Rapport projet Sokoban

Jeulin Elian – S2D’

Ce rapport décrit la conception et la réalisation d’un programme de jeu en mode texte.
Dans ce jeu, le joueur* doit faire que chaque caisse* recouvre une cible* pour que le niveau soit validé.

*Les caractères associés à chacun sont:
- ‘C’ pour la caisse
- ‘P’ pour le joueur
- ‘X’ pour la cible
- ‘#’ un mur permettant de délimiter le terrain de jeu
- ‘.’ une case vide

Pour cela, nous devions programmer en plusieurs temps.
Dans un premier temps, nous devions faire que le jeu marche bien avec la création d’une carte faites directement dans le programme.
Dans un second temps, la création de la carte devait se créer grâce à la lecture d’un fichier texte.
Enfin, les cartes devaient être gérées par une base de donnée.

Plan du rapport

Rapport Sokoban – Jeulin Elian – S2D'

1. Analyse

a. Spécification des besoins

Le programme doit fonctionner en mode texte. Le but est d'écrire un programme Java permettant de jouer au jeu Sokoban.
Ce jeu fonctionne de la manière suivante:

- •Un terrain rectangulaire. Certaines cases sont occupées par des murs, qui bloquent le passage.
- • Sur le terrain, il y a des caisses.
- • Un personnage déplacé (4 directions) par le joueur qui peut aller
  - dans une case vide voisine
  - ou pousser une série de caisses, si il y a une case vide derrière
- • objectif : amener toutes les caisses vers des destinations indiquées.

Les objectif généraux du programme sont de permettre à un joueur de mener des parties sur des plateaux différents. Les plateaux seront obtenus à partir de différentes sources, soit des fichiers, soit une base de données.

b. Conception

Choix de conception:
Pour représenter l'état de l'interaction avec un joueur, j'utilise la classe Game, sur lequel le joueur agit en:

- • Lui demandant que veut-il faire entre:
  - Jouer au niveau 1 qui correspond à l'objectif 1
  - Jouer au niveau 2 qui correspond à l'objectif 2
  - Créer son propre niveau qui est un bonus que j'ai crée
  - L'utilisation des bases de données qui correspond à l'objectif 3
  - De quitter le jeu

Pour représenter l'interaction entre l'administrateur et la base de donnée, j'utilise la classe Administrator, sur lequel l'adminitrateur agit en:

• Lui demandant que veut-il faire entre:
- Créer la base de donnée
- Lister tout les plateaux présents dans la base de donnée
- Montrer le contenu d'un plateau en fonction de son id

- Ajouter un plateau dans la base à partir d'un fichier
- Supprimer un plateau de la base de donnée
- Quitter le jeu

c. Données et traitements

Pour une partie

• Le programme lance les interactions avec le joueur grâce à une boucle do while qui vérifie que l'entrée du joueur est bonne.

• Par l'intermédiaire d'un switch il va créer un plateau différent en fonction de la demande du joueur si il n'y a pas d'erreur de création comme par exemple que la base de donnée n'est pas instanciée.

• Il exécute une boucle tant que la partie n'est pas finie, c'est à dire que toutes les positions des cibles ne sont pas recouvertes par des caisses.

• Il avertie le joueur quand il a passé le niveau, puis par l'intermédiaire d'un while va relancer la boucle pour afficher le menu etc.

2. Réalisation

a. Organisation en packages

J'ai choisi de segmenter le jeu en quatre paquets:
- Le package Game gère plutôt l'affichage et le fonctionnement du jeu.
- Le package Board gère le plateau, les cases et les assets.
- Le package File gère tout ce qui est relatif aux fichiers.

- Le package Database gère la base de donnée du jeu.

Ces choix ont été fait dans l'optique de bien tout séparer les objectifs, les classes Board, File et Database font écho aux trois objectifs bien distincts de ce projet et doivent donc être séparé. Enfin le package Game est la relation entre les trois donc doit forcément être aussi un package à part.

