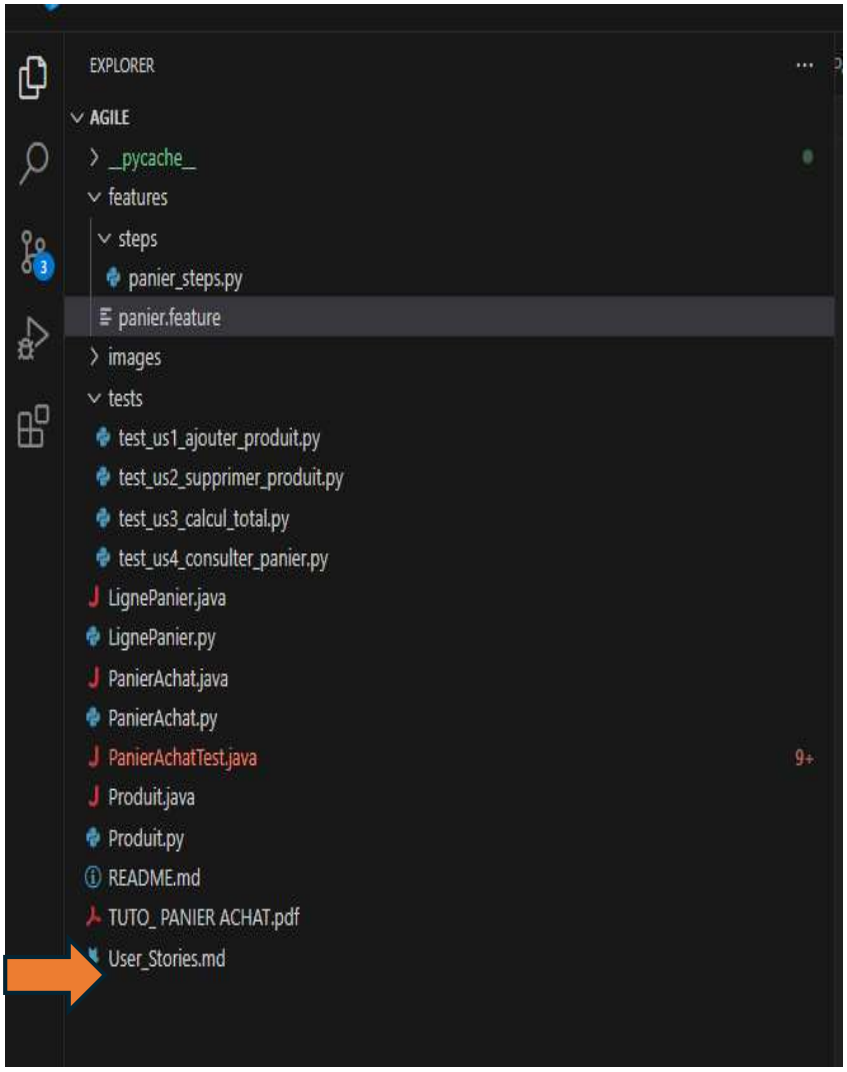
Tutoriel Python : Panier d'achat pas à pas

