PROJETS DE DÉVELOPPEMENT : POKÉMON

par

Lucas ARIES
Terry TEMPESTINI

IFT785 - Approches Orientées Objets

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, 10 avril 2025

Projet Pokémon

Notre projet a été de reproduire le jeu Pokémon. Plus particulièrement la mécanique de combat entre Pokémons. Ce rapport présente notre projet et comment il a été construit.

Le code est disponible sur le répertoire GitHub suivant : https://github.com/terytmps/Pokemon

Architecture

Notre application Pokémon utilise le framework `Flask` et adopte une architecture en couches, inspirée du modèle `MVC`, pour garantir la **séparation des préoccupations**.

1. Vue (Présentation)

- `templates/` : Génère l'HTML côté serveur (Jinja2).
- `static/` : Fournit les CSS, JavaScript (pour l'interactivité client) et autres assets.

2. Contrôleur et Service

- `app.py` & `pokemon_app/api/` : Gèrent les requêtes HTTP entrantes via les routes `Flask` (Blueprints).
- `service/` : Orchestre la logique applicative, faisant le lien entre les contrôleurs et le modèle.

3. Modèle (Logique Métier & Données)

- **Logique Métier (`pokemon_app/core/`)** : Contient les règles fondamentales du jeu (Pokémon, Combat, IA, Niveaux, Types) et utilise des design patterns (Decorator, Strategy, Factory). Indépendant du web et de la BDD.
- **Accès aux Données (`pokemon_app/data/`)** : Gère la persistance dans la base de données `SQLite` (`instance/database.db`) via :
  - `models/` : Schémas ORM (ex : `SQLAlchemy`).
  - `repositories/` : Abstraction de l'accès aux données (CRUD).
  - `adapters/` : Conversion entre objets métier et objets BDD.

Cette structure favorise la modularité, la testabilité et la maintenance de l'application.

La partie contenant les tests regroupe les tests unitaires, les tests d'intégration et les tests end-to-end (figure 2) :

```
├── pokemon_app/                # Package principal de l'application
│   ├── api/                    # Contrôleurs (gestion des routes/requêtes HTTP)
│   │   ├── game_routes.py
│   │   └── shop_routes.py
│   ├── core/                   # Modèle (Logique métier principale)
│   │   ├── ai/                 #   ↳ Logique de l'IA
│   │   ├── battle.py           #   ↳ Mécaniques de combat
│   │   ├── factories/          #   ↳ Création d'objets (Pokemon, Joueur...)
│   │   ├── level/              #   ↳ Gestion des niveaux/XP/Stats
│   │   ├── move.py             #   ↳ Représentation d'une capacité
│   │   ├── player.py           #   ↳ Représentation du joueur
│   │   ├── pokemon.py          #   ↳ Représentation d'un Pokémon
│   │   └── pokemon_type/       #   ↳ Logique des types (Decorators)
│   │   └── ... (autres logiques métier)
│   └── data/                   # Modèle (Couche d'accès aux données)
│       ├── database.py         #   ↳ Configuration BDD
│       ├── models/             #   ↳ Modèles de données (schéma BDD)
│       │   ├── player_model.py
│       │   └── pokemon_model.py
│       └── repositories/       #   ↳ Logique d'accès (CRUD)
│           ├── player_repository.py
│           └── pokemon_repository.py
├── service/                    # Couche de service (logique applicative)
│   ├── game_service.py
│   └── initialization_service.py
├── static/                     # Vue (Fichiers statiques: CSS, JS, Images)
│   ├── css/
│   ├── img/
│   └── js/
├── templates/                  # Vue (Gabarits HTML)
│   ├── pages/                  #   ↳ Vues principales (pages complètes)
│   │   ├── game.html
│   │   └── menu.html
│   ├── partials/               #   ↳ Fragments de vues réutilisables
│   └── errors/                 #   ↳ Pages d'erreur
└── README.md                   # Documentation du projet
```

