TUTORIEL COMPTE BANCAIRE

Prologue – L'histoire du projet

Notre conte commence simplement : un Client vit sans compte, puis décide d'en ouvrir un.

Le Compte Bancaire devient alors notre "héros" (classe fétiche) : il gagne de l'argent (dépôt), en dépense (retrait), et doit respecter des règles strictes (montants valides, solde suffisant).

À chaque étape, nous observons l'état de nos objets, puis nous sécurisons l'histoire avec des tests unitaires jusqu'à obtenir la fameuse barre verte.

Scénario du TP (captures à chaque étape)

1. Créer le projet

L'objectif ici est simplement de créer "la maison" du projet : un dossier qui contiendra tous les fichiers.

1. Créer un nouveau dossier sur ton ordinateur, par exemple :

   projet-agile/

2. Ouvre VS Code, puis :

   - File > Open Folder…
   - Sélectionne tp-agilite-banque/

3. Tu dois maintenant voir le dossier dans l'explorateur de fichiers de VS Code (à gauche).

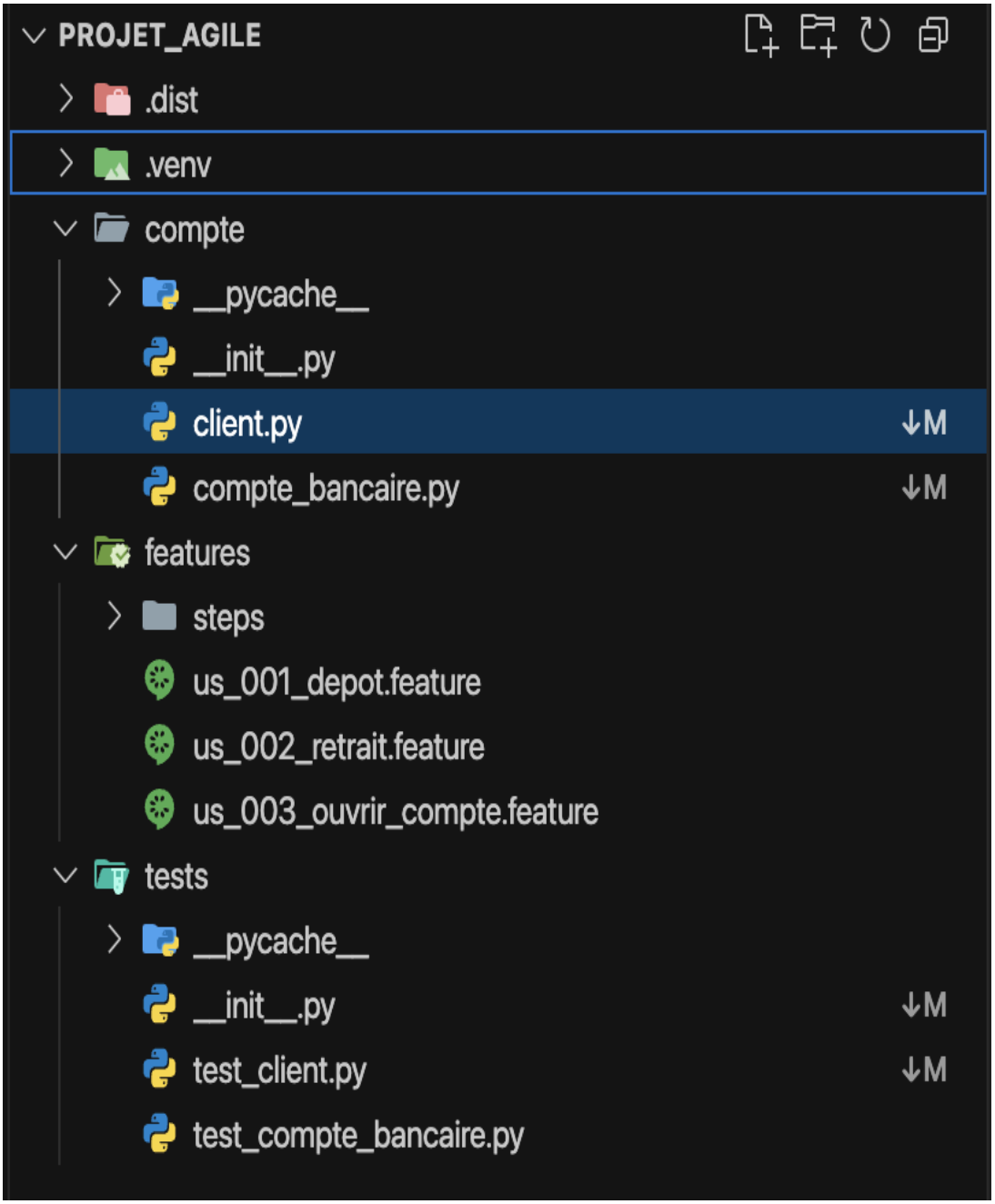
Ce que ça veut dire en clair : VS Code "travaille" dans ce dossier, et tous les fichiers du projet seront dedans.

