Activité intégrative (B1)

Checklists non-officielles

A faire tout au long des itérations. Ceci ne correspondent aux choses demandées dans le PDF de consignes des itérations et sont faites pour avancer au fur et à mesure dans les itérations. Contrairement aux checklists officielles qui sont faites pour s'auto-évaler à la fin.

Checklist itération 1 (non-officielle)

> **Attention** cette checklist n'est pas officielle, elle est seulement basée sur le PDF de l'itération 1 et général. Pour la checklist officielle cliquez ici

Contraintes générales

☑ La structure des fichiers corresponds à celle montrée dans le PDF

☑ Le projet utilise la version 11 de Java et intègre l'interface graphique Swing

☑ Le projet est compatible avec la version d'Eclipse de référence (voir ressources informatiques sur Learn)

☑ Les tests unitaires sont effectués à l'aide de JUnit5

☑ Le projet utilise le plug-in PMD avec le ruleset donné sur Learn

Acquis d'apprentissages

☑ Respect des principes de la programmation orientée objet

☑ Les comportements du programme sont testé à l'aide de tests unitaires

☑ Le programme est bien documenté à l'aide de la JavaDoc (et calcul de la CTT si demandée)

☑ Les utilisations des interfaces de collections et de ses implémentations sont justifiées

Fonctionalités

AI-1 : Créer une partie

En tant que joueur, je souhaite démarrer une nouvelle partie afin de l'afficher à l'écran

☑ L'application démarre en affichant un menu principal composé de deux items : « Nouvelle partie » et « Quitter »

☑ L'activation de l'item « Nouvelle partie » débute la création d'une partie.

☑ La création d'une partie commence par le chargement de la carte depuis le fichier resources/maps/map-sample.txt. Nous vous fournissons à cette fin la classe treasurequest.io.CharArrayFileReader.

☑ Une fois la carte chargée, l'application y place des trésors sur les cases creusables. Une case est creusable si elle n'est pas de type Eau.

☑ Le nombre de trésors à placer correspond à 10% du nombre de cases creusables, avec au moins 1 trésor à trouver.

☑ La valeur d'un trésor est un nombre aléatoire allant de 10 à 20 pièces.

☑ Toutes les cases sont non-creusées.

☑ Une case active est désignée. Elle correspond à une case du centre de la carte.

☑ Le joueur possède une bourse correspondant à 2 fois le nombre de trésors. Cette bourse est exprimée en pièces (s'il y 20 trésors à placer, la bourse de départ du joueur sera de 40 pièces).

Tests d'intégration

☑ Quand je lance l'application, celle-ci me présente le menu principal composé de deux items « Nouvelle partie » et « Quitter »

☑ Quand je sélectionne l'item « Quitter », l'application termine son exécution.

☑ Quand je sélectionne l'item « Nouvelle partie », l'application change d'écran

AI-2 : Voir une partie

En tant que joueur, je souhaite voir l'état d'une partie afin de décider des actions à entreprendre

☑ Une seconde vue présente l'état de la partie.

☑ Une carte montre les cases. En début de partie, aucune case n'est creusée.

☑ La case active de départ est au « centre » de la carte.

☑ Un panneau affiche la bourse du joueur, le nombre de trésors à trouver et le cout à payer pour creuser la case active.

☑ Creuser une case Sable coute 1 pièce. Le cout des autres types de cases est déduit de cette valeur.

Tests d’acceptation

Important : pour les tests d’acceptation, nous utiliserons des cartes autres que celles fournies.

☑ Quand je lance une nouvelle partie, l’application me présente la carte donnée dans l’exemple.

☑ Quand je lance une nouvelle partie, la case active est celle au « centre » de la carte

☑ Quand je lance une nouvelle partie, l’application me dit qu’il y a XX trésors à trouver, que le joueur à une bourse de 2*XX

☑ Quand je lance une nouvelle partie, l’application m’affiche le cout correct de la case active

AI-3 : Revenir au menu principal

En tant que joueur, je peux abandonner une partie et revenir au menu principal, afin d’en relancer une nouvelle.

☑ Appuyer sur la touche « Esc » quand une partie est en cours permet de revenir à l’écran principal.

☑ Demander une nouvelle partie relance la création d’une nouvelle partie. En particulier, le rechargement de la carte.

Tests d’acceptation

☑ Soit une partie lancée avec une case creusée, quand je reviens au menu principal et que je relance une nouvelle partie, le jeu revient à son état de départ.

☑ Soit une partie lancée, quand je change le fichier à charger et que je reviens au menu principal, alors l’application présente la carte du nouveau fichier

Phase de conception et problèmes à résoudre

Diagramme de conception générale

☑ Les Candidats et les Responsabilités ont été identifiées

☑ Les cartes CRC et les liens entre elles ont été faites

☑ Le diagramme de séquence/collaboration a été créé

☑ Le diagramme de classe a été fait

La classe de fabrique (Factory)

☑ Une classe de fabrique (Factory) est créée dans le package `domains` comme les autres classes et sert à synchroniser le jeu sur les différents superviseurs

☑ La classe de fabrique est passée aux superviseurs au travers d'une interface

☑ Les superviseurs se synchronise avec la classe de fabrique dans les méthodes `onEnter` et `onLeave`

☑ La classe de fabrique est insanciée dans `Program` et est passée au constructeur des superviseurs concernés

La représentation de la carte de cases

☑ Pas de tableau 2D pour preprésenter l'association des coordonnées aux cases et choisir la collection appropriée pour mémoriser la carte et les cases

Questions supplémentaires d'algo et de POO

Questions algorithmiques

☑ Ecrire dans l'entête de la méthode `PlayGameSupervisor.onEnter` les post-conditions les classes `Player`, `CaseMap` et les `Case` de la carte doivent respecter après la création de la partie pour pouvoir commencer à creuser/jouer.

☑ Indiquer dans l'entête JavaDoc de la classe `CaseMap` l'interface de collection a été utliisée pour représenter la carte et justifier son choix en expliquant les différentes opérations à y effectuer

☑ Indiquer dans cette même JavaDoc quelle implémentation de la collection a été utilisée pour représenter la carte et justifier son choix en déterminant la CTT des principales opérations utilisées dans l'itération 1

☑ **Indiquer la CTT du placement des trésors sur les cases creusables dans la javadoc en entête de TreasureQuestGameFactory** (Pour répondre à cette question, examinez la partie de votre code concernée, et identifiez les boucles, les imbrications, mais aussi les collections utilisées et leurs opérations ou les appels de sous-méthodes ou de méthodes d'autres objets. Quand vous répondez, n'oubliez pas d'indiquer à quoi correspondent vos libellés N,M, etc)

☑ Les Superviseurs collabore avec la fabrique (Factory) de Game au travers d'une interface

Questions POO

☑ Les classes sont dans `treasurequest.domains` et n'ont pas de lien avec les vues (tout ce qui pourrait être en dehors de `domains`)

☑ L'encapsulation (attributs private) et les copies défencives sont bien effectuées sur toutes les classes du domains

☑ Ecrire des méthodes courtes et peu complexes (et tenter de respecter la règle de Demeter)

☑ Ecrire des classes timides (c'est à dire qui font le boulot et qui ne nécessite pas de tout le temps leur demander des choses depuis d'autres classes)

☑ Bon usage du polymorphisme

☑ Bonne utilisation des interfaces

☑ Aucune alerte PMD

☑ Tests unitaires JUnit5 dans le dossier `tests` qui donne un coverage d'au moins 90% sur chaque classe du domains (pour un degré de maitrise supplémentaire → coverage de 100% de ces classes)

Checklist itération 2 (non-officielle)

> **Attention** cette checklist n'est pas officielle, elle est seulement basée sur le PDF de l'itération 2 et général. Pour la checklist officielle cliquez ici