Je ne pense pas qu'il faille encore plus segmenter car cette segmentation là me paraît optimal et cohérente et je ne vois pas ce qu'il pourrait y avoir de plus à segmenter.

b. Package Game

Classe Player: Cette classe étend la classe Menu. Elle gère le déroulement du jeu (menu, fin de partie, etc…). Cette classe est composée du main et cette dernière gère le déroulement du jeu du joueur. Elle appelle la classe Game pour gérer la partie.

Classe Game: Cette classe est construite à partir d'un plateau. Elle gère notamment tout ce qui est les mécanismes du jeu. Les deux méthodes permettant de gérer cela sont oneTurn et playerMvt. La première permet de prendre les entrée du joueur et vérifier si l'entrée est valide. Puis, si elle est valide elle appelle la méthode playerMvt. Cette dernière va donc effectuer l'entrée du joueur et faire déplacer le joueur dans une direction si elle est possible. Son constructeur est protected car seulement la classe Player qui est dans le même package doit l'implémenter.
J'ai choisi de faire cette classe pour gérer tous les fonctionnements internes relatifs au jeu sans «polluer» la fonction main.

Classe GamePlayerLeaves: Cette classe étend Exception. Elle permet de gérer quand un joueur quitte le jeu. Cette exception a été faites pour faciliter et rendre plus agréable le code.

Classe Menu: Étant donné que ce programme comporte deux main et donc deux menus j'ai crée cette classe pour ne pas dupliquer du code. Cette classe comporte donc toutes les méthodes qui permettent l'affichage général du menu. Ses méthodes sont protected car elles doivent être utilisées seulement pas les classes héritants de Menu.

c. Package Board

Classe Board: Classe qui gère le plateau, son affichage, sa création. Cette classe a été faite pour s'occuper du plateau qui est un tableau de tableau de cases (classe Tiles). Elle est construite avec un nom, une taille et une hauteur. Son constructeur est public car la plupart des classes doivent créer un nouveau plateau.

Classe Tiles: Ce sont les cases du plateau, elle contienne une asset qui les attribue un affichage. Faire un tableau de tableau de cases me paraissait plus claire et plus intuitif que des collections prenant les positions de chacune des assets différentes qui aurait été possible aussi. Cette classe est construite avec une asset. Son constructeur est protected car seulement la classe Board doit l'implémenter.

Enum Asset: Liste les différents assets possibles d'une case (Joueur, Cible, Caisse, Mur ou Vide). Il était nécessaire de faire cette classe pour faciliter la création de case et bien distinguer chaque assets.

d. Package File

Classe TextBoardBuilder: Cette classe permet la récupération des informations d'un plateau tel que sa taille et sa hauteur, mais elle contient aussi dans un ArrayList les lignes du plateau permettant alors de restituer ce dernier dans le programme.

Classe FileBoardBuilder: Cette classe étend BoardBuilder. Elle gère tout ce qui est relatif aux fichiers. En effet, c'est cette classe qui permet de lire ou créer un fichier, puis grâce à la classe TextBoardBuilder va restituer le plateau.

Classe BoardBuilder: Cette classe permet d'éviter la duplication de code entre les classes FileBoardBuilder et DatabaseBuilder notamment dans la restitution des informations pour créer le plateau. Ses méthodes sont protected car elles doivent être utilisées seulement pas les classes héritants de BoardBuilder.

e. Package Database

Classe Administrator: Classe qui gère l'interface administrateur/machine avec l'affichage du menu, le lancement des différentes méthodes, etc. Elle étend la classe Menu pour éviter une fois de plus la duplication de code. Elle appelle la classe Database pour gérer tout ce qui est rapport avec les bases de données.

Classe Database: Cette classe gère toutes les commandes SQL permettant à la classe Administrator d'afficher ce qui est voulu. Elle est construite avec une Connection qui permet de se connecter à la base de donnée. Chaque méthode à une référence précise qui correspond à un entier dans la classe Administrator. Son constructeur et ses méthodes sