ECOLE
POLYTECHNIQUE
DE BRUXELLES

UNIVERSITÉ LIBRE DE BRUXELLES

INFO-H2001
Remise finale
Groupe 66

Auteur:
Julien VAN DELFT
Charly WITMEUR

2018-2019

1 Architecture

1.1 MVC

L'architecture du jeu est organisée via un Design Pattern MVC (Model-View-Controller). Ce design pattern a l'avantage de fournir une séparation claire des différentes responsabilités au sein du programme. Le découplage est bien plus clair entre les fonctionnalités front-end (ce que l'utilisateur voit en lançant le programme, l'interface graphique) et les implémentations back-end (ce qui est calculé en fond, tous les modèles, tous les algorithmes,...). De plus, la gestion des évènements est bien meilleure car les évènements sont localisés dans le controller qui est indépendant de la vue. Il est donc bien plus aisé d'organiser son code via cette architecture.
Les cartes, les fenêtres et le HUD (*Head-up display*, les barres montrant les différents besoins, l'inventaire à l'écran,...) sont gérés par la View. Les touches appuyées par le joueur sont récupérées par le Controller qui l'envoie au Model. Enfin ce dernier s'occupe de toutes les données, de toutes les interactions au sein du programme et de toutes les classes du jeu.

Model :

Le modèle contient avant tout une superclasse *GameObject*. Toutes les classes présentes dans le modèle représentant des objets du jeu (personnage, nourriture, meubles, etc) sont des classes filles de *GameObject*. Cette dernière possède une position, une taille et une image représentant l'objet. Ces deux paramètres permettent de situer ou placer l'objet sur la carte. Cette superclasse permet d'effectuer des méthodes indépendantes du type de l'objet (par exemple tous les objets ont un sprite qui doit être dessiné sur la carte).

Dans cette classe, il y a encore d'autres grandes classes regroupant la majorité des objets à savoir *Building*, *ActivableObject*, *Block* et *ImageDraw*. La première contient tous les bâtiments, leur seul point commun étant une porte d'entrée. La seconde, contenant tous les objets avec lesquels le joueur peut interagir, définit le type de l'objet (par exemple *sit* pour un sofa) et les utilisateurs possibles (par exemple les cigarettes sont seulement utilisables par les adolescents). La troisième regroupe tous les objets bloquant le passage du personnage et la dernière regroupant tous les objets de décoration donc ne bloquant pas le passage et avec lequel le personnage ne peut pas interagir. Cette première distinction permet déjà d'organiser le code entre les objets. De nouveau, ces grandes classes permettent d'effectuer des méthodes sur tous les buildings par exemple indépendamment de leur type (on initialise une porte à tous les bâtiments peu importe leur classe).

On peut encore noter les classes importantes présentes dans *ActivableObject* tels que *Sums* et *DeletableObject*.Les sums sont les personnages jouables dans le jeu. Ils ont des besoins (faim, énergie, hygiène, bonheur et toilette) qu'il faut satisfaire sans quoi le sums meurt petit à petit. Chaque sums possède une *HashMap* contenant tous les autres sums. Elle sert à gérer l'affection entre les sums. Les sums dans le dictionnaire sont associés à un entier représentant l'amour que leur porte le sums possédant la *HashMap*. Au delà d'un certain niveau d'affection, certaines actions sont débloquées (comme faire des enfants). Ils ont aussi une liste de *GameObjet* représentant l'inventaire. Les sums sont découpés en 4 sous-classes correspondant à des tranches d'âges différentes, ayant des fonctions dédiées (seuls les enfants peuvent jouer avec un jouet par exemple). Les *DeletableObject* sont les objets utilisables et disparaissant après un certain nombre d'utilisation (comme la nourriture).

La classe la plus importante se trouve dans le modèle, il s'agit de la classe *Game* qui s'occupe d'instancier toutes les cartes et tous les objets, de lancer des timers et des threads afin que les personnages se déplacent et que le temps passe. C'est également elle qui gère le changement de carte lorsque le joueur passe une porte. L'intérêt de cette classe centrale est qu'il est très aisé de modifier le déroulement du jeu car il est presque entièrement défini dans cette classe. C'est cette classe qui va être sauvegardée car elle contient tous les objets du jeu.

