Master 1 Mathématiques et Applications

Parcours Data Science et Données Numériques

Année Universitaire 2025-2026

Rapport du Projet Deepsea Adventure

Présenté par :

- DANIELLE ISABELLE Nana Fotzeu
- KHAMASSI Mehdi
- WADE Abdou
- GEORGES Jean Constantin

Sous la direction de :

DAVOT-GRANGE Tom

Dans le cadre du module :

Programmation Oriente Objet

Préface

Ce rapport présente le travail complet réalisé pour l'implémentation du jeu Deep Sea Adventure dans le cadre du cours de programmation orientée objet. Ce projet représente l'aboutissement de plusieurs semaines de développement collaboratif et illustre notre capacité à concevoir une application Python complexe en respectant les principes d'architecture logicielle modernes.

1. INTRODUCTION

Le présent rapport décrit la conception et l'implémentation du jeu Deep Sea Adventure en Python. Ce jeu se joue de 2 à 6 joueurs et se base sur le mécanisme de jeu stop ou encore. L'objectif est de modéliser et simuler le jeu en Python selon une approche orientée objet tout en développant une architecture modulaire permettant deux interfaces distinctes : une interface ASCII(CLI) et une interface graphique (GUI) basée sur PySide6.

2. PRENSENTATION DU JEU

2.1 Architecture globale du projet

L'architecture de notre projet a été pensée pour séparer clairement la logique métier, les interfaces utilisateur (CLI/GUI) et les tests unitaires.

Le code est réparti en quatre dossiers principaux, chacun jouant un rôle précis dans l'organisation du jeu :

a) Dossier src/ **– Moteur du jeu**

Contient les classes centrales du cœur logique du jeu :

- Classes d'entités (joueurs, IA, plateau, cases, trésors)
- Classe Game qui orchestre toutes les mécaniques (tours, déplacements, air, actions, score)

Indépendance : Base fonctionnelle utilisable par CLI et GUI, sans dépendance à l'interface.

b) Dossier tests/ **– Tests unitaires**

Contient les tests automatisés pour valider le fonctionnement des classes src/.

Objectifs :

- Vérifier la conformité des règles
- Détecter les régressions
- Garantir la fiabilité du moteur

Structure : Tests unitaires isolés et tests d'intégration pour scénarios complets.

c) Dossier cli/ **– Interface textuelle ASCII**

Version ASCII/terminal du jeu avec :

- Configuration de partie ; Affichage texte du plateau
- Saisie manuelle des actions ; Intégration complète des IA

Utilité : Jeu rapide sans interface graphique, outil de test minimaliste.

d) Dossier gui/ **– Interface graphique (PySide6)**

Regroupe les composants graphiques :

- Widgets (plateau, cases, fiches joueurs, configuration)
- Fichier principal game_gui.py assemble les éléments visuels et utilise la classe Game

Avantages :

- Expérience intuitive et visuelle ; Affichage plateau en grille
- Animations (lancer de dés) ; Dialogues fins de partie ; etc.

3. MOTEUR DU JEU

3.1 Présentation de la classe Dice

- Cette classe **sert** à modélise les dés du jeu (2 dés à 3 faces par défaut). Centralise la gestion du hasard pour cohérence et testabilité.

- Elle est **paramétrée** par **num_dice** pour le nombre de dés (défaut 2) ; **faces** nombre de faces par dé (défaut 3) ; **rng** qui est le générateur aléatoire (injectable pour tests).

- Le **tirage** des dés est assuré par : **roll_individual()** qui retourne la liste des résultats individuels (débogage) et **roll()** qui retourne la somme des résultats (utilisée par le moteur).

Sérialisation

La classe inclut deux méthodes utilitaires : **to_dict()** qui exporte l'état du set de dés sous forme de dictionnaire JSON-sérialisable et **from_dict()** qui reconstruit un objet Dice à partir d'un dictionnaire.

Ces fonctions facilitent l'enregistrement ou la restauration d'une partie.

3.2 Présentation de la classe Space

- Cette classe sert a représenté une case du chemin sous-marin. Chaque case peut contenir des ruines, être vide, ou correspondre au sous-marin. Structure uniforme du plateau avec case 0 comme sous-marin.

