LO02

Diagrammes finaux

Maëly TAN | Léo KEMPLAIRE

Automne

utt
UNIVERSITÉ DE TECHNOLOGIE
TROYES

Table des matières

Diagramme de classe initial

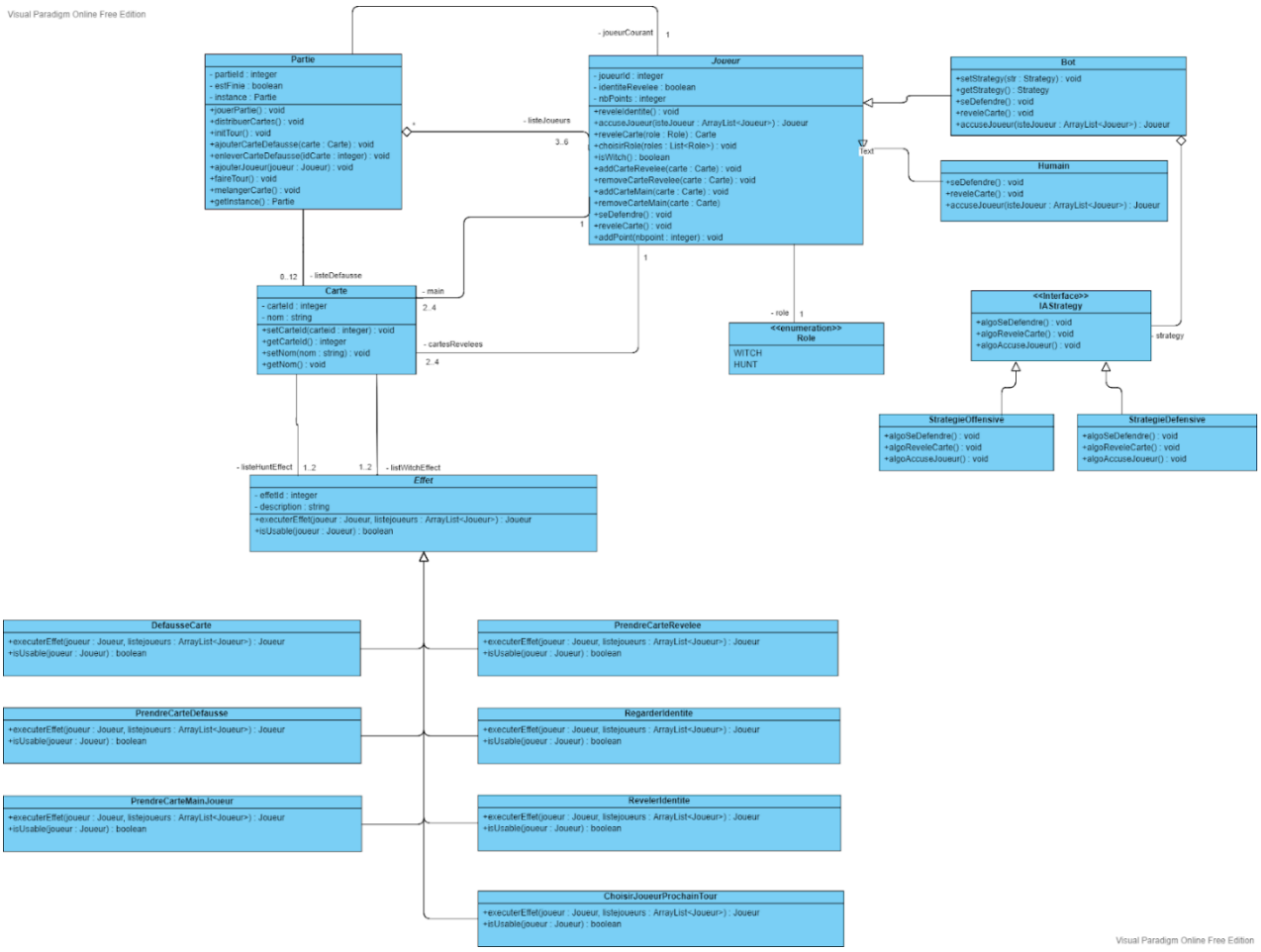

Diagramme de classe final

➔ **Voir le repertoire “/doc” du code source du projet**

Etat de l'application

Changements entre le diagramme initial et le diagramme final

La modélisation initiale de notre application était une première projection de la structure que devait adopter l'application en vue de respecter le cahier des charges, contenant ainsi les principales classes et les liens évidents entre ces dernières. Nous avons dû confronter ce plan à la réalité de l'implémentation, ce qui nous a permis de valider ou d'invalider certaines hypothèses de conception et de nous rendre compte des changements et ajouts nécessaires pour obtenir une réalisation complète et viable. Cela nous a principalement permis d'enrichir ce premier diagramme en ayant une approche plus concrète et détaillée et d'opérer certains changements que nous allons expliciter ici.

