Projet d'étude – Azul
Rapport final

CANAVAGGIO Thibault - CHOUBRAC Thomas – GANDO DIALLO Mouhamed – MOHDAD Ralph – QUATELA Nicolas

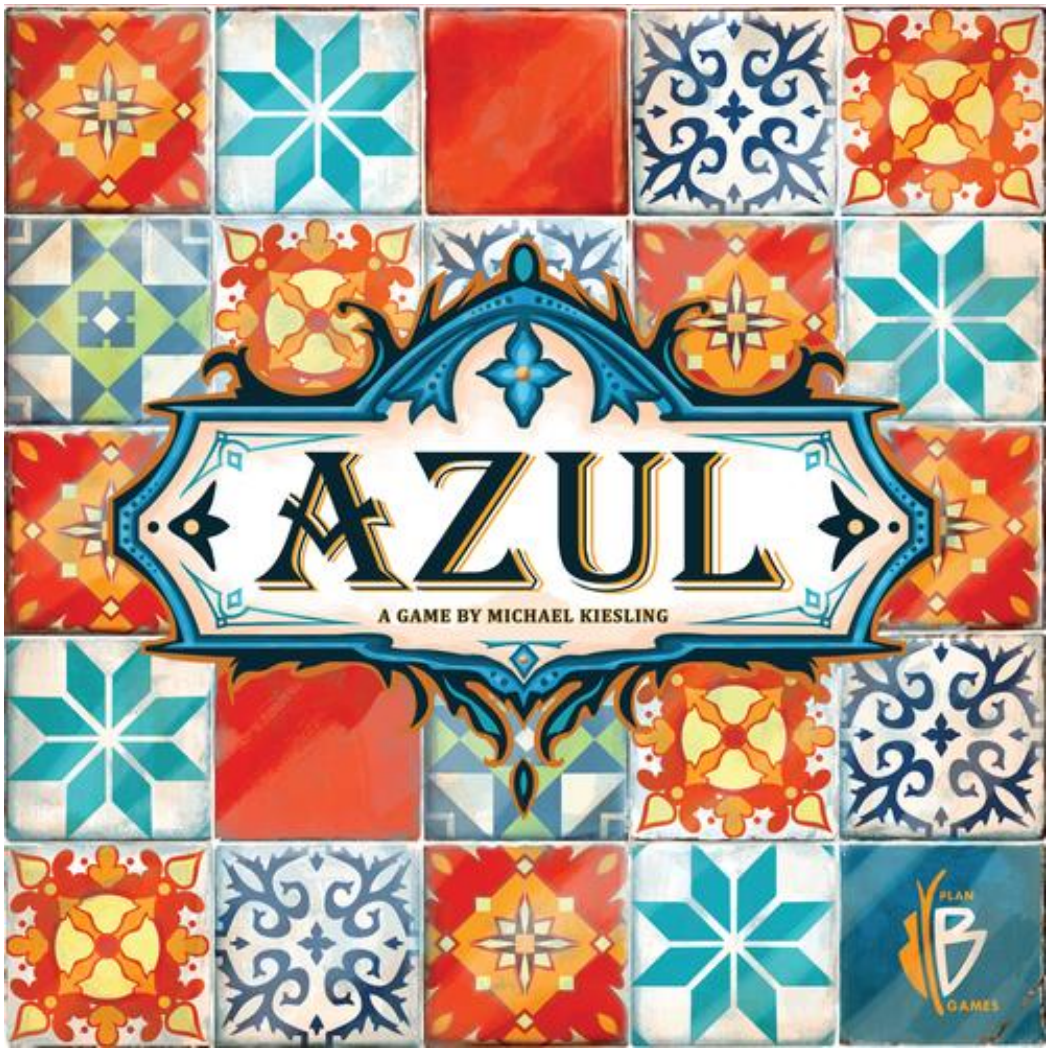


Table des matières

1. Point de vue général de l'architecture et des fonctionnalités

1.1. Glossaire

Tuile : Élément du jeu qui est pioché par les joueurs lors des tours de jeu. Il en existe de plusieurs couleurs (rouge, bleu, noir, jaune, blanc, jeton, et vide permettant d'initialiser les éléments du jeu)

Fabrique : elle contient 4 tuiles venant de la pioche. C'est dans cet endroit où le joueur pioche les tuiles qu'il va jouer.

Lignes de motif : chaque joueur possède ses lignes de motif et les remplis à l'aide des tuiles piochées depuis les fabriques

Plancher : propre à chaque joueur, récupère les tuiles dépassant des lignes de motif et applique un malus de point en fonction du nombre de tuile placé à l'intérieur

Mur : propre à chaque joueur, les lignes de motif pleines remplissent une case du mur correspondante. Une ligne pleine est nécessaire pour gagner la partie.