- Ces fonctionnalités principales sont : **is_submarine** qui indique si c'est la case de départ (sans ruines) ; **ruins** qui représente la liste des RuinTile (pile LIFO fidèle aux règles) ; **max_stack_size** qui est une limite optionnelle (règle avancée : max 3 ruines) ; validation automatique lors de l'instanciation

Les propriétés dérivées utiles

- **has_ruin** : Au moins une ruine présente ; **ruin_count** : Nombre total de ruines
- **is_empty** : Case vide ; **top_ruin** : Tuile au sommet (sans retrait).

Manipulation des ruines

- Ajout : add_ruin() (avec vérifications) et push_ruin() (interne) / Retrait : pop_ruin() (sommet) et remove_all_ruins() (toutes).
- Fusion : pop_all_ruins_as_single() combine toutes les ruines en une seule tuile (somme valeurs, niveau max).

-L'affichage et la représentation se fait avec **__str__()** : "SUB" (sous-marin), "." (vide), "[Lx:y]" (1 ruine), "nR" (multiple ruines).

Sérialisation
RuinTile inclut des méthodes utilitaires pour sauvegarder et restaurer les tuiles : **to_dict()** → convertit une tuile en dictionnaire JSON-compatible et **from_dict()** → crée une nouvelle tuile à partir de données sérialisées.
Ces fonctionnalités sont essentielles pour la sauvegarde/récupération d'une partie ou pour les tests.

3.3 Présentation de la classe Board

Cette classe sert à représenter l'ensemble du plateau avec les cases en structure linéaire. Gère la configuration initiale et l'évolution du plateau pendant la partie. Case 0 = sous-marin.

Fonctionnalités principales

- Le plateau du jeu est construit par la fonction : **create_default()** elle crée un plateau fidèle au jeu original, génère les tuiles par niveaux (1-4), mélange optionnel et crée une case par tuile dans l'ordre de progression.
- On a accès aux cases avec **get_space(i) ou board[i]**

La représentation est assuré par **__str__()** qui donne un affichage compact type "SUB - [L1:0] - [L1:1] - ..." et **__repr__()** qui donne une représentation détaillée pour débogage

Sérialisation intégrée
Pour les sauvegardes ou les tests, Board inclut: **to_dict()** : export JSON-compatible (appelle Space.to_dict() pour chaque case) , **from_dict()** qui donne une reconstitution complète du plateau.

Cette fonctionnalité permet de : sauvegarder l'état d'une partie, générer des scénarios de test, restaurer un plateau identique à celui d'un moment précis du jeu.

Hooks d'extension et règles avancées

Le Board inclut deux méthodes essentielles pour gérer le déroulement de la manche selon les règles officielles et avancées du jeu **: 1) compress_path()** qui en fin de manche seules les cases contenant encore des ruines sont conservées, toutes les cases vides sont supprimées, le sous-marin reste toujours en première position. Cela traduit la mécanique de "réduction du chemin" entre deux manches du jeu.

2**) drop_tiles_to_bottom()** qui permet d'ajouter de nouvelles tuiles à la fin du chemin, les regrouper par piles de taille fixe (par défaut 3), etc.

3.4 Présentation de la classe Player

Cette classe sert à gérer l'identité, la position, l'état de manche, trésors portés et score du joueur. De plus elle fait la séparation entre état (Player) et stratégie (AIPlayer).

- **L'Initialisation et la gestion des manches** sont fait par : **reset_for_new_game**() qui efface le score et la réinitialise pour une nouvelle partie et **reset_for_new_round()** qui replace au sous-marin, vide les trésors portés, réinitialiser les indicateurs.
- **Propriétés dérivées** : **carrying_count** qui compte le nombre de trésors portés ; **total_score** qui somme les valeurs de tuiles sécurisées et **is_on_submarine** : Position sur case 0
- **La gestion de la position et de la direction** sont gérer par les méthodes : **move_to()** ; **start_going_back()** ; **continue_descending()** ; **mark_as_returned()**.
- **La Gestion des trésors est assurée par : take_ruin() / drop_ruin()**, **drop_all_carrying()**, **secure_carried_treasures()**.

