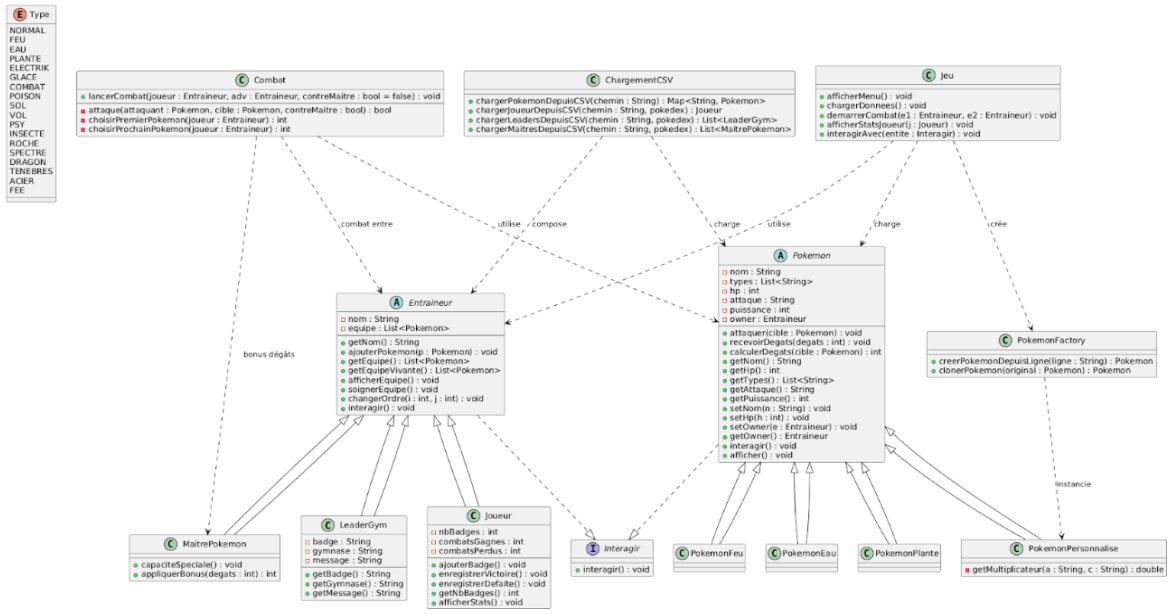
RAPPORT C++

Diagramme UML



Héritage et structure de classes

L'architecture du projet repose sur une hiérarchie claire de classes. La classe Pokemon est abstraite, et les types spécifiques de Pokémon héritent d'elle. De même, Entraineur est une classe de base abstraite dont héritent Joueur, LeaderGym et MaitrePokemon.

- ❖ Classe abstraite Pokemon.h

```cpp
#ifndef POKEMON_H
#define POKEMON_H

#include <string>
#include <vector>
#include <iostream>

class Entraineur; /// Cela informe le compilateur qu'un type nommé Entraineur existe, sans devoir inclure Entraineur.h


class Pokemon {
protected:
    std::string nom;
    std::vector<std::string> types;
    int hp;
    std::string attaque;
    int puissance;
    Entraineur* owner = nullptr; // propriétaire du Pokémon

public:
    void setOwner(Entraineur* o) { owner = o; }
    Entraineur* getOwner() const { return owner; } // Permet d'attribuer le propriétaire au pokemon
    Pokemon(std::string n, std::vector<std::string> t, int h, std::string a, int p)
        : nom(n), types(t), hp(h), attaque(a), puissance(p) {}

    virtual ~Pokemon() {}

    virtual void interagir() const = 0;

    std::string getNom() const { return nom; }
    int getHP() const { return hp; }
    void setHP(int h) { hp = h; }

    virtual int calculerDegats(Pokemon* cible) = 0;
    void recevoirDegats(int degats) {
        hp -= degats;
        if (hp < 0) hp = 0;
    }

    virtual void afficher() const {
        std::cout << nom << " [";
        for (size_t i = 0; i < types.size(); ++i) {
            std::cout << types[i];
            if (i < types.size() - 1) std::cout << "/";
        }
        std::cout << "] - HP: " << hp << " - Attaque: " << attaque << " (" << puissance << ")" << std::endl;
    }

    std::vector<std::string> getTypes() const { return types; }
    std::string getAttaque() const { return attaque; }
    int getPuissance() const { return puissance; }
};

#endif
```