Centre de la table : lorsque les joueurs piochent une couleur dans une des fabriques, les autres tuiles sont placées au centre de la table. Les joueurs ont le choix de piocher des tuiles depuis les fabriques ou bien depuis le centre de la table

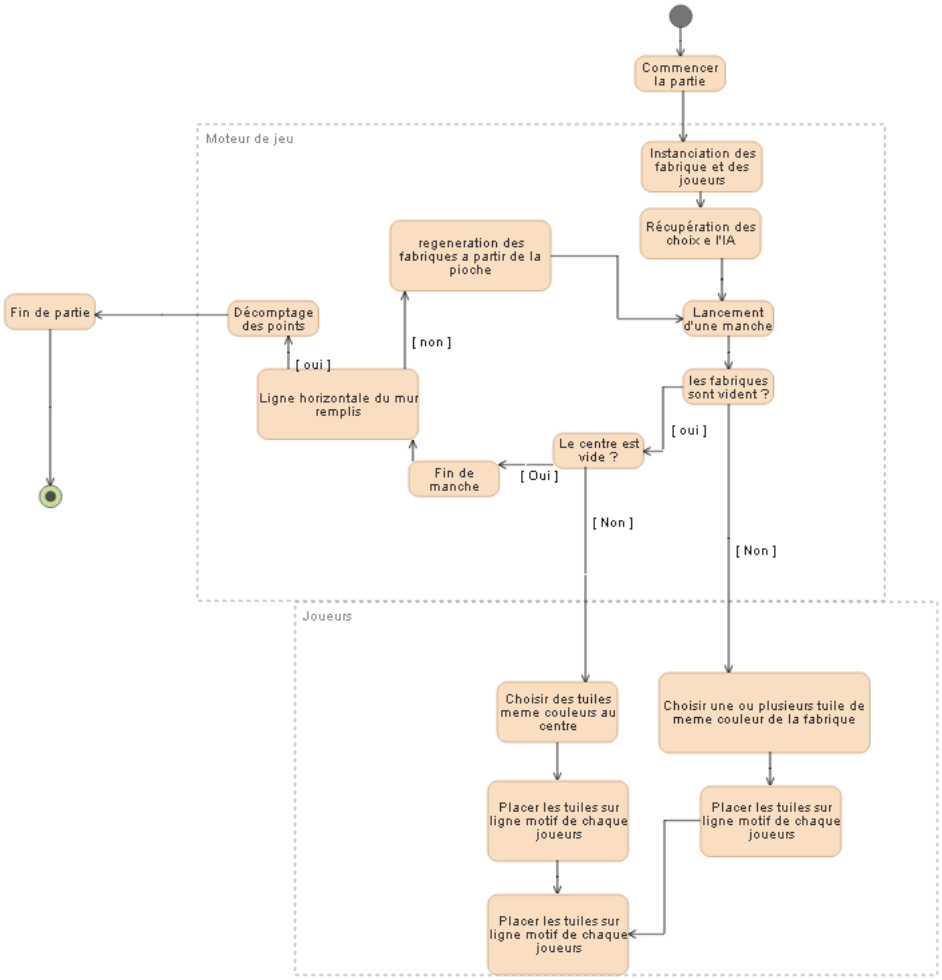
Pioche : contient 100 tuiles (20 de chaque couleur). Les fabriques se remplissent au début de chaque manche en piochant des tuiles venant du sac.
Couvercle : intermédiaire entre la pioche et le joueur, les tuiles dépassant du plancher et celles restant sur les lignes de motif après passage au mur sont placées dans le couvercle. Le couvercle remplit la pioche à la fin d'une manche.

Plateau : représente la surface sur laquelle est placé tous les éléments du jeu qui n'est pas propre à un joueur (les fabriques, le centre de la table, le couvercle et la pioche)

Tour de jeu : un tour de jeu est caractérisé par la pioche et le placement d'une tuile par chaque joueur

Manche : une manche se caractérise par plusieurs tours de jeu, jusqu'à l'épuisement des fabriques et du centre de la table en tuiles. Elles sont remplies ensuite avec des tuiles venant de la pioche

1.2. Représentation générale (diagramme d'activité)



Voici le diagramme d'activité de notre programme un aperçu très simplifié de comment fonctionne le Jeu Azul.

1.3. Point sur les fonctionnalités traitées

- Modélisation du jeu : nous avons représenté la totalité des éléments du jeu (cf. glossaire pour la liste)
- Moteur de jeu mettant en place le jeu, réalise les actions (vérifie les conditions de fin de manche et de partie, place les tuiles sur le mur, lance la partie, gère le jeton, récupère le choix des IA) , compte les points, fais le classement des joueurs
- Trois robots de jeu, dans l'ordre
    - Joueur 1 : IA Random, le joueur récupère des tuiles aléatoirement entre les fabriques ou bien le centre de la table puis pose les tuiles sur une ligne de motif aléatoire
    - Joueur 2 : IA Intermédiaire « Plus de couleur », ce joueur pioche le plus de tuile possible d'une même couleur et les place sur des lignes de motif de taille adapté

    - Joueur 3 : IA Supérieur « Remplir ligne 1 et 2 en premier », ce dernier joueur cherche à remplir sa ligne 1 et ensuite 2 du mur le plus rapidement possible dans le but de terminer la partie le plus rapidement possible
- Possibilité de lancer 500 parties (et un nombre de partie que l'on veut) avec trois robots différents, classement entre les robots et statistiques des parties
- Visualisation du déroulement d'une partie (si une seule partie a été lancée) avec les couleurs dans le terminal
- Variantes du mur gris et variante avec le mur 2.
- Possibilité de lancer une partie, plusieurs à la suite, deux en parallèle, changer la variante et changer le nombre de joueur via l'exécution avec Maven.

Évolutions :

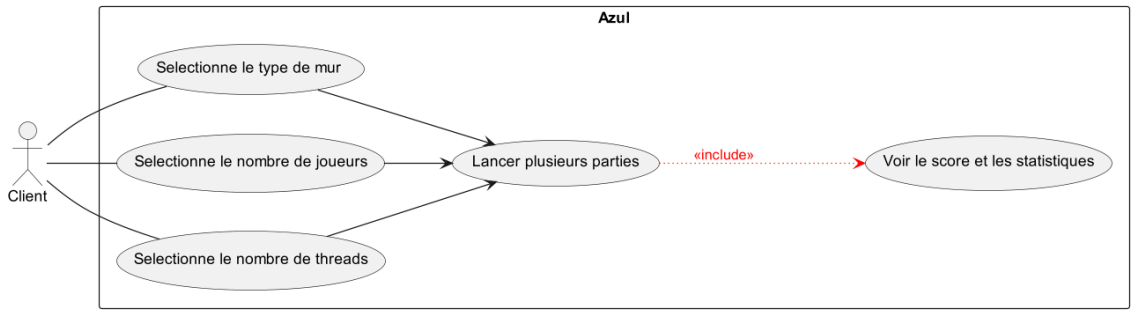
1* : Couleurs sur le terminal, lancer une partie avec la variante plancher 2

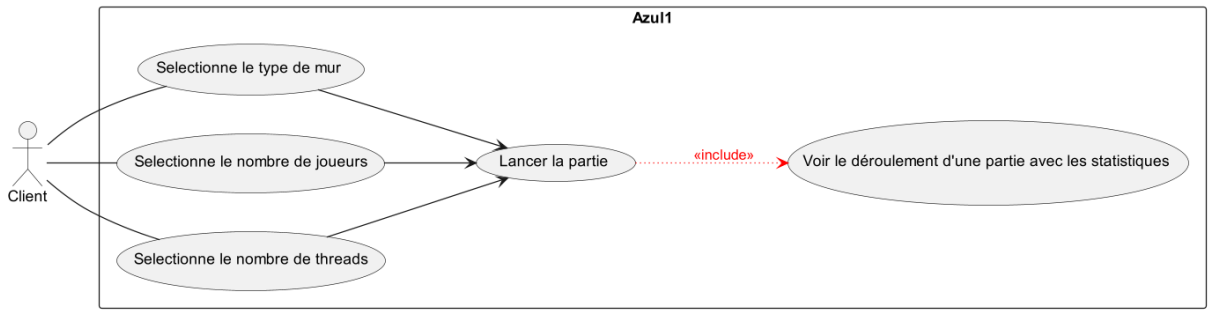
2* : Exécution de parties en parallèles

2. Modélisation de l'application

2.1. Analyse des besoins : cas d'utilisation

Diagramme de cas d'utilisation + use case





Le client doit pouvoir :
- sélectionner le type de mur
- le nombre de joueurs qu'il souhaite
- le nombre de threads
- lancer la partie
- voir le déroulement d'une partie avec les statistiques.

Le client doit pouvoir :
- sélectionner le type de mur
- le nombre de joueurs qu'il souhaite
- le nombre de threads
- lancer plusieurs parties
- voir le score et les statistiques

Cas d'utilisation détaillé

Fiche descriptive du cas d'utilisation (user case) "lancement d'une partie Azul"

Scénario nominal :

L'utilisateur choisit les paramètres de la partie (1 partie, nombre de joueur, couleur du mur, thread) et lance la commande.

Le système prend en compte les paramètres entrés par l'utilisateur et lance la partie.
Le système affiche le déroulement de la partie et le score.
Scénario alternatif :
L'utilisateur choisit les paramètres de la partie (plusieurs parties, nombre de joueurs, couleur du mur, thread) et lance la commande.
Le système prend en compte les paramètres entrés par l'utilisateur et lance les parties.
Le système affiche le score et les statistiques.