FIGURE 1 – Architecture

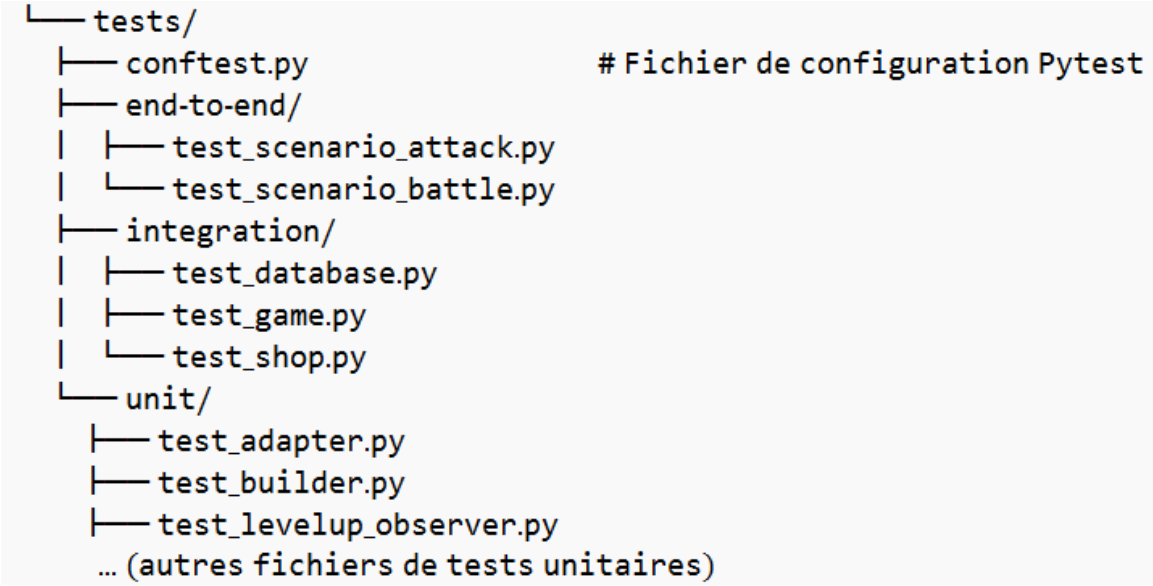

FIGURE 2 – Architecture des Tests

Design patterns

Decorator

Nous avons utilisé le *design pattern* décorateur pour gérer les types de nos Pokémon. Grâce à ce *design pattern*, nous restons ouverts à l'extension tout en respectant le principe de fermeture des classes. Nous étendons les fonctionnalités de notre Pokémon en lui ajoutant des décorateurs représentant son type, ce qui modifie son comportement.

De plus, cette méthode permet à un Pokémon d'avoir plusieurs types différents sans nécessiter de modification directe de son code. Enfin, grâce à l'abstraction, nous simplifions la gestion des résistances et des faiblesses de chaque Pokémon en fonction de son type.

