



# Tutoriel Final : Pokémon au cinéma

Baptiste Matrat  
Jules Maurice  
Marie Probert  
Ferdinand Martin-Lavigne

## Je vais devenir une star !

Incroyable mais vrai : le monde des Pokémon et celui du Cinéma ne font plus qu'un ! Tu te retrouves face à un défi digne des plus grands producteurs. Les studios de Poké-Hollywood ont besoin de toi pour un projet révolutionnaire : faire jouer tes propres Pokémon dans les plus grands chefs-d'œuvre du cinéma.

Imagine un Dracaufeu dans un film d'action ou un Rondoudou dans une comédie musicale ! Ton rôle n'est plus seulement de coder, mais de devenir le pont entre ces deux univers. Tu vas devoir inviter tes Pokémon à franchir la frontière de leur propre code pour rejoindre le casting de tes films.

Le script est écrit, les stars sont dans leurs Pokéballs, et le tapis rouge est déployé...

Prêt pour la grande aventure cinématographique ? C'est à toi de jouer !





## Sommaire :

Je vais devenir une star !.....	1
Etape 1 : Fusionner les mondes.....	3
Etape 2 : Rendre nos Pokémons des acteurs avec le Design Pattern Adapter.....	6
Etape 3 : Le recrutement d'Acteurs avec le Design Pattern Factory.....	8
Etape 4 : Diagramme de classe du projet.....	9
Etape 6 : User Stories et tests fonctionnels.....	11
1. US_007 : L'Enrôlement des Stars (Le Casting).....	11
2. US_005 : Silence, Moteur... Action ! (La Performance).....	12
Etape 7 : Tests unitaires.....	15
Etape 8 : Couverture du code.....	16
Etape 9 : Optimisation et finitions du code.....	18





# Etape 1 : Fusionner les mondes

Originellement, nos deux univers vivaient chacun de leur côté ! D'un côté, nous avions le répertoire de tes Films, et de l'autre, celui de tes Pokémon. Mais pour que nos créatures puissent enfin devenir les stars du grand écran, il est temps de les réunir dans un seul et même projet... sans tout casser !

Première étape : La réunion sur GitHub. Nous avons commencé par rassembler les deux codes dans un projet commun. Mais attention, c'est un peu le désordre dans les loges ! Les noms de dossiers se répètent et se mélangent. Pour y voir clair, nous allons tout harmoniser : toutes tes classes doivent maintenant cohabiter dans le dossier Models. C'est là que Pokémons et Acteurs se préparent ensemble avant d'entrer en scène.

Comme tu peux le voir ici, nous avions deux projets distincts, chacun avec son propre laboratoire et ses propres dossiers :

The image shows two GitHub repository interfaces side-by-side. The left repository is named 'Tutoriel\_Film' and has a public status. It contains a 'main' branch, 1 branch, and 0 tags. The right repository is named '-Agilite--TPJU-Pokemon-MatratProbert' and also has a public status. It contains a 'main' branch, 1 branch, and 0 tags. Both repositories show a list of files and their descriptions.

Repository	Branch	Last Commit	Description
Tutoriel_Film	main	21-Ferdi-21	corrected all behave tests and made some changes
-Agilite--TPJU-Pokemon-MatratProbert	main	Baaptistee	test Infected

The 'Tutoriel\_Film' repository contents:

- \_\_pycache\_\_
- features
- .gitignore
- Agilité Itération 1 Tutoriel.pdf
- Agilité Itération 2 Tutoriel.pdf
- Categorie.java
- Film.java
- FilmTest.java
- README.md
- categorie.py
- film.py
- main.py
- realisateur.py
- test.py

The '-Agilite--TPJU-Pokemon-MatratProbert' repository contents:

- Code\_Java
- Code\_Python
- .DS\_Store
- .gitignore
- PYTHON Agile Pokemon\_Partie 2.pdf
- Projet Pokemon Agile.pdf



En fusionnant ces deux univers et en ne gardant que les éléments essentiels (ton code Python), tu obtiens une nouvelle structure :

-Agilite-TPJU-FilmPokemon Public

main 1 Branch 0 Tags Go to file Add file Code

Baaptistee test	86e7d47 · 13 minutes ago	6 Commits
Features made sure all tests are workin	45 minutes ago	
Models pas encore test	16 minutes ago	
Tests test	13 minutes ago	
.gitignore pas encore test	16 minutes ago	
main.py Code Pokemon	54 minutes ago	

