MIAGE RÉSEAU DES MIAGE DE FRANCE Nice

Rapport intermédiaire

PROJET INFORMATIQUE KNARR – DONATI LEO

DOGLIOLI-RUPPERT 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.

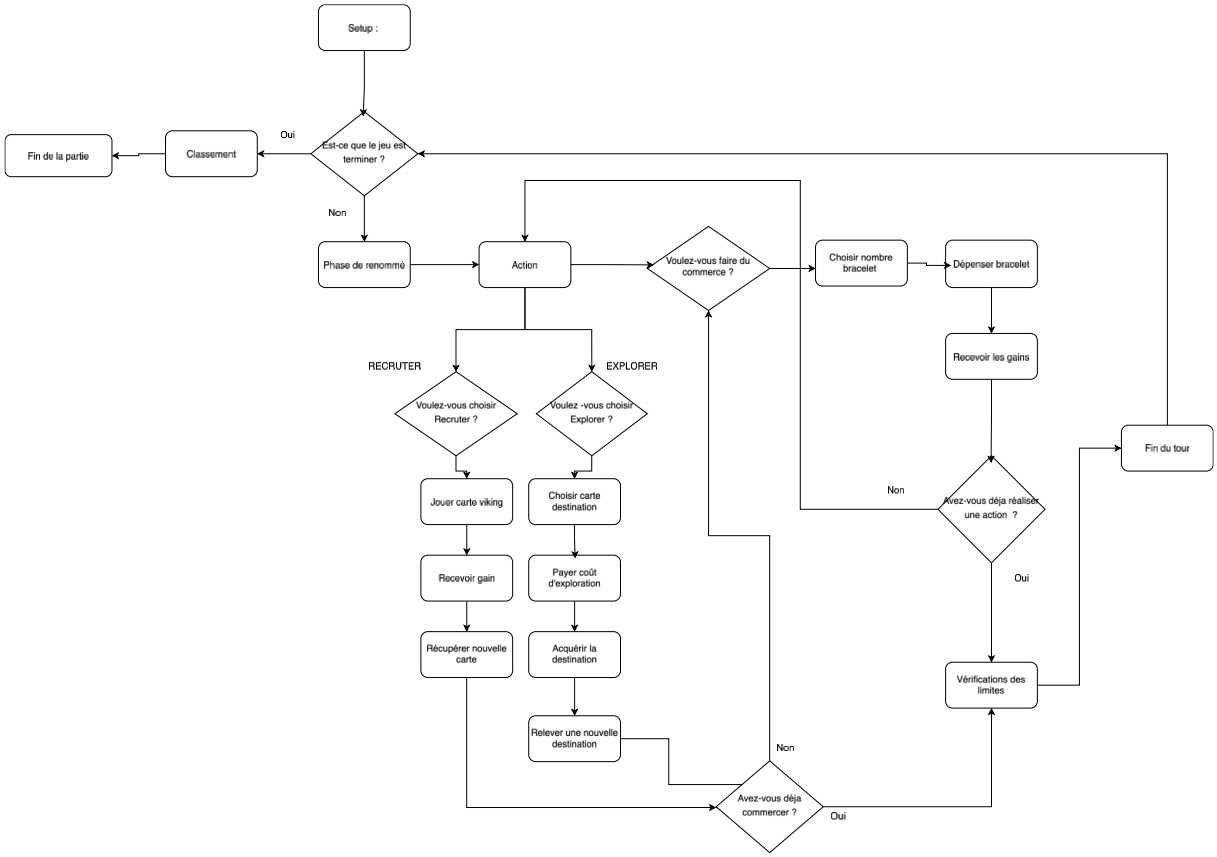
Ressources : Cartes ou pions possédés par le joueur, utilisés pour effectuer des actions ou acquérir des cartes spéciales.

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

Les fonctionnalités que nous avons implémentées dans le jeu jusqu'à présent incluent l'initialisation des cartes et des ressources pour les joueurs. Nous avons mis en place le processus de distribution des cartesViking et des cartesDestination, avec trois cartesViking assignées à chaque joueur, et chaque joueur reçoit une CarteBateau contenant un pion Recrut et un pion Bracelet, assurant ainsi la distribution des ressources de départ. Nous avons également initialisé la main du joueur, qui lui permet de conserver les trois cartes, ainsi que le VikingDeck, représentant le paquet de cartesViking sur le plateau. L'étape d'initialisation se déroule correctement et peut être validée comme fonctionnelle.

En ce qui concerne les actions de jeu, les joueurs (bots non intelligents) peuvent effectuer certaines actions, notamment le "Recrut", c’est-à-dire le processus de pioche d’une carteViking. La fonctionnalité "Explorer" est également en place, bien qu’elle présente une erreur partiellement corrigée nécessitant un ajustement. Ces fonctionnalités ont été testées uniquement pour le premier tour de jeu (tour n°1).

Nous avons également intégré une fonctionnalité de récupération des données en format JSON à l’aide de la bibliothèque Jackson, permettant d’intégrer et de manipuler des données spécifiques aux cartes, telles que la couleur et les gains associés. Ces informations enrichissent les cartes en apportant des caractéristiques authentiques et détaillées. Actuellement, les véritables cartes sont intégrées uniquement pour l’action de recrutement, garantissant que chaque pioche de carteViking reflète les données réelles des cartes.

2. Modélisation de l'application

Analyse des besoins

Introduction des User Stories

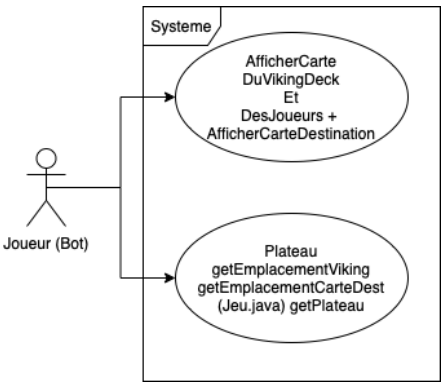
Dans la version numérique du jeu knarr, nous avions pu identifier plusieurs besoin utilisateurs qui vont être fondamental pendant le développement de notre jeu. Ces besoins utilisateurs sont la plupart des interactions utilisateurs. Ces besoins utilisateurs permettent aux joueurs, représentés par des 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

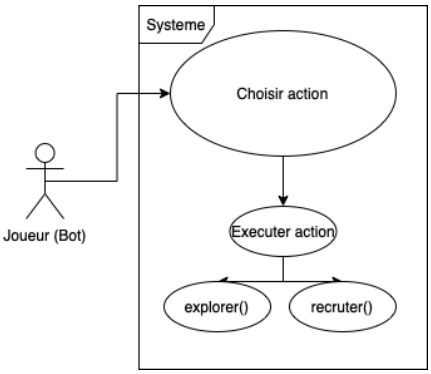
Cas d'utilisation US1 :



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.