Comme les autres classes cœur métier, Player est sérialisable : **to_dict()** : convertit l'état complet du joueur en dictionnaire JSON-compatible (y compris les trésors portés et les trésors de score, via RuinTile.to_dict())

3.5 Présentation de la classe AIPlayer

C'est une classe abstraite héritant de Player. Elle ajoute des méthodes de décision comme : **choose_direction()** un booléen pour remonter sinon : **choose_action()** : Choix "A"/"B"/"C" qui choisit l'action du tour.

Ces méthodes doivent être implémentées dans chaque sous-classe, ce qui rend le moteur de jeu flexible et extensible.

AIPlayer hérite de Player et initialise automatiquement un joueur comme étant contrôlé par l'IA (is_ai=True) ; peut être manipulée exactement comme un joueur humain dans le moteur de jeu ; permet de distinguer proprement les logiques de décision grâce à isinstance(player, AIPlayer).

Implémentations concrètes des IA

Le projet inclut trois IA aux comportements distincts : équilibrée, prudente et aventureuse. Ces IA sont faciles à interchanger, à tester et à améliorer.

On a donc AIPlayerNormal : IA "équilibrée » qui constitue une IA polyvalente et robuste ; **AIPlayerCautious** : IA prudente qui est utile pour simuler un joueur prudent ou débutant ; **AIPlayerAdventurous** : IA aventureuse qui produit des parties spectaculaires et souvent chaotiques, idéale pour tester des extrêmes du moteur de jeu.

3.6 Presentation de la classe Game : moteur central du jeu

Cette classe coordonne le plateau, les joueurs, les dés, l'air, les manches, les tours. Encapsule la logique métier uniquement.

Indépendance vis-à-vis de l'interface graphique : La classe Game ne fait aucun print(), ne lit aucun input (), ne connaît pas la notion de fenêtre, de bouton ou de clic.

Tout ce qu'elle fait c'est recevoir des paramètres simples (entiers, chaînes, codes d'action comme "A", "B", "C") ; renvoyer des structures de données propres (objets, dataclasses, dictionnaires) décrivant l'état du jeu.

La classe Game s'appuie sur plusieurs autres composants comme **Board** : plateau et tuiles de ruines ; **Player / AIPlayer** : état et stratégie des joueurs ; **Dice** : gestion des tirages de dés ; **TurnResult** : résumé d'un tour ; **PlayerState, SpaceState, GameState** : vues simplifiées pour l'interface etc.

Game maintient aussi l'état global de la partie : numéro de manche (**round_number**), nombre total de manches (**num_rounds**), réserve d'air (air), joueur courant (**current_player_index**), phase du jeu (**_phase**), journal des retours au sous-marin, etc

Cycle de vie d'une partie

Initialisation
Création avec vérification des joueurs (2-6), plateau par défaut, dés, air, numéro de manche et joueur courant. Réinitialisation des joueurs et passage en phase PLAYING.

Gestion des manches
Partie divisée en plusieurs manches. Début : réinitialisation air et états joueurs, détermination du premier joueur. Fin : application règles selon air ou retour joueurs, sécurisation/perte trésors, compression plateau. Passage manche suivante ou fin de partie.

Gestion des tours

1. Début tour : détermination direction, réduction air selon trésors, lancer dés et déplacement, production résultat.
2. Action : choix entre ne rien faire, ramasser ou déposer trésor.
3. Passage joueur suivant jusqu'à fin manche.

Fin de manche et partie

Manche termine si air épuisé ou tous joueurs retournés. Résolution selon cause. Partie termine après toutes manches.

État global pour UI

Méthode **get_state()** retourne l'objet structuré avec toutes informations nécessaires à l'interface.

Sérialisation
Sauvegarde/chargement via **to_dict()/from_dict()** pour persistance et rejeu.

Extensibilité
Architecture modulaire avec points d'extension pour nouvelles règles, phases et éléments.