I. Découvrir Nos User Stories

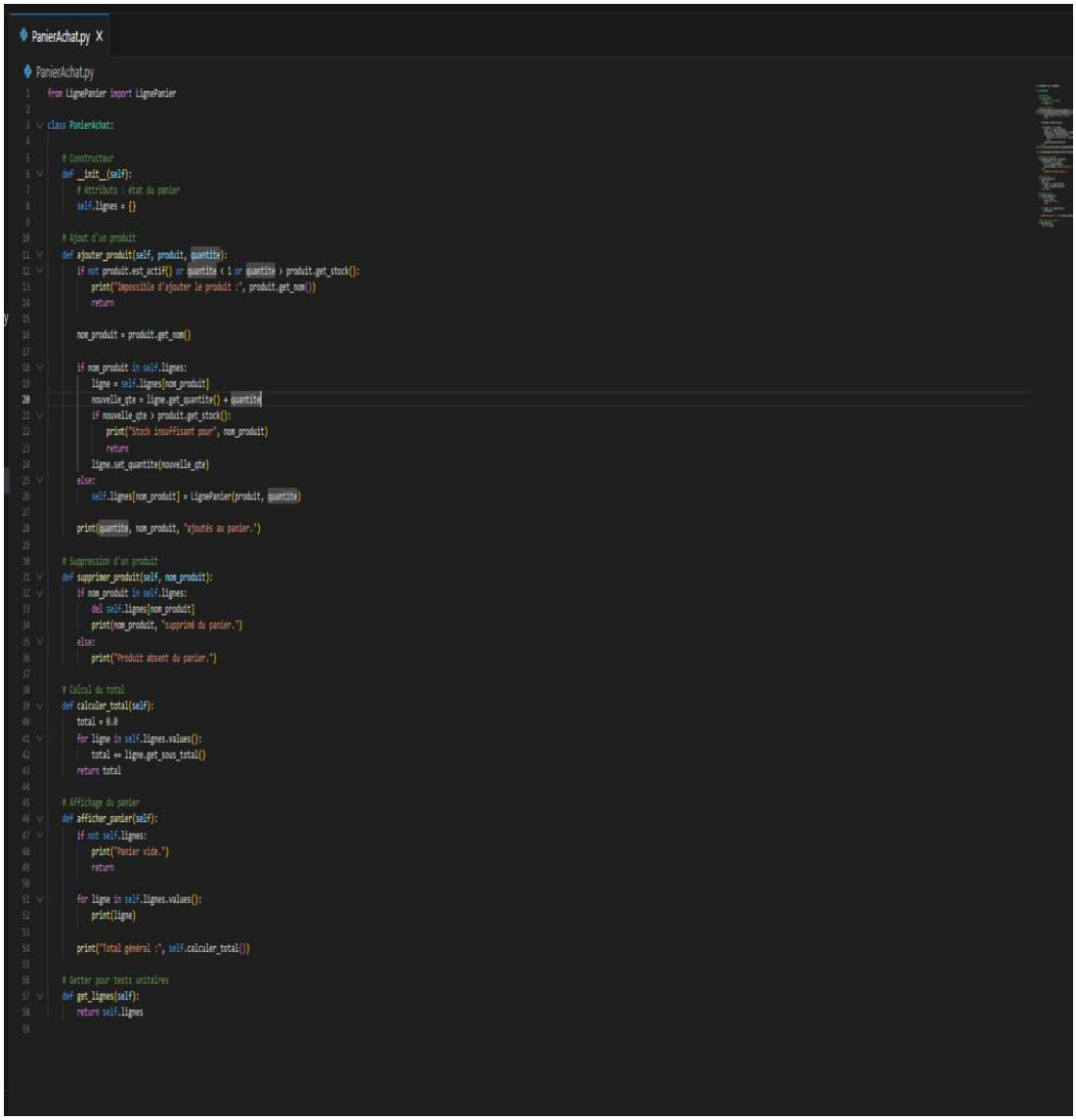
- Ouvrer le navigateur et aller sur le lien du projet : https://github.com/adrienfomat/AGILE
- Une fois sur la page du dépôt, on verra la liste des fichiers du projet. Cherche **User_Stories.md** dans la liste.



II. Programmation et codage

NB : Les codes initialement écrits en Java ont été convertis en Python.

1. Création de la classe Fétiche (PanierAchat)



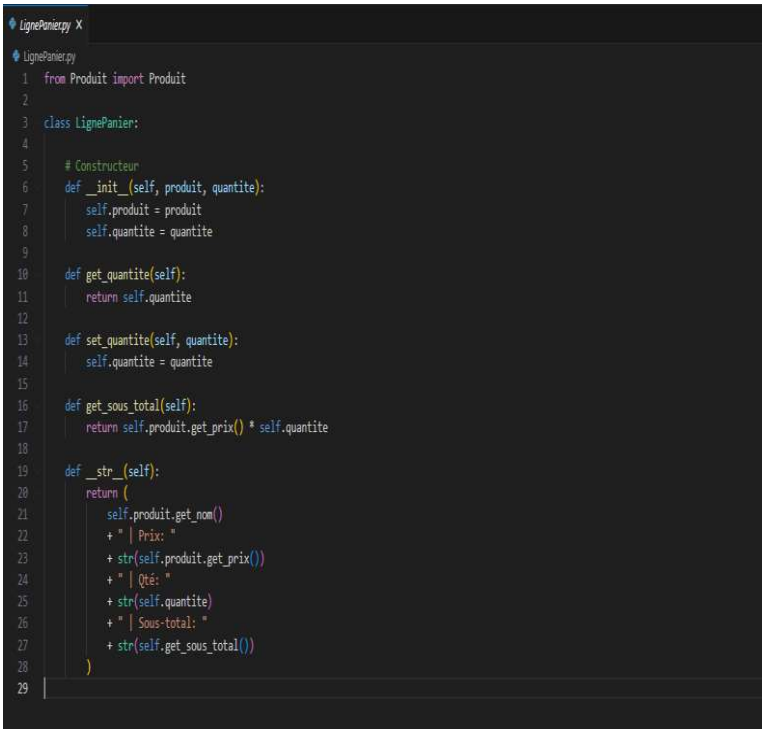
```python
from LignePanier import LignePanier

class PanierAchat:

    # Constructeur
    def __init__(self):
        # Attributs : état du panier
        self.lignes = {}

    # Ajout d'un produit
    def ajouter_produit(self, produit, quantite):
        if not produit.est_actif() or quantite < 1 or quantite > produit.get_stock():
            print("Impossible d'ajouter le produit :", produit.get_nom())
            return

        nom_produit = produit.get_nom()

        if nom_produit in self.lignes:
            ligne = self.lignes[nom_produit]
            nouvelle_qte = ligne.get_quantite() + quantite
            if nouvelle_qte > produit.get_stock():
                print("Stock insuffisant pour", nom_produit)
                return
            ligne.set_quantite(nouvelle_qte)
        else:
            self.lignes[nom_produit] = LignePanier(produit, quantite)

        print(quantite, nom_produit, "ajoutés au panier.")

    # Suppression d'un produit
    def supprimer_produit(self, nom_produit):
        if nom_produit in self.lignes:
            del self.lignes[nom_produit]
            print(nom_produit, "supprimé du panier.")
        else:
            print("Produit absent du panier.")

    # Calcul du total
    def calculer_total(self):
        total = 0.0
        for ligne in self.lignes.values():
            total += ligne.get_sous_total()
        return total

    # Affichage du panier
    def afficher_panier(self):
        if not self.lignes:
            print("Panier vide.")
            return

        for ligne in self.lignes.values():
            print(ligne)

        print("Total général :", self.calculer_total())

    # Getter pour tests unitaires
    def get_lignes(self):
        return self.lignes
```

Le PanierAchat utilise une **HashMap** pour stocker ses articles, où la clé est le nom du produit et la valeur est un objet **LignePanier** contenant le produit et sa quantité.

Les principales méthodes sont :

- **ajouterProduit** : ajoute ou met à jour un produit si le stock le permet.
- **supprimerProduit** : retire un produit du panier.
- **calculerTotal** : calcule le montant total du panier.
- **afficherPanier** : affiche le détail des articles et le total.
- **getLignes** : permet d'accéder aux articles, utile pour les tests.

2. Création de la classe LignePanier



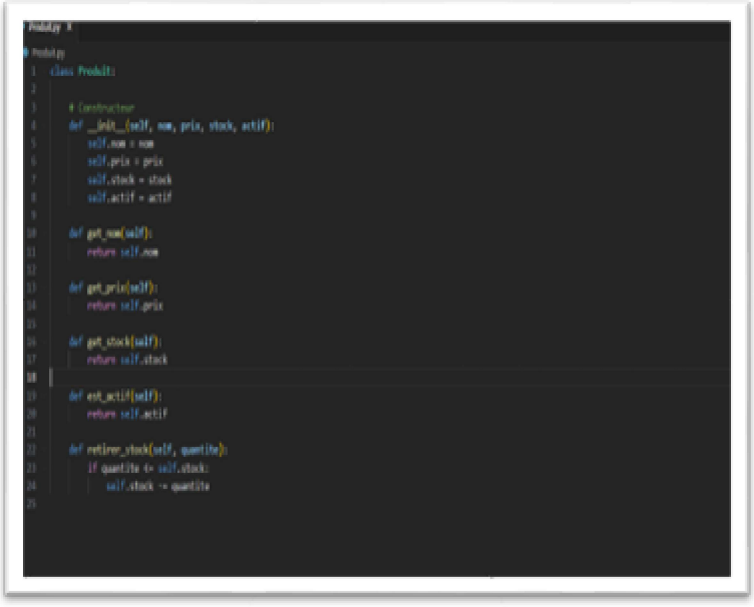
```python
from Produit import Produit

class LignePanier:

    # Constructeur
    def __init__(self, produit, quantite):
        self.produit = produit
        self.quantite = quantite

    def get_quantite(self):
        return self.quantite

    def set_quantite(self, quantite):
        self.quantite = quantite

    def get_sous_total(self):
        return self.produit.get_prix() * self.quantite

    def __str__(self):
        return (
            self.produit.get_nom()
            + " | Prix: "
            + str(self.produit.get_prix())
            + " | Qté: "
            + str(self.quantite)
            + " | Sous-total: "
            + str(self.get_sous_total())
        )
```

La **classe LignePanier** représente une ligne du panier, associant un **produit** à une **quantité**.

Principales fonctionnalités :

- **Constructeur** : crée une ligne avec un produit et sa quantité.
- **getSousTotal** : calcule le sous-total en multipliant le prix du produit par la quantité.
- **toString** : affiche la ligne de manière lisible (Nom | Prix | Qté | Sous-total).

3. Création de la classe Produit



La **classe Produit** représente un article à vendre avec ses caractéristiques : **nom**, **prix**, **stock** et **statut actif**.

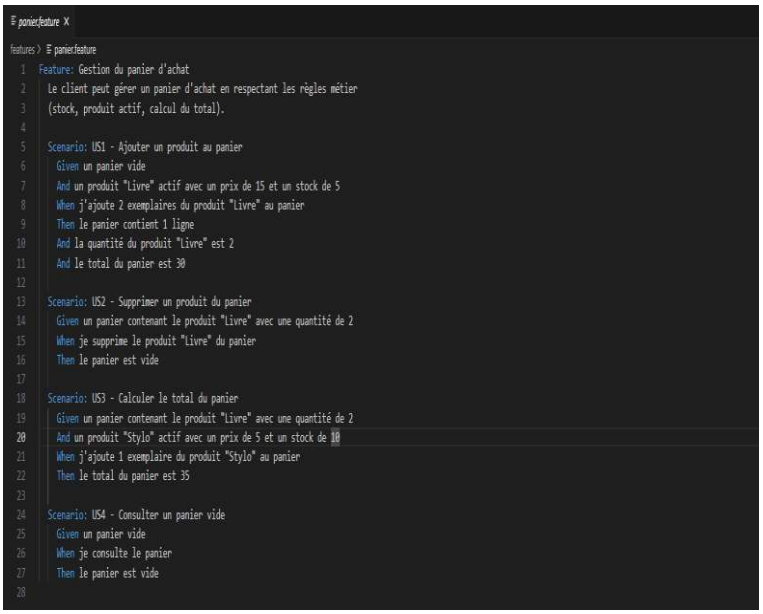
Principales fonctionnalités :