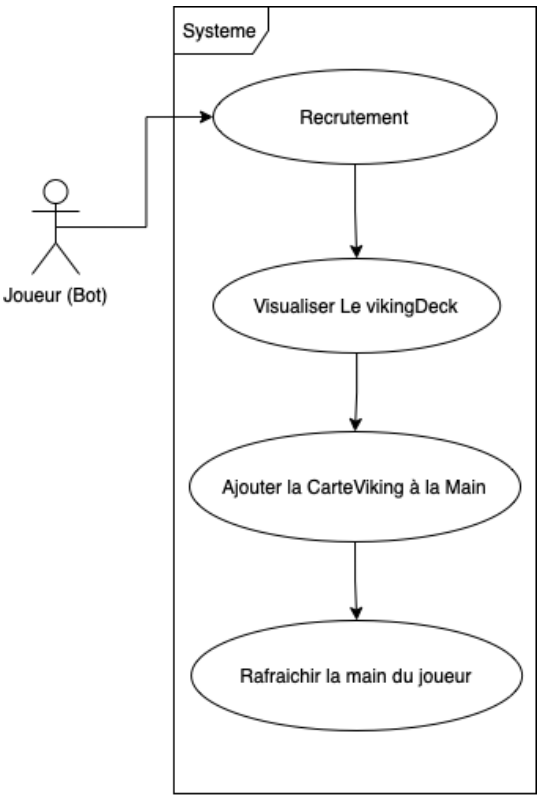
Cas d'utilisation US2 :



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()**

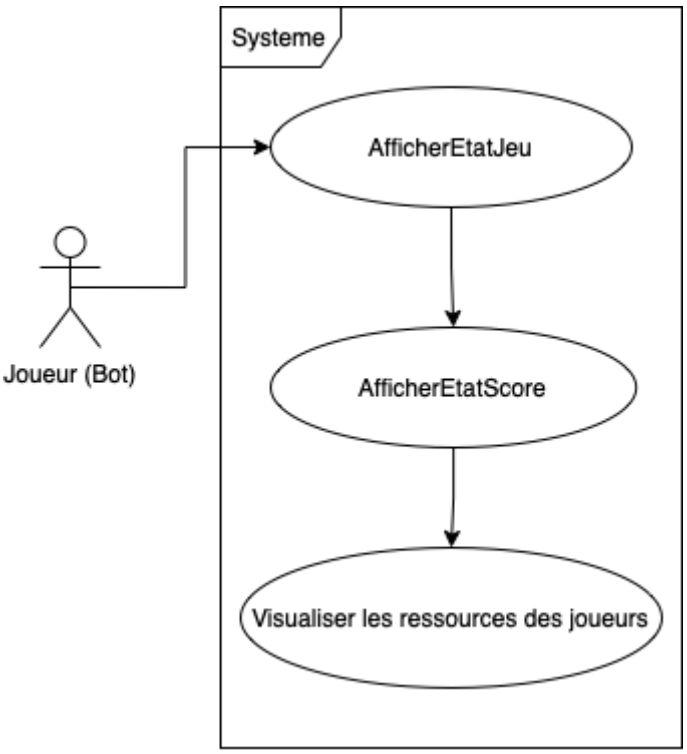
Cas d’utilisation US3 :



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 :

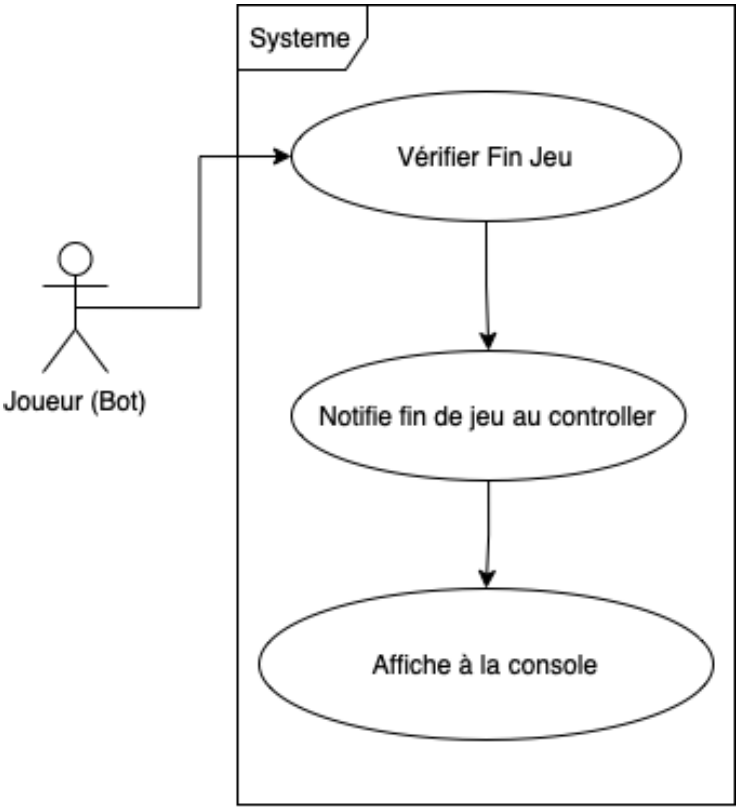


Acteur : Joueur (Bot)

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 :

