

Projet : PCII

RYAN GERMAIN
TOM NOEL
ALEXANDRE NOVAIS

Table des matières

1. Introduction.....	2
2. Analyse globale.....	3
3. Plan de développement	5
I. Fonctionnalités.....	5
II. Diagramme de GANTT.....	8
Aperçu Vue d'ensemble :	8
Répartition quotidienne :	8
4.1 Conception générale	9
4.2 Conception détaillée.....	11
1 - Génération de la Carte du Jeu	12
2 - Création des Tuile.....	16
3 - Création d'une minimap	19
4 - Création de minerais.....	22
5 - Interaction avec la carte.....	25
6 - Création d'un inventaire	29
7 - Création d'un Timer Globale.....	32
8 – Implémentation des Mineurs	35
9 – Création des modèles d'actions.....	39
10 – Dynamisation Visuelle via les Sprites	45
11 – Implémentation des unités hostiles	47
12 – Implémentation du magasin	49
13. Implémentation de fermeture entre les zones	53
14. Implémentation d'interfaces UI/UX	57
15. Création du gestionnaire de partie.....	60
16. Implémentation de l'action de capture des voleurs	62
5. Résultats.....	66
6. Documentation utilisateur	69
Procédure commune :	69
Méthode 1 : depuis l'IDE :	69
7. Documentation développeur.....	70
Point d'Entrée	70
Paramètres du Jeu.....	70
Guide d'Ajout de Fonctionnalités	70
Fonctionnalités non implémentées :	71
8. Conclusion et perspectives.....	73

1. Introduction

Projet : Évasion de la Grotte

Objectif du jeu :

Le joueur incarne un groupe de mineurs piégés dans une grotte qui doivent trouver un moyen de s'échapper. Le jeu combine des éléments de survie, de gestion des ressources et d'exploration.

Exploration de la grotte :

Le joueur explore une grotte générée de manière procédurale pour trouver des ressources et des voies de sortie. La carte est un ensemble de tuiles formant un labyrinthe généré à l'aide de l'algorithme de Kruskal afin d'assurer que toutes ces tuiles soient atteignables. Les tuiles seront affichées une à une, elle représente l'aire de jeu.

Afin de pouvoir se déplacer dans le labyrinthe, le joueur sera assisté d'une minimap lui permettant de voir les tuiles explorées, la position de ses mineurs et l'activité des créatures hostiles.

L'objectif sera d'atteindre la tuile rouge qui comporte la sortie représentée par une porte bloquée par de l'or.

Gestion des ressources :

Les mineurs doivent progresser dans la grotte en récoltant différents types de ressources leur permettant de fabriquer de meilleur équipement. Ces équipements leurs permettront de récolter des ressources de tier supérieur et ainsi débloquent de nouvelle porte bloquée afin de progresser dans le labyrinthe jusque-là sorti.

Les ressources sont soit des minerais soit des composants que les créatures hostiles laisseront derrière elles.

Les ressources demanderont un temps défini de récolte afin de donner une quantité semi-aléatoire de ladite ressource. Elles auront un temps de réapparition prédéfinie. Elles sont disposées de manière semi aléatoire dans le labyrinthe, la quantité de chaque minerai dépend du niveau de la zone de la tuile qui sera défini lors de la création du labyrinthe.

Magasin :

Un magasin proposant au joueur d'acheter de l'équipement par mineur ou de recruter des mineurs sera mis en place.

Le joueur peut ainsi recruter davantage de mineurs afin d'accélérer sa progression. Chaque mineur dispose de son propre inventaire et équipement qu'il faudra améliorer pour avancer.

Créatures hostiles :

Des créatures hostiles peuvent apparaître dans la grotte afin de voler les ressources des mineurs et gêner le joueur dans sa progression. Il sera possible de les faire fuir ou de les capturer pour récolter leurs ressources. Toutes ressources récoltées par ces derniers seront des ressources non exploitables et perdues pour les mineurs.

2. Analyse globale

Le tableau suivant énumère les fonctionnalités attendues ainsi que l'estimation de leurs difficultés et l'évaluation de leurs priorités.

	Fonctionnalités	Description	Difficulté	Priorité
1	Génération de la carte du jeu	Création d'une carte de jeu découpée en tuile (matrice). L'association des tuiles et de murs permet de concevoir cette carte comme un labyrinthe. Cette carte comportera des zones distinctes.	Moyen	1
2	Création de tuiles	Tuile composant les différentes parties de la carte de jeu. Ces tuiles représenteront des salles dans lesquelles le joueur évolue.	Moyen	1
3	Création d'une minimap	Réaliser une minimap pour permettre au joueur de se repérer dans la carte et de voir les asset notables (mineur, enemies etc...)	Facile	2
4	Création des minerais	Créer des collectables sous la forme de minerais récoltable par le joueur.	Moyen	1
5	Interaction avec la carte de jeu	Créer des Événements de clic permettant d'interagir avec les minerais et les porte pour changer de tuile.	Moyen	1
6	Création d'un inventaire	Réaliser un inventaire de ressources unique à chaque partie qui inclurait les différentes ressources.	Facile	2
7	Création d'un Timer Globale	Implémenter un système afin de connaître le temps passer en partie et l'afficher au joueur.	Facile	3
8	Implémentation des mineurs	Créer les personnages contrôlés par le joueur permettant de miner, se déplacer et interagir avec les créatures hostiles.	Facile	1
9	Création des modèles d'actions des mineurs	Implémenter un système d'action contrôlé par un thread séparé pour gérer les différentes actions des mineurs.	Difficile	2
10	Dynamisation visuelle via les Sprites	Ajouter un système de gestion de Frame pour animer les mineurs.	Facile	3
11	Implémentation des voleurs	Création d'unités avec une logique indépendante volant les minerais et gênant la progression du joueur.	Difficile	1
12	Implémentation du magasin	Créer une interface utilisable par l'utilisateur permettant différentes actions.	Moyen	1

13	Implémentation de fermeture entre les zones	Bloquer les portes entre les zones de niveaux différents pour imposer au joueur de progresser.	Difficile	1
14	Implémentation d'interfaces UI/UX	Créer des interfaces afin de visualiser les détails des éléments interactifs (mineurs, minerais, portes etc...).	Facile	3
15	Création du gestionnaire de partie	Créer une logique centralisant la gestion des instances afin d'avoir la main sur le cycle de jeu.	Facile	3
16	Implémentation de l'action de capture des voleurs	Créer une action permettant aux mineurs de capturer les voleurs pour gagner de l'argent.	Moyen	2

3. Plan de développement

I. Fonctionnalités

Le projet comprend un total de 14 tâches distinctes. Chacune des fonctionnalités a été soigneusement décomposée pour répondre simultanément aux exigences du projet et à la répartition des ressources disponibles.

1. Génération de la carte du jeu

- Préparation de l'environnement, étude de Kruskal
- Implémentation du labyrinthe
- Conception des différentes zones
- Implémentation des observateurs
- Test et validation

2. Création des tuiles

- Conceptualisation et recherche d'asset pour une tuile
- Implémentation de l'algorithme de créations des tuiles
- Création d'un système de mise en cache des images
- Implémentation des observateurs
- Ajout à l'interface graphique
- Test et validation

3. Création d'une minimap

- Enrichir les modèles pour créer une minimap
- Ajout de la minimap à l'interface graphique
- Implémentation des observateurs
- Test et validation

4. Création des minerais

- Recherche d'assets
- Conception de la logique/cycle de vie des minerais
- Implémentation des minerais et leurs relations avec les modèles
- Ajout des minerais à l'interface graphique
- Implémentation des observateurs
- Test et validation

5. Interaction avec la carte du jeu

- Conceptualisation de la logique d'interaction avec la carte
- Implémentation de la logique
- Conception de hitbox avec les éléments interactifs
- Ajout des feedback interaction à l'interface graphique
- Test et validation

6. Création d'un inventaire

- Création d'un objet contenant les ressources du joueur
- Interaction avec l'inventaire
- Ajout de l'inventaire à l'interface graphique
- Implémentation des observateurs

7. Création d'un Timer Globale

- Conception du thread de temps
- Ajout du timer à l'interface graphique

8. Implémentation des mineurs

- Recherche d'assets
- Conception d'un modèle de mineur
- Conception de ses actions communes à toutes interaction
- Ajout des mineurs à l'interface graphique
- Implémentation d'observateurs

9. Création des modèles d'action des mineurs

- Conception d'un thread gestionnaire d'action
- Conception des différents modèles d'actions
- Conception des différentes logiques d'action
- Ajout des feedbacks dans l'interface graphique
- Implémentation d'observateurs
- Test et validation

10. Dynamisation visuelle via les Sprites

- Implémentation d'un gestionnaire de frame
- Recherche d'assets
- Implémentation de la logique de changement d'image dans l'interface graphique
- Test et validation

11. Implémentation des unités hostiles

- Recherche d'assets
- Conception des logiques d'action et du modèle de voleur
- Thread de déplacement
- Test déplacement vers un minerai préalablement calculé
- Ajout des unités hostiles à l'interface graphique
- Implémentation des observateurs

12. Implémentation du magasin

- Thread de décrémentation de l'argent du joueur
- Implémentation de différentes interfaces dynamiques au joueur
- Implémentation des différentes action déclenchable par l'utilisateur :

Achat d'un mineur - Achat d'un objet concernant un mineur

- Implémentation de la logique du Shop notamment entre les changements entre les différents frames
- Ajout de la consommation des différentes ressources pour les actions déclenchable par l'utilisateur

13. Implémentation de fermeture entre les zones

- Création des assets
- Implémentation et intégration des modèles de portes bloquées
- Modification du flux d'états interne des mineurs
- Création du modèle d'action
- Implémentation des observers
- Implémentation des retours utilisateur

14. Implémentation d'interfaces UI/UX

- Recherche et création de fenêtres sur le thème d'une grotte
- Implémentation des feedback

15. Création du gestionnaire de partie

- Abstraction des fonctionnalité du cycle de jeu
- Implémentation du flux de cycle dans le gestionnaire

16. Implémentation de l'action de capture des voleurs

- Modification du flux d'états interne des mineurs
- Modification du flux d'états interne des voleurs
- Création du modèle d'action
- Implémentation des observers
- Implémentation des retour utilisateur

II. Diagramme de GANTT

Pour améliorer la clarté et la lisibilité des activités planifiées, le diagramme de Gantt sera scindé en deux sections distinctes. La première section présentera un aperçu des principales tâches à réaliser pour chaque jour ouvrable, mettant en évidence les éléments essentiels du projet.

La seconde section, quant à elle, fournira une répartition détaillée des heures attribuées à ces tâches, subdivisées en sous-tâches spécifiques, pour chaque journée concernée.

La répartition des ressources est illustrée par le code couleur suivant :

- **Violet** : Tâche partagée
- **Vert** : Ryan
- **Rouge** : Tom
- **Bleu** : Alexandre

Aperçu Vue d'ensemble :

	Statut	Responsable(s)	Lun 04/03/2024	Mer 06/03/2024	Dim 10/03/2024	Lun 11/03/2024	Mer 13/03/2024	Dim 17/03/2024	Lun 18/03/2024	Mer 20/03/2024	Dim 24/03/2024	Lun 25/03/2024	Lun 01/04/2024
1. Génération de la carte du jeu	Terminée	Alexandre											
2. Création des tuiles	Terminée	Alexandre											
3. Création d'une minimap	Terminée	Alexandre											
4. Création des minéraux	Terminée	Alexandre											
5. Interaction avec la carte du jeu	Terminée	Alexandre											
6. Création d'un inventaire	Terminée	Alexandre											
7. Création d'un Timer Globale	Terminée	Alexandre											
8. Implémentation des mineurs	Terminée	Alexandre - Tom											
9. Création des modèles d'action des mineurs	Terminée	Alexandre											
10. Dynamisation visuelle via les Sprites	Terminée	Alexandre											
11. Implémentation des unités hostiles	Terminée	Tom											
12. Implémentation du magasin	Terminée	Ryan											
13. Implémentation de fermeture entre les zones	Terminée	Alexandre											
14. Implémentation d'interfaces UI/UX	Terminée	Alexandre											
15. Création du gestionnaire de partie	Terminée	Alexandre											
16. Implémentation de l'action de capture des voleurs	Terminée	Alexandre											
15. Rédaction cahier des charges	Terminée	Alexandre - Ryan - Tom											

Note : Ce diagramme figure également dans l'annexe et y est plus lisible.

Répartition quotidienne :

Afin d'alléger le cahier des charges, le rapport quotidien est en PDF dans le dossier source.

4.1 Conception générale

L'application a été conçue selon le design pattern MVC. La logique de ce MVC a été découpé de la manière suivante :

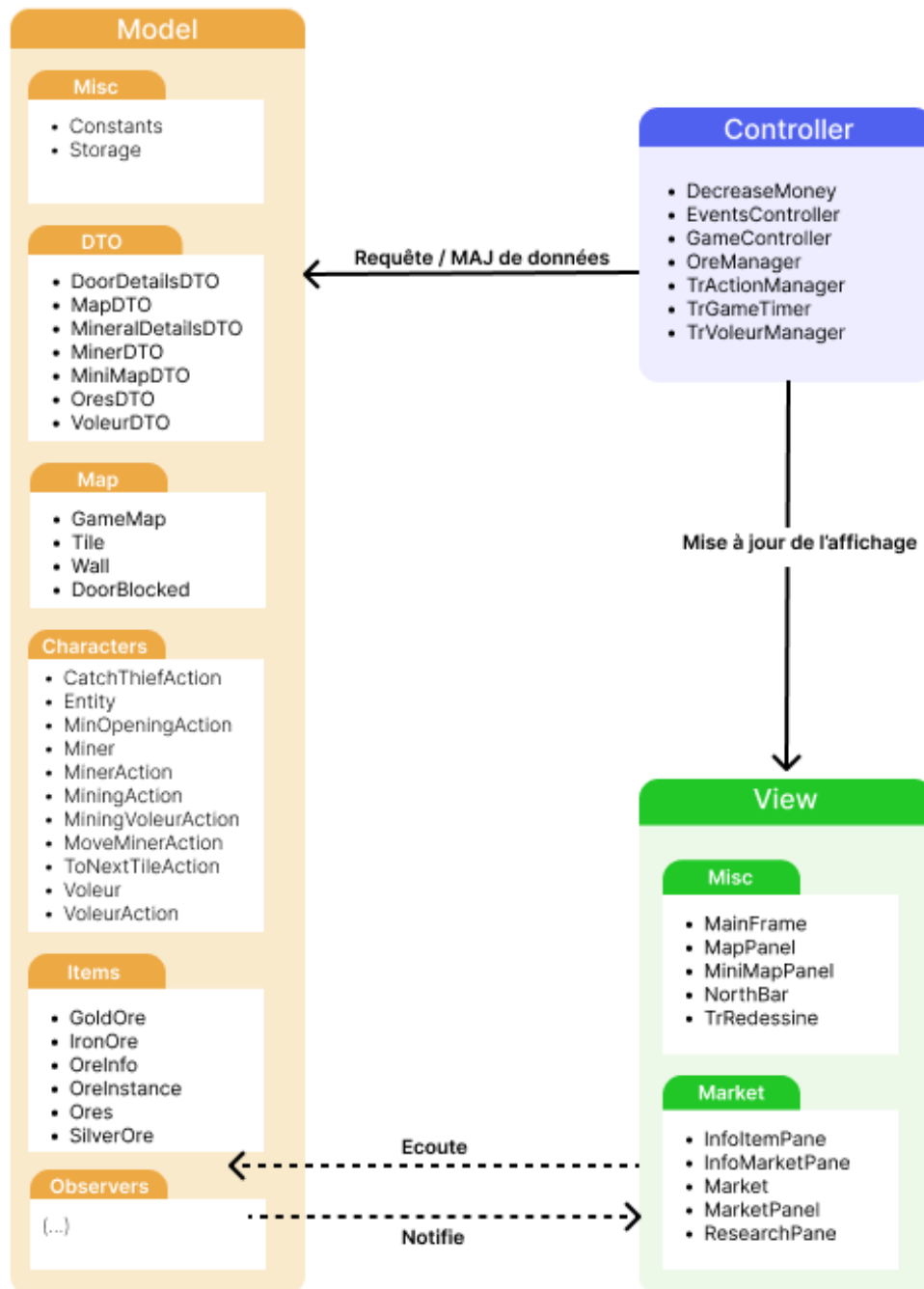
Contrôleur : Le contrôleur fait le lien entre la vue et le modèle. Il interprète les actions de l'utilisateur, orchestre les modifications du modèle et actualise la vue. Les Manager sont des thread orchestrant la logique dynamique de jeu.

- *ActionManager et VoleurManager* : Encapsule et assure la bonne gestion des actions liées aux état interne des mineurs et voleurs.

Model : Divisé en paquet distinct pour catégoriser les model selon leurs logiques :

- *DTO* : On adopte le DTO pattern pour regrouper et transférer les données observables dans les notifications, simplifiant ainsi la communication entre les couches du modèle et de la vue
- *Observers* : Inclut les observateurs chargés de notifier la vue des changements dans le modèle. Ces observateurs permettent une communication asynchrone entre le modèle et la vue, garantissant que la vue est mise à jour en réponse aux changements d'état dans le modèle.
- *characters, items, etc...* : Définis les modèles correspondants.

Vue : La vue concentre tous les composants liés à l'interface utilisateur. Chargée d'afficher les données et de faciliter l'interaction avec l'utilisateur, la vue est délibérément détachée du modèle pour préserver le MVC et son indépendance fonctionnelle



4.2 Conception détaillée

Cette section détaille l'ensemble des fonctionnalités, en fournissant une description textuelle précise pour chaque fonctionnalité, ainsi que la logique sous-jacente à sa conception. Pour faciliter la compréhension des flux impliqués, un pseudo-code simplifié et explicatif est présenté. De plus, pour chaque fonctionnalité, un diagramme de classe simplifié vient compléter la description, offrant une représentation visuelle supplémentaire du processus.

La carte étant composée d'une structure un peu complexe mêlant deux matrices, nous allons décomposer son explication en trois parties, la génération de la carte, les layout des tuiles et l'intégration de la minimap.

1 - Génération de la Carte du Jeu

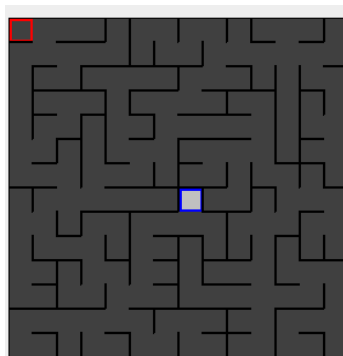
Explications :

La carte du jeu est générée sous la forme d'un labyrinthe au lancement de la partie. Les joueurs naviguent de case en case à travers ce labyrinthe. Pour synchroniser l'état de la carte entre différentes vues (par exemple, la vue principale et la *minimap*), des *observers* (*MapObserver* et *MiniMapObserver*) sont mis en place. Ces *observers* reçoivent les informations nécessaires via des objets de transfert de données (*MapData* et *MiniMapData*).

Génération du Labyrinthe :

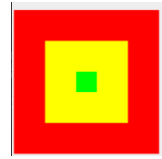
La carte est constituée d'une matrice de tuiles (*Tile*) et de murs (*Wall*), générée à l'aide de l'algorithme de Kruskal.

1. Initialisation : Chaque case de la matrice est initialisée comme une instance de *Tile*. La taille de la matrice est déterminée par des constantes prédéfinies.
2. Regroupement : Chaque *tuile* est placée dans son propre ensemble.
3. Construction des Murs : Des murs sont créés autour de chaque *tuile*. Les ensembles de tuiles sont ensuite mélangés pour permettre la sélection aléatoire nécessaire à la génération d'un labyrinthe unique.
4. Fusion des Ensembles : Les ensembles de *tuiles* sont fusionnés en supprimant les murs adjacents entre deux ensembles qui partagent un mur commun, mais qui ne sont pas encore fusionnés. Cela crée des passages dans le labyrinthe.
5. Unification : L'étape de fusion est répétée jusqu'à ce qu'il reste un unique ensemble, assurant que toutes les tuiles sont accessibles depuis n'importe quel point de la carte. Etape 5 : On réitère l'étape 4 jusqu'à avoir un unique set qui nous garantiras que d'une manière ou d'une autre, toutes les *Tile* sont accessibles depuis n'importe quel point de la carte.



Définition des Zones :

Le labyrinthe comprend des zones concentriques avec différentes propriétés (par exemple, types de minerais ou de monstres différents). Les zones sont définies en calculant dynamiquement la distance minimale des bords et sont divisées en quartiers : central, extérieur, et intermédiaire.



Placement de la Tuile de Départ et de Sortie :

La tuile de départ est située au centre du labyrinthe. La tuile de sortie est choisie aléatoirement parmi les quatre coins de la carte.

Notification des Observateurs :

Lorsque la tuile courante change, les observateurs sont notifiés via la méthode *setCurrentTile()*, qui définit également les attributs *Point* et *isExplored* de la tuile courante.

Toutes les propriétés tel que la tuile de départ/sortis, l'ouverture dans un mur, la tuile courante etc... sont définis directement au niveau du modèle de tuile qu'on discutera dans la prochaine fonctionnalité.

Pseudo code :

Fonction initLabyrinthe()

Initialiser matriceTuiles avec taille définie par constantes

Pour chaque case dans matriceTuiles

Initialiser chaque case comme instance de Tile

Attribuer des propriétés de base à chaque Tile (état, visibilité, etc.)

Fin Pour Chaque

appliquerAlgorithmeKruskal(matriceTuiles)

définirZonesConcentriques(matriceTuiles)

placerTuileDeDépartEtSortie(matriceTuiles)

notifierObservateursChangement()

Fin Fonction

Fonction appliquerAlgorithmeKruskal(matriceTuiles)

Initialiser ensembles pour chaque Tile

Créer ensemble murs entre Tiles

Mélanger et sélectionner murs pour création passages selon Kruskal

Fusionner ensembles jusqu'à unicité pour accessibilité complète

Fin Fonction

Fonction définirZonesConcentriques(matriceTuiles)

Calculer distance minimale depuis bords pour chaque Tile

Assigner zone (centrale, intermédiaire, extérieure) selon distance

Fin Fonction

Fonction placerTuileDeDépartEtSortie(matriceTuiles)

Choisir position centrale pour tuile de départ

Choisir aléatoirement un coin pour tuile de sortie

Mettre à jour états de tuiles correspondantes dans matriceTuiles

Fin Fonction

Fonction notifierObservateursChangement()

Créer MapData et MiniMapData avec informations actuelles

Pour chaque observateur dans listeObservateurs

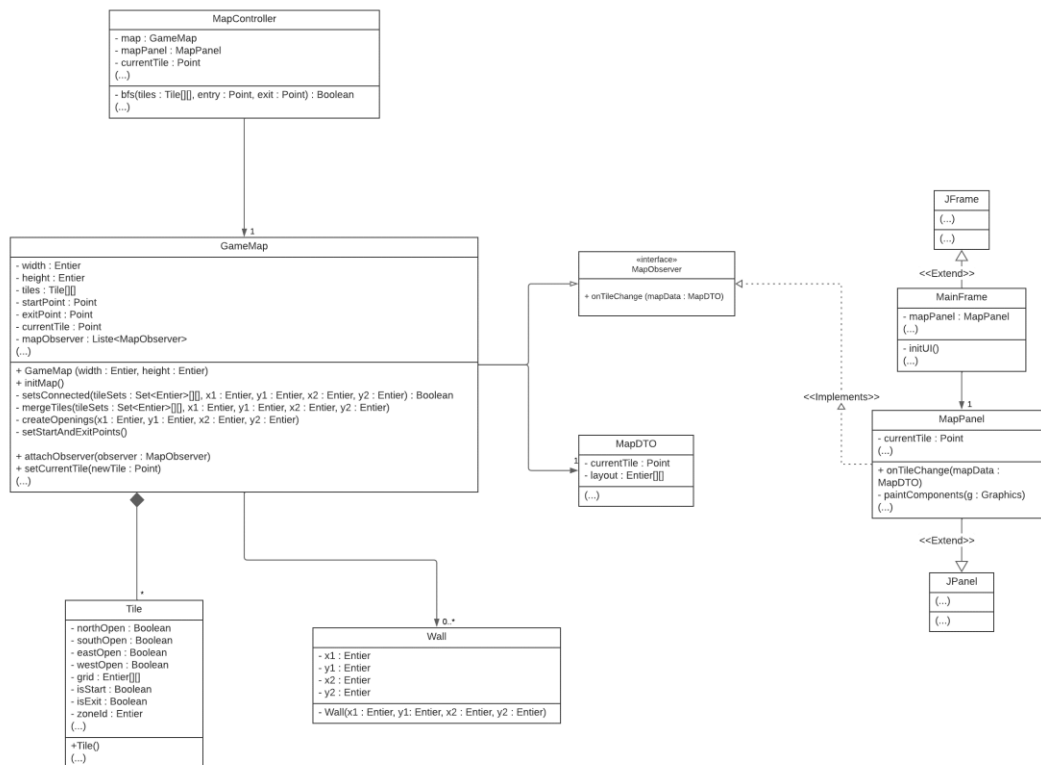
Notifier observateur avec données pertinentes

Fin Fonction

// Exécution au démarrage du jeu ou lors de la nécessité de régénérer la carte

initLabyrinthe()

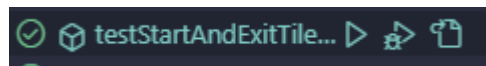
Diagramme de classe :



Test : src/test/TestMap.java

Une carte comporte une entrée et une sortie :

Condition de test : 1000 fois



La sortie est atteignable depuis l'entrée :

Condition de test : 1000 fois



Toutes les tuiles sont accessibles :

Condition de test : 1000 fois



2 - Création des Tuile

Explications :

Chaque tuile est conçue avec des attributs pour définir ses ouvertures, son état d'exploration, et si elle constitue l'entrée ou la sortie du labyrinthe. En outre, chaque tuile comprend une grille interne définie par une taille constante, qui sera utilisée pour donner forme à la tuile en termes de contenu et d'apparence.

Configuration des Ouvertures :

Pendant la phase de création du labyrinthe, on finalise les tuiles en leur donnant leur forme spécifique.

Pour chaque tuile, on parcourt l'ensemble des indices de la grille interne pour définir les bords et les coins selon les ouvertures existantes.

Définition du Contenu des Tuiles :

On définit la partie interne de la tuile en se basant sur une grille de valeurs. Ces valeurs sont associées à différents types d'assets, déterminés par des constantes.

Les bords et les coins de la grille interne de chaque tuile sont définis en fonction des ouvertures correspondantes, permettant ainsi de créer des portes dans les directions appropriées.

On remplace aléatoirement certains éléments de la tuile par des obstacles pour ajouter de la variété et éviter la monotonie.

Affichage dans la vue :

On crée une nouvelle classe *ImageLoader* qui va charger les différents assets et les mapper avec une chaîne de caractère pour clé, et ce, afin de mettre en cache les assets. Quand le modèle notifie un changement dans la tuile, la vue reçoit désormais également le layout de la tuile courante. Elle va donc parcourir ce layout et utiliser *//ImageLoader* pour charger chaque Image dans la grille selon le type de ce dernier.

Pseudo code :

Fonction initTuiles()

Pour chaque tuile dans la carte du jeu

Initialiser chaque tuile avec attributs par défaut

Définir si tuile est entrée, sortie, ou standard

Initialiser grille interne de la tuile selon taille constante

configurerOuvertures(tuile)

définirContenuTuile(tuile)

Fin pour

notifierObservateursAvecChangement()

Fin fonction

Fonction configurerOuvertures(tuile)

Pour chaque côté dans [Nord, Est, Sud, Ouest]

Si côté doit avoir une ouverture

Définir ouverture sur le côté spécifié de la tuile centrale

Fin si

Fin pour

ajusterBordsEtCoins(tuile)

Fin fonction

Fonction définirContenuTuile(tuile)

Pour chaque cellule dans grille interne de tuile

Si cellule est bord ou coin et ouverture est présente

Configurer cellule pour représenter ouverture

Sinon

Attribuer type de contenu basé sur la position de la tuile

// coins, bordure ou sol

Fin si

Fin pour

Fin fonction

Fonction notifierObservateursAvecChangement()

Pour chaque observateur dans listeObservateurs

Créer MapData avec état actuel de la carte

Notifier observateur avec MapData

Fin pour

Fin fonction

Fonction afficherTuile(layoutTuile)

Pour chaque cellule dans layoutTuile

typeAsset = déterminerTypeAsset(cellule)

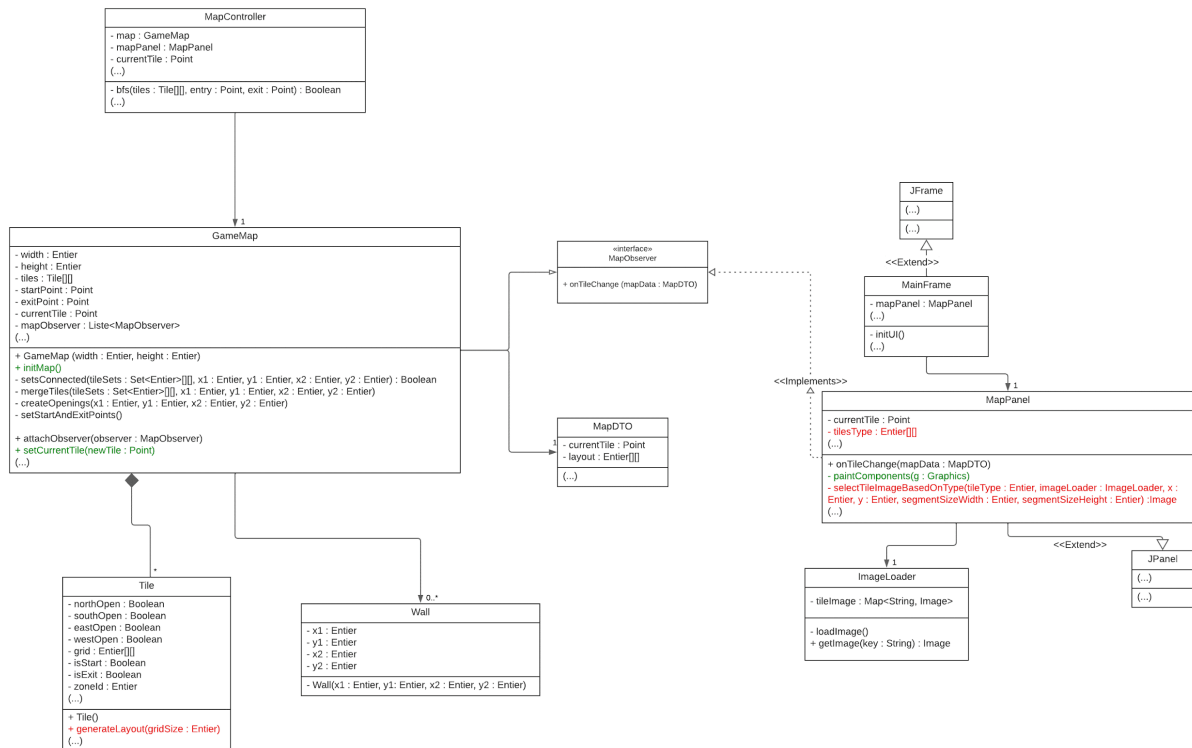
image = ImageLoader.chargerImage(typeAsset)

Afficher image à la position correspondante dans la vue

Fin pour Fin fonction

Diagramme de classe :

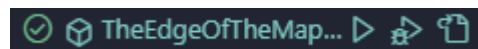
Sensiblement le même que la fonctionnalité 1, on ajoute les fonctions en rouge, les fonctions vertes sont celle modifié par l'ajout de cette fonctionnalité :



Test : src/test/TestTuile.java

La tuile est entourée de bordure / coin ou porte
et les reste représente le sol/obstacle/asset :

Condition de test : 100 fois



3 - Création d'une minimap

Explications :

Définition :

La *minimap* est un outil essentiel pour la navigation dans le labyrinthe du jeu. Pour sa création, une matrice booléenne est utilisée pour indiquer la présence de murs sur chaque tuile selon les quatre points cardinaux.

Génération pendant Kruskal :

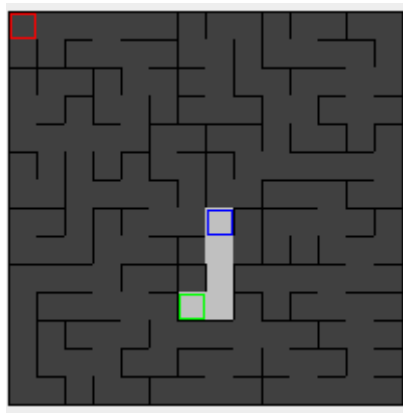
La matrice des murs (*wallsMatrix*) est définie pendant l'application de l'algorithme de Kruskal. Cette matrice est ensuite utilisée par le modèle *GameMap* pour informer les observateurs des attributs de la *minimap* via un DTO (*MiniMapDTO*), qui encapsule les informations sur les tuiles d'entrée, de sortie et courante, ainsi que la matrice des murs.

Affichage de la *Minimap* :

La vue *MiniMapPanel* écoute les notifications et réagit en dessinant la *minimap*, en se basant sur la taille de la matrice des murs. Les tuiles sont positionnées et les murs indiqués selon la matrice. L'entrée et la sortie sont marquées distinctement, les tuiles non explorées sont en gris et la position courante du joueur est mise en évidence.

Interaction Temporaire :

L'utilisateur peut interagir avec la *minimap* via une pression de la touche M pour changer sa visibilité. Cette interaction est gérée par la *MainFrame* mais sera à terme reléguée au contrôleur responsable des événements.



Pseudo code :

Fonction initMinimap()

appliquerAlgorithmeKruskalPourGénérerMatriceDesMurs()
préparerMiniMapDTOavecMatriceDesMurs()
notifierObservateursAvecMiniMapDTO()

Fin fonction

Fonction appliquerAlgorithmeKruskalPourGénérerMatriceDesMurs()

Initialiser wallsMatrix comme matrice booléenne basée sur la taille de la carte

Pour chaque tuile dans GameMap

Définir les murs de chaque tuile dans wallsMatrix selon l'algorithme de Kruskal

Fin pour

Fin fonction

Fonction préparerMiniMapDTOavecMatriceDesMurs()

Créer MiniMapDTO avec les données actuelles, incluant:

- la matrice des murs (wallsMatrix)
- les positions des tuiles d'entrée, de sortie et courante

Retourner MiniMapDTO

Fin fonction

Fonction notifierObservateursAvecMiniMapDTO()

Pour chaque observateur dans listeObservateurs

Notifier observateur avec MiniMapDTO

Fin pour

Fin fonction

Fonction dessinerMinimap(miniMapDTO)

Pour chaque cellule dans miniMapDTO.matriceDesMurs

Si cellule indique un mur

Dessiner mur dans MiniMapPanel

Sinon

Dessiner espace ouvert

Si cellule est tuile d'entrée, sortie ou courante

Marquer distinctement

Si tuile n'est pas explorée

Colorier en gris

Fin pour

Fin fonction

Fonction changerVisibilitéMinimap()

Écouter pression touche M

Si minimap est visible

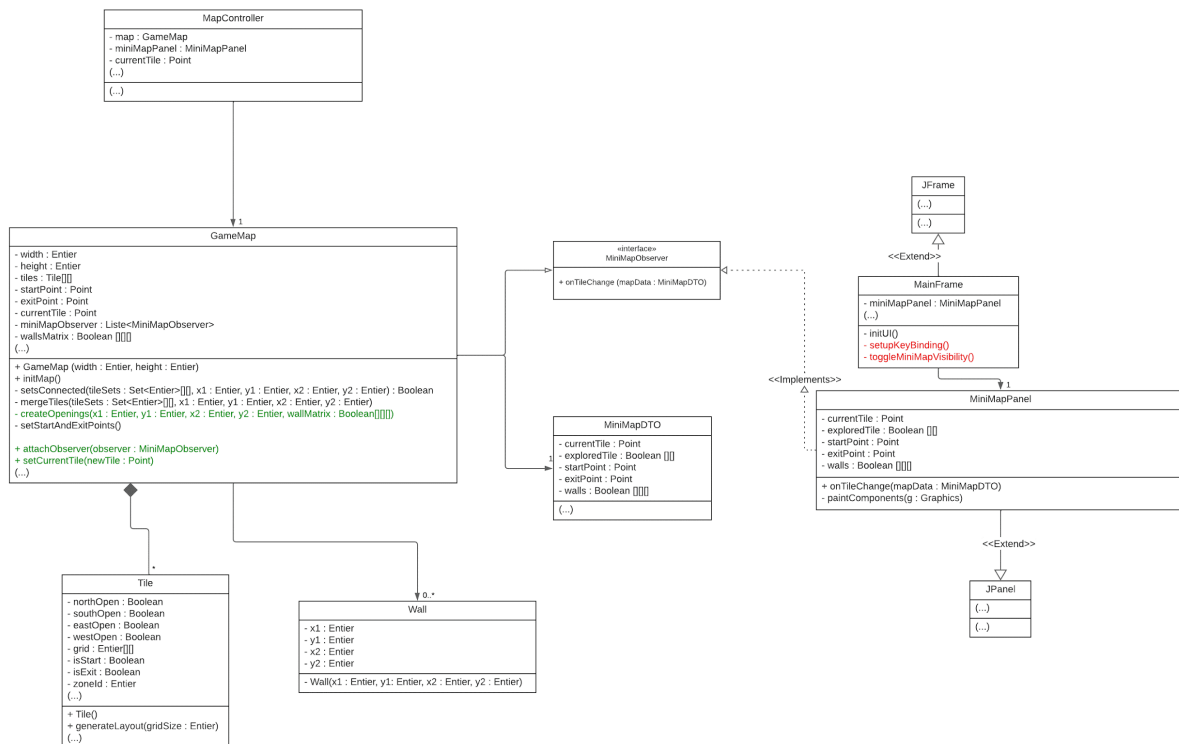
Cacher minimap

Sinon

Montrer minimap Fin fonction

Diagramme de classe :

La encore c'est très proche de la fonctionnalité deux, la vue notifiée est différente.