- Classe abstraite Entraineur.h

```cpp
#ifndef ENTRAINEUR_H
#define ENTRAINEUR_H

#include <string>
#include <vector>
#include <iostream>
#include "Pokemon.h"
#include "Interagir.h"

class Entraineur : public Interagir {
protected:
    std::string nom;
    std::vector<Pokemon*> equipe;

public:
    Entraineur(std::string n) : nom(n) {}
    virtual ~Entraineur() {
        for (auto p : equipe) delete p;
    }

    std::string getNom() const { return nom; }

    void ajouterPokemon(Pokemon* p) {
    if (equipe.size() < 6) {
        p->setOwner(this); // lie le Pokémon à son dresseur
        equipe.push_back(p);
        }
    }

    std::vector<Pokemon*>& getEquipe() { return equipe; }
```

```cpp
    void afficherEquipe() const {
        std::cout << "Équipe de " << nom << " :\n";
        for (size_t i = 0; i < equipe.size(); ++i) {
            std::cout << i + 1 << ". ";
            equipe[i]->afficher();
        }
    }

    std::vector<Pokemon*> getEquipeVivante() const {
        std::vector<Pokemon*> vivants;
        for (auto* p : equipe) {
            if (p->getHP() > 0) {
                vivants.push_back(p);
            }
        }
        return vivants;
    }

    void soignerEquipe() {
    for (auto* p : equipe) {
        if (p->getHP() <= 0) {
            std::cout << p->getNom() << " est réanimé !\n";
        }
        p->setHP(100); // réanime + soigne
        }
    }
```

```cpp
    void changerOrdre(int i, int j) {
        if (i >= 0 && j >= 0 && i < equipe.size() && j < equipe.size()) {
            std::swap(equipe[i], equipe[j]);
        }
    }

    virtual void interagir() override = 0;
};

#endif
```

- Classe dérivée Joueur.h

```h
#ifndef JOUEUR_H
#define JOUEUR_H
#include "Entraineur.h"
class Joueur : public Entraineur {
private:
    int nbBadges;
    int combatsGagnes;
    int combatsPerdus;
public:
    Joueur(std::string n, int badges = 0, int g = 0, int p = 0)
        : Entraineur(n), nbBadges(badges), combatsGagnes(g), combatsPerdus(p) {}

    void ajouterBadge() { nbBadges++; }
    void enregistrerVictoire() { combatsGagnes++; }
    void enregistrerDefaite() { combatsPerdus++; }

    void afficherStats() const {
        std::cout << "Badges : " << nbBadges
                  << ", Victoires : " << combatsGagnes
                  << ", Défaites : " << combatsPerdus << "\n";
    }
    int getNbBadges() const {
        return nbBadges;
    }

    void interagir() override {
        std::cout << nom << " dit : Je suis prêt pour le prochain combat !\n";
    }
};
#endif
```

- Classe dérivée LeaderGym.h

```h
#ifndef LEADERGYM_H
#define LEADERGYM_H

#include "Entraineur.h"

class LeaderGym : public Entraineur {
private:
    std::string badge;
    std::string gymnase;
    std::string message;

public:
    LeaderGym(std::string n, std::string b, std::string g, std::string m)
    : Entraineur(n), badge(b), gymnase(g), message(m) {}

    std::string getBadge() const { return badge; }
    std::string getGymnase() const { return gymnase; }
    std::string getMessage() const { return message; }

    void interagir() override {
        std::cout << nom << " (Leader du gymnase " << gymnase << ") : " << message << "\n";
        }

};

#endif
```

- Classe dérivée MaitrePokemon.h

```
#ifndef MAITREPOKEMON_H
#define MAITREPOKEMON_H

#include "Entraineur.h"

class MaitrePokemon : public Entraineur {
public:
    MaitrePokemon(std::string n) : Entraineur(n) {}

    void interagir() override {
        std::cout << nom << " (Maître Pokémon) : Tu as obtenu tous les badges ? Montre-moi ta vraie force.\n";
    }

    static int appliquerBonus(int degats) {
        return static_cast<int>(degats * 1.25);
    }
};

#endif
```

Encapsulation et accesseurs

Les attributs de chaque classe sont déclarés en private ou protected, assurant la sécurité des données. L'accès se fait via des accesseurs et mutateurs (getNom(), getHP(), setHP(), etc.). Cela garantit une bonne encapsulation, limite les effets de bord et permet un contrôle précis sur la modification des états internes des objets. Ainsi, l'encapsulation se fait où il est possible de cacher ses attributs (protected) , là où il y a l'interface publique (getters/setters) et où il est possible de protéger ses invariants.

- ➔ Encapsulation
- ❖ Classe Pokemon.h

```cpp
#ifndef POKEMON_H
#define POKEMON_H

#include <string>
#include <vector>
#include <iostream>

class Entraineur; /// Cela informe le compilateur qu'un type nommé Entraineur existe, sans devoir inclure Entraineur.h


class Pokemon {
protected:
    std::string nom;
    std::vector<std::string> types;
    int hp;
    std::string attaque;
    int puissance;
    Entraineur* owner = nullptr; // propriétaire du Pokémon

public:
    void setOwner(Entraineur* o) { owner = o; }
    Entraineur* getOwner() const { return owner; } // Permet d'attribuer le propriétaire au pokemon
    Pokemon(std::string n, std::vector<std::string> t, int h, std::string a, int p)
        : nom(n), types(t), hp(h), attaque(a), puissance(p) {}

    virtual ~Pokemon() {}

    virtual void interagir() const = 0;

    std::string getNom() const { return nom; }
    int getHP() const { return hp; }
    void setHP(int h) { hp = h; }

    virtual int calculerDegats(Pokemon* cible) = 0;
    void recevoirDegats(int degats) {
        hp -= degats;
        if (hp < 0) hp = 0;
    }

    virtual void afficher() const {
        std::cout << nom << " [";
        for (size_t i = 0; i < types.size(); ++i) {
            std::cout << types[i];
            if (i < types.size() - 1) std::cout << "/";
        }
        std::cout << "] - HP: " << hp << " - Attaque: " << attaque << " (" << puissance << ")" << std::endl;
    }

    std::vector<std::string> getTypes() const { return types; }
    std::string getAttaque() const { return attaque; }
    int getPuissance() const { return puissance; }
};

#endif
```

- Classe Entraineur.h

```cpp
#ifndef ENTRAINEUR_H
#define ENTRAINEUR_H

#include <string>
#include <vector>
#include <iostream>
#include "Pokemon.h"
#include "Interagir.h"

class Entraineur : public Interagir {
protected:
    std::string nom;
    std::vector<Pokemon*> equipe;

public:
    Entraineur(std::string n) : nom(n) {}
    virtual ~Entraineur() {
        for (auto p : equipe) delete p;
    }

    std::string getNom() const { return nom; }

    void ajouterPokemon(Pokemon* p) {
    if (equipe.size() < 6) {
        p->setOwner(this); // lie le Pokémon à son dresseur
        equipe.push_back(p);
        }
    }

    std::vector<Pokemon*>& getEquipe() { return equipe; }
```

méthodes contrôlées, garantissant l'intégrité de l'objet

```cpp
    void afficherEquipe() const {
        std::cout << "Équipe de " << nom << " :\n";
        for (size_t i = 0; i < equipe.size(); ++i) {
            std::cout << i + 1 << ". ";
            equipe[i]->afficher();
        }
    }

    std::vector<Pokemon*> getEquipeVivante() const {
        std::vector<Pokemon*> vivants;
        for (auto* p : equipe) {
            if (p->getHP() > 0) {
                vivants.push_back(p);
            }
        }
        return vivants;
    }

    void soignerEquipe() {
    for (auto* p : equipe) {
        if (p->getHP() <= 0) {
            std::cout << p->getNom() << " est réanimé !\n";
        }
        p->setHP(100); // réanime + soigne
        }
    }
```

```cpp
    void changerOrdre(int i, int j) {
        if (i >= 0 && j >= 0 && i < equipe.size() && j < equipe.size()) {
            std::swap(equipe[i], equipe[j]);
        }
    }

    virtual void interagir() override = 0;
};

#endif
```

- Classe_Joueur.h

```
#ifndef JOUEUR_H
#define JOUEUR_H
#include "Entraineur.h"
class Joueur : public Entraineur {
private:
    int nbBadges;
    int combatsGagnes;
    int combatsPerdus;
public:
    Joueur(std::string n, int badges = 0, int g = 0, int p = 0)
        : Entraineur(n), nbBadges(badges), combatsGagnes(g), combatsPerdus(p) {}

    void ajouterBadge() { nbBadges++; }
    void enregistrerVictoire() { combatsGagnes++; }
    void enregistrerDefaite() { combatsPerdus++; }

    void afficherStats() const {
        std::cout << "Badges : " << nbBadges
                  << ", Victoires : " << combatsGagnes
                  << ", Défaites : " << combatsPerdus << "\n";
    }
    int getNbBadges() const {
        return nbBadges;
    }

    void interagir() override {
        std::cout << nom << " dit : Je suis prêt pour le prochain combat !\n";
    }
};
#endif
```

méthodes

➔ Afficher les statistiques du joueur : nombre de badges, combats gagnés et perdus (fichier joueur.h)

```
void afficherStats() const {
    std::cout << "Badges : " << nbBadges
              << ", Victoires : " << combatsGagnes
              << ", Défaites : " << combatsPerdus << "\n";
}
```

➔ Afficher les Pokémon et leurs attributs (fichier pokemon.h)

```
virtual void afficher() const {
    std::cout << nom << " [";
    for (size_t i = 0; i < types.size(); ++i) {
        std::cout << types[i];
        if (i < types.size() - 1) std::cout << "/";
    }
    std::cout << "] - HP: " << hp << " - Attaque: " << attaque << " (" << puissance << ")" << std::endl;
}
```

➔ Accesseurs et modificateurs de chaque classe

❖ Dans pokemon.h

getNom(), getHP(), setHP(), getTypes(), getAttaque(), getPuissance()

❖ Dans entraineur.h

getNom(), getEquipe(), getEquipeVivante()

- Dans joueur.h

getNbBadges()

Polymorphisme et comportement dynamique

Le polymorphisme est exploité dans les appels aux méthodes interagir() et calculerDegats() via des pointeurs de type abstrait (Pokemon*, Entraineur*, Interagir*). Le comportement réel exécuté dépend du type dynamique de l'objet ciblé. Par exemple, interagir avec un LeaderGym ou un MaitrePokemon appelle une implémentation différente sans changer le type du pointeur.

- main.cpp

```cpp
#include "IncludesGlobaux.h"

// Ajoute ici les fonctions chargerLeadersDepuisCSV et chargerMaitresDepuisCSV.
// [Fonctions supprimées ici pour compacité mais incluses dans le fichier]

int main() {
    srand(time(NULL)); // initialise le générateur de nombres aléatoires
    const std::string basePath = "./";
    auto pokedex = chargerPokemonDepuisCSV(basePath + "pokemon.csv");
    Joueur* joueur = chargerJoueurDepuisCSV(basePath + "joueur.csv", pokedex);
    auto leaders = chargerLeadersDepuisCSV(basePath + "leaders.csv", pokedex);
    auto maitres = chargerMaitresDepuisCSV(basePath + "maitres.csv", pokedex);

    std::vector<Entraineur*> vaincus;
    int choix = -1;
    while (choix != 0) {
        std::cout << "\n+++ MENU +++\n";
        std::cout << "1. Afficher l'équipe\n";
        std::cout << "2. Soigner l'équipe\n";
        std::cout << "3. Changer l'ordre des Pokémons\n";
        std::cout << "4. Statistiques\n";
        std::cout << "5. Affronter un leader\n";
        std::cout << "6. Affronter un Maître\n";
        std::cout << "7. MENU INTERACTION \n";
        std::cout << "8. Redémarrer le jeu\n";
        std::cout << "0. Quitter\n";
        std::cout << "Choix : ";
        std::cin >> choix;
```

....

```cpp
delete joueur;
for (auto& p : pokedex) delete p.second;
for (auto* l : leaders) delete l;
for (auto* m : maitres) delete m;
return 0;
```

Comportement dynamique (avecl ~Entraineur() et virtual ~Pokemon())

- Combat.h

```cpp
#ifndef COMBAT_H
#define COMBAT_H

#include "Entraineur.h"
#include "MaitrePokemon.h"
#include <thread>
#include <chrono>
#include <limits>

class Combat {
public:
    static void lancerCombat(Entraineur* joueur, Entraineur* adversaire, bool contreMaitre = false) {
        std::cout << "Début du combat entre " << joueur->getNom() << " et " << adversaire->getNom() << "\n";

        auto& eqJoueur = joueur->getEquipe();
        auto& eqAdversaire = adversaire->getEquipe();

        int i = choisirPremierPokemon(joueur);
        int j = 0;
        bool joueurCommence = true;

        while (i < (int)eqJoueur.size() && j < (int)eqAdversaire.size()) {
            Pokemon* pJoueur = eqJoueur[i];
            Pokemon* pAdv = eqAdversaire[j];

            std::cout << "\n" << joueur->getNom() << " envoie " << pJoueur->getNom() << "!\n";
            std::cout << adversaire->getNom() << " envoie " << pAdv->getNom() << "!\n";

            while (pJoueur->getHP() > 0 && pAdv->getHP() > 0) {
```