Les cartes sont aussi une classe prépondérante dans le modèle. En effet, chaque carte possède la liste des objets sur cette dernière. Elles ont également une matrice de booléens de taille identique à la carte. Les objets bloquent des positions en se plaçant sur la carte. Ils sont marqués par des *true* dans la matrice. Les cartes gèrent aussi le placement des meubles (si il y a une possibilité de placer des meubles). L'avantage que chaque carte soit un objet à part entière vient du fait que les objets qui sont sur cette carte sont aisément atteignable et il est aussi facile

d'ajouter les objets sans qu'ils se mettent l'un sur l'autre grâce à la matrice de booléen.

Les cartes (les objets sur les cartes et les cases formant le sol et les bords de la carte) sont dessinées via des fichiers texte dans lesquels chaque case est représentée par un caractère. Les objets à ajouter à la carte sont indiqué en dessous des caractères suivis de la position de l'objet. La classe *MapReader* s'occupe de parcourir le fichier texte et d'ajouter les images à dessiner dans une liste que la view va récupérer. Les objets sont créés et ajoutés aux objets de la carte. Les fichiers textes sont pratiques car il est facile de modifier la carte en modifiant les caractères dans le fichier.

Il y a également une classe de Constantes qui sert à toutes les autres classes. Toutes les constantes sont stockées dans cette classe ainsi que toutes les images. Les sprites sont dans les constantes pour que le jeu ne les charge qu'une seule fois, lors du démarrage. La classe Constantes contient uniquement des variables static (et finale pour les variables autres que les images). Les constantes permettent de modifier le jeu simplement en modifiant les valeurs dans cette classe.

Il y a encore quelques classes à part dans le modèle servant au bon fonctionnement du jeu, comme la classe *Sound* permettant de jouer des sons et des musiques pendant le jeu et les classes *Load* et *Save* permettant de sauvegarder l'état du jeu et de reprendre le jeu à l'endroit où le joueur l'a laissé. Ces sauvegardes sont faites via l'interface *Serializable* qui permet de sérialiser un objet et de l'écrire dans un fichier *.serial*. Comme expliqué précédemment c'est l'objet game qui est sérialisé. Pour éviter que la sauvegarde soit trop lourde, les images et les arguments inutiles car recréés lors de l'initiation du jeu (par exemple les timers) ne sont pas gardés en mémoire (via le mot-clé *transient*).

Controller :

Il y a assez peu de choses à dire sur ce package car il ne contient que quelques classes (il y a seulement deux contrôleurs, le clavier et la souris et plusieurs classes servant de *Listeners* aux classes de la *view*). Dans le modèle MVC, la view instancie les boutons mais c'est bien le contrôleur, qui contient les *Listeners*, qui envoie les informations au modèle lors de la pression des boutons à l'écran. L'avantage de laisser la view gérer uniquement la GUI et de laisser les contrôleurs en *Listener* est d'alléger la tâche de la view qui est assez lourde car elle doit dessiner tous les objets et s'actualiser assez souvent.

View :

La view contient surtout des classes de *JPanel*. La classe centrale de ce package est la classe *Window*. Il s'agit d'une *JFrame* qui va contenir tous les *JPanels*. Les différents panels sont organisés dans la *Window* via un *CardLayout*. L'avantage de ce type de Layout manager est qu'il est très aisé de passer d'un panel (ou groupe de panels) à un autre. De fait, il suffit de faire passer la *Window* à la "carte" suivante ou à la précédente pour changer le panel affiché.

Les panels principaux sont les menus, le *MapDrawer* qui dessine toutes les tiles et tous les sprites et les panels qui instancie les boutons, le *HUD* et dessine l'inventaire. Le groupe de panels qui s'occupe du jeu est constitué de 4 panels. Le premier s'occupant de la carte, le deuxième du *HUD* et les deux autres des boutons et de l'inventaire à afficher. Les différents panels sont organisés différemment ce qui permet plus de liberté dans le placement des *JComponents* (certains utilisent des *GridBagLayout* pendant que d'autres utilisent simplement des *BorderLayout* par exemple).