2. Créer une structure minimale

Pour que le projet soit facile à comprendre, on sépare :

- le code principal (les classes)
- les tests (les vérifications automatiques)
- les features

on obtient la structure suivante:



À quoi servent ces dossiers ?

- compte/ : contient le code de notre “mini-banque”
- tests/ : contiendra les tests (on s’en servira plus tard)
- Les **features** (fonctionnalités), c’est tout ce que l’utilisateur peut faire, et tout ce que le système garantit grâce aux règles métier.
- __init__.py : petit fichier “technique” qui indique à Python que compte/ (et tests/) sont des dossiers de code (des “packages”). C’est comme un **marqueur** qui aide Python à s’y retrouver.

3. Créer la classe fétiche : CompteBancaire

Pourquoi “classe fétiche” ? C’est la classe principale du projet : celle autour de laquelle tout tourne.

4. “Compiler” / vérifier que le code s’exécute

En Python, on ne “compile” pas comme en Java : on lance simplement le programme.

Si Python arrive à lire le fichier sans erreur, on considère que tout est OK.

5. Instancier un objet CompteBancaire

“Instancier” veut dire : **créer un exemplaire concret** à partir de la classe.

La classe est un “moule”, l’objet est “le gâteau”.

6. Ajouter 2 attributs, des accesseurs, et une méthode qui les manipule

Objectif : enrichir le compte avec deux “réglages” simples, et permettre de les lire/modifier proprement.

compte > compte_bancaire.py > ...

```python
from dataclasses import dataclass

@dataclass
class CompteBancaire:
    titulaire: str
    solde: float = 0.0

    def deposer(self, montant: float) -> None:
        if montant <= 0:
            raise ValueError("montant invalide")
        self.solde += montant

    def retirer(self, montant: float) -> None:
        if montant <= 0:
            raise ValueError("montant invalide")
        if montant > self.solde:
            raise ValueError("solde insuffisant")
        self.solde -= montant
```

Choix rationnel des 2 attributs

Ici : un compte bancaire avec :

- un titulaire (nom)
- un solde (argent disponible)

Et deux actions simples :

- déposer de l’argent (le solde augmente)
- retirer de l’argent (le solde diminue, si possible)

C’est quoi un “accesseur” ?

C’est une façon “propre” d’accéder à un attribut :

- *lire* sa valeur (getter)
- *modifier* sa valeur (setter) en vérifiant que c’est cohérent

Comment expliquer ce code simplement ?

- titulaire : le nom du propriétaire du compte
- solde : l’argent sur le compte
- deposer(50) : ajoute 50 au solde
- retirer(30) : enlève 30 du solde, seulement si le compte a au moins 30

Quand ce n’est pas possible (ex : retirer plus que ce qu’on a), on “stoppe” l’action avec un message d’erreur.

7. Ré-instancier et observer l’effet sur l’état

Ici, on veut une preuve visuelle : **l’objet change** après une méthode.

8. Tester unitairement la classe et obtenir la “barre verte”

Nous utilisons **pytest** (équivalent jUnit pour l’objectif “barre verte”).