Scénario d'erreur :
L'utilisateur rentre comme paramètre plus de thread que de partie.

2.2. Conception logicielle

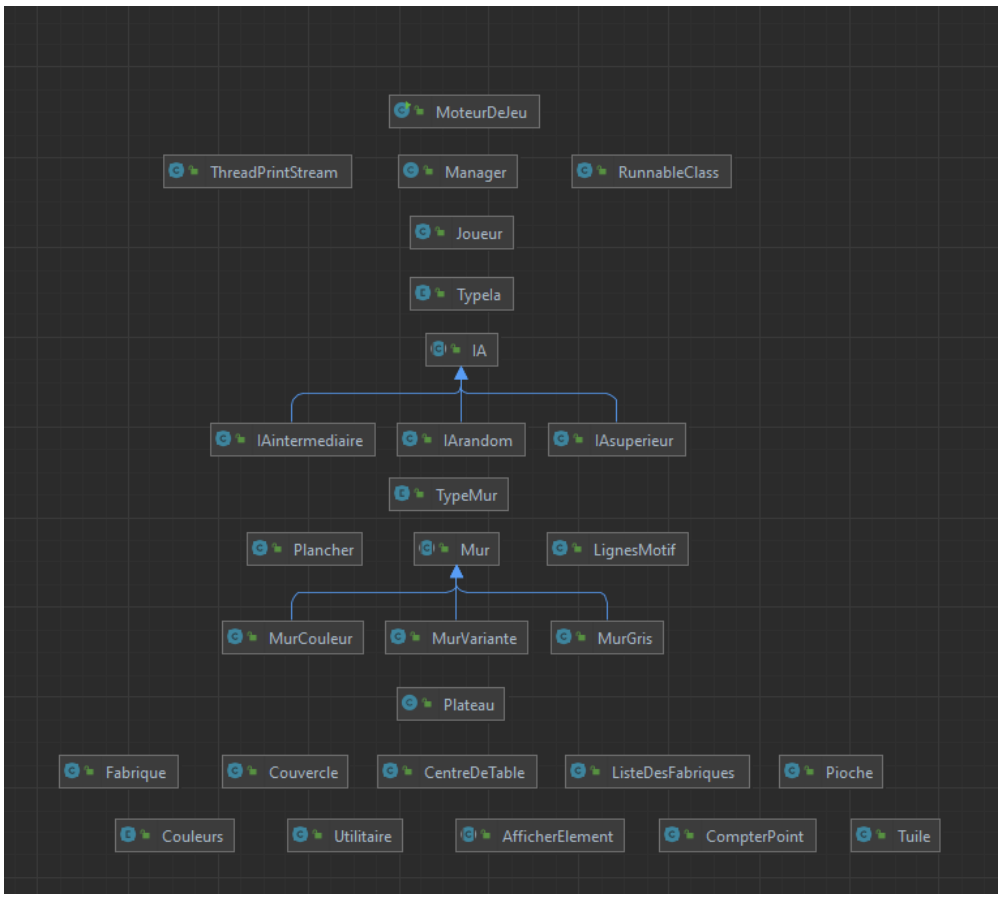
a. Point de vue statique : diagrammes de classe simplifié

