Étudiant : **O. ROGELET, L. ROUILLE, V.RIDOUX, J.DEBOUDT, N.MOGIS**
Semestre : **A25**

LO21
PROGRAMMATION ET CONCEPTION
ORIENTÉES OBJET

Rendu suivi 3

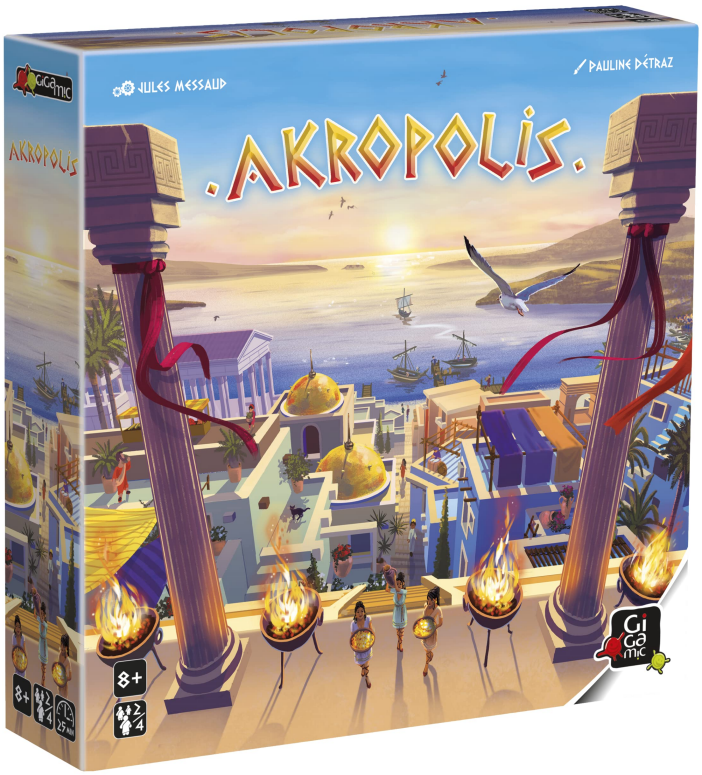

Table des matières

1. Introduction

Pour la troisième et avant dernière phase de notre projet de LO21, nous sommes fiers de proposer une version proche du résultat final. C'est d'ailleurs la partie la plus motivante du projet car nous voyons enfin tout le temps que nous avons investi dans notre programme se concrétiser.

L'objectif pour ce rendu était d'avoir une version minimaliste mais jouable d'Akropolis, notamment grâce à une interface console relativement avancée et un back-end quasiment terminé. En parallèle, notre travail nous permet aussi d'analyser les bugs potentiels et les optimisations (full-stack) à prévoir.

2. État d'Avancement depuis le livrable 2

2.1 Nouvelles tâches

T12 - Création de la classe Partie et refonte de l'Architecture « Console vs GUI » :

Après s'être rendu compte que l'architecture composée uniquement de la classe/du fichier Partie n'était pas modulable conformément aux attentes du sujet, nous avons opté pour la séparation de Partie en 3 composantes qui gèrent respectivement le moteur du jeu, l'interface console et l'interface graphique.

» **Chargé de la tâche :** Valentin

T13 - Création de la classe Pile :

La Pile qui ne devait servir qu'à piocher des cartes a finalement une importance capitale dans notre implémentation puisque nous avons décidé que c'était elle qui gérerait le cycle de vie des tuiles. En effet, les tuiles que le joueur utilise viennent dans un premier temps des piles, il était donc évident que ces dernières devaient permettre de créer et détruire les tuiles.

» **Chargée de la tâche :** Louane

T14 - Création de la classe Chantier :

Nous nous sommes rendu compte pendant la création de la classe Partie qu'une gestion de la pioche uniquement avec la classe Pile serait délicate et ferait perdre le jeu en modularité. Nous avons par conséquent choisi de créer une classe chantier qui gère spécialement les composantes du chantier décrit dans les règles du jeu (nombre de tuiles que le joueur peut piocher, nombre initial de tuiles...).

» **Chargé de la tâche :** Valentin

T15 - Amélioration et optimisation du back-end :

Tout au long du projet, nous nous sommes efforcé d'avoir une architecture respectant à la fois le principe d'encapsulation ainsi que de modularité comme demandé. Cependant, il est possible que certains

de ces aspects puissent encore évoluer d'ici la fin du projet car nous nous sommes dans un premier temps porté sur la fonctionnalité du jeu et avons parfois pu minimiser ces objectifs.