Tu peux voir ici que les deux univers sont totalement fusionnés, sont dans des dossiers en commun :

- AGILITE-TPJU-FILMPOKEMON
  - Features
    - steps
    - afficher\_type\_pokemon.feature
    - capturer\_un\_pokemon.feature
    - US\_001\_gestion\_infos\_films.feature
    - US\_002\_modification\_duree.feature
    - US\_003\_bidirectionnalite.feature
  - Models
    - \_\_pycache\_\_
    - acteur.py
    - categorie.py
    - dresseur.py
    - film.py
    - pokeball.py
    - pokemon.py
    - realisateur.py
  - Tests
    - test\_films.py
    - tests\_pokemon.py





Et c'est maintenant le crash-test du tournage : C'est ici que tes réflexes de dresseur font la différence. Grâce à tous tes tests unitaires et tes récits utilisateurs (US), tu peux vérifier en un clic si cette cohabitation se passe bien.

Au début, certains tests risquent de passer au rouge : c'est normal, les fichiers ont changé d'adresse et le système est un peu perdu. Mais pas de panique ! En ajustant les liens et en relançant tes tests, tout rentre dans l'ordre.

Une fois que tout est au vert, le tournage peut enfin commencer !





## Etape 2 : Rendre nos Pokémons des acteurs avec le Design Pattern Adapter

**Actuellement, tes deux mondes fonctionnent chacun de leur côté.** Tes Pokémon sont prêts au combat et tes Films sont prêts à être tournés, mais ils ne se parlent pas encore. On veut qu'un Pokémon puisse enfin devenir une star de cinéma !

Pour commencer, il nous faut une classe **Acteur**.

```
acteur.py  X
Models > acteur.py > Acteur > set_dialogue > dialogue
1   from Models.film import Film
2
3   class Acteur:
4       def __init__(self, nom: str):
5           self._nom = nom
6           self._film = None
7           self._dialogue = None
8
9       def get_nom(self) -> str:
10          return self._nom
11
12      def set_nom(self, nom: str):
13          self._nom = nom
14
15      def get_film(self) -> Film:
16          return self._film
17
18      def set_film(self, film: Film):
19          self._film = film
20
21      def jouer_scene(self):
22          return self._dialogue
23
24      def set_dialogue(self, dialogue: str):
25          self._dialogue = dialogue
26
```



Mais un Pokémon n'est pas, à l'origine, un acteur : il n'a pas appris à suivre un script ou à se placer sous les projecteurs. Alors, comment faire pour qu'ils puissent jouer ensemble sans tout transformer ?

C'est là qu'intervient ce qu'on appelle un **Design Pattern** (un "modèle de conception"). C'est une solution standard et éprouvée pour résoudre un problème d'organisation précis. Pour notre situation, le pattern **Adapter** (l'Adaptateur) est parfait !



L'Adaptateur va “envelopper” ton Pokémon pour lui donner l'apparence d'un Acteur. Ainsi, le réalisateur du film verra un acteur, alors qu'en réalité, c'est un Pokémon qui fait le travail.

```
poke pokemon_acteur_adapter.py M ●
Models > poke pokemon_acteur_adapter.py > ...
1  from Models.acteur import Acteur
2
3  class PokemonAdapter(Acteur):
4      def __init__(self, pokemon):
5          super().__init__(pokemon.nom)
6          self.pokemon = pokemon
7
8      def jouer_scene(self):
9          action_base = self.pokemon.utiliser_capacite()
10
11         film = self.get_film()
12         titre_film = film.titre if film and hasattr(film, 'titre') else "un film inconnu"
13
14         capacite = self.pokemon.capacite if self.pokemon.capacite else "rien"
15
16         return f"{self.pokemon.nom} joue dans {titre_film} et utilise {capacite}"
17
18
```



## Etape 3 : Le recrutement d'Acteurs avec le Design Pattern Factory

Maintenant que nous avons nos Humains et nos Pokémon (grâce à l'Adaptateur), un nouveau problème se pose : le réalisateur ne veut pas s'occuper de toute la technique de création en coulisses. Il veut simplement dire : "Je veux un Humain" ou "Je veux un Pokémon", et que l'acteur apparaisse sur le plateau de son film, prêt à jouer.

