Spécification fonctionnelle Algorithmique

Thayssa Oulhaci, Hadi Nour El Dine

20 mai 2024

Pourquoi rédigé une spécification ?

Cette spécification a pour but de détailler le paradigme de programmation que nous allons employer, ainsi que le structures de donnés que nous allons implémenter pour nous rapprocher au mieux de la réalité, afin d'offrir une expérience utilisateur transparente et fluide. Nous porterons une attention particulière à la compléxité spacial et temporelle. De plus, nous suivrons les conventions de nommages telles que le ***camelCase*** *et l'utilisation de* ***l'anglais*** *dans le code.*

Table des matières

1 Choix de la conception

1.1 La POO comme paradigme

Dans le contexte d'un jeu tel que le Spin'UN, le paradigme de programmation le plus adapté est celui de la programmation orientée objet. Ce paradigme offre des fonctionnalités qui seront très utiles, telles que l'abstraction, l'héritage, le polymorphisme et l'encapsulation des données. De plus, il permet une grande modularité au sein de l'implémentation.

1.2 UML comme langage de modélisation

Dans le cadre du jeu Spin'UN, l'utilisation du diagramme de classe UML s'aligne parfaitement avec le paradigme de programmation orientée objet. UML offre une représentation visuelle claire des entités du jeu, comme les classes "Joueur" et "Carte", facilitant l'abstraction des données et des comportements associés ainsi les relations entre les entités, telles que celles entre les joueurs et les cartes, peuvent être modélisées efficacement. L'héritage et le polymorphisme d'UML correspondent bien à la complexité du jeu, permettant de représenter la hiérarchie des classes et les comportements spécifiques.
L'encapsulation des données est assurée par les attributs et méthodes des classes.

1.3 Compléxité spaciale et temporelle

2. Structures de Données

*Nous privilégions l'utilisation des structures de données de type **FIFO** (First-In-First-Out) et **FILO** (First-In-Last-Out) dans notre implémentation, assurant ainsi une gestion efficace des éléments selon les besoins du jeu. Compte tenu d'améliorer davantage l'intelligence artificielle, nous envisageons d'explorer la possibilité d'implémenter une structure en **arbre**, notamment pour appliquer l'algorithme du **Depth First Search** (DFS), si le temps le permet. Cela ouvrirait des perspectives pour une IA plus sophistiquée et réactive.*

Dans ce projet, l'efficacité des algorithmes est cruciale, et nous prenons des décisions de conception pour optimiser la complexité spatiale et temporelle. La recherche dans nos structures de données utilise une approche de **dichotomie** plutôt que séquentielle, améliorant considérablement la vitesse d'accés, surtout pour des ensembles de données importants. Pour la gestion des listes de cartes, nous privilégions **l'ajout en tête** plutôt qu'à la fin, minimisant ainsi le coût des opérations d'empilement et de dépilement. Ces choix de conception garantissent une exécution fluide et efficiente du jeu, même avec des volumes de données significatifs. sectionTests

1.4 Tests unitaires

Pour assurer la fiabilité et la robustesse de notre implémentation, nous avons mis en place des tests unitaires. Chaque méthode et chaque classe ont été minutieusement testées pour v'erifier qu'elles se comportent correctement dans différentes situations. Les tests unitaires nous ont permis de détecter et de corriger des erreurs précoces, garantissant ainsi une base solide pour notre code.

1.5 Tests pratiques

En plus des tests unitaires, nous avons passé de nombreuses heures à jouer au jeu Spin'UN pour identifier d'éventuelles erreurs qui pourraient ne pas avoir été capturées par les tests automatisés. Ces sessions de jeu nous ont permis de tester les interactions entre les différentes composantes du jeu dans des scénarios réalistes et variés, d'évaluer l'équilibrage du jeu, et de nous assurer que l'expérience utilisateur est fluide et agréable. Cette approche pratique a été cruciale pour détecter des erreurs subtils et optimiser les mécanismes du jeu.

