Projet - Approche objet

Kerim Canakci (22018932), Corentin Drezen (22309435)

Décembre 2023

Analyse

On doit concevoir un jeu qui doit permettre à un joueur de gérer des **Resource**, des **habitant**s et des **Building**s, tout en cherchant à maximiser leur économie.

Une classe **Resource** doit permettre de représenter les différentes ressources disponibles dans le jeu (*Gold, Food, Wood, etc*).

- Comment faire en sorte qu'une **Resource** puisse représenter différentes sortes de ressources ?
- Comment doit-on gérer leurs quantités ?

Une classe **Building** pour représenter les bâtiments. Ils doivent contenir un nombre prédéfini d'**habitant**s et/ou de **travailleur**s. Un **Building** peut produire et/ou consommer des **Resource**s proportionnellement au nombre de travailleurs. Il doit être construit en une durée avec des **Resource**s.

- Comment gérer la différence entre un **Building** "usine", "logement" ou "logement d'usine" ? Comment différencier les **travailleur**s des **habitant**s si une "usine" est aussi "logement" ?
- Comment gérer la production/consommation de **Resource** suivant le **Building** ?
- Comment doit-on faire progresser la construction ?

Des **habitant**s peuvent être **travailleur**s et doivent être logés dans des **Building** "logement" ou "logement d'usine". Ils doivent consommer une **Resource** "Food" par unité de temps.

- Un **habitant** peut-il être simplement un entier ? A-t-on besoin de savoir si un **habitant** est **travailleur** et s'il est logé ?

Une classe **Manager** doit orchestrer la consommation/production des **Building**s et **habitant**s et permettre aux joueur de construire ou détruire des **Building**s, d'y ajouter ou retirer des **travailleur**s.

— Comment doit-on représenter les **Resource**s, **Building**s et **habitant**s pour qu'ils soient facilement modifiables par cet objet ?

— Comment faire en sorte que l'utilisateur puisse intervenir dans cette gestion ?

Il doit y avoir une **interface utilisateur** qui permette à l'utilisateur d'intervenir sur **Manager**. Chaque saisie au clavier doit correspondre à une unité de temps ou une action.

— Comment interpréter les différentes commandes entrées par l'utilisateur ?

Nous devons donc trouver une architecture qui convienne aux différents objets "**Resource**", "**Building**", et "**habitant**", qui permettent à l'utilisateur d'interagir et qui respecte l'approche objet.

Conception

Voici le diagramme de classe réduit de notre projet :

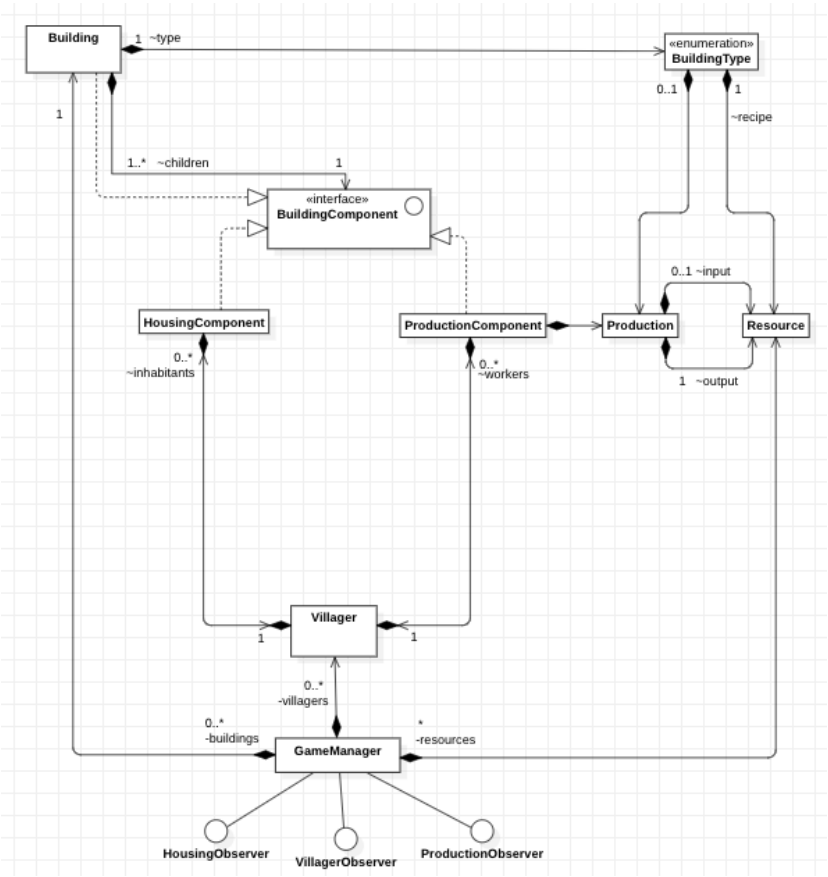

FIGURE 1 – Diagramme UML résuite de notre projet

Pour faire en sorte qu'une **Resource** puisse représenter différentes sortes de ressources on a fait un enum **ResourceType**, ainsi notre **Resource** est composé d'un attribut 'type' et d'un attribut 'quantity' (int) pour gérer sa quantité.