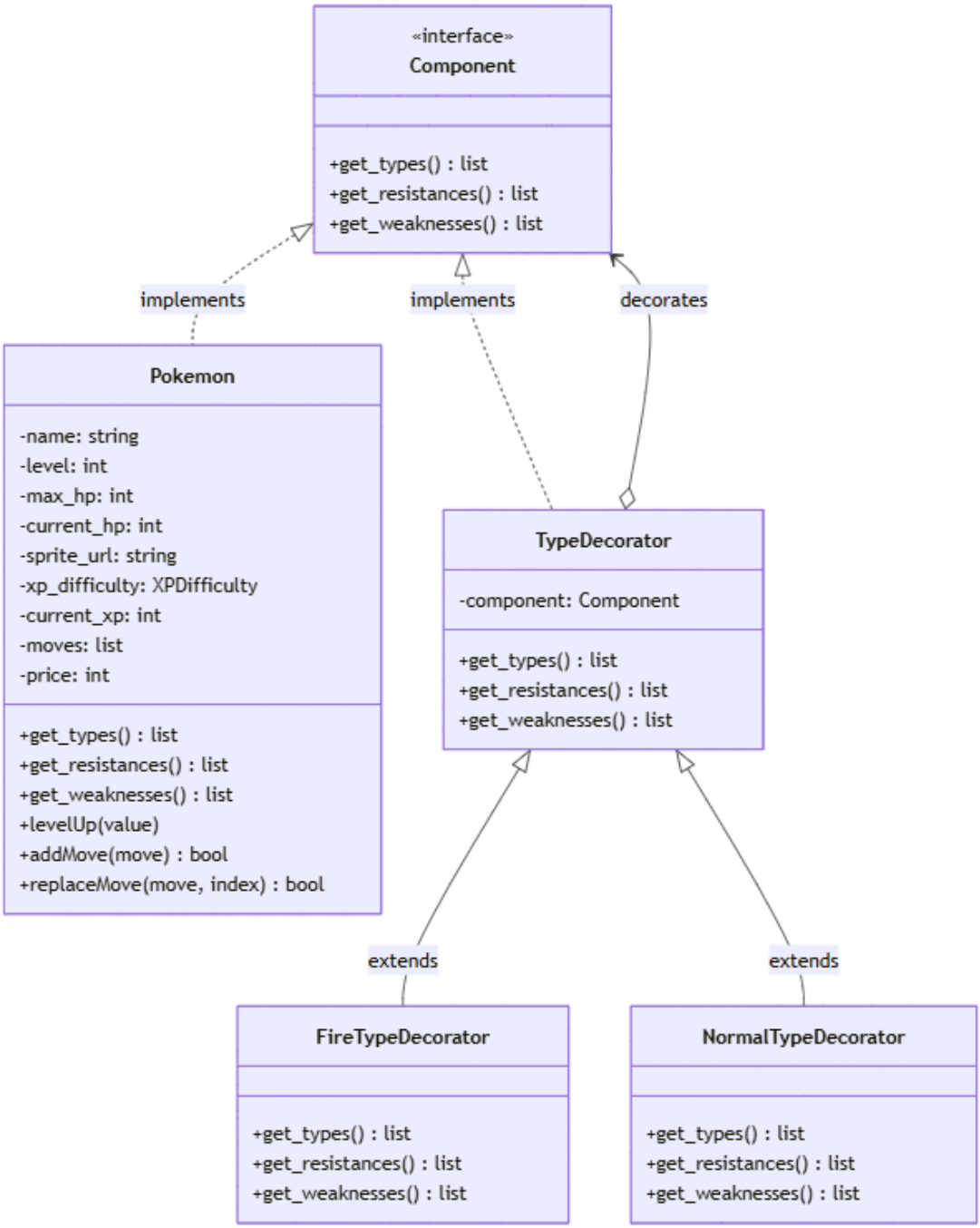

FIGURE 3 – Décorateur

Strategy

Nous utilisons le *design pattern* stratégie pour gérer les différents effets de statut qu'un Pokémon peut posséder. En effet, celui-ci ne peut en posséder qu'un seul à la fois, et chacun a des effets distincts.

Grâce à ce *pattern*, nous pouvons, à l'aide d'une interface, définir un contrat qui ne se soucie pas de l'implémentation. De plus, grâce au polymorphisme, chaque statut possède sa propre logique, indépendante de la classe `Pokémon`, assurant ainsi une bonne répartition des responsabilités.

Ce modèle est également ouvert à l'extension, car l'ajout d'un nouveau statut ne nécessite que la création d'une nouvelle classe.

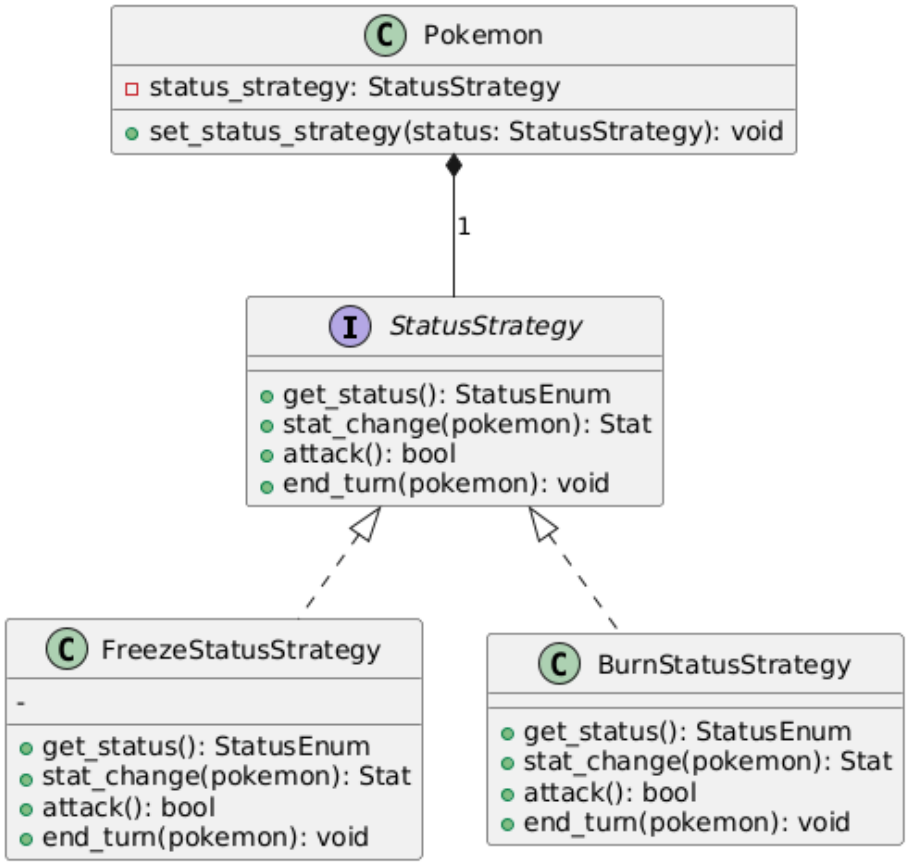

FIGURE 4 – Stratégie

Builder

Nous utilisons le *design pattern* Builder pour simplifier la création des Pokémon. Notre classe est très complexe à instancier, surtout si l'utilisateur n'est pas familier avec son implémentation.

En utilisant un *Builder*, nous facilitons grandement son instanciation en rendant le processus plus clair et structuré.

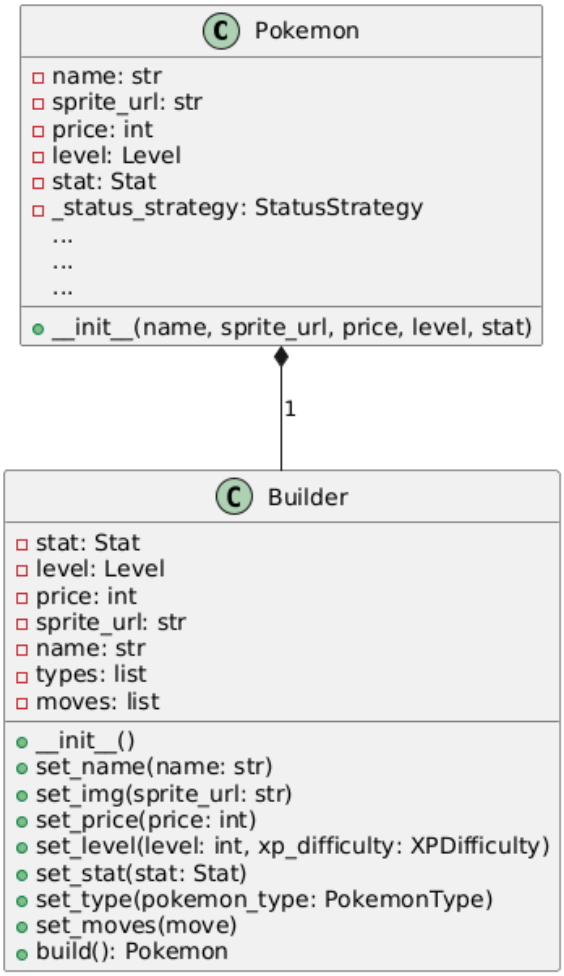

FIGURE 5 – Builder

Factory

Nous avons créé une factory pour simplifier la création de tous nos éléments récurrents (Pokémon, compétences d'attaque, etc.). Nous utilisons également le builder pour faciliter cette création. Grâce à cette factory, tous les Pokémon récurrents sont déjà disponibles et centralisés avec une abstraction qui rend leur instanciation facile.