2 Description

2.1 Contenu du Spin'UN

Le Spin'UN est identique sur tous les points au UNO Spin. Il est nécessaire d'en rappeler les règles du jeu ainsi que le déroulement de la partie.

Le Spin'UN est un jeu de cartes, dont ces dernières sont réparties de la manière suivante :

- **19 cartes bleues** – 0 à 9 (1 carte 0, 2 cartes allant de 1 à 9 dont 5 cartes Spin allant de 1 à 5).
- **19 cartes jaunes** – 0 à 9 (1 carte 0, 2 cartes allant de 1 à 9 dont 5 cartes Spin allant de 1 à 5).
- **19 cartes vertes** – 0 à 9 (1 carte 0, 2 cartes allant de 1 à 9 dont 5 cartes Spin allant de 1 à 5).
- **19 cartes rouges** – 0 à 9 (1 carte 0, 2 cartes allant de 1 à 9 dont 5 cartes Spin allant de 1 à 5).
- **8 cartes +2** - 2 pour chaque couleur
- **8 cartes Inversion** – 2 pour chaque couleur
- **9 cartes Skip** - 2 pour chaque couleur
- **4 cartes Joker**
- **4 cartes Super Joker**

Ce qui fait au total 108 cartes.

Une roue de défis est également présente, et ci-dessous, vous trouverez la liste de ces défis :

- **Presque UN**
- **1 chiffres en moins**
- **1 couleur en moins**
- **Carte rouge**
- **Carte Bleue**
- **Échange de mains**
- **Main visible**
- **La Guerre**

La description de chaque défi n'est pas précisée ici. Je vous invite à vous référer aux règles du jeu de base pour obtenir ces détails.

2.2 Liste des objets à implémenter

Cette section offre une vue d'ensemble des principaux objets qui seront intégrés dans l'implémentation du jeu Spin'UN. Les détails spécifiques, y compris les défis associés à chaque objet, seront présentés dans une section ultérieure du document. Cette liste sert de point de départ, fournissant une perspective générale sur les composants clés du jeu.

- ***Player*** : Représente un joueur participant au jeu.
  - **IA** : Représente un joueur contrôlé par l'intelligence artificielle.
  - **Real** : Représente un joueur humain réel.
  - **War** : Classe permettant de jouer le défi **Guerre**.
- ***Card*** : Classe de base pour toutes les cartes.
  - **NumerotedCard** : Sous-classe de Card, représentant une carte numérotée.
  - **AddTwo** : Sous-classe de Card, représentant une carte +2.
  - **Skip** : Sous-classe de Card, représentant une carte passer son tour.
  - **TurnDirection** : Sous-classe de Card, représentant une carte inversion.
  - **Joker** : Sous-classe de Card, représentant une carte changement de couleur.
    - **SpinCard** : Extension de la classe NumerotedCard, intégrant les éléments spécifiques au Spin'UN.
    - **SuperJoker** : Extension de la classe Joker, représentant une carte +4 avec changement de couleur.
- ***Game*** : Représentation abstraite du jeu.
  - **Un** : Classe pour le jeu Un de base.
  - **SpinUn** : Extension de la classe Un, intégrant les éléments spécifiques au Spin'UN.

- **Room** : Représente une salle de jeu virtuelle où les joueurs se réunissent.
- ***GameReseau*** :Une classe permettant de faciliter l'implémentation de l'algorithme avec le réseau sera appelée dans le code qui lie l'algorithme avec les autres modules.

3 Analyse approfondie des objets et de leurs interactions

Dans cette section, nous approfondirons notre compréhension des objets que nous avons décidé d'implémenter pour le jeu Spin'UN. Chacun de ces objets joue un rôle crucial dans le fonctionnement du jeu, et nous examinerons en détail leurs caractéristiques, leurs interactions et comment ils contribuent à l'expérience globale du joueur.