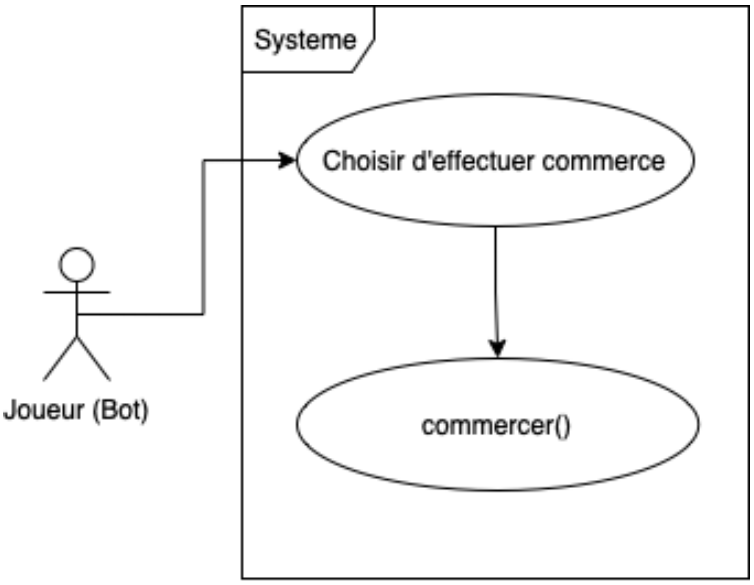


Acteur : Joueur (Bot)

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 :

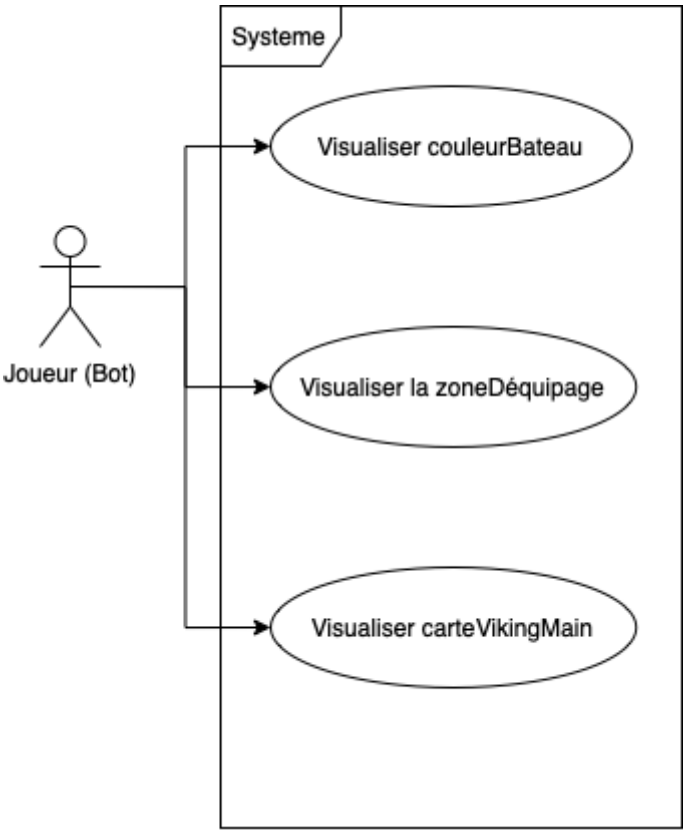


Acteur : Joueur (Bot)

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 :



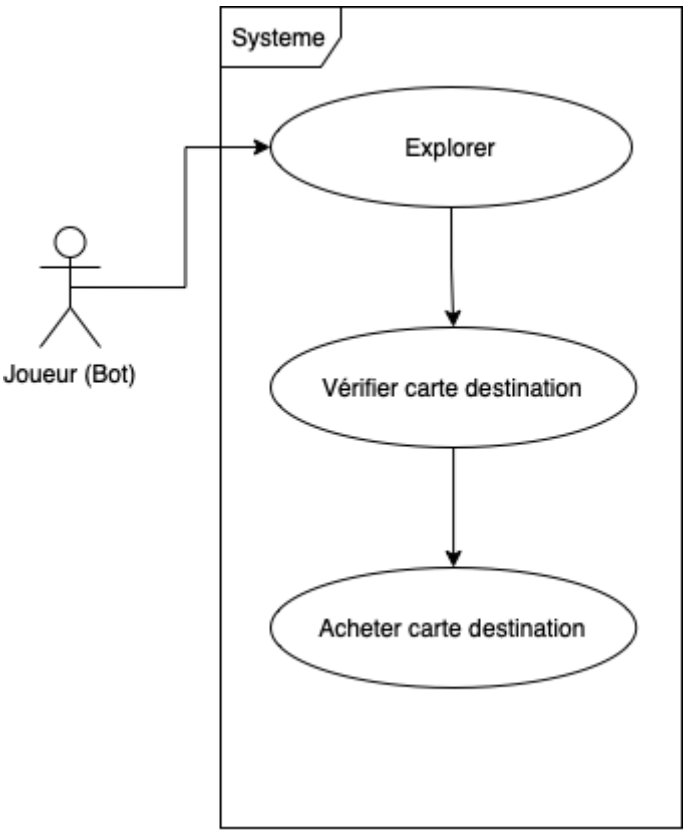
Acteur : Joueur (Bot)

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)