C'est là qu'intervient un second Design Pattern : la **Factory** (la Fabrique).

Imagine la Factory comme une agence de casting. Au lieu de créer toi-même chaque acteur manuellement en vérifiant s'il s'agit d'un Pokémon ou d'un humain, tu délegues cette tâche à une classe spécialisée : la ActeurFactory.

```
pokemon_acteur_adapter.py acteur_factory.py X
Models > acteur_factory.py > ActeurFactory > creer_acteur
1  from Models.acteur import Acteur
2  from Models.pokemon import Pokemon
3  from Models.pokemon_acteur_adapter import PokemonAdapter
4
5  class ActeurFactory:
6      @staticmethod
7      def creer_acteur(type_source, nom, dialogue, film=None):
8
9          if type_source == "humain":
10              acteur = Acteur(nom)
11              acteur.set_dialogue(dialogue)
12              if film:
13                  acteur.set_film(film)
14              return acteur
15
16      elif type_source == "pokemon":
17
18          pokemon = Pokemon(nom=nom, type1="Inconnu", type2="Inconnu", capacite=dialogue)
19          acteur_adapter = PokemonAdapter(pokemon)
20          if film:
21              acteur_adapter.set_nom(film)
22          return acteur_adapter
23
24      else:
25          raise ValueError("Type de source inconnu pour créer un acteur")
```

Le réalisateur n'a pas besoin de connaître des détails sur comment créer un Pokémon ou un acteur classique, il lui suffit d'indiquer "humain" ou "pokemon" et la Factory s'occupe de créer les objets et de les associer au film !



## Etape 4 : Diagramme de classe du projet

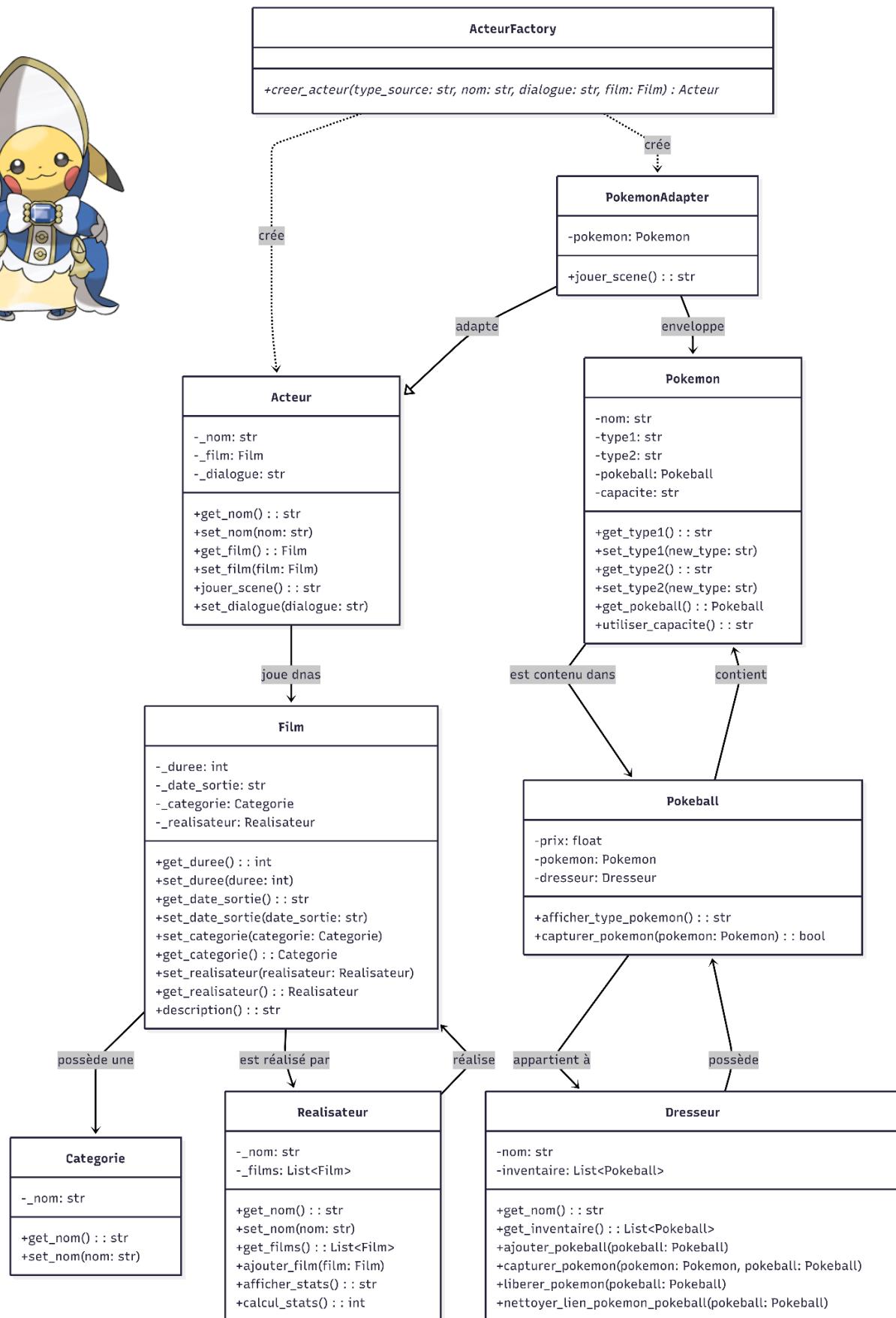
Maintenant que nous avons réuni nos projets et créé des ponts entre eux, notre résultat est impressionnant mais avec toutes ces classes qui communiquent entre elles, il est facile de s'y perdre. C'est ici qu'entre en jeu le **diagramme de classe**.

