CHAMI Raëd & MATHIEU Julien

Institut Gaspard-Monge
L3 Informatique

CASCADIA - Manuel d'architecture

12 janvier 2025

—

Version finale

Table des matières

Introduction

Ce projet a pour objectif de développer une version électronique du jeu de société *Cascadia*. Il s'agit de concevoir une application en **Java** qui soit à la fois fidèle aux règles originales du jeu et agréable à utiliser, tout en mettant en pratique les principes fondamentaux de la **programmation orientée objet**.

L'ensemble du travail a été divisé en plusieurs phases progressives, permettant de construire une application robuste, modulaire et extensible. Chaque étape a été conçue pour s'assurer que le produit final est conforme aux attentes pédagogiques, en mettant un accent particulier sur la conception logicielle, la lisibilité du code et le respect des bonnes pratiques de développement.

Le projet se distingue par les défis techniques suivants :

- La gestion d'une **interface utilisateur mixte** (ligne de commande et interface graphique) ;
- L'implémentation d'une logique complexe respectant les règles officielles du jeu *Cascadia* ;
- L'utilisation de la bibliothèque graphique **Zen** pour développer une interface visuelle simple et fonctionnelle ;
- L'application du modèle de développement **MVC (Modèle-Vue-Contrôleur)** afin de séparer les responsabilités logiques, graphiques et de gestion des événements.

Le développement du projet a également suivi un processus rigoureux qui inclut :

1. **L'analyse des règles du jeu** pour une adaptation fidèle en version numérique.
2. **La conception d'une architecture objet** permettant d'intégrer facilement de nouvelles fonctionnalités.
3. **La séparation en package** des classes servant une même fonctionnalité.
4. **L'écriture de documentation technique et utilisateur** pour accompagner le produit final.
5. **Des étapes de validation intermédiaires**, incluant une soutenance et un rendu partiel, afin de garantir un projet conforme aux attentes.

En outre, ce projet vise à proposer une expérience utilisateur immersive et intuitive. En ce sens, chaque fonctionnalité a été pensée pour répondre à un double public : d'une part, les amateurs de jeux de société souhaitant retrouver l'expérience authentique de *Cascadia*, et d'autre part, les utilisateurs moins familiers avec ce type de jeu, qui bénéficieront de guides simples et d'un design accessible.

Grâce à cette approche, nous espérons offrir à **Bosphore, 11 ans**, une application qui lui permettra de jouer avec plaisir, tout en démontrant notre capacité à relever les défis liés au développement d'une application Java complexe et complète.

Contexte et Exigences du Projet

La réalisation de ce projet, **le jeu Cascadia**, s'inscrit dans un cadre académique avec des exigences précises en termes de conception et de méthodologie. Mon binôme et moi avons eu la chance de travailler dans des conditions favorables, car nous nous voyions tous les jours pendant la période scolaire, ce qui a facilité la communication et la coordination. Cependant, malgré cette proximité, plusieurs défis se sont présentés, principalement liés à l'organisation et à la compréhension des attendus du projet.

L'un des principaux obstacles a été la nécessité de s'adapter à une approche orientée objet (POO), qui, bien que déjà introduite dans notre formation, n'avait pas été explorée en profondeur. Ce projet nous a plongés dans un univers où la maîtrise des concepts de POO — tels que l'abstraction, l'encapsulation, l'héritage et le polymorphisme — était non seulement essentielle, mais aussi centrale à la réussite du projet. Cela nous a demandé un effort important de recherche, d'apprentissage et de mise en pratique, afin d'appliquer efficacement ces principes dans la conception et le développement du jeu.

En parallèle, les exigences strictes en matière de structuration du code, de documentation et de rendu final nous ont poussés à adopter une rigueur méthodologique accrue. Nous avons dû nous assurer que chaque étape du projet était conforme aux consignes, en veillant à maintenir un code propre, bien organisé et évolutif, tout en produisant une documentation claire et accessible.

Enfin, nous avons aussi dû trouver un équilibre entre la gestion du temps, les tâches académiques en parallèle, et la compréhension détaillée des règles et mécaniques du jeu Cascadia, nécessaires pour bien les transcrire dans notre programme. Ces défis ont été autant d'opportunités de nous renforcer dans nos compétences techniques et organisationnelles, en nous préparant à aborder des projets de plus grande envergure à l'avenir.