- **Constructeur** : initialise un produit avec toutes ses informations.
- **Getters :** permettent d'accéder au nom, prix et stock.
- **retirerStock** : réduit le stock lors d'un achat si la quantité est disponible.

4. Features

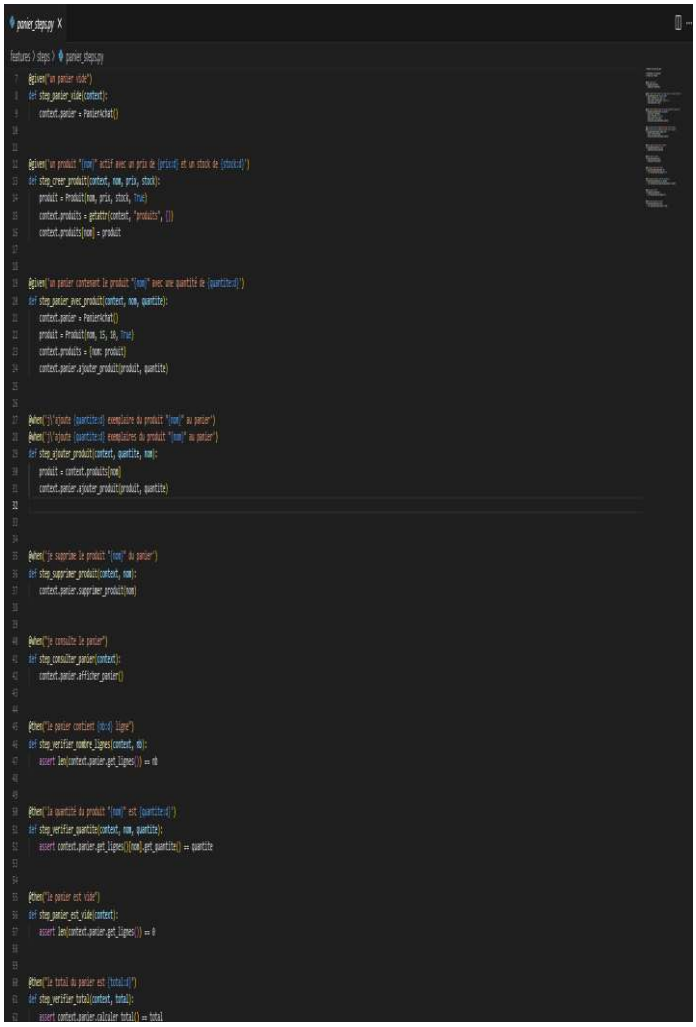
Les fonctionnalités du panier ont été décrites sous forme de scénarios BDD dans un fichier **.feature.**

Ces scénarios traduisent les User Stories en langage naturel structuré **(Given / When / Then)** et servent de référence fonctionnelle.



❖ Steps Behave

Les étapes définies dans les fichiers .feature ont été reliées au code Python via des **steps Behave**.
Chaque étape appelle directement les méthodes du code métier afin de valider le comportement du système.



```
PS C:\Users\adrien\Documents\GitHub\AGILE> behave
USING RUNNER: behave.runner:Runner
Feature: Gestion du panier d'achat # features/panier.feature:1
  Le client peut gérer un panier d'achat en respectant les règles métier
  (stock, produit actif, calcul du total).
    And le total du panier est 30                          # features/steps/panier_steps.py:60 0.000s
    And le total du panier est 30                          # features/steps/panier_steps.py:60
    Then le panier est vide                                   # features/steps/panier_steps.py:55 0.000s
    Then le panier est vide                                   # features/steps/panier_steps.py:55
    Then le total du panier est 35                            # features/steps/panier_steps.py:60 0.000s
    Then le total du panier est 35                            # features/steps/panier_steps.py:60
  Scenario: US4 - Consulter un panier vide  # features/panier.feature:24
    Given un panier vide                    # features/steps/panier_steps.py:7 0.001s
    When je consulte le panier              # features/steps/panier_steps.py:40 0.000s
    Then le panier est vide                 # features/steps/panier_steps.py:55 0.000s

1 feature passed, 0 failed, 0 skipped
4 scenarios passed, 0 failed, 0 skipped
16 steps passed, 0 failed, 0 skipped
Took 0min 0.006s
```