Imagine-le comme un plan d'architecte. Au lieu de lire des centaines de lignes de code pour comprendre qui fait quoi, on utilise un schéma visuel pour tout résumer. Dans ce diagramme, chaque classe (Pokémon, Film, Acteur...) est représentée par une boîte qui contient trois informations essentielles :

1. **Le nom** : Pour savoir de qui on parle.
2. **Les attributs** : Ce qui décrit l'objet (son nom, son prix, ses types...).
3. **Les méthodes** : Les actions qu'il peut faire (capturer, afficher...).

Mais le plus important, ce sont **les liens** entre ces boîtes. Elles nous montrent d'un seul coup d'œil comment un Dresseur possède des Pokémon, ou comment un Adaptateur permet à un Pokémon de rejoindre le casting d'un Film.

Le diagramme de classe du projet est ci-dessous.





## Etape 6 : User Stories et tests fonctionnels

Nous ne nous contentons plus de capturer des créatures ; nous gérons désormais la carrière de véritables stars de cinéma avec des tests fonctionnels avec Behave (comme dans les tutoriels précédents !).

Chaque scénario suit une structure logique :

- **Given (Étant donné)** : Le contexte initial.
- **When (Quand)** : L'action que tu fais.
- **Then (Alors)** : Le résultat attendu.

Voici le détail des deux nouvelles fonctionnalités (User Stories) qui permettent de transformer un simple Pokémon en tête d'affiche.

### 1. US\_007 : L'Enrôlement des Stars (Le Casting)

```

pokemon_acteur_adapter.py  US_007_Casting_Pokemon.feature  main.py
Features > US_007_Casting_Pokemon.feature
  Feature: US_004 Casting d'un Pokémons pour un film
    En tant que Agent de Casting
    Je veux attribuer un film à un Pokémons (considéré comme un Acteur)
    Pour l'enregistrer officiellement dans la distribution du projet

    Scenario Outline: Enregistrer un Pokémons dans un film
      Given un Pokémons nommé "<nom_pokemon>"
      And un film
      When j'attribue le film au Pokémons
      Then le Pokémons est associé à un film
      And le nom de l'acteur est bien "<nom_pokemon>"

    Examples:
      | nom_pokemon |
      | Pikachu      |
      | Psykokwak   |
      | Mewtwo       |

```

L'Agent de Casting veut officialiser la participation d'un Pokémons à un long-métrage. Cette fonctionnalité répond au besoin critique de nos agents de casting : associer formellement un Pokémons à une production cinématographique.



Concrètement, le système permet désormais de prendre un Pokémon (considéré techniquement comme un "Acteur") et de l'inscrire à la distribution d'un film spécifique.