» **Chargé de la tâche :** Louane & Valentin

T16 - Sauvegarde de la partie :

L'objectif de la T16 est de pouvoir garder tout l'état du jeu pour pouvoir arrêter et reprendre une session (seul ou avec d'autres) sans perdre de données. Le système doit veiller à ce que la cité restaurée suive exactement les règles de placement qui sont en cours, tout en contrôlant bien la mémoire et les divers types de joueurs (humains et IA).

» **Chargé de la tâche :** Jeanne

2.2 Réaffectation des tâches

Au cours du développement, nous avons procédé à une réévaluation de la répartition des tâches pour optimiser notre efficacité et respecter les délais impartis. Afin de garantir une cohérence technique accrue sur les modules critiques, nous avons opéré les transferts de responsabilité suivants :

» **T5 :** Noémie → Louane & Noémie

» **T6 :** Noémie → Valentin & Louane & Noémie

» **T7 :** Oscar & Jeanne → Valentin

» **T8 :** Jeanne & Noémie & Oscar & Jeanne & Noémie → Oscar & Louane & Valentin

2.3 Tâches réalisées

T5 - Gestion du décompte des points :

» **Objectif :** Implémenter la logique complète de calcul des scores conformément aux règles officielles du jeu. Cela inclut la valorisation dynamique des quartiers en fonction de leur placement (adjacence, hauteur), l'application des multiplicateurs (places, étoiles) et la prise en compte des pierres restantes, tout en permettant l'activation optionnelle des variantes. Suite aux constats du Livrable 2, elle a fait l'objet d'une refonte majeure.

» **Sous-tâches :**

1. **Mise en place du Pattern Strategy :** Création d'une interface `RegleScore` et de classes distinctes pour chaque quartier (`RegleHabitation`, etc.). Cette structure sépare proprement les règles et simplifie l'ajout de variantes. → *Louane*
2. **Algorithme pour les Habitations (BFS) :** Développement d'un parcours en largeur pour identifier le « plus grand groupe » d'habitations. Cet algorithme détecte les connexions entre quartiers, même lorsqu'ils sont situés à des étages différents (z). → *Louane*
3. **Utilisation des Itérateurs :** Adaptation du calcul des points pour s'appuyer sur les itérateurs de la classe `Cite`. Cela assure que seules les tuiles visibles (non recouvertes) sont comptabilisées, avec le bon multiplicateur de hauteur ($z + 1$). → *Louane*
4. **Gestion des variantes :** Adaptation du calcul du score selon les variantes activées pour chaque type de quartier. Les classes variantes appliquent automatiquement les règles spécifiques (doublement du plus grand groupe, tuiles isolées, voisins manquants, temples élevés, etc.) → *Noémie*

» **Difficultés rencontrées :** Lors du précédent livrable, nous pensions avoir terminé cette tâche. Cependant, nous avons réalisé que notre première implémentation était trop simpliste : elle ne respectait pas certaines règles fines (comme la gestion du score du plus grand groupe d'habitations) et le code rendait l'intégration des variantes très risquée. Il a fallu accepter de « casser » l'existant pour reconstruire sur de meilleures bases.

» **Solution proposée :** L'adoption du pattern **Strategy** a permis de découpler chaque règle de score. Si une règle change (ou une variante est activée), on remplace simplement l'objet stratégie correspondant sans impacter le reste du moteur. L'utilisation des itérateurs a également fiabilisé l'accès aux données du plateau. Pour remettre score à jour avec les bonnes règles, la classe Score a été refondue en utilisant des stratégies par type de quartier et variante, avec un calcul des groupes basé sur Hexagone ::estPlace() et l'intégration automatique des pierres. Le code est désormais plus clair, modulable et facilement extensible.

» **Responsables :** Noémie & Louane

» **Progression :** 100% (Validé avec les règles officielles et variantes).

» **Temps passé :** *Total :* 35h Le dépassement du temps initialement prévu est dû à la dette technique du Livrable 2 qu'il a fallu corriger par cette réécriture complète. / *Louane :* 20h / *Noémie :* 15h (3h sur les variantes + 12h sur l'ancienne version de score)

T6 - Gestion du mode multijoueur :

» **Objectif :** Orchestrer une partie complète de 2 à 4 joueurs en respectant l'ordre des tours, la gestion des ressources partagées et les conditions de fin.

» **Sous-tâches :**

1. **Corps de Partie :** Le cœur du système repose sur la classe Partie (Singleton) qui agit comme chef d'orchestre. → *Noémie*
   - **Singleton Partie :** La classe Partie a été transformée en Singleton pour qu'une seule instance contrôle l'ensemble du déroulement de la partie.
   - **Constructeur privé :** Le constructeur de Partie est inaccessible de l'extérieur, empêchant toute création manuelle d'instance et évitant la duplication de l'état global.
   - **Méthode statique getInstance() :** `getInstance()` fournit la seule instance existante de Partie, avec :
     (a) Création unique
     (b) Allocation contrôlée
     (c) Destruction automatique en fin de programme
   - **Centralisation des ressources :** Partie joue le rôle d'orchestrateur en gérant :
     (a) Les joueurs (humains et IA)
     (b) Les piles de tuiles et le chantier
     (c) L'index du joueur actuel et la rotation des tours
     (d) Les ressources, pierres et validation des règles
     (e) Les variantes et le mode solo
     (f) Le système de scores et le calcul du vainqueur
2. **Fonctionnement de Partie :** → *Valentin*
   - **Gestion des tours :** Nous avons implémenté un index cyclique `indexJoueurActuel` qui parcourt le vecteur des joueurs. La méthode `passerAuJoueurSuivant()` gère la rotation automatique.

— **Architecte en Chef :** Au début de la partie, un joueur est désigné aléatoirement comme premier joueur via `std::shuffle`.

— **Validation des actions :** La méthode `actionPlacerTuile` centralise toute la logique de validation :

(a) Vérification du coût en pierres.

(b) Vérification géométrique du placement (`Cite ::placer`).

(c) Transaction (paiement des pierres, gain des pierres sur la tuile, transfert de propriété).

3. **Classe Joueur :** Refonte complète de la classe pour assurer la cohérence de l'architecture et l'encapsulation des données utilisateur. → *Louane*

— **Composition Forte :** La classe `Joueur` est désormais seule responsable de l'instanciation (dans le constructeur) et de la destruction (dans le destructeur) de ses objets `Cite` et `Score`, garantissant une gestion saine de la mémoire.

— **Architecture des données :** Nous avons structuré les interactions via des accesseurs et méthodes dédiés :

(a) Accesseurs publics (`getScore()`, `getCite()`) pour permettre au moteur `Partie` d'accéder à l'état du joueur.

(b) Gestion sécurisée des ressources via `ajouterPierres` et `utiliserPierres` pour empêcher les valeurs incohérentes.

(c) Initialisation automatique de la cité avec la tuile de départ dès la création du joueur.

» **Responsable :** Valentin & Louane & Noémie

» **Progression :** 100%

» **Temps passé :** *Total :* 17h / *Valentin :* 12h soit 3h de moins que prévu car le mode multijoueur fait partie de la classe Partie. Il est intégré dans les méthodes de cette dernière avec des condition sur le nombre de joueurs. / *Louane :* 3h / *Noémie :* 2h

T7 - Intégration de l'IA :

Nous avons dû réflechir à l'implémentation d'une IA qui ne complexifie pas inutilement la classe Partie et qui puisse être modifiée facilement par la suite.

» **Sous-tâches :** Pour se faire, plutôt que de créer des classes héritées de Partie lourdes (JeuSolo, JeuMulti), nous avons utilisé le concept d'héritage pour créer une classe `IA` qui dérive de la classe `Joueur` (Approche modulaire).

1. **Polymorphisme de données :** Contrairement à un joueur humain qui possède une `Cité`, l'IA stocke ses tuiles dans un vecteur simple `tuilesAcquises` (car elle ne les place pas, elle les « vole » pour le score).
2. **Algorithme de décision :** Nous avons implémenté la méthode `choisirTuile` qui implémente la stratégie de l'IA : elle scanne le chantier pour trouver la tuile la moins chère contenant une « Place » (étoile), sinon elle prend la première, comme décrit dans les règles du jeu.

» **Difficulté rencontrée :** Le défi était d'intégrer l'IA dans la boucle de jeu existante sans avoir à réécrire toute la logique de tour. Le moteur s'attendait à ce qu'un joueur ait une `Cité`.