4. INTERFACES GRAPHIQUES

Pour la partie graphique, nous avons choisi PySide6, l'implémentation officielle de Qt pour Python car PySide6 permet d'écrire l'interface en Python pur, ce qui s'intègre parfaitement avec le reste du projet (moteur de jeu, logique métier, IA…). A noter que l'interface fonctionne de la même façon sous Windows, macOS et Linux, sans modifications du code.

- La classe **SetupWindow** est la première fenêtre affichée dans la version graphique du jeu.

Elle sert d'écran de configuration avant de lancer une partie. son role principale est de permettre à l'utilisateur de configurer les joueurs avant de démarrer la partie.

- Structure générale de la fenêtre : **SetupWindow** hérite de **QMainWindow** et construit sa propre interface dans la méthode **_build_ui()**

- Configuration des joueurs :
Chaque joueur est configuré sur une ligne construite dans **_create_player_rows()** nous permettant de choisir le nombre de joueurs (avec un QSpinBox), le nom, le type, le niveau pour l'IA. Les informations de chaque ligne sont stockées dans une structure Python (**self.player_rows**) qui contient tous les widgets utiles (nom, radios, combo, etc.). Cela simplifie la récupération des données au moment de lancer la partie.

- Création des objets Player / AI et démarrage du jeu :
Lorsque l'utilisateur clique sur "Lancer la partie", la méthode **on_start_clicked()** parcourt les lignes visibles de **player_rows** et récupère le nom, le type, le niveau d'IA si nécessaire, ensuite crée les instances correspondantes.
Il appelle le callback **self.on_start_game(players)** en lui passant la liste des joueurs. C'est ce callback (défini dans main.py) qui se charge ensuite de : créer l'objet Game, ouvrir la fenêtre et fermer la SetupWindow.

C'est ce qui permet d'avoir une architecture propre, respectant le principe de séparation des responsabilités avec **BoardWidget** qui est responsable de l'affichage graphique du plateau, **SpaceWidget** qui donne la représentation graphique d'une case du plateau. Dans ce **SpaceWidget** on a choisi les couleurs en fonction de la profondeur (niveau), chaque niveau est associé à une palette visuelle : **vert** pour les valeurs faibles, **jaune** pour les valeurs qu'on a consideré comme neutre, **orange** pour dire attention on commence a atteindre les profondeurs et **rouge** danger elever.

Ce choix fournit un feedback immédiat dans la GUI : plus la tuile est profonde, plus elle est rouge. les cases vides sont representés par la couleur grise.

L'affichage des joueurs présents sur une case se fait avec **players_text =",".join(self.players_here)** et **text += f"\n{players_text}"**

Cela permet d'identifier immédiatement quels joueurs se trouvent sur une case et donc d'avoir une vision dynamique du plateau.

Ce mécanisme est utilisé notamment par BoardWidget, qui transmet la liste des joueurs présents à chaque SpaceWidget.

Fin de partie

Le composant **EndGameDialog** est une petite fenêtre modale (héritée de QDialog) affichée automatiquement lorsque la partie de Deep Sea Adventure est terminée qui afficher les scores finaux de tous les joueurs, mettre en évidence le vainqueur. Il permet la gestion de deux actions essentielles : rejouer ou quitter. Et aussi Cette fenêtre permet de clôturer proprement la session de jeu, que l'interface graphique soit console, GUI ou autre, grâce à un fonctionnement entièrement découplé du moteur de jeu.

Fenêtre principale de jeu

La classe **GameWindow** est la fenêtre principale du jeu. Elle fait le lien entre le moteur de jeu (Game), qui contient toute la logique métier et l'interface utilisateur PySide6 (widgets, boutons, labels, animations). Elle ne réimplémente aucune règle, du jeu elle se contente d'orchestrer les appels au moteur, de lire son état, puis de mettre à jour la vue.

Dans son constructeur, **GameWindow** reçoit éventuellement une liste de Player (créés depuis la SetupWindow) sinon, crée une partie par défaut avec un joueur humain et un bot.

Elle instancie un objet **Game, self.game=Game(players,num_rounds=3, air_per_round=25)** qui gère tout : déplacements, air, trésors, manches, fin de partie.

