MIAGE RÉSEAU DES MIAGE DE FRANCE Nice

Rapport final

PROJET INFORMATIQUE KNARR – DONATILEO

DOGLIOLI-RUPPER Germain, LAPORTE Logan,
SANTHAKUMARAN Akira
GROUPE C | NOVEMBRE 2024

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

Glossaire

Carte : Élément de base du jeu représentant soit un Viking, soit une destination. Les cartes possèdent des caractéristiques spécifiques et sont gérées par des classes dérivées.

CarteBateau : Carte spéciale distribuée à chaque joueur en début de partie, contenant des pions de départ (recrue et bracelet).

CarteDestination : Carte représentant un lieu à explorer. Elle offre des gains ou des avantages lorsque les conditions d'exploration sont remplies.

CarteViking : Carte représentant un personnage Viking. Distribuée aux joueurs et utilisée pour réaliser des actions, elle peut octroyer différents types de gains.

Commerce : Action permettant aux joueurs d’échanger des ressources pendant leur tour, soit avant, soit après une action principale (recruter, explorer).

Exploration : Action permettant aux joueurs de piocher une carte destination et de l'ajouter à leur collection, à condition de remplir les exigences de coût.

Gain : Avantage ou récompense obtenu lors de certaines actions (comme recruter ou explorer). Les gains peuvent inclure des points, des recrues, ou des bracelets.

Jeu : Classe centrale qui maintient l'état général de la partie, incluant le plateau, les joueurs, et les decks de cartes.

JeuController : Composant responsable de l'orchestration des interactions entre les joueurs et le système de jeu. Il assure la gestion des actions et des affichages.

Joueur : Représentation d'un participant dans le jeu. Il possède une main de cartes, une zone d’équipage, et peut effectuer des actions pendant son tour.

MoteurJeu : Composant principal qui gère le déroulement des tours, les vérifications des conditions de fin de jeu, et l'exécution des actions des joueurs.

Pion : Ressource spéciale donnée aux joueurs au début de la partie (pion recrue et pion bracelet). Elle est utilisée pour effectuer certaines actions dans le jeu.

Recruter : Action permettant aux joueurs de piocher une carte Viking et de l’ajouter à leur main. Cette action est une des principales du jeu et permet de renforcer la stratégie des joueurs.

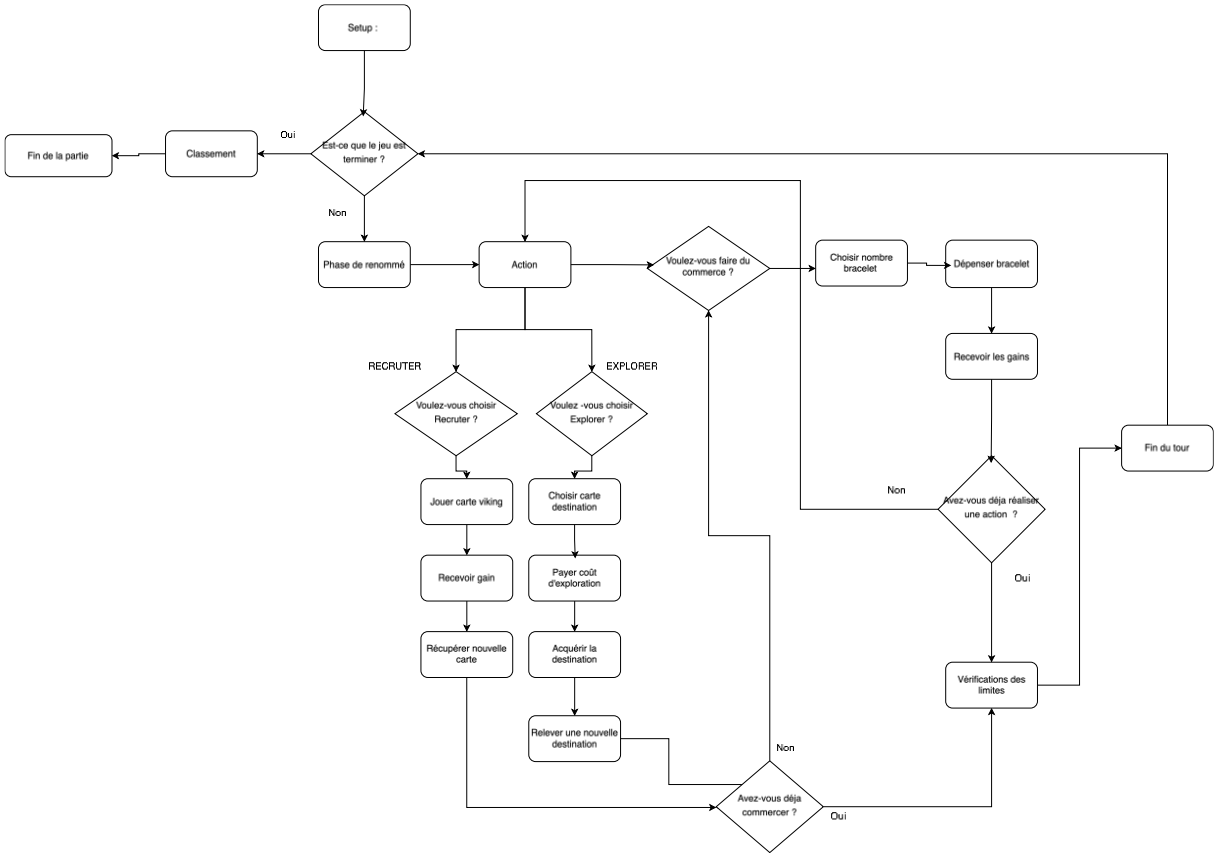
Ressources : Cartes ou pions possédés par le joueur, utilisés pour effectuer des actions ou acquérir des cartes spéciales.

User Story (US) : Description d’un besoin fonctionnel du point de vue de l’utilisateur (ici, le joueur ou bot). Les User Stories servent à structurer et organiser le développement des fonctionnalités.

VikingDeck : Paquet de cartes Viking utilisé pour distribuer des cartes aux joueurs pendant le recrutement.

Zone d’équipage : Espace où les joueurs placent leurs cartes Viking actives. Cette zone représente les ressources et personnages que le joueur a à sa disposition pour effectuer des actions.

Diagramme d’activité



Fonctionnalités traitées

Dans cette version finale, l’implémentation des règles de base du jeu (sans les artefacts) a été réalisée avec succès. Les bots peuvent désormais participer à une partie complète et chaque joueur commence avec trois cartes Viking et une carte Bateau. L'initialisation des ressources de départ, comprenant les pions Recrut et Bracelet, est parfaitement fonctionnelle. Les actions principales du jeu, telles que "Recrut" et "Explorer", sont entièrement opérationnelles. L'action "Recrut" permet aux joueurs de piocher une carte Viking, et "Explorer" leur permet d'explorer une destination, bien que cette dernière ait nécessité un léger ajustement pour résoudre un problème d’affichage initial.

L’action "Commercer" est également maintenant pleinement fonctionnelle, permettant aux bots de commercer au début ou à la fin de leur tour, ce qui leur offre des opportunités stratégiques pour obtenir des gains variés. Ces actions ont été testées et validées lors du premier tour de jeu, garantissant une gestion correcte du déroulement de la partie.

Un élément essentiel de cette version est l'intégration des cartes et des ressources via des fichiers JSON. L'utilisation de la bibliothèque Jackson permet de récupérer et manipuler efficacement les données des cartes, telles que leur couleur, leurs gains associés, ainsi que les cartes de destination. Les cartes Viking sont utilisées correctement pour l'action de recrutement, tandis que les cartes de destination sont utilisées lors de l'exploration, chaque pioche reflétant fidèlement les informations réelles des cartes.

Les ajouts et améliorations par rapport à la version précédente incluent :

1. **Système de sauvegarde des données** : Un système de sauvegarde a été intégré pour permettre aux joueurs d'enregistrer l'état du jeu à tout moment. Cette fonctionnalité permet aux joueurs de sauvegarder une partie en cours et de reprendre plusieurs parties simultanément. Les données essentielles, telles que l'état des ressources, des cartes et des pions, sont stockées et récupérées correctement via des fichiers JSON, assurant ainsi la continuité du jeu sans perte d'informations.

2. **Tests automatisés** : Un ensemble de tests automatisés a été mis en place pour valider le bon fonctionnement du jeu dans différents scénarios. Ces tests permettent de s'assurer que toutes les actions du jeu, telles que "Recrut", "Explorer", et "Commercer", sont correctement exécutées. Ils vérifient également le respect des règles du jeu et la fluidité des interactions entre les joueurs et les bots.

3. **Gestion des tours de jeu** : Le système de gestion des tours a été amélioré pour permettre l'exécution fluide de plusieurs tours. À chaque tour, les actions des bots sont correctement prises en compte, et le déroulement des tours est bien structuré. Les bots peuvent maintenant enchaîner leurs actions de manière cohérente, et l'ensemble du jeu avance de manière fluide.