Compilation et Exécution du Projet

Le projet utilise Ant pour compiler, créer le jar, et générer la Javadoc. Voici les principales commandes :

- **Compiler les sources** : `ant compile`
- **Créer le jar exécutable** : `ant jar`
- **Générer la Javadoc** : `ant javadoc`
- **Nettoyer le projet** : `ant clean`

Pour exécuter ou déboguer depuis Eclipse :

1. Importez le projet en tant que "Projet Java existant".
2. Ajoutez les librairies du répertoire `lib` dans le Build Path d'Eclipse.
3. Lancez la classe principale en exécutant le fichier `Main.java` dans `graphical.main`.

Aperçu de l'architecture

Le projet **Cascadia** est structuré selon une approche modulaire et organisée, facilitant la maintenance, l'évolutivité et la clarté du code. Nous avons adopté une architecture basée sur des **packages**, chaque package regroupant des classes ou interfaces qui remplissent une responsabilité spécifique. Cela permet une séparation claire des préoccupations tout en assurant une organisation logique des fonctionnalités.

Packages et Contenu

1. core

Le package `core` contient le cœur logique et fonctionnel du jeu. Il est subdivisé en plusieurs sous-packages pour regrouper les responsabilités.

- **game** : Ce sous-package encapsule les fonctionnalités générales du jeu, telles que :
  - `Game` : Classe principale qui gère la logique de la partie.
  - `GameVariant` : Permet de définir et gérer les variantes de jeu.

- **game.grid** : Regroupe les classes relatives à la gestion de la grille de jeu et des stratégies de voisinage :
  - `Grid` : Représente la grille de jeu.
  - `Tile` : Modélise les tuiles de jeu.
  - Stratégies de voisinage :
    - `NeighborStrategy` et ses implémentations comme `AllNeighbors` ou `HexNeighborsOdd` gèrent les règles pour identifier les voisins des tuiles.
- **game.mechanics** : Ce package couvre les éléments mécaniques du jeu tels que :
  - `Deck` : Gestion des cartes et pioches.
  - `Habitat` et `Wildlife` : Modélisation des habitats et de la faune.

- **game.player** : Contient tout ce qui est relatif aux joueurs, incluant :
  - `Player` : Classe représentant un joueur.
  - `PlayerScore` : Interface pour le système de calcul des scores.
  - Implémentations des scores en fonction de la variante : `FamilyScoring`, `IntermediateScoring`, `StandardScoring`.

- `ScoringStrategyFactory` : Fabrique pour créer dynamiquement une stratégie de scoring en fonction de la variante de jeu.

2. graphical

Le package `graphical` est dédié à la partie graphique et suit une architecture MVC (**Modèle-Vue-Contrôleur**) pour garantir une gestion claire et extensible des interfaces utilisateur.

- **main** :
  - `Main` : Point d'entrée du jeu avec l'interface graphique. Permet de basculer vers une interface terminal via les options.
- **model** : Regroupe les classes définissant l'état du jeu :
  - `GameMode` : Définit les modes de jeu de choix d'interface.
  - `GameStateManager` : Gère les états et transitions entre scènes de jeu.
- **view** : Contient les classes liées à l'affichage :
  - **components** : Définit les éléments graphiques (tuiles, cartes, grilles).
    - Exemple : `GraphicalTile`, `GraphicalCard`.
  - **renderers** : Gère le rendu graphique des différentes scènes :
    - Exemple : `RenderGame`, `RenderMenu`.
  - **resources** : Gère le chargement des ressources graphiques (images, cartes, etc.).
    - Exemple : `ScoringCardsImageManager`, `TileImageManager`.
- **controller** : Contient la logique de contrôle entre la vue et le modèle :
  - **components** : Gère les interactions utilisateur sur des éléments graphiques spécifiques (grilles, tuiles, etc.).
  - **events** : Gère les événements comme les clics ou les changements d'état.

3. images

Ce package contient toutes les ressources graphiques utilisées dans le projet. Les sous-dossiers organisent les images par type :

- **cards** : Contient les cartes de scoring (faune).
- **habitats** : Représente les tuiles d'habitat.
- **wildlife** : Stocke les images des jetons de faune.
- **Autres** : Éléments décoratifs pour l'interface graphique.

4. terminal

Ce package regroupe les classes pour la version terminal du jeu, permettant une exécution sans interface graphique.