GameWindow crée aussi un BoardWidget pour représenter graphiquement le plateau : **self.board_widget = BoardWidget(self.game.board, self.game.players)**

Ensuite, elle construit l'interface avec _build_ui(), applique les styles (load_styles) et appelle _refresh_ui() pour synchroniser la vue avec l'état du jeu.

Gestion de fin de partie : EndGameDialog

Lorsqu'une partie est terminée GameWindow calcule les scores et les gagnants (**get_scores(), get_winners()**), ouvre un EndGameDialog(scores, winners selon la réponse (Accept ou Reject), elle appelle éventuellement **on_request_new_game()** (callback fourni, par ex. par main.py), ferme la fenêtre de jeu actuelle **(self.close()**).
Cela permet de rejouer proprement ou de quitter, sans mélanger cette logique dans la fenêtre principale.

Le fichier main.py est le point central de l'application graphique du jeu.

Il a trois rôles principaux : 1) Créer et configurer l'application Qt (QApplication); 2) Charger les styles globaux (thème graphique) ; 3) Orchestrer la navigation entre la fenêtre de configuration (SetupWindow), la fenêtre principale de jeu (GameWindow) et la fin de partie (via les callbacks).

Navigation : SetupWindow → GameWindow → SetupWindow

Le cœur de la logique de navigation repose sur deux fonctions internes :

1. show_setup() : Cette fonction ferme la fenêtre courante si nécessaire, crée une nouvelle SetupWindow, en lui passant un callback **on_start_game**.

Quand l'utilisateur clique sur **"Lancer la partie"** dans SetupWindow, celle-ci appelle le callback **on_start_game(players)**, qui pointe vers **start_game**.

start_game(players) appelle alors **show_game(players)** avec la liste de joueurs configurés.
SetupWindow ne connaît pas GameWindow, elle se contente d'appeler le callback. C'est main.py qui décide quoi faire ensuite : c'est une très bonne séparation des responsabilités.

2. show_game(players)
Cette fonction ferme la fenêtre courante si nécessaire, crée une nouvelle GameWindow, en lui passant la liste des players configurés, un callback on_**request_new_game**.

Lorsque la partie se termine, GameWindow affiche un EndGameDialog.

Si l'utilisateur clique sur "Rejouer", le EndGameDialog appelle accept(), puis GameWindow appelle on_request_new_game().

on_request_new_game() est ici lié à request_new_game(), qui lui-même rappelle show_setup().

Cycle complet de l'application
main() lance **show_setup()** → ouverture de la fenêtre de configuration.
L'utilisateur choisit les joueurs → SetupWindow appelle **on_start_game(players)** → **show_game(players)**. GameWindow crée un Game et gère la partie.

En fin de partie : GameWindow ouvre EndGameDialog, si l'utilisateur clique sur "Rejouer" → **on_request_new_game()** → **show_setup()** → retour à l'écran de configuration, sinon, la fenêtre est fermée et l'application se termine si aucune autre fenêtre n'est ouverte.

Le tout se termine par : **sys.exit(app.exec()**) qui lance la boucle d'événements Qt jusqu'à fermeture de l'application.

5. **TESTS**

5.1 Tests unitaires du moteur de jeu

Afin de garantir la fiabilité et la non-régression du moteur de jeu, j'ai mis en place une batterie de tests unitaires avec pytest sur les principales classes métier :

Space, RuinTile (testée indirectement), Player, Dice, Board, AIPlayer et ses variantes (AIPlayerNormal, AIPlayerCautious, AIPlayerAdventurous)

L'objectif de ces tests est double :

-Vérifier les règles du jeu et les invariants métiers (ex. : un sous-marin ne contient jamais de ruines, une pile ne dépasse pas sa taille maximale sauf cas particuliers, etc.).

-Assurer la non-régression : si j'améliore ou refactore le code, ces tests permettent de vérifier que le comportement reste cohérent avec les règles définies.