3.1 Implémentation du module Carte

Le module "Carte" constitue un élément fondamental pour la suite du jeu. Tout d'abord, la classe de base ***Card*** établit les fondations communes pour les autres cartes du jeu Spin'UN. Ensuite, les sous-classes **NumerotedCard** et celles des cartes spéciales apportent des spécifité à ces cartes en représentant respectivement les cartes numérotées et les cartes spéciales.

La classe **SpinCard** étend la logique de la sous-classe **NumerotedCard** pour intégrer des éléments spécifiques au Spin'UN.
En résumé, le module "Carte" offre une structure hiérarchique permettant de représenter de manière modulaire les différentes catégories de carte, contribuant ainsi à la fléxibilité et à la clarté de notre implémentation du jeu.

Card :

- **Attributs** :
  - **color : protected string** - attribut représentant la couleur du carte.
  - **loginPlayer : protected string or null** - attribut indiquant à qui cette carte appartient. Cet attribut sera mis à jour si la carte passe dans la main d'autre joueur.
  - **points : protected number** - attribut qui donnera le coût d'une carte lors de la fin du manche.
  - **isVisible : protected boolean** - attribut permettant de savoir si la carte est visible, ou pas, par les autres joueurs. Cet attribut sera mis à jour à chaque appel de **showHand()**.
  - **effectToApply : protected boolean** - attribut permettant de savoir si l'effet de la carte a été appliqué ou pas.
  - **number : protected number** - attribut permettant de repesenter la classe auquelle les cartes appartiennent.(ex : de 0 à 9 pour les cartes numérotées et les cartes spin, 10 pour les cartes spin, 14 pour les cartes joker, etc...)
- **Méthodes** :
  - → **getters** - les getters seront définis pour chacun des attributs.
  - → **setters** - Les attributs **color** et **points** sont immutables. les setters des autres attributs seront définis.

NumerotedCard : Cette classe héritera de Card. Elle permettra de différencier les cartes numérotées et simples des autres cartes. Elle ne contiendra les attributs et méthodes de la classe mère ainsi que :

- **Méthodes** :
  - → **public displayInfo() :String** : une méthode abstraite pour afficher les information de la carte.

Joker : Cette classe héritera de Card, avec :

- **Attributs** :

- **chosenColor : protected string** - attribut indiquant la couleur choisie par le joueur qui a jouer cette carte.

- **Méthodes** :
  → **getters et setter** - les getter et setter de attribut seront définis.
  → **public displayInfo() :String** : une méthode abstraite pour afficher les information de la carte.

Skip : Cette classe héritera de **Card**. Elle ne contiendra rien de plus que les attributs et méthodes de la classe mère. Elle permettra à un joueur d'empêcher le joueur suivant de jouer son tour.

AddTwo : Cette classe héritera de **Card**. Elle ne contiendra rien de plus que les attributs et méthodes de la classe mère. Elle permettra de faire piocher 2 cartes au joueur qui suit le joueur qui l'a joué.

TurnDirection : Cette classe héritera de **Card**. Elle ne contiendra rien de plus que les attributs et méthodes de la classe mère. Elle permettra de changer le sens de jeu.

SpinCard : Cette classe héritera de **NumerotedCard**. Elle ne contiendra rien de plus que les attributs et méthodes de la classe mère. Elle permettra à un joueur de fair tourner la roue si le joueur précédent a posé une carte de cette classe. Tout sera possible en vérifiant **l'instance** de la carte sur le haut du dépot.

SuperJoker : Cette classe héritera de **Joker**. Elle ne contiendra rien de plus que les attributs et méthodes de la classe mère. Ellle permettera à un joueur de choisir la couleur avec laquelle le jeu continu et fera piocher 4 cartes de la pile au joueur suivant.