1.2 Desing Pattern

Singleton :

Pour que toutes les classes aient accès aux objets qui ne sont créés qu'une seule fois, le design pattern *Singleton* a été utilisé. Donc, les constructeurs sont passés en privé et une fonction static permet de récupérer l'instance de

l'objet créé si ce dernier a été construit ou de le construire sinon. Toutes les classes récupère donc le même objet. C'est très utile par exemple pour les changement de cartes. Les portes (classe *Door*) vont faire changer la carte dans le game. Parmi les classes qui possèdent un singleton, on peut notamment pointer le *Game*, les différents panels et la *Window*.

Observers :

Pour soulager un maximum le processeur du joueur, le Design Pattern Observer a également été mis en place. Les observers permettent d'actualiser la View uniquement lorsqu'il y a un changement notable (activation d'un objet, mouvement, etc). Ceci a pour avantage considérable d'éviter de devoir actualiser la View tous les X temps ce qui utilise beaucoup plus le processeur du joueur. Chaque objet est donc lié à un observer (si celui-ci est activable ou cassable). Lors de la création de l'objet dans la classe *Map*, l'objet se fait associer à un *DeletableObserver*. Certains objets possèdent des "points de vie" ce qui leur permet de ne pas disparaître lors de la première utilisation (par exemple les cigarettes sont utilisables 5 fois avant de se faire supprimer).

Factory :

Une *Factory* est utilisée (via la classe *BlockFactory*). Elle est surtout utile lors de la lecture des fichiers textes des cartes contenant les objets à placer. Le *MapReader* lit le fichier et envoie le String récupéré à la factory qui s'occupe de crée l'objet correspondant. Puis, le MapReader le récupère et l'envoie à la carte qui l'ajoute à ses objets. Ce design pattern permet une gestion bien plus dynamique car elle lit directement l'objet dans le fichier.

2 Fonctionnalités implémentées

1) Menu : Le jeu commence par ouvrir un menu proposant plusieurs choix au joueur. Ce dernier peut décider si il lance une nouvelle partie, si il continue une partie via une sauvegarde ou si il quitte le jeu. Le choix s'effectue via des *JButtons*. Fonctionnalité faite par : **Julien van Delft**.

2) Sauvegarde : Le joueur peut sauvegarder sa partie à tout moment, via un menu, pour pouvoir quitter le jeu et le reprendre plus tard. Pour ce faire, la positions des Sums, leur inventaire, leurs attributs et les positions des objets dans le jeu sont mis dans un fichier *.serial* afin de les garder en mémoire. Le jeu peut relancer la partie en lisant les informations dans ce fichier, en créant le jeu et en donnant toutes les valeurs sauvegardées à ce jeu. Fonctionnalité faite par : **Julien van Delft**.

3) Carte type Labyrinthe : Le jeu se divise en plusieurs cartes chacune accessible depuis la carte de base via des portes, des chemins ou des escaliers. Les cartes sont générées en utilisant le principe du tile mapping, c'est à dire en subdivisant la carte en un nombre définis de carrés appelés tiles. Chaque carte est stockée dans un fichier texte composée de caractères représentant chaque tile. Elles sont ensuite lues et dessinées à l'écran lorsque ce sera nécessaire. Les cartes sont lues une seule fois lors de l'instanciation des cartes lors de la création du jeu et ensuite les tiles sont gardés comme attribut de chaque objet *Map*. Fonctionnalité faite par : **Julien van Delft**.

4) Inventaire : Les Sums disposent chacun d'un inventaire de taille fixe. A chaque instant, il est affiché à l'écran, l'inventaire du Sums contrôlé par le joueur. De plus, le joueur peut cliquer sur les images de l'inventaire pour utiliser les objets. De plus, en fonction des objets sur lesquels le joueur a cliqué, des *JButtons* correspondant à l'objet sont affichés (par exemple si le joueur clique sur la pomme, un bouton "eat" apparaît au dessus de l'inventaire. Fonctionnalité faite par : **Julien van Delft**.

5) Évolution des Sums : Le temps passant, les Sums grandissent. Ils passent d'abord par l'enfance, puis par l'adolescence, l'âge adulte et enfin l'âge avancé. La suite logique étant qu'après l'âge avancé, les Sums redeviennent poussières. Chaque étape des Sums est accompagnée de fonctions dédiées. Par exemple, les enfants sont les seuls à pouvoir jouer avec des jouets. Fonctionnalité faite par : **Charly Witmeur et Julien van Delft**.