Contraintes générales

☑ La structure des fichiers corresponds à celle montrée dans le PDF

☑ Le projet utilise la version 11 de Java et intègre l'interface graphique Swing

☑ Le projet est compatible avec la version d'Eclipse de référence (voir ressources informatiques sur Learn)

☑ Les tests unitaires sont effectués à l'aide de JUnit5

☑ Le projet utilise le plug-in PMD avec le ruleset donné sur Learn

Acquis d'apprentissages

☑ Respect des principes de la programmation orientée objet

☑ Les comportements du programme sont testé à l'aide de tests unitaires

☑ Le programme est bien documenté à l'aide de la JavaDoc (et calcul de la CTT si demandée)

☑ Les utilisations des interfaces de collections et de ses implémentations sont justifiées

Réponse feedback

☑ Faire une copie défensive de `getCoordinates()` (par exemple à l'aide de `Collections.unmodifiableSet(Set)`)

☑ Passer la case active dans Game plus tot que Player

☑ Potentiellement faire passer les messages de statistiques sur le jeu dans Game plus tot que superviseur

☑ Rendre toutes les classes `final` pour empécher d'en hériter

☑ Supprimer toutes les définitions de `equals` et `hashcode` quand ce n'est pas nécessaire (par exemple Player)

☑ Faire en sorte que MainMenuSuperviser crée le jeu et que PlayGame le récupère

☑ Rendre la classe Coordinate plus intelligente pour Player.move(deltaX, deltaY)

☑ Expliquer le `+` dans le calcul de la CTT du placement des trésors dans CaseMap

☑ Faire en sorte que la factory crée la map et l'injecte dans le jeu

Fonctionalités

AI-4 : Changer de case active

En tant que Joueur, je souhaite changer de case active pour poursuivre mon exploration.

☑ Appuyer une touche fléchée déplace la case active.

☑ Changer de case active met à jour le type et le cout de la case active affichée.

☑ A priori, le joueur peut se déplacer hors des limites de la carte. On peut voir une case hors des limites d'une carte comme une case de type Eau

Tests d'acceptation

Les tests d'acceptation partent de la Carte 1. On suppose que la case active de départ est celle du centre (encadrée en rouge). Attention, nous pouvons jouer les tests d'acceptation sur d'autres cartes.

☑ Etant donnée une partie démarrée avec la carte donnée en exemple, quand j'appuie sur la flèche du haut, alors le sélecteur se déplace sur la case (0,1) de type Sable et de cout 1.

☑ Etant donnée une partie démarrée avec la carte donnée en exemple et la case active placée sur la case (0,1), quand j'appuie sur la flèche du bas puis sur la flèche de droite, alors le sélecteur est sur la case (1,2) de type Prairie et de cout 2.

☑ Etant donnée une partie démarrée avec la carte donnée en exemple et la case active placée sur la case (1,2), quand j'appuie sur la flèche de bas, alors le sélecteur se déplace sur

la case (2, 2) de type Eau et de cout 0.

AI-5 : Creuser une case

En tant que joueur, je souhaite creuser la case active afin de trouver des trésors

☑ Si cette case est de type Eau, rien ne se passe.

☑ Si cette case est de type autre que Eau et sans trésor, l'application affiche le sprite « Dug ».

☑ Si cette case possède un trésor, l'application affiche le sprite « Treasure ».

☑ Si cette case ne possède pas de trésor, la bourse du joueur est diminuée du cout de la case.

☑ Si cette case possède un trésor, la bourse du joueur est diminuée du cout de la case et augmentée de la valeur du trésor découvert.

☑ Si cette case possède un trésor, le nombre de trésor à trouver est diminué de 1.

☑ Si le joueur tente de creuser alors qu'il est en dehors de la carte, rien ne se passe : en particulier il n'y a pas de messages d'erreur remontés par l'application.

Tests d'acceptation

Les tests d'acceptation partent de la Carte 1. On suppose que le seul trésor se trouve en case (1,2). Attention, nous pouvons jouer les tests d'acceptation sur d'autres cartes.

☑ Soit une partie démarrée dont la case active est (0,0), quand j'essaie de la creuser, rien ne se passe.

☑ Soit une partie démarrée dont la case active est (0,1), quand je la creuse, alors l'application retire 1 pièce de ma bourse.

☑ Soit une partie démarrée dont la case active est (0,1) et est déjà creusée, quand je la creuse, rien ne se passe.

☑ Soit une partie démarrée dont la case active est (1,2), quand je la creuse, alors ma bourse perd 2 pièces et reçoit autant de pièces qu'en compte le trésor.

☑ Soit une partie démarrée dont la case active est (1, 0), quand je la creuse, alors rien se passe.

AI-6 : Fournir des indices

En tant que Joueur, quand je creuse une case, je souhaite avoir un indice sur le trésor le plus proche afin de trouver des trésors

Une case peut contenir un indice sur le trésor le plus proche prenant la forme d'une flèche.

☑ La flèche d’une case indique la direction du trésor le plus proche.

☑ Un trésor peut être le plus proche de toutes les cases situées dans un carré de 5 cases de côté et centré sur le trésor.

☑ Il y a 8 directions possibles : O, NO, N, NE, E, SE, S, SO.

☑ Seules les cases creusables sans trésor peuvent contenir un indice.

☑ Quand une case C se trouve à égale distance de deux trésors, l’application choisit celui de plus grande valeur. Quand une case C se trouve à égale distance de deux trésors de même valeur, on privilégie celui dont la position est la plus proche du coin supérieur gauche.

☑ La distance entre deux cases c1 = (row1 ; col1) et c2 = (row2 ; col2) vaut $\sqrt{(row1 - row2)^2 + (col1-col2)^2}$

☑ Les indices ne sont pas modifiés en cours de partie, ce qui implique qu’ils peuvent être tous calculés au début de la partie.

Tests d’acceptation

Nous partons de la Carte 2 pour décrire les tests d’acceptation. Les trésors y sont prédisposés avec les valeurs entre parenthèses. Attention, nous pouvons jouer les tests d’acceptation sur d’autres cartes.

☑ Étant donnée une partie démarrée avec la Carte 2, quand je creuse la case (4, 4), je découvre une flèche orientée vers le bas.

☑ Étant donnée une partie démarrée avec la Carte 2, quand je creuse la case (3, 5), je découvre une flèche orientée vers le Sud-Ouest.

☑ Étant donnée une partie démarrée avec la Carte 2, quand je creuse la case (3, 1), je découvre une flèche orientée vers le Nord-Ouest.

☑ Étant donnée une partie démarrée avec la Carte 2, quand je creuse la case (8, 0), je ne découvre ni indice, ni trésor.

Phase de conception et problèmes à résoudre

Diagramme de conception générale

☑ Les Candidats et les Responsabilités ont été identifiées

☑ Les cartes CRC et les liens entre elles ont été faites

☑ Le diagramme de séquence/collaboration a été créé

☑ Le diagramme de classe a été fait

Controller le placement des trésors pour les tests

☑ Déclarer une interface pour remplacer `Math.random`, `Random` et/ou `Collections.shuffle()`

☑ Implémenter cette interface en faisant appel à la/les méthodes ci-dessus

☑ Remplacer les dépendances à la méthode de base dans le code par l'interface

☑ Injecter la dépendence (ajouter l'objet implémentant l'interface dans le constructeur des classes qui ont besoin d'aléatoire)

☑ Implémenter cette interface pour les tests (elle peut par exemple retourner une collection de coordonnées fournies lors de la construction de l'objet)

☑ Utiliser cette implémnetation pour les tests

Gestion des indices

> **TODO**

Questions supplémentaires d'algo et de POO

Questions algorithmiques

☑ Ecrire les post-conditions qui seront vérifiées par le montant ajouté à la bourse du joueur (gain minimum d'un creusage et gain maximum d'un creusage) dans la javadoc de `PlayGameSuperviser.onDig()`

☑ Indiquer la CTT du calcul des indices à affecter aux cases creusables dans la javadoc en entête de TreasureQuestGame (Pour répondre à cette question, examinez la partie de votre code concernée, et identifiez les boucles, les imbrications, mais aussi les collections utilisées et leurs opérations ou les appels de sous-méthodes ou de méthodes d'autres objets. Quand vous répondez, n'oubliez pas d'indiquer à quoi correspondent vos libellés N,M, etc)

☑ Indiquer dans l'entête de la classe `TreasureQuestGame` quelle interface de collection a été utilisée pour représenter les coordonnées environantes (Justifiez votre choix en identifiant les principales opérations dont vous aurez besoin au cours de cette itération ? Avez-vous notamment besoin d'accéder à un élément précis ? Si oui, sur base de quelle sorte de clé ?)

☑ Indiquer dans l'entête de la classe `TreasureQuestGame` quelle implémentation de collection a été utilisée pour représenter ces coordonnées (justifier votre choix en déterminant les CTT des principales opérations pour l'itération 2)

Questions POO

☑ Les classes sont dans `treasurequest.domains` et n'ont pas de lien avec les vues (tout ce qui pourrait être en dehors de `domains`)

☑ L'encapsulation (attributs private) et les copies défencives sont bien effectuées sur toutes les classes du domains

☑ Ecrire des méthodes courtes et peu complexes (et tenter de respecter la règle de Demeter)

☑ Ecrire des classes timides (c'est à dire qui font le boulot et qui ne nécessite pas de tout le temps leur demander des choses depuis d'autres classes)

☑ Bon usage du polymorphisme

☑ Bonne utilisation des interfaces

☑ Aucune alerte PMD

☑ Tests unitaires JUnit5 dans le dossier `tests` qui donne un coverage d'au moins 90% sur chaque classe du domains (pour un degré de maitrise supplémentaire → coverage de 100% de ces classes)

☐ (degré de maitrise supplémentaire) Un plan de test exhaustif valide le code correspondant à la CTT demandée du placement des trésors et du calcul des indices

Checklist itération 3 (non-officielle)

> **Attention** cette checklist n'est pas officielle, elle est seulement basée sur le PDF de l'itération 3 et général. Pour la checklist officielle cliquez ici

Contraintes générales

☑ La structure des fichiers corresponds à celle montrée dans le PDF

☑ Le projet utilise la version 11 de Java et intègre l'interface graphique Swing

☑ Le projet est compatible avec la version d'Eclipse de référence (voir ressources informatiques sur Learn)

☑ Les tests unitaires sont effectués à l'aide de JUnit5

☑ Le projet utilise le plug-in PMD avec le ruleset donné sur Learn

☑ Ajouter la `big-map.txt` dans le projet

☑ Télécharge l’archive `ai2023.results.zip` et décompresse son contenu dans le dossier `resources/images/results`

☑ Dans l’enum `treasurequest.supervisors.views.ResultType`, renomme la valeur `BALANCE` en `DURATION`

☑ Édite ensuite la constante `RESULT_TYPE` de la classe `treasurequest.swing.Theme` pour y ajouter les entrées correspondant aux statistiques comme ci-dessous.

```
/**
* Définit les images associées aux différents résultats.
* */
public static final Map<ResultType, Image> RESULTS_SPRITES = Map.of(
ResultType.NONE, new ImageIcon("resources/images/results/none.png").getImage(),
ResultType.LOSS, new ImageIcon("resources/images/results/loss.png").getImage(),
ResultType.GAIN, new ImageIcon("resources/images/results/gain.png").getImage(),
```

```
ResultType.DURATION, new ImageIcon("resources/images/results/duration.png").getImage(),
ResultType.TOURIST, new ImageIcon("resources/images/results/tourist.png").getImage(),
ResultType.FARMER, new ImageIcon("resources/images/results/farmer.png").getImage(),
ResultType.LUMBERJACK, new ImageIcon("resources/images/results/lumberjack.png").getImage(),
ResultType.MINER, new ImageIcon("resources/images/results/miner.png").getImage()
);
```

☑ Un bug empêche l'affichage de la bonne image. Pour le résoudre, modifie la ligne 79 de la classe `treasurequest.swing.GameOverSwingView` comme ci-dessous.

```
panels.add(new GameResultPanel(type, message, getWidth()/100*3 + col*(256+5), getHeight()/3 +
row*(200+5)));
```

Acquis d'apprentissages

☑ Respect des principes de la programmation orientée objet

☑ Les comportements du programme sont testé à l'aide de tests unitaires

☑ Le programme est bien documenté à l'aide de la JavaDoc (et calcul de la CTT si demandée)

☑ Les utilisations des interfaces de collections et de ses implémentations sont justifiées

Réponse feedback

☑ Coriger la valeur de M dans le calcul de la CTT

☑ Rendre Case en tant que non-data class

☑ Vérifier les références de Coord dans la méthode de fabrique de HintOrientation

☑ Complèter les préconditions et postconditions de toutes les méthodes publiques

☐ Changer les conditions de game over pour les cases creusable et leur coût

Fonctionalités

AI-7 Finir une partie

En tant que joueur, je souhaite savoir quand la partie se termine, afin de connaître mes résultats.

☑ La partie se termine si le joueur a découvert tous les trésors, s'il fait banqueroute ou s'il décide d'abandonner (voir itération 1).

☑ Le joueur fait banqueroute quand il n'a plus assez de pièces pour creuser les cases restantes. Autrement dit, la partie se termine quand il n'existe plus de cases pouvant être creusées par le joueur.

☑ Quand la partie est terminée, l'application redirige le joueur vers l'écran de fin de partie. Cependant, en cas d'abandon, l'application redirige le joueur vers le menu principal

Tests d'intégration

Les tests d'acceptation partent de la Carte 1 (voir ci dessous). On suppose également que le seul trésor à trouver est sur la case P, de type « Prairie ». Attention, nous pouvons jouer les tests d'acceptation sur d'autres cartes.

```
E S E
F R P
E S E
```

☑ Étant donné une partie débutée avec la Carte 1, quand je creuse la case de coordonnées (1,2), alors l'application affiche l'écran de fin de partie.

☑ Étant donné une partie débutée avec la Carte 1, quand je creuse les cases de coordonnées (0,1) et (2, 1), alors l'application affiche l'écran de fin de partie.

☑ Étant donné une partie débutée avec la Carte 1, quand j'appuie sur la touche « Esc », alors l'application affiche le menu principal

AI-8 Afficher les statistiques simples

En tant que joueur, je souhaite connaître les statistiques sur mes parties, afin de savoir si j'ai gagné ou perdu de l'argent.

☑ On souhaite connaître les dépenses que le joueur a faites. Ces dépenses correspondent à la somme des pièces dépensées par le joueur pour creuser les cases.

☑ On souhaite connaître les gains que le joueur a faits. Les gains correspondent à la somme des pièces que le joueur a reçues pendant ses parties.

☑ On souhaite connaître le temps de jeu du joueur. Le temps de jeu sera exprimé au format MM:SS, comme vous avez déjà eu l'occasion de le faire pendant les cours théoriques de POO.

☑ La durée d'une partie abandonnée reste comptabilisée dans le temps de jeu.

☑ On souhaite connaître le profil du joueur. Le profil correspond au type de la plus grande zone de cases adjacentes de même type que le joueur a creusées pendant une partie. Les

profils sont sable → touriste, prairie → fermier, forêt → bucheron, rocher → mineur.

☑ Quand deux zones candidates sont de même taille, l'application privilégie la plus ancienne, c'est-à-dire celle des deux qui a été commencée en premier.

☑ Deux cases sont contigües si leur distance est inférieure à 2 afin de tenir compte des cases diagonales.