- **main** :
  - `Main` : Point d'entrée pour démarrer le jeu en mode terminal.

- **ui** :
  - Contient les classes d'affichage pour les interactions dans le terminal. Exemple : gestion des grilles ou affichage des scores.

Conclusion

Cette organisation modulaire permet d'isoler les différentes responsabilités dans des packages bien définis, ce qui facilite :

- **La lisibilité** : Chaque package a une responsabilité précise et clairement identifiable.
- **La maintenabilité** : Les modifications dans une partie du projet n'affectent pas les autres parties.

- **L’évolutivité** : La structure peut être étendue facilement pour ajouter de nouvelles fonctionnalités (nouvelles variantes, nouveaux modes, etc.)..

Décisions Architecturales

1. **Utilisation des Packages**

Nous avons organisé le projet en packages logiques pour assurer une séparation claire des responsabilités. Cette structure modulaire garantit :

- Une meilleure lisibilité du code.
- Une collaboration plus fluide entre les membres de l’équipe.
- Une facilité d’évolution et de maintenance pour les futurs développeurs. Les packages tels que `core`, `graphical`, et `terminal` permettent de distinguer le cœur fonctionnel du jeu, l’interface graphique, et l’interaction en mode terminal.

2. **Approche Orientée Objet (POO)**

   L’approche POO a été adoptée pour maximiser la modularité, la réutilisabilité et la maintenabilité. Les classes telles que Player, Tile, et Grid encapsulent des responsabilités spécifiques.

   - L’héritage et les interfaces (par exemple, `PlayerScore`) offrent une flexibilité accrue pour introduire de nouvelles fonctionnalités.
   - Cette approche facilite également l'application des principes SOLID, renforçant la robustesse et la qualité du code.

3. **Interface `PlayerScore` et Usine `ScoringStrategyFactory`**

   L'introduction de l'interface PlayerScore et de la classe ScoringStrategyFactory illustre notre choix de conception orienté vers l'extensibilité et l'adaptabilité :

   - **Interface `PlayerScore`** : Fournit une abstraction pour le calcul des scores. Cela permet de développer de nouvelles stratégies (comme `FamilyScoring` ou `IntermediateScoring`) sans impact sur le reste du code, en accord avec le principe ouvert/fermé.
   - **Usine `ScoringStrategyFactory`** : Centralise la création des instances de stratégies, rendant la logique plus modulable et facile à modifier en cas d'ajout ou de suppression de variantes.

4. **Modularité et Extensibilité**

   La modularité a été un axe central de notre conception.

   - La classe `GameVariant` et son intégration dans des composants comme `Game` facilitent l'ajout de nouvelles variantes de jeu sans nécessiter de modifications profondes des autres composants.
   - La conception modulaire des grilles (`Grid`) et des stratégies de voisinage (`NeighborStrategy`) permet de varier les configurations et les règles selon les besoins.

5. **Points d'Entrée Centralisés**

   - **Interface Graphique** : Le fichier `Main.java` dans le package `graphical.main` centralise le lancement de l'interface graphique.
   - **Mode Terminal** : Un point d'entrée dédié (`Main.java` dans `terminal.main`) offre une alternative simple et testable. Cette séparation garantit que chaque mode fonctionne indépendamment, tout

en partageant une logique commune au sein des packages `core`.

6. **Isolation des Règles de Scoring**

   L'externalisation des règles de scoring dans des classes dédiées (par exemple, FamilyScoring, IntermediateScoring) permet :

   - Une maintenance simplifiée : Les ajustements aux règles ou l'ajout de nouvelles stratégies n'affectent pas le cœur de la logique de jeu.
   - Une flexibilité accrue : Il est possible de tester ou d'intégrer des variantes personnalisées sans réécrire de code existant.

7. **Adaptabilité aux Exigences**

   L'architecture mise en place est pensée pour évoluer avec les besoins. Quelques exemples d'adaptabilité incluent :

   - **Gestion des positions** : La classe `PositionInput` encapsule les coordonnées des grilles et prépare le terrain pour des intégrations avancées (comme les clics en interface graphique ou des interactions multijoueurs en réseau).
   - **Interface graphique et terminal** : La coexistence des interfaces graphiques et terminales témoigne de cette adaptabilité. Cela ouvre la voie à une extension future pour des plateformes variées ou un jeu en ligne.