6) Famille : Les Sums sont divisés en famille. Ces dernières sont liées aux maisons des Sums. Ainsi, si un Sums va travailler, il ramène de l'argent à son foyer. De même, les dépenses des membres de la famille font diminuer l'argent de la maison. Si l'argent de la maison devient suffisant, les Sums peuvent aller s'acheter une nouvelle maison plus grande au magasin. Fonctionnalité faite par : **Charly Witmeur**.

7) Besoins : Les Sums ont des besoins changeant en fonction de leur âge. Néanmoins chaque Sums a des besoins similaires : ils ont besoin de se nourrir, de se laver, d'aller aux toilettes, de dormir et d'être heureux. Pour être heureux, ils doivent effectuer certaines actions pour leur redonner le sourire (comme jouer avec un jouet pour un enfant ou jouer avec le chien). Les besoins augmentent avec le temps via l'utilisation de timers (plus le temps avance et plus le Sums a faim ou besoin de dormir par exemple).Fonctionnalité faite par : **Julien van Delft et Charly Witmeur**.

8) Menu In Game : A tout moment, le joueur peut mettre le jeu en pause via un menu in game. Le fond de ce menu est animé via un gif. Plusieurs choix sont disponibles au joueur via des *JButtons* notamment sauvegarder, quitter ou une page d'aide fournissant des informations sur les commandes.Fonctionnalité faite par : **Julien van Delft**.

9) Musique : Le jeu possède une musique qui tourne en fond pendant que le joueur joue sa partie. De plus, quelques objets font des sons lors de leur activation (exemple nourriture, toilettes,...) Fonctionnalité faite par : **Charly Witmeur**.

10) Horloge : Une date est affichée à l'écran en permanence montrant le temps avançant dans le jeu. Le joueur commence le jeu à minuit le premier janvier 2019 et le temps passe à partir de la création du jeu. A noter que le temps dans le jeu passe nettement plus vite que le temps dans la vie réelle (une minute dans le jeu = 0,3 secondes dans la vraie vie). Fonctionnalité faite par : **Julien van Delft**.

11) Interactions : Les Sums ont une barre d'affection qui peut augmenter en fonction des interactions entre eux telles qu'offrir un bouquet de fleurs. De nouvelles actions sont alors possible une fois l'affection ayant atteint un seuil, telle que la reproduction pour les Sums adultes. Fonctionnalité faite par : **Charly Witmeur**.

12) Contenant : En plus de l'inventaire, les objets peuvent être stockés dans des contenant tel qu'un frigo. Une fois le frigo ouvert, son contenu apparaît à l'écran et le joueur peut décider de prendre un objet dans le frigo ou bien d'y déposer un objet depuis son inventaire. Fonctionnalité faite par : **Julien van Delft**.

13) Magasin : Un magasin est accessible depuis la carte centrale du jeu. Ce magasin est composé de plusieurs rayons à côté desquels un prix est indiqué via une pancarte. Les joueurs peuvent donc sélectionner le produit à acheter dans l'étagère et celui-ci est directement mis dans son inventaire, après que le montant correspondant au produit ait été prélevé au joueur. Fonctionnalité faite par : **Charly Witmeur**.

14) Personnalisation de maison : Lorsqu'un joueur entre pour la première fois dans sa maison, il doit choisir où placer différents éléments tel quel les lits, le frigo ou le canapé. Pour chaque objet, le joueur clique à l'endroit où il veut le placer et, si l'endroit est disponible, il peut choisir l'orientation de cet objet. Les différents placements sont alors enregistrés pour les prochaines fois où le joueur entre dans la maison. Fonctionnalité faite par : **Julien van Delft**.

15) Déplacement automatique des Sums non joué : Les personnages qui ne sont pas joués se déplacent aléatoirement sur la carte. Les chiens se déplacent de la même manière. De temps en temps, un Sums va jusqu'à une porte de la carte et la traverse. De même, un Sums sur une autre carte choisi aléatoirement revient sur la carte montrée à l'écran à intervalle de temps régulier. Ces déplacements sont gérés via des Threads et un algorithme de recherche du plus court chemin, *AStar*. Fonctionnalité faite par : **Julien van Delft**.