- De manière générale, dans la plupart des classes du modèle, nous avons ajouté un certain nombre d'attributs pour caractériser de manière plus détaillée l'état de ces classes. La classe Partie contient par exemple plusieurs nouveaux attributs qui nous renseignent mieux sur l'état du jeu : nbRound (numéro du round en cours), joueur, carteChoisi, actionChoisi (qui nous permettent de connaître le résultat des dernières actions initiées), joueurCourant, prochainJoueur…

- L'ajout de ces attributs a été accompagné de l'ajout de méthodes utilitaires qui facilitent l'accès aux données du modèle pour simplifier certains traitements. Par exemple, la classe Joueur possède maintenant une méthode getJoueursAvecCarteRevelees() qui nous renvoie une collection de joueurs en jeu ayant leur carte révélée. Cette collection peut ensuite être passée en paramètre de méthodes où l'on a à choisir un joueur correspondant à un critère précis (ici les joueurs qui ont une carte révélée).

- Changements relatifs à la classe Partie : La partie contient des listes de joueurs différenciés : les joueurs en jeu ou les joueurs de la liste initiale de jeu. Nous avons ajouté des méthodes pour être renseignés sur l'état de la Partie (verifRoundTermine(), isProchainJoueurDetermine(), isError()) accompagné de méthodes d'affichage (afficherPointsJoueur() …) et pour le modifier (ajouter des cartes à la défausse, modifier les listes de joueurs..). Nous avons aussi implémenté d'autres méthodes qui participent à gérer les évènements et la boucle de jeu pour qu'une partie puisse se dérouler entièrement en plusieurs étapes : Initialisation et lancement de la partie (méthodes d'initialisation de la partie avec création de cartes, initialisation des joueurs...), Initialisation et déroulement de chaque round (distribution des cartes, réinitialisation de l'état des joueurs…) et fin de la partie.

- Concernant les joueurs, l'organisation de base des classes (un humain et un bot hérite de joueur, un bot a une stratégie) n'a pas changé. La classe Joueur a maintenant des méthodes lui permettant de réaliser chacun de ses comportements de jeu (joueTour(), seDefendre(), reveleIdentite(), reveleCarte(), accuseJoueur()...) et de changer son état en fonction des évènements qui surviennent (ajouteCarteMain(), defausseCarte() …).
  Contrairement au premier diagramme, elle possède maintenant une liste importante de méthodes abstraites qui sont des méthodes relatives aux choix que les joueurs ont à faire durant la partie. Nous les avons mis en abstraite car selon si le joueur est un bot ou un Humain, ces choix ne seront pas faits de la même manière : pour un humain on attend une entrée, pour un Bot la décision est déterminée automatiquement à partir de sa stratégie. Les classes Humain et Bot donc des Joueurs augmentés qui implémentent de manière différente les méthodes où ils ont à faire des choix. Exemple, pour la méthode choisirJoueurAAccuser(), du

côté de l'humain on attend une entrée alors que du côté du bot on fait appel à une méthode algoChoisirJoueurAAccuser() qui renvoie automatiquement un joueur à accuser. Ces méthodes sont appelées à partir de joueur dès qu'un joueur a à faire un choix. Nous avons ajouté plusieurs stratégies qui, chacune hérite de IAStrategy qui déclare de manière abstraite toutes méthodes censées contenir un algorithme pour résoudre automatiquement un choix (prendre une décision). Nous avons ajouté des classes de stratégie concrètes : StratégieRandom, StratégieSmart…

- Nous avons ajouté un certain nombre d'énumérations pour caractériser des attributs de classes qui ne peuvent prendre que des valeurs précises : CarteNom (qui définit l'ensemble des noms que peut avoir une carte) pour le nom d'une carte, CarteNom…

- Nous avons rendu les composantes Action et Role centrales pour le fonctionnement du jeu : Un joueur doit souvent choisir entre plusieurs actions (définies dans l'enum ChoixAction) pour déterminer l'action qu'il va entreprendre (définie dans l'enum Action) parmi les actions qu'il a la possibilité de faire. Certaines méthodes vont aussi prendre Role en paramètre puisque toutes ne prévoient pas les mêmes traitements selon si on l'exécute en tant que Witch ou Hunt (exemple : reveleCarte() en tant que witch ou hunt).