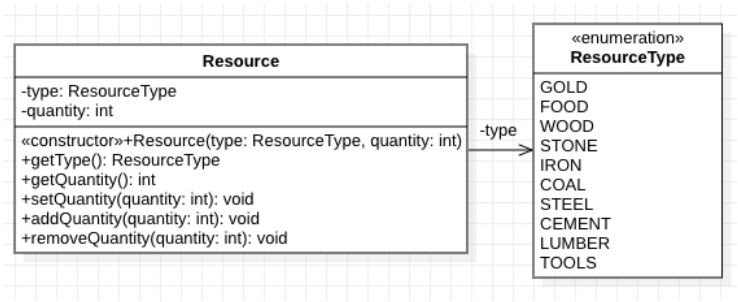

FIGURE 2 – Représentation des classes Resource et ResourceType

De ce fait les **Resource** gérées par le **Manager** pourront être un Hash-Map¡**ResourceType**, **Resource**¿ et les production/consommation/blueprint des **Building** pourront être des tableaux de **Resource** indépendants. (À défaut de faire un nouvel objet “Stockage” ou “Recette” qui facilitera encore plus son utilisation.)

Pour notre implémentation des bâtiments, nous avons décidé d’utiliser un composite. Les bâtiments **Building** peuvent être composés à la fois de parties logement et de partie usine et on pourrait vouloir ajouter d’autres sortes de composants de bâtiment comme des parties loisir ou magasin ou encore des parties militaires qui pourraient former des villageois en soldats. Les parties de bâtiments pourraient aussi être composées de pièces et on pourrait aussi vouloir composer les logements de lit et les usines de différentes sortes de machines. Cependant, notre implémentation actuelle n’a pas réellement besoin d’un composite et on a pensé plusieurs fois à le remplacer par un ensemble de classes plus simples avec des ‘parties’ de **Building** directement navigables et moins de redondance de code ou de méthodes surchargées vides.

Voici notre de diagramme de classe pour l’objet **Building** :

