Projet Logiciel Transversal: SmallWorld

Julien METZELARD – Tarek TALSI – Maël ARCHENAULT – Victor MOREL

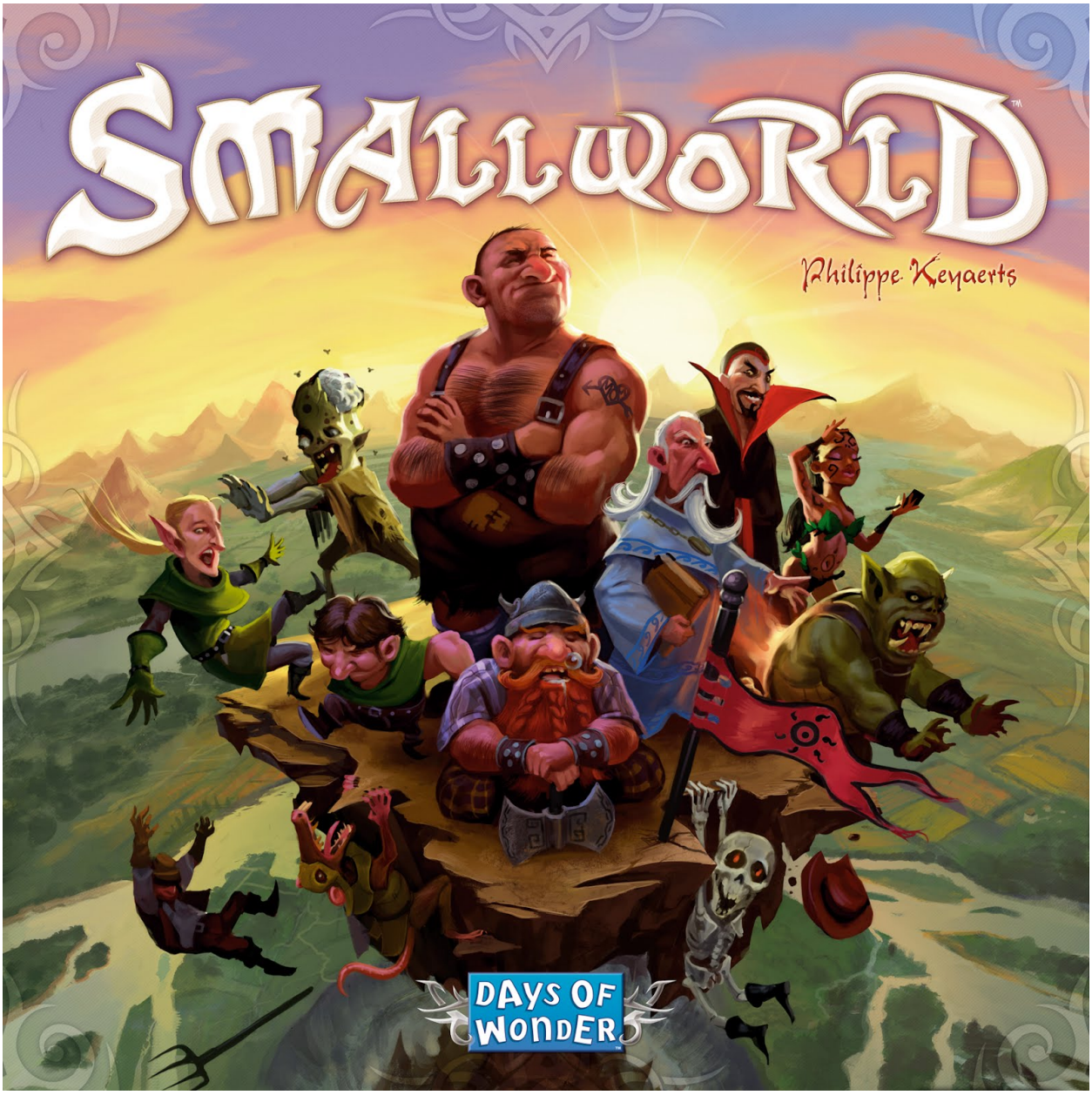

Figure 1: Illustration sur la boite du jeu

1 Objectif

1.1 Présentation générale

L'objectif de ce projet est de créer une version numérique du jeu de plateau "SmallWorld". Le projet est mené par 4 étudiants. Le jeu est codé en C++.

