Comment on a développé notre Cascadia

CASCADIA
RANDY FLYNN

Maksat MUNAITPASOV **Mehdi MAOUCHE**

Sommaire

1. Introduction

1.1 Objectif du document

Ce document a pour objectif de fournir une description détaillée de l'architecture et des choix techniques réalisés lors du développement du jeu **Cascadia** en Java. Ce manuel inclut également une section dédiée aux améliorations et corrections apportées après la soutenance β, afin de présenter l'évolution du jeu et les modifications effectuées pour répondre aux retours reçus.

Le document est structuré de manière à :

- Expliquer l'architecture du système, les choix de conception, et la structure du code.
- Détaillez les fonctionnalités implémentées et les technologies utilisées.
- Fournir des éléments sur la qualité du code, les tests effectués, ainsi que les possibilités d'extension du projet.
- Mettre en lumière les principales améliorations et corrections apportées après la soutenance β, et fournir des pistes pour le futur développement du jeu.

1.2 Présentation rapide du jeu

Cascadia est un jeu de société de stratégie où les joueurs doivent créer et gérer leur propre écosystème en plaçant des tuiles représentant différents habitats (forêt, montagne, rivière, etc.) et en y ajoutant des animaux (saumon, ours, etc.). Le but du jeu est de marquer le plus de points en formant des combinaisons d'animaux et en optimisant les placements dans les habitats, tout en répondant à des objectifs spécifiques.

Dans cette version numérique du jeu, l'utilisateur peut choisir entre deux types de configurations de plateau : **carré** ou **hexagonal**, et sélectionner différentes cartes de décompte pour ajuster la difficulté et les objectifs de la partie. Le jeu prend en charge de 2 à 4 joueurs, avec des interfaces disponibles en mode **Terminal** et **Graphique**.

1.3 Technologies utilisées

Langage de programmation : Java

Bibliothèque graphique utilisé : Zen

2. Architecture du Jeu

2.1 Structure du projet

L'architecture du projet **Cascadia** est conçue pour être modulaire et extensible, en permettant une gestion claire et efficace des différentes fonctionnalités du jeu. Le projet est organisé en plusieurs répertoires et classes, afin de séparer les différentes responsabilités du jeu. La structure du projet est la suivante :

```
Cascadia/
├── src/                        # Code source
│   ├── fr.uge/                 # Code principal du jeu
│   │   ├── fr.uge.data/        # Données principales pour jouer au jeu
│   │   │   ├── bag/            # Sac virtuelle d'où le joueur récupère les jetons et les tuiles
│   │   │   ├── core/           # Là ou le tour du joueur est réalisé (gameboard, player …)
│   │   │   ├── environment/    # Création de l'environnement avec les tuiles habitats
│   │   │   ├── util/           # Définition de constantes utilisés dans l'ensemble du projet
│   │   │   ├── scoring/        # Calculs des différentes variantes du projet
│   │   ├── main/               # Où on entre dans le projet
│   │   ├── view/               # Création de l'interface graphique
│   │   ├── controler/          # Gère le bon déroulement de la partie
├── build.xml                   # Fichier de configuration Ant
├── config                      # fichier texte d'où les tuiles sont créés
├── docs                        # Documents en lien avec le jeu
├── img                         # Ensemble des images utilisés dans le projet
└── lib                         # Librairie graphique Zen
```

2.2 Description des modules principaux

fr.uge.data

Ce package contient les modules essentiels liés aux données du jeu. Il est subdivisé en plusieurs sous-modules pour une gestion claire et modulaire des différentes composantes :

1. Module `bag/`

Ce module représente le **sac virtuel** où les jetons (animaux et habitats) sont stockés et d'où le joueur peut les récupérer au cours du jeu. Il s'agit d'un élément clé du gameplay, car il gère la **distribution aléatoire** des tuiles et des jetons nécessaires pour former un écosystème.

2. Module `core/`

Ce module est responsable de la logique **principale du jeu**, c'est-à-dire la gestion du déroulement de la partie, des tours des joueurs, et des règles qui régissent le jeu.

- **GameBoard** : Représente l'ensemble du plateau de jeu, où les tuiles sont placées pour créer l'environnement des joueurs. Elle s'occupe de la **gestion de l'espace de jeu**.
- **Player** : Contient les informations relatives à chaque joueur. Ce module gère également les actions possibles d'un joueur pendant son tour.
- **TurnManager** : Assure la gestion du **tour de chaque joueur**, en alternant entre les joueurs, et en vérifiant si la partie est terminée.

3. Module `environment/`

Ce module est responsable de la **création et de la gestion de l'environnement** dans lequel se déroule le jeu. Il permet de générer les habitats et d'ajouter des tuiles à l'aire de jeu.

- **Habitat** : Gère les types d'habitats disponibles dans le jeu, comme la **forêt**, les **rivières**, etc., et permet de valider si un animal peut être ajouté dans un habitat spécifique.

4. Module `scoring/`

Le module **scoring** est responsable du calcul des scores et des variantes du jeu, telles que les objectifs de décompte.

- **ScoringCard**: Calcule les scores en fonction des placements des animaux.

5. Module `util/`

Ce module contient des **fonctions utilitaires** et des constantes utilisées dans l'ensemble du projet, ainsi que des configurations globales.

- **Constants** : Contient les constantes globales, telles que les types de tuiles, les règles du jeu, et les configurations des objectifs.

fr.uge.main

Ce package est le point d'entrée du projet. Il contient les classes nécessaires pour initialiser le jeu. Selon l'interface choisie, il démarre soit :

- La version terminal.
- La version graphique utilisant la bibliothèque Zen 6.0.

fr.uge.view

Le module **view** gère l'**interface graphique** du jeu. Il permet de rendre le jeu interactif et agréable à jouer, que ce soit en mode **Terminal** ou en mode **graphique** (avec **Zen**).

fr.uge.controler

Le contrôleur agit comme un intermédiaire entre la vue et la logique du jeu. Il est responsable de :

- Gérer les actions des joueurs (placer une tuile, choisir un jeton, etc.).
- Maintenir la synchronisation entre l'état du jeu (modèle) et l'interface (vue).
- Appliquer les règles du jeu à chaque action.

3. Mécaniques de Jeu

3.1 Système de grille hexagonale

Le système de grille hexagonale est au cœur de l'organisation du plateau de jeu dans **Cascadia**. Le plateau est constitué de **tuiles hexagonales**, chaque tuile représentant un habitat (forêt, rivière, montagne, etc.) où les joueurs doivent placer leurs jetons animaux.

Tile : Chaque tuile sur la grille est représentée par un objet **Tile**. Ce record gère les propriétés de la tuile, telles que son type (habitat ou vide), et la possibilité de placer un animal dessus.

HexagonalEnvironment: La classe **HexagonalEnvironment** gère la grille hexagonale en elle-même. Elle permet de créer et d'afficher un plateau constitué de tuiles hexagonales. Le système utilise des coordonnées **axiales** (x, y) pour déterminer la position des tuiles sur la grille.

HexagonalCell: La classe **HexagonalCell** gère la création de la tuile hexagonale. Elle permet de vérifier si un animal occupe cette cell

```java
/**
 * search in the different direction of the hexagonal tile the neighbor of the given tile and return it in a list
 *
 * @param cell the cell that we want the neighbor
 * @return a list of neighbor of the given cell
 */
@Override  9 usages  Maks *
public List<Cell> getNeighbors(Cell cell) {
  Objects.requireNonNull(cell,  message: "Cell can't be null in getNeighbors()");
  return IntStream
      .range(0, HexagonalEnvironment.HEXAGONE_DIRECTION_DIFFERENCES[cell.getCoordinates().y() & 1].length)
      .mapToObj(direction → getOneNeighbor(cell, direction))
      .toList();
}
```

Ici, getNeighbor est utilisé dans le cas spécifique des hexagonal tile pour chercher les voisins d'une tuile hexagonale dans une grille hexagonale en fonction des directions de cette dernière.