III. Exécution des tests

→ **Ces tests permettent de garantir la cohérence fonctionnelle et la fiabilité du panier d'achat.**

test_us1_ajouter_produit.py

```python
import unittest

from PanierAchat import PanierAchat
from Produit import Produit


class TestUS1AjouterProduit(unittest.TestCase):

    def test_ajout_produit_valide(self):
        panier = PanierAchat()
        produit = Produit("Livre", 15, 5, True)

        panier.ajouter_produit(produit, 2)

        self.assertEqual(1, len(panier.get_lignes()))
        self.assertEqual(2, panier.get_lignes()["Livre"].get_quantite())
        self.assertEqual(30, panier.calculer_total())


if __name__ == "__main__":
    unittest.main()
```

test_us2_supprimer_produit.py

```python
import unittest

from PanierAchat import PanierAchat
from Produit import Produit


class TestUS2SupprimerProduit(unittest.TestCase):

    def test_suppression_produit(self):
        panier = PanierAchat()
        produit = Produit("Livre", 15, 5, True)

        panier.ajouter_produit(produit, 2)
        panier.supprimer_produit("Livre")

        self.assertEqual(0, len(panier.get_lignes()))


if __name__ == "__main__":
    unittest.main()
```

test_us3_calcul_total.py

```python
import unittest

from PanierAchat import PanierAchat
from Produit import Produit


class TestUS3CalculTotal(unittest.TestCase):

    def test_calcul_total_panier(self):
        panier = PanierAchat()

        livre = Produit("Livre", 15, 5, True)
        stylo = Produit("Stylo", 5, 10, True)

        panier.ajouter_produit(livre, 2)
        panier.ajouter_produit(stylo, 1)

        self.assertEqual(35, panier.calculer_total())


if __name__ == "__main__":
    unittest.main()
```

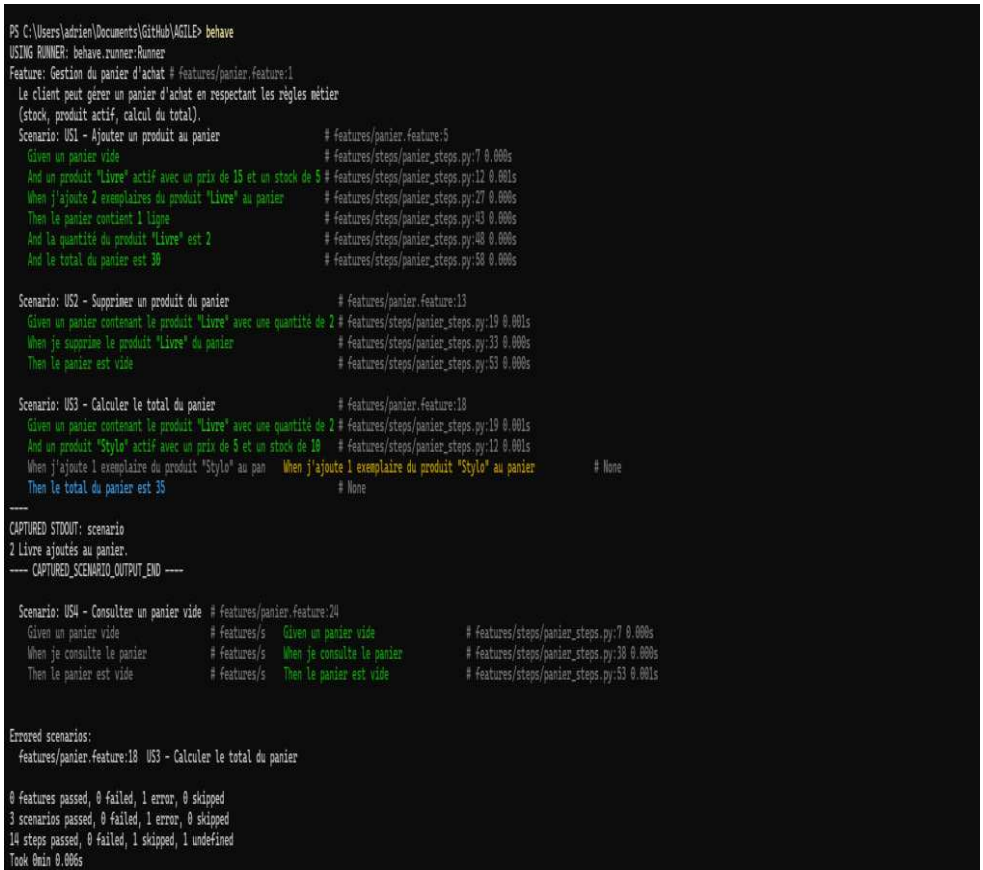
test_us4_consulter_panier.py

```python
import unittest

from PanierAchat import PanierAchat


class TestUS4ConsulterPanier(unittest.TestCase):

    def test_panier_vide(self):
        panier = PanierAchat()

        self.assertEqual(0, len(panier.get_lignes()))
        self.assertEqual(0, panier.calculer_total())


if __name__ == "__main__":
```

❖ **RUN**

Exécution des tests fonctionnels avec Behave.
Les scénarios US1, US2 et US4 sont validés avec succès.
Le scénario US3 échoue initialement en raison d'une étape non définie dans les steps Behave (1 exemplaire au singulier).
Le problème a été résolu en généralisant la définition de l'étape afin de gérer à la fois le singulier et le pluriel.