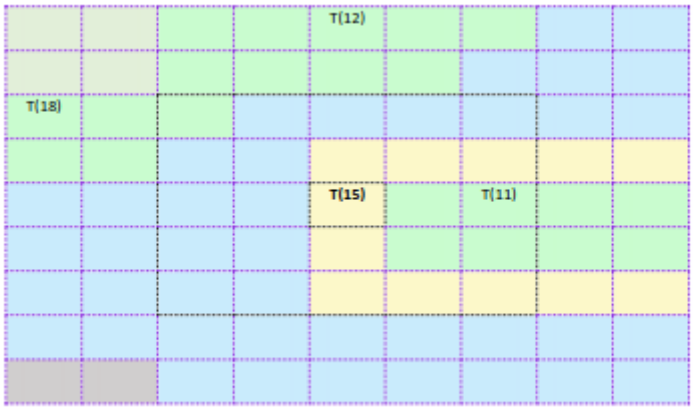
☑ Un document annexe présente un algorithme pour trouver la plus grande zone creusée.

☑ Ces statistiques perdurent d'une partie à l'autre.

☑ Les gains et les pertes en cas d'abandon restent comptabilisés. Cependant, le profil ne sera pas adapté.

Tests d'acceptation

Nous partons de la Carte 2 pour décrire les tests d'acceptation. On fait l'hypothèse que les trésors sont cachés dans les cases indiquées. Attention, nous pouvons jouer les tests d'acceptation sur d'autres cartes.



Carte 2 carte composée de plusieurs trésors

☑ Soit une nouvelle partie débutée avec la Carte 2 où le joueur a reçu 8 pièces, si je creuse les cases de (ligne : 5, colonne : 4), (5, 5), (5,6), (5,7) et (6,7) alors la partie se termine, j'ai dépensé 8 pièces, j'ai gagné 8 pièces (celles reçues en début de partie) et je suis fermier.

☑ Soit une nouvelle partie débutée avec la Carte 2 où le joueur a reçu 8 pièces, si je creuse les cases (ligne : 3, colonne : 4), (4, 4), (5, 4), (5, 5), (5,6), (4,6), (2,0) et (0,4), alors la partie se termine, j'ai dépensé 13 pièces, j'ai gagné 64 pièces (56 pièces + 8 reçues en début de partie) et je suis touriste.

☑ Soit une seconde partie débutée après la partie précédente, alors la bourse du joueur vaut 59 pièces (51 pièces encore disponibles et 8 pièces reçues en début de partie).

☑ Soit une nouvelle partie débutée avec la Carte 2, quand la partie se termine, le temps de jeu affiché par l'application respecte le format demandé.

☑ Soit une seconde partie débutée avec la Carte 2, quand la partie se termine, alors le temps de jeu affiché est plus grand que le temps de jeu affiché précédemment

AI-9 Jouer une partie sur une carte pseudo-aléatoire

☑ Un joueur peut lancer une partie qui se déroule sur un terrain aléatoire à l'aide d'un nouvel item « Partie aléatoire » affiché par le menu principal

☑ Le terrain correspond à une sous-zone carrée de 16 cases de côté extraite du fichier `resources/maps/big-map.txt` (voir les préalables).

☑ Cette sous-zone sera tirée aléatoirement

Tests d'acceptation

Quand je démarre une partie aléatoire, alors l'application m'affiche une carte carrée de 16 cases de côté.

☑ Quand je démarre une nouvelle partie classique, que je l'abandonne et que je démarre une partie aléatoire, alors l'application m'affiche une carte carrée de 16 cases de côté.

☑ Quand je démarre une partie aléatoire, que je l'abandonne et que je démarre une seconde partie aléatoire, alors l'application m'affiche une carte carrée de 16 cases de côté différente de la précédente

Phase de conception et problèmes à résoudre

Diagramme de conception générale

☑ Les Candidats et les Responsabilités ont été identifiées

☑ Les cartes CRC et les liens entre elles ont été faites

☑ Le diagramme de séquence/collaboration a été créé

☑ Le diagramme de classe a été fait

Questions supplémentaires d'algo et de POO

Questions algorithmiques

☑ Ecrire les conditions pour qu'une partie se termine dans la Javadoc de GameOverSupervisor (Détaillez-les précisément en fonction des états des objets manipulés par votre implémentation) → expliquer quels sont les objets qui détienne l'information

☑ Calculer et justifier la CTT du calcul de la plus grande zone de case adjacente dans la JavaDoc de la méthode qui effectue cette opération puis indiquer clairement dans GameOverSupervisor où cette méthode est implémentée

☑ Indiquer le choix de l'interface de collection utilisée pour mémoriser les cases visitées dans la JavaDoc de la classe où la CTT est calculée (voir précédemment)

☑ Indiquer le choix d'implémentation de l'interface de collection et justifier son choix en calculant la CTT des différentes opérations faites avec la collection et justifier. Indiquer le raisonnement dans la JavaDoc de la même classe (voir précédemment)

Questions POO

☑ Les classes sont dans `treasurequest.domains` et n'ont pas de lien avec les vues (tout ce qui pourrait être en dehors de `domains`)

☑ L'encapsulation (attributs private) et les copies défencives sont bien effectuées sur toutes les classes du domains

☑ Ecrire des méthodes courtes et peu complexes (et tenter de respecter la règle de Demeter)

☑ Ecrire des classes timides (c'est à dire qui font le boulot et qui ne nécessite pas de tout le temps leur demander des choses depuis d'autres classes)

☑ Bon usage du polymorphisme

☑ Bonne utilisation des interfaces

☑ Aucune alerte PMD

☑ Tests unitaires JUnit5 dans le dossier `tests` qui donne un coverage d'au moins 90% sur chaque classe du domains (pour un degré de maitrise supplémentaire → coverage de 100% de ces classes)

☐ (degré de maitrise supplémentaire) Un plan de test exhaustif valide le code correspondant à la CTT demandée

Checklists officielles

A faire à la fin de chaque itération. Ceci sont les checklists officielles reprises depuis les tableaux excels, elle sont bien pour s'auto-évaler mais moins bien pour savoir exactement sur quoi avancer dans le projet.

Template (officielle)

Préalable

> Tous ces éléments sont à observer pour que le travail soit évaluable

☐ Une archive .zip est déposée sur Moodle.

☐ L'archive comporte un projet eclipse mentionnant le nom et le prénom (une infraction de nommage tolérée sur les 3 itérations)

☐ Le projet respecte la structure demandée (présence des répertoires src et tests, fichiers sources dans les bons paquetages. -0,25 sur la note finale par fichier mal placé)

☐ Les versions de Java, Junit et Eclipse sont conformes à celles demandées (une infraction tolérée sur les 3 itérations)

☐ Eclipse ne mentionne pas de problème de compilation (sauf problèmes de liaison au JDK, à JUnit 5)

☐

Acquis 1 : Programmer des énoncés de conception en Java selon les principes de la POO

> Seuil de réussite (tous sont à valider pour obtenir l'acquis)

☐ [Prog. Procedurale] PMD relève au plus 2 méthodes trop longues et/ou trop complexes

☐ [Cohésion] PMD relève au plus 2 classes comportant 5 attributs ou plus.

☐ [Cohésion] PMD relève au plus 1 classe divine ou une classe de données

☐ [Encapsulation] PMD relève au plus 2 attributs d'accès autres que private

☐ [Encapsulation] Au moins 2 méthodes ou constructeurs respectent le principe de copies défensives pour les collections.

☐ [Masquage] Les classes métiers respectent la loi de Déméter (2 infractions tolérées)

☐ [RDD] La plupart des types du domaine correspondent aux candidats identifiés pendant l'étape de conception.

☐ [Polymorphisme] Les paramètres des constructeurs des superviseurs sont des types abstraits (interfaces ou classe abstraite)

☐ [Polymorphisme] PMD ne lève pas d'alerte LooseCoupling

☐ [Architecture] Les tests ArchUnit ne révèlent pas de problème d'architecture

Acquis 2 : Valider les comportements des objets programmés par des tests unitaires

> Seuil de réussite (tous sont à valider pour obtenir l'acquis)

☐ Chaque classe du domaine est validée par une classe de test. Les classes de test sont définis dans le même package (mais dans le répertoire tests).