1.2 Règles du jeu

Le jeu choisi est un jeu de stratégie en tour par tour. Pour gagner, les joueurs doivent avoir à la fin du jeu le plus d'argent.

SmallWorld se joue grâce à un plateau représentant une carte de territoires, et des pions faisant office de troupes. Lorsque son tour arrive, le joueur doit attaquer les territoires de ses adversaires afin de conquérir un maximum de terrain. A la fin de son tour, il reçoit autant d'argent que de terrains qu'il possède.

Figure 2: Plateau de jeu SmallWorld

À ces mécaniques de bases s'ajoutent des systèmes de pouvoirs. Chaque joueur choisit une "espèce" pour ses troupes ainsi qu'un "pouvoir". Les combinaisons espèce/pouvoir sont définies de manière aléatoire. Chaque espèce a un effet différent, de même pour les pouvoirs. Cela peut aller d'un bonus de troupes lors d'un attaque à un bonus de récompense à la fin du tour. Les pouvoirs sont très variés.

Chaque joueur peut, s'il le veut, abandonner son espèce actuelle et en prendre une nouvelle. L'espèce abandonnée reste sur le plateau, mais est marquée comme "en déclin". Les territoires occupés par une espèce en déclin rapportent toujours des récompenses à son ancien propriétaire.

Tout l'intérêt du jeu réside dans la capacité à changer d'espèce au bon moment, et à choisir la bonne combinaison espèce/pouvoir parmi celles proposées.

1.3 Conception Logiciel

Présenter ici les packages de votre solution, ainsi que leurs dépendances.

2 Description et conception des états

2.1 Description des états

L'état global du jeu est centralisé dans la classe Game_State, qui contient toutes les informations nécessaires pour décrire la partie à un instant donné :

- la carte (Map) contenant l'ensemble des informations sur les zones de la carte avec ses specificités ainsi que les troupes des joueurs,
- la liste des joueurs (Player),
- les tribus (combinaison d'espèce et de pouvoir) disponibles via une pile (Tribe_Stack),
- des paramètres de gestion de tour comme le nombre de joueur actif ainsi que le nombre de rounds.

Chaque Player représente un joueur de la partie et possède :

- un identifiant unique,
- un ensemble de tribus actives qu'il contrôle,
- un compteur de points de victoire accumulés,
- les méthodes associées à la conquête ou au déploiement d'unités.

Une Tribe correspond à la combinaison d'une espèce et d'un pouvoir spécial, ce qui définit les capacités et bonus de la tribu. Chaque tribu possède :

- un nombre d'unités disponibles,
- des descriptions (Species_Description, Power_Description) déterminant ses effets,
- des méthodes associé aux tribes comme go_in_decline et autre qui seront explicités plus tard.

Les zones (Area) modélisent les régions de la carte. Elles contiennent :

- un biome (Area_Biome) et éventuellement des spécial tokens (forteresse, tanière, etc.),
- une liste de voisins, pour représenter les connexions de la carte,
- un propriétaire (Tribe), ici il est important que le propriétaire soit une Tribe et non pas un Player car un Player peut avoir plusieurs Tribe et que ces Tribe peuvent s'attaquer mutuellement,
- et des méthodes permettant la conquête ou le déploiement d'unités.

L'ensemble de ces zones est géré par la classe Map, qui stocke leur liste et permet le chargement depuis un fichier JSON (utilisé pour initialiser la carte).

Enfin, les effets spéciaux sont gérés par la hiérarchie Effects_Bundle :

- La classe abstraite Effects_Bundle définit une interface générique (apply_first_round_effect, apply_conquest_effect, etc.).
- Des classes concrètes comme Dwarf_Effects_Bundle, Ratmen_Effects_Bundle ou Giant_Effects_Bundle héritent de cette interface pour appliquer des bonus spécifiques. Il est important de préciser que le diagramme UML actuel ne contient pas toutes les classes concrètes car chacune d'entre-elles "viole" les règles du jeu ce qui complique leur implémentation.

2.2 Conception logicielle

Le diagramme UML présente l'architecture des classes du module state. L'organisation suit une approche orientée objet modulaire (plus que cruciale dans notre cas avec autant d'effet différent) :

- Game_State joue le rôle de façade, offrant un point d'entrée unique pour manipuler l'état global du jeu.
- Map et Area forment le modèle spatial, décrivant la topologie/état physique du monde.
- Player et Tribe constituent le modèle des acteurs de ce monde, reliés entre eux par composition.
- Species_Description, Power_Description et Effects_Bundle réalisent le modèle des caractéristiques et utilisent une hiérarchie d'héritage pour encapsuler les comportements spécifiques à chaque race ou pouvoir.