Test : src/test/TestTuile.java

Les tuiles explorées sont bien celle affiché :

Condition de test : 1000 fois

testMinimapExploredTiles()

Les tuiles marqué entrée/sortie sont bien celle affiché :

Condition de test : 1000 fois

testStartAndExitTilesArePresent()

4 - Création de minerais

Explications :

Classe Ores :

La création des minerais s'appuie sur le principe d'héritage en Java, avec une classe mère abstraite *Ores* qui établit un cadre pour le comportement commun des différents types de minerais. Chaque minerai possède un ensemble d'attributs détaillés :

- Identifiant Unique (*id*) : Permet des interactions spécifiques avec chaque instance de minerai.
- État de minage (*OreState*) : Énumération qui décrit les différentes phases de minage d'un minerai, telles qu'intact, en cours de minage, ou épuisé.
- Statut d'Interaction (*OreStatus*) : Énumération qui indique la disponibilité pour le minage, par exemple disponible, en cours d'exploitation ou en régénération.
- Temps de Récolte (*harvestingTime*) : Entier qui mesure le progrès dans le processus de minage.
- Temps Maximum de Récolte (*maxHarvestTime*) : Entier qui représente le temps total nécessaire pour régénérer un minerai.
- Prérequis (*requirement*) : Conditions nécessaires pour commencer le minage.
- Quantité (*quantity*) : La capacité totale de minerai disponible.
- Quantité Récoltée (*harvestedAmount*) : La quantité de minerai déjà extraite par le joueur.

Ces attributs sont essentiels pour simuler une expérience de minage réaliste et pour définir des règles de jeu cohérentes autour de l'exploitation minière.

Héritage et Spécialisation :

Des classes telles que *GoldOre*, *IronOre* et *SilverOre* héritent de *Ores*, permettant la réutilisation de code tout en personnalisant le comportement par le biais de méthodes surchargées. Cela favorise l'extension de la logique de minage pour divers types de minerais avec des caractéristiques et des comportements uniques.

OreManager :

OreManager orchestre la logique de placement des minerais sur la carte. À travers sa méthode *placeMineralsOnMap*, cette classe utilise la liste des tuiles de la carte pour déterminer où placer les minerais. La méthode tient compte des segments de sol appropriés et attribue aléatoirement une quantité prédéfinie de minerais à chaque tuile. De plus, *OreManager* dispose d'une méthode qui évalue le taux d'apparition des minerais en fonction du niveau de la zone dans laquelle se trouve la tuile.

Instance de Minerais :

La classe *OreInstance* crée une correspondance entre la localisation d'un minerai (*Point*) et l'objet *Ores* lui-même. Chaque *OreInstance* possède un identifiant unique et est gérée par *OreManager* pour une association précise entre les minerais et leur emplacement sur la carte. Les *OreInstance* sont ajoutés aux attributs de la tuile correspondante pour faciliter le suivi et l'affichage.

Notification et Affichage :

Lorsque les minerais sont générés et placés, *OreManager* notifie les observateurs à l'aide d'un *OresDTO*, qui transmet les informations de chaque minerai. *MapPanel*, la vue responsable de l'affichage, utilise ces informations pour représenter graphiquement les minerais en tenant compte de leurs types et états. Ce système garantit que seules les instances pertinentes pour la tuile courante sont prises en compte, permettant ainsi une gestion efficace de la mémoire et une mise à jour dynamique de l'affichage lors des déplacements du joueur.

Pseudo code :

Fonction initMinerais()

Pour chaque type de minerai dans [GoldOre, IronOre, SilverOre]

Créer un nouvel objet du type de minerai

Attribuer un ID unique à l'objet minerai

Définir l'état initial du minerai (par exemple, intact)

Définir le statut initial du minerai (par exemple, disponible)

Définir le temps de récolte initial et maximal

Définir les prérequis nécessaires pour miner l'objet

Déterminer la quantité totale et la quantité extraite par le joueur

Fin Pour

Fin fonction

Fonction placeMineraisSurCarte()

Récupérer la liste des tuiles de la carte

Pour chaque tuile dans la liste des tuiles

Déterminer si la tuile est éligible pour placer un minerai

Si oui, choisir aléatoirement une quantité de minerais à placer

Pour chaque quantité de minerai

Choisir aléatoirement une position sur la tuile

Créer une nouvelle instance de OreInstance

Attribuer le minerai à l'instance

Ajouter l'instance de minerai à la liste de tuile

Fin Pour

Fin Pour

Fin fonction

Fonction notifierObservateursDesMinerais()

Créer un nouvel OresDTO

Ajouter les informations de chaque minerai à OresDTO

Notifier chaque observateur avec OresDTO

Fin fonction

Test :



5 - Interaction avec la carte

Centralisation des Événements de Souris :

Un *EventController* est utilisé pour centraliser tous les événements de souris. Il capte les interactions telles que les clics et les mouvements de la souris, puis les redirige vers les contrôleurs appropriés en fonction de la logique de l'interaction, spécifiquement le *GameController* pour les interactions avec la carte.

Gestion des Clics et du Survol :

L'*EventController* doit être capable de différencier les événements et de déterminer l'objet à l'origine de l'appel. Pour cela, chaque objet interactif sur la carte, comme les ouvertures entre les tuiles, doit avoir une "hitbox" – un rectangle définissant la zone cliquable de l'objet. Ces hitbox permettent de distinguer précisément les interactions malgré le changement de la taille de la fenêtre. Cette "hitbox" est interne au modèle des minerais/portes, elle ne sera pas connue de la vue. La vue se contente de dessiner la réponse à l'interaction pendant que le modèle et le controller se charge des parties calculs de la position de la hitbox et détection de collision.

Création des Hitbox :

Les hitbox sont générées en associant les événements de clic et de survol avec la fonction *getOpeningsRectangle*. Cette fonction crée des rectangles pour chaque ouverture basée sur le modèle de la carte, permettant à la souris de reconnaître ces ouvertures. On enrichit donc le Model *Tile* de cette fonction qui va consister à mapper la direction d'une ouverture avec un Rectangle.

On est maintenant en mesure de déterminer si un rectangle contient la position cliqué ou survolée et donc d'envoyer au *GameController* les informations permettant d'appliquer sa logique.

Redimensionnement Dynamique :

La taille du rectangle de clic est ajustée dynamiquement en fonction de la taille actuelle de la fenêtre du jeu grâce à un événement *componentResized*, qui met à jour les attributs *segmentWidth* et *segmentHeight*. Ces dimensions correspondent à la taille d'une tuile dans l'affichage de la carte et assurent que les hitbox s'adaptent correctement lors du redimensionnement de la fenêtre.

Logique d'Événement dans le *GameController* :

Pour gérer la logique d'événement dans *GameController*, on commence par distinguer les différents éléments de la carte. On utilise pour cela un *enum* avec des valeurs indiquant l'absence d'interaction (*None*) ou la présence d'une ouverture survolée (*Opening*). Le *GameController* utilisera ces informations pour déterminer la réponse appropriée à une interaction de la souris.

Interaction avec les minerais

Lors d'un clic sur un minerais on affiche les détails de ce dernier dans une fenêtre qui sera calculée relativement à la position du minerai en le décalant. On donne les informations à dessiner à la vue qui passera le boolean *showOreDetails* à vrai et redessineras le panel. Tout clic commence par réinitialiser ce boolean à faux.



Observers & setters

Si un clic est détecté, on utilise la direction renvoyée par

l'EventController pour charger la tuile désignée comme nouvelle tuile courante.

Si un événement de survol est détecté on change l'état de *l'enum* et on demande à la vue de modifier le type du curseur.

Pseudo code :

Fonction initEventController()

Écouter tous les événements de souris (clics, survols)

Associer ces événements à leur traitement spécifique

Fin fonction

Fonction traiterClicsEtSurvolSouris(positionSouris)

Si clic détecté

objetCliqué = identifierObjetCliqué(positionSouris)

Si objetCliqué est valide

traiterClicObjet(objetCliqué)

Sinon si survol détecté

objetSurvolé = identifierObjetSurvolé(positionSouris)

Si objetSurvolé est valide

traiterSurvolObjet(objetSurvolé)

Fin fonction

Fonction identifierObjetCliqué(positionSouris)

Pour chaque objet interactif dans la carte

hitbox = générerHitbox(objet)

Si hitbox contient positionSouris

GameController.chargerNouvelleTuile (position, direction)

Fin fonction

Fonction identifierObjetSurvolé(positionSouris)

// Logique similaire à identifierObjetCliqué

Fin fonction

Fonction générerHitbox(objet)

// Basé sur la fonction getOpeningsRectangle pour les ouvertures

// Utilise les attributs segmentWidth et segmentHeight pour l'adaptabilité

Retourner hitbox calculée

Fin fonction

Fonction traiterSurvolObjet(objet)

changerÉtatEnum(objet.type)

Si objet.type est ouverture

demanderVueDeChangerTypeCursueur()

Fin fonction

Fonction ajusterDimensionsHitbox()

Écouter événement componentResized

Ajuster segmentWidth et segmentHeight

// Assure que les hitbox s'adaptent à la taille de la fenêtre

Fin fonction

Fonction chargerNouvelleTuile(position, direction)

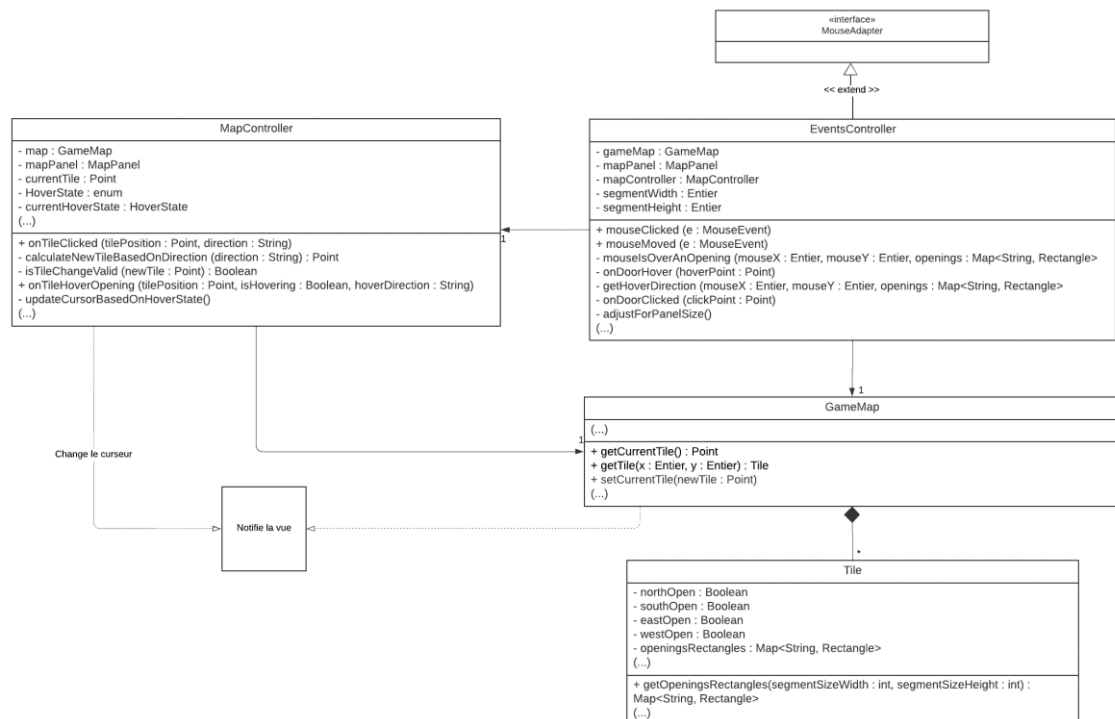
Vérifier que la nouvelle tuile soit dans les limites de la carte

Changer la tuile courante

Appeler setNouvelleTuile(tuileCourante) // notifie les observers

Fin Fonction

Diagramme de classe :



Test : src/test/TestTuile.java

Vérifier que le clique sur une ouverture renvoie
La bonne tuile suivante :



Condition de test : 1000 fois

6 - Création d'un inventaire




Explications :

Création d'un objet contenant les ressources du joueur

Afin de stocker les ressources de l'utilisateur, nous avons mis en place un *singleton* qui est donc la classe *Storage*. Elle regroupe les différents minerais (fer, argent, or) et tout ce qui est relatif à l'argent du joueur (la monnaie, pas le minerai). Au début de partie, tous ces composants valent 0. On a donc une instance de la classe unique récupérable grâce à la fonction *getInstance*.

Ajout de l'inventaire à l'interface graphique

Les ressources sont affichées à l'écran grâce à la classe *NorthBar* qui est un *JPanel* qui va s'occuper simplement d'afficher chaque ressource une par une dans une sorte de bandeau en haut de l'écran. Voici son rendu :

Temps: 00:34  26  6  0

Interaction avec l'inventaire

Afin de collecter des ressources et donc d'augmenter nos quantités de fer, d'argent... on a une quantité de minerai renvoyée par le thread *TrActionManager* que l'on va stocker via la fonction *addStorage*(type, montant) qui s'occupera d'additionner les valeurs que l'on a dans l'inventaire avec celles reçues du Thread.

De plus, le Thread *TrGameTimer* vient également ajouter 1 de monnaie chaque seconde via toujours la même méthode.

Enfin on peut également augmenter notre argent (toujours la monnaie), en attrapant les voleurs, qui se fait cette fois-ci par la classe *CatchThiefAction*.

On a aussi la possibilité de perdre des ressources dans le cas précis où l'on achète un mineur ou bien que l'on améliore les outils de nos mineurs déjà existants. Cela se fait par le *Thread DecreaseMoney* qui fera appel à *removeStorage*(type, montant) , qui fait symétriquement le contraire de *addStorage*.

Implémentation des observateurs

D'abord on va attacher l'intégralité des Observers nécessaire à une liste *d'observers* via la fonction *attachObservers*(observer)

On va ensuite faire en sorte que l'affichage dans la *NorthBar* s'actualise grâce à *notifyObservers*(), depuis la classe *Storage*, qui va associer nos quantités de ressources aux différents labels affichés dans le *NorthBar*.

Pseudo code :

Fonction getInstance()

 Si l'instance n'existe pas

 on la créer

 renvoyer l'instance

Fin fonction

Fonction addStorage(type, montant)

 On récupère la ressource r associé au type

$r = r + \text{montant}$

 notifyObserver(type, r)

Fin fonction

Fonction removeStorage(type, montant)

 On récupère la ressource r associé au type

$r = r - \text{montant}$

 notifyObserver(type, r)

Fin fonction

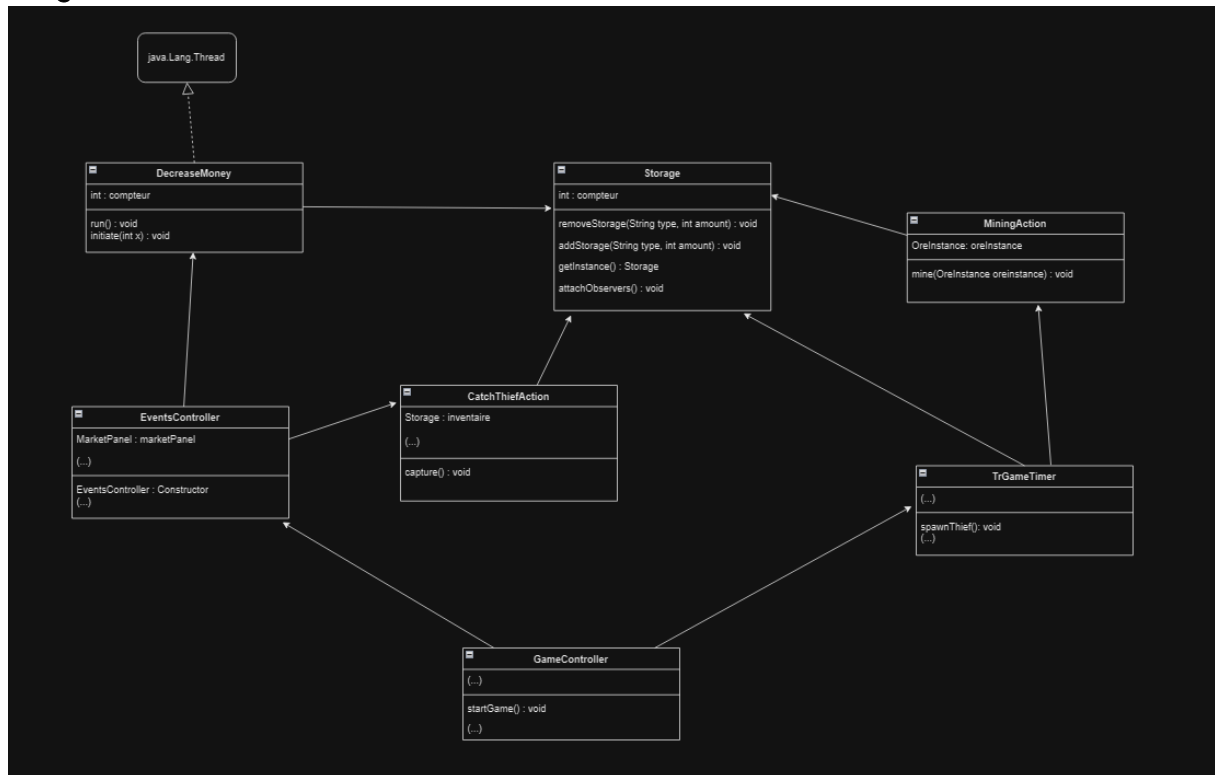
Fonction notifyObserver(type, montant)

Pour chaque observer parmi ceux qui nous sont attachés

 observer.onStorageChange(type, montant);

Fin fonction

Diagramme de classe :



7 - Création d'un Timer Globale

Explications :

La classe *TrGameTimer* est responsable de la gestion du temps dans le jeu. Elle hérite de la classe *Thread* et s'exécute en tant que thread séparé, ce qui lui permet de mettre à jour périodiquement différentes composantes du jeu. Son constructeur prend en paramètres plusieurs éléments essentiels du jeu tel que la barre de progression, la carte du jeu, le panneau de la mini-carte, etc.

C'est ce Thread qui permet de mettre à jour le temps de jeu et l'argent global du joueur dans la barre de jeu.

