🚀 Scénario d'Intégration Agile pour le Projet "Pilot & Spaceship"

1. Rédiger les User Stories & Critères d’acceptance

User Story 1 : En tant que pilote, je veux pouvoir être assigné à un vaisseau pour piloter des missions.

Critères d’acceptance :

- Un pilote doit pouvoir être lié à un vaisseau.
- Un vaisseau peut accueillir plusieurs pilotes.
- Si un pilote est déjà assigné, il ne peut pas être ajouté à un autre vaisseau sans être retiré du premier.

User Story 2 : En tant que vaisseau, je veux pouvoir gérer mon carburant pour optimiser mes missions.

Critères d’acceptance :

- Le vaisseau doit pouvoir vérifier son niveau de carburant avant un voyage.
- Un voyage ne peut être lancé que si le carburant est suffisant.
- Si le carburant est insuffisant, un message d'erreur doit être retourné.

User Story 3 : En tant qu'ingénieur système, je veux pouvoir exécuter des tests automatiques pour garantir la robustesse du système.

Critères d’acceptance :

- Chaque composant (`Pilot`, `Spaceship`) doit avoir des tests unitaires.
- Une suite de tests doit être exécutée avant chaque nouvelle itération.

---

2. Planifier les Itérations (quoi / quand)

Itération	Objectifs
Sprint 1	Implémentation des classes `Pilot` et `Spaceship` avec leurs relations
Sprint 2	Ajout de la gestion du carburant et des voyages
Sprint 3	Création des tests unitaires pour `Pilot` et `Spaceship`
Sprint 4	Déploiement de tests fonctionnels et amélioration de la gestion des erreurs
Sprint 5	Intégration continue et exécution automatisée des tests

3. Écrire Code + Tests Unitaires & Fonctionnels

Implémentation du Code

Les classes `Pilot` et `Spaceship` sont développées avec une association bidirectionnelle permettant la gestion des affectations.

Exemple d'implémentation :

```python
class Spaceship:
    def __init__(self, name: str, fuel_level: int):
        self.name = name
        self.fuel_level = fuel_level
        self.pilots = []

    def add_pilot(self, pilot: "Pilot"):
        if pilot not in self.pilots:
            self.pilots.append(pilot)
            pilot.assign_spaceship(self)
```

Tests Unitaires

Utilisation de `unittest` pour valider les fonctionnalités de `Pilot` et `Spaceship` :

```python
class TestSpaceship(unittest.TestCase):
    def test_travel_success(self):
        spaceship = Spaceship("Enterprise", 100)
        result = spaceship.travel(30)
        self.assertEqual(result, "Le vaisseau Enterprise a voyagé !")
```

4. Archiver les Implémentations

L'ensemble des développements est stocké dans un **dépôt Git**, permettant de suivre chaque modification et de collaborer en équipe.

📌 **Commandes Git utilisées :**

```
git init
git add .
git commit -m "Ajout des classes Spaceship et Pilot"
git push origin main
```

5. Exécuter et Tester Automatiquement

L'intégration continue est assurée par **GitHub Actions** ou **Jenkins**, permettant de tester chaque modification avant son intégration finale.

Exécution automatique des tests :

```
python -m unittest discover Partie2 -v
```

6. Signaler et Corriger les Anomalies et Suivre l'Avancement

- **Utilisation d'un backlog Agile** pour suivre les tâches et prioriser les corrections de bugs.
- **Chaque anomalie est signalée dans un outil de gestion comme Trello ou Jira.**

📌 **Exemple d'un bug identifié et corrigé : Problème** : `ImportError : Circular Dependency Detected` 🚨 **Solution** : Utilisation de `if typing.TYPE_CHECKING:` pour éviter les importations circulaires.

7. Assurer la Reproductibilité & Développement Parallèle

Automatisation de l'environnement de travail

Un fichier `requirements.txt` permet d'installer toutes les dépendances nécessaires :

```
pip install -r requirements.txt
```

Développement Parallèle avec Branching Git

- **Branche **`main` : Code stable et validé.
- **Branches **`feature` : Nouvelles fonctionnalités en cours de développement.
- **Branche **`bugfix` : Corrections de bugs avant réintégration dans `main`.