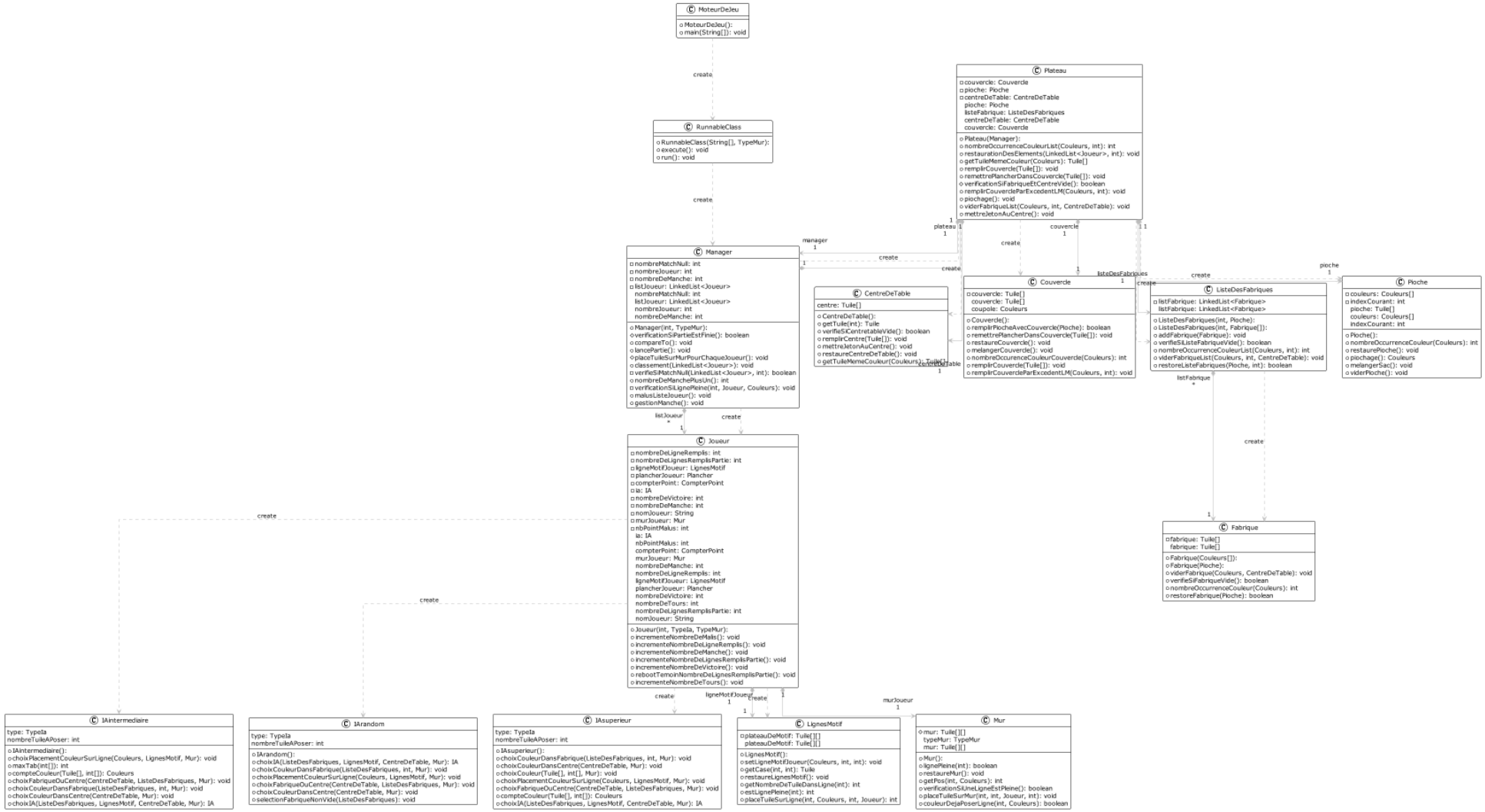
Le composant « Moteur de Jeu » est la classe main qui exécute le programme et où les arguments sont spécifiés.

Le composant « RunnableClass » est la classe qui permet de gérer les threads et le nombre de parties avec un affichage plus épuré.

Le composant « Manager » contient toute la partie logique d'Azul. Représentation d'un diagramme de classes simplifié du moteur de jeu. Le nombre de classes étant important, il est difficile de Représenter toute la structure du code sur un seul diagramme.

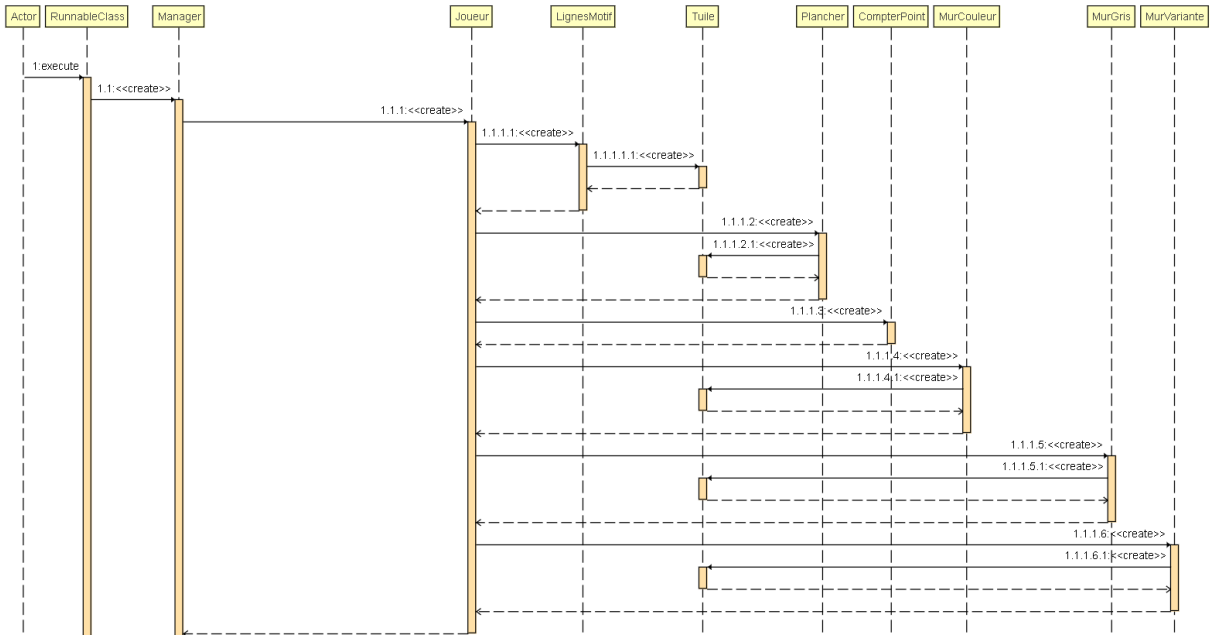
Le composant « ThreadsPrintStream » permet de récupérer les threads de chaque partie dans un fichier texte avec tout l'affichage de chaque partie pour pouvoir visualiser les différentes parties en parallèle.



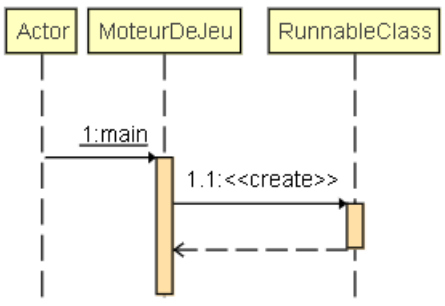

MoteurDeJeu
MoteurDeJeu():
main(String[]): void
create
RunnableClass
RunnableClass(String[], TypeMur):
execute(): void
run(): void
create
Plateau
couvercle: Couvercle
pioche: Pioche
centreDeTable: CentreDeTable
pioche: Pioche
listeFabrique: ListeDesFabriques
centreDeTable: CentreDeTable
couvercle: Couvercle
Plateau(Manager):
nombreOccurrenceCouleurList(Couleurs, int): int
restaurationDesElements(LinkedList<Joueur>, int): void
getTuileMemeCouleur(Couleurs): Tuile[]
remplirCouvercle(Tuile[]): void
remettrePlancherDansCouvercle(Tuile[]): void
verificationSiFabriqueEtCentreVide(): boolean
remplirCouvercleParExcedentLM(Couleurs, int): void
piochage(): void
viderFabriqueList(Couleurs, int, CentreDeTable): void
mettreJetonAuCentre(): void
plateau
couvercle
create
Manager
nombreMatchNull: int
nombreJoueur: int
nombreDeManche: int
listJoueur: LinkedList<Joueur>
nombreMatchNull: int
listJoueur: LinkedList<Joueur>
nombreJoueur: int
nombreDeManche: int
Manager(int, TypeMur):
verificationSiPartieEstFinie(): boolean
compareTo(): void
lancePartie(): void
placeTuileSurMurPourChaqueJoueur(): void
classement(LinkedList<Joueur>): void
verifieSiMatchNull(LinkedList<Joueur>, int): boolean
nombreDeManchePlusUn(): int
verificationSiLignePleine(int, Joueur, Couleurs): void
malusListeJoueur(): void
gestionManche(): void
manager
create
CentreDeTable
centre: Tuile[]
CentreDeTable():
getTuile(int): Tuile
verifieSiCentretableVide(): boolean
remplirCentre(Tuile[]): void
mettreJetonAuCentre(): void
restaureCentreDeTable(): void
getTuileMemeCouleur(Couleurs): Tuile[]
centreDeTable
Couvercle
couvercle: Tuile[]
couvercle: Tuile[]
coupole: Couleurs
Couvercle():
remplirPiocheAvecCouvercle(Pioche): boolean
remettrePlancherDansCouvercle(Tuile[]): void
restaureCouvercle(): void
melangerCouvercle(): void
nombreOccurrenceCouleurCouvercle(Couleurs): int
remplirCouvercle(Tuile[]): void
remplirCouvercleParExcedentLM(Couleurs, int): void
listeDesFabriques
create
ListeDesFabriques
listFabrique: LinkedList<Fabrique>
listFabrique: LinkedList<Fabrique>
ListeDesFabriques(int, Pioche):
ListeDesFabriques(int, Fabrique[]):
addFabrique(Fabrique): void
verifieSiListeFabriqueVide(): boolean
nombreOccurrenceCouleurList(Couleurs, int): int
viderFabriqueList(Couleurs, int, CentreDeTable): void
restoreListeFabriques(Pioche, int): boolean
pioche
create
Pioche
couleurs: Couleurs[]
indexCourant: int
pioche: Tuile[]
couleurs: Couleurs[]
indexCourant: int
Pioche():
nombreOccurrenceCouleur(Couleurs): int
restaurePioche(): void
piochage(): Couleurs
melangerSac(): void
viderPioche(): void
listJoueur
create
Joueur
nombreDeLigneRemplis: int
nombreDeLignesRemplisPartie: int
ligneMotifJoueur: LignesMotif
plancherJoueur: Plancher
compterPoint: CompterPoint
ia: IA
nombreDeVictoire: int
nombreDeManche: int
nomJoueur: String
murJoueur: Mur
nbPointMalus: int
ia: IA
nbPointMalus: int
compterPoint: CompterPoint
murJoueur: Mur
nombreDeManche: int
nombreDeLigneRemplis: int
ligneMotifJoueur: LignesMotif