8. **Gestion des Ressources Graphiques**

   Les ressources graphiques (cartes, tuiles, jetons) sont isolées dans le package images et organisées par catégories (cards, habitats, wildlife, etc.). Cette organisation :

- Facilite la localisation et le remplacement des fichiers.
- Offre une flexibilité pour intégrer de nouveaux éléments visuels sans impact sur le code.

9. **Support Multi-Modes**

   La conception permet un basculement entre différents modes de jeu (graphique et terminal) avec une logique commune. Cela a été possible grâce à l'utilisation de structures comme `GameStateManager` dans le package `graphical.model`, qui gère les transitions d'état de manière générique.

Conclusion

Ces choix architecturaux reflètent un équilibre entre modularité, extensibilité et adaptabilité. L'organisation en packages, l'utilisation d'interfaces et de modèles comme les usines, ainsi que l'accent mis sur l'isolation des responsabilités garantissent un projet robuste, maintenable, et évolutif. Ces décisions rendent également le projet accessible pour les nouveaux développeurs, facilitant leur prise en main.

Améliorations et Corrections depuis le rendu intermédiaire

1. **Renforcement de l'isolation des responsabilités**
    - Les packages et leurs contenus ont été réorganisés pour renforcer la séparation des responsabilités. Par exemple, les règles spécifiques de scoring ont été davantage encapsulées dans des sous-classes dérivées de `VariantScoring`, réduisant ainsi les dépendances entre les packages.
    - Des ajustements ont été apportés aux classes comme `PositionInput` pour minimiser les croisements inutiles avec d'autres modules, facilitant ainsi l'intégration future.

2. **Évolution des variantes du jeu**
    - La classe `GameVariant` a été enrichie pour inclure des fonctionnalités supplémentaires permettant de mieux gérer les paramètres spécifiques de chaque variante. Ces améliorations ont permis une meilleure gestion de la modularité et un ajout plus fluide de nouvelles variantes.

3. **Optimisation de la gestion des scores**
    - L'interface `PlayerScore` a été simplifiée et documentée pour garantir une meilleure lisibilité et compréhension des stratégies de scoring.
    - Des tests supplémentaires ont été développés pour valider la cohérence des calculs dans différentes variantes.

4. **Amélioration de l'interface utilisateur**
    - Bien que l'objectif principal reste de fournir une interface en ligne de commande, les bases ont été posées pour intégrer une interface graphique utilisant la bibliothèque Zen. Ces modifications incluent la mise à jour de `PositionInput` pour permettre une prise en charge flexible des clics utilisateur dans une future version graphique.

5. **Tests approfondis et robustesse**
    - Une couverture de tests a été élargie, notamment pour valider les cas limites (par exemple, des erreurs de positionnement de tuiles ou des règles de scoring complexes). Ces tests ont permis d'identifier et de corriger des erreurs mineures dans la gestion des scores et l'interaction entre les composants `Grid` et `Tile`.

6. **Documentation technique et pédagogie**
    - Une documentation technique détaillée a été ajoutée pour chaque package et classe, y compris des exemples d'utilisation. Cette documentation a été rédigée pour faciliter la reprise du projet par de nouveaux développeurs ou dans un cadre éducatif.
    - Les commentaires dans le code ont été enrichis afin de fournir plus de contexte sur la logique métier et les décisions architecturales.

7. **Amélioration des performances**
    - Plusieurs sections du code ont été retravaillées pour éliminer les redondances et améliorer les performances. Par exemple, la classe `Draw`

a été optimisée pour gérer plus efficacement les tirages aléatoires, réduisant ainsi le temps de calcul dans des scénarios à grande échelle.

8. **Préparation à l'intégration réseau**
    - Bien que l'implémentation complète d'un mode réseau soit hors de portée pour cette version, une architecture de base a été conçue pour prendre en charge des interactions en réseau dans le futur, en anticipant des éléments comme la synchronisation des scores et des tours.

Points Restants ou Futures Améliorations

- Ajout d'habitats idéaux, d'habitats avec deux biomes, de jetons nature.
- Permettre la rotation des tuiles.
- Ajout d'un mode solo et d'une liste de succès.
- Ajout d'une fonctionnalité de sauvegarde et de reprise des parties.
- Exploration de l'intégration multijoueur en ligne pour enrichir l'expérience utilisateur.