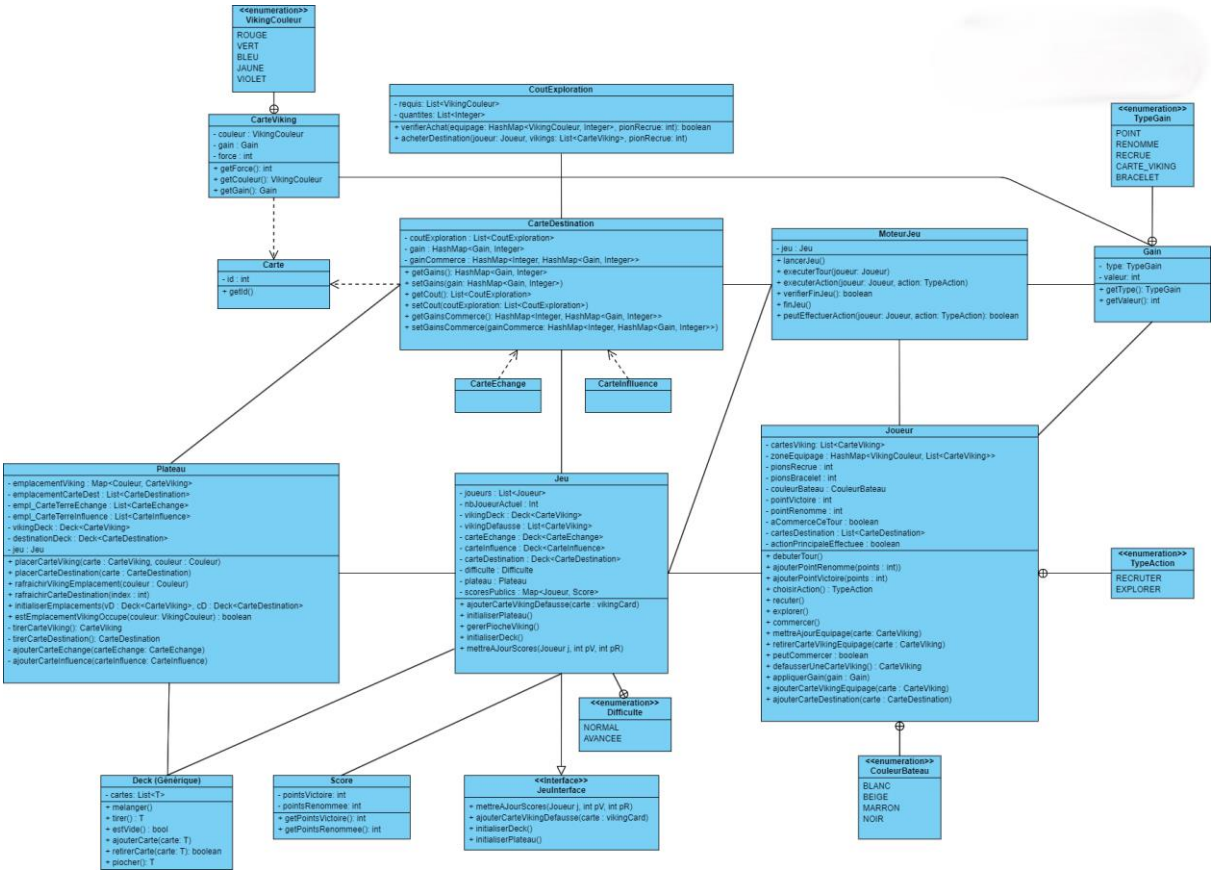
Scénario :

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



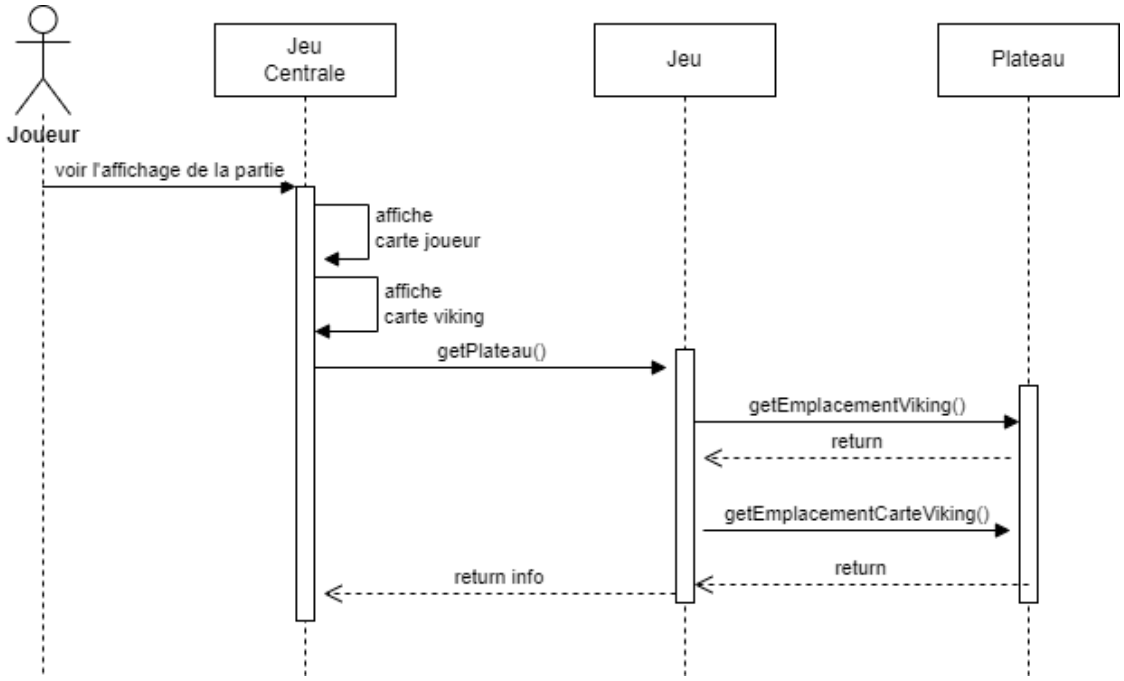
Le diagramme de classes représente une conception orientée objet claire pour gérer les différentes parties du jeu. On retrouve les éléments principaux comme les cartes, les joueurs, le plateau, et le moteur de jeu, chacun étant représenté par des classes spécifiques. Les cartes sont organisées de façon 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, mais laisse également de la place pour des évolutions futures.

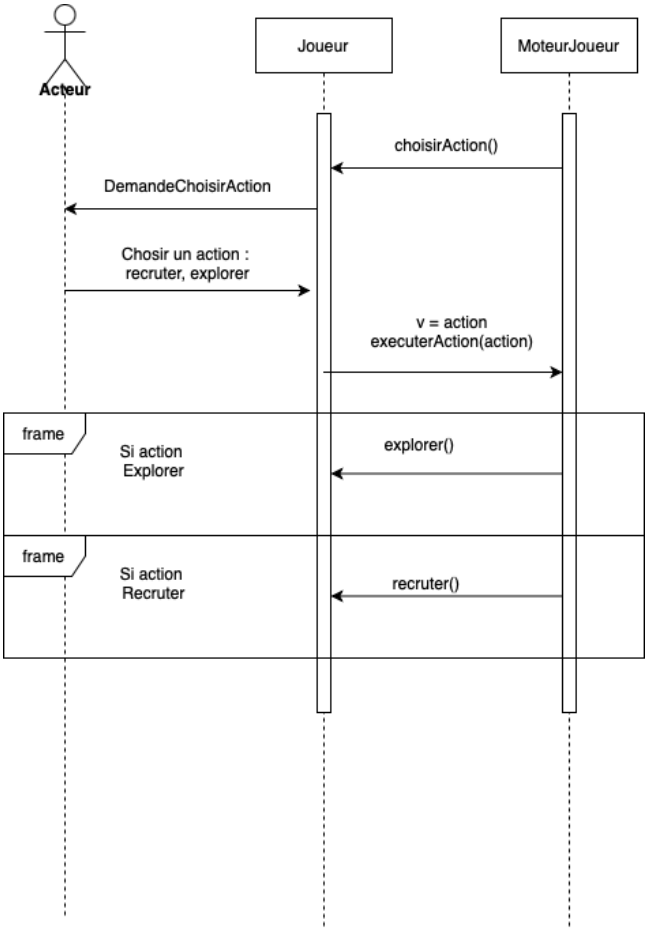
Point de vue dynamique – Diagramme de séquence