Spawn des Unités Hostiles, les Voleurs

Le nombre de voleurs dans le jeu est défini dans la classe *Constants* (*Constants.MAX_THIEF*). Le Thread fait apparaître un voleur toutes les 30s et seulement si le nombre max de voleur n'est pas dépassé à l'aide de la méthode *spawnThief()*.

Lorsqu'un voleur est créé et est donc placé sur la map, il apparaît forcément sur une tuile où se trouve un mineur (le mineur étant choisi au hasard via le paramètre *random*). Ainsi les voleurs apparaissent seulement sur des tuiles explorées et pas n'importe où sur la map.

Les voleurs ayant, tout comme les mineurs, des outils pour miner (ce qui définit quel minerai ils peuvent miner), l'outil donné aux voleurs lorsqu'ils apparaissent est calculé via le paramètre *currentVoleurTool* qui est mis à jour dans la méthode *updateThiefTools()*.

Mise à jour des outils des différents Voleurs

L'avancée du jeu et l'évaluation de l'avancement du joueur est calculé via son argent stockés dans la classe *Storage*. Ainsi, l'update des outils des voleurs se fait selon différents paliers:

- < 50 money : les voleurs spawn avec une pioche en bois (la plus faible).
- à partir de 50 money : les voleurs spawn avec une pioche en fer.
- à partir de 100 money : les voleurs spawn avec une pioche en argent.
- au-dessus de 150 : les voleurs spawn avec une pioche en or (la plus forte).

Pour éviter les bugs lorsqu'un voleur est en mouvement et que sa pioche se voit être améliorée, l'update d'outil d'un voleur se fait seulement s'il est dans les états "inactif", c'est-à-dire dans les deux états suivants : "*WAITING*" et "*IDLE*". S'il n'est pas dans l'un des deux états, alors il est ajouté à une liste stockant tous les voleurs qui ont besoin d'une amélioration. Sa pioche sera alors améliorée seulement la prochaine fois qu'il sera dans un des deux états "inactifs" cités plus haut.

Pseudo code :

Fonction TRGamTimer(northbar, gameMap, mapPanel, miniMapPanel)

Initialiser les attributs avec les valeurs passées en paramètres

Initialiser startTime avec le temps actuel

Initialiser random

Fin fonction

Fonction run()

Tant que isRunning est vrai:

Attendre 1 seconde

Mettre à jour le temps écoulé

Incrémenter les pièces du joueur

updateThiefTools()

Mettre à jour l'affichage de la barre de progression

Si le nombre de voleurs est inférieur à la limite et que 30s se sont écoulées :

spawnThief()

Fin fonction

Fonction spawnThief()

Si des mineurs existent :

Sélectionner un mineur au hasard

Créer un voleur à la position du mineur sélectionné

Ajouter le voleur à la carte

Abonner le voleur aux changements dans la carte du jeu

Abonner le voleur aux changements d'état des minerais

Abonner le voleur aux changements de tuile sur la minimap

Abonner le voleur aux changements de tuile sur la carte du jeu

Fin fonction

Fonction updateThiefTools()

Si l'argent dépasse un certain palier (50/100/150) :

initialiser la *currentVoleurTool* à la bonne valeur (Iron/Silver/Gold)

Pour tous les voleurs

Si le voleur est inactif :

ajouter la *currentVoleurTool* à son inventaire

Sinon :

ajouter le voleur à *voleurUpdate*

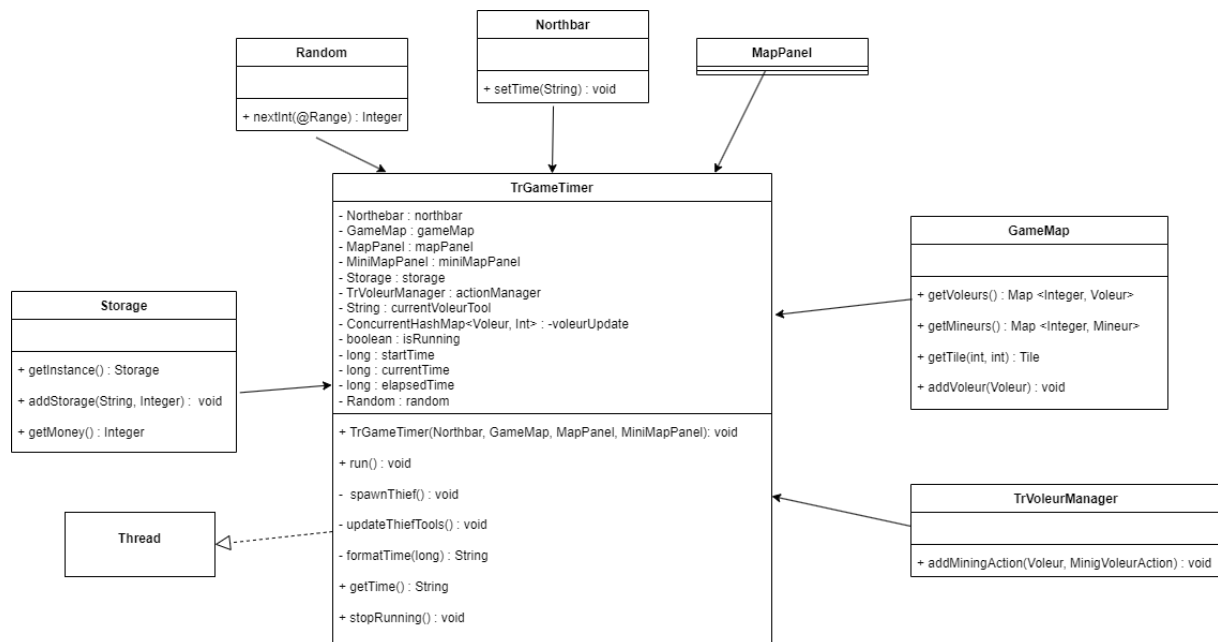
Pour chaque voleur dans *voleurUpdate* :

Si le voleur est inactif :

ajouter la *currentVoleurTool* à son inventaire

Fin fonction

Diagramme de classe :



8 – Implémentation des Mineurs

Explications :

Les mineurs sont des entités clés dans le jeu, agissant comme des unités contrôlables par le joueur pour l'exploitation minière. Héritant de la classe *Entity*, ils requièrent une gestion dynamique, notamment à travers des interactions utilisateurs et une mobilité sur la carte de jeu. Chaque mineur est doté d'une *hitbox* pour gérer les collisions et les interactions, et est associé à une position spécifique sur la carte (*tuile*), bien qu'il puisse se déplacer à travers la *GameMap*.

Gestion des Collisions/Hitbox :

Contrairement aux minerais qui sont statiques, les mineurs se déplacent et nécessitent donc une méthode de mise à jour dynamique de leur *hitbox* à travers *updateHitbox*, basée sur leur position actuelle. Cette HitBox est également interne au modèle des mineurs, elle ne sera pas connue de la vue. La vue se contente de dessiner la réponse à l'interaction pendant que le modèle et le controller se charge des partie calculs de la position de la hitbox et détection de collision.

Gestion des États :

Les comportements variés des mineurs sont gérés à travers un système d'états, représentés par un *enum* listant toutes leurs actions possibles. Ce système permet de contrôler le flux logique des interactions et des mouvements des mineurs.

Gestionnaire d'Événements :

Le jeu alterne entre deux états principaux - *isSelected* pour interagir avec les mineurs, et un autre pour l'interaction avec la carte. *L'EventController* capte les événements de clic et de survol, activant les méthodes correspondantes, notamment pour la sélection et le déplacement des mineurs.

Méthode de Déplacement :

Le mouvement des mineurs est calculé en fonction de leur position absolue sur l'écran, en utilisant un vecteur basé sur la position cliquée et la position initiale. Chaque mineur possède une vitesse propre, permettant un déplacement fluide et adapté.

Inventaire des Mineurs :

Pour miner différents types de minerais, les mineurs doivent posséder l'outil adéquat. Un inventaire (*liste de String*) est associé à chaque mineur, et une vérification est effectuée avant le minage pour s'assurer que le mineur dispose de l'outil requis par le minerai. Un système de *feedback* visuel informe l'utilisateur en cas d'outil manquant.



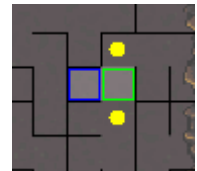
Affichage des changements :



La vue *mapPanel* écoute les changements dans le modèle des mineurs de plusieurs manière :

- Elle associe l'identifiant de chaque mineur à une *state*, afin de modifier son état visuel (partie action)
- Elle associe *l'identifiant* de chaque mineur à une *position de tuile* sur la carte, l'objectif étant de n'afficher que les mineurs présents sur la tuile courante.
- Elle associe *l'identification* des mineurs présents sur la carte à une *position en Pixel*, afin de les dessiner sur la tuile.

Enfin, par soucis d'UI/UX, un mineur doit être représenté sur la *minimap* pour qu'en tout temps, le joueur soit capable de voir où se situent ses mineurs. C'est pourquoi lorsqu'un mineur change de tuile, on notifie par le biais d'observateur *MiniMapPanel* de la position du mineur pour y dessiner un cercle jaune représentant les différents mineurs présents sur la tuile.



Pseudo code :

Classe Miner hérite de Entity

Fonction updateHitbox(segmentWidth, segmentHeight)

Calculer et mettre à jour les dimensions et la position de la hitbox

Fonction deplacerVers(destinationPixel)

Calculer le vecteur de déplacement vers la destination

Mettre à jour la position du mineur

Mettre à jour la hitbox avec la nouvelle position

Notifier les observateurs du changement de position

Fonction containsAny(requiredTools)

Vérifier si minerStorage contient au moins un des outils requis

Retourner vrai ou faux

Fonction addObserver(observer)

Ajouter un observateur à la liste des observateurs

Fonction notifyObservers()

Construire un objet MinerDTO avec les données actuelles du mineur

Notifier chaque observateur avec les données du mineur

Fonction buildMinerDTO()

Construire et retourner un objet MinerDTO avec les données du mineur

Fin Classe

Classe EventsController étend MouseAdapter

Fonction mouseMoved(e)

Gérer le survol des différents éléments (mineurs, minerais, ouvertures)

Fonction mouseClicked(e)

Traiter les clics sur les différents éléments interactifs

Gérer le déplacement ou l'action spécifique du mineur sélectionné

Fonction handleInteractiveElements(clickPoint)

Déterminer si un élément interactif (mineur, minerai, ouverture) a été cliqué

Traiter l'interaction spécifique si nécessaire

Fonction handleMoveSelectedMiner(destination)

Déplacer le mineur sélectionné vers la destination

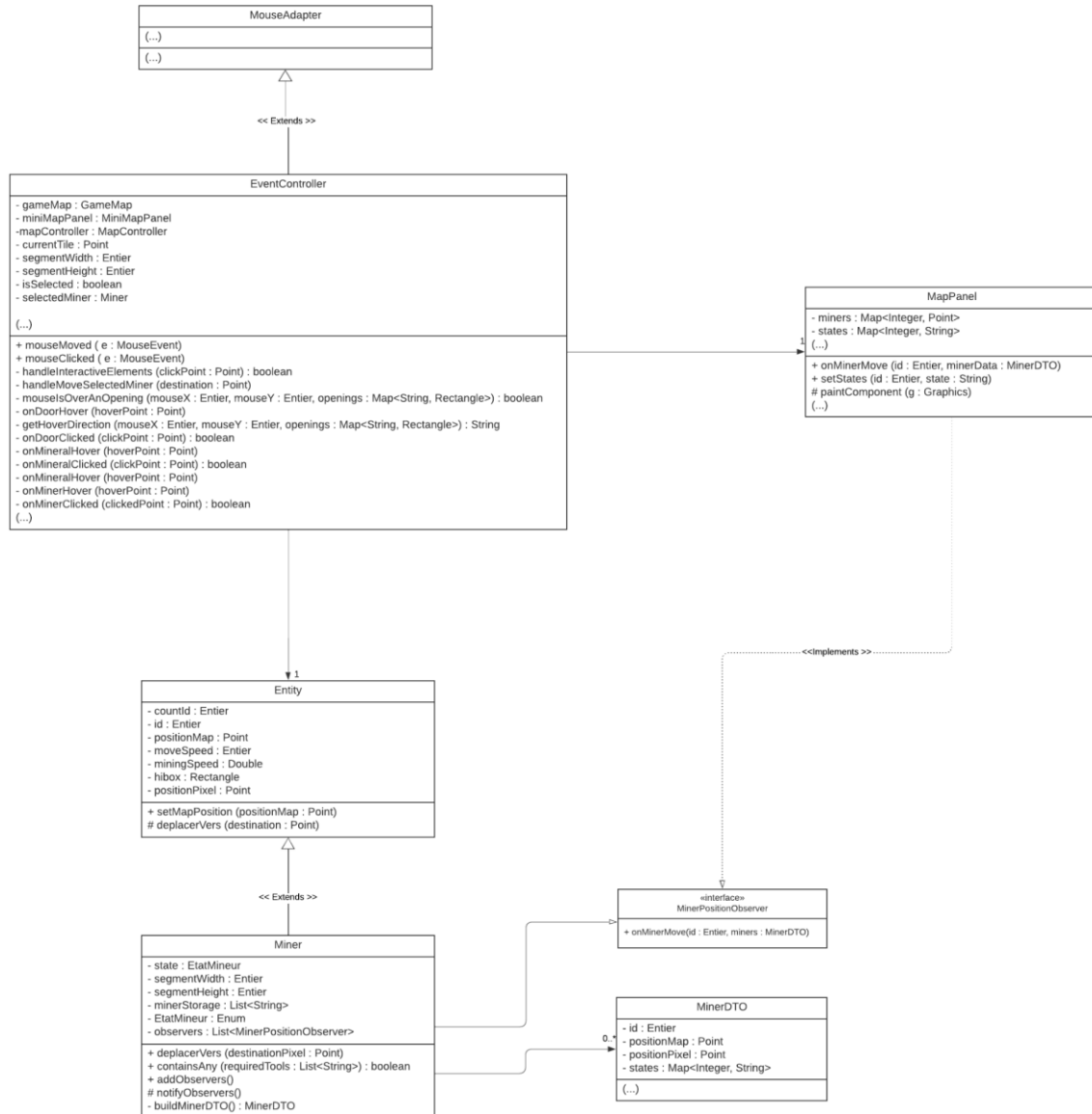
Mettre à jour l'état du mineur et effectuer l'action de déplacement

Fonction deselectMiner()

Désélectionner le mineur actuellement sélectionné

Fin Classe

Diagramme de classe :



9 – Création des modèles d'actions

Explications :

Le jeu implémente une architecture d'actions pour modéliser les interactions et le comportement des mineurs. La classe abstraite *MinerAction* sert de base, définissant la méthode *execute()* qui encapsule la logique spécifique à chaque type d'action. Cette approche favorise une évolution souple des entités et un découplage efficace des interactions.

Gestionnaire d'Événements :

Le contrôleur d'événements, *EventController*, gère deux états principaux : l'interaction avec la carte et l'interaction avec un mineur sélectionné. Un boolean *isSelected* permet de basculer entre ces états en fonction des interactions utilisateur, facilitant la distinction des actions à réaliser.

Cycle des Actions :

Les actions suivent un cycle de vie impliquant une phase initiale de déplacement vers l'objectif, puis l'exécution de l'action proprement dite. Une "marge acceptable" est calculée pour assurer que le mineur atteint sa destination. Le type d'action est déterminé par l'état actuel du mineur, permettant d'initialiser l'action appropriée (ex. *MiningAction*, *ToNextTileAction*, *MoveAction*).

Le manager d'action :

Avant de détailler ces actions il faut introduire le thread : *TrActionManager*. Son rôle est d'orchestrer le déroulement des objets de type actions. Il va donc lancer à intervalle régulier la méthode *execute* des différentes actions et passer l'état des mineurs et des minerais à *IDLE/AVAILABLE* une fois ces actions terminées. Il doit être en mesure d'adapter sa fréquence d'exécution aux unités de temps requise pour les différentes actions.

Il possède donc trois listes distinctes :

- Une liste d'action active (la classe mère), qui contiendras toutes les actions en cours à exécuter associé aux mineurs.
- Une liste de minerais épuisés qu'il faut régénérer sur laquelle on reviendra dans *MinerAction*.
- Une liste d'action terminée et donc à supprimer sans risquer la manipulation concurrente.

En outre, la gestion des actions au travers du thread permet de les rendre persistantes même si la vue ne dispose pas des références vers les mineurs en cours d'action. C'est-à-dire qu'un mineur peut apparaître et disparaître des tuiles dynamiquement, miner même si les tuiles ne sont pas les tuiles courantes etc...

MinerAction :

Le cœur du jeu est donc l'action la plus complexe. On va donc initier cette action grâce à *EventController*, l'utilisateur a choisis de cliquer sur un minerai en mode sélection, le controller va donc vérifier que le minerai sélectionné est bien disponible (*AVAILABLE*), s'il l'est, il faut maintenant vérifier que le mineur dispose de l'outils requis au minage. Si c'est le cas, on déclenche l'action en faisant passer la state du mineur à *TO_MINING* indiquant qu'il

se déplace vers l'action de miner, on crée cette action et ensuite on abonne la vue à cette action et on donne l'action au manager d'action qui va l'ajouter à sa liste d'action active.

Son traitement au sein du *thread* va être différent des autres actions. En effet, le *DELAY* d'exécution des *Thread* est défini à 16 pour simuler approximativement le 60 FPS. Hors, pour l'action de minage et de régénération, on veut incrémenter les compteur de 1 par seconde. On va donc devoir chercher les instances d'action de minage en particulier afin de lancer leurs méthodes *execute* toutes les secondes.

Premier Cycle de Vie des Minerais :

Une fois le minerai atteint, l'état du mineur change à *MINING* pour démarrer sa conditionnel dans *execute* permettant de lancer sa méthode *mine*.

Chaque seconde, à chaque appel de mine on notifie les observateurs avec un *MineralDetailsDTO*, contenant les informations encapsulées par *OreInfo*. Cette dernière inclut l'identifiant, le type, la quantité, la position, l'état du minerai, et le *harvestTime*. *MapPanel* utilise ces informations pour afficher un cercle visuel indiquant le temps de récolte restant. On notifie également dans le même temps les informations du mineur avec *MinerDTO* afin de modifier son état visuel et surtout le faire correspondre au minerais en cours de minage.

Deuxième Période du Cycle de Vie des Minerais :

La méthode *Mine()* de la classe *Ores* est appelée toutes les secondes par *TrMining* pour gérer le processus de minage, réduisant *harvestTime* et ajustant l'état du minerai en fonction. Lorsque *harvestTime* est à moitié écoulé, l'état passe à *HALF*, et une partie des ressources est distribuée au joueur. Lorsque *harvestTime* atteint zéro, l'état passe à *EMPTY*, et le minerai entre en période de régénération (*RESPAWN*), avec distribution du reste des ressources. *MapPanel* est régulièrement mis à jour pour refléter ces changements visuellement, notamment sur le cercle de progression et en affichant un *label* indiquant la quantité de minerai reçue.