9. Ajouter une seconde classe et l’associer (unidirectionnelle 0..1 ↔ 0..1)

Maintenant, on ajoute un “personnage” : **Client**.

compte > client.py > ...

```python
from dataclasses import dataclass
from typing import Optional
from compte.compte_bancaire import CompteBancaire

@dataclass
class Client:
    nom: str
    compte: Optional[CompteBancaire] = None #Optional[CompteBancaire] soit il a un compte soit non 0..1
```

Explication (très simple)

- Un client peut **ne pas avoir de compte** (0)
- ou **en avoir un** (1)

Donc : **0..1 compte**.

“Unidirectionnelle” = on met le lien **dans un seul sens** :

- le Client connaît son Compte
- mais le Compte n’a pas besoin de connaître son Client

10. Ajouter une méthode collaborative (résultat basé sur les 2 objets)

Une “méthode collaborative”, c’est une action qui **utilise deux objets à la fois**.

```
10      def ouvrir_compte(self, solde_initial: float = 0.0):
11          self.compte = CompteBancaire(self.nom, solde_initial)
```

Ici, notre histoire : le **Client** ne manipule pas directement l’argent, il demande à **son CompteBancaire** d’exécuter l’action.

Exemple simple : le Client retire de l’argent via son compte

Pourquoi c’est collaboratif ?

- Client décide de l’action (le “qui”)
- CompteBancaire applique la règle et modifie le solde (le “comment”)

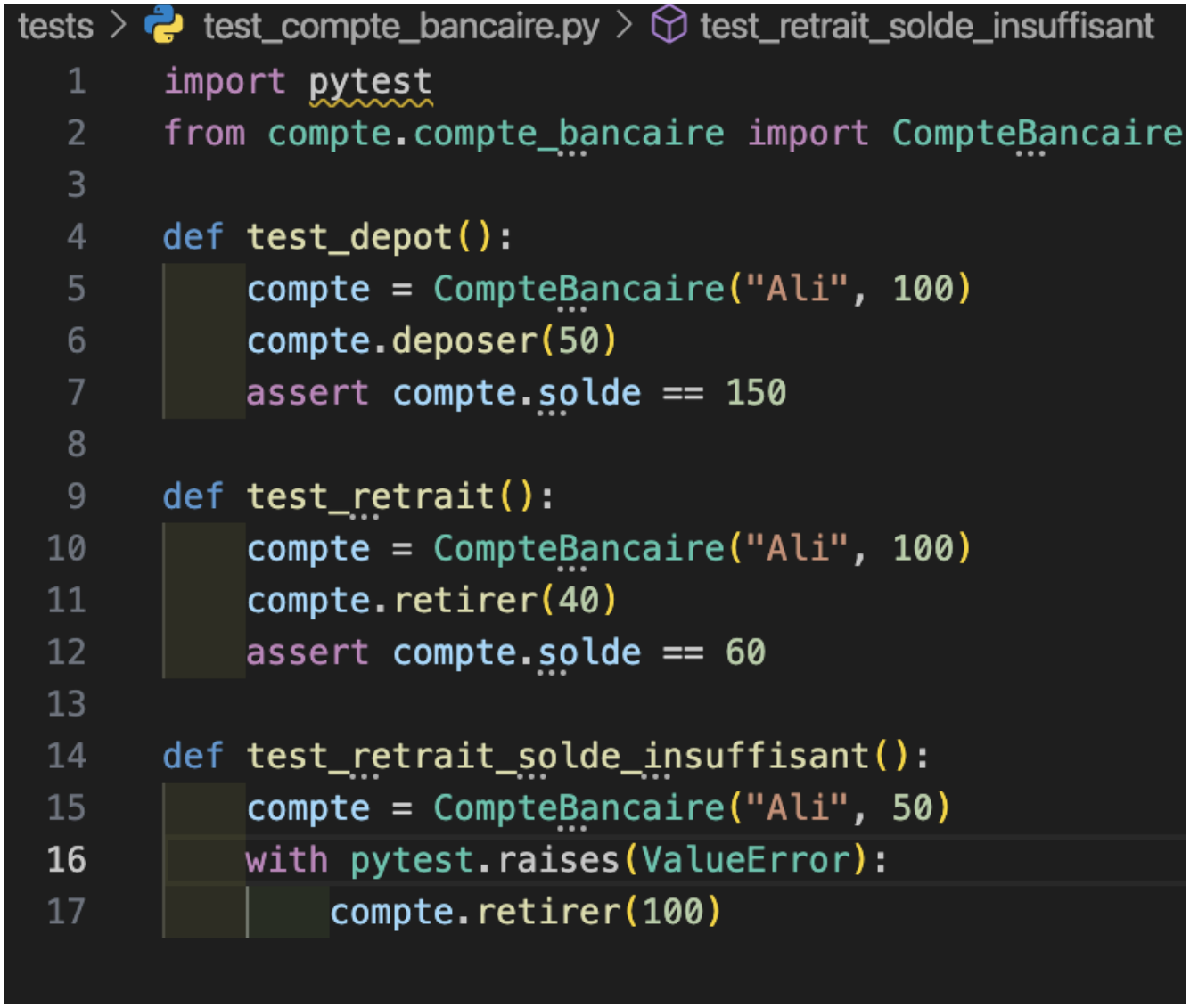
11. Instancier, relier les objets, et sauvegarder dans une fixture (setup)

Une **fixture**, c’est comme “préparer le décor” avant un test.

Au lieu de recréer à la main Client + Compte dans chaque test, on le fait une seule fois, puis on réutilise.

12. Créer un test qui utilise la fixture + montrer le résultat et la “barre”

Un test dit : “Quand je fais X, je m’attends à Y”.



13. Association bidirectionnelle 0..1 ↔ * + robustesse

Jusqu'ici : un Client a **0..1 compte**.

Maintenant, on fait évoluer l'histoire : un Client peut avoir **plusieurs comptes** (ex. compte courant + épargne).

C'est le * : "plusieurs".

Bidirectionnelle veut dire :

- le Client connaît ses comptes
- et chaque compte sait à quel Client il appartient

Robustesse = on évite les incohérences :

- un compte ne doit pas être listé deux fois
- un compte ne doit pas "appartenir" à deux clients en même temps

14. Deux refactorings (rename + extract method)

Refactoring = améliorer le code sans changer son comportement.

La preuve que le comportement n'a pas changé : **les tests restent verts**.

Refactoring 1 : Rename (renommage)

compte > compte_bancaire.py > ...

```python
from dataclasses import dataclass

@dataclass
class CompteBancaire:
    titulaire: str
    solde: float = 0.0

    def crediter(self, montant: float) -> None:
        if montant <= 0:
            raise ValueError("montant invalide")
        self.solde += montant

    def debiter(self, montant: float) -> None:
        if montant <= 0:
            raise ValueError("montant invalide")
        if montant > self.solde:
            raise ValueError("solde insuffisant")
        self.solde -= montant
```

Dans cet exemple on a renommé **retirer** en **débiter** et **déposer** en **créditer**.

- Ancien nom : retirer(montant)
- Nouveau nom : debiter(montant)

15. jUnit + "Test infected" + amélioration équivalente dans notre exemple

Même si notre projet est en Python, le TP demande de lire "Test infected" (jUnit) et d'en tirer une leçon.

Traduction simple de l'idée "Test infected"

Quand on commence à tester sérieusement :

- on découvre des défauts (c'est normal)
- on est tenté de "tricher" ou d'abandonner les tests (mauvaise idée)
- la bonne approche est d'en faire une routine : **Red** → **Green** → **Refactor**

Amélioration équivalente dans notre projet

Nous adoptons une règle de travail simple :

- **Red** : on écrit d'abord un test qui échoue (ex. "retirer un montant négatif doit échouer")
- **Green** : on écrit le minimum pour que le test passe
- **Refactor** : on nettoie le code (rename/extract) sans casser les tests

16. Loi de Murphy + situation rencontrée

Loi de Murphy : "Si quelque chose peut mal tourner, alors ça finira par arriver."

Exemple typique dans ce projet (à raconter)

- Les tests passent dans VS Code, mais échouent en terminal.
- Cause fréquente : on n'a pas activé le bon environnement Python (venv), ou les imports ne pointent pas pareil.