- User Story 1 :



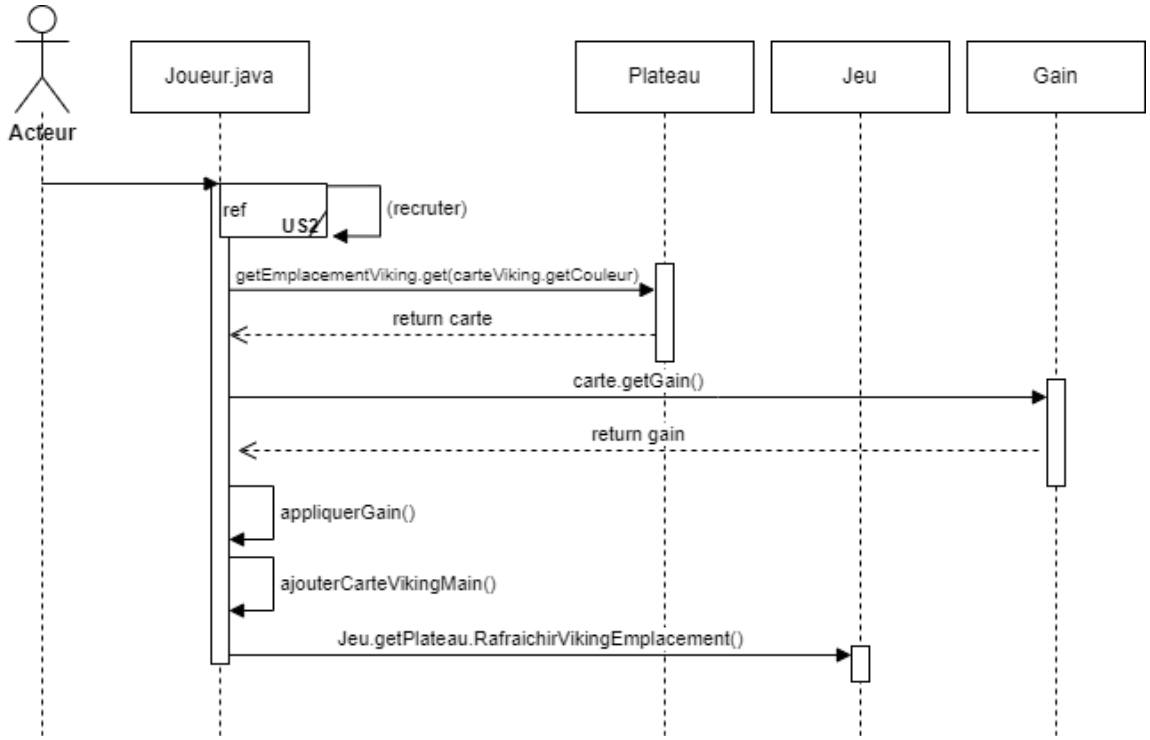
Le diagramme de séquence montre comment un joueur demande à voir l'état de la partie. Cette requête est envoyée à *Jeu Centrale*, qui commence par afficher les cartes du joueur et les cartes des Vikings. Pour obtenir les données du plateau, *Jeu Centrale* appelle la méthode de *Jeu*, qui récupère les 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 :



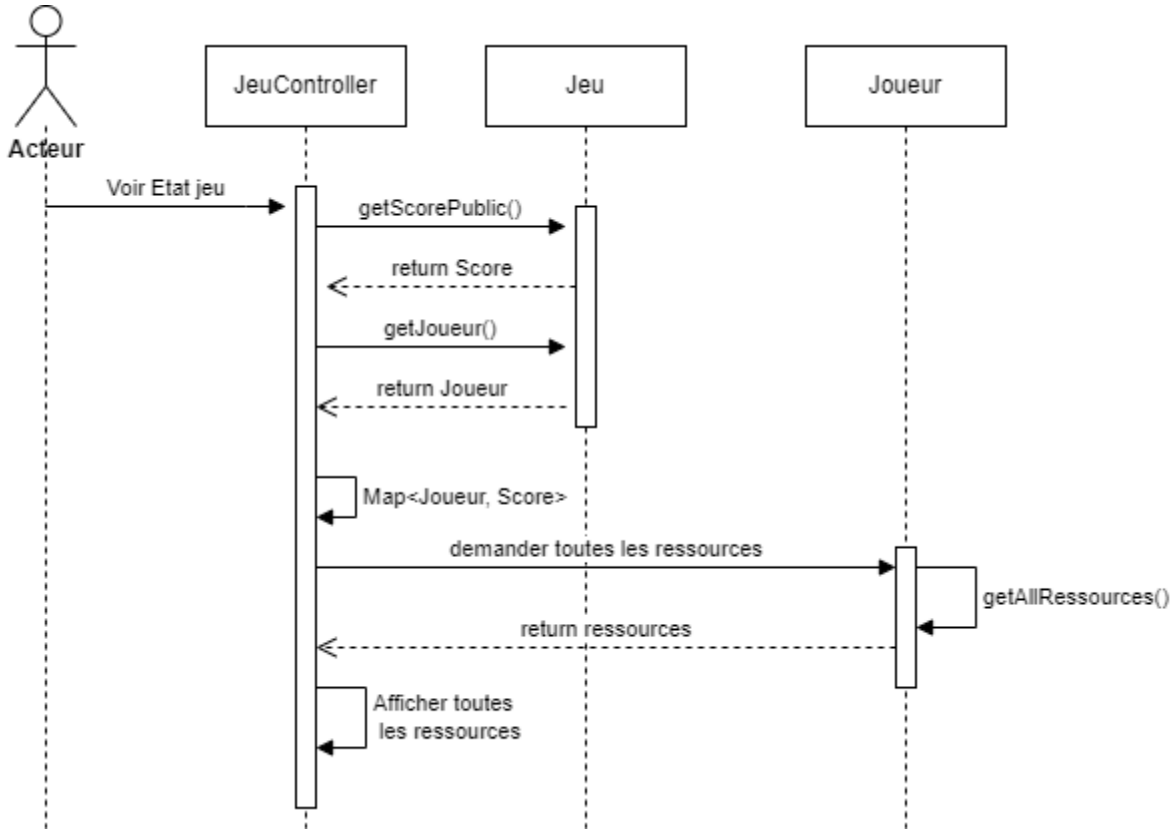
Ce diagramme de séquence montre comment l'acteur demande à un joueur de choisir une action dans le jeu, soit *recruter* soit *explorer*. Le joueur utilise la méthode *choisirAction()* pour sélectionner une action, puis envoie cette action au *MoteurJoueur* avec la méthode *executerAction(action)*. Le 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 :