4. **Optimisation des bots et amélioration de la prise de décision** : L'algorithme MCTS (Monte Carlo Tree Search) a été ajouté pour la prise de décision des bots C et D. Ce nouvel ajout permet aux bots d'analyser les situations du jeu de manière plus stratégique et réactive. L'optimisation des calculs a permis d'améliorer la gestion des ressources par les bots, tout en réduisant la consommation de ressources pour des calculs plus efficaces.

5. **Refactorisation du code** : Le code a été revu et optimisé pour améliorer la lisibilité, la maintenance et la performance. Des révisions ont été apportées aux classes des bots afin d'améliorer leur logique, et des lignes de code inutiles ont été supprimées pour rendre le code plus propre et plus efficace. Cette refactorisation a permis de réduire la complexité du système tout en conservant une bonne performance.

En résumé, cette version finale a permis non seulement l'implémentation complète des règles de base du jeu, mais aussi l'ajout de fonctionnalités essentielles telles que la sauvegarde des parties, un système de tests automatisés, et l'optimisation des bots grâce à l'algorithme MCTS. Cette version prépare le terrain pour de futures améliorations, telles que l'intégration de la suite des règles du jeu (difficulté Avancé, Artefacts) et la mise en place d'une interface graphique pour les joueurs.

Algorithme de Monte-Carlo

Vue d'ensemble du fonctionnement du bot avec l'algorithme de Monte Carlo (MCTS)

Le MCTS (Monte Carlo Tree Search) est un algorithme d'intelligence artificielle souvent utilisé pour prendre des décisions optimales dans des jeux. Il consiste en plusieurs étapes successives :

1. **Sélection (Select)** : À partir de l'état courant du jeu, on descend dans l'arbre de décisions déjà exploré en choisissant à chaque nœud l'action qui semble la plus prometteuse, jusqu'à arriver à un nœud qui n'est pas complètement développé (c'est-à-dire qu'il reste encore des actions possibles non explorées).

2. **Expansion (Expand)** : À partir de ce nœud partiellement exploré, on ajoute un ou plusieurs nouveaux nœuds enfants correspondant aux actions possibles non encore simulées.

3. **Simulation (Rollout)** : À partir d'un de ces nouveaux nœuds, on simule la partie de manière aléatoire ou semi-aléatoire jusqu'à un état terminal (ou jusqu'à atteindre une certaine limite de profondeur). L'idée est d'obtenir une estimation du résultat final si l'on suit la ligne de jeu représentée par ce nœud. Ces simulations "aléatoires" ne sont pas forcément réalistes, mais elles donnent une mesure statistique de la qualité de la position.

4. **Rétropropagation (Backpropagation)** : Le résultat de la simulation (par exemple, le score final obtenu par le joueur ou tout autre critère de performance) est ensuite propagé en remontant l'arbre jusqu'à la racine. Chaque nœud met à jour ses statistiques (visites, score cumulé) afin de refléter la qualité estimée de la position qu'il représente.

En répétant ce processus de nombreuses fois (les fameuses "itérations" ou "simulations"), l'arbre se peuple, et les actions les plus prometteuses ressortent statistiquement. À la fin, l'action choisie est généralement celle du nœud enfant de la racine ayant la meilleure valeur moyenne ou un ratio victoire/visite le plus élevé.

Comment cela se traduit dans le code des bots (BotC et BotD) :

- Les deux bots initialisent un **noeud racine** représentant l'état courant du jeu et le joueur actuellement en train d'agir.
- Ils lancent un grand nombre de simulations (par défaut 1000) :
    - À chaque simulation, ils sélectionnent un chemin dans l'arbre selon une fonction UCT (qui combine exploitation et exploration).
    - Ils développent ce chemin si possible (expansion).
    - Ils simulent une séquence d'actions jusqu'à un état final ou une profondeur limite.
    - Ils obtiennent un score de cette simulation (en fonction de critères déterminés dans le code).
    - Ce score est ensuite remonté dans l'arbre jusqu'à la racine (backpropagation).

Après toutes les simulations, le bot choisit l'action correspondant à l'enfant du nœud racine offrant le meilleur ratio de score moyen (winScore/visitCount).

Interactions avec les fonctions de jeu

Les fonctions internes ("simulateAction", "simulateRecruter", "simulateExplorer", etc.) ne sont pas décrites en détail dans cette explication, mais leur rôle est assez clair :

- Elles permettent, dans un environnement cloné du jeu (pour ne pas modifier l'état réel de la partie), de simuler la prise de différentes actions (recruter, explorer, commercer...) par le joueur.
- Le MCTS travaille essentiellement sur des clones de l'état de jeu afin de pouvoir explorer de multiples futurs possibles sans affecter la partie en cours.

En pratique, le bot :

- Clone l'état du jeu (Jeu) et le joueur (Joueur) pour chaque simulation.
- Exécute des actions potentielles jusqu'à un certain état.
- Évalue le résultat sous forme d'un score (points de victoire, renommée, etc.) obtenu par le joueur simulé.

C'est ce mécanisme global qui permet aux bots de tester statistiquement différentes séquences d'actions avant de choisir l'action principale de leur tour réel.

Différences entre BotC et BotD

Bien que les deux bots utilisent le même algorithme MCTS, il existe plusieurs différences notables dans leurs stratégies et leurs critères :

1. **Critère d'évaluation des simulations** :
    a. BotC semble utiliser les **points de victoire (pointVictoire)** comme principal indicateur du résultat de la simulation ("return joueur.getPointVictoire()").
    b. BotD, en revanche, utilise la **renommée (pointRenomme)** comme résultat final lors de la simulation ("return simulationPlayer.getPointRenomme();").

Cette différence d'évaluation change la façon dont chaque bot perçoit la qualité d'une action. L'un se focalise sur l'obtention de points de victoire, l'autre sur la renommée, ce qui oriente leurs décisions vers différentes priorités stratégiques.

2. **Heuristique des choix internes (hors MCTS)** :

Même si le cœur MCTS est similaire, BotD inclut des heuristiques plus complexes dans le choix des cartes (par exemple, dans le recrutement ou le choix de la destination à explorer).

BotD essaye de choisir la "meilleure" carte Viking ou la "meilleure" carte Destination à acquérir en fonction de leur gain potentiel, alors que BotC semble plus basique (souvent en prenant la première carte de la main ou la première destination exploitable).

3. **Profondeur et limites de simulation** :

Dans les deux codes, on retrouve un paramètre limitant la profondeur de simulation pour éviter les boucles infinies. Par exemple, BotC a un maxSimulationDepth de 10, tandis que BotD utilise une profondeur plus élevée (par exemple maxSimulationDepth = 40).

Cela fait que BotD explore potentiellement plus loin dans le futur, ce qui peut changer la "vision" de ses simulations et donc son évaluation des actions.

4. **Critère UCT (recherche du meilleur enfant)** :

Les deux bots utilisent le même type de sélection du meilleur enfant (UCT, avec une constante d'exploration), mais les valeurs finales et le scoring étant différents, les choix finaux en seront aussi indirectement impactés.

En résumé, la structure globale (MCTS) est la même : les deux bots construisent un arbre de recherche, simulent des actions, et choisissent la meilleure action en fonction des résultats agrégés. Néanmoins, BotC et BotD diffèrent dans :

- Leur critère de scoring (points de victoire vs renommée),
- Leurs heuristiques internes pour sélectionner certaines cartes ou actions en dehors de la simple structure MCTS,
- Leur profondeur de simulation,
- Le paramétrage (ex. ordre des actions, utilisation de pions recrue, commerce, etc.).

Ces variations entraînent des comportements différents, faisant que, malgré l'usage du même algorithme MCTS, chaque bot privilégie certaines actions et évolue différemment au cours de la partie.

2. Modélisation de l’application

Analyse des besoins

Introduction des User Stories

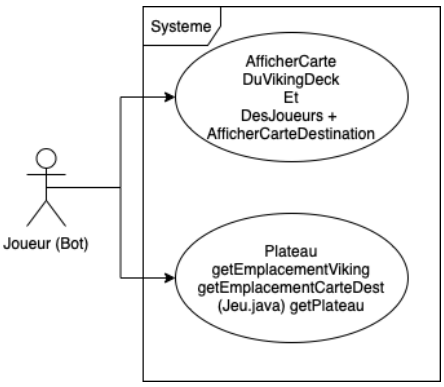
Dans la version numérique du jeu knarr, nous avions pu identifier plusieurs besoin utilisateurs qui vont être fondamental pendant le développement de notre jeu. Ces besoins utilisateurs sont la plupart des interactions utilisateurs. Ces besoins utilisateurs permettent aux joueurs, représentés par des bots, de configurer et de suivre la partie en effectuant diverse actions avec une stratégie ou pas.

Voici les besoins que nous avions identifié sous forme d’User Stories :

1. **US1** : En tant que joueur, je veux pouvoir voir le plateau avec les positions des cartes et des Vikings à travers la console ainsi que les cartes destinations.
2. **US2** : En tant que joueur, je veux pouvoir réaliser des actions (exploration, recrutement) en fonction de ma stratégie.
3. **US3** : En tant que joueur, je souhaite pouvoir piocher une carte lorsque je recrute.
4. **US4** : En tant que joueur, je veux pouvoir voir l'état de la partie en cours (score des joueurs et ressources disponibles).
5. **US5** : En tant que joueur, je souhaite pouvoir être informer de la fin de la partie pour conclure le jeu.
6. **US6 :** En tant que joueur, à chaque tour je veux pouvoir commercer avant ou après la réalisation d'une action.
7. **US7 :** En tant que joueur, je veux avoir accès aux informations de ma zone d’équipage, de ma main et de mon bateau.
8. **US8** : En tant que joueur, je veux pouvoir acheter une carte destination si je possède les ressources nécessaires à cet achat.

Scénario de Cas d'Utilisation

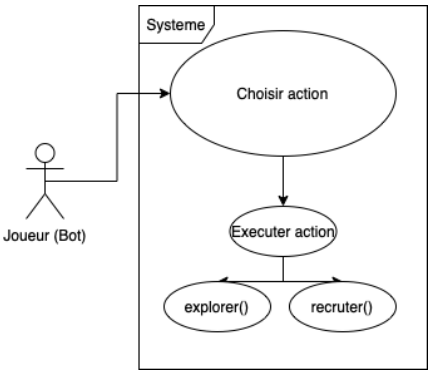
Cas d'utilisation US1 :



Acteur : Joueur (Bot)
Scénario : Dans une partie de jeu, le joueur (ou bot) souhaite consulter l'état du jeu pour prendre une décision stratégique. À tout moment pendant son tour, il peut demander à voir :

1. Tous les cartes dans le Viking disponible sur le plateau.
2. Les cartes destination disponibles sur le plateau.

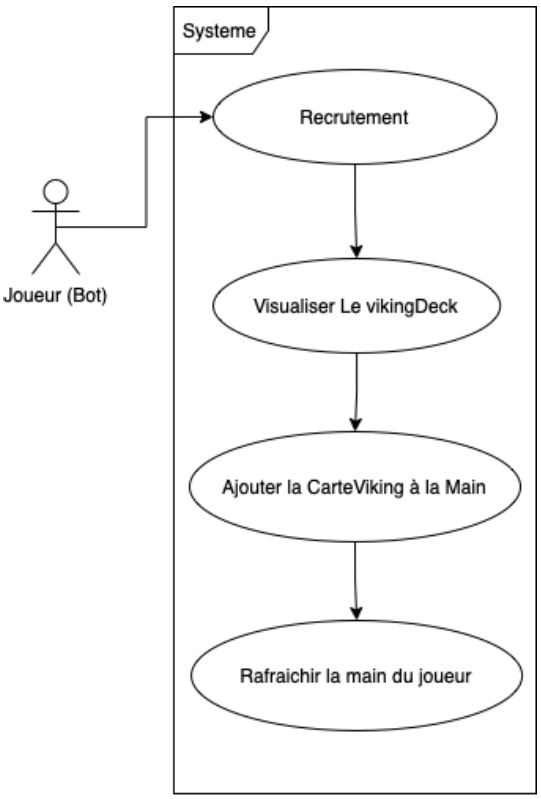
Cas d'utilisation US2 :



Acteur : Joueur (Bot)
Scénario : Pendant son tour de jeu, le joueur (ou bot) doit décider de l'action à entreprendre en fonction de sa stratégie :

1. **Choisir Action :**
    - Le joueur (ou bot) commence par sélectionner une action pour ce tour via la méthode choisirAction(), qui analyse les options disponibles.
2. **Exécuter Action :**
    - Le système reçoit l'action choisie et passe à l'exécution via executerAction().
3. **Explorer ou Recruter :**
    - Le système appelle ensuite l'une des deux méthodes, selon le choix du joueur :
        i. **explorer()**
        ii. **recruter()**

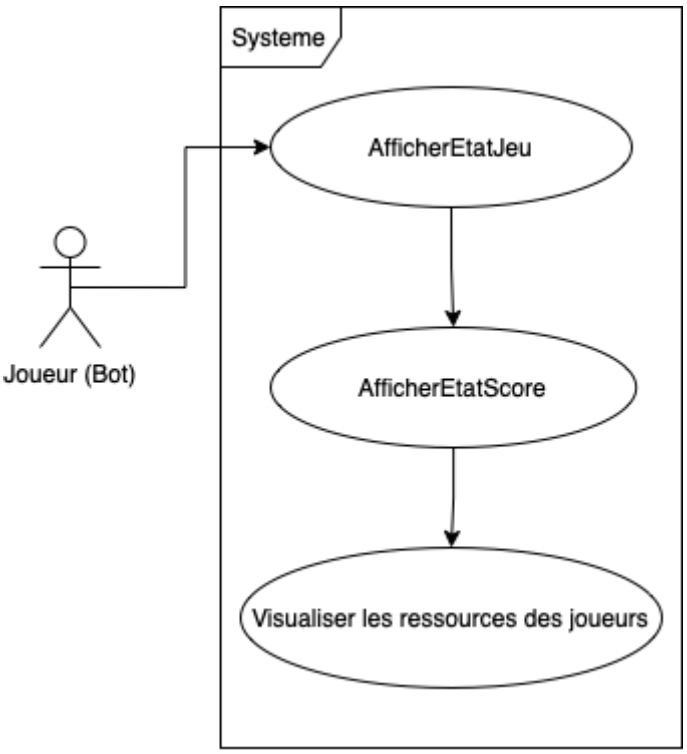
Cas d’utilisation US3 :



Acteur : Joueur (Bot)
Scénario : Lorsque le joueur (ou bot) choisit d'effectuer une action de recrutement, le système suit les étapes suivantes :

1. **Recrutement :**
    a. Le joueur déclenche l'action de recrutement via la méthode recruter().
2. **Visualiser le Viking Deck :**
    a. Le système affiche le contenu du Viking Deck pour montrer les cartes disponibles.
3. **Ajouter la Carte Viking à la Main :**
    a. Le système tire une carte du Viking Deck et l'ajoute à la main du joueur en utilisant ajouterCarteVikingMain().
4. **Rafraîchir la Main du Joueur :**
    a. Le système met à jour l'état de la main du joueur, reflétant l'ajout de la nouvelle carte.

Cas d'utilisation US4 :

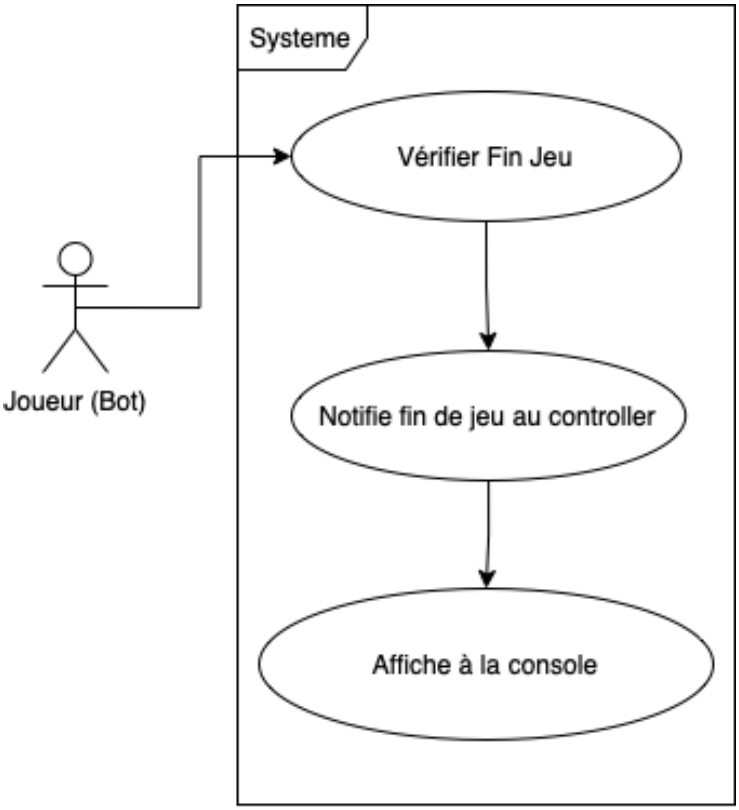


Acteur : Joueur (Bot)

Scénario : Le joueur (ou bot) souhaite consulter l'état de la partie pour évaluer sa position et ajuster sa stratégie. Voici les étapes du scénario :

1. **Afficher État du Jeu :**
    a. Le joueur demande au système d'afficher l'état actuel de la partie via afficherEtatJeu().
2. **Afficher État des Scores :**
    a. Le système affiche ensuite les scores des différents joueurs, obtenus via getScoresPublics().
3. **Visualiser les Ressources des Joueurs :**
    a. Enfin, le système affiche les ressources disponibles pour chaque joueur (cartes Viking, pions recrue, etc.), en utilisant les méthodes d'accès des ressources (getCartesVikingMain(), getPionsRecrue(), etc.).