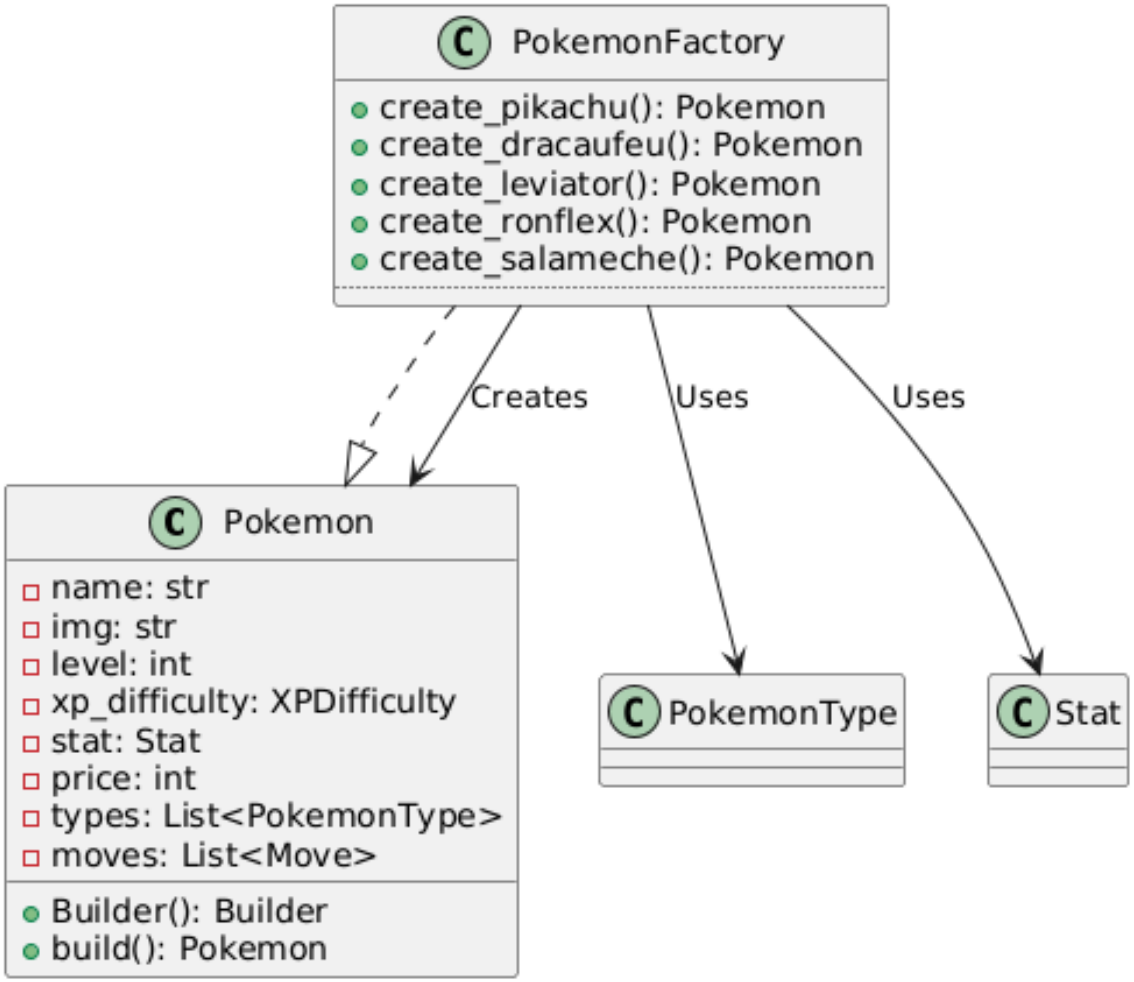

FIGURE 6 – Factory

Observer

Nous utilisons un observer pour que la gestion de l'XP d'un Pokémon et son level-up soient gérées automatiquement et puissent notifier les différents composants lorsqu'un événement se produit.

Par exemple, l'augmentation d'un niveau va notifier la classe `Stat` que les statistiques du Pokémon ont évolué, et celles-ci seront recalculées en fonction du nouveau niveau du Pokémon.

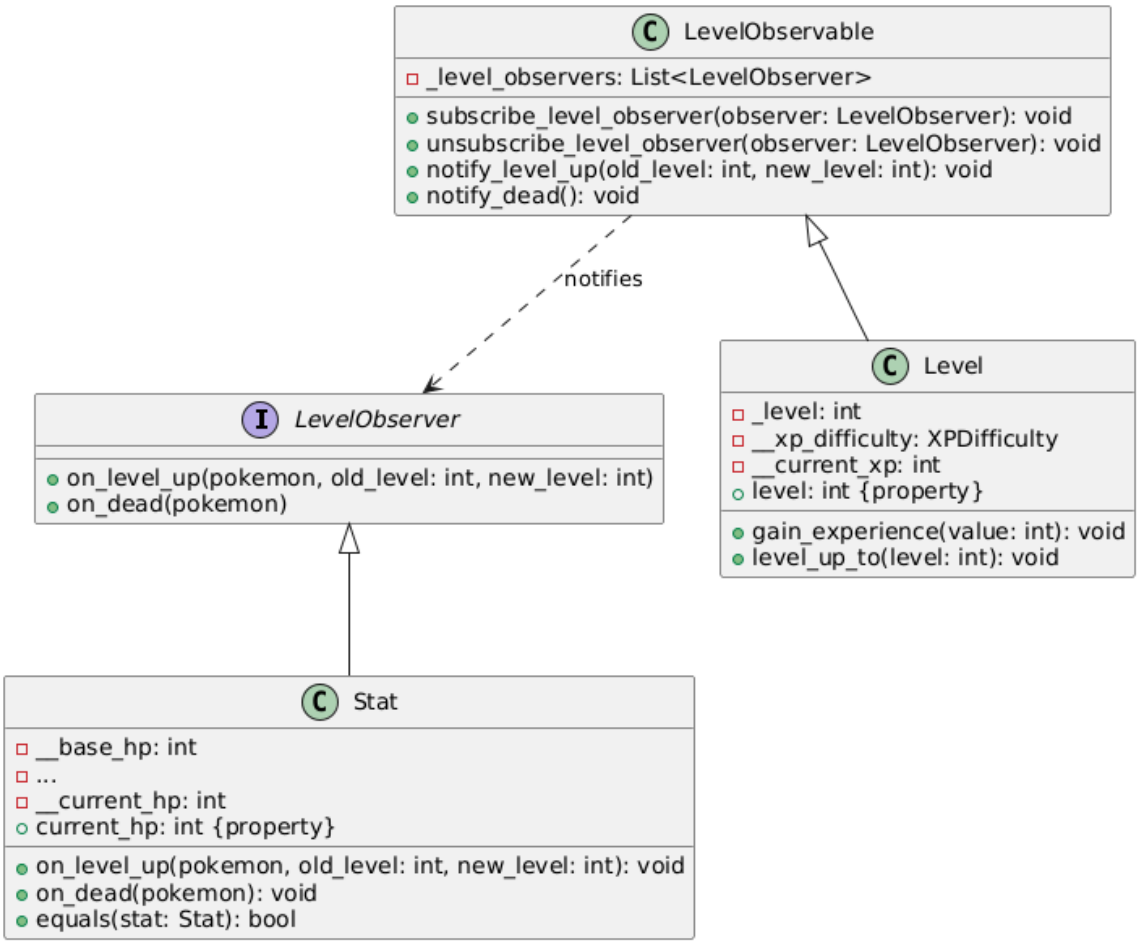

FIGURE 7 – Observer

Singleton

Nous utilisons le singleton pour la base de données et le générateur des ennemis (RoundGenerator). Grâce à cela, nous nous assurons d'avoir une seule instance de ces classes, ce qui est pertinent, car elles ont pour but d'être instanciées une seule et unique fois.

En utilisant un singleton, nous garantissons leur instanciation unique ainsi que la facilité de récupération de cette instance partout dans le code.

