Approche Objet - Rapport Projet

Alexis Bion, Groupe 2
Florian Claisse, Groupe 2

8 décembre 2023

Table des matières

1 Analyse

Ce projet consistait à développer les bases d'un jeu de stratégie mettant en avant la gestion d'une cité (avec ses habitants et ses bâtiments) et d'un stock de ressources lié à l'économie de ladite cité.

1.1 Comment gérer la croissance des habitants ?

La stratégie du jeu n'étant pas décrite dans le sujet et indiquée comme libre, une première question s'est posée à nous lorsque nous nous sommes mis à implémenter la gestion des habitants : Comment faire croître le nombre d'habitants dans la cité ?

Pour répondre à cette question nous avons donc réfléchi à plusieurs possibilités :
- Un système où les habitants se reproduisent au fur et à mesure du temps.
- Un système où des vagues d'habitants plus ou moins grosses arrivent tout les X jours.
- Des variantes plus ou moins différentes de ces deux systèmes (avec des possibilités de famines, de grèves, une contrainte sur la croissance en fonction du nombre de logements/du temps, etc.)
- Un système très simple où le nombre d'habitants est toujours égal à la capacité d'accueil de la cité, c'est-à-dire la capacité en habitants de tout les bâtiments.

Nous avons fini par choisir ce dernier système afin de partir sur une base simple (que nous aurions pu faire évoluer par la suite) et ainsi ne pas perdre trop de temps. Au final, nous l'avons gardé.

1.2 Comment assigner des habitants aux bâtiments ?

Il est indiqué dans le sujet que chaque bâtiment abrite un minimum/maximum d'habitants et un minimum/maximum de travailleurs et que le joueur peut donc faire varier le nombre d'habitants et de travailleurs entre ces minimum/maximums.
Cependant, pour rester cohérent avec notre système de croissance simpliste nous avons préféré faire en sorte que le nombre d'habitants dans un bâtiment soit invariable.

De plus, comme le tableau donné dans le sujet ne donnait pas de minimum/maximum mais juste un nombre fixe nous avons considéré ce nombre comme le minimum ET le maximum par défaut.
Ainsi nous avons fait en sorte qu'un bâtiment ne puisse être crée que si le nombre d'habitants libres (= qui ne travaillent pas déjà) dans la cité + le nombre potentiel d'habitants générés par la construction du bâtiment permettent de satisfaire ce nombre de travailleurs.
Aussi nous avons préféré que le nombre de travailleurs soit directement assigné au bâtiment sans que l'utilisateur ait à le faire manuellement.

Puisque le joueur ne peut pas enlever de travailleurs en-dessous du minimum et en rajouter au-dessus du maximum nous avons pris la liberté d'implémenter un niveau d'amélioration aux bâtiments. Tout bâtiment (sauf les bâtiments d'habitations) peuvent être améliorés du niveau 1 au niveau 5 et chaque niveau permet justement d'augmenter le nombre maximum de travailleurs. Ainsi le joueur peut assigner de nouveaux travailleurs (si la cité a assez d'habitants libres) et par la suite en retirer.
La production (et consommation) d'un bâtiment augmente en fonction du nombre de travailleurs supplémentaires qui lui est assigné.

1.3 Comment gérer la fin du jeu ?

Pour la fin du jeu, nous avons rajouté un bâtiment LaunchingPlatform utilisant des ressources auparavant non utilisées dans le tableau d'exemple fournie dans le sujet ("Cement" et "Tools"). Si ce bâtiment est amélioré jusqu'au niveau 5 et que son maximum de travailleurs lui est assigné alors la partie s'arrête et le joueur a gagné.

La partie peut aussi s'arrêter sur une défaite si le joueur n'a plus assez de ressources pour satisfaire la consommation des habitants/bâtiments lors d'un tour.

2 Conception - Design Patterns

2.1 BuildingBuilder et BuildingFactory

Pour générer des bâtiments nous avons utilisés les design patterns Builder et Factory.

Notre BuildingBuilder est composé de méthodes qui permettent de définir chaque attribut d'un Building puis de retourner ce Building nouvellement instancié avec une méthode build.
Notre BuildingFactory est composé de méthodes statiques qui permettent d'instancier des Building spécifiques (avec des attributs définis) à l'aide de BuildingBuilder.

Ces méthodes sont ensuite utilisées dans la méthode makeBuilding() de la classe City.
Cette méthode prend en paramètre un type de Building et va utiliser la méthode de BuildingFactory adéquate à l'aide d'un switch case pour ensuite retourner le bâtiment nouvellement crée.

Pour créer un nouveau bâtiment il faut donc rajouter un type de Building dans la classe Building, rajouter une méthode pour définir les attributs du bâtiment dans la classe BuildingFactory et enfin rajouter un case dans le switch de makeBuilding() dans la classe City.

2.2 ResourceFactory

Nous avons aussi utilisé le design pattern Factory pour pouvoir créer facilement des ressources. Cela nous est utile lorsque l'on cherche à accéder à la quantité associée à une ressource précise dans une HashMap (classe Resources), comme par exemple le stock du player.

À noter que pour cela on doit écraser la méthode equals() dans les différentes classes de ressources (càd les classes implémentant l'interface Resource) afin de faire la correspondance entre deux ressources identiques (mais dont les objets Java sont différents).

2.3 Observer (Publisher et Subscribers)

Pour gérer la production et la consommation de ressources nous avons utilisé le Design Pattern Observer avec une interface ResourcesManager ayant le rôle de "Publisher" et des interfaces ResourcesProvider et ResourcesConsumer ayant le rôle de "Subsribers".

ResourcesManager est une interface implémentée par le joueur (Player).
Les interfaces ResourcesProvider et ResourcesConsumer sont implémentées par les bâtiments (Building) puisqu'ils produisent/consomment des ressources qui doivent être ajoutées/supprimées dans le stock du joueur. L'interface ResourcesConsumer est aussi implémentée par une classe Citizen qui représente les habitants de la cité. En effet, les habitants consomment eux aussi des ressources (de la nourriture).

Ainsi, à chaque fin de journée dans la cité, le joueur (aka le ResourcesManager de la cité) notifie ces ResourcesProvider et ResourcesConsumer pour qu'ils ajoutent ou suppriment des ressources dans/de son stock.