3.2 Implémentation du module Joeur

Le module "Joueur" constitue un élément crucial pour le jeu. La classe de base ***Player*** établit les caractéristiques communes à tous les participants, tandis que les sous-classes **AI** et **Real** définissent respectivement les joueurs contrôlés par l'intelligence artificielle et les joueurs réels. La classe **AI** intègre une logique de prise de décision automatisée, pouvant impliquer des algorithmes sophistiqués, ou non, tandis que la classe **Real** représente les joueurs interactifs prenant des décisions réelles pendant le jeu.

Player :

- **Attributs** :
  - **hand : protected Card[]** - attribut correspondant à la main du joueur.
  - **points : protected number** - attribut correspondant aux points gagnés par le joueur en fin de partie. Il sera **mis à jour à la fin de chaque manche si le joueur la remporte**.
  - **login : protected string** - attribut correspondant à l'identifiant d'un joueur. Il est **unique** à chacun ;
  - **isWinner : protected boolean** - attribut booléen qui sera mis à **true** si le joueur ressort gagnant de la partie. Par défaut il est mis à **false**.
  - **isHandVisile : protected boolean** - attribut permettant d'appliquer un des défis, en rendant toutes les cartes du joueur visibles par tous les joueur de la partie.
  - **mustSayUno : protected boolean** - attribut permettant de detecter le moment où le joueur n'a plus qu'une carte.

- **Méthodes** :
  → **public drawCard(drawStack : Card[]) : Card** - permet à un joueur de piocher une carte du sommet de la pioche. Cette méthode renvoie la carte piochée.
  → **public playCard(c : Card, cardDeposit : Card[]) :** : Void - Cette méthode permet au joueur de jouer une carte précise de sa main dans la pile du jeu.
  → ***public treatementChallengeWheeltchallengWheel(playersList :Player[], drawStack : Card[], cardDeposit : Card[]) : Void :*** - Cette mèthode dera implémenter differament dans IA et Real, elle permet à un joueur de tourner la roue et applique un des défis (sur tous les joueurs de la partie si nécessaire).

→ **public challenge(p : Player) : void** - permet à un joueur de **défier** un joueur adverse s'il a posé un **joker ou un superkojer**.

→ **public spinWheel(wheel : number[]) : Number** - permet à un joueur de faire **tourner la roue du Spin'UN**, si au tour précédent,

→ **public sayUno() : void** - permet à un joueur prévenir les autres qu'il a une seule carte dans sa main.

→ Des mèthodes qui permettent au joueur de tenter de jouer une carte sachant que la carte du sommet du dépôt est une carte spéciale (d'une instance précise) seront aussi implémentées.

→ **getters et setter** - les getters et setters pour chacun des attributs seront définis.

AI et Real : Ces deux classes hériteront de la classe abstraire ***Player***. Elles implémenteront chacune le corps de la méthode abstraite ***treatementChallengeWheeltchallengWheel*** différement. Real ne contiendra rien de plus que les attributs et méthodes de la classe mère. IA contiendra la méthode public playableRandomCard(cardDeposit : Card[]) : number - qui sera appelée lorsqu'un joueur tente de jouer une carte. Si la carte passée en paramètre est valide alors **elle est retirée de sa main** et placée dans le **dépot**. La méthode renvoie **true si succès sinon false**.

War : Une classe qui permet d'implémenter le défi WAR auquel tous les joueurs participent.

3.3 Implémentation du module Jeu

Le module "Jeu" occupe une position primordiale dans notre implémentation, il correspond tout bonnement au Spin'UN. La classe de base, ***Game***, définit les caractéristiques communes aux deux modes de jeux, tandis que les sous-classes **Un** et **SpinUn** définissent respectivement les spécificités des jeux Uno classique ainsi que sa variante appelée Uno Spin.

