil, tu vois qu'un « mock » n'est qu'une fonction qui **se souvient** de ses appels et sait répondre selon un script.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Tout doubler	Le test ne vérifie plus que les doubles	Doubler aux frontières ; un test d'intégration sans double
Mocker ce qu'on ne possède pas	Couplage à une bibliothèque externe	Envelopper dans un adaptateur à soi, doubler l'adaptateur
Vérifier la séquence d'appels internes	Le moindre refactoring casse les tests	Vérifier l'effet observable
Fake qui diverge du réel	Tests verts, production cassée	Tests de contrat partagés fake/réel

Mini-boss

Mini-boss

Service de notification robuste. Écris `ServiceNotification(depot_utilisateurs, canaux, horloge, journal)` : pour chaque utilisateur, envoie un message par son canal préféré (e-mail, SMS) ; si le canal échoue, bascule sur le suivant ; ne dérange pas entre 22 h et 7 h (reporte) ; limite à 3 messages par jour.

1. écris le service avec injection de toutes les dépendances ;
2. écris 12+ tests en utilisant les 5 sortes de doubles (explique pourquoi chacun) ;
3. écris un `FauxCanal` (fake) avec pannes programmables, et un test de contrat partagé avec un canal « réel » simulé ;
4. écris une version « mal testée » qui mocke tout (séquence d'appels internes), montre qu'elle casse au refactoring alors que ta bonne suite survit.

Mini-boss · réponse

Pistes de correction

Un bon travail : horloge injectée (`horloge()` renvoie un `datetime`) ; canaux avec `envoyer(utilisateur, texte)` ; tests : bascule sur le canal suivant (stub qui lève), report la nuit (horloge stub), plafond de 3 messages (spy compteur), journalisation des échecs (spy), dummy pour le journal quand inutile ; fake `FauxCanal` avec `programmer_pannes([True, False])`. La démonstration finale : renommer une méthode privée interne casse la suite « mock-tout » (vérifie `_choisir_canal` appelé) mais ne change rien à la bonne suite (qui vérifie les messages effectivement reçus).

Quiz éclair

Quiz éclair

1. Un stub sert à :

A. fournir des réponses préparées au code testé
B. enregistrer des appels pour les vérifier
C. stocker des données en base

2. Un fake est :

A. un test qui échoue toujours
B. une implémentation simplifiée mais réellement fonctionnelle
C. un paramètre inutilisé

3. Pour rendre testable un code dépendant de l'heure, on :

A. attend minuit

Quiz éclair

Quiz éclair (suite)

B. lit l'horloge dans chaque fonction
C. injecte l'horloge en paramètre

4. 🔁 *Révision mission 06* : Un test E2E est approprié pour :

A. un parcours utilisateur critique complet
B. chaque règle de calcul isolée
C. la syntaxe d'une fonction

5. 🔁 *Révision mission 06* : Pour attendre qu'une page soit prête, on préfère :

A. un `sleep` fixe de 10 secondes
B. une attente explicite d'une condition avec délai maximal
C. relancer jusqu'à ce que ça passe

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Dans ton code, quelles dépendances (heure, hasard, réseau) rendent aujourd'hui les tests imprévisibles ?
2. Quel test pourrait survivre à un refactoring complet parce qu'il ne vérifie que l'effet observable ?

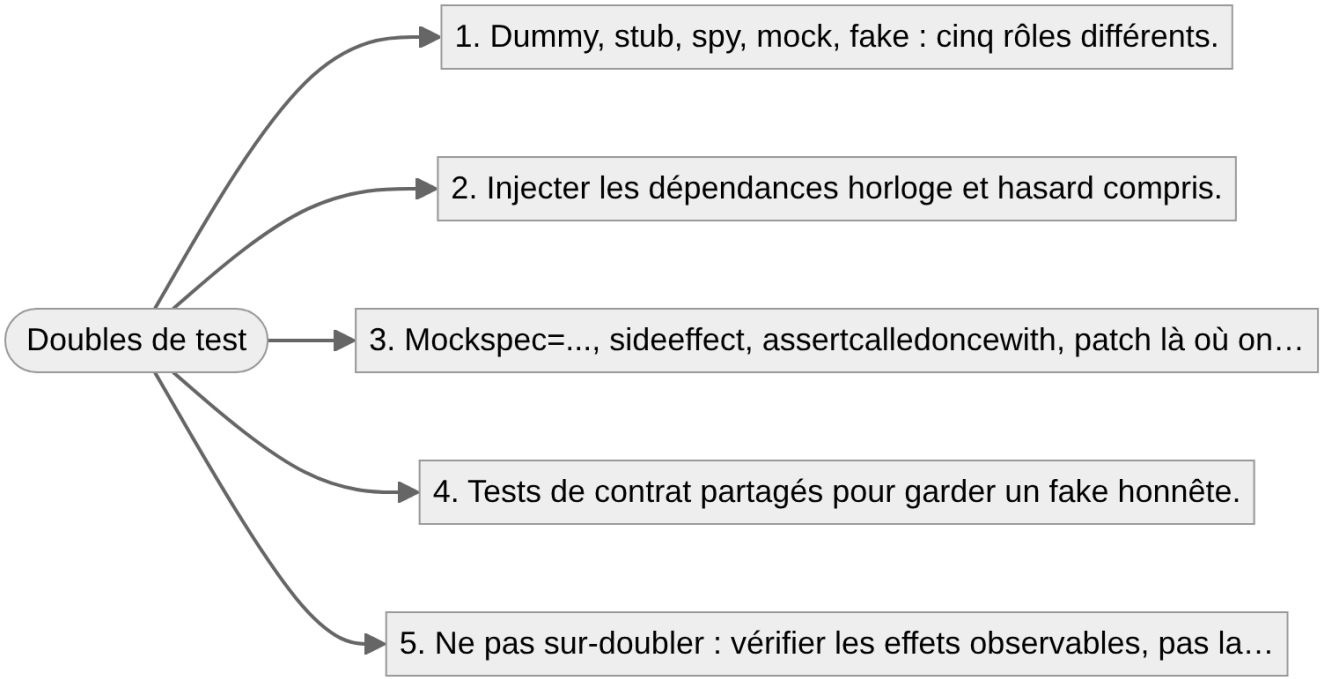
À retenir

À retenir

1. Dummy, stub, spy, mock, fake : cinq rôles différents.
2. Injecter les dépendances (horloge et hasard compris).
3. `Mock(spec=...)`, `side_effect`, `assert_called_once_with`, `patch` là où on cherche le nom.
4. Tests de contrat partagés pour garder un fake honnête.
5. Ne pas sur-doubler : vérifier les effets observables, pas la mécanique interne.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Maître des doublures

Prochaine mission : TDD et refactoring