## 2. US\_005 : Silence, Moteur... Action ! (La Performance)

```

pokemon_acteur_adapter.py   US_005_Capacité_Pokémon.feature  main.py
Features > US_005_Capacité_Pokémon.feature
  1 Feature: US_005 Performance d'acteur Pokemon
  2
  3   En tant que Réalisateur
  4     Je veux que le Pokémon joue sa scène en utilisant sa capacité spéciale
  5     Afin de générer une action spécifique liée au film en cours
  6
  7     Scenario Outline: Un Pokémon joue sa scène dans un film
  8       Given un Pokémon nommé "<nom>"
  9       And sa capacité spéciale est "<capacite>"
 10      When le Pokémon joue sa scène
 11      Then l'action produite doit être "<nom> utilise <capacite>"
 12
 13      Examples:
 14      | nom        | capacite      |
 15      | Pikachu    | Tonnerre      |
 16      | Dracaufeu | Lance-Flammes |
 17      | Rondoudou | Berceuse     |

```

Le moment est venu pour nos Pokémon de prouver qu'ils ont le sens du spectacle. Le réalisateur est exigeant : il veut que la star du film réalise une action qui lui est propre, comme une attaque signature, pour rendre la scène inoubliable.

Le scénario commence donc par la confirmation de la star choisie, comme Rondoudou ou Dracaufeu, et on s'assure que ce dernier possède bien la capacité spécifique attendue pour le rôle. Au moment où le Pokémon joue sa scène, le système doit transformer cette action de combat en un véritable dialogue de cinéma. L'objectif est de vérifier que le texte qui ressort est parfaitement formaté sous la forme : « [Nom du Pokémon] utilise [Capacité] ». Par exemple, si notre star est Dracaufeu et qu'il lance son attaque favorite, le script devra afficher précisément : « Dracaufeu utilise Lance-Flammes ». C'est grâce à cette vérification que l'on s'assure que l'intégration entre nos deux univers est un succès total.



## On traduit ensuite nos histoires en code Python :

Behave va ensuite lire ton fichier texte et chercher les fonctions Python correspondantes. Dans "steps/nom\_de\_la\_feature.py", tu vas lier les phrases au code

```
Features > steps > casting_pokemon_steps.py > ...
1  from behave import when, then
2  from Models.pokemon_acteur_adapter import PokemonAdapter
3  from Models.film import Film
4
5  @when("j'attribue le film au Pokémons")
6  def step_impl_attribution_film(context):
7      if not hasattr(context, 'film'):
8          context.film = Film()
9
10     context.acteur = PokemonAdapter(context.pokemon)
11     context.acteur.set_film(context.film)
12
13 @then("le Pokémons est associé à un film")
14 def step_impl_verification_association(context):
15     film_associe = context.acteur.get_film()
16
17     assert film_associe is not None, "L'acteur n'a pas de film."
18     assert film_associe == context.film
19
20 @then('le nom de l\'acteur est bien "{nom_pokemon}"')
21 def step_impl_verification_nom(context, nom_pokemon):
22     nom_reel = context.acteur.get_nom()
23
24     assert nom_reel == nom_pokemon, f"Attendu: {nom_pokemon}, Reçu: {nom_reel}"
```



```
Features > steps > 🎬 capacite_pokemon_steps.py > ...
1  from behave import given, when, then
2  from Models.pokemon_acteur_adapter import PokemonAdapter
3
4  @given('sa capacité spéciale est "{capacite}"')
5  def step_impl_set_capacite(context, capacite):
6      context.pokemon.set_capacite(capacite)
7
8  @given('le Pokémon est casté dans un film ')
9  def step_impl_casting_contextualise(context):
10     from Models.film import Film
11     context.film = Film()
12
13     context.acteur = PokemonAdapter(context.pokemon)
14     context.acteur.set_film(context.film)
15
16  @when('le Pokémon joue sa scène')
17  def step_impl_action(context):
18      if not hasattr(context, 'acteur'):
19          context.acteur = PokemonAdapter(context.pokemon)
20
21      context.resultat_reel = context.acteur.jouer_scene()
22
23  @then('l\'action produite doit être "{phrase_attendue}"')
24  def step_impl_verification(context, phrase_attendue):
25      assert context.resultat_reel == phrase_attendue, \
26          f"Erreur : \nAttendu : {phrase_attendue}\nObtenu : {context.resultat_reel}"
```

Lance tes tests en tapant “behave” dans la console et si tous les tests sont verts, tu es prêt pour la suite de l'aventure !

```
Given un Pokémons nommé "Mewtwo"
s.py:5 0.001s
    And un film
n_duree_steps.py:4 0.001s
    When j'attribue le film au Pokémons
emon_steps.py:5 0.001s
        Then le Pokémons est associé à un film
emon_steps.py:13 0.001s
        And le nom de l'acteur est bien "Mewtwo"
emon_steps.py:20 0.001s
```

```
7 features passed, 0 failed, 0 skipped
17 scenarios passed, 0 failed, 0 skipped
70 steps passed, 0 failed, 0 skipped
Took 0min 0.081s
```



## Etape 7 : Tests unitaires

C'est l'heure de passer aux tests unitaires ! Nous avons décidé de rassembler les tests des films et les tests des pokémons en un seul script : [test.py](#).

On a bien les classes TestActeur, TestPokemon... précédentes, mais on va surtout en ajouter pour tester nos nouvelles fonctionnalités.

Concentrons-nous sur un exemple : la classe TestActeurFactory qui te permet de tester les méthodes de ce nouveau Design Pattern :

On essaie les 3 cas : la création d'un acteur humain, pokémon et avec un type non reconnu.

```
64  class TestActeurFactory(unittest.TestCase):
65      def test_creer_acteur_humain(self):
66          acteur = ActeurFactory.creer_acteur("humain", "Tom Hanks", "Salut", Film())
67          self.assertEqual(acteur.get_nom(), "Tom Hanks")
68
69      def test_creer_acteur_pokemon_nom(self):
70          acteur_pika = ActeurFactory.creer_acteur("pokemon", "Pikachu", "Pika!")
71          self.assertEqual(acteur_pika.get_nom(), "Pikachu")
72
73      def test_creer_acteur_erreur(self):
74          with self.assertRaises(ValueError):
75              ActeurFactory.creer_acteur("alien", "Zog", "Gloup")
```

Ces tests unitaires testent chaque ligne de code de la nouvelle classe : sans vouloir dévoiler la prochaine partie, on peut dire que tout le code est "couvert"...



## Etape 8 : Couverture du code

Comme tu l'as vu tout du long de ces tutos, le plus important est de s'assurer que nos tests couvrent toutes nos méthodes ! Pour ce faire, on installe le module “coverage” sur VSCode. Il permet de révéler quelles lignes de code ne sont pas couvertes par un test.

On lui fait lancer les tests en observant puis on lui demande les résultats avec des commandes bash.

Voici le premier résultat en lançant le coverage sur les tests unitaires. On utilise le fichier `test_films_pokemon.py` qui lance seulement 11 tests.

```
TOTAL          148    14    91%
PS C:\Users\Ferdinand\Documents\Dauphine\M2 Dauphine\Agile\git_agile\Agilite-TPJU-FilmPokemon> coverage run -m unittest .\Tests\test_films_pokemon.py
.....
Ran 11 tests in 0.001s

OK
PS C:\Users\Ferdinand\Documents\Dauphine\M2 Dauphine\Agile\git_agile\Agilite-TPJU-FilmPokemon> coverage report -m
Name           Stmts   Miss  Cover   Missing
----- 
Models\acteur.py      18      2    89%  13, 16
Models\acteur_factory.py  19      5    74%  18-22
Models\categorie.py     7       3    57%  3, 6, 9
Models\dresseur.py      18      7    61%  7, 10, 13, 16, 20, 23-24
Models\film.py          28      9    68%  17, 20, 23, 26, 32, 35-40
Models\pokeball.py      26      4    85%  6, 8, 18, 22
Models\pokemon.py        26      5    81%  17, 20, 23, 26, 29
Models\pokemon_acteur_adapter.py  8       0   100%
Models\realisateur.py    25      2    92%  10, 13
Tests\test_films_pokemon.py 77      1    99%  116

TOTAL           252     38    85%
```

En bas, on voit que 85% de nos lignes de codes sont couvertes par un test. Plus haut dans le tableau, la colonne “Missing” indique les lignes qui ne sont pas couvertes ! Cela nous aide pour ajouter des tests dans la bonne direction.