✓ Les quatre tests unitaires correspondant aux User Stories US1 à US4 ont été exécutés avec succès.
Tous les tests sont validés, confirmant le bon fonctionnement des méthodes et des règles métier du panier d'achat.



❖ Test unitaire – Association bidirectionnelle Panier ↔ Produit (US14)

L'objectif de ce test est de vérifier l'**association bidirectionnelle** entre les classes PanierAchat et Produit.

test_us14_association_bidirectionnelle.py

```python
import unittest

from PanierAchat import PanierAchat
from Produit import Produit


class TestUS14AssociationBidirectionnelle(unittest.TestCase):

    def test_produit_connait_son_panier(self):
        panier = PanierAchat()
        produit = Produit("Livre", 10, 5, True)

        panier.ajouter_produit(produit, 1)

        self.assertEqual(produit.get_panier(), panier)


if __name__ == "__main__":
    unittest.main()
```

Concrètement, lorsqu'un produit est ajouté à un panier :

- Le panier doit contenir le produit
- Le produit doit connaître le panier auquel il est associé

Résultat avant correction

Lors de l'exécution du test unitaire, celui-ci échoue avec l'erreur suivante :

AssertionError: None != <PanierAchat object>

```
PS C:\Users\adrien\Documents\GitHub\AGILE> python -m unittest tests/test_us14_association_bidirectionnelle.py
1 Livre ajoutés au panier.
F
======================================================================
FAIL: test_produit_connait_son_panier (tests.test_us14_association_bidirectionnelle.TestUS14AssociationBidirectionnelle)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\adrien\Documents\GitHub\AGILE\tests\test_us14_association_bidirectionnelle.py", line 15, in test_produit_connait_son_panier
    self.assertEqual(produit.get_panier(), panier)
AssertionError: None != <PanierAchat.PanierAchat object at 0x0000024EC06B2230>

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

Cela signifie que :

- Le produit a bien été ajouté au panier
- Mais le produit ne référence pas le panier
- L'association n'était donc pas réellement bidirectionnelle

Le test met en évidence un manque dans l'implémentation métier.

Correction apportée

Pour corriger ce problème, une référence au panier a été ajoutée dans la classe Produit, ainsi qu'une méthode d'accès contrôlée.

Lors de l'ajout du produit au panier :

- Le panier assigne explicitement sa référence au produit
- La cohérence bidirectionnelle est ainsi garantie

Résultat après correction

Après modification du code :

- Le test unitaire passe correctement
- Le produit connaît désormais le panier auquel il est associé
- L'association bidirectionnelle **0..1 ↔ * est fonctionnelle et robuste**

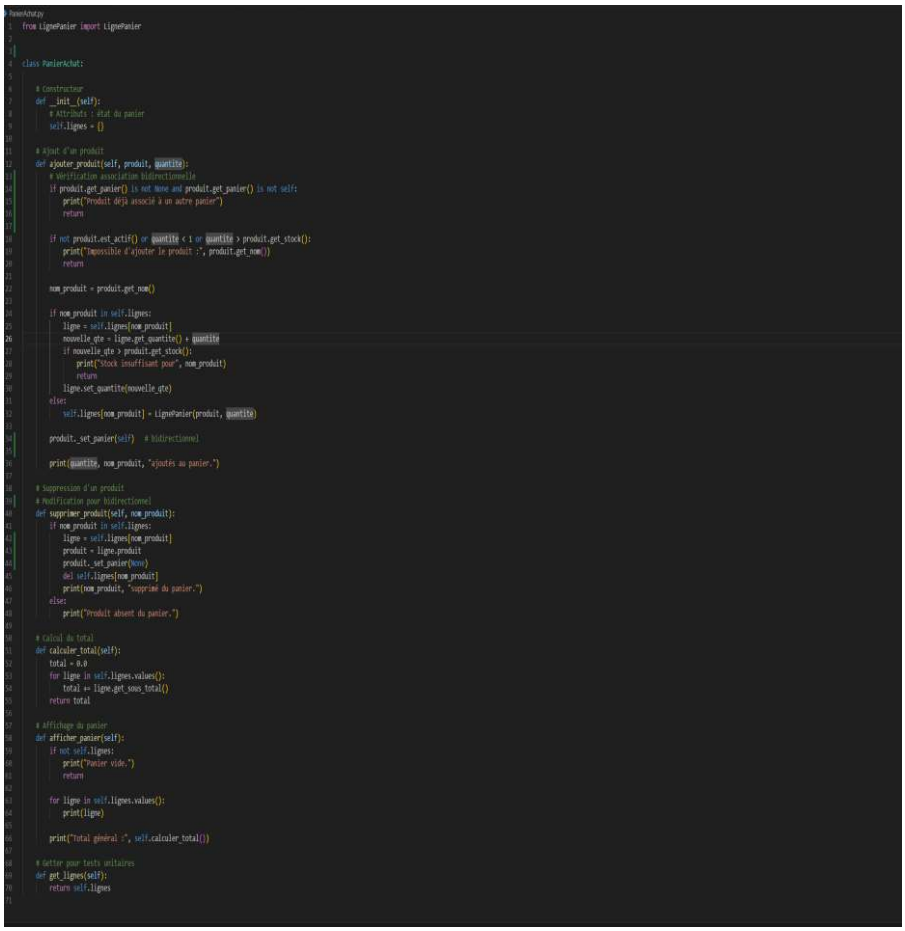
Le test valide donc correctement le comportement attendu.

```
PS C:\Users\adrien\Documents\GitHub\AGILE> python -m unittest tests/test_us14_association_bidirectionnelle.py
1 Livre ajoutés au panier.
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
PS C:\Users\adrien\Documents\GitHub\AGILE>
```

IV. Refactoring du code

Deux techniques de refactoring ont été appliquées afin d'améliorer la lisibilité et la maintenabilité du code, sans modifier le comportement fonctionnel de l'application.

- Code d'avant



```python
from LignePanier import LignePanier


class PanierAchat:

    # Constructeur
    def __init__(self):
        # Attributs : état du panier
        self.lignes = {}

    # Ajout d'un produit
    def ajouter_produit(self, produit, quantite):
        # Vérification association bidirectionnelle
        if produit.get_panier() is not None and produit.get_panier() is not self:
            print("Produit déjà associé à un autre panier")
            return

        if not produit.est_actif() or quantite < 1 or quantite > produit.get_stock():
            print("Impossible d'ajouter le produit :", produit.get_nom())
            return

        nom_produit = produit.get_nom()

        if nom_produit in self.lignes:
            ligne = self.lignes[nom_produit]
            nouvelle_qte = ligne.get_quantite() + quantite
            if nouvelle_qte > produit.get_stock():
                print("Stock insuffisant pour", nom_produit)
                return
            ligne.set_quantite(nouvelle_qte)
        else:
            self.lignes[nom_produit] = LignePanier(produit, quantite)

        produit._set_panier(self)   # bidirectionnel

        print(quantite, nom_produit, "ajoutés au panier.")

    # Suppression d'un produit
    # Modification pour bidirectionnel
    def supprimer_produit(self, nom_produit):
        if nom_produit in self.lignes:
            ligne = self.lignes[nom_produit]
            produit = ligne.produit
            produit._set_panier(None)
            del self.lignes[nom_produit]
            print(nom_produit, "supprimé du panier.")
        else:
            print("Produit absent du panier.")

    # Calcul du total
    def calculer_total(self):
        total = 0.0
        for ligne in self.lignes.values():
            total += ligne.get_sous_total()
        return total

    # Affichage du panier
    def afficher_panier(self):
        if not self.lignes:
            print("Panier vide.")
            return

        for ligne in self.lignes.values():
            print(ligne)

        print("Total général :", self.calculer_total())