Cas d'utilisation US5 :

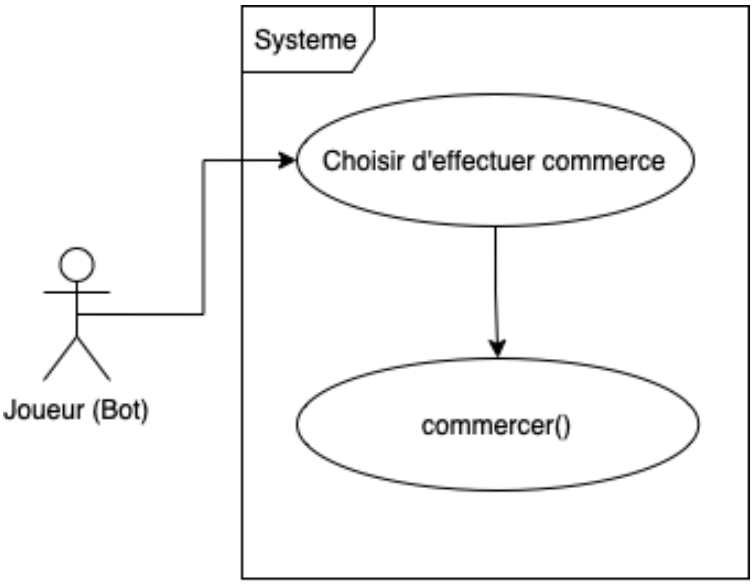


Acteur : Joueur (Bot)

Scénario : Le joueur (ou bot) souhaite savoir si la partie est terminée. Le système procède comme suit :

1. **Vérifier Fin de Jeu :**
    - Le système vérifie si les conditions de fin de jeu sont remplies en appelant la méthode verifierFinJeu().
2. **Notifier Fin de Jeu au Contrôleur :**
    - Si la partie est terminée, le système notifie le contrôleur dans la classe JeuController.
3. **Affiche à la Console :**
    - Le système alerte avec un message aux joueurs que c'est le dernier tour. On utilise le loggerController

Cas d’utilisation US6 :

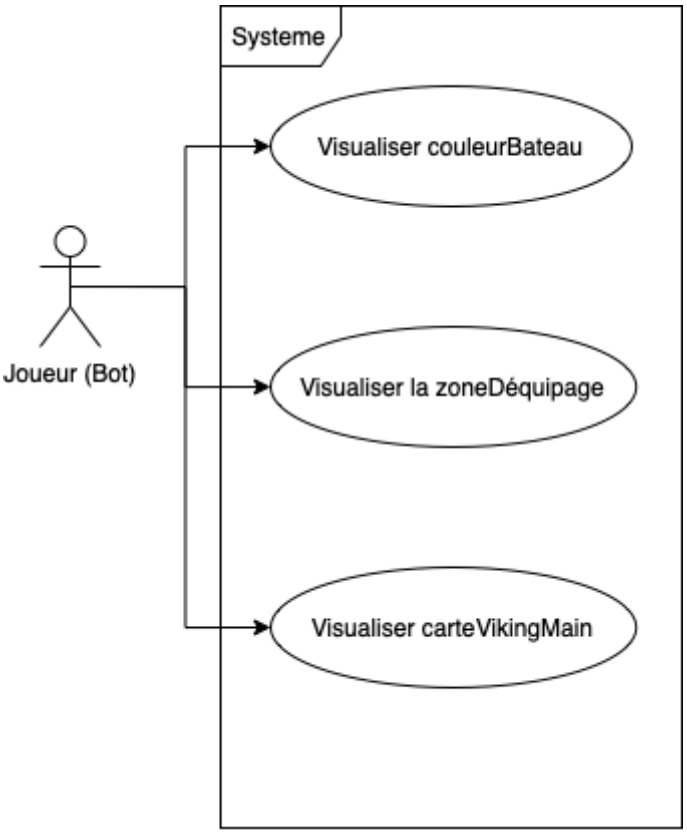


Acteur : Joueur (Bot)

Scénario : Durant son tour de jeu, le joueur (ou bot) a la possibilité de commercer avant ou après son action principale. Le système suit le processus suivant :

1. **Choisir d'Effectuer le Commerce :**
    - Le joueur évalue ses ressources et décide s'il souhaite effectuer une action de commerce en appelant deciderEtEffectuerCommerce().
2. **Commercer :**
    - Si le joueur décide de commercer, le système exécute l'action via la méthode commercer(), qui gère l'utilisation des ressources (pions bracelet) et applique les gains obtenus.

Cas d’utilisation US7 :



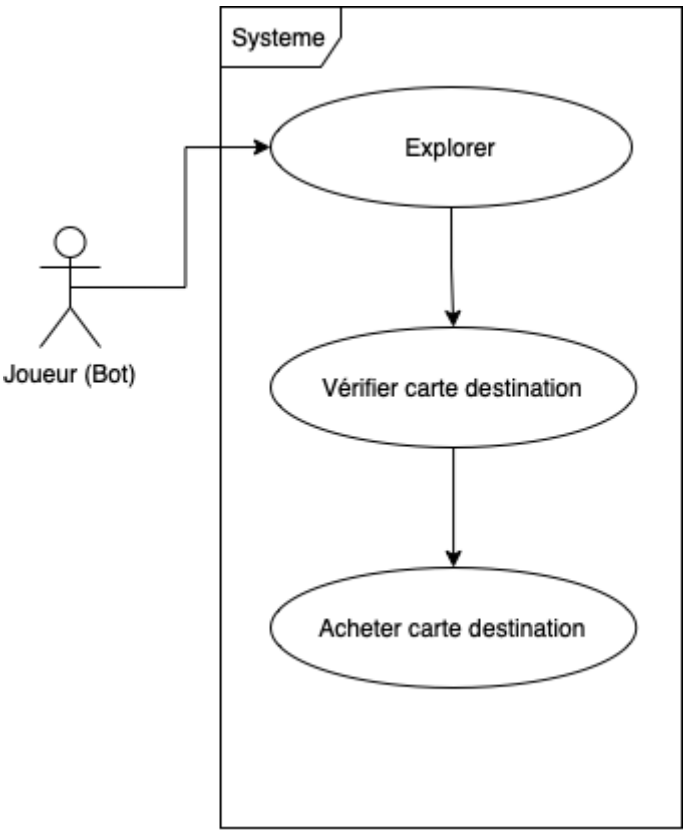
Acteur : Joueur (Bot)

Scénario :

Le joueur (ou bot) souhaite accéder aux informations concernant ses propres ressources pour évaluer sa situation. Le système suit les étapes suivantes :

1. **Visualiser Couleur du Bateau :**
    a. Le joueur demande à connaître la couleur de son bateau via la méthode getCouleurBateau(). Le système affiche cette information.
2. **Visualiser la Zone d'Équipage :**
    a. Le joueur consulte ensuite les détails de sa zone d’équipage à l'aide de la méthode getZoneEquipage(), qui renvoie une liste des cartes Viking dans cette zone.
3. **Visualiser les Cartes en Main :**
    a. Enfin, le joueur demande à voir les cartes Viking qu’il a en main via getCartesVikingMain(), et le système affiche cette information.