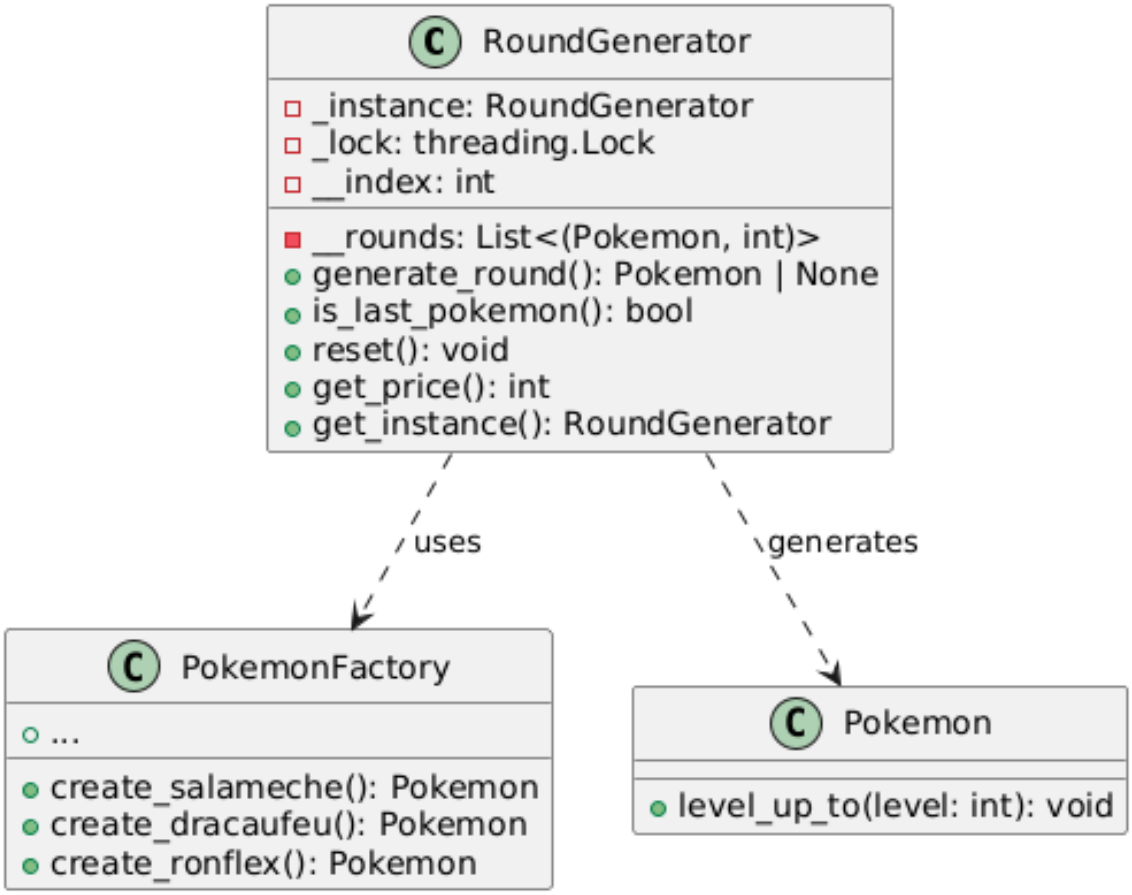

FIGURE 8 – Singleton

Adapter

Nous utilisons le pattern Adapter pour convertir nos objets de domaine en modèles de persistance, assurant ainsi une séparation claire entre la logique métier et la gestion de la base de données.

Ce mécanisme permet de faciliter l'évolution de l'application en rendant les changements au niveau de la persistance transparents pour le reste du système.

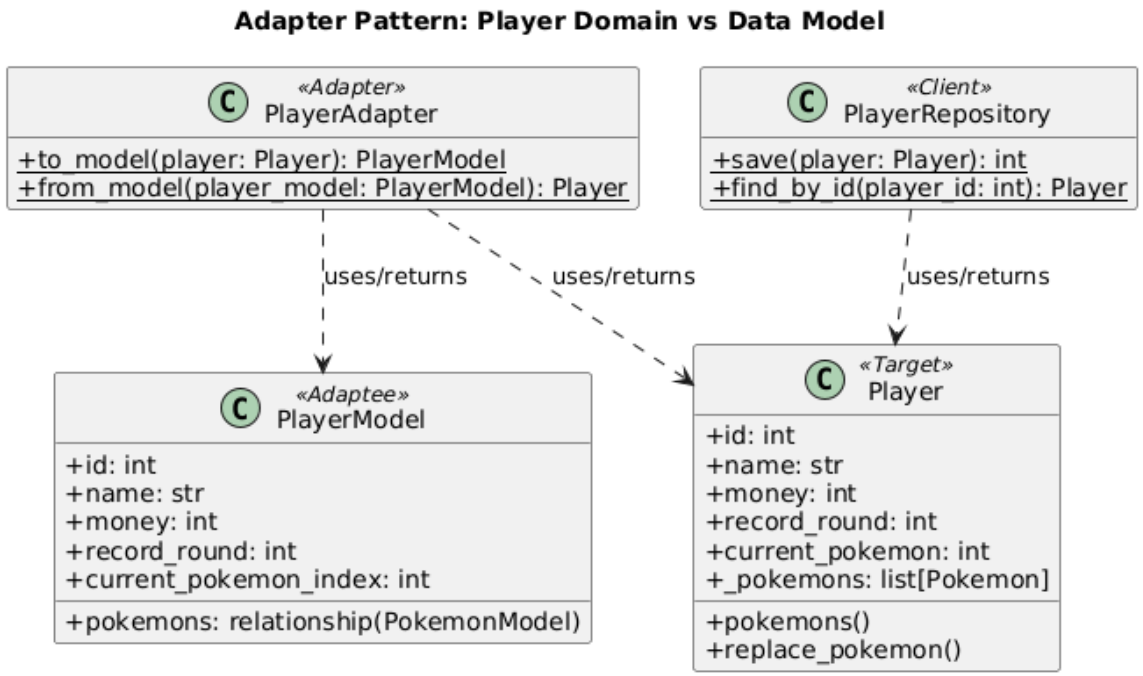

FIGURE 9 – Adapter

Description de l'API Pokémon

Notre API est basée sur Flask et comprend deux contrôleurs principaux :

Contrôleur de Boutique (Shop Controller)

— **Endpoint principal (`/`)** : Affiche le menu principal où le joueur peut acheter des Pokémon.

— **Achat de Pokémon (`/buy_pokemon`)** : Permet au joueur d'acheter un Pokémon du magasin et de l'ajouter à son équipe.
  — Effectue diverses validations (fonds suffisants, index valide).
  — Retourne les informations mises à jour sur l'achat et le joueur.

Contrôleur de Jeu (Game Controller)

— **Initialisation du combat (`/game`)** : Prépare et crée un nouveau combat.

— **Actions de combat** :
  — `/attack/<attack_id>` : Exécute une attaque choisie par le joueur.
  — `/change/<pokemon_id>` : Permet au joueur de changer de Pokémon actif.
  — `/forfeit` : Abandonne le combat en cours.

— **Informations de jeu** :
  — `/is/winner` : Vérifie si le joueur a gagné le combat.
  — `/battle_log` : Récupère le journal du dernier combat.

— **IA de l'adversaire (`/change_strategy`)** : Permet de modifier la stratégie utilisée par l'adversaire (aléatoire ou dégâts maximaux).

L'API utilise des modèles de données et des services pour gérer les joueurs, les Pokémon et les combats.

Guide déploiement

```
make venv
source .venv/bin/activate       # dépend linux/windows/mac
make install
make run
```

Description des Scénarios de Tests End-to-End

Voici une courte description des deux scénarios de tests end-to-end présentés dans votre code :

Test 1 : Achat d'Évoli, Combat et Abandon

Ce test simule le parcours complet d'un joueur qui :

- Accède à la boutique Pokémon.
- Achète un Évoli (prix : 15) et le place dans le premier emplacement de son équipe.
- Vérifie que l'achat a correctement modifié l'équipe du joueur et son argent.
- Entre en combat avec son nouveau Pokémon.
- Exécute sa première attaque contre l'adversaire.
- Vérifie que l'attaque est bien enregistrée dans les logs de combat.
- Abandonne le combat et retourne au menu principal.
- Confirme qu'aucune récompense n'est attribuée suite à l'abandon.

Test 2 : Navigation Shop-Combat et Vérification de l'Interface

Ce test vérifie l'expérience utilisateur complète en :

- Commençant par la boutique.
- Entrant en combat avec le Pokémon par défaut (Aquali).
- Exécutant la première attaque (Hydrocanon).
- Vérifiant en détail l'état du DOM (HTML analysé avec la bibliothèque BeautifulSoup) après l'attaque :
  - Présence correcte des noms des Pokémon (Aquali pour le joueur, Salamèche pour l'adversaire).
  - Mise à jour appropriée des points de vie des deux Pokémon.
  - Affichage correct des boutons d'attaque.

— Abandonnant le combat et confirmant le retour à la boutique.

Ces tests valident le flux d'utilisation complet de l'application et assurent que les interactions utilisateur, les mises à jour d'état, et les changements d'interface fonctionnent comme prévu.

Rapport de test avec métriques de couverture :

— **Outils utilisés** : `pytest`, `pytest-cov`
— **Nombre total de tests** : 45 tests collectés
— **Taux de réussite** : 100% des tests ont réussi
— **Couverture globale** : 93%

Détails par Module :

Module	Couverture
`pokemon_app/core/factories/pokemon_factory.py`	100%
`pokemon_app/core/move.py`	100%
`pokemon_app/core/pokemon_type/`	100%
`tests/`	100%
`app.py`	92%
`pokemon_app/api/game_routes.py`	96%
`pokemon_app/core/battle.py`	82%
`pokemon_app/core/pokemon.py`	94%
`pokemon_app/data/adapters/pokemon_adapter.py`	87%

TABLEAU 1 – Modules avec leur couverture de test.