☐ L'ensemble des classes de test du domaine couvre au moins 90% des classes du domaine.

☐ Tous les tests unitaires des classes du domaine réussissent. (1 test en échec toléré)

Acquis 3 : Documenter son code

> Seuil de réussite (tous sont à valider pour obtenir l'acquis)

☐ Les types et les méthodes publics sont documentés par de la javadoc inspirée par les énoncés (cf. PMD, 5 fautes tolérées)

☐ Les pré ou postconditions demandées figurent dans la javadoc, dans l’en-tête de la méthode , et les réponses concernent la classe est correcte.

☐ La CTT demandée est correctement (justifiées, avec le pire des cas si il existe, avec signification des vairables N, M, etc) évaluée dans la Javadoc de la classe

Acquis 4 : Justifier le choix d'une structure de données particulière

> Seuil de réussite (tous sont à valider pour obtenir l'acquis)

☐ Les choix de type de collection sont correctement documentés dans le code pour les cases du terrain de jeu, dans la javadoc de la classe .

☐ Les choix d'implémentation sont correctement documentés dans le code pour les cases du terrain de jeu, dans la javadoc de la classe .

Degré de maitrise

> Vérifiés SI tous les critères de seuil de réussite ont été observés pour affiner la cote

☐ PMD ne rélève aucun problème de méthode trop complexe et/ou trop longue

☐ Plus de 2 méthodes ou constructeurs appliquent le principes de copies défensives sur les collections

☐ Au moins 2 constructeurs ou méthode de fabrique valident leurs paramètres.

☐ PMD ne relève aucun problème de type data class ou god object

☐ Les méthodes ne faisant pas partie du diagramme de classe ou des diagrammes de collaboration sont private (voire protected)

☐ Le paquetage domain est couvert à 100% par les classes de tests du même package.

☐ La documentation de plusieurs méthodes fait état de précondition et de postcondition correctes

☐ Les collections sont utilisées sans réinventer la roue (pas de reprogrammation de méthodes existantes, ou de parcours séquentiel quand l'accès direct est prévu par exemple)

☐ Un plan de test exhaustif valide le code correspondant à la CTT demandée

Checklist itération 1 (officielle)

Préalable

> Tous ces éléments sont à observer pour que le travail soit évaluable

☑ Une archive .zip est déposée sur Moodle.

☑ L'archive comporte un projet eclipse mentionnant le nom et le prénom (une infraction de nommage tolérée sur les 3 itérations)

☑ Le projet respecte la structure demandée (présence des répertoires src et tests, fichiers sources dans les bons paquetages. -0,25 sur la note finale par fichier mal placé)

☑ Les versions de Java, Junit et Eclipse sont conformes à celles demandées (une infraction tolérée sur les 3 itérations)

☑ Eclipse ne mentionne pas de problème de compilation (sauf problèmes de liaison au JDK, à JUnit 5)

☑ AI-1: Créer une partie

☑ AI-2: Voir une partie

☑ AI-3 : Revenir au menu principal

Acquis 1 : Programmer des énoncés de conception en Java selon les principes de la POO

>

> Seuil de réussite (tous sont à valider pour obtenir l'acquis)

☑ [Prog. Procedurale] PMD relève au plus 2 méthodes trop longues et/ou trop complexes

☑ [Cohésion] PMD relève au plus 2 classes comportant 5 attributs ou plus.

☑ [Cohésion] PMD relève au plus 1 classe divine ou une classe de données

☑ [Encapsulation] PMD relève au plus 2 attributs d'accès autres que private

☑ [Encapsulation] Au moins 2 méthodes ou constructeurs respectent le principe de copies défensives pour les collections.

☑ [Masquage] Les classes métiers respectent la loi de Déméter (2 infractions tolérées)

☑ [RDD] La plupart des types du domaine correspondent aux candidats identifiés pendant l'étape de conception.

☑ [Polymorphisme] Les paramètres des constructeurs des superviseurs sont des types abstraits (interfaces ou classe abstraite)

☑ [Polymorphisme] PMD ne lève pas d'alerte LooseCoupling

☑ [Architecture] Les tests ArchUnit ne révèlent pas de problème d'architecture

Acquis 2 : Valider les comportements des objets programmés par des tests unitaires

> Seuil de réussite (tous sont à valider pour obtenir l'acquis)

☑ Chaque classe du domaine est validée par une classe de test. Les classes de test sont définis dans le même package (mais dans le répertoire tests).

☑ L'ensemble des classes de test du domaine couvre au moins 90% des classes du domaine.

☑ Tous les tests unitaires des classes du domaine réussissent. (1 test en échec toléré)

Acquis 3 : Documenter son code

> Seuil de réussite (tous sont à valider pour obtenir l'acquis)

☑ Les types et les méthodes publics sont documentés par de la javadoc inspirée par les énoncés (cf. PMD, 5 fautes tolérées)

☑ Les pré ou postconditions demandées figurent dans la javadoc, dans l’en-tête de la méthode PlayGameSuperviser.onEnter, et les réponses concernent les 3 classes Player, CaseMap et Case et sont correctes.

☑ La CTT demandée est correctement (justifiées, avec le pire des cas si il existe, avec signification des vairables N, M, etc) évaluée dans la Javadoc de la classe TreasureQuestGameFactory

Acquis 4 : Justifier le choix d'une structure de données particulière

> Seuil de réussite (tous sont à valider pour obtenir l'acquis)

☑ Les choix de type de collection sont correctement documentés dans le code pour les cases du terrain de jeu, dans la javadoc de la classe CaseMap.

☑ Les choix d'implémentation sont correctement documentés dans le code pour les cases du terrain de jeu, dans la javadoc de la classe CaseMap.

Degré de maitrise

> Vérifiés SI tous les critères de seuil de réussite ont été observés pour affiner la cote

☑ PMD ne rélève aucun problème de méthode trop complexe et/ou trop longue

☑ Plus de 2 méthodes ou constructeurs appliquent le principes de copies défensives sur les collections

☑ Au moins 2 constructeurs ou méthode de fabrique valident leurs paramètres.

☑ PMD ne relève aucun problème de type data class ou god object

☑ Les méthodes ne faisant pas partie du diagramme de classe ou des diagrammes de collaboration sont private (voire protected)

☑ Le paquetage domain est couvert à 100% par les classes de tests du même package.

☑ La documentation de plusieurs méthodes fait état de précondition et de postcondition correctes

☑ Les collections sont utilisées sans réinventer la roue (pas de reprogrammation de méthodes existantes, ou de parcours séquentiel quand l'accès direct est prévu par exemple)

☐ Un plan de test exhaustif valide le code correspondant à la CTT demandée

Checklist itération 2 (officielle)

Préalable

> Tous ces éléments sont à observer pour que le travail soit évaluable

☑ Une archive .zip est déposée sur Moodle.

☑ L'archive comporte un projet eclipse mentionnant le nom et le prénom (une infraction de nommage tolérée sur les 3 itérations)

☑ Le projet respecte la structure demandée (présence des répertoires src et tests, fichiers sources dans les bons paquetages. -0,25 sur la note finale par fichier mal placé)

☑ Les versions de Java, Junit et Eclipse sont conformes à celles demandées (une infraction tolérée sur les 3 itérations)

☑ Eclipse ne mentionne pas de problème de compilation (sauf problèmes de liaison au JDK, à JUnit 5)

☑ AI-4 : Changer de case active

☑ AI-5 : Creuser une case

☑ AI-6 : Fournir des indices

Acquis 1 : Programmer des énoncés de conception en Java selon les principes de la POO

>

> Seuil de réussite (tous sont à valider pour obtenir l'acquis)

☑ [Prog. Procedurale] PMD relève au plus 2 méthodes trop longues et/ou trop complexes

☑ [Cohésion] PMD relève au plus 2 classes comportant 5 attributs ou plus.