5.2 Tests sur Space (tests/test_space.py)

Les tests sur Space vérifient :

Les contraintes d'initialisation avec **max_stack_size**; les propriétés utilitaires avec **has_ruin, ruin_count**, **is_empty** et **top_ruin**; les opérations sur les ruines avec **add_ruin(),**

push_ruin() qui permet de dépasser volontairement max_stack_size (comportement différent de add_ruin testé explicitement), **pop_ruin()**, **remove_all_ruins()** et **pop_all_ruins_as_single()**.

5.3 Tests sur Player (tests/test_player.py)

Les tests Player portent sur :

- L'état initial et les resets caracterisé par les fonctions **reset_for_new_round()** et **reset_for_new_game()**.

- La gestion de la position et de la direction caracterisé par **move_to()**, **start_going_back()**, **continue_descending()**, **mark_as_returned()**.

- La gestion des trésors caracterisé par **take_ruin(), drop_ruin()**, **drop_all_carrying()**, **secure_carried_treasures()** et **drop_one_ruin()**.

- La représentation car la méthode str doit contenir le nom, le rôle (Humain/IA), la position, le nombre de trésors portés, et le score.

- La sérialisation avec **to_dict()** et **from_dict()**

- Tests de non-régression sur le fait que les listes carrying et score_tiles ne sont pas partagées entre deux instances (copie profonde logique).

5.4 Tests sur Dice (tests/test_dice.py)

Les tests sur les dés (Dice) assurent les paramètrages invalides (num_dice <= 0 ou faces <= 0) lèvent bien des ValueError et **roll_individual()** renvoie exactement **num_dice** résultats, chacun entre 1 et faces; etc.

La représentation str garde le format "XdY" (par exemple 2d3, 4d6), ce qui est utile à la fois pour l'UI texte et pour le debug.

5.5 Tests sur Board (tests/test_board.py)

Les tests Board couvrent la construction et la validation.

- un plateau sans case lève une erreur c'est a dire que la case d'index 0 doit être le sous-marin, sinon erreur ;

- L'accès et les propriétés sont gerés par **last_index** et **get_space() / getitem**.

- La sérialisation est gerer par **to_dict()** et **from_dict()**.

- Les fonctions de fin de manche sont gerer par **compress_path() et drop_tiles_to_bottom()**

5.6 Tests sur les IA (tests/test_ai_player.py)

Enfin, les tests sur les classes IA (AIPlayerNormal, AIPlayerCautious, AIPlayerAdventurous) vérifient toutes les caracteristiques d'un jour normal de plus la strategie (prudente, normale ou aventureuse)

6. INTERFACE CLI (ASCII)

Le fichier cli/cli_game.py implémente une interface textuelle (ASCII) permettant de jouer le jeu directement dans la terminale.

Elle a été conçue comme une alternative légère à la version graphique (GUI), permettant de tester rapidement la logique de jeu tout en conservant un fonctionnement complet et cohérent avec les règles de Deep Sea Adventure.

Dans notre jeu, la CLI repose principalement sur un seul fichier : cli_game.py

Ce module sert de point d'entrée console et joue le rôle d'un "contrôleur" minimal reliant l'utilisateur et le moteur du jeu (Game, Player, AIPlayer, etc.).

Elle utilise exclusivement des impressions textuelles (print) et des entrées utilisateur (input()), ce qui rend le système extrêmement simple mais entièrement fonctionnel.

Notre fichier CLI contient la fonction principale (main), c'est la boucle de jeu complete. Elle orchestre l'ensemble de la partie.

Elle s'organise en plusieurs etape :

a) Selection des joueurs

La fonction choose_players_cli() provient du module src.player_setup et a pour responsabilité de construire la liste des joueurs selon l'interaction utilisateur.

b) Creation de la partie

game = Game(players, num_rounds=3, air_per_round=25)

La CLI appelle directement le moteur du jeu en créant une instance de Game. Les valeurs de configuration sont fixées dans la CLI (ex. 3 manches)

c) Boucle principale du jeu

La CLI joue jusqu'à ce que : while not game.is_game_over():

Pour chaque manche, la CLI :

- Affiche les informations clés : par exemple le numero de manche, la quantite d'air restante, l'état du plateau en ASCII via game.get_board_ascii(), l'état des joueurs via game.get_players_status_ascii(), ces deux méthodes permettent d'obtenir un affichage textuel simple du jeu.
- Itère sur les joueurs : La CLI laisse chaque joueur jouer dans l'ordre du jeu tant que la manche n'est pas finie : player = game.players[game.current_player_index]

d) Gestion d'un tour (play turn)
Pour chaque joueur, la CLI :
- Détermine la direction (descendre / remonter)
  Si IA : utilise player.choose_direction(game.air)
  Si humain : pose la question

Cette étape est essentielle car la stratégie dans Deep Sea Adventure dépend fortement du choix de commencer ou non la remontée.