Cas d’utilisation US8 :



Acteur : Joueur (Bot)

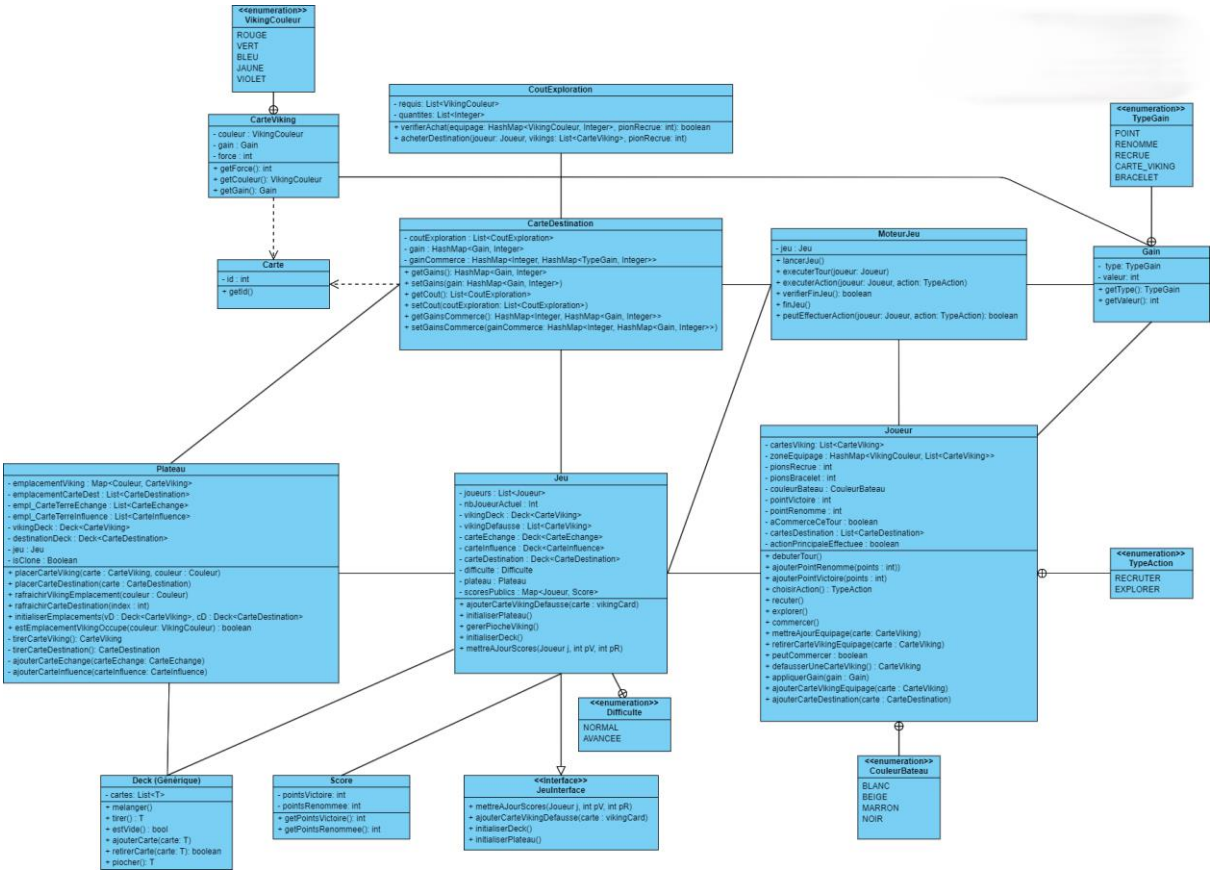
Scénario :

Le joueur (ou bot) souhaite acheter une carte destination s'il possède les ressources nécessaires. Le système suit les étapes suivantes :

1. **Explorer :**
    a. Le joueur décide d'explorer une carte destination en appelant la méthode explorer().
2. **Vérifier Carte Destination :**
    a. Le système vérifie si le joueur a les ressources requises (cartes Viking, pions recrue, etc.) via la méthode acquerirCarteDestination(). Si les conditions sont remplies, le joueur peut procéder à l'achat.
3. **Acheter Carte Destination :**
    a. Le système déduit les ressources nécessaires et ajoute la carte destination à la collection du joueur.

Conception logicielle

Point de vue statique – Diagramme de classes



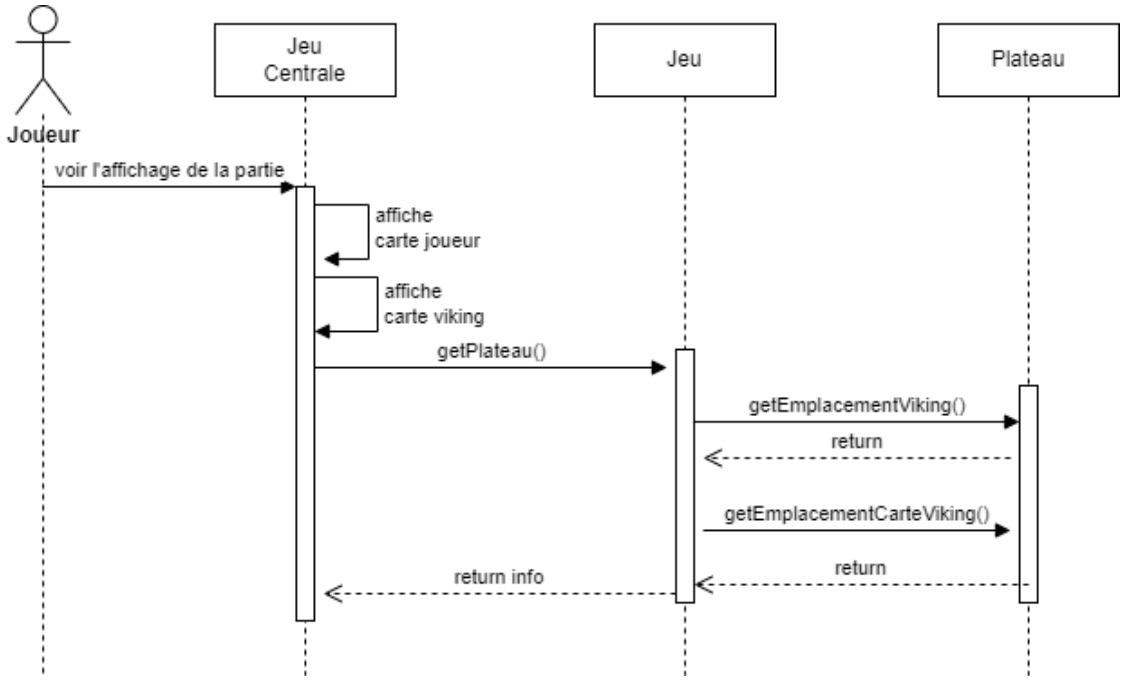
Le diagramme de classes représente une conception orientée objet claire pour gérer les différentes parties du jeu. On retrouve les éléments principaux comme les cartes, les joueurs, le plateau, et le moteur de jeu, chacun étant représenté par des classes spécifiques. Les cartes sont organisées de façon logique : *CarteViking* et *CarteDestination* héritent de la classe abstraite *Carte*, ce qui simplifie leur gestion tout en permettant de facilement ajouter de nouveaux types de cartes si besoin.

La classe *Joueur* modélise l'état et les actions des joueurs, incluant leurs cartes et ressources. Elle définit aussi les méthodes nécessaires pour effectuer les actions classiques du jeu, comme recruter ou explorer. Le *MoteurJeu*, quant à lui, assure le déroulement des phases de jeu, tandis que la classe *Jeu* maintient l'état général et dirige les interactions entre les joueurs et le plateau.

Le *Plateau* centralise les emplacements des cartes et permet de manipuler celles-ci de manière ordonnée, tandis que la classe générique *Deck* gère le tirage et le mélange des cartes. L'ensemble du diagramme montre une approche modulaire, avec une organisation claire et des relations bien pensées entre les classes. Cette conception permet une gestion efficace des différentes composantes du jeu.

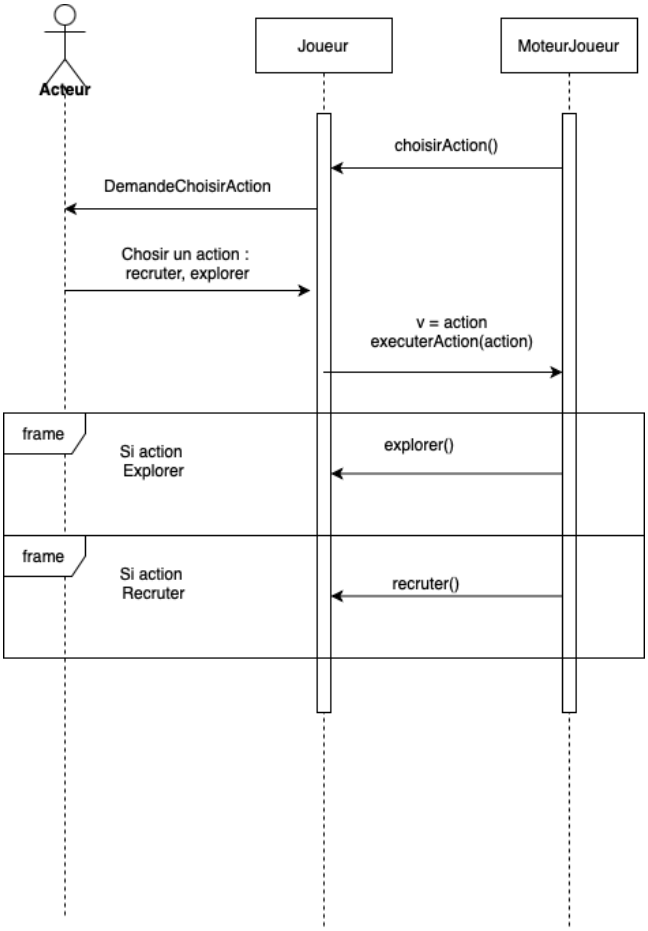
Point de vue dynamique – Diagramme de séquence