3.2 Logique de rotation

La logique de rotation existe mais n'a pas encore été lié au jeu principal

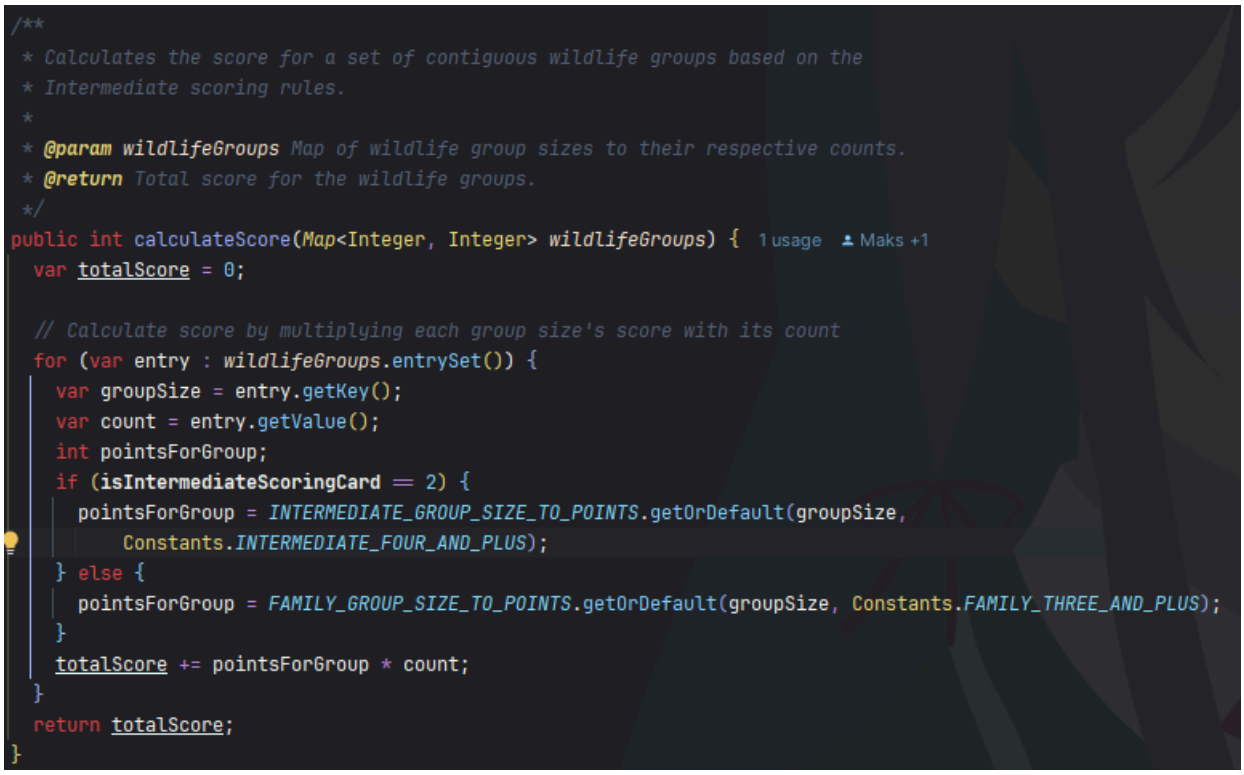
```java
/**
 * Only in Hexagonal version
 */
public final void turnCounterClockwise() {  no usages  Maks
    this.currentRotation = (this.currentRotation + 1) % Constants.MAX_ROTATIONS;
}

/**
 * Only in Hexagonal version
 */
public final void turnClockwise() {  no usages  Maks
    this.currentRotation = (this.currentRotation - 1 + Constants.MAX_ROTATIONS)
        % Constants.MAX_ROTATIONS;
}

/**
 * Only in Hexagonal version
 */
public final int getRotation() { return this.currentRotation; }  no usages  Maks
```

3.3 Système de calcul de score pour le joueur

Le système de calcul de score dans **Cascadia** est basé sur le placement des **animaux** et des **tuiles**, ainsi que sur l'accomplissement des **objectifs** définis par les règles de décompte. Chaque placement correct d'un animal sur une tuile ou chaque combinaison de tuiles réalisée rapporte un certain nombre de points.