» **Solution proposée :** Nous avons utilisé le mécanisme de `dynamic_cast` dans le contrôleur (`JeuConsole`) pour détecter si le joueur courant est une IA. Si c'est le cas, le jeu bifurque vers une méthode `jouerTourIA` spécifique, sinon il demande l'input utilisateur. Cette structure est très extensible : la classe `IA` possède un attribut `difficulte`. Pour l'instant, nous avons implémenté la logique de base,

mais nous pourrons facilement ajouter des stratégies plus complexes (Callicratès, Hippodamos) en surchargeant simplement la méthode `choisirTuile` ou en ajoutant des conditions basées sur le niveau de difficulté.

» **Responsable :** Valentin

» **Progression :** 100%

» **Temps passé :** 19h soit 6h de moins que prévu Nous avions sûr estimé le temps de cette tâche car nous pensions qu'il fallait créer une IA capable de réflechir aux coups à jouer. Or, le jeu joue selon des règles bien précises et facilement implémentables.

T8 - Gestion de l'interface console :

Cette tâche est complexe et a nécessité beaucoup d'heures de travail. Elle comprend tout type d'affichage et a été décomposée en diverses soustâches.

» **Sous-tâches :**

1. **Gestion des menus et intéractions utilisateurs :** Cette sous-tâche a été implémenté grâce à la classe `jeuConsole` appelée par `Partie` et qui permet au joueur d'intéragir avec le jeu via les entrées et sorties standard (`cout`/`cin`). → *Valentin*
2. **Moteur d'affichage du Plateau (Quadrillage) :** Conception initiale de la matrice d'affichage en ASCII Art. Le plateau est géré comme une immense chaîne de caractères représentant la grille hexagonale. Un algorithme convertit les coordonnées logiques (x, y) en indices de caractères pour placer visuellement les tuiles au bon endroit. → *Oscar & Louane*
3. **Rendu des Hexagones et Niveaux :** Développement de la logique d'affichage dynamique des hexagones (`Hexagone::affiche`). Pour rendre la 3D lisible en console, chaque cellule affiche son type et son étage (ex : « C1 » pour Caserne niv.1, « H3 » pour Habitation niv.3). → *Louane*
4. **Visualisation du Chantier :** Implémentation de l'affichage du *Chantier* (`afficherChantier`), présentant les tuiles disponibles côte à côte avec leur design ASCII complet capable de s'inverser / tourner selon le choix de l'utilisateur. De plus le niveau ne s'affiche pas encore car la tuile n'est pas posée. → *Louane*

» **Difficultés rencontrées :** La gestion de l'alignement des caractères pour placer les hexagones dans le quadrillage et gérer la différence de niveau.

» **Solution proposée :** Utilisation de codes textuels explicites (H1, M2) à l'intérieur des hexagones pour symboliser la hauteur et découpage des chaînes de caractères (« substrings ») pour afficher les tuiles du chantier ligne par ligne. Calcul du nb de caractères pour le placement sur le quadrillage.

» **Responsable :** Oscar & Louane & Valentin

» **Progression :** 100%

» **Temps passé :** *Total :* 40h soit plus du double de ce qui était prévu initialement car nous avons voulu que le jeu soit visuellement plaisant à jouer. La gestion de l'affichage en console n'est pas aisée car cela se joue souvent au caractère près et nécessite beaucoup de tests. / *Oscar :* 7h / *Valentin :* 18h / *Louane :* 29h

T12 - Création de la classe Partie et refonte de l'Architecture :

L'un des défis majeurs a été de permettre au jeu de fonctionner soit en mode Console, soit en mode Graphique (Qt), sans dupliquer le code des règles du jeu.

» **Sous-tâches :**

1. **Le Moteur - `Partie` :** Cette classe est devenue un « Singleton » pur, elle ne contient aucun `cout` ni `cin`. et utilise des méthodes fonctionnelles quelle que soit le mode d'affichage : `initialiser()`, `actionPlacerTuile()`, `getJoueurActuel()`. Elle renvoie des booléens ou des exceptions pour signaler le succès ou l'échec d'une action, laissant l'interface décider comment afficher l'action ou l'erreur.
   Avantage architectural : Cela facilite grandement l'accès aux données du jeu depuis n'importe où dans le code (notamment depuis l'interface graphique future) via `Partie::getInstance()`, sans avoir à passer des pointeurs en paramètres de toutes les fonctions.
2. **Le Contrôleur Console - `JeuConsole` :** C'est une classe dédiée qui gère la boucle de jeu textuelle (`cout`/`cin`). Elle s'occupe exclusivement des interactions utilisateur (`saisieNombre`, `afficherPlateau`) et appelle les méthodes du Moteur `Partie`.
3. **Le Contrôleur Graphique - `MainWindow` (Qt) :** Grâce à cette séparation, l'interface graphique pourra se brancher sur le même Moteur `Partie` sans modifier une seule ligne de code logique. L'interface Qt étant en phase initial d'implémentation, ce module n'est pas encore rattaché à la classe `Partie`.