- User Story 1 :



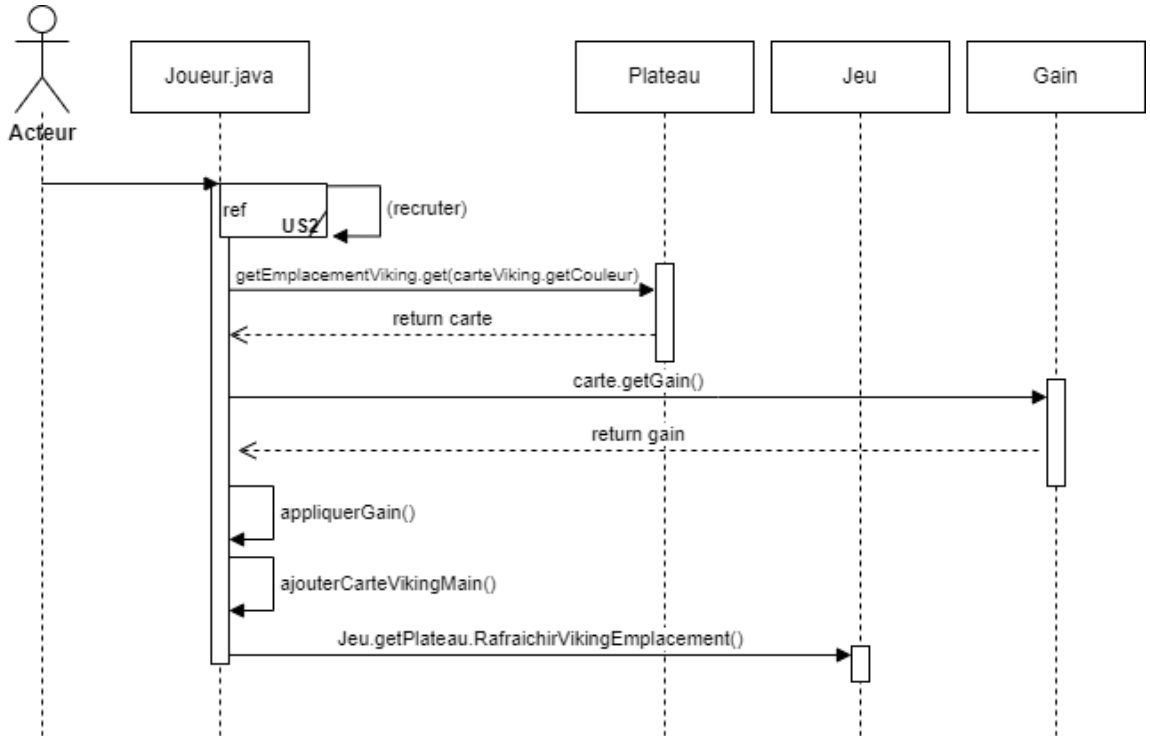
Le diagramme de séquence montre comment un joueur demande à voir l'état de la partie. Cette requête est envoyée à *Jeu Centrale*, qui commence par afficher les cartes du joueur et les cartes des Vikings. Pour obtenir les données du plateau, *Jeu Centrale* appelle la méthode de *Jeu*, qui récupère les emplacements des cartes et des Vikings auprès du *Plateau*. Les informations sont ensuite renvoyées à *Jeu Centrale*, qui les affiche au joueur de manière complète et claire.

- User Story 2 :



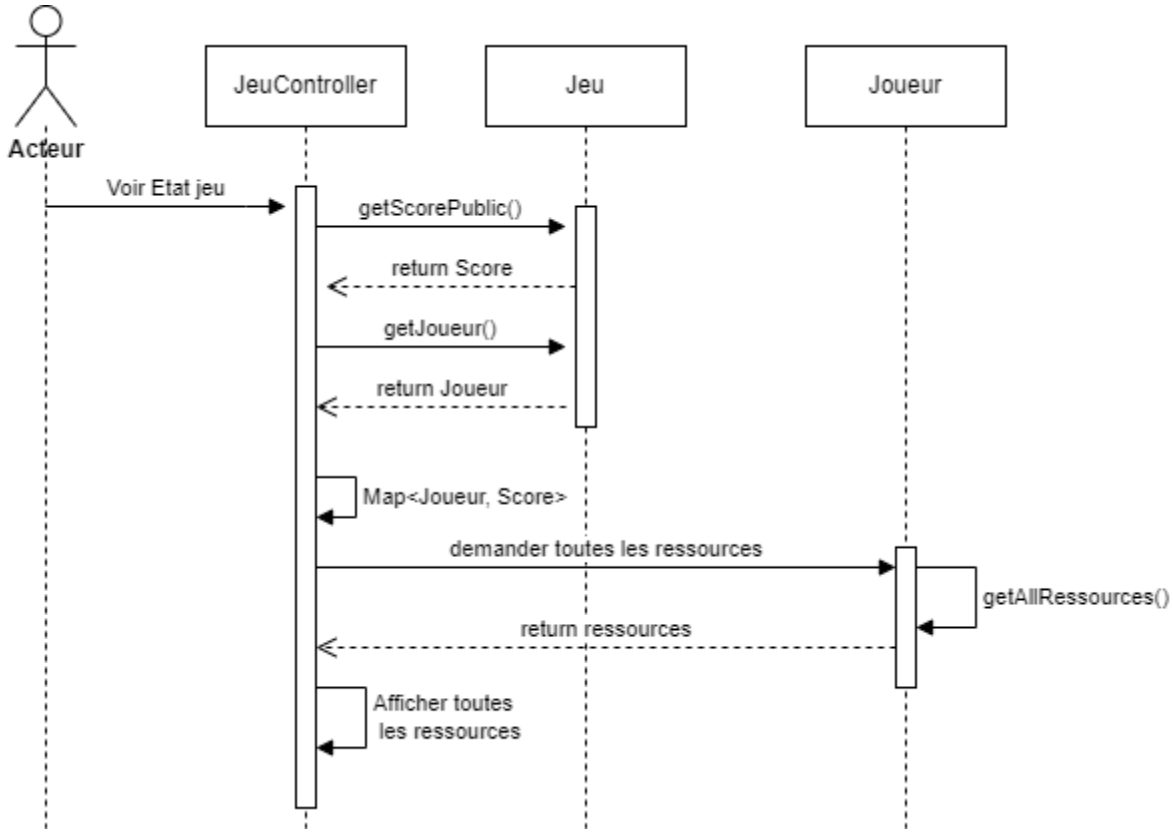
Ce diagramme de séquence montre comment l'acteur demande à un joueur de choisir une action dans le jeu, soit *recruter* soit *explorer*. Le joueur utilise la méthode *choisirAction()* pour sélectionner une action, puis envoie cette action au *MoteurJoueur* avec la méthode *executerAction(action)*. Le moteur vérifie l'action choisie : si c'est *explorer*, la méthode *explorer()* est appelée, et si c'est *recruter*, la méthode *recruter()* est appelée. Cette séquence illustre la logique de prise de décision du joueur et l'exécution des actions correspondantes par le moteur du jeu.

- User Story 3 :



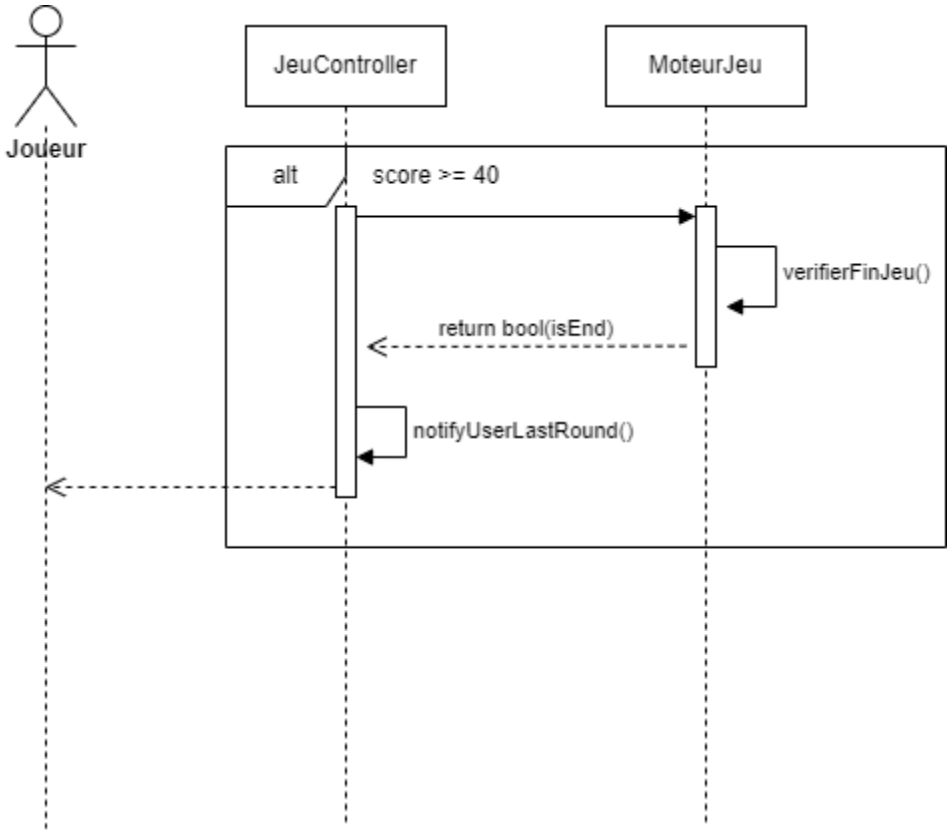
Le diagramme montre l'action *recruter* : l'acteur initie la demande via *Joueur.java*, qui obtient une carte Viking du Plateau selon sa couleur. Ensuite, *Joueur.java* récupère le gain de la carte, applique ce gain, ajoute la carte à la main du joueur, et appelle Jeu pour rafraîchir l'emplacement sur le plateau. Cette séquence décrit le déroulement complet de l'action de recrutement dans le jeu.