- Appelle game.begin_turn()
  result = game.begin_turn(player, going_back=go_back)
  Cela provoque : un jet de dé, un déplacement du joueur, potentiellement une gestion de manque d'air, la vérification si le joueur peut agir sur sa nouvelle case
- Action sur la case
  Si result.can_act_on_space est vrai :
  La CLI propose une action : pour IA : player.choose_action(space, air) et pour humain : un menu textuel.
  Elle valide l'entrée et l'envoie ensuite au moteur par : game.perform_action(player, action)
- Passage au joueur suivant avec : game.advance_to_next_player()
  La boucle continue jusqu'à ce que la manche se termine.

- Fin de manche et fin de partie
  La fin d'une manche est gerer par : game.end_round() et game.next_round(), ces méthodes calculent les scores des trésors ramenés au sous-marin, réinitialisent l'air, repositionnent les joueurs, ajustent les profondeurs de ruines (selon les règles). Et la CLI affiche un résumé pour la manche.
- À la fin de la partie on a:
  scores = game.get_final_scores()
  winners = game.get_winners(). Ces methode affichent le score final.

Notre cli est extensible dans le sens ou on pourrait ajouter des couleurs ou des styles, ajouter la possibilité de rejouer sans relancer le programme etc.

7. LIMITES ET PERSPECTIVES DU JEU

Limites actuelles et pistes d'amélioration

7.1 Système de sauvegarde et chargement

Le jeu possède déjà une base pour pouvoir sauvegarder et recharger une partie, mais cette fonctionnalité n'est pas encore totalement finalisée. Aujourd'hui, on peut sérialiser les éléments du jeu, mais il manque encore une vraie interface simple qui permet au joueur d'enregistrer sa progression, reprendre plus tard, et gérer plusieurs sauvegardes. C'est une amélioration naturelle pour rendre le jeu plus complet.

7.2 Graphismes et immersion

L'interface actuelle est claire et fonctionnelle, mais elle reste assez sobre (pas d'annimation, de sons, cartes non personnalisées etc.)

7.3 Intelligence artificielle

Les IA actuelles fonctionnent bien et proposent trois styles de jeu différents (prudente, normale, aventureuse). Mais elles restent encore simples : elles prennent leurs décisions au coup par coup, sans vraiment s'adapter au joueur ou anticiper les risques. On pourrait enrichir leur comportement pour qu'elles réagissent mieux à la partie, observent ce que font les autres plongeurs et deviennent un peu plus malignes.

8. RÉPARTITION DES RÔLES

La réalisation du projet s'est appuyée sur une bonne coordination entre les membres de l'équipe. Mehdi et Abdou ont collaboré étroitement pour développer le jeu : ils ont conçu l'architecture du code, programmé l'ensemble des fonctionnalités et assuré le bon fonctionnement technique du projet. Jean s'est consacré à la rédaction, la clarification et la correction des règles du jeu afin de garantir une description précise et accessible. Enfin, Isabelle a réalisé le dossier final, en rassemblant et présentant clairement l'ensemble du travail accompli.

9. CONCLUSION

La séparation claire des fonctionnalités et le respect des principes de la programmation orientée objet ont permis de construire un code lisible, bien structuré et facilement maintenable. Cette organisation rend le projet non seulement plus robuste, mais aussi scalable, c'est-à-dire capable d'évoluer facilement

si de nouvelles règles, options ou modes de jeu doivent être ajoutés dans le futur. Grâce à cette approche, le jeu peut continuer à grandir sans devenir complexe ou difficile à gérer.