» **Difficultés recontrées :**

1. Le code initial utilisait la classe `Partie` à la fois pour l'interface console et l'interface graphique (Qt). Cela rendait impossible l'utilisation de `Partie` dans Qt sans bloquer l'interface à cause des entrées et sorties console.
2. En créant les parties, nous nous sommes rendu compte que la tuile de départ n'existait pas.

» **Solutions aportées**

1. Nous avons séparé chaque module en trois fichiers/classes.
2. Nous avons dû modifier la classe `Tuile` en implémentant une classe fille `TuileDepart` qui permet de créer une tuile de 4 hexagones placée en (0, 0, 0) comme indiqué dans les règles.

» **Modularité apportée :** Cette architecture garantit une maintenance facile : si on change une règle de placement dans `Partie`, elle s'applique instantanément aux deux versions du jeu (Console et Qt).

» **Responsable :** Valentin

» **Progression :** 100%

» **Temps passé :** 27h

T13 - Création de la classe Pile :

Cette tâche est fondamentale car elle constitue une des bases du jeu. Elle a pour but de mettre en place la structure de données qui génère et stocke les tuiles tout au long de la partie.

» **Sous-tâches :**

1. **Structure de données et Stockage :** Mise en place d'un `std::vector<Tuile*>` pour stocker l'ensemble des tuiles générées au début de la partie. Ce vecteur sert de réserve unique pour le jeu.
2. **Gestion du Cycle de Vie (Mémoire) :** Implémentation de la logique d'allocation dynamique (`new`) dans le constructeur et de libération (`delete`) dans le destructeur. La classe `Pile` agit comme le « propriétaire » exclusif des objets, garantissant qu'aucune tuile n'est perdue en mémoire.

3. **Encapsulation et Itérateurs :** Développement d'un itérateur dédié (`Pile::Iterator`) pour permettre à la classe `Chantier` de récupérer les tuiles séquentiellement sans jamais accéder directement au conteneur interne, renforçant la sécurité du code.

» **Responsable :** Louane

» **Progression :** 100%

» **Temps passé :** 8h

T14 Création de la classe Chantier :

» **Objectif :** Gérer la zone tampon entre les piles de pioche et les joueurs, où les tuiles sont exposées, achetées et où les pierres sont posées.

» **Sous-tâches :**

1. **Structure de données :** Nous avons utilisé un `std::vector<Tuile*>` pour stocker les tuiles disponibles.
2. **Mécanique de flux :**
   — Nous avons implémenté la méthode `ajouterPile(Pile& p)` qui parcourt une pile et transfère tous ses pointeurs de tuiles vers le chantier.
   — La gestion des pierres sur les tuiles (coût et gain) est directement intégrée aux objets `Tuile` via `setPrix()`. Ici chaque tuile « porte » son propre coût en pierres, ce qui simplifie la gestion lors des déplacements de `Pile` à `Chantier`.

» **Difficulté rencontrée :** Nous n'avons pas rencontré de grande difficulté durant l'implémentation de cette classe. La partie la plus difficile a cependant été la réflexion pré-implémentation pour savoir comment gérer la place du chantier dans le jeu car elle était d'abord destiné à être encapsulé dans `Pile` ou dans `Partie`.

» **Solution apportée :** Cette classe rend le code très modulaire : la logique de gestion du chantier est isolée. La classe `Partie` n'a plus qu'à appeler `chantier.ajouterPile()` sans se soucier des détails d'itération. De plus, l'utilisation de `Tuile::setPrix` permet de garder l'information « Coût/Gain » solidaire de l'objet, ce qui est plus robuste.