- La classe carte a été complétée avec l'ensemble des méthodes qui permettent de modifier son état (ajout / retrait d'effets) et une méthode pour exécuter tous les effets d'une carte. Une carte a maintenant une condition Witch, et une condition Hunt de type Condition. Une condition est une condition pour exécuter les effets d'une carte (effets witch ou hunt). Une condition est une classe abstraite qui prévoit une méthode pour vérifier si la condition est valide. Nous avons ainsi créé une classe pour chaque condition existante qui hérite de Condition (AvoirReveleCarte, EtreReveleVillageois..). Cette condition remplace la méthode isUsable() que nous avions mise sur chaque effet.

- Nous avons ajouté des effets qu'il manquait au jeu sur le premier diagramme.

- Nous avons des classes utilitaires qui sont venues se rajouter, comme Utils qui est censée gérer les entrées au clavier saisies par l'utilisateur, et SortByPoint qui est une classe implémentant Comparator pour trier des listes de joueurs (selon leurs points par exemple).

- Pour l'implémentation de la 3e phase, nous avons adopté une organisation MVC ce qui a abouti à l'ajout de nouvelles classes : des classes pour les Vues (ViewInitialiser, ViewGame) et les contrôleurs qui leurs sont associés (ControllerInitialiser et ControllerGame). Le reste des classes est considéré comme appartenant au modèle.
  Les vues présentent le modèle et recueillent les intéractions utilisateurs.
  Les controlleurs gèrent ces intéractions utilisateurs et les utilisent pour déterminer le déroulement de la partie, changer l'état du modèle en fonction mais aussi agir sur la vue. Les différentes vues initialisent leurs controlleurs respectifs qui se connaissent mutuellement pendant que le modèle n'a pas connaissance de leur existence.

Ce qui a été implémenté

Relativement au cahier des charges, nous avons pu implémenter les éléments suivants :

- Respecter le nombre de joueurs (entre 3 et 6)
- L'implémentation de joueurs virtuels (classe Bot.java) et de joueurs humains (Humain.java). Ces deux classes héritent de la classe abstraite Joueur.java
- L'implémentation d'une interface graphique : l'application de jeu peut être lancée et jouée via une interface graphique. Il n'est plus possible de jouer via la console, mais le code de la phase 2 le permet
- Le respect d'une organisation fonctionnant sous le modèle MVC
- La présence de javadoc sur tous les attributs (sauf attributs relatifs à swing) et toutes les méthodes complexes (toutes les méthodes sauf les assesseurs et constructeurs)

Pour implémenter ce logiciel, nous avons utilisé plusieurs **patrons de conception** tels que :

- **Le Singleton** : la classe Partie.java a été développée avec ce modèle, ce qui nous a permis de pouvoir accéder à tous les éléments relatifs à une partie depuis les autres classes.
- **Le patron Strategy** : Les classes IAStrategye.java, StrategyRandom.java, StrategySmart.java, StrategyPassive.java, StrategyActive.java et Bot.java suivent ce modèle. Cela a permis d'associer une stratégie différente lors de la création du robot. A sa création, le robot pouvait alors comporter une stratégie parmi les quatre proposées.
- **Le patron Observable / Observer** : Les Vues héritent de la classe Observers et certains éléments du modèle (Partie et Joueur principalement) implémentent Observable. Ainsi sur certains aspects, lorsque les éléments du modèle changent, ils notifient les vues pour qu'elles se mettent à jour.

Ce qui n'a pas été implémenté

Dû au manque de temps, certains éléments n'ont pas pu être implémentés et restent aujourd'hui une amélioration à apporter à notre application. Les éléments sont les suivants :

- La possibilité de jouer au jeu avec l'interface graphique en même temps qu'en ligne de commande, ce qui aurait impliqué d'utiliser les principes de programmation concurrente (avec Threads...). Nous n'avons pas utilisé de Threads.