Game :

- **Attributs** :
  - **drawStack : protected Card[]** - attribut correspondant à la pioche. Définit à **108 cartes** au départ du jeu.
  - **cardDeposit : protected Card[]** - attribut correspond au dépot de carte. C'est ici que les joueurs joueront leurs cartes.
  - **players : protected Player[]** - attribut correspondant aux **joueurs** présents dans la partie.
  - **direction : protected static number** - attribut correspond au sens du jeu. Si le jeu est dans le sens des aiguilles d'une montre alors **direction sera égale 1, sinon -1**.
- **Méthodes** :
  - → **public static cardDistribution(playersList : Players[]) : void** - permet la distribution des cartes de **drawStack** aux différents joueurs présents dans **playersList** et de mettre une carte dans la pile **cardDeposit** .
  - → **public static checkCardDeposit() : Card** - renvoie la carte qui est sur le haut de pile **cardDeposit**.
  - → **public deleteCardTopOfDrawStack() : Card** - Cette méthode supprimant la carte sur le haut de **drawStack**.
  - → **public remakeDrawStackIfNoCardIn() : Void** - Cette méthode permet de remplacer la pioche par les cartes du depôt mélangées une fois la piche est vide.
  - → **public abstract createDrawStack() : Card[]** - Cette méthode abstraite permettant d'initiliaser **drawStack** selon l'instance du jeu. Son corps sera implémenter dans les classes filles **ClassicUn** et **SpinUn**
  - → **public cardDistribution(players : Player[]) : void** - Cette méthode permettant la distribution des cartes aux joueurs.
  - → **public isEndRound() : boolean** - Cette méthode permettant de vérifier si un **round** est terminé ou pas.
  - → **public endRound() : void** - Cette méthode permettant de mettre fin au **round** courant et de passer au suivant.

→ **public turnDirection() : void** - Cette méthode permettant de mettre à jour **direction** en le multipliant par **-1**.

→ **getters et setter** - les getters et setters pour chacun des attributs seront définis.

Un : La classe Un représente le jeu de base. Elle ne rajoute rien de plus par rapport à la classe dont elle hérite. Sa seule utilitée est de pouvoir être **instancié** contrairement à sa classe mère ***Game***.

SpinUn :

- **Attributs** :
  - **wheel : private number[]** - attribut représentant la roue de défis du Spin'Un. Chaque défis est traduit par un entier constant qui permettra l'appel à la **méthode de la classe associée**.

3.4 Intégration et relation entre les modules

Dans cette dernière section, nous allons interconnecter tous les modules précédemment décrits dans l'objet **Room** afin d'obtenir un jeu pleinement opérationnel. L'objet **Room** sera ainsi conçu pour **instancier** plusieurs parties. Toutes les parties pourront **s'exécuter en paralléle**.

Room :

- **Attributs** :
  - **idRoom : private number** - attribut indiquant **l'identifiant unique d'une instance de Room**.
  - **playersList : private Player[]** - attribut correspondant aux **participants d'une partie**. Ce sont des **instances des classe AI et Real**.
  - **game : private Game** - attribut correspondant au jeu. Il contiendra l'instance de **Un** dans le cadre d'une partie classique sinon de **SpinUn**.
  - **private : private boolean** - attribut indiquant si la partie a été définie comme **privée** ou non.
  - **keyRoom : private string | undefined** - attribut correspondant à la clef d'entrée si **private** est initialisè à **true**.
  - **modeGame : private number** - attribut indiquant le **mode de jeu instancié**.
  - **pointsNumber : private number** - attribut correspondant au nombre de points à atteindre par un joueur pour **gagner la partie**.
  - **nbMaxPlayers : private number** - attribut indiquant le nombre de participants à attendre pour **waitingPlayers()**. Cet attribut sera initialisé lors de **l'instanciation d'un Room**.
  - **whosTurn : private number** - attribut correspondant à l'indice d'un joueur dans la liste **playersList**.
  - **isFinished : private boolean** - attribut indiquant si la partie est terminée ou non.
  - **isBlocked : private boolean** - attribut indiquant indiquant si la roue doit être tournée ; la partie est bloquée pendant ce temps. .
- **Méthodes** :
  - → **public waitingPlayers() : void** - permet d'attendre **nbMaxPlayers** joueurs qui seront placés dans **playersList**. Le créateur peut lancer la partie même si le nombre de joueurs n'est pas suffisant : des instances d'**IA** seront utilisées pour les remplacer.
  - → **public gameStart() : void** - permettra de commencer la partie en **instanciant Un ou SpinUn dans Game**.
  - → **public isEnd() : Boolean** - méthode permettant de vérifier si une partie est **terminée ou non**. Elle sera appélée continuellement jusqu'à ce qu'elle renvoie **true**.

De plus, une classe **GameReseau** permettant de faciliter l'implémentation de l'algorithme avec le réseau sera intégrée. Cette classe servira de liaison entre le code de l'algorithme et les autres modules du jeu.

GameReseau :

- **Attributs** :
  - **room : public Room** - Attribut permettant d'instancier un objet Room.
  - **roundWinner : public String** - Attribut correspondant au login du joueur gagnant à la fin de la partie.
  - **wheelEffect : protected Number** - Attribut correspondant au défi dans la roue.
  - **isRealToPlay : public boolean** - Attribut permetant de detecter si le joueur est réel ou non(c'est donc une IA).

    .
- **Méthodes** :
  - → **public updateisRealToPlay() : Void** - permet metttre à jour la propriété isRealToPlay pour indiquer si c'est au tour d'un joueur réel de jouer.
  - → **public playableCard(c : Card) : Boolean** - permet de vérifier si une carte peut être jouée sur la pile de dépôt. Retourne true si la carte peut être jouée, sinon false.
  - → **public isItIAToPlay() : Boolean** - permet de vérifier si c'est au tour de l'IA de jouer. Si oui, fait jouer l'IA et retourne true, sinon retourne false.
  - → **public playCardReseau(p : Player, indexCard : Number) : Status** - permet de vérifier tenter de jouer une carte pour un joueur donné à un index spécifié. Retourne un statut indiquant le succès ou l'échec de l'action, ainsi qu'un message d'information.
  - → **public drawCardReseau(p : Player) : Status** - permet de vérifier tenter de faire piocher une carte à un joueur donné. Retourne un statut indiquant le succès ou l'échec de l'action, ainsi qu'un message d'information.
  - → **public changeRound() : Void** - permet de passer à la manche suivante.
  - → **public isRoundFinished() : Status** - permet de vérifier si la **manche** est terminée. Retourne un statut indiquant le succès de l'action et le nom du gagnant de la manche si elle est terminée.
  - → **public isGameFinished() : Status** - permet de vérifier si la **partie** est terminée. Retourne un statut indiquant le succès de l'action et le nom du gagnant de la manche si elle est terminée.
  - → **private async delay(ms : number) : Promise<void>** - permet de retourner une promesse qui se résout après un certain délai en millisecondes.
  - → **public async changeTurnPlayer() : Promise<void>** - permet de faire passer le tour du le joueur actif après un délai et gère les tours des joueurs IA. Si la manche ou la partie est terminée, passe à la manche suivante ou affiche le gagnant.
  - → **public spinWheel() : Number** - permet de faire tourner la roue et retournera le résultat si le jeu est un jeu de type Spin. Sinon, un message d'erreur sera affiché.
  - → **public finishRoundOrGame() : Boolean** - permet de vérifier si la manche ou la partie sont terminées. Si oui, les joueurs passent à la manche suivante sinon le gagnant de la partie sera affiché.
  - → **public replacePlayerByIa(players : (Real | IA)[], login : String) : Void** - permet de remplacer un joueur réel par une IA en transférant les cartes du joueur réel à l'IA.
  - → Des mèthodes qui permetteront de verifier si la carte sur le sommet de la pile de dépôt est d'une instance précise et d'appliquer l'effet de la carte spéciale si ce n'a dérnier n'a pas déjà été appliqué, seront aussi implémentées.