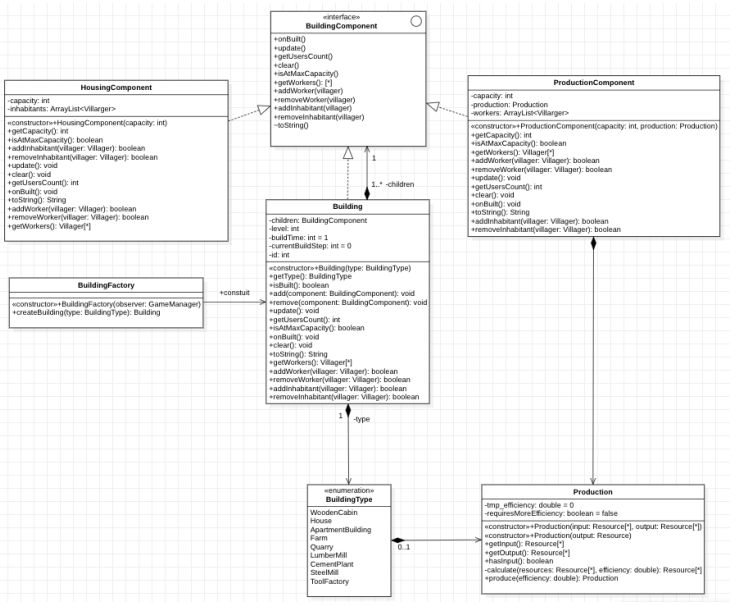

FIGURE 3 – Diagramme UML de notre projet

On peut voir que les composite **Building** peuvent être construits avec un factory à l'aide d'un enum **BuildingType**. Il sert à énumérer et à mettre en correspondance des "recettes" de **Building** composé de **Resource** et éventuellement de **Production** si c'est un bâtiment qui produit. Notre **BuildingFactory** va s'occuper de construire les composants et de leur donner une capacité maximum de villageois en fonction du **BuildingType** choisi, le reste de la construction est réalisé par le constructeur de **Building**. De ce fait, pour rajouter un bâtiment au jeu on a simplement à rajouter un élément à l'énumération (ie. Nom(new **Resource**[], new **Production**)) puis d'ajouter ses composants et leurs capacités dans le factory en implémentant son cas switch (pour l'élément de l'enum qui lui correspond).
La production de nos **ProductionComponent** est définie par le **BuildingType**, elle est géré par le **ProductionComponent** qui vérifie si il y a besoin de la calculer, et par un objet **Production** qui contient les ressources par défaut à produire et à consommer et qui gère le calcule de la production en créant une nouvelle instance de **Production** qui contient alors les ressources consommés et produites proportionnel au nombre de travailleurs. Comme nous pouvons avoir des quantités de **Resource** produites inférieurs à 0 avec une proportion trop faible de travailleurs (quotient d'efficacité) et que nous avons souhaité continuer d'utiliser des entiers dans la classe **Resource**, on a décidé de faire une somme de cette proportion à chaque tour tant qu'une quantité est en dessous de 0. On produit alors une quantité de **Resource** supérieure à 0 au bout de plusieurs tours.

La construction d'un bâtiment est géré principalement dans **Building**. Lors de sa mise à jour (méthode update), il incrémente un compteur d'une unité de temps et ne met pas à jour les composants tant qu'il n'a pas atteint le temps de construction défini dans son **BuildingType**.

Afin de différencier les **travailleurs** des **habitants** si une "usine" est aussi "logement", l'interface de notre composite a des méthodes pour ajouter et supprimer des **travailleurs** et des méthodes pour des **habitants** qui sont surchargés dans **Building** et dans les composants. Il y a une ArrayList de Villager (habitant/travailleur) dans **HousingComponent** et dans ProductionComponent, ainsi chacun s'occupe de sa liste et est implémenté par un return quand les méthodes de l'interface ne lui correspondent pas. Toutefois, comme évoqué précédemment, on aurait aimé éviter cette redondance, peut être avec une classe abstraite supérieur à **HousingComponent** et ProductionComponent mais l'intérêt d'utiliser un composite serait moins visible tant que l'on ajoutera pas d'autre composant plus différent de ceux ci.

Pour représenter les habitants et travailleurs on a fait une classe **Villager**. On ne voulait pas qu'ils soient de simple **int** car on voulait pouvoir les gérer plus 'verbalement' et pouvoir les placer facilement selon si leurs propriétés, et pouvoir plus tard leurs donner des noms.

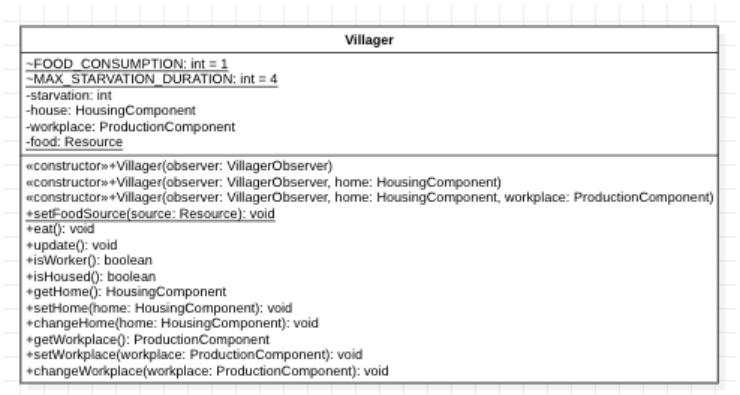

FIGURE 4 – Représentation de la classe Villager

Le **Villager** va retirer une quantité 1 de la **Resource** 'food' à chaque tour avec sa méthode appelée à chaque mise à jour. Si il n'y a plus de nourriture, le villageois va entrer en famine et va mourir au bout d'une durée en unité de temps. Grâce au constructeur on peut affecter le villageois à un composant de **Building** dès sa construction.