FamilyAndIntermediate :

- Ici, chaque groupe de tuiles avec un animal précis rapporte des points.



```java
/**
 * Calculates the score for a set of contiguous wildlife groups based on the
 * Intermediate scoring rules.
 *
 * @param wildlifeGroups Map of wildlife group sizes to their respective counts.
 * @return Total score for the wildlife groups.
 */
public int calculateScore(Map<Integer, Integer> wildlifeGroups) {  1 usage   Maks +1
  var totalScore = 0;

  // Calculate score by multiplying each group size's score with its count
  for (var entry : wildlifeGroups.entrySet()) {
    var groupSize = entry.getKey();
    var count = entry.getValue();
    int pointsForGroup;
    if (isIntermediateScoringCard == 2) {
      pointsForGroup = INTERMEDIATE_GROUP_SIZE_TO_POINTS.getOrDefault(groupSize,
          Constants.INTERMEDIATE_FOUR_AND_PLUS);
    } else {
      pointsForGroup = FAMILY_GROUP_SIZE_TO_POINTS.getOrDefault(groupSize, Constants.FAMILY_THREE_AND_PLUS);
    }
    totalScore += pointsForGroup * count;
  }
  return totalScore;
}
```

Scoring Card :

- Ici, le but est de créer une méthode unique de compter les points pour chaque animal, certains animaux rapportent des points en étant isolés (FOX ou EAGLE) alors que d'autres en rapportent en étant en groupe (BEAR ou SALMON).
- On a pas eu le temps de l'implémenter dans la version finale mais on a réalisé 15 méthodes fonctionnelles

```java
/**
 * This method take all the tiles in the player's environment and stock the number of group of Bear in a List
 *
 * @param player The player whose score is calculated
 * @return list of number of 1, 2, 3 and 4 groups of bear
 */
public ArrayList<Integer> numberBear(Player player){  1 usage  Sekiro *
  List<Cell> cells = player.getEnvironment().getCells();
  ArrayList<Integer> numbers = new ArrayList<>();
  numbers.add(0);
  numbers.add(0);
  numbers.add(0);
  numbers.add(0);
  var onlyBear = cells.stream().filter(cell -> cell.getAnimal() == BEAR)
                .toList();
  List<Integer> res = WildlifeScoringCard.calculateGroupScore(onlyBear, player, BEAR);
  for(var each : res){
    if(each>0 && each < 5) numbers.set(each, numbers.get(each) + 1);
  }
  return numbers;
}
```

```java
/**
 * calculate the size of all the group of connected cells with targetAnimal
 *
 * @param cellsWithAnimal A list of Cell with a specific WildlifeType in the player's environment
 * @param player current player which we want to calculate his score
 * @param targetAnimal the animal WildlifeType that we want to focus for our research
 * @return a list of integer where each integer represents the size of a group
 */
static List<Integer> calculateGroupScore(List<Cell> cellsWithAnimal, Player player, WildlifeType targetAnimal) {
  var visited = new HashSet<Cell>();
  var groupScores = new ArrayList<Integer>();

  for (var cell : cellsWithAnimal) {
    if (!visited.contains(cell)) {
      int groupSize = groupAnimal(cell, visited, player, targetAnimal);
      if (groupSize > 0) {
        groupScores.add(groupSize);
      }
    }
  }
  return groupScores;
}
```

```java
/**
 * Scan all the values in speciesCount and, depends on the parameter, count specifically one or multiple type of animal in speciesCount than return the result
 *
 * @param speciesCount a Map<WildlifeType, Integer> who stock the animal and their respective number in the neighborhood of a cell
 * @param exception a boolean who permits, if false, to skip the count of the parameter animal
 * @param animal the wildlife Type of the animal that we maybe want to skip or not, depends on the choice of the player
 * @param onlyMostCommonAnimal a boolean who permits, if true, to count only the most present Wildlife type in speciesCount
 * @return either the sum of species in speciesCount (with or without a specific animal) or only the most present animal in species count
 */
static int numberNeighborSpecies(Map<WildlifeType, Integer> speciesCount, boolean exception, WildlifeType animal, boolean onlyMostCommonAnimal){  5 usages  Sekiro *
  int res = 0;
  ArrayList<WildlifeType> count = new ArrayList<>();
  for(var each : speciesCount.entrySet()) {
    if(each.getKey() == animal) { // l'animal important
      if(exception && !count.contains(each.getKey())) { // vérifie si tous les animaux sont bien différents
        res += each.getValue();  // on compte l'animal
        count.add(each.getKey());
        continue;
      }
      if(onlyMostCommonAnimal){  // si on compte uniquement l'animal le plus présent
        res += each.getValue();
        continue;
      }
      continue; // on ne compte dans aucun cas l'animal
    }
    res += each.getValue();  // sinon on compte juste le nombre d'espece différente autour
  }
  return res;
}
```

Les deux premières méthodes permettent de calculer le nombre de groupe pour les scoring card des ours, la dernière est utilisée pour fox et hawk dans le cas spécial où on veut compter ou non un voisin spécifique dans les cellules adjacentes.

3.4 Gestion de la bibliothèque graphique Zen

La bibliothèque graphique **Zen** est utilisée pour créer l'interface graphique de **Cascadia**. Elle permet d'afficher le plateau de jeu, les tuiles, les animaux, ainsi que l'interface utilisateur permettant d'interagir avec le jeu.

```java
/**
 * Handles user's action, in this case only PointerEvent.
 * If user wants to leave he can leave by pressing ESCAPE or Q.
 * We ignore any other event.
 * @param context
 * @param width
 * @param height
 * @return
 */
public Coordinates getCoordinatesFromUser(ApplicationContext context, int width, int height) {
  for (;;){
    var event = context.pollOrWaitEvent(10);
    switch (event) {
      case null -> { continue; /* do nothing */ }
      case PointerEvent pe -> {
        var coords = getCoordinatesFromPointerEvent(pe, width, height);
        if (coords != null) {
          return coords;
        }
      }
      case KeyboardEvent ke -> {
        if (userWantsLeave(ke, width, height)) {
          return null;
        }
      }
      default -> { /* do nothing */ }
    }
  }
}
```

Model View Controler: L'implémentation de **Model View Controler** à été effectué et utilisé dans l'affichage graphique.

4. Améliorations apportées

Depuis la phase Soutenance β, des améliorations significatives ont été apportées. Cette section décrit ces améliorations en détail.

4.1 Amélioration de la performance

4.1.1 Utilisation de nouvelles structures de données : HashMap

- Les **HashMap** ont été utilisées pour améliorer l'efficacité des recherches et des mises à jour, en particulier pour :
    - Suivre les connexions entre les habitats.

4.1.2 Utilisation des Sets

- Les **Sets** (tels que HashSet) ont remplacé les listes là où les duplications étaient inutiles, par exemple pour gérer les listes ou la duplication étaient inutiles

4.1.3 Optimisation avec les Streams

- Les **Streams** de Java ont été utilisés pour :
    - Filtrer et traiter les données de manière plus concise et lisible.
    - Réaliser des calculs de score et d'autres opérations répétitives sur les collections

4.2 Ajout de nouvelles fonctionnalités

4.2.1 Écriture de code selon les bonnes pratiques de Java 23

- Le code a été adapté pour tirer parti des nouvelles fonctionnalités introduites avec Java 23, notamment :
    - L'utilisation de records pour des structures de données immuables (comme les tuiles et les jetons).

4.2.2 Fonctionnalités étendues avec Zen 6.0

- Une meilleure maîtrise de Zen a permis l'ajout de nouvelles fonctionnalités graphiques

5. Contribuer au Projet

5.1 Documentation et commentaires dans le code

De la javadoc en anglais est également disponible dans toutes les méthodes du code pour apporter une description plus clair et une compréhension plus rapide de la méthode

```java
/**
 * retrieve the tile at the given index in the GameBoard's tiles list
 *
 * @param index the index which we want the tile in the list tiles
 * @return the tile at the given index
 */
public Tile getTile(int index){  1 usage  Maks
  if (index < 0 || index ⩾ Constants.TILES_ON_BOARD) {
    throw new IllegalArgumentException("Index of tile out of bounds");
  }
  var tile = GameBoard.tiles.get(index);
  GameBoard.tiles.set(index, this.bag.getRandomTile());  /* replace the tile */
  return tile;
}
```

Ici, la javadoc donne une description claire et concise en anglais des paramètres de la méthode, de ce que retourne la méthode et de ce qu'elle fait.

5.2 Utilisation de GitHub

On utilise régulièrement github pour s'échanger du code, partagez des problèmes à régler ou encore suivre l'avancé du programme

Conclusion

Le fichier **dev.pdf** constitue une documentation essentielle pour comprendre l'architecture, le développement et les améliorations du projet **Cascadia**. Il détaille la structure du code, les choix technologiques, les optimisations de performances, et les nouvelles fonctionnalités ajoutées depuis la phase initiale.