...

```cpp
private:
    static bool attaque(Pokemon* attaquant, Pokemon* cible, bool contreMaitre) {
        int degats = attaquant->calculerDegats(cible);
        if (contreMaitre && dynamic_cast<MaitrePokemon*>(attaquant->getOwner()) != nullptr)
            degats = MaitrePokemon::appliquerBonus(degats); /// applique le bonus uniquement si l'attaquant est un adver
        cible->recevoirDegats(degats);
        std::cout << attaquant->getNom() << " attaque " << cible->getNom()
                  << " et inflige " << degats << " dégâts. (HP: " << cible->getHP() << ")\n";
        return cible->getHP() == 0;
    }

    static int choisirPremierPokemon(Entraineur* joueur) {
        joueur->afficherEquipe();
        std::cout << "Quel Pokémon veux-tu envoyer en premier ? (1-" << joueur->getEquipe().size() << "): ";
        int choix;
        std::cin >> choix;
        while (choix < 1 || choix > (int)joueur->getEquipe().size() || joueur->getEquipe()[choix - 1]->getHP() <= 0) {
            std::cout << "Choix invalide. Réessaie : ";
            std::cin >> choix;
        }
        return choix - 1;
    }
```

- Pokemon.h

```cpp
    virtual ~Pokemon() {}

    virtual void interagir() const = 0;

    std::string getNom() const { return nom; }
```

```
    int getHP() const { return hp; }
    void setHP(int h) { hp = h; }

    virtual int calculerDegats(Pokemon* cible) = 0;
    void recevoirDegats(int degats) {
        hp -= degats;
        if (hp < 0) hp = 0;
    }

    virtual void afficher() const {
        std::cout << nom << " [";
        for (size_t i = 0; i < types.size(); ++i) {
            std::cout << types[i];
            if (i < types.size() - 1) std::cout << "/";
        }
        std::cout << "] - HP: " << hp << " - Attaque: " << attaque << " (" << puissance << ")" << std::endl;
    }

    std::vector<std::string> getTypes() const { return types; }
    std::string getAttaque() const { return attaque; }
    int getPuissance() const { return puissance; }
};

#endif
```

- ❖ Entraineur.h

```
    void changerOrdre(int i, int j) {
        if (i >= 0 && j >= 0 && i < equipe.size() && j < equipe.size()) {
            std::swap(equipe[i], equipe[j]);
        }
    }

    virtual void interagir() override = 0;
};

#endif
```

Implementation : Système de combat et logique de types

Le combat est tour par tour. Chaque Pokémon attaque selon sa puissance et ses types, en tenant compte des faiblesses et résistances de l'adversaire. Les multiplicateurs (×2, ×0.5) sont appliqués via des fonctions de correspondance de type. Les Maîtres Pokémon bénéficient d'un bonus de 25 % sur leurs attaques, renforçant le défi final.

- ➔ Système de combat (fichier combat.h)

```
static void lancerCombat(Entraineur* joueur, Entraineur* adversaire, bool contreMaitre = false)
    …
    // Boucle principale de combat :
    while (i < (int)eqJoueur.size() && j < (int)eqAdversaire.size()) {
        …
        // Chaque tour : attaque du joueur puis de l'adversaire (ou inversement)
        if (joueurCommence) {
            if (attaque(pJoueur, pAdv, contreMaitre)) break;
            if (attaque(pAdv, pJoueur, contreMaitre)) break;
        } else {
            if (attaque(pAdv, pJoueur, contreMaitre)) break;
            if (attaque(pJoueur, pAdv, contreMaitre)) break;
        }
        …
    }
    …
}
```

➔ Système de faiblesses et résistances (fichier PokemonPersonnalise.h)

```
int calculerDegats(Pokemon* cible) override {
    double multiplicateur = 1.0;
    std::string typeAttaque = this->types[0];
    for (const std::string& typeCible : cible->getTypes()) {
        multiplicateur *= getMultiplicateur(typeAttaque, typeCible);
    }
    return static_cast<int>(puissance * multiplicateur);
}
…
double getMultiplicateur(const std::string& attaquant, const std::string& cible) const {
    if (attaquant == "Feu") {
        if (cible == "Plante" || …) return 2.0;
        if (cible == "Eau" || …)     return 0.5;
    }
    // etc. pour Eau, Plante, Psy…
    return 1.0;
}
```