» **Responsable :** Valentin

» **Progression :** 100%

» **Temps passé :** 9h

2.4 Tâches en cours et prochaines étapes

T15 - Amélioration et optimisation du back-end :

Cette tâche a été spécifiquement créée car nous avons constaté que notre architecture initiale ne respectait pas suffisamment le principe d'encapsulation. De nombreuses classes exposaient directement leurs structures de données internes (vecteurs, maps) via des accesseurs bruts, ce qui rendait le code fragile et difficile à faire évoluer.

» **Sous-tâches :**

1. **Création d'itérateurs pour la classe Partie :** Développement de `JoueurIterator` et `PileIterator`. Ces outils permettent de parcourir la liste des joueurs et des piles de manière séquentielle, sans jamais donner un accès direct aux vecteurs privés qui les stockent.

2. **Itérateur du Chantier :** Mise en place de `Chantier::Iterator`. L'interface utilisateur et l'IA peuvent désormais examiner les tuiles disponibles une par une, sans avoir besoin de connaître leur mode de stockage en mémoire.
3. **Itérateur constant pour la Cité :** Développement de `Cite::ConstIterator`. C'est un point crucial pour le module de Score, qui doit pouvoir parcourir le plateau de jeu pour compter les points sans risque de l'altérer.
4. **Architecture Modulaire (Strategy) :** Finalisation de la restructuration du moteur de score, isolant chaque règle de quartier dans une classe dédiée pour garantir l'ajout facile de variantes.

» **Difficultés rencontrées :** Le code initial privilégiait l'efficacité immédiate avec des accès directs aux tableaux. La difficulté a été de supprimer ces dépendances fortes pour intercaler la couche d'abstraction des itérateurs, sans introduire d'erreurs dans la logique de jeu existante (notamment dans les algorithmes de l'IA).

» **Solution proposée & Justification :**

— **Choix du Design Pattern Iterator :** Ce modèle assure une protection totale des données. Le code qui utilise les données (l'Interface, l'IA) est désormais déconnecté de la façon dont elles sont stockées. Si nous remplacions demain la `map` de la Cité par un tableau à trois dimensions, le reste du programme fonctionnerait sans aucune modification.

— **Justification du Const Iterator :** Pour la Cité, nous avons spécifiquement imposé un itérateur *constant* (`const_iterator`). Cela répond à un impératif de sécurité : le calculateur de score a un droit de « lecture seule » sur le plateau. Il est ainsi techniquement impossible qu'une erreur de programmation dans le calcul du score vienne modifier l'état de la ville (supprimer une tuile ou changer son type involontairement), car le compilateur l'interdirait.

— **Design Pattern Strategy :** Il permet d'étendre les règles du jeu (nouvelles variantes) sans toucher au code source du moteur de score existant, ce qui le rend beaucoup plus stable.

» **Défi à venir :** La généralisation des erreurs reste à implémenter. Chaque methode pouvant amener à une erreur renverra un message adapté avec la classe exception correspondante.
Certaines autres améliorations pourront être faites par la suite après relecture du code.

» **Responsable :** Louane

» **Progression :** 60%

» **Temps passé :** 15h

» **Temps restant estimé :** 9h

T9 Gestion de l'Interface Graphique

L'objectif de cette étape était de passer d'une logique purement textuelle à une interface visuelle interactive pour rendre le jeu jouable et intuitif.

» **Sous-tâches :**

1. **Mise en place de l'environnement graphique :** Utilisation de `QGraphicsView` et `QGraphicsScene` pour gérer le rendu 2D performant.
2. **Création des Hexagones Graphiques (HexagonItem) :**
   — Conception d'une classe héritant de `QGraphicsPolygonItem`.
   — Dessin géométrique des hexagones (calcul des sommets par trigonométrie) pour remplacer les images statiques.
   — Gestion des couleurs dynamiques selon le type de quartier (Bleu=Habitation, Rouge=Caserne, etc.).

3. **Gestion des Tuiles (TuileItem) :**
   - Utilisation de `QGraphicsItemGroup` pour regrouper 3 hexagones en une seule entité manipulable.
   - Implémentation de la rotation graphique des tuiles.
4. **Système de d'aimantation des tuiles :** Développement d'un algorithme de conversion de coordonnées Pixel <-> Grille Hexagonale permettant au joueur de cliquer approximativement sur une zone et de voir sa tuile se placer parfaitement dans la case correspondante.