☑ [Cohésion] PMD relève au plus 1 classe divine ou une classe de données

☑ [Encapsulation] PMD relève au plus 2 attributs d'accès autres que private

☑ [Encapsulation] Au moins 2 méthodes ou constructeurs respectent le principe de copies défensives pour les collections.

☑ [Masquage] Les classes métiers respectent la loi de Déméter (2 infractions tolérées)

☑ [RDD] La plupart des types du domaine correspondent aux candidats identifiés pendant l'étape de conception.

☑ [Polymorphisme] Les paramètres des constructeurs des superviseurs sont des types abstraits (interfaces ou classe abstraite)

☑ [Polymorphisme] PMD ne lève pas d'alerte LooseCoupling

☑ [Architecture] Les tests ArchUnit ne révèlent pas de problème d'architecture

Acquis 2 : Valider les comportements des objets programmés par des tests unitaires

> Seuil de réussite (tous sont à valider pour obtenir l'acquis)

☑ Chaque classe du domaine est validée par une classe de test. Les classes de test sont définis dans le même package (mais dans le répertoire tests).

☑ L'ensemble des classes de test du domaine couvre au moins 90% des classes du domaine.

☑ Tous les tests unitaires des classes du domaine réussissent. (1 test en échec toléré)

Acquis 3 : Documenter son code

> Seuil de réussite (tous sont à valider pour obtenir l'acquis)

☑ Les types et les méthodes publics sont documentés par de la javadoc inspirée par les énoncés (cf. PMD, 5 fautes tolérées)

☑ Les pré ou postconditions demandées figurent dans la javadoc, dans l’en-tête de la méthode PlayGameSuperviser.onDig, et les réponses concernent la classe TreasureQuestGame est correcte.

☑ La CTT demandée est correctement (justifiées, avec le pire des cas si il existe, avec signification des vairables N, M, etc) évaluée dans la Javadoc de la classe TreasureQuestGame

Acquis 4 : Justifier le choix d'une structure de données particulière

> Seuil de réussite (tous sont à valider pour obtenir l'acquis)

☑ Les choix de type de collection sont correctement documentés dans le code pour les cases du terrain de jeu, dans la javadoc de la classe TreasureQuestGame.

☑ Les choix d'implémentation sont correctement documentés dans le code pour les cases du terrain de jeu, dans la javadoc de la classe TreasureQuestGame.

Degré de maitrise

> Vérifiés SI tous les critères de seuil de réussite ont été observés pour affiner la cote

☑ PMD ne rélève aucun problème de méthode trop complexe et/ou trop longue

☑ Plus de 2 méthodes ou constructeurs appliquent le principes de copies défensives sur les collections

☑ Au moins 2 constructeurs ou méthode de fabrique valident leurs paramètres.

☐ PMD ne relève aucun problème de type data class ou god object

☑ Les méthodes ne faisant pas partie du diagramme de classe ou des diagrammes de collaboration sont private (voire protected)

☑ Le paquetage domain est couvert à 100% par les classes de tests du même package.

☑ La documentation de plusieurs méthodes fait état de précondition et de postcondition correctes

☑ Les collections sont utilisées sans réinventer la roue (pas de reprogrammation de méthodes existantes, ou de parcours séquentiel quand l'accès direct est prévu par exemple)

☐ Un plan de test exhaustif valide le code correspondant à la CTT demandée

Checklist itération 3 (officielle)

Préalable

> Tous ces éléments sont à observer pour que le travail soit évaluable

☑ Une archive .zip est déposée sur Moodle.

☑ L'archive comporte un projet eclipse mentionnant le nom et le prénom (une infraction de nommage tolérée sur les 3 itérations)

☑ Le projet respecte la structure demandée (présence des répertoires src et tests, fichiers sources dans les bons paquetages. -0,25 sur la note finale par fichier mal placé)

☑ Les versions de Java, Junit et Eclipse sont conformes à celles demandées (une infraction tolérée sur les 3 itérations)

☑ Eclipse ne mentionne pas de problème de compilation (sauf problèmes de liaison au JDK, à JUnit 5)

☑ AI-7 Terminer une partie

☑ AI-8 Générer des statistiques

☑ AI-9 Générer un terrain aléatoire

Acquis 1 : Programmer des énoncés de conception en Java selon les principes de la POO

>

> Seuil de réussite (tous sont à valider pour obtenir l'acquis)

☑ [Prog. Procedurale] PMD relève au plus 2 méthodes trop longues et/ou trop complexes

☑ [Cohésion] PMD relève au plus 2 classes comportant 5 attributs ou plus.

☑ [Cohésion] PMD relève au plus 1 classe divine ou une classe de données

☑ [Encapsulation] PMD relève au plus 2 attributs d'accès autres que private

☑ [Encapsulation] Au moins 2 méthodes ou constructeurs respectent le principe de copies défensives pour les collections.

☑ [Masquage] Les classes métiers respectent la loi de Déméter (2 infractions tolérées)

☑ [RDD] La plupart des types du domaine correspondent aux candidats identifiés pendant l'étape de conception.

☑ [Polymorphisme] Les paramètres des constructeurs des superviseurs sont des types abstraits (interfaces ou classe abstraite)

☑ [Polymorphisme] PMD ne lève pas d'alerte LooseCoupling

☑ [Architecture] Les tests ArchUnit ne révèlent pas de problème d'architecture

Acquis 2 : Valider les comportements des objets programmés par des tests unitaires

> Seuil de réussite (tous sont à valider pour obtenir l'acquis)

☑ Chaque classe du domaine est validée par une classe de test. Les classes de test sont définis dans le même package (mais dans le répertoire tests).

☑ L'ensemble des classes de test du domaine couvre au moins 90% des classes du domaine.

☑ Tous les tests unitaires des classes du domaine réussissent. (1 test en échec toléré)

Acquis 3 : Documenter son code

> Seuil de réussite (tous sont à valider pour obtenir l'acquis)

☑ Les types et les méthodes publics sont documentés par de la javadoc inspirée par les énoncés (cf. PMD, 5 fautes tolérées)

☑ La CTT demandée est correctement (justifiées, avec le pire des cas si il existe, avec signification des vairables N, M, etc) évaluée dans la Javadoc de la classe `GameOverSupervisor`

Acquis 4 : Justifier le choix d'une structure de données particulière

> Seuil de réussite (tous sont à valider pour obtenir l'acquis)

☑ Les choix de type de collection sont correctement documentés dans le code pour les cases du terrain de jeu, dans la javadoc de la classe `GameOverSupervisor`

☑ Les choix d'implémentation sont correctement documentés dans le code pour les cases du terrain de jeu, dans la javadoc de la classe `GameOverSupervisor`

Degré de maitrise

> Vérifiés SI tous les critères de seuil de réussite ont été observés pour affiner la cote

☑ PMD ne rélève aucun problème de méthode trop complexe et/ou trop longue

☑ Plus de 2 méthodes ou constructeurs appliquent le principes de copies défensives sur les collections

☑ Au moins 2 constructeurs ou méthode de fabrique valident leurs paramètres.

☑ PMD ne relève aucun problème de type data class ou god object

☑ Les méthodes ne faisant pas partie du diagramme de classe ou des diagrammes de collaboration sont private (voire protected)

☑ Le paquetage domain est couvert à 100% par les classes de tests du même package.

☐ La documentation de plusieurs méthodes fait état de précondition et de postcondition correctes

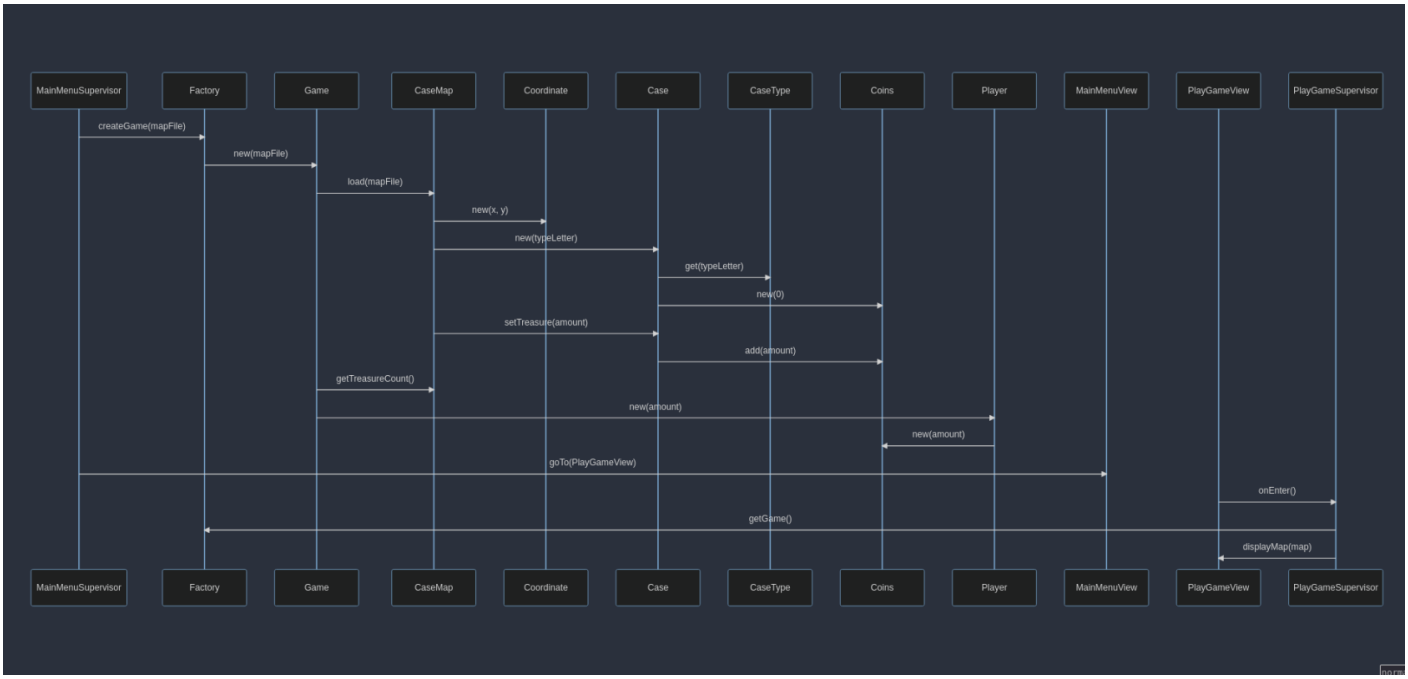
☑ Les collections sont utilisées sans réinventer la roue (pas de reprogrammation de méthodes existantes, ou de parcours séquentiel quand l'accès direct est prévu par exemple)

☐ Un plan de test exhaustif valide le code correspondant à la CTT demandée

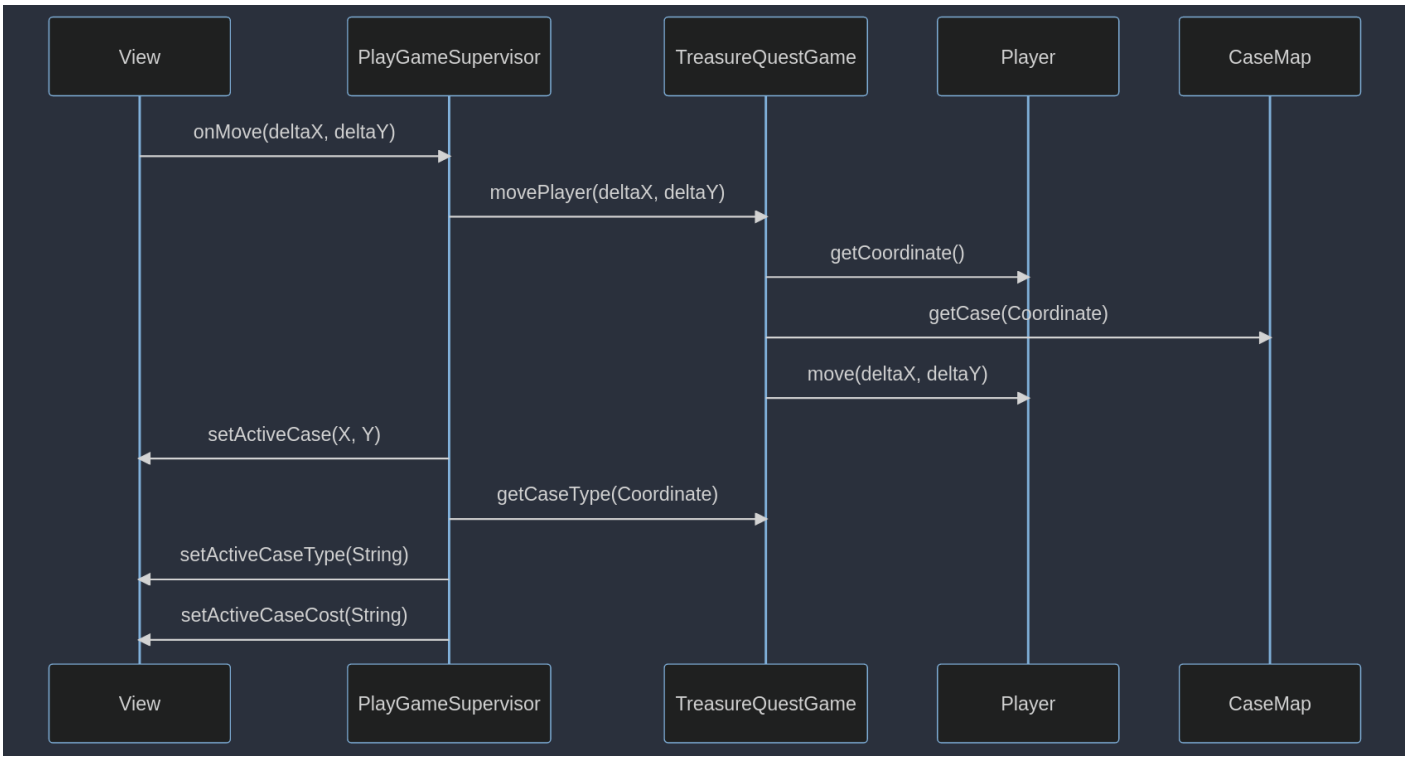
Conception du projet

Diagrammes séquences

- Créer un nouveau jeu



- Se déplacer de case en case



- Creuser une case

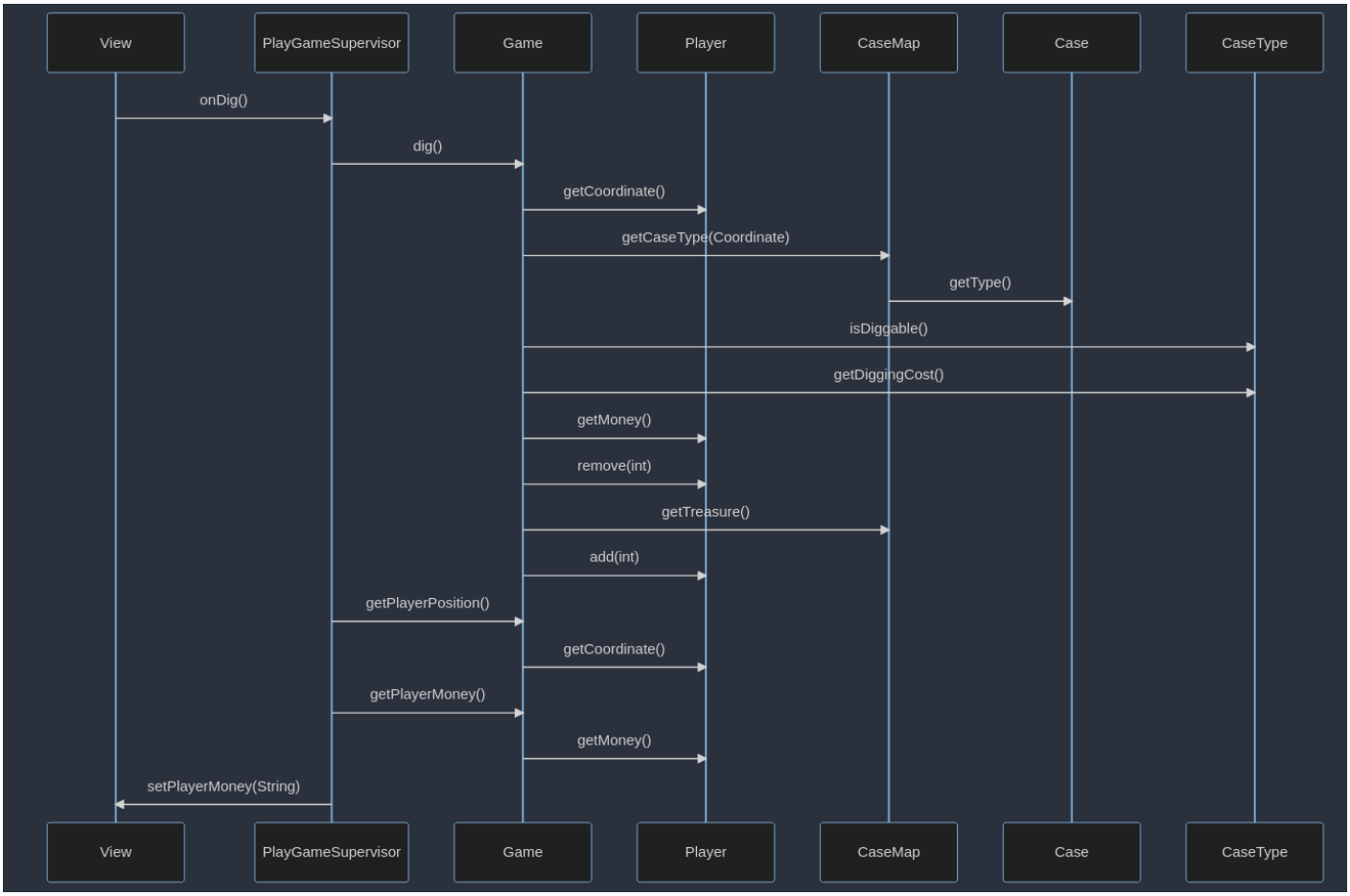


Diagramme de classe

nan

Bonne pratiques

Les bonnes pratiques données ci-dessous viennent principalement de la chaine YouTube "Code Aestetics" et du language de programmation Rust

Nommer les choses

- ☐ Eviter d'utiliser des nombres random dans le code et les remplacer par des constantes
- ☐ Ne pas abréger le noms des méthodes ou le nom des variables
- ☐ Ne pas répèter les types dans les noms des variables
- ☐ Spécifier les unités dans les variables, ou mieux, créer un type qui le représente
- ☐ Eviter d'utiliser des noms/préfixes génériques tel que "I", "Utils", "Helper", "Abstract" dans les noms des classes
- ☐ Il n'est pas nécessaire de donner des informations sur les détails de fonctionnement (par exemple écrire "Interface" ou "I" dans le nom d'une interface n'est pas nécessaire)

Eviter les commentaires et rendre le code plus robuste

- ☐ Eviter d'utiliser trop de commentaires, car ils peuvent devenir obsolètes et ne sont pas nécessaire quand le code est proprement documenté. A la place essayer de rendre le code si clair qu'il s'exprimme comme le commentaire.
- ☐ Utiliser des commentaires quand l'implémentation semble quelque peu bizarre (par exemple pour des raisons d'optimisation)
- ☐ Les conditions peuvent être extraites dans des méthodes séparées
- ☐ Utiliser des enums pour tous les états fixent pour empécher des états impossibles d'être représenter
- ☐ Si possible utiliser des types tel que "Option" ou "Result" pour obliger de prendre en compte les cas de valeurs `null` ou d'erreurs

Améliorer la lisibilité du code encore plus

- [ ] Eviter d'utiliser plus de 3 niveau d'intendation dans une méthode
  - [ ] Extraire des bouts d'une méthode dans plusieurs autres
  - [ ] Notter les conditions d'une méthode au desus (cela permet de supprimer les `else`)
- [ ] Ecrire une documentation (comment *utiliser* le code) de très bonne qualité et la garder à jour.
  - [ ] Décrire le but d'une classe et des interfaces et ce qu'elle représente
  - [ ] Décrire les attentes que les interfaces ont par rapport à leurs implémentations (comment elles doivent gérer X ou Y)
  - [ ] Décrire les buts des méthodes, leurs préconditoins et leurs post-conditions

Principes orienté objet

- [ ] Eviter de faire des classes intuiles (qui ne font que très peu de choses) → data class
- [ ] Eviter de faires des classes qui font tout (classes dieu)
- [ ] Eviter d'appeller des méthodes d'objets qui ne sont pas des attributs, des paramètres, créé par, ou de même classe dans une méthode. → eviter d'appeller des méthodes d'objets retournés par une autre méthode. Et eviter de les donner en premier lieu
- [ ] Rendre tous les attributs `private` et rendre un maximum de classes immuable si possible
- [ ] Vérifier les paramètres des méthodes et des constructeurs pour empécher que des états invalides se présentent (par exemple avec des valeurs `null`)
- [ ] Quand quelque chose en lien direct avec les attributs est renvoyé, s'assurer qu'il est immuable
- [ ] Eviter de faire ensorte que les autres classes fassent le boulot d'une classe à sa place → extraire des bouts de méthode dans une autre classe
- [ ] Eviter les dépendences mutuelles ou circulaires entre classes
- [ ] Empécher l'héritage en metant `final` sur toutes les classes

MVC et MVP

☐ Les classes métier ne doivent avoir aucun contact avec des classes de la vue

☐ La vue peut être liée au superviseur par le biais d'une interface (on crée d'abord le superviseur, crée une vue qui implémente une interface de vue en lui passant le superviseur, on donne la vue à travers l'interface au superviseur)

☐ Pour des objets qui doivent avoir un état partagé entre tous les supervisuers on peut utiliser une Factory qui va être passée à tous les supervisuers et qui va garder en mémoire l'objet en cours pour le créer et le récupèrer)

☐ Dans le cas d'un *presenter (MVP)* les superviseurs vont récupérer les informations brutes depuis les classes métier, les process et les afficher dans la vue

☐ Dans le cas d'un *controlleur (MVC)* les superviseurs vont récupérer les informations déjà bien formattée et vont simplmeent les mettre dans la vue sans plus de traitement

☐ Les classes métier ne doivent pas dépendre du superviseur

Tests unitaires

☐ Créer une interface d'aléatoire et une fausse et vrai classe aléatoire pour les tests (et faire une surcharge de constructeurs)

☐ Tenter d'obtenir un code coverage de 90% au minimum sur toutes les classes

☐ Décrire les tests sous forme de "Étant donné *une condition* Quand *une action* Alors *une/plusieurs postcondition*"

Git

☐ Do not make bunduled commits of a lot of things, really commit one thing at the time

☐ You can use emojis in your commits to represent differnet sort of tasks (documentation, refactoring, testing, etc) see GitMoji

☐ Add a README (you can pimp it using shields, GIF, installation instructions, usage instructions, license information, general description, and features)