L'architecture du système centralise l'état global du jeu dans la classe Game_State, qui regroupe toutes les informations essentielles pour décrire la partie à un moment donné ce qui en fait une facade qui sera utile pour la suite. La carte (Map) agrège plusieurs zones (Area) connectées entre elles, offrant ainsi une structure modulaire qui reflète la topologie du monde de jeu. Cette modulation potentielle est importante car la map n'est pas un objet fixé sans modification. Des special token venant de pouvoir de classe modifient cette map. La classe contient également la liste des joueurs (Player), ainsi que les tribus, qui combinent espèces et pouvoirs, accessibles via une pile (Tribe_Stack). En outre, des paramètres de gestion des tours, tels que le nombre de joueurs actifs et le nombre de rounds, sont également inclus, garantissant une interface pour la manipulation de l'état du jeu. La hiérarchie d'Effects_Bundle encapsule divers comportements d'effets appliqués aux tribus, permettant une flexibilité dans la gestion des capacités spécifiques comme vu précedemment.

2.3 Conception logicielle: extension pour le rendu

Comme vu avec nos encadrants, l'architecture du module state est conçue pour être indépendante de l'engine. Les classes ne contiennent aucune logique d'affichage ni de physique pur du jeu ; elles ne stockent que des données et des états. L'intérêt de cette séparation est de ne pas exposer la structure interne de notre state dans le cas où ne serions amené à changer quelque chose, il ne faut pas que ces changements impactent la façon dont l'engine appele le state. Le module de rendu peut donc interagir avec cet état via des interfaces de lecture (ex. : positions des tribus, nombre d'unités sur chaque zone, propriétaire d'une zone, etc) sans altérer la logique du jeu.

2.4 Conception logicielle: extension pour le moteur de jeu

Le moteur de jeu utilise le module state comme base de données dynamique. Il peut :

- appeler les méthodes Game_State::conquer(), Player::redeploy_units(), ou Area::set_owner() pour faire évoluer la partie,
- interroger l'état courant pour déterminer les actions possibles,
- notifier le module de rendu pour mettre à jour l'affichage.

Cette conception permet une séparation claire entre la logique du jeu et les actions du joueur qui par ailleurs rend le projet facilement testable via des tests unitaires.

2.5 Ressources

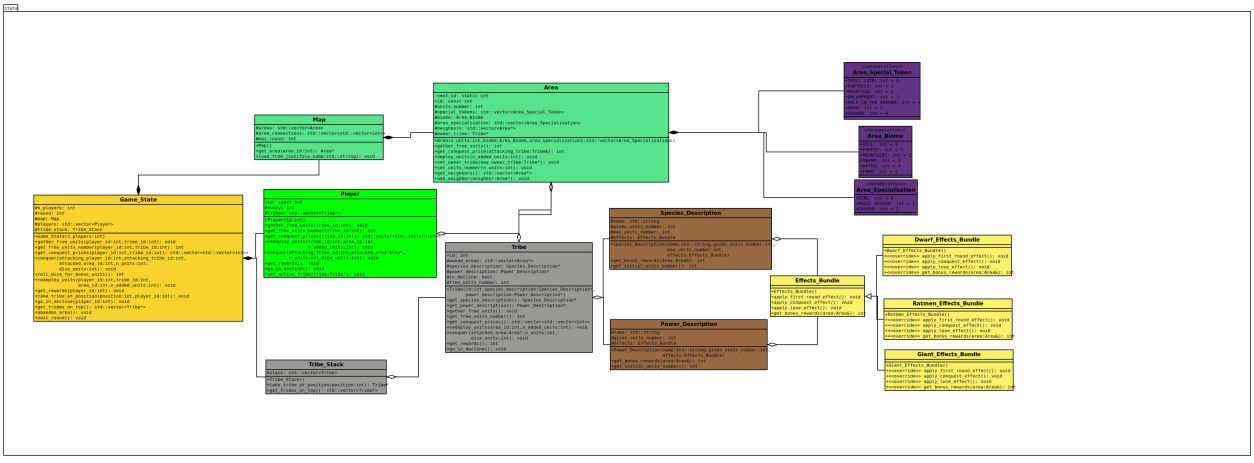

Figure 3: Diagramme des classes_d'état