➔ Affronter un gymnase(fichier main.cpp)

```
case 5: {
    …
    auto* leader = leaders[choixLeader - 1];
    Combat::lancerCombat(joueur, leader);    //
    …
}
```

➔ Affronter un Maître Pokémon (fichier main.cpp)

```
case 6: {
    …
    auto* maitre = maitres[rand() % maitres.size()];
    Combat::lancerCombat(joueur, maitre, true);  // ←
    …
}
```

```
if (contreMaitre && dynamic_cast<MaitrePokemon*>(attaquant->getOwner()) != nullptr)
    degats = MaitrePokemon::appliquerBonus(degats);
```

Interface et interactions avec Pokémon et vaincus

es classes Pokemon et Entraineur possèdent des méthodes virtuelles pures, ce qui en fait des classes abstraites. Par exemple, Pokemon impose l'implémentation de calculerDegats() et interagir(), tandis qu'Entraineur impose interagir(). Une interface Interagir est également définie et implémentée par toutes les classes de dresseurs pour permettre une interaction unifiée dans le jeu.

- ❖ Classe Interagir.h

```
#ifndef INTERAGIR_H
#define INTERAGIR_H

class Interagir {
public:
    virtual void interagir() = 0;
    virtual ~Interagir() {}
};

#endif
```

Le menu propose une option d'interaction permettant de choisir entre interagir avec un Pokémon de l'équipe ou un entraîneur déjà vaincu. L'interface Interagir rend cela possible, car elle est implémentée par toutes ces entités. Chaque Pokémon a un comportement propre (message personnalisé) et chaque vaincu peut répondre avec un message fixe ou unique.

- ❖ Dans main.cpp

```
case 7: {
    // … Affichage du menu d'interaction …
    if (choixInter == 1) {
        // Interaction avec un entraîneur vaincu
        auto* v = vaincus[choixVaincu - 1];
        std::cout << "Interaction avec " << v->getNom() << " :\n";
        v->interagir();          // ← ici : appel à Entraineur::interagir()
    }
    else if (choixInter == 2) {
        // Interaction avec un Pokémon de l'équipe
        auto* p = equipe[choixPoke - 1];
        std::cout << "Interaction avec " << p->getNom() << " :\n";
        p->interagir();          // ← ici : appel à Pokemon::interagir()
    }
    // …
    break;
}
```

- ❖ Dans PokemonPersonnalise.h

```cpp
void interagir() const override {
    std::cout << nom << " est étonné de te voir.\n";
}
```

(bonus) Chargement dynamique des données

Les données du jeu sont stockées dans des fichiers CSV (pokemon.csv, joueur.csv, leaders.csv, maitres.csv) et chargées au lancement du programme. Des fonctions spécifiques comme chargerPokemonDepuisCSV() ou chargerLeadersDepuisCSV() permettent d'instancier dynamiquement les objets du jeu à partir des fichiers, en exploitant un dictionnaire (map) pour l'accès rapide par nom.

- chargement.cpp

```cpp
#include "IncludesGlobaux.h"

std::map<std::string, Pokemon*> chargerPokemonDepuisCSV(const std::string& chemin) {
    std::map<std::string, Pokemon*> pokedex;
    std::ifstream fichier(chemin);
    std::string ligne;
    std::getline(fichier, ligne);
    while (std::getline(fichier, ligne)) {
        Pokemon* p = creerPokemonDepuisLigne(ligne);
        if (p != nullptr) {
            pokedex[p->getNom()] = p;
        }
    }
    return pokedex;
}

Joueur* chargerJoueurDepuisCSV(const std::string& chemin, const std::map<std::string, Pokemon*>& pokedex) {
    std::ifstream fichier(chemin);
    std::string ligne;
    std::getline(fichier, ligne); // entête
    if (std::getline(fichier, ligne)) {
        std::stringstream ss(ligne);
        std::string nom;
        std::getline(ss, nom, ',');

        Joueur* joueur = new Joueur(nom);
        std::string pnom;
        while (std::getline(ss, pnom, ',')) {
            if (pokedex.count(pnom)) {
```