plancherJoueur: Plancher
nombreDeVictoire: int
nombreDeTours: int
nombreDeLignesRemplisPartie: int
nomJoueur: String
Joueur(int, TypeIa, TypeMur):
incrementeNombreDeMalls(): void
incrementeNombreDeLigneRemplis(): void
incrementeNombreDeManche(): void
incrementeNombreDeLignesRemplisPartie(): void
incrementeNombreDeVictoire(): void
rebootTemoinNombreDeLignesRemplisPartie(): void
incrementeNombreDeTours(): void
create
create
create
ligneMotifJoueur
murJoueur
create
IAintermediaire
type: TypeIa
nombreTuileAPoser: int
IAintermediaire():
choixPlacementCouleurSurLigne(Couleurs, LignesMotif, Mur): void
maxTab(int[]): int
compteCouleur(Tuile[], int[]): Couleurs
choixFabriqueOuCentre(CentreDeTable, ListeDesFabriques, Mur): void
choixCouleurDansFabrique(ListeDesFabriques, int, Mur): void
choixCouleurDansCentre(CentreDeTable, Mur): void
choixIA(ListeDesFabriques, LignesMotif, CentreDeTable, Mur): IA
IArandom
type: TypeIa
nombreTuileAPoser: int
IArandom():
choixIA(ListeDesFabriques, LignesMotif, CentreDeTable, Mur): IA
choixCouleurDansFabrique(ListeDesFabriques, int, Mur): void
choixPlacementCouleurSurLigne(Couleurs, LignesMotif, Mur): void
choixFabriqueOuCentre(CentreDeTable, ListeDesFabriques, Mur): void
choixCouleurDansCentre(CentreDeTable, Mur): void
selectionFabriqueNonVide(ListeDesFabriques): void
IAsuperieur
type: TypeIa
nombreTuileAPoser: int
IAsuperieur():
choixCouleurDansFabrique(ListeDesFabriques, int, Mur): void
choixCouleurDansCentre(CentreDeTable, Mur): void
choixCouleur(Tuile[], int[], Mur): void
choixPlacementCouleurSurLigne(Couleurs, LignesMotif, Mur): void
choixFabriqueOuCentre(CentreDeTable, ListeDesFabriques, Mur): void
compteCouleur(Tuile[], int[]): Couleurs
choixIA(ListeDesFabriques, LignesMotif, CentreDeTable, Mur): IA
LignesMotif
plateauDeMotif: Tuile[][]
plateauDeMotif: Tuile[][]
LignesMotif():
setLigneMotifJoueur(Couleurs, int, int): void
getCase(int, int): Tuile
restaureLignesMotif(): void
getNombreDeTuileDansLigne(int): int
estLignePleine(int): int
placeTuileSurLigne(int, Couleurs, int, Joueur): int
Mur
mur: Tuile[][]
typeMur: TypeMur
mur: Tuile[][]
Mur():
lignePleine(int): boolean
restaureMur(): void
getPos(int, Couleurs): int
verificationSiUneLigneEstPleine(): boolean
placeTuileSurMur(int, int, Joueur, int): void
couleurDejaPoserLigne(int, Couleurs): boolean
Fabrique
fabrique: Tuile[]
fabrique: Tuile[]
Fabrique(Couleurs[]):
Fabrique(Pioche):
viderFabrique(Couleurs, CentreDeTable): void
verifieSiFabriqueVide(): boolean
nombreOccurrenceCouleur(Couleurs): int
restoreFabrique(Pioche): boolean

Dans ce diagramme de classe, nous pouvons voir la partie la plus importante du jeu Azul et la partie logique notamment avec les classes Manager Runnableclass et Moteur de jeu.
Pour la partie gestion de jeu dans le manager, nous pouvons voir que le plateau permet de gérer les différents éléments du jeu propre au plateau et non pas aux joueurs.
De plus, nous pouvons voir que le joueur possède un mur propre et une ligne motif propre a lui-même.