Troisième Période du Cycle de Vie des Minerais :

Durant la phase de régénération, *Ores* utilise la méthode *regenerateOre()* pour augmenter progressivement *harvestTime*. Une fois ce temps de régénération terminé, les attributs du minerai sont réinitialisés, et une nouvelle quantité de ressources est définie.

Pour réaliser cette logique, dès que la méthode *mine* réduit à 0 le temps de récolte, on met le statut du minerais à *RESPAWN*, et on l'ajoute à la liste des minerais à régénérer de *l'ActionManager*. Ainsi, *l'actionManager* va appeler la méthode *regenerateOre* pour re-incrémenter cet attribut.

Une fois le minerai complètement régénéré, on notifie la vue et on le retire de la liste des actions en passant par une liste intermédiaire pour éviter les problèmes de concurrence.

ToNextTileAction :

Le traitement initial est identique à *MinerAction*, on déplace donc le mineur jusque-là porte en mettant son état à *TO_NEXT_TILE* indiquant un déplacement. Une fois que le mineur atteint la position, on ajuste cette dernière et on passe son état à *TRAVEL* pour signaler qu'on peut désormais le changer de tuile.

Ici, on calcul sa position sur la prochaine tuile en inversant sa position par rapport à l'entrée emprunter et en ajoutant une marge pour qu'il ne soit pas à l'extrémité de la tuile. Enfin, on change le mineur de tuile en modifiant son attribut.

Il ne reste plus qu'à notifier les observateurs pour faire apparaître ou disparaître le mineur de la liste à dessiner et mettre fin à l'action dans *l'ActionManager* en la supprimant de la liste.

MoveAction :

Représente les déplacements généraux des mineurs non liés à des actions spécifiques. Si un mineur est sélectionné et qu'un clic est effectué hors d'un contexte d'action précis, cette action de mouvement est initiée pour déplacer le mineur vers la position cliquée.

Note : On aborde pas la logique de détection des hitbox puisqu'elle est identique à celle utilisée pour le mode non sélectionné dans la fonctionnalité « interaction avec la map », on définit juste une conditionnelle et logique différente si un mineur est SELECTED..

Pseudo code :

Classe MiningAction étend MinerAction

Fonction execute()

Si l'état du mineur est TO_MINING et qu'il est suffisamment proche de sa destination

Changer l'état du mineur en MINING

Mettre à jour le statut du minerai en MINING

Définir la position du mineur à la destination

Sinon si l'état est TO_MINING

Déplacer le mineur vers la destination

Sinon si l'état est MINING

Exécuter la logique de minage sur oreInstance

Sinon

Annuler l'action de minage

Fonction mine(oreInstance : OreInstance)

Si le minerai est en train d'être miné et que l'état du mineur permet le minage (MINING ou SELECTED)

Miner le minerai via sa méthode mine() et ajouter la quantité minée à l'inventaire

Notifier les observateurs de la progression du minage

Si le minerai est épuisé

Ajouter le minerai à la liste de régénération dans TrActionManager

Sinon

Notifier que le minage est terminé

Fonction cancelMining()

Si le minerai est en cours de minage mais que le mineur n'est plus en état de minage

Réinitialiser le statut du minerai en AVAILABLE

Notifier que le minage a été annulé

Fonction pour la gestion des observateurs

Fonction auxiliaires pour construire et notifier avec des DTO :

buildDTO(oreId, ore, position, qtyMined) : Construit et retourne un MineralDetailsDTO

buildMinerDTO() : Construit et retourne un MinerDTO basé sur l'état actuel du mineur

Fin Classe

Classe MoveMinerAction hérite de MinerAction

Fonction execute()

Calculer la distance entre la position actuelle du mineur et la destination

Si l'état du mineur est "WALKING" et la distance est inférieure ou égale à Constants.MOVE_THRESHOLD Alors

Mettre à jour l'état du mineur à "IDLE"

Mettre à jour la position du mineur à la destination

Sinon

```

    Déplacer le mineur vers la destination via méthode de Mineur
  Fin Si
Fin Fonction
Fin Classe

Classe ToNextTileAction
Fonction execute()
  Calculer la distance entre la position actuelle du mineur et la destination
  Si l'état du mineur est TO_NEXT_TILE et la distance est inférieure au seuil de mouvement
    Changer l'état du mineur à TRAVEL
    Mettre à jour la position pixel du mineur à la destination
  Sinon si l'état du mineur est toujours TO_NEXT_TILE
    Déplacer le mineur vers la destination
  Sinon si l'état du mineur est TRAVEL
    Changer l'état du mineur à IDLE
    Mettre à jour la position de la carte du mineur à la nouvelle position
    Ajuster la position pixel du mineur sur la nouvelle tuile en fonction de la direction
    Notifier les observateurs que le mineur a changé de tuile
Fin fonction

Fonction attacherObservateur(observer)
  Ajouter l'observateur à la liste des observateurs
Fin fonction

Fonction notifierFinVoyage()
  Pour chaque observateur dans la liste des observateurs
    Notifier l'observateur que le mineur a terminé son voyage et a changé de tuile
  Fin pour chaque
Fin fonction
Fin Classe

Classe TrActionManager
Fonction run()
  Tant Que le thread n'est pas interrompu
    Parcourir la liste des actions des mineurs pour exécuter chaque action
    Si l'action est un minage et une seconde s'est écoulée depuis la dernière exécution
      Exécuter l'action de minage
      Enregistrer le moment de cette exécution
    Sinon si l'action est un déplacement ou un changement de tuile
      Exécuter l'action correspondante

    Vérifier si le moment est venu de régénérer des minerais
    Si une seconde s'est écoulée depuis la dernière régénération
      Parcourir la liste des minerais à régénérer
      Pour chaque minerai, vérifier si sa régénération est complète

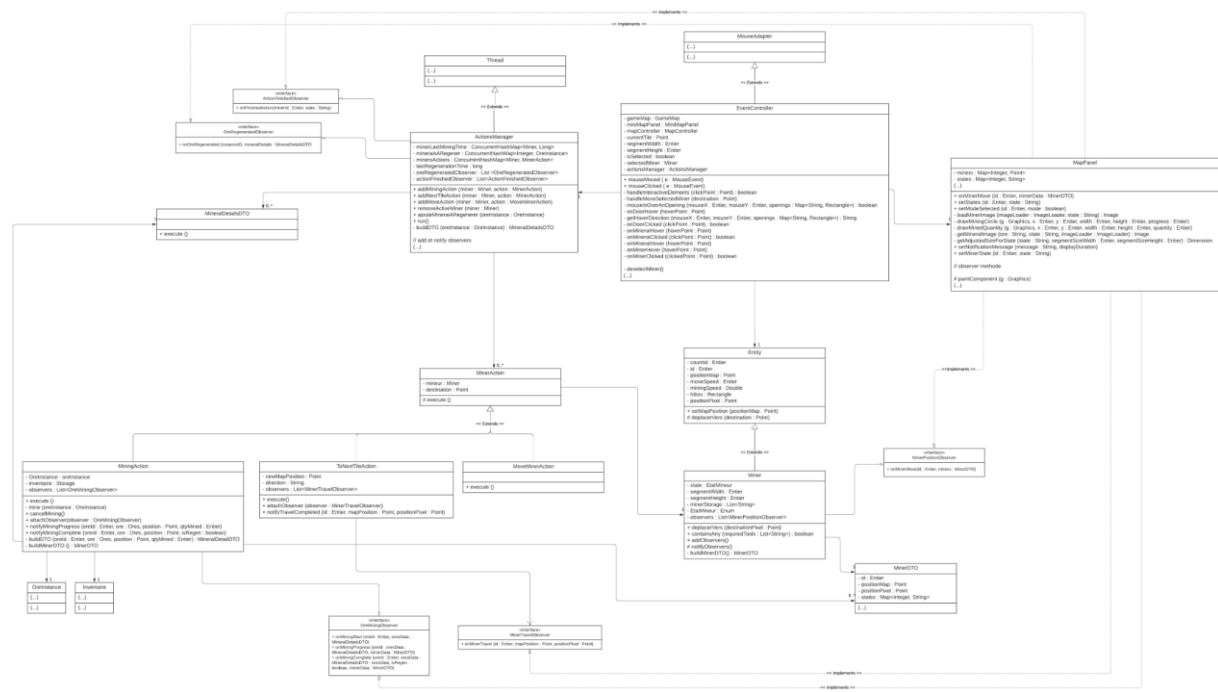
```



```

        Répéter cette boucle tant que le thread est actif
    Fin Fonction
Fin Classe

```



10 – Dynamisation Visuelle via les Sprites

L'intégration de sprites animés joue un rôle crucial en dynamisant l'expérience de jeu et en fournissant un retour visuel immédiat des interactions des joueurs avec les mineurs. Chaque état (*IDLE*, *MINING*, *MOVE*, *SELECTED*) est représenté par une série de 8 images, créant ainsi une animation fluide qui reflète l'activité des mineurs dans le jeu.

Préparation des Assets : Les différents sprites pour chaque état des mineurs sont préalablement découpés en 8 frames. Ces images sont chargées et mises en cache via *ImageLoader*, optimisant ainsi leur accès et leur rendu dans le jeu.

Gestion des Frames :

Variables de Contrôle : Deux variables, *spriteCounter* et *spriteNum*, sont utilisées pour orchestrer l'animation. *spriteCounter* augmente à chaque appel de *repaint()*, servant de base pour la mise à jour de l'animation. Quant à *spriteNum*, cette variable permet de définir le numéro du sprite à afficher.

Cycle d'Animation : *spriteNum* est incrémenté toutes les 10 itérations de *spriteCounter*, adaptant ainsi le rythme de l'animation à une fréquence visuelle agréable (~60 FPS). Cette logique permet de sélectionner la frame appropriée à partir du cache en fonction de l'état actuel du mineur associé à son *spriteNum*, assurant une transition fluide et cohérente entre les frames.

Thread de Rendu :

Afin de maintenir une animation constante, y compris dans l'état *IDLE* où les actions sont moins fréquentes, un thread dédié au rendu est mis en place. Ce dernier a pour mission d'invoquer *repaint()* périodiquement, garantissant ainsi que l'animation des mineurs reste active et réactive aux changements d'état induits par les interactions des joueurs.

Ce mécanisme assure que même en l'absence d'actions directes sur les mineurs, leur représentation visuelle reste dynamique et informative, reflétant fidèlement leur état et leurs activités dans le jeu.

Retour Visuel : L'état *SELECTED* est distinctement marqué par une aura bleue d'un pixel autour du sprite, signalant clairement au joueur quel mineur est sélectionné. Cette nuance visuelle renforce l'interaction utilisateur en mettant en évidence les mineurs ciblés pour des actions spécifiques.

Pseudo code :

Classe MapPanel étend JPanel implements <observers>

Fonction updateFrame

incrémente une variable interne spriteCounter

Si le compteur atteint 8 frame **Alors**

charger les assets de frame suivante

Si l'asset de frame est la dernière **Alors**

Remettre les numéro de sprite à 1

Fin Si

Fin Si

Fin Fonction

Fonction loadMinerImage et loadVoleurImage

recupere l'image correspondante mise en cache dans l'imageLoader

Fin Fonction

Fonction paintComponent

Associer les images récupérés aux Mineurs correspondants

Fin Fonction

Fin Classe

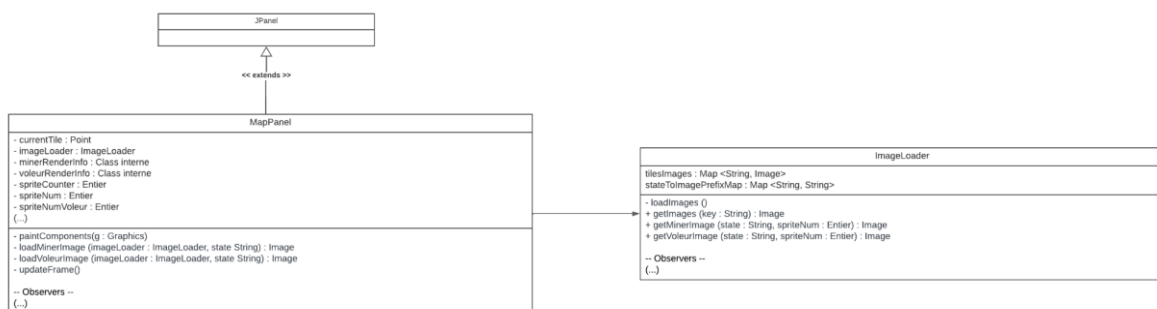
Classe ImageLoader

Met en cache les image

Créer des fonction pour concatener les String et récupérer les images correspondantes

Fin Classe

Diagramme de classe :



11 – Implémentation des unités hostiles

Explications :

Dans notre mini-jeu, les créatures hostiles non contrôlables par le joueur sont représentées par des voleurs. Les voleurs sont là pour nuire aux mineurs en volant leurs ressources. Pour cela, ils peuvent miner à leur place. Chaque ressource volée par les voleurs ne pourra pas être récupérée par les mineurs.

Les voleurs héritant de la classe *Entity* comme les mineurs, requièrent une gestion dynamique, notamment à travers des interactions utilisateurs et une mobilité sur la carte de jeu. Ces aspects de l'implémentation des voleurs n'a pas encore été effectué et n'est qu'à titre indicatif. Cependant, comparé aux mineurs, le joueur ne peut pas directement interagir avec les voleurs. En effet, les interactions utilisateurs devront forcément passer par les mineurs, c'est-à-dire que pour chasser les voleurs, les "tuer", l'utilisateur aura besoin de cliquer sur un mineur puis sur le voleur. Bien sûr, pour être efficace, le mineur devra remplir certaines conditions telles que l'utilisation d'outils appropriés et la capacité à anticiper les actions du voleur, voir à le poursuivre s'il tente de s'échapper pour éviter d'être appréhendé et continuer de voler.

Action de déplacement vers un minerai:

Pour faire bouger un voleur, la méthode utilisée est la même que pour les mineurs à un détail près. En effet, les voleurs ne vont que vers le minerai le plus proche d'eux, dans la tuile où il se situe. Cette liste de minerai est calculé à chaque fois que le voleur change de tuile et qu'il spawn. Ce minerai le plus proche est calculé via un algorithme très simple dans *Voleur.java* dans la méthode *getClosestMineral()*. Le mouvement du voleur se fait dans le *Thread TrVoleurManager*. Dans ce thread est stocké dans une *ConcurrentHashMap* toutes les actions des voleurs. Ainsi, dans le thread est appelé toutes les actions une à une. Toutes les actions des voleurs se trouvent être des instances de *MiningVoleurAction*. Différentes actions se trouvent dans cette classe, et selon l'état du voleur, une action est effectuée. En ce qui concerne le déplacement vers un minerai, le voleur doit être dans l'état *IDLE*. Ainsi le déplacement va débuter et le voleur va passer à l'état *TO_MINING* tant que le voleur n'est pas arrivé au niveau du minerai et n'a pas changé d'état, il fait appel à la fonction *déplacerVers()* qui déplace le voleur vers le minerai le plus proche. Lorsque le voleur est arrivé, le voleur change d'état et débute le minage

Chaque mineur possède une vitesse propre, permettant un déplacement fluide et adapté.

Action de minage:

L'action de minage se lance lorsque le voleur est dans l'état *READY_TO_MINE*, cela veut dire qu'il est sur un minerai. Ainsi l'excute fait appel à la fonction *mine(OreInstance)* qui va miner le minerai donné en paramètre.

La fonction *mine()* vérifie que le minerai est prêt à être miné et que le voleur est en état de minage. Ainsi *mine()* prévient la vue sur l'avancée du minage et mine le minerai. Si le minerai ne peut plus être miné, alors l'action de minage est finie et le minerai est épuisé. Ainsi, le

voleur passe en état *WAITING* et ne bouge plus pendant 8 secondes. Après ces 8 secondes, le voleur pourra continuer à chercher des minerais à miner et changer de tuile si nécessaire.

Action de changement de tuile:

Si le voleur ne trouve plus de minerai à miner dans la tuile (`getClosestMineral() == null`) alors le voleur va changer de tuile et va traverser une portes disponibles aux hasard. Pour cela il va changer d'état et passer en *TO_NEXT_TILE*. Et appeler la méthode `getNotBlockedOpen()` qui renvoie la direction d'une porte au hasard ouverte et traversable et va appeler `déplacervers(direction)`. Lorsque le voleur est arrivé à la porte, il change son état en *TRAVEL*. Il va ainsi changer de positionMap (position dans la mapGlobal) et donc changer de tuile, il va mettre à jour la liste des minerais qu'il peut miner et passer en état *IDLE*.

Les différentes actions se trouvant dans la fonction `execute()` de la classe *MiningVoleurAction* dépendent la plupart du temps de l'état du voleur. En effet, en se référant à l'état du voleur, on peut savoir à tout instant, quelle action il est en train de mener.

L'attente du voleur:

Étant quelque chose qui est géré à côté directement dans *TrVoleurManager*, cela se doit d'être expliqué apart.

Comme dit plus haut, le thread appel toutes les actions des voleurs, ainsi lorsqu' une action est appelée sur un voleur qui est en état *WAITING*, alors il est ajouté à la liste des voleurs en attente. Ce n'est pas réellement une liste mais une `ConcurrentHashMap<Voleur, Long>`, avec comme clé le voleur et comme valeur la seconde à laquelle il a été ajouté dans la HashMap. Ainsi si le voleur est dans cette `ConcurrentHashMap`, on vérifie si l'attente des 8 secondes a été respectée, le voleur passe en état *IDLE* et il peut de nouveau effectuer des actions. Sinon le voleur ne fait rien et attend. Si le voleur n'est pas dans la `ConcurrentHashMap` il peut effectuer ses actions en toute tranquillité.

En ce qui concerne les parties Pseudo code et diagramme de classe, nous n'avons pas eu le temps de les finir. Nous avons préféré peaufiner l'explication et vous rendre le rapport dans les temps. Le pseudo code étant très similaire à la fonctionnalité d'action des mineurs et le diagramme à la fonctionnalité des portes bloquées.

Pseudo code :

Diagramme de classe :

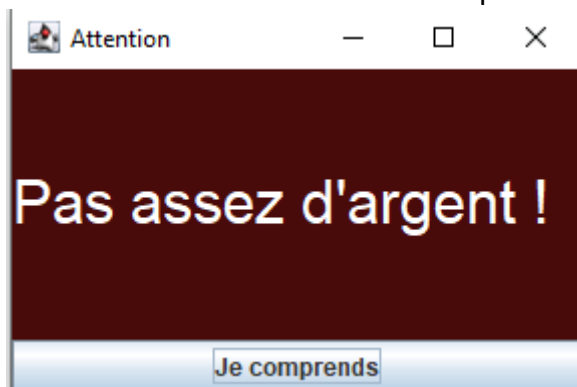
12 – Implémentation du magasin

Explications :

Le magasin étant une partie à part entière du projet peut être considéré comme assez générique mais elle procède comme suit. Il s'agit simplement d'une fenêtre héritant de *JPanel* afin de l'incorporer à notre mini-jeu. Cette interface est activable grâce à la touche S par le joueur et peut donc ouvrir l'interface du *Shop*. Son but est, pour le joueur, de rajouter un mineur dans la mine afin de rajouter de la main d'œuvre dans sa grotte ou alors, le joueur peut très bien améliorer l'(es) équipement(s) de son/ses mineur(s) sélectionné(s) en appuyant sur les boutons disposés sur l'interface. Le magasin contient de nombreux *JPanel* ou classes héritant de *JPanel* permettant un affichage optimal du shop. L'objectif ici est de concevoir une interface simple mais facile pour le joueur de le comprendre de manière claire.

Chaque amélioration d'objets de mineurs ou bien d'ajout de mineurs dans la grotte nécessite un certain montant de ressources lié à la classe *Storage*. Pour l'ajout de mineur, cela représente une certaine somme d'argent, et pour l'amélioration d'objets il s'agit d'un montant d'argent et de certains minerais particuliers à chaque amélioration. Lors d'un achat, on vérifiera que les ressources dont le joueur dispose sont bien supérieures au coût de l'achat et dans le cas contraire on rendra visible une fenêtre de classe *JFrame* qui indiquera au joueur son manque de ressource.

Voici le rendu de cette fenêtre, simplement composé d'un *Jpanel* contenant un *JLabel* au centre et un *JButton* afin de faire disparaître la fenêtre :

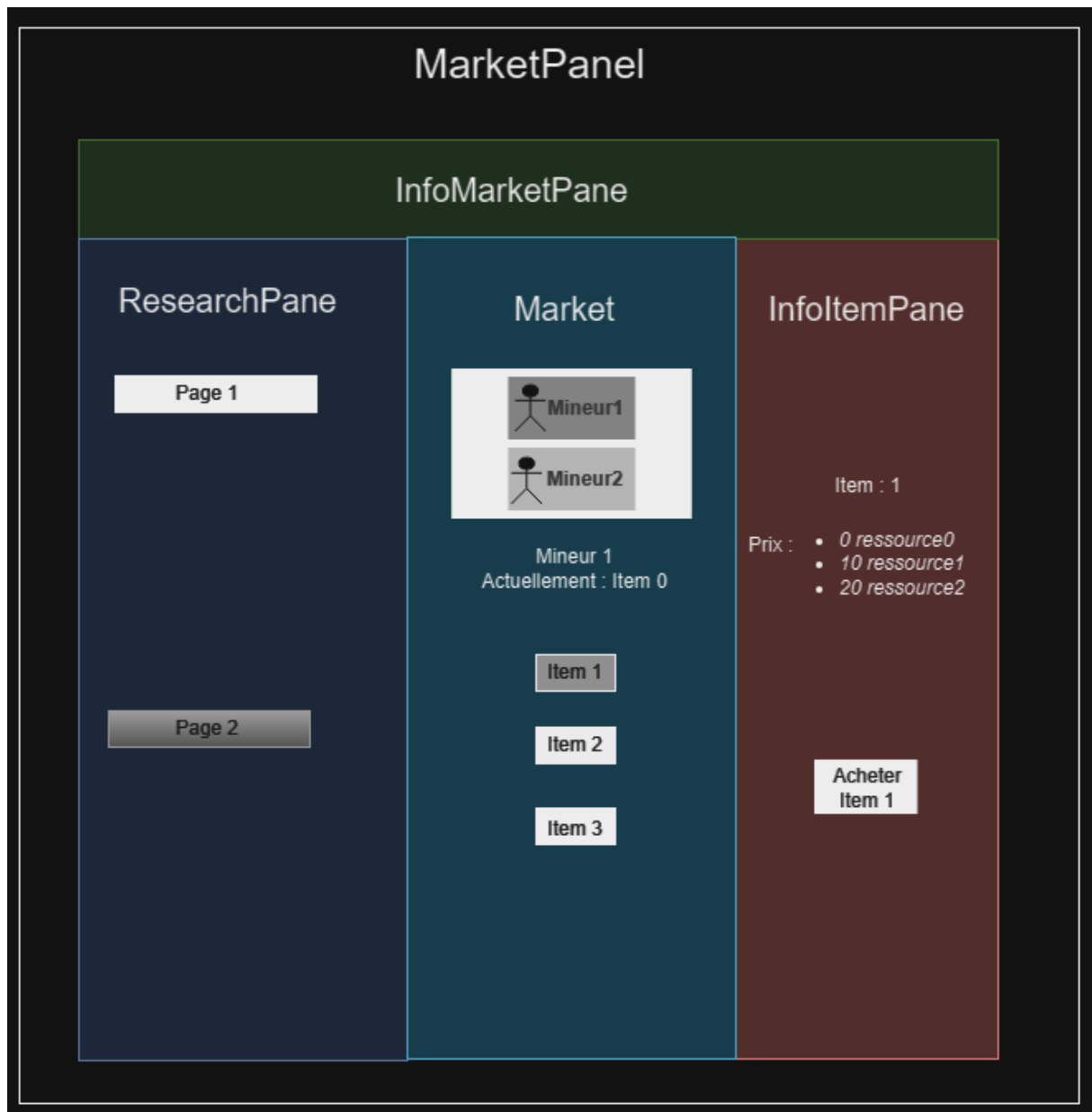


Dans le cas où l'on dispose d'assez de ressources, on fait appel au *Thread DecreaseMoney* décrit ci-dessous. Afin d'améliorer un mineur il faut donc le sélectionner, chose possible grâce au *Market* et à sa disposition.

Plus précisément on a une classe *MarketPanel*, représentant l'interface entière du *Shop* qui vient se découper en plusieurs *JPanel*: On place trois *JPanels* successivement de manière alignée verticalement. Le premier Panel étendu par la classe *ResearchPane* représente le panneau de recherche, afin de déterminer si l'on veut acheter un mineur ou bien en améliorer un. Ensuite, juxtaposé à sa droite, on a un *Market*, qui représente l'interface des mineurs actuellement présent dans le jeu mais ce qui nous permet d'interagir avec les mineurs et d'ainsi les améliorer (ou du moins leurs outils) représenté par un *JPanel* contenant des *JButtons* respectifs aux mineurs. On trouve également l'interface du marché et plus exactement de ce que l'on peut cliquer afin d'acheter. Finalement, On pose un *InfoItemPane*

qui va représenter les informations détenues via le *Market* qui va lui envoyer les informations (surtout grâce aux *eventsController*) de ce qui doit être affiché sur le panel afin d'aider le joueur à savoir combien coûte par exemple l'objet à acheter.

Voici le modèle du market :



Thread de Décrémentement de l'argent du joueur :

Pour ce Thread, on a une classe *DecreaseMoney* qui hérite de *Thread* et qui va nous permettre à chaque fois qu'un bouton pour acheter est appuyé, celui-ci va créer une nouvelle instance de la classe *DecreaseMoney*. On va également initier l'instance avec le type de ressource qui doit être décrémenté. Lorsque le thread est créé, on récupère alors la quantité de ressource qui doit être soustrait, puis le thread est lancé et on décrémente tous les x

delais, 1 à la ressource associée du joueur dans le singleton Storage et ceci entraînant le rafraîchissement du bandeau contenant l’affichage des ressources. Si une action requiert différentes ressources, on instancie la classe le même nombre de fois que l’on a de ressources différentes. Voici le pseudo-code lié au *Thread*:

Pseudo code :

Classe DecreaseMoney **hérite** de Thread

Fonction Initiate(entier n, entier type)

la borneMaximale = n

typeDeLaRessource = type

Fonction run ()

Tant que k < borneMaximale:

On décrémente la ressource associé au typeDeLaRessource du joueur de 1

On rafraîchit le bandeau informatif

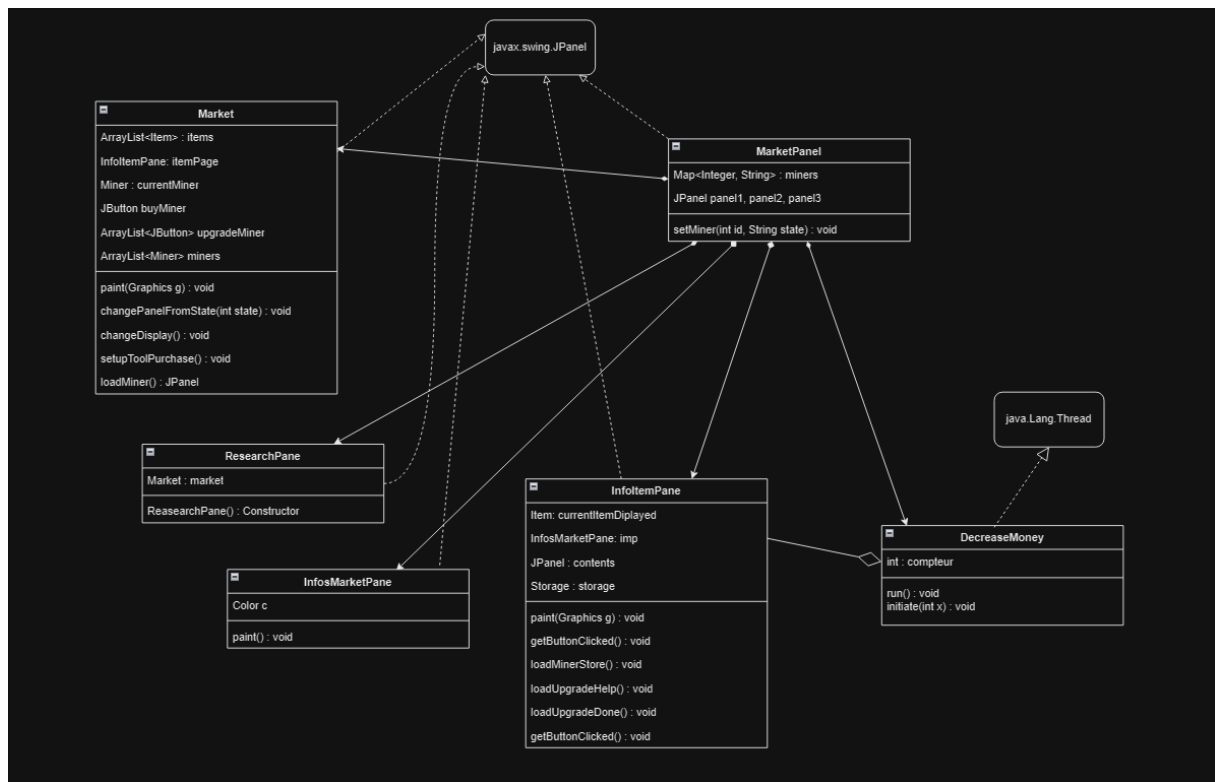
On attends un certain délai

On incrémente k de 1

Fin Classe

Diagramme de classe :

Voici le diagramme de classe du package *Market*:



13. Implémentation de fermeture entre les zones

Explications :

Cette fonctionnalité vise à intégrer des objectifs de jeu clairs, en utilisant les mécaniques de minage existantes. À travers le système de niveaux concentriques de la carte, des portes verrouillées seront mises en place, obligeant le joueur à posséder l'équipement approprié pour progresser.

Description des objets :

- Porte en fer : Nécessite une pioche de niveau Fer pour être débloquée. Visuellement, elle est représentée par une ouverture obstruée par des débris de la couleur de l'asset fer du jeu.
- Porte en argent : Nécessite une pioche de niveau Argent. Représentée par une ouverture obstruée par des débris de la couleur de l'asset argent.
- Porte en or : Seule une pioche de niveau Or peut ouvrir cette porte, donnant accès à la zone suivante (non implémentée pour cette version). Visuellement, elle est représentée par une ouverture obstruée par des débris de la couleur de l'asset or.

Les temps de déblocage varient selon le type de porte, incitant le joueur à acquérir des mineurs supplémentaires, car plusieurs mineurs peuvent travailler sur la même porte, réduisant le temps de minage proportionnellement au nombre de mineurs impliqués. Cela encourage également le joueur à répartir judicieusement ses ressources entre ses mineurs.

Création de l'objet porte :

Un objet *BlockedDoor* est créé pour encapsuler les propriétés d'une porte verrouillée. Cet objet stocke :

- Un identifiant unique, pour la gestion par la vue.
- Une direction pour localiser la porte sur une tuile.
- Une liste de prérequis correspondant aux outils nécessaires, utilisant les mêmes constantes que l'inventaire du mineur.
- Une durée de minage maximum et le temps restant à décompter.
- La tuile est associée à la porte, pour déterminer si la vue doit l'afficher.
- La position de la porte symétrique, pour une gestion conjointe.
- La position relative à la tuile et la position absolue sur la carte.

Des méthodes utilitaires sont implémentées pour manipuler les attributs de la porte (*getGridIndexForDirection*, *getOppositeDirection* et *updateDoorAsset*).

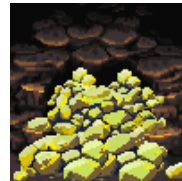
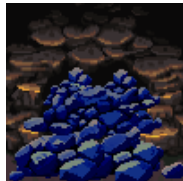
Deux fonctionnalités logiques principales sont ajoutées à l'objet *BlockedDoor* :

- *Unlock* : Similaire à l'action de minage, cette fonction décrémente la durabilité de la porte chaque seconde.
- *unlockDoor* : Une fois la porte complètement dégagée, cette fonction change l'asset de la tuile pour afficher une porte ouverte, donnant ainsi accès à la zone suivante.

Intégration des portes verrouillées dans le processus de création de la carte :

Une fois la carte et le *layout* des tuiles créées, le processus suivant est mis en place pour déterminer quelles portes doivent être verrouillées et avec quelles propriétés :

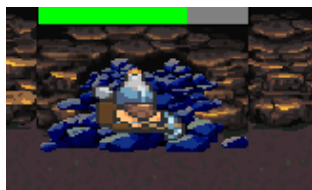
- Un parcours de l'ensemble de la carte est effectué pour identifier les tuiles aux frontières entre zones de différents niveaux.
- Pour chaque paire de tuiles identifiée, on détermine la direction de la frontière et on ferme les portes de façon symétrique entre les zones.
- Le type de porte (fer, argent ou or) et les prérequis sont déterminés en fonction des niveaux de zone adjacente. Une porte en fer est placée entre les zones de niveau 1 et 2, une porte en argent entre les niveaux 2 et 3.
- La grille de la tuile est mise à jour avec l'asset correspondant au type de porte déterminé, et un objet *BlockedDoor* est créé et stocké dans une *Map* associant la direction à la porte verrouillée.



Enrichissement des actions de mineur :

Un nouveau type d'action, *MinerOpeningAction*, est ajouté. Cette action, héritant de *MinerAction*, prend un mineur et une destination et implémente la fonction *execute*. Elle se voit ajouter un attribut *BlockedDoor*.

L'EventsController, lors d'un clic sur une porte verrouillée, va soit afficher des informations si aucun mineur n'est sélectionné, soit créer une *MinerOpeningAction* si un mineur est sélectionné, débutant ainsi le processus de déblocage.



La fonction *execute* de *MinerOpeningAction* dirige le mineur vers sa destination, change son état en *UNLOCKING_DOOR* une fois à proximité, et notifie la vue du début du minage. La fonction *unlock* de *BlockedDoor* est appelée chaque seconde pour simuler le temps de déblocage, et la vue est notifiée de la progression.

À la fin du minage, *unlockDoor* de *BlockedDoor* est appelée pour ouvrir les portes symétriques, changer l'état du mineur et notifier les observateurs.

Observateurs et vue :

L'observateur *MinerOpenDoorObserver* implémente trois méthodes pour chaque phase d'ouverture : *onUnlockStart*, *onUnlockProgress* et *onUnlockCompleted*, transmettant les informations pertinentes pour la représentation graphique de l'état des mineurs et de la progression du déblocage des portes.

Du côté de la vue, une structure de données composée de *Maps* imbriquées est créée pour associer chaque tuile à ses portes verrouillées. La vue est mise à jour au début, pendant, et à la fin du processus d'ouverture, reflétant les changements d'état des portes et des mineurs concernés. Ces changements interviennent uniquement sur les porte présente dans la *Map* ci-dessus, leur présence est déterminée lors des différentes notifications : si la tuile courante correspond à la tuile de ou des portes.

Visuellement, le mineur sera représenté par ses frames de minage et on créera une barre de progression basée sur le temps max et le temps restant du déblocage.

Ces informations sont transmises via des DTO *MinerData* et *DoorDetailsDTO*.

Pseudo code :

Note : le code est dense, on le simplifie à ses fonctionnalités clés

```
Classe BlockedDoor
    Définir les caractéristiques d'une porte fermée.

    Fonction unlock
        Décrémente la variable interne de durabilité
    Fin Fonction

    Fonction unlockDoor
        Débloque la porte en changeant son état interne et celui de sa porte
        symétrique sur la tuile adjacente
    Fin fonction
Fin classe

Classe GameMap
    Fonction closeDoorBetweenZone
        Pour chaque tuile dans la carte Faire
            // Vérifier les quatre cardinalités
            Si la cardinalité est ouverte Alors
                closeDoor porte et sa symétrique sur la tuile adjacente
            Fin Si
        Fin pour chaque
    Fin Fonction

    Fonction closeDoor
        Dans tile bloquer la porte avec un boolean
        Déterminer le niveau des tuiles adjacentes
        Si niveau 2 et 3 <- porte en argent
        Sinon porte en Fer
        Créer la porte avec les prérequis correspondants
    Fin Fonction
Fin Classe

Classe MineOpeningAction extends MinerAction
    Fonction execute
        Récupérer la distance séparant la position du mineur de la porte
        Si le mineur est dans l'état TO_ENEMY cette distance est inférieure ou égale à
        la marge définie Alors
            Commencer le déblocage
            Notifier les observateurs du début de déblocage
        Sinon Si vers_porte Alors
            Déplacer le mineur vers la porte
```

Sinon Si déblocage Alors

Unlock <- Appel porte.unlock()

Si unlock est vrai Alors

Le mineur passe IDLE

Appeler unlockDoor

Notifier les observateurs de la fin du déblocage

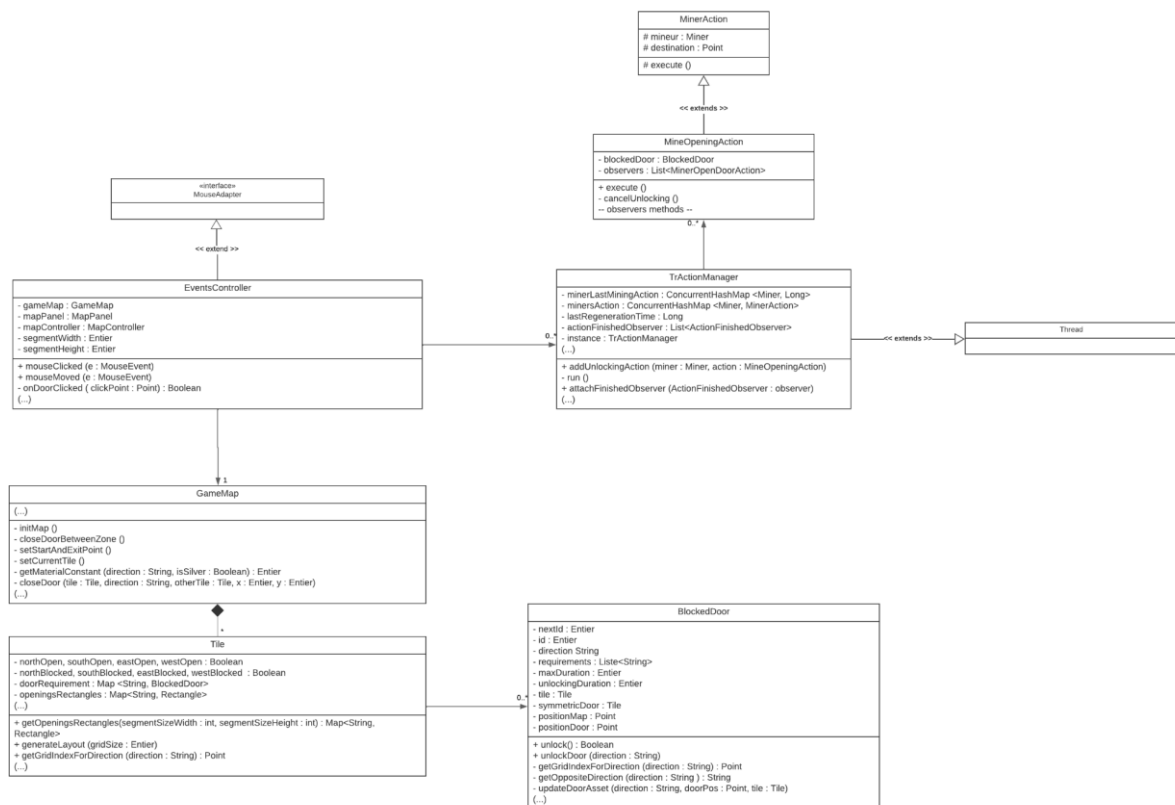
Sinon mettre fin au déblocage

Fin Fonction

Fin Classe

Diagramme de classe :

Les interactions étant relativement similaires sur la manière dont sont récupérés les Miner et les observers, on se concentre sur le flux de l'action entre l'événement et le gestionnaire d'action.



14. Implémentation d'interfaces UI/UX

Explications :

L'objectif de cette fonctionnalité est d'enrichir l'expérience utilisateur en fournissant des détails sur les interactions avec les objets tels que les minerais et les portes verrouillées. Ces interfaces visuelles offriront des informations sur les outils nécessaires pour l'interaction et le temps requis pour le minage ou l'ouverture.

Point d'entrée de l'interaction :

L'*EventController* sera enrichi pour gérer les clics sur les minerais ou les portes fermées lorsque aucun mineur n'est sélectionné. Lors de tels clics, les détails de l'élément interactif seront transmis à la vue. A l'inverse, le clic sur un mineur affichera automatiquement ses informations.

Gestion de la vue :

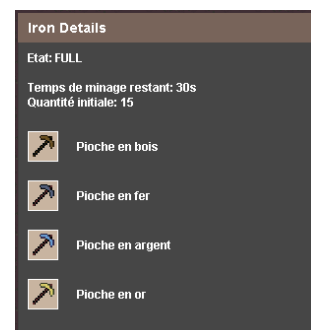
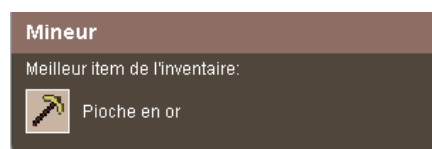
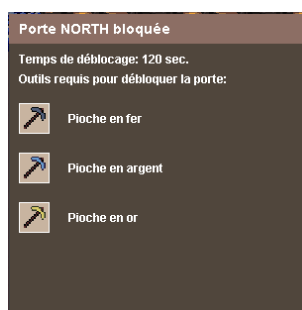
La vue maintiendra un booléen pour les portes fermées et les minerais indiquant si les informations ont été assignées par l'*EventController*. Lors du rafraîchissement de l'interface (*repaint*), si ce booléen est vrai, les interfaces correspondantes seront dessinées.

Dessin des interfaces :

Pour représenter visuellement les informations, un rectangle sera dessiné, positionné relativement au point de clic avec un décalage pour l'associer visuellement à l'objet interactif. Ce rectangle contiendra deux zones distinctes :

- Header : Cette zone affichera le nom de l'élément cliqué, permettant une identification rapide par l'utilisateur.
- Body : Cette zone contiendra les informations détaillées telles que les outils requis pour l'interaction. Pour les prérequis de minage, des icônes représentant les outils nécessaires (représenté par des assets de pioche) seront utilisées à côté de label décrivant les outils.

Ces interfaces fourniront au joueur des informations contextuelles essentielles, facilitant la prise de décision stratégique sur les actions à entreprendre dans le jeu, notamment quel mineur utiliser et quand investir dans des améliorations ou de nouveaux outils. *Cette logique sera étendue à toute les info-bulle UI/UX.*



Pseudo code :

Classe EventController **hérite** de Thread

Fonction mouseClicked

Mettre showDoorDetails et ShowOreDetails de mapPanel à **faux**

Fin Fonction

Fonction onDoorClicked

Pour chaque porte **Faire**

Si mineur n'est pas sélectionné **Faire**

Si la porte est fermée **Faire**

Envoyer à la vue les information de DoorBlocked

Mettre showDoorDetails de mapPanel à **vrai**

Fin Si

Fin Si

Fin Pour Chaque

Fin Fonction

Même logique onMineralClicked

Fonction onMinerClicked

Pour chaque mineur **Faire**

Donner à la vue l'inventaire du mineur sélectionné

Fin Pour Chaque

Fin Fonction

Fin Classe

Classe MapPanel

Fonction paintComponent

Si showDoorDetails est vrai **Faire**

Dessiner la fenêtre contenant les informations envoyées depuis

onDoorDetails

Fin si

Si showOreDetails est vrai **Faire**

Dessiner la fenêtre contenant les informations envoyées depuis

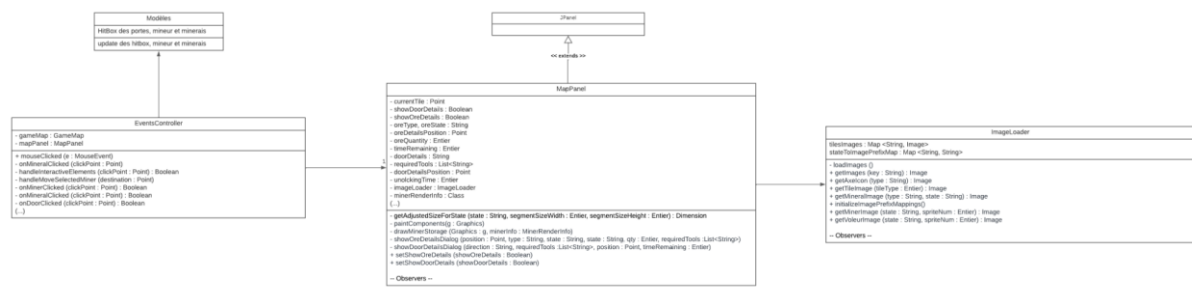
onOreDetails

Fin Si

Fin Fonction

Fin Classe

Diagramme de classe :



15. Création du gestionnaire de partie

Explications :

Le gestionnaire de partie centralise et orchestre les différents éléments du jeu, permettant une transition fluide entre les niveaux et facilitant la gestion des états de jeu. Cette fonctionnalité est essentielle pour implémenter les niveaux suivants du jeu et maintenir une structure cohérente.

Structure et Initialisation :

Initialisation des Vues : Les vues du jeu sont instanciées une seule fois dans l'application pour éviter des créations multiples inutiles. Elles sont conservées tout au long de la vie de l'application et passées au gestionnaire de partie pour être utilisées à chaque nouvelle partie ou niveau.

Rôle du *GameController* : Le *GameController* agit comme le cœur de la gestion de partie, en prenant la responsabilité de lancer le jeu. Il initialise les composants essentiels tels que les mineurs, les minerais et autres éléments interactifs du jeu. Le contrôleur configure également les observers nécessaires pour surveiller et réagir aux changements d'état dans le jeu.

Gestion de la Partie :

Cycle de Vie de la Partie : Le gestionnaire de partie gère le cycle de vie complet des parties, en démarrant avec la configuration initiale et en continuant avec la gestion des interactions et des transitions entre les niveaux.

Paramétrage des Composants de Jeu : Le *GameController* établit et configure les instances des différents éléments de jeu comme les mineurs et les minerais, en s'assurant que chaque composant est correctement initialisé et prêt à être utilisé dans le jeu.

Flexibilité et Extensibilité :

Relance et transition entre Niveaux : Avec ce système, il est possible de relancer une partie ou de passer à un niveau supérieur en réinitialisant ou en reconstruisant simplement la configuration via le gestionnaire de partie. Ceci est particulièrement utile pour intégrer de nouveaux niveaux ou pour redémarrer le jeu après la fin d'une partie.

Abstraction et Maintenance : L'abstraction des fonctionnalités liées au jeu dans le gestionnaire de partie simplifie la maintenance et l'évolution du code, permettant des mises à jour ou des extensions du jeu avec une moindre complexité.

Cette architecture garantit une gestion efficace des ressources du jeu et une expérience utilisateur améliorée, avec des transitions fluides entre les niveaux et une gestion centralisée des composants du jeu.

16. Implémentation de l'action de capture des voleurs

Explications :

Cette fonctionnalité permet d'intercepter et de neutraliser les voleurs qui consomment les ressources de la carte, transformant leur capture en une source de revenus pour le joueur. Chaque capture réussie rapporte de l'argent.

Intégration avec MinerAction :

États du Mineur : Plusieurs états seront intégrés pour gérer la capture :

- *TO_ENEMY*: Permet au mineur de se déplacer vers le voleur.
- *CAPTURING*: État synchrone avec celui du voleur, initié pour démarrer la séquence de capture.
- *REMOVE_THIEF*: État de transition utilisé pour nettoyer les références au voleur avant sa suppression définitive.

Création de CatchThiefAction :

Implémentation de la logique de la méthode *execute* qui s'active à chaque cycle du thread de gestion des actions des mineurs. Le déroulement est conditionné par l'état du mineur :

- Approche du Voleur : En sélectionnant un mineur et en cliquant sur un voleur, l'action *TO_ENEMY* est déclenchée, amorçant le déplacement vers le voleur. Chaque cycle vérifie si le mineur est à portée pour initier la capture.
- Initiation de la Capture : À portée du voleur, les états du mineur et du voleur passent à *CAPTURING*. Les observateurs sont notifiés du début de la capture.
- Processus de Capture :
 - Le thread temporise les actions avec les états *CAPTURING* pour simuler les secondes.
 - La fonction *inCapture* du voleur est appelée pour décrémenter une variable de durabilité, visualisée par une barre de progression au-dessus du voleur.



Scénarios de Capture :

- Relâchement Prématuré : Si le mineur est redirigé avant la fin de la capture, il relâche sa cible et se déplace vers une autre tuile.
- Capture Réussie : Si la durabilité du voleur atteint zéro, le voleur est capturé et



renvoie une somme d'or qui est stockée et affichée via une infobulle temporaire. Les références du voleur sont nettoyées, ses cibles libérées, et il est placé dans la file d'attente pour suppression. Le mineur passe en état *IDLE* après notification de la fin de la capture et le voleur est complètement retiré du jeu.

Avantages Stratégiques :

Cette fonctionnalité enrichit l'expérience de jeu en ajoutant une couche stratégique de gestion des menaces et de récompense financière, tout en offrant des interactions dynamiques et des choix tactiques au joueur pour la gestion de ses mineurs et la protection de ses ressources.

Pseudo code :

Classe Voleur

Fonction inCapture

Si le temps restant de capture est supérieur à 0 **Alors**

 Décrémenter ce temps

Sinon

 Marquer le voleur pour sa suppression avec l'état TO_REMOVE

 Renvoyer la valeur d'argent rapporté par la capture

Fin Si

 Dans les autre cas renvoyer 0

Fin Fonction

Fin Classe

Classe CatchThiefAction **extends** MinerAction

Fonction execute

 Récupérer la distance séparant la position du mineur du voleur

Si le mineur est dans l'état TO_ENEMY et que cette distance est inférieure ou égale à la marge définie **Alors**

 Commencer la capture

 Notifier les observers du début de la capture

Sinon Si le mineur est dans l'état TO_ENEMY **Alors**

 DéplacerVers le voleur ciblé

Sinon Si le mineur est dans l'état CAPTURING **Alors**

 Appeler capture

 Notifier les observers avec la progression du temps de capture

Sinon Si le mineur est dans l'état REMOVE_THIEF **Alors**

 Notifier les observateur de la capture réussie

```

        Nettoyer les references du voleurs en les supprimant
        Passer le mineur à l'état IDLE
    Sinon
        cancelCapture
    Fin Si
Fin Fonction

Fonction capture
    Si le voleur est dans l'état CAPTURING Alors
        drop <- appeler inCapture

        Si drop est supérieur à 0 Alors
            Le voleur est capturé ajouter drop à l'inventaire
            Marquer le voleur pour retrait
        Fin Si
    Fin Si
    Sinon appeler cancelCapture
Fin Fonction

Fonction cancelCapture
    Si le voleur est dans l'état CAPTURING et que le mineur n'est ni dans l'état
CAPTURING ni SELECTED Alors
        notifier les observateurs
        faire fuir le voleur avec l'état TO_NEXT_TILE
        retirer et rendre disponible le minerai cible du voleur
    Fin Si
Fin Fonction
Fin Classe

Class MiningVoleurAction
    Fonction execute
        (...)
        Sinon Si l'état du voleur est CAPTURING Alors
            Libérer le minerais que le voleur miné
            Rendre ce même minerais disponible aux autres entités
        (...)
    Fin Fonction
Fin Class

Class TrVoleurManager
    Fonction run
        (...)
        Pour chaque voleurActions Faire
            Sinon Si le voleur est dans l'état TO_REMOVE Alors
                marquer l'action pour être supprimé Thread Safe

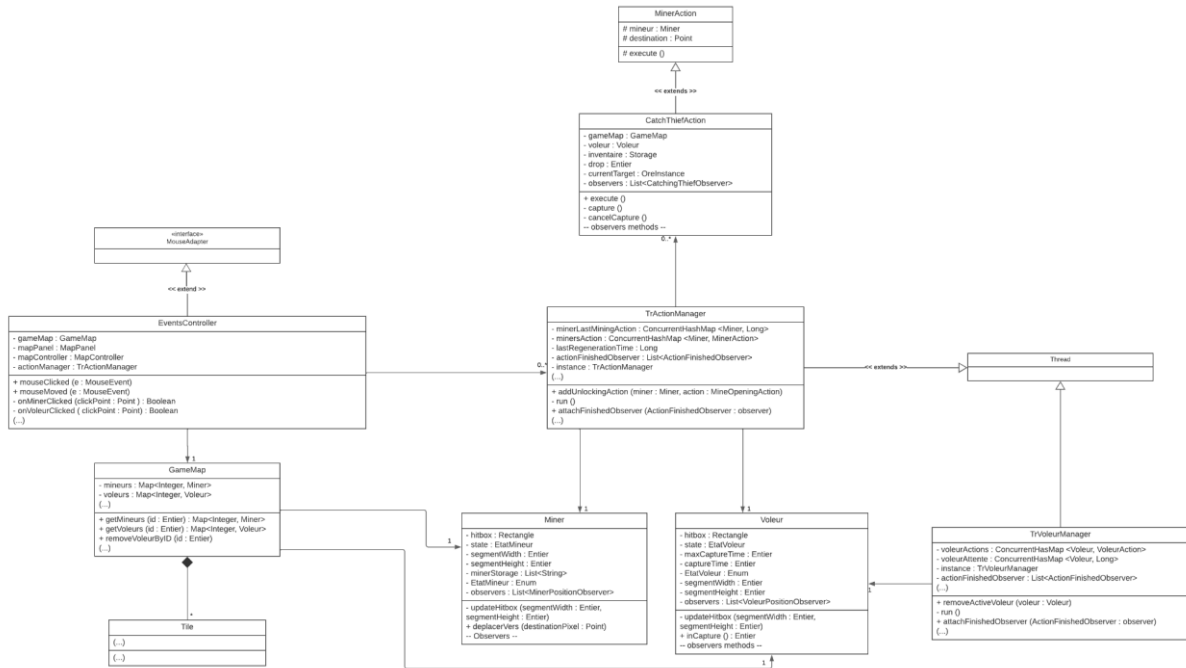
```

(...)

Fin Class

Diagramme de classe :

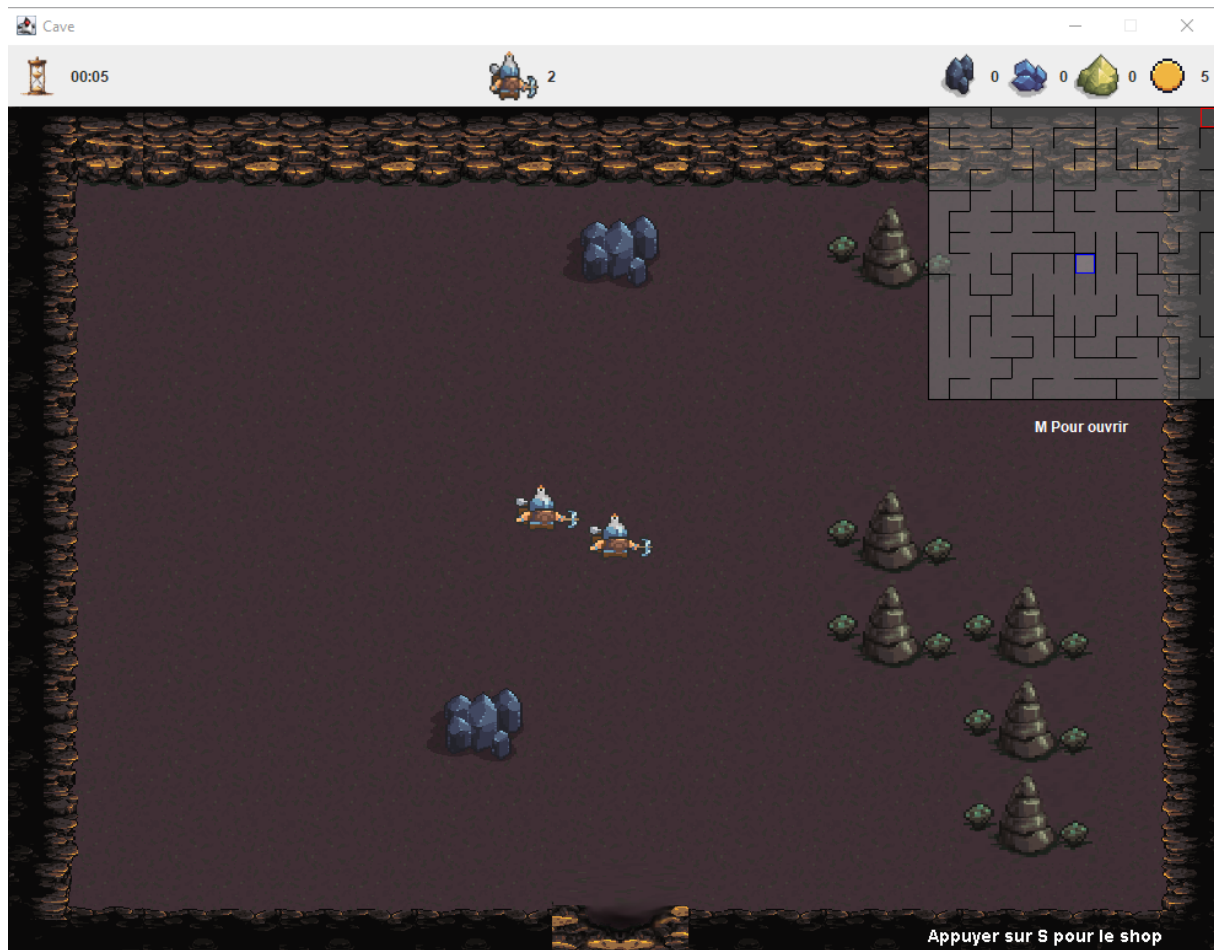
A nouveaux le modèle d'action a un schéma similaire aux autres, cependant on doit incorporer à la gestion d'états de flux d'états de voleurs pour les synchroniser.



5. Résultats

Voici ci-contre les différents aperçus du jeu :

- Aperçu Initiale lors du lancement du jeu



- Aperçu du mineur récoltant des ressources



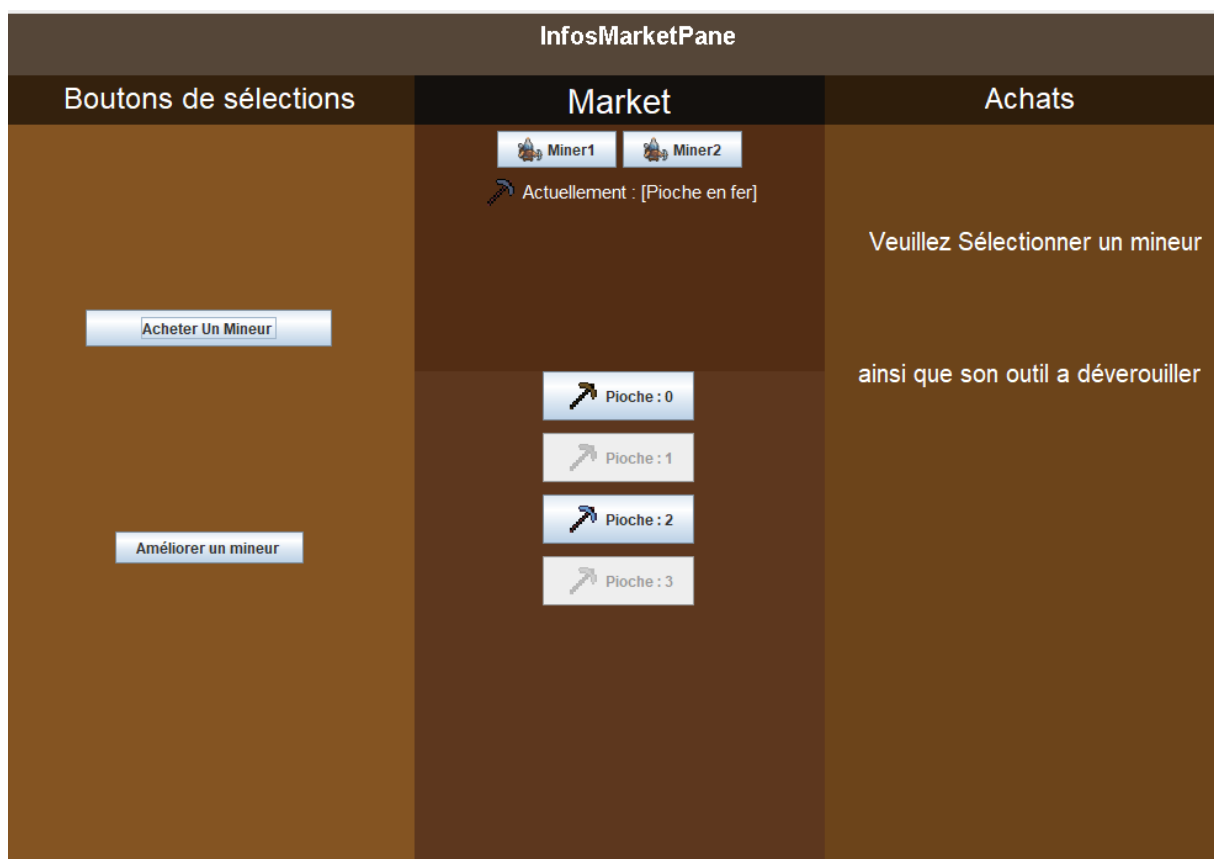
- Aperçu du mineur débloquant différentes portes



- Aperçu du voleur récoltant des ressources



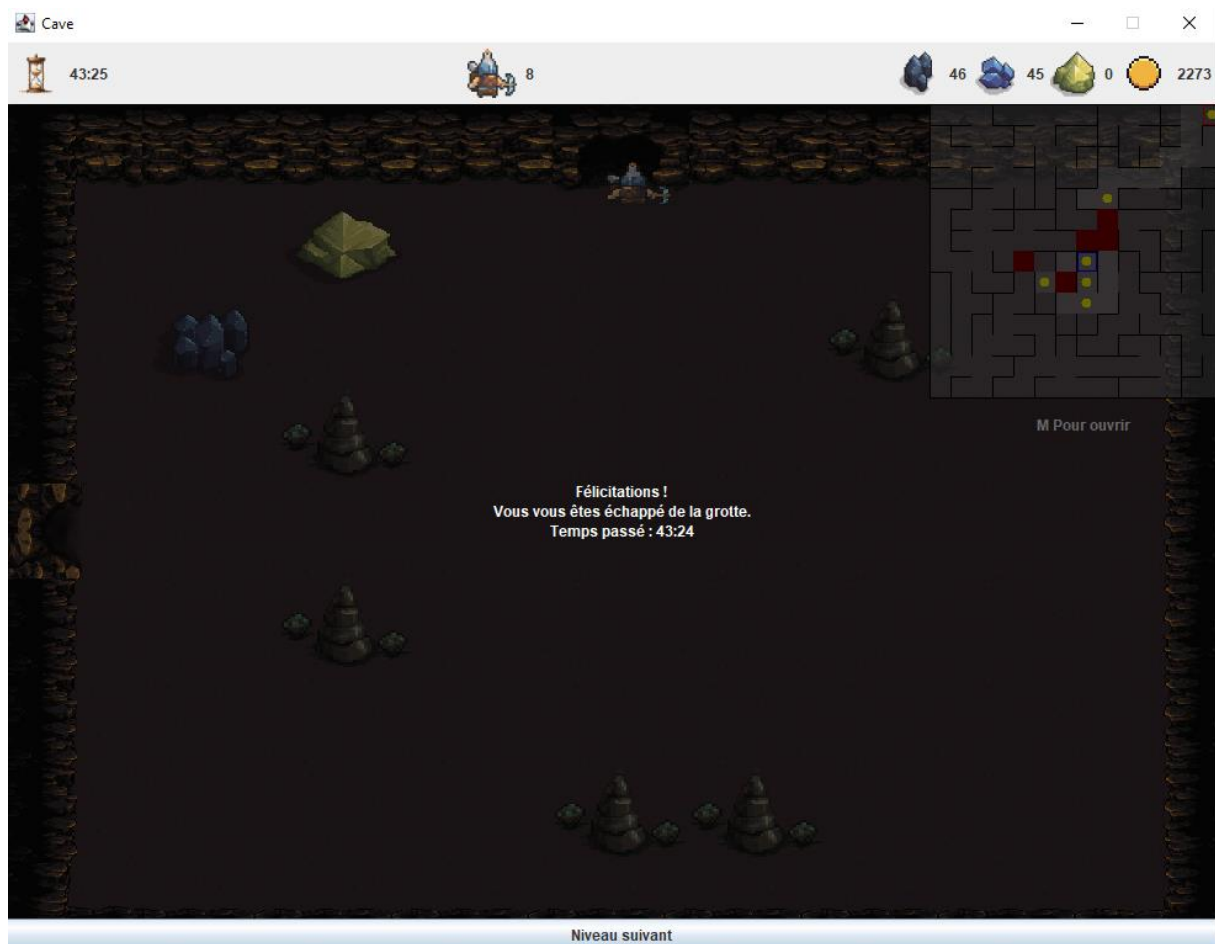
- Aperçu du Market



- Aperçu de l'interface de la mini-map qui se découvre au fur et à mesure que la partie se déroule



- Aperçu de la fin de partie après s'être échappé



6. Documentation utilisateur

Pour démarrer le jeu, il existe deux méthodes : via un IDE ou en utilisant l'exécutable .jar. Suivez les instructions ci-dessous pour la méthode de votre choix.

Procédure commune :

1. Téléchargez l'archive.
2. Décompressez l'archive dans un dossier de votre choix.

Méthode 1 : depuis l'IDE :

Eclipse :

1. Ajoutez le dossier « Cave » à votre espace de travail (Workspace).
2. Ouvrir le dossier « src ».
3. Double-Cliquez sur « App.java ».
4. Lancez le jeu avec le bouton « Run ».

VSCode :

Note : veillez à bien avoir installé les plugins et composants nécessaires à l'utilisation de Java sous VSCode.

Vous trouverez ces composants dans la section « plugin » dans l'onglet gauche de VSCode en recherchant « java ».

1. Ouvrir le dossier « Cave » dans une nouvelle fenêtre de VSCode.
2. Cliquez sur le bouton « Run java »
3. Sélectionnez « App.java »

Méthode 2 : Utilisation du fichier .jar :

1. Naviguez jusqu'au dossier « Cave » décompressé.
2. Double-cliquez sur le fichier .jar (par exemple, « Cave.jar ») pour lancer le jeu.

Note : Un JDK (Java Development Kit) doit être installé sur votre ordinateur pour exécuter le fichier .jar. Si le double-clic ne fonctionne pas, ouvrez un terminal ou une invite de commande dans le dossier où se trouve le .jar et tapez `java -jar Cave.jar`.

7. Documentation développeur

Point d'Entrée

Classe principale : La classe **App** située dans le dossier **source** sert de point d'entrée principal. Elle est responsable de la création des instances uniques qui persistent tout au long du cycle de jeu. Ces instances sont ensuite transmises au **GameController**, qui se charge de l'initialisation et de la gestion du cycle de jeu. C'est également ici que sont configurées les interactions entre les composants essentiels du jeu.

Paramètres du Jeu

Configuration : Les paramètres de configuration du jeu sont centralisés dans la classe **Constants**. Cette classe organise les paramètres par catégories, permettant ainsi leur modification pour ajuster divers aspects du jeu comme la taille de la carte, la dimension des tuiles, les ressources graphiques, etc.

Guide d'Ajout de Fonctionnalités

Ajout d'Actions

Pour intégrer de nouvelles actions aux entités existantes, telles que les voleurs ou les mineurs, suivez ces étapes :

- Création d'une Action : Développez un modèle d'action dans le package **model > characters** qui hérite de l'entité concernée.
- Implémentation des États et Actions : Implémentez les états internes de l'entité et la méthode **execute** de l'action.
- Intégration avec ActionManager : Ajoutez une méthode **addNouvelleAction** à l'ActionManager de l'entité pour que le thread puisse exécuter cette action.

Gestion des Événements

Pour ajouter des événements (clicks, pressions, etc.), modifiez le **EventController** dans le package **Controller**. Ce contrôleur centralise tous les événements de l'application. Il est facile d'y ajouter de nouveaux événements en utilisant les observateurs existants ou en créant de nouveaux accesseurs dans les vues associées.

Passage de Données

L'application utilise un modèle d'observateurs pour passer des données. Ajoutez des interfaces d'observateurs dans le dossier **src > model > observers**. Ce modèle permet une implémentation généralisable et simplifiée adaptée à des applications de petite taille. Pour optimiser les performances, utilisez des Data Transfer Objects (DTO) pour notifier les vues avec uniquement les informations nécessaires, configurables dans **src > model > DTO**.

Structure de la Vue

- **MainFrame** : Fenêtre principale qui englobe tous les composants de l'interface utilisateur.
- **Northbar** : Barre d'état située au nord de la MainFrame, affichant les informations du joueur.

- **MiniMapPanel** : Affiche la mini-carte.
- **MapPanel** : Gère l'affichage de toutes les informations de jeu, y compris les éléments graphiques et les événements.
- **Market** : Interface utilisateur du magasin du jeu, construite avec des éléments Swing. Modifiez cette partie pour ajuster les aspects économiques du jeu.

Gestion des Ressources Graphiques

Les ressources graphiques sont organisées dans le dossier **src > assets**. Pour charger ces ressources, utilisez la classe **ImageLoader** dans **src > utils**, qui implémente le préchargement et la gestion des ressources. Une instance de cette classe est créée une seule fois lors de l'initialisation de **MapPanel** et est accessible sans affecter les performances.

Événements Temporels

Pour les actions nécessitant une synchronisation temporelle, utilisez le thread **TrGameTimer** dans **src > controller**. Ce timer fonctionne comme une horloge interne pour le jeu, gérant les délais et les intervalles nécessaires à la logique du jeu.

Fonctionnalités non implémentées :

Certaines fonctionnalités n'ont pas pu être implémenter voici une sélection que nous jugeons pertinente afin d'améliorer sensiblement l'expérience des joueurs :

Fonctions Hold & Drag

Pour améliorer la gestion de plusieurs mineurs, l'introduction d'un événement hold & drag dans **EventController** pourrait permettre de sélectionner plusieurs mineurs simultanément. En maintenant et en déplaçant la souris, un rectangle de sélection pourrait être dessiné pour englober les hitbox des mineurs concernés, remplissant ainsi une liste de mineurs sélectionnés pour des actions groupées.

Niveau Suivant

L'ajout de nouveaux niveaux pourrait se faire en augmentant le nombre d'assets de carte et en enrichissant les paliers d'objets. En franchissant une porte en or, une variable de niveau serait ajustée pour charger de nouveaux assets et créer une nouvelle carte, tout en conservant les progrès du joueur (mineurs et équipements).

Implémentation de Salles Spéciales

Pour diversifier l'exploration, une classe mère Salle pourrait être créée avec des classes dérivées définissant différents types de salles (coffres, pièges, échange, etc.). Ces salles pourraient être générées de manière semi-aléatoire et placées dans des zones à difficulté appropriée lors de la création de la carte.

Varier le Nombre d'Obstacles et Layout des Salles

Pour dynamiser l'environnement de jeu, l'ajout de nouveaux obstacles et un placement varié à l'aide de l'algorithme de Kruskal directement sur les tuiles pourraient enrichir le design des salles. Ces obstacles pourraient prendre la forme de ravins, lacs souterrains, etc.

Améliorer la Mise en Cache

Au lieu de charger manuellement chaque asset dans ImageLoader, il serait plus efficace de charger dynamiquement le répertoire assets et ses sous-répertoires. Cela simplifierait l'utilisation de ImageLoader en automatisant la gestion des ressources graphiques.

Ajouter des Statistiques aux Mineurs et Voleurs

Pour approfondir les stratégies de jeu, l'introduction de statistiques pour les entités et les objets serait bénéfique. En ajoutant des attributs tels que la vitesse de minage ou de déplacement, le magasin pourrait offrir une variété d'options aux performances et coûts divers, permettant aux joueurs de personnaliser davantage leurs stratégies. Cette implémentation pourrait être réalisée en appliquant les effets des objets contenus dans l'inventaire des entités directement sur leurs attributs correspondants.

8. Conclusion et perspectives

Au cours de ce projet, nous avons adopté le modèle MVC (Modèle-Vue-Contrôleur), ce qui a renforcé notre compréhension de cette structure architecturale cruciale. Cela nous a permis de développer une application bien organisée, où la logique métier, l'interface utilisateur et les interactions sont distinctement séparées, facilitant ainsi la maintenance et l'extension future du code.

Nous avons rencontré des défis majeurs, notamment la gestion des interactions complexes entre les divers composants de l'application. Pour résoudre ces problèmes, nous avons utilisé plusieurs motifs de conception, notamment le motif Observateur pour les mises à jour dynamiques et le motif Commande pour la gestion des actions. L'implémentation de ces motifs a non seulement amélioré la communication entre les composants mais a aussi rendu notre application plus adaptable.

Ce projet a été une excellente opportunité de mettre en pratique nos connaissances en design patterns et architecture logicielle, en particulier l'utilisation du modèle MVC. Nous avons vu l'importance de la planification initiale dans la conception de nos systèmes, comme en témoigne l'introduction d'abstractions telles que les ActionManagers, qui ont facilité l'ajout de nouvelles fonctionnalités au fil du temps.

Un autre apprentissage clé a été l'importance de l'anticipation et d'une communication efficace au sein de l'équipe, ce qui a amélioré notre coordination, surtout lorsque nos tâches devaient s'intégrer pour former un tout cohérent. Cette méthode a non seulement optimisé notre efficacité mais a également amélioré notre compréhension des différents aspects du projet.

Travailler en équipe sur ce projet a renforcé nos compétences en gestion de projet et collaboration. L'utilisation de Git a facilité la coordination et maintenu un rythme de développement constant, nous permettant d'adapter notre produit aux besoins changeants.

Cette expérience a solidement préparé notre croissance en tant que développeur de logiciels. Professionnellement, la maîtrise du modèle MVC et des motifs de conception sera cruciale pour gérer des projets plus complexes à l'avenir. Ces compétences sont essentielles pour créer des architectures logicielles robustes et évolutives, et seront un atout précieux dans notre carrière.

Ce projet a renforcé notre capacité à collaborer efficacement et à gérer le stress lié aux délais et exigences évolutives. Les défis rencontrés et les solutions développées ont souligné l'importance de l'adaptabilité et de l'innovation continue, enrichissant nos compétences de développement.

En conclusion, ce projet a été une exploration enrichissante des principes avancés de développement logiciel, confirmant l'efficacité des modèles de conception dans la création d'applications évolutives et maintenables. Nous sommes convaincus que cette expérience sera un véritable atout pour nos futures carrières.