» **Difficultés rencontrées :**
- La gestion des coordonnées hexagonales est complexe par rapport à une grille carrée classique. L'alignement des tuiles a demandé des calculs précis.
- Difficultés d'installation et de prise en main de QtCreator qui est un logiciel capricieux.
- La création d'une interface graphique prend beaucoup plus de temps que prévu.

» **Défi à venir :** Le prochain défi sera de connecter les signaux de la souris (clics sur la vue Qt) aux méthodes existantes de notre moteur `Partie` (comme `actionPlacerTuile`), afin que le moteur console pilote réellement l'interface graphique.

» **Responsable :** Tous

» **Progression :** 30%

» **Temps passé :** 12h (Valentin)

» **Temps restant estimé :** 30h

T16 - Gérer l'enregistrement et le chargement :

» **Sous tâches :**

1. **Formatage et Sauvegarde :** Écriture séquentielle des données dans un fichier texte (paramètres globaux, état du chantier, attributs des joueurs). Gestion spécifique des chaînes de caractères : remplacement des espaces par des tirets bas (_) pour éviter les erreurs de lecture sur les noms composés.
2. Chargement par reconstruction (Historique) : Implémentation d'une structure `Action` qui stocke les coups joués. Au lieu de copier l'état final, la fonction `charger()` lit cet historique et « rejoue » la partie coup par coup. Cela assure que tous les calculs (hauteur, voisins) sont ré-exécutés proprement.
3. **Correction de l'allocation mémoire (`Tuile`) :** Modification du constructeur par défaut de la classe `Tuile`. Il force désormais l'initialisation immédiate du vecteur de 3 hexagones (`resize(3)`), ce qui empêche le jeu de planter lorsqu'on crée des tuiles vides avant de les remplir avec les données du fichier. Restauration contextuelle (Polymorphisme) : Ajout d'une logique de détection lors du chargement pour distinguer les Humains des IA. Si le mode est solo et qu'on charge le deuxième joueur, le système instancie automatiquement un objet `IA` au lieu d'un `Joueur` standard.

» **Difficultés rencontrées :** La principale difficulté a été technique : nous avons fait face à des plantages systématiques (crashs) lors du chargement. Le programme tentait d'écrire des informations sur des tuiles qui n'avaient pas encore alloué leur mémoire interne pour les hexagones. De plus, nous avons eu un problème logique où l'IA, une fois chargée, devenait un joueur humain inerte car le fichier de sauvegarde ne précisait pas explicitement le type de l'objet.

- **Solution proposée :** Nous avons sécurisé la classe `Tuile` en modifiant son constructeur par défaut pour qu'il prépare toujours la mémoire nécessaire. Pour l'accès aux données protégées (comme le score privé) sans briser l'encapsulation, nous avons utilisé le mécanisme de `friend class`, donnant à la classe `Partie` les droits nécessaires pour sauvegarder. Enfin, l'approche de reconstruction coup par coup garantit que la sauvegarde est toujours mathématiquement valide par rapport aux règles du jeu.
- **Responsable :** Jeanne
- **Progression :** 70% (Cohérence avec le reste du code)
- **Temps passé :** Total : 20h

3. Analyse conceptuelle

3.1 UML

Voir annexe *(Image vectorielle : zoom possible)*

4. Bilan de cohésion et d'implication

A l'approche des échéances de rendu, le stress rend la cohésion difficile, nous faisons très attention à communiquer calmement et à se mettre d'accord en amont sur les décisions importantes afin d'éviter des incompréhensions. Nous devons également veiller à se mettre au courant les un.e.s les autres du travail effectué et des parties en cours de correction pour ne pas risquer d'empiéter sur le travail des autres. Nous réunir en physique régulièrement (en plus des TD) nous permet aussi de faire le point sur nos avancées pour être efficaces dans le travail à venir.

Nous arrivons tout de même à travailler efficacement, notamment grâce à la volonté de tous.tes de se concentrer sur un livrable final optimisé quitte à accepter d'avoir passé du temps à travailler « pour rien » quand les solutions envisagées ne fonctionnent pas.

5. Conclusion

Ce rendu nous permet de nous rendre compte précisément du travail restant, en effet, il nous reste du travail pour pouvoir proposer une version stable et propre d'Akropolis et prendre du recul sur l'avancée nous permet de mieux appréhender les tâches restantes. Cette échéance nous motive également beaucoup à améliorer notre projet et nous rend fier.e.s du travail déjà accompli.