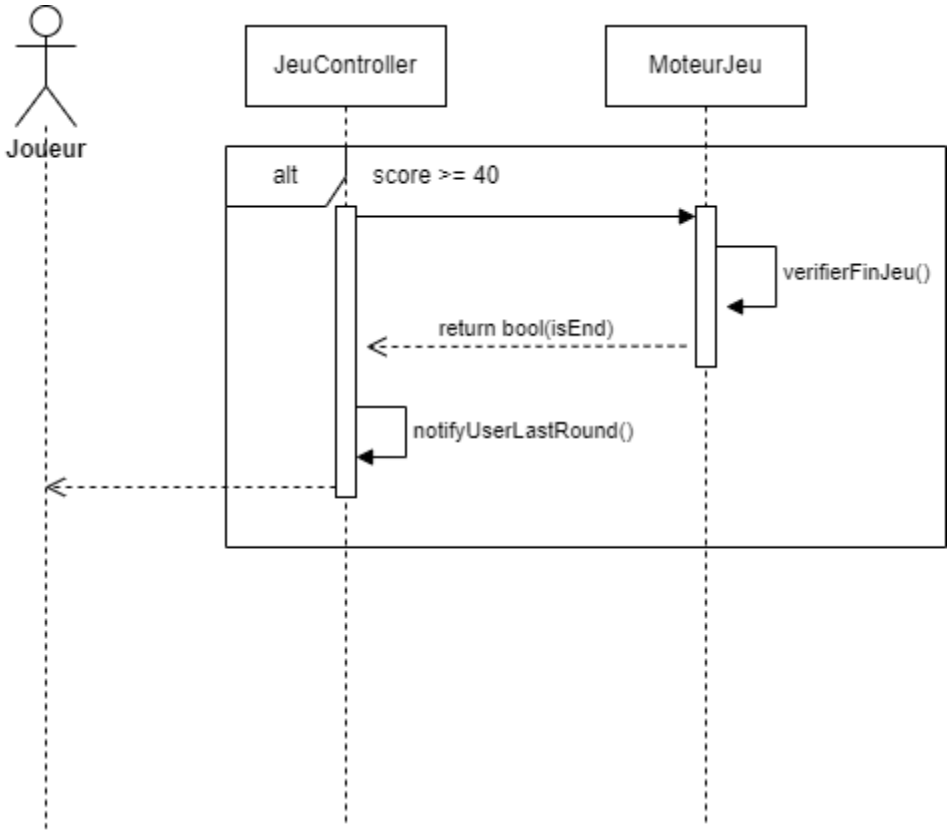
Le diagramme montre l'action *recruter* : l'acteur initie la demande via *Joueur.java*, qui obtient une carte Viking du Plateau selon sa couleur. Ensuite, *Joueur.java* récupère le gain de la carte, applique ce gain, ajoute la carte à la main du joueur, et appelle Jeu pour rafraîchir l'emplacement sur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

- User Story 4 :



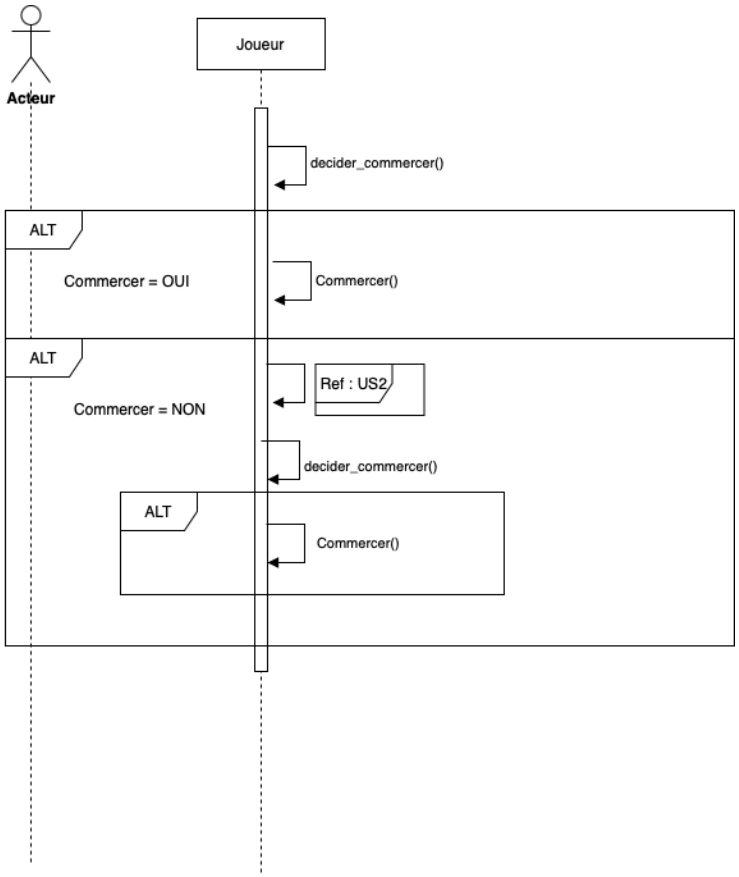
Ce diagramme de séquence illustre la demande de l'acteur pour voir l'état du jeu. L'acteur interroge *JeuController*, qui appelle *Jeu* pour obtenir le score public (*getScorePublic())* et le joueur courant (*getJoueur()*). Ensuite, *JeuController* demande toutes les ressources au joueur via la méthode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