On a choisi d'avoir une classe **GameManager** ("Game" pour plus de clarté) singleton car on n'a besoin d'un seul objet qui orchestre le jeu :

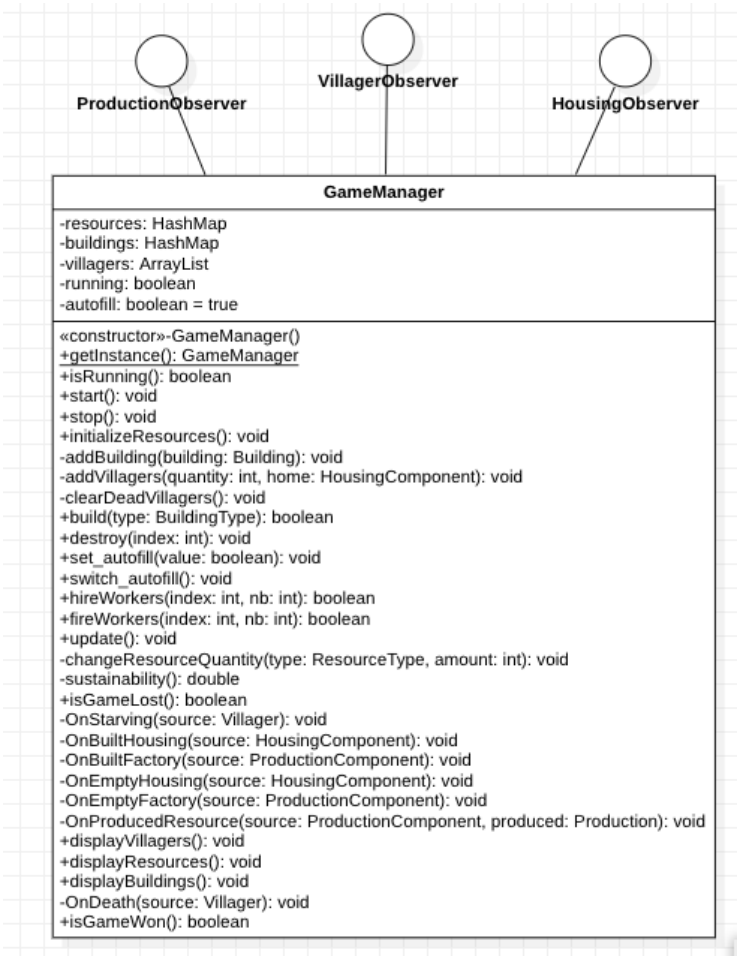

FIGURE 5 – Représentation de la classe GameManager

Sa méthode update qui appelle la méthode update de chaque instance d'objet du jeu va être appelée dans une boucle while extérieur, ainsi chaque appel à **GameManager**.update() vaudra un tour ou une unité de temps. Nous avons choisi de stocker les **Resource** et les **Building** dans un HashMap pour pouvoir y accéder suivant leur **ResourceType** et leur identifiant respectivement. Nous stockons les **Villager** dans un ArrayList. Les objets **HousingComponent**, **ProductionComponent** et **Villager** sont tous surveillés par cette classe à l'aide des interfaces Observer correspondant. Au lieu d'avoir de multiples éléments qui s'abonnent à ses observables on a choisi de faire un unique observer (**GameManager**) qui s'abonne à toutes les instances d'observables. Par conséquent, nous pouvons agir suite au évènement déclenché par chacun des objets du jeu, certains des événements ne provoquent rien mais pourraient être utiles plus tard.

Il y a une méthode pour mesurer le taux de soutenabilité/durabilité de la partie c'est-à-dire un quotient de nourriture/nombre de villageois. C'est en lien avec la possibilité de famine. Ainsi on peut décider de faire apparaître des villageois dans un logement avec de l'espace disponible lorsqu'un bâtiment est vide mais uniquement si il y a suffisamment de nourriture pour les nourrir.

Il y aussi des méthodes pour gérer l'initialisation du jeu et détecter une partie perdu ou gagner. Par exemple, si il n'y a plus de villageois et plus de nourriture ou plus suffisamment d'or et qu'il n'y a plus de Quarry le joueur a perdu.

On a dû gérer le chômage et les villageois sans abri dû aux interactions du joueur. Des villageois sans-abri seront automatiquement ajoutés au premier logement vide ou à la construction d'un bâtiment et les villageois au chômage seront ajoutés à la première usine en manque de travailleurs. Cette fonctionnalité peut être désactivée.

Il y a aussi des méthodes pour exécuter les commandes utilisateur. On a choisi de faire en sorte que l'utilisateur puisse effectuer plusieurs actions par tour, et puisse passer au tour suivant en entrant uniquement 'Entrée' afin de permettre plus de possibilités de gameplay.

Les actions des utilisateurs sont gérées depuis le point d'entrée de l'application Main.java qui va utiliser le singleton **GameManager** et interpréter les commandes avec une énumération **CommandeType** qui implémente Commande, une interface fonctionnelle. Cela permet d'exécuter directement des éléments énumérables. Les entrées sont parsées dans **CommandeType** dans les fonctions correspondant aux entrées qui peuvent aussi utiliser **GameManager** facilement car c'est un singleton, il est alors assez simple de rajouter une commande.