    # Getter pour tests unitaires
    def get_lignes(self):
        return self.lignes
```

Refactoring 1 : Rename

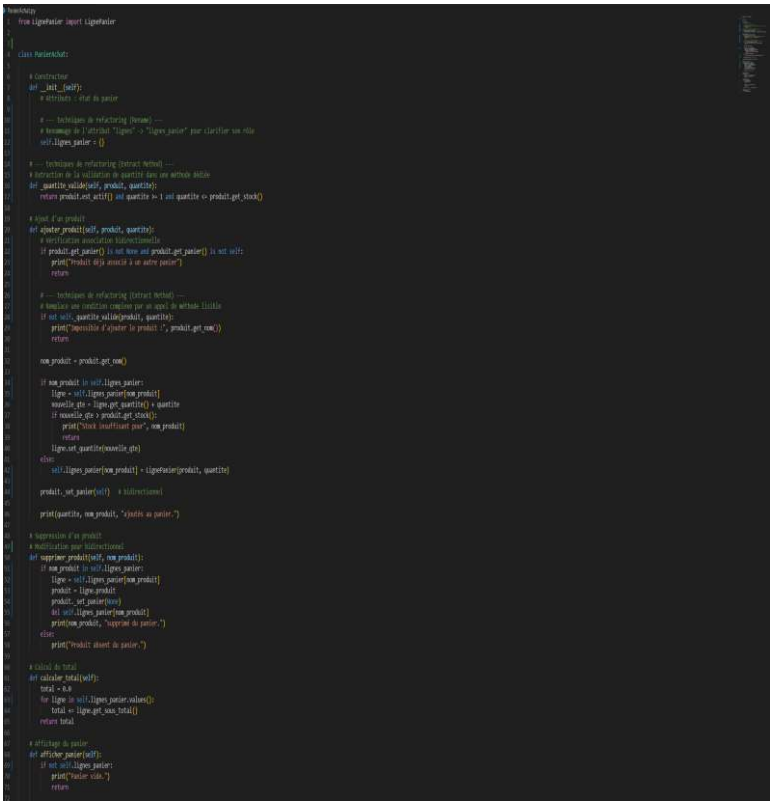
L'attribut interne du panier a été renommé afin de mieux refléter son rôle métier.
Ce renommage permet de rendre le code plus explicite et plus compréhensible lors de la lecture et de la maintenance.

Refactoring 2 : Extract Method

La logique de validation de la quantité et de l'état du produit a été extraite dans une méthode dédiée.
Cela permet de :

- Simplifier la méthode ajouter_produit,
- Éviter la duplication de logique,
- Améliorer la lisibilité et la testabilité du code.

❖ **Code après**



Validation

Après refactoring :

- Tous les tests unitaires passent,
- Les scénarios Behave restent fonctionnels,
- Aucun changement de comportement n'a été observé.

Les images avant / après illustrent l'amélioration structurelle du code tout en conservant la même logique métier.

V. Amélioration inspirée de l'article « Test Infected » (JUnit)

L'article Test Infected, publié sur le site officiel de JUnit, met en avant une approche de développement basée sur l'écriture de tests fréquents et automatisés afin d'obtenir un retour immédiat sur le comportement du code. Cette démarche permet de détecter rapidement les erreurs, de limiter les effets de bord et de faciliter les phases de refactoring.

❖ Amélioration proposée pour le projet Panier d'achat

En s'inspirant de cette approche, une amélioration équivalente a été appliquée au projet en renforçant la stratégie de tests autour des règles métier critiques.

L'amélioration consiste à ajouter systématiquement des tests qui échouent avant la correction, puis à adapter le code pour les faire réussir. Cette démarche a notamment été appliquée à :

- La gestion de l'association bidirectionnelle entre le panier et les produits,
- Le respect des contraintes métier (produit actif, stock disponible),
- La sécurisation des Refactorings réalisés ultérieurement.

❖ Application concrète

Un test unitaire a d'abord été écrit pour vérifier qu'un produit connaît le panier auquel il appartient.
Ce test a échoué dans un premier temps, révélant une incohérence dans le modèle objet.
Après correction du code, le test est devenu valide, confirmant la robustesse de l'implémentation.

Cette démarche illustre le principe « coder un peu, tester un peu » décrit dans l'article, et démontre l'intérêt d'une suite de tests active pour accompagner l'évolution du code.
L'intégration de tests inspirés de l'approche *Test Infected* a permis d'améliorer la fiabilité du projet, de détecter rapidement les erreurs et de réaliser des refactorings en toute confiance.
Cette méthode favorise un développement plus stable, plus lisible et plus maintenable.

VI. Exécution des tests en ligne de commande

Exécution des tests unitaires :

```
PS C:\Users\adrien\Documents\GitHub\AGILE> python -m unittest discover -s tests
1 Livre ajoutés au panier.
.2 Livre ajoutés au panier.
.2 Livre ajoutés au panier.
Livre supprimé du panier.
.2 Livre ajoutés au panier.
1 Stylo ajoutés au panier.
..
----------------------------------------------------------------------
Ran 5 tests in 0.002s

OK
```

Exécution des tests fonctionnels (BDD):

```
PS C:\Users\adrien\Documents\GitHub\AGILE> behave
USING RUNNER: behave.runner:Runner
Feature: Gestion du panier d'achat # features/panier.feature:1
  Le client peut gérer un panier d'achat en respectant les règles métier
  (stock, produit actif, calcul du total).
    And le total du panier est 30                          # features/steps/panier_steps.py:60 0.000s
    And le total du panier est 30                          # features/steps/panier_steps.py:60
    Then le panier est vide                                   # features/steps/panier_steps.py:55 0.000s
    Then le panier est vide                                   # features/steps/panier_steps.py:55
    Then le total du panier est 35                            # features/steps/panier_steps.py:60 0.000s
    Then le total du panier est 35                            # features/steps/panier_steps.py:60
  Scenario: US4 - Consulter un panier vide  # features/panier.feature:24
    Given un panier vide                    # features/steps/panier_steps.py:7 0.000s
    When je consulte le panier              # features/steps/panier_steps.py:40 0.000s
    Then le panier est vide                 # features/steps/panier_steps.py:55 0.000s

1 feature passed, 0 failed, 0 skipped
4 scenarios passed, 0 failed, 0 skipped
16 steps passed, 0 failed, 0 skipped
Took 0min 0.000s
```

VII. Loi de Murphy

« Tout ce qui est susceptible de mal tourner finira par mal tourner. »

Lors du développement du panier d'achat, une situation concrète illustre cette loi. Un test unitaire a été écrit pour vérifier l'association bidirectionnelle entre un produit et un panier. Bien que l'ajout du produit au panier semblât fonctionner correctement, le test a révélé que le produit ne connaissait pas le panier auquel il appartenait.

Cette erreur, non visible lors de l'exécution manuelle du programme, a été mise en évidence uniquement grâce aux tests automatisés. Sans ces tests, l'incohérence aurait pu passer inaperçue et provoquer des comportements incorrects lors de futures évolutions du code.