Enfin, nous sommes tous.tes content.e.s d'avoir eu l'opportunité de découvrir la création d'un vrai projet complet et concret, encore plus dans le cadre d'un travail de groupe qui apporte encore d'autres défis. Nous pensons que ce challenge nous a beaucoup appris et nous avons hâte de le finir pour voir le résultat.

A. UML

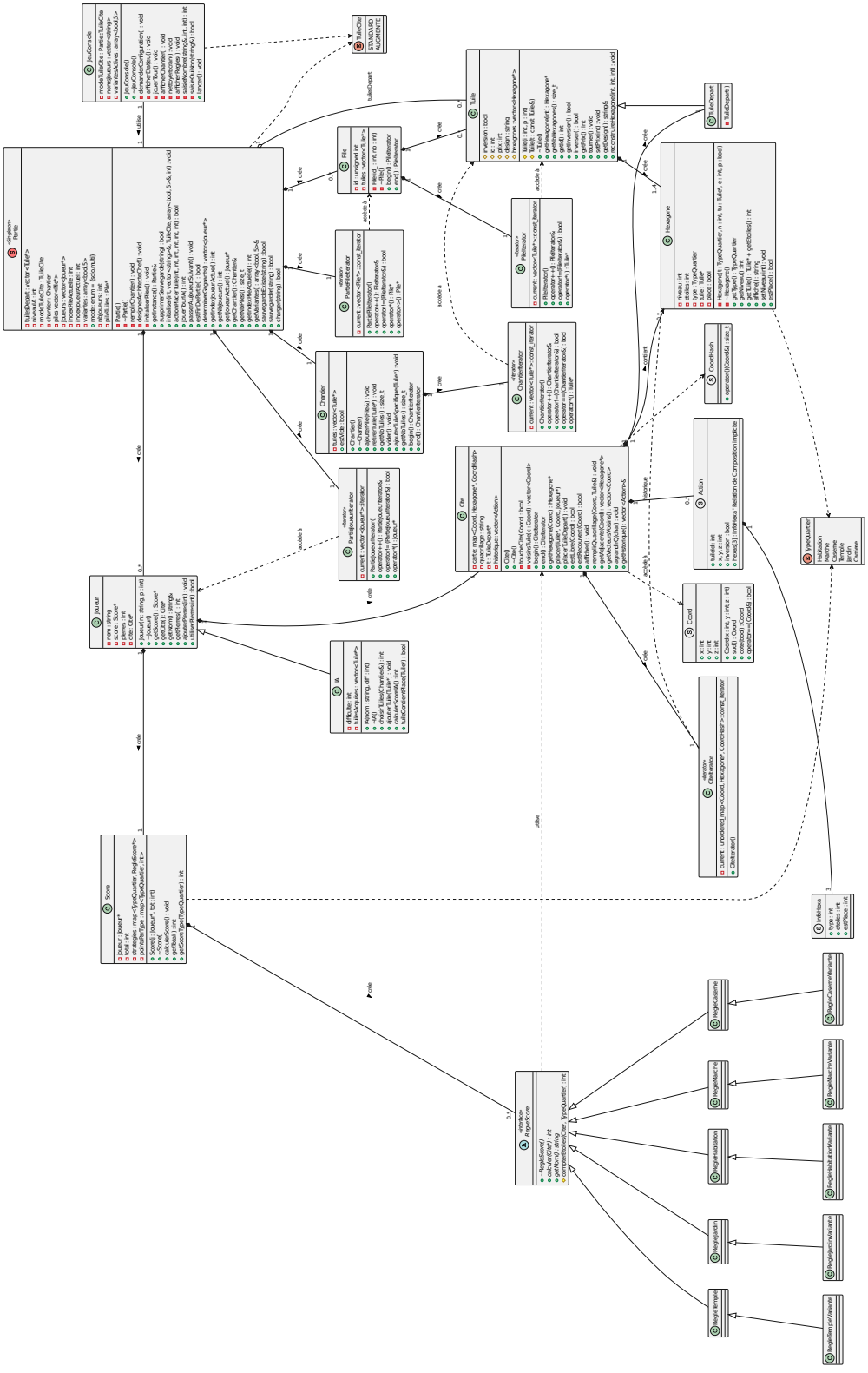

FIGURE A.1 – Modèle conceptuel du jeu

B. Preuve de fonctionnement

Lien