Comme nous avons décidé dans la dernière partie de regrouper tous les tests dans un seul fichier, nommé [test.py](#) et on y a ajouté quelques tests !

```
PS C:\Users\Ferdinand\Documents\Dauphine\M2 Dauphine\Agile\git_agile\Agilite-TPJU-FilmPokemon> coverage run -m unittest .\Tests\test.py
.....
Ran 19 tests in 0.002s

OK
PS C:\Users\Ferdinand\Documents\Dauphine\M2 Dauphine\Agile\git_agile\Agilite-TPJU-FilmPokemon> coverage report -m
Name           Stmts   Miss  Cover   Missing
----- 
Models\acteur.py      18      0   100%
Models\acteur_factory.py  19      1    95%  21
Models\categorie.py     7       0   100%
Models\dresseur.py      18      3    83%  7, 10, 13
Models\film.py          28      2    93%  26, 32
Models\pokeball.py      26      4    85%  8, 22, 26, 28
Models\pokemon.py        26      1    96%  12
Models\pokemon_acteur_adapter.py  11      5    55%  11-21
Models\realisateur.py    25      3    88%  10, 13, 16
Tests\test.py          113     1    99%  144

TOTAL           291     20    93%
```

On ajoute quelques tests et voilà, **nous passons à 93% !** 🎉



Regardons aussi la couverture de nos tests fonctionnels. Pour ce faire, on lance les commandes suivantes :

`coverage run -m behave`

`coverage report -m`

On obtient les résultats suivants :

Name	Stmts	Miss	Cover	Missing
Features\steps\afficher_type_pokemon_steps.py	23	0	100%	
Features\steps\bidirectionnalite.py	17	2	88%	20-21
Features\steps\capacite_pokemon_steps.py	20	1	95%	24
Features\steps\capturer_un_pokemon_steps.py	19	0	100%	
Features\steps\casting_pokemon_steps.py	16	0	100%	
Features\steps\common_steps.py	10	0	100%	
Features\steps\gestion_infos_films.py	19	0	100%	
Features\steps\modification_duree.py	15	0	100%	
Models\acteur.py	18	3	83%	13, 22, 25
Models\acteur_factory.py	19	13	32%	9-25
Models\categorie.py	7	0	100%	
Models\film.py	28	4	86%	17, 26, 32, 38
Models\pokeball.py	26	11	58%	8, 19-22, 25-31
Models\pokemon.py	26	8	69%	8, 12, 17, 20, 23, 26, 29, 35
Models\pokemon_acteur_adapter.py	11	0	100%	
Models\realisateur.py	25	9	64%	10, 13, 24-26, 29-32
TOTAL	299	51	83%	

**93 % pour l'unité et 83 % pour le scénario, c'est une performance digne d'un Maître Pokémon !** Ton système est maintenant robuste, tes Pokémon sont de vraies stars et ton code est prêt pour la postérité. Bravo !



## Etape 9 : Optimisation et finitions du code

**Tes Pokémon sont en pleine forme, les projecteurs sont réglés et les tests sont au vert !** Mais attention, avant la grande première au cinéma, il reste un détail crucial. Ton projet est le fruit d'une fusion entre deux univers différents : le monde des Films et celui des Pokémon. Les noms ne correspondent pas toujours, et des affichages de test ont pu se glisser entre tes lignes de code !

On va donc faire un grand nettoyage :

- **Le Conventions de nommage** : On repasse sur nos variables pour qu'elles portent toutes une tenue uniforme. Si les variables s'appellent parfois "nom" et d'autres fois "prenom", on uniformise tout ça avec "\_" devant chaque nom de variable. On harmonise aussi les noms des US et des steps.
- **Affichages dans le code** : On supprime les print qui traînent. Dans un vrai film, on ne veut pas entendre le réalisateur crier ses instructions en plein milieu de la scène.

N'oublie pas que tes tests unitaires sont là pour toi. Si tu changes un nom de variable et que tu te trompes, le test passera au rouge pour te dire où corriger.

Une fois que ton code est aussi propre qu'une Pokéball neuve, il ne reste plus qu'à lancer la projection.

Les lumières s'éteignent, le rideau se lève... Profite bien du show dans les salles de cinéma !