- User Story 4 :



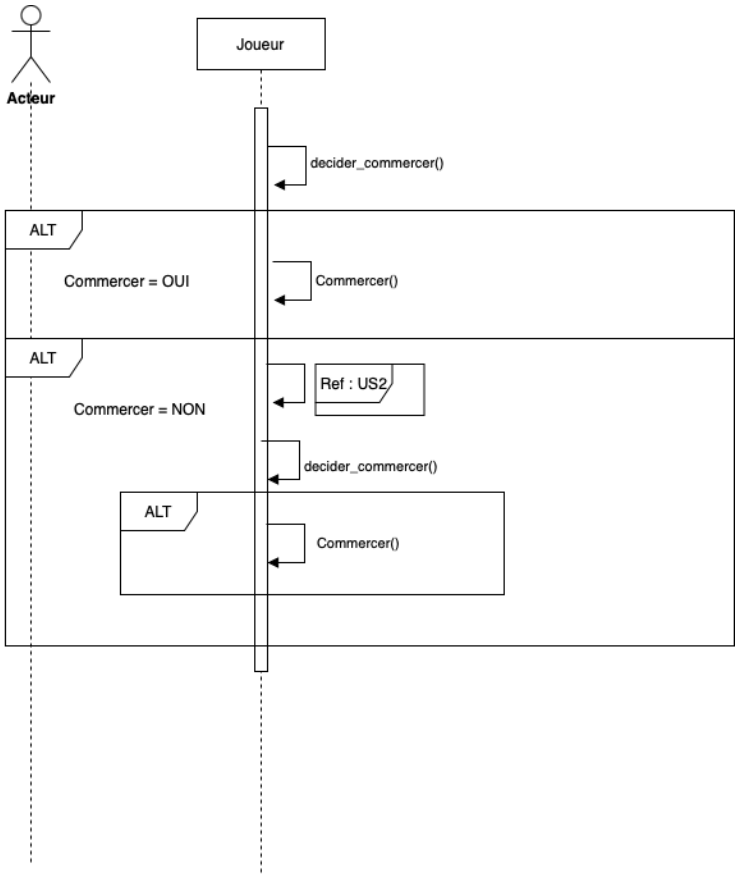
Ce diagramme de séquence illustre la demande de l'acteur pour voir l'état du jeu. L'acteur interroge *JeuController*, qui appelle *Jeu* pour obtenir le score public (*getScorePublic())* et le joueur courant (*getJoueur()*). Ensuite, *JeuController* demande toutes les ressources au joueur via la méthode *getAllRessources*(). Les ressources sont renvoyées et affichées par *JeuController*, offrant ainsi une vue complète de l'état actuel du jeu.

- User Story 5 :



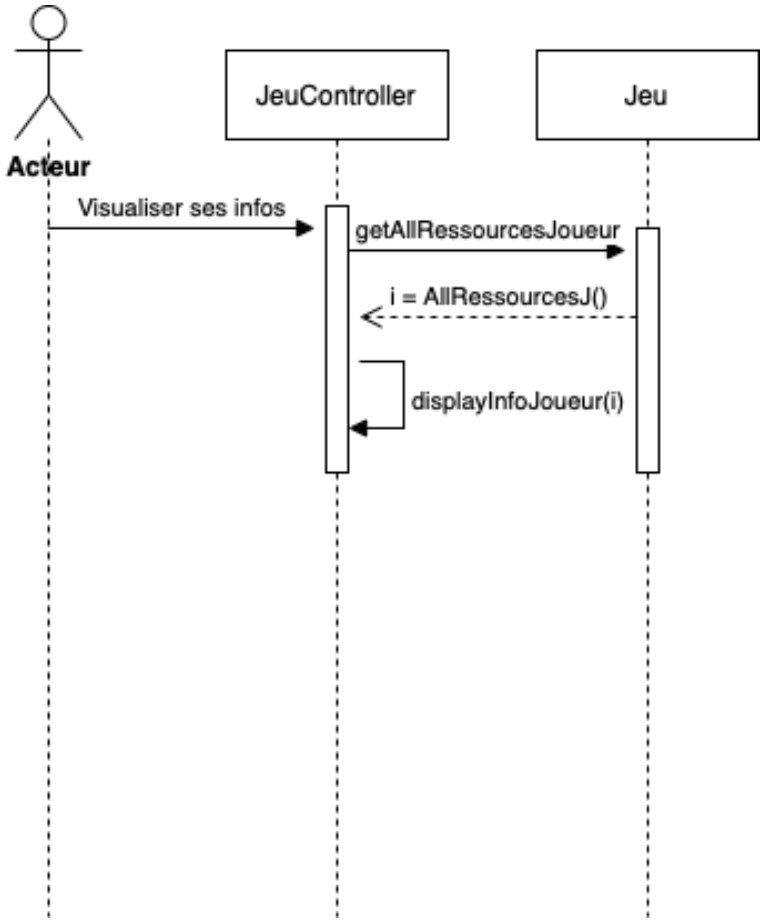
Ce diagramme de séquence montre la vérification de la fin du jeu. Si le score atteint ou dépasse 40, *JeuController* appelle la méthode *verifierFinJeu*() du *MoteurJeu*. Le résultat (*isEnd*) est renvoyé sous forme de booléen. Si la fin est confirmée, *JeuController* notifie le joueur du dernier tour via *notifyUserLastRound*(). Cette séquence assure que le joueur est informé quand le jeu touche à sa fin.

- User Story 6 :



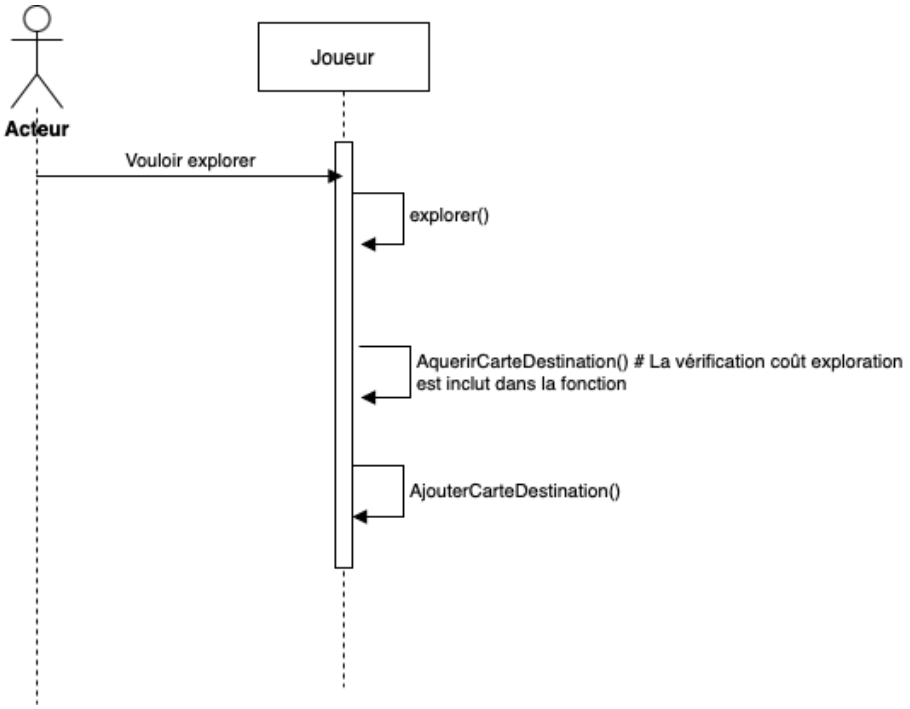
Le diagramme de séquence montre la décision du joueur de commercer. L'acteur appelle *decider_commercer*(). Si le joueur choisit de commercer, la méthode *Commercer*() est exécutée. Sinon, le flux passe à une action référencée par l'US2 (exploration ou recrutement), avant de vérifier de nouveau la possibilité de commercer. Si le joueur change d'avis, *Commercer*() est appelée. Le diagramme illustre ainsi le choix du joueur pour l'action de commerce.