- User Story 5 :



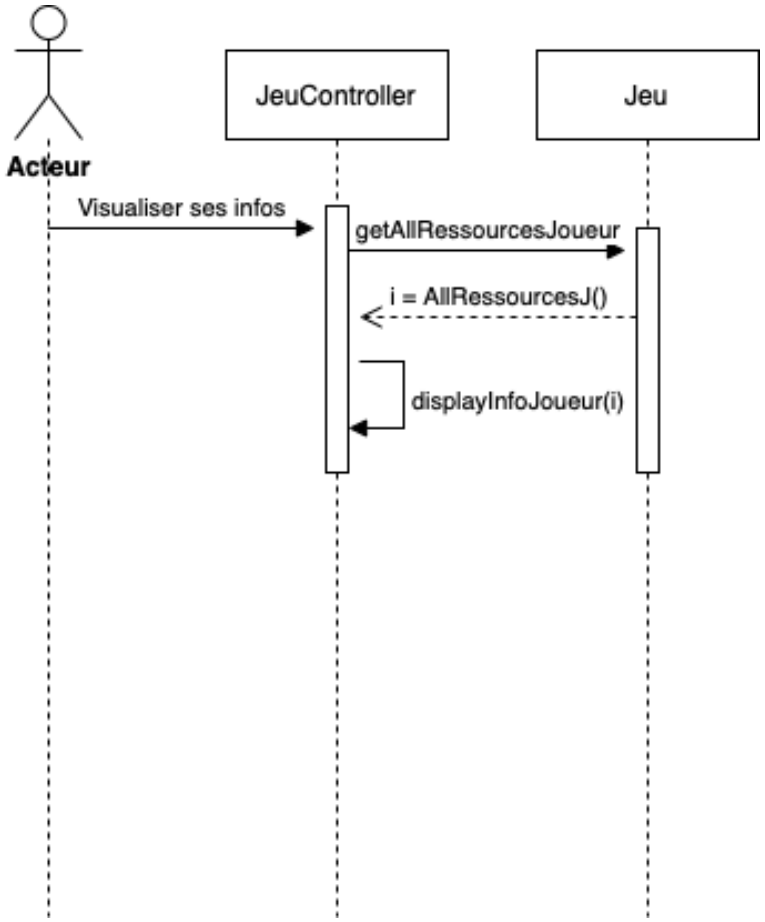
Ce diagramme de séquence montre la vérification de la fin du jeu. Si le score atteint ou dépasse 40, *JeuController* appelle la méthode *verifierFinJeu*() du *MoteurJeu*. Le résultat (*isEnd*) est renvoyé sous forme de booléen. Si la fin est confirmée, *JeuController* notifie le joueur du dernier tour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

- User Story 6 :



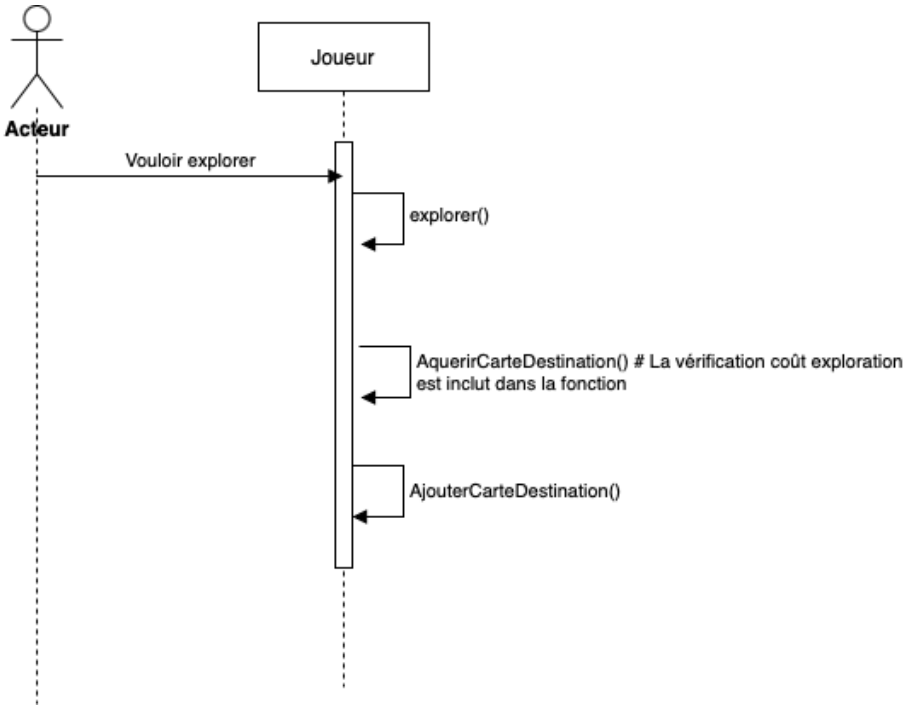
Le diagramme de séquence montre la décision du joueur de commercer. L'acteur appelle *decider_commercer*(). Si le joueur choisit de commercer, la méthode *Commercer*() est exécutée. Sinon, le flux passe à une action référencée par l'US2 (exploration ou recrutement), avant de vérifier de nouveau la possibilité 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 suit une architecture MVC. Bien que le projet n'implique pas d'interface graphique, le contrôleur inclut les éléments normalement attribués à la vue. Par exemple, au lieu de créer un package distinct pour le logging, nous avons intégré cette classe directement dans le contrôleur, car il n'était pas pertinent de créer un package entier pour une seule classe. Cette approche permet une séparation nette des responsabilités, facilitant la maintenance et l'extension du code pour les futures itérations du projet.

- **Conception avec UML** : L'utilisation de diagrammes UML, notamment des diagrammes de classes et d'activité lors du Sprint 1, a offert une vision claire du système à développer. Ces diagrammes ont servi de guide pour l'implémentation des différentes classes et interactions, ce qui a permis une gestion plus fluide du développement. Cela nous a aidés à identifier les éléments clés à implémenter en priorité, optimisant ainsi le déroulement du projet.

- **Utilisation des énumérations** : L'intégration des énumérations comme CouleurBateau, Difficulte, et TypeAction a amélioré l'organisation du code en évitant les chaînes de caractères hardcodées. Cette pratique a rendu le code plus lisible et moins sujet aux erreurs, facilitant également l'ajout de nouvelles fonctionnalités sans introduire de bugs. Les énumérations ont contribué à une évolution progressive et contrôlée du projet.

Points faibles

- **Gestion des données JSON** : Durant le Sprint 3, la gestion des fichiers JSON pour initialiser les cartes du jeu a rencontré des difficultés. Des bugs sont apparus lors du chargement des fichiers, affectant l'intégrité des données. Ces problèmes ont engendré des retards, nécessitant plusieurs ajustements pour assurer une gestion correcte des cartes et une bonne interopérabilité avec le moteur de jeu.

- **Gestion des dépendances** : Des problèmes liés aux versions de Java et aux dépendances des plugins sont apparues dans le pom.xml, rendant la gestion des dépendances complexe et instable. Ces problèmes ont été corrigés, mais cela a mis en évidence le besoin d'une meilleure gestion des versions et d'une attention accrue lors de l'ajout de nouvelles dépendances.

- **Problèmes de conception** : Lors du Sprint 3, un manque d'anticipation a été constaté concernant l'action "Commercer". Initialement définie comme une énumération (TypeAction), cette approche était inadaptée, car "Commercer" n'est pas une action comme "Recruter" ou "Explorer". Elle nécessite une gestion spécifique des échanges entre joueurs, et ne devait pas être choisie par le bot. Cette simplification a entraîné des erreurs dans la logique du jeu et dans les choix des actions par les IA.
  De plus, ce problème a mis en évidence des lacunes dans la conception UML, qui manquait de précision sur certains aspects. Par exemple, l'absence d'une classe dédiée "MoteurDeJeu" pour gérer le déroulement de la partie (lancement, tour, fin) a nécessité des ajustements imprévus pour corriger ces limitations.

Bilan de l'organisation

La première partie du projet a été découpé en trois sprints clairs, chacun focalisé sur des aspects spécifiques :

- **Sprint 1 : Conception du jeu** Nous avons commencé par la réalisation des diagrammes UML (classes et activités). Cette phase de conception a permis d'obtenir une vision globale et structurée du projet. Les diagrammes UML ont aidé à poser les bases du développement et à mieux anticiper les fonctionnalités à implémenter. Cependant, certaines lacunes sont apparues plus tard, notamment sur des interactions complexes comme l'action "Commercer", qui auraient pu être mieux modélisées dès cette étape.
- **Sprint 2 : Modélisation du jeu** Durant cette itération, les classes principales ont été programmées pour représenter les entités du jeu. Le découpage des classes a été efficace et a permis de poser une base solide pour le moteur de jeu. La modularité a facilité l'implémentation, bien que la gestion des interactions complexes ait nécessité des ajustements mineurs. Le travail réalisé a bien préparé le terrain pour les phases suivantes.
- **Sprint 3 : Implémentation du moteur de jeu** Ce sprint a marqué l'implémentation du moteur de jeu, incluant le déroulement complet d'une partie. Nous avons rencontré des difficultés inattendues avec la gestion des fichiers JSON, utilisés pour initialiser les cartes du jeu. Des erreurs de parsing et des problèmes d'intégrité des données ont causé des retards, révélant un manque d'anticipation sur cet aspect technique. De plus, l'action "Commercer", initialement représentée par une énumération, a nécessité une refactorisation en une classe dédiée pour mieux gérer les échanges, car elle n'était pas une action standard comme "Recruter" ou "Explorer".

Bilan sur les User Stories

Le découpage en User Stories a permis de structurer efficacement le développement autour des fonctionnalités essentielles :

Le découpage en User Stories a permis de structurer le développement autour des fonctionnalités principales. Voici l'état d'avancement pour chaque User Story :

1. **US1 : Affichage du plateau et des positions des cartes**
    - Statut : Implémentée
    - L'affichage du plateau via la console a été correctement réalisé, permettant de visualiser les positions des cartes et des Vikings ainsi que les cartes destination. Cette fonctionnalité a été validée et fonctionne comme prévu.
2. **US2 : Réalisation des actions (exploration, recrutement)**
    - Statut : Implémentée
    - Les actions "Exploration" et "Recrutement" ont été intégrées avec succès, offrant aux joueurs la possibilité de choisir et d'exécuter des actions en fonction de leur stratégie. Cette fonctionnalité est stable et opérationnelle.
3. **US3 : Pioche d'une carte lors du recrutement**
    - Statut : Implémentée
    - La pioche d'une carte après une action de recrutement a été bien intégrée. Les joueurs reçoivent automatiquement une carte lorsqu'ils recrutent, conforme aux règles définies.
4. **US4 : Affichage de l'état de la partie (score et ressources)**
    - Statut : Non implémentée
    - Cette fonctionnalité n'a pas encore été développée. L'affichage des scores des joueurs et des ressources disponibles n'est pas présent, ce qui limite le suivi de l'état de la partie.
5. **US5 : Notification de fin de partie**
    - Statut : Non implémentée
    - La fonctionnalité d'informer les joueurs de la fin de la partie n'a pas été mise en place. Le jeu ne dispose pas encore d'un mécanisme pour signaler la conclusion de la partie aux joueurs.
6. **US6 : Action "Commercer" avant ou après une action**
    - Statut : Partiellement Implémentée
    - L'action "Commercer" a été implémentée, mais de manière incomplète. Le mécanisme de commerce est appelé mais n'est toujours pas fonctionnelle et donc pas implémenté.

7. **US7 : Accès aux informations de la zone d'équipage, main et bateau**
    - Statut : Non implémentée
    - Cette fonctionnalité n'a pas été développée. Les joueurs ne peuvent pas consulter les détails de leur zone d'équipage, de leur main ou de leur bateau via la console, ce qui limite l'information disponible pendant le jeu.
8. **US8 : Achat de cartes destination**
    - Statut : Implémentée
    - L'achat de cartes destination a été implémenté avec succès. Les joueurs peuvent acquérir des cartes s'ils disposent des ressources nécessaires, et cette fonctionnalité fonctionne de manière stable.

En résumé, l'organisation du projet a été globalement satisfaisante. Le découpage en itérations et en User Stories a permis de livrer des fonctionnalités concrètes à chaque sprint. Les bases solides posées ont facilité le développement, malgré quelques défis techniques. Les prochaines phases viseront à renforcer la stabilité et à affiner les fonctionnalités pour offrir une expérience de jeu plus fluide.