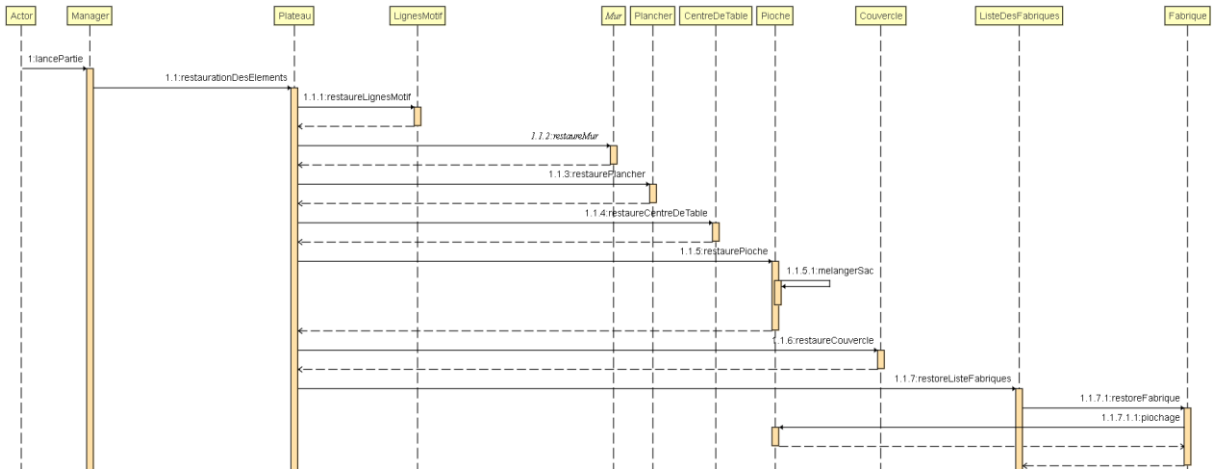
b. Point de vue dynamique : diagrammes de séquences



Voici le diagramme de séquence pour l'exécution des threads avec la classe Runnableclass qui permet au client d'avoir plusieurs ou une seule partie en parallèle.



Voici le diagramme de séquence du moteur de jeu dans le moteur de jeu, on crée la classe Runnable qui permet de lancer avec un ou plusieurs threads, de plus dans la classe Runnable, on fait appel au manager qui est l'arbitre de jeu et la partie logique d'Azul.
Dans le moteur de jeu de plus, on spécifie le type de mur à l'aide des arguments si l'argument saisi dans Maven est « gris » alors le mur gris sera sélectionné si « variante » est sélectionné alors ça sera le mur variant.
Afin de lancer plusieurs parties, on spécifie avec les arguments sur Maven le nombre de parties voulu et les différents types d'argument threads nombre de joueurs, etc.



Voici le diagramme de séquence de la classe manager, on fait appel à la méthode lance partie qui permet de récupérer les choix des IA de plus permet de restaurer les éléments lors de la fin de manche et permet de lancer une partie.
Afin de voir le déroulement de la partie, on utilise une classe Affiche Élément qui permet d'afficher tout le déroulement du jeu. Lors de l'exécution du jeu Azul nous avons fait en sorte que si le nombre de parties est supérieur à 1, on n'affiche pas tout le déroulement du jeu, mais seulement le score et les statistiques qui vont avec et tout ça se fait à l'aide de la classe Runnable et moteur de jeu et le pom.xml.

3. Conclusion

a. Analyse de la solution

Notre solution a le point fort d'avoir une représentation claire du jeu de plateau AZUL, de simuler trois comportements avec 3 robots de niveau différents jouant entre eux, de pouvoir les faire jouer entre eux et obtenir des statistiques, et ce, sur les différentes variantes à savoir : mur de couleur, mur gris, et mur variant numéro 2.

À partir de l'itération 4, nous avons fait un gros refactoring de notre code ce qui nous a permis de mieux nous attribuer les taches et la répartition de travail.

De plus dès lors de l'itération 5, nous avons commencé à nous occuper des fonctionnalités que le client souhaite en rajoutant notamment le couvercle et les 500 parties avec notamment deux niveaux d'IA assez intelligente.

À partir de l'itération 6, nous nous sommes concentrés sur le déroulement des parties en simultané avec les threads et nous nous sommes occupés de respecter les règles du jeu à savoir la création du mur gris et les statistiques demander au client sur 500 parties et sur une seule partie.

Pour l'itération 7 nous nous sommes demandé si nous avons le temps de commencer la partie serveur client et nous avons remarqué que ça nous demandera un temps conséquent de remodifier toute la structure du code afin de l'adapter à la structure client-serveur par suite nous avons décidé de réaliser la fonctionnalité mur variantes et un troisième niveau de IA qui sera la plus intelligentes.

Lors de l'itération 8, nous avons remarqué que nous n'avons pas assez de test sur tout le jeu par suite nous nous sommes attardés dessus et avons établis les dernières modifications qu'il faut et nettoyer le code.

La version finale du jeu est complète, et répond aux attentes du client.

b. Suite du développement

Pour la suite du développement du projet, il aurait fallu sortir le concept de tour du manager dans une méthode ou bien dans une classe. De même pour la manche, qui pourrait mieux être sortie qu'actuellement dans le but de bien différencier tous les éléments. On pourrait aussi rendre le déroulement du jeu encore plus clair.

En termes de fonctionnalité qui reste à implémenter, il manque la variante du mur 1 ainsi que la mise en réseau.