- User Story 7 :



Ce diagramme de séquence montre comment un acteur visualise ses informations. L'acteur demande à *JeuController* d'afficher ses ressources via *getAllRessourcesJoueur*(). *JeuController* interroge *Jeu*, qui retourne toutes les ressources du joueur. Ces informations sont ensuite envoyées à *JeuController*, qui les affiche avec *displayInfoJoueur*(). Ce processus permet à l'utilisateur de consulter l'état de ses ressources en jeu.

- User Story 8 :



Ce diagramme de séquence montre l'action *explorer* initiée par l'acteur. L'acteur demande à *Joueur* d'explorer via la méthode *explorer*(). Ensuite, *Joueur* appelle *AcquerirCarteDestination*(), qui inclut la vérification du coût d'exploration. Si cette vérification est réussie, la méthode *AjouterCarteDestination*() est exécutée pour ajouter la carte destination au joueur. Cette séquence décrit l'enchaînement des actions lors de l'exploration d'une carte dans le jeu.

3. Conclusion

Analyse de la solution

Points forts

• **Modularité et architecture** : La solution adoptée présente une architecture bien structurée, facilitant la maintenance et l'évolution du projet. Le découpage en classes et en fonctionnalités distinctes a permis une meilleure lisibilité et une maintenance facilitée. Bien que le projet n'utilise pas d'interface graphique, l'intégration des fonctionnalités dans des composants distincts (comme les classes des entités et des bots) garantit une séparation des responsabilités.

• **Respect des principes SOLID** : L'application des principes SOLID a amélioré la qualité du code. Les classes sont modulaires, bien définies, et respectent les responsabilités uniques. Cela a permis une meilleure évolutivité du projet tout en limitant le couplage entre les différents composants.

• **Conception avec UML** : Les diagrammes UML réalisés au Sprint 1 ont joué un rôle clé dans la planification du projet. Ils ont offert une vision claire des entités principales et de leurs interactions. Le travail préliminaire sur les diagrammes de classes et d'activités a permis une implémentation plus fluide des fonctionnalités de base.

• **Format JSON pour les données** : L'utilisation du format JSON pour initialiser les cartes et configurer le jeu a permis une configuration flexible et facilement modifiable. Ce format standardisé favorise la compatibilité et simplifie l'ajout de nouvelles cartes ou paramètres de jeu.

• **Automatisation des tests** : Le système d'automatisation des parties facilite les tests en permettant de vérifier rapidement la fiabilité des bots et du moteur de jeu. Cela a permis de s'assurer du respect des règles du jeu et de détecter les anomalies rapidement.

• **Fiabilité des bots A et B** : Les bots A et B se sont montrés performants et fiables dans leurs stratégies. Ils respectent les règles du jeu et fonctionnent correctement dans la majorité des scénarios testés.

Points faibles

• **Cas complexes nécessitant plus de tests** : Certains cas complexes, comme les interactions avancées entre les joueurs (notamment l'action "Commercer"), nécessitent encore des tests plus approfondis. Cela met en avant des lacunes dans la couverture des tests pour des scénarios complexes.

• **Problèmes d'optimisation des classes** : Bien que le projet respecte les principes SOLID, certaines classes nécessitent des optimisations. Par exemple, la gestion des bots C et D présente une consommation excessive de ressources, ce qui affecte la performance globale du moteur de jeu. Une optimisation du code et une révision des algorithmes sont nécessaires pour améliorer leur efficacité.

• **Gestion des données JSON** : Durant le Sprint 3, des difficultés ont été rencontrées avec la gestion des fichiers JSON, notamment en raison d'erreurs de parsing et de problèmes d'intégrité des données. Cela a entraîné des retards et nécessité des ajustements pour garantir un chargement correct des cartes et des paramètres du jeu.

• **Forte consommation pour Bot C et Bot D** : Les bots C et D présentent une consommation excessive de ressources, ce qui impacte les performances globales du moteur de jeu. Cette consommation élevée est due à une logique inefficace et à un manque d'optimisation des algorithmes. Une refonte des stratégies et une optimisation du code permettraient de réduire cette consommation tout en améliorant l'efficacité des bots.

Suite du développement

Le projet a déjà fait l'objet d'un travail de conception et d'implémentation solide, mais plusieurs fonctionnalités doivent encore être ajoutées ou améliorées pour rendre le jeu plus complet et plus performant.

1. Optimisation des bots :

- **Réduction de la consommation des ressources pour les bots C et D :** Les bots C et D consomment actuellement beaucoup de ressources, ce qui impacte la performance du jeu. Il est nécessaire de revoir leurs algorithmes afin de les rendre plus efficaces, notamment en optimisant leurs stratégies et en évitant les traitements lourds inutiles. Des ajustements sur la gestion de leur état ou de leur cycle d'actions pourraient être envisagés pour alléger leur impact sur les ressources.
- **Amélioration des stratégies des bots :** Bien que l'algorithme MCTS ait déjà été implémenté pour les bots C et D, il pourrait être optimisé davantage pour améliorer la prise de décision et la performance des bots. Cela pourrait inclure des ajustements dans la manière dont les simulations sont réalisées, l'ajout de mécanismes pour réduire le temps de calcul ou de stratégies permettant de supprimer les branches inutiles de l'arbre de recherche. L'objectif serait d'améliorer l'efficacité du calcul tout en maintenant la qualité de la prise de décision dans des situations complexes, rendant les bots plus compétitifs et réactifs face aux joueurs.

2. Refactoring et amélioration de la structure du code :

- **Révision des classes liées aux bots :** Les classes des bots, en particulier celles des bots avancés (BOTC et BOTD), nécessitent une révision pour améliorer leur efficacité. Une analyse approfondie de la gestion de l'état du jeu et des interactions stratégiques permettrait d'optimiser la logique de prise de décision et d'adapter les comportements des bots pour qu'ils consomment moins de ressources tout en conservant leur complexité stratégique. L'objectif serait de rendre les bots plus réactifs et intelligents sans impacter la fluidité du jeu.

- **Amélioration de la gestion des données JSON :** Bien que la gestion des fichiers JSON pour l'initialisation des cartes et des paramètres soit déjà en place, des problèmes de parsing et d'intégrité des données ont été rencontrés. Il serait bénéfique de réviser cette gestion pour garantir que toutes les données, y compris les cartes et autres ressources du jeu, soient toujours correctement chargées, avec une attention particulière portée à l'intégrité des fichiers et à la gestion des erreurs.

3. Tests et validation :

- **Automatisation des tests pour de nouveaux cas :** Des tests automatisés ont déjà été réalisés pour simuler des parties, mais l'ajout de nouveaux cas de test est nécessaire pour valider les fonctionnalités récemment ajoutées, telles que l'affichage des scores, le commerce, et les actions de fin de partie. Ces tests permettront de garantir que le jeu fonctionne correctement dans une variété de scénarios complexes.

- **Tests de performance :** Des problèmes de performance liés à la consommation des ressources par les bots ont été identifiés. Il est donc essentiel de réaliser des tests de performance approfondis pour repérer les zones de ralentissement et optimiser les parties du code qui nuisent à la fluidité du jeu. Ces tests permettront également d'assurer une exécution stable lors de longues sessions de jeu.

4. Amélioration de l'expérience utilisateur :

- **Interface utilisateur (UI) :** Bien que le projet n'ait pas encore d'interface graphique, l'ajout d'une interface utilisateur de base pourrait grandement améliorer l'expérience de jeu. Une interface simple, qui afficherait l'état du jeu, les scores des joueurs et les ressources disponibles, rendrait le jeu plus interactif et plus facile à suivre, en particulier pour les joueurs non familiers avec la logique du jeu.

- **Documentation et tutoriel :** La création d'une documentation détaillée, accompagnée d'un tutoriel pour expliquer les règles et les mécanismes du jeu, serait bénéfique pour rendre le jeu plus accessible. Cela aiderait notamment les nouveaux joueurs à mieux comprendre les règles et à démarrer plus facilement, contribuant ainsi à une expérience de jeu plus fluide et plus agréable.

Pour continuer à faire avancer le projet, il est essentiel de se concentrer sur l'optimisation des performances. En particulier, les bots doivent être améliorés pour devenir plus stratégiques et moins consommateur en ressources. Un refactoring du code, une gestion plus robuste des données et des tests supplémentaires contribuera à renforcer la fiabilité et la fluidité du jeu. L'ajout d'une interface utilisateur simple et d'une documentation complète améliorera l'accessibilité et l'interaction, offrant ainsi une meilleure expérience aux utilisateurs.