- Le patron de conception Visitor n'a pas été implémenté pour le comptage des points de chaque joueur.

Ce qui fonctionne

Les éléments de l'application fonctionnels sont les suivants :

- Pouvoir sélectionner le nombre de joueurs humains et robots
- Pouvoir choisir un rôle et un nom manuellement (pour les humains) et automatiquement (pour les robots)

- Pouvoir jouer une partie avec des joueurs Humain et des joueurs Bot (leurs actions sont déterminées automatiquement grâce à leur stratégie)

- Une partie se termine lorsqu'un des joueurs est arrivé à 5 points :
    - On obtient le joueur qui gagne la partie
    - On affiche le nombre de points de chaque joueur

- Un round se termine lorsque tous les joueurs ont révélé leur identité sauf un, ou que plus aucun joueur n'a de cartes sauf un :
    - On obtient le joueur qui gagne le round : lorsqu'il gagne, il remporte le nombre de points relatifs à son rôle
    - On affiche les points des joueurs
    - On passe au prochain round

- Lorsqu'un nouveau round est lancé :
    - Les joueurs peuvent choisir un nouveau rôle
    - Les joueurs qui ont précédemment été éliminés peuvent rejoindre le nouveau round
    - Chaque joueur conserve les points qu'il a gagné au(x) round(s) précédent(s)

- Lorsqu'un joueur joue :
    - Il peut **se défendre** :
        - Révéler son identité
        - Révéler une carte rumeur avec l'effet HUNT
    - Il peut **attaquer** :
        - Accuser un autre joueur
        - Révéler une carte rumeur avec l'effet WITCH

- Possibilité de choisir un joueur en fonction de la situation :
    - Si on veut accuser quelqu'un :
        - Choix du joueur parmi les joueurs qui n'ont pas leur identité révélée

    - Si on veut choisir un joueur lors d'un effet (ex : ChoisirProchainJoueur.java)
        - Choix du joueur parmi les joueurs toujours dans la partie (les joueurs ayant choisi le rôle HUNT peuvent avoir révélé leur identité mais restent tout de même dans la partie)

- Possibilité de choisir une carte parmi une liste de carte (main d'un joueur, défausse, cartes révélées)
- Les effets présents sur les cartes peuvent être joués en respectant les conditions qu'ils comportent. Si un des effets ne peut être joué mais qu'un autre le peut, alors on peut quand même jouer l'effet qui peut être exécuté.

Ce qui reste bogué et à améliorer

Concernant les bogues qui sont toujours présents, nous pourrions améliorer notre amélioration en :

- Corrigeant le bogue d'affichage sur l'interface graphique lorsqu'une erreur (seulement les erreurs qui surviennent lorsqu'un effet d'une carte ne peut pas s'effectuer) ne s'efface pas et

reste affichée lorsqu'on passe à l'étape suivante. Ainsi dans certaines situations, certains messages affichés sur l'interface se superposent, les rendant illisibles.

- Un ou deux effets qui impliquent le choix de la part d'un autre joueur sur l'interface que le joueur courant (ayant exécuté l'effet) ne sont pas implémentés à 100% dans le cadre du jeu avec l'interface. C'est le cas de l'effet RevelerOuDefausser.

- Une partie du code pourrait être restructurée car en raison de contraintes de temps, nous avons lors de la phase 3 déplacé une petite partie de la logique qui se trouvait dans le modèle au sein du controlleur, lorsqu'il est nécessaire d'initier des traitements à la suite d'actions réalisées sur la vue par les utilisateurs, pour assurer le déroulement d'une partie. C'est une amélioration possible qui aurait été envisageable si nous avions eu plus de temps.

- Quelques méthodes que nous utilisions dans la phase 2 sont maintenant dépassées car leurs instructions sont appelées dans le controlleur (exemple : dans Joueur, Partie, la méthodes joueTour()). Une restructuration pourrait aussi être nécessaire à ce niveau-là.

- Les conditions au niveau des cartes BroomStick et Wart "While revealed, you cannot be chosen…" n'ont pas été implémentées ; c'est un oubli.

- Nous pourrions implémenter le fait de jouer de manière concurrente avec l'interface et la console grâce à des Threads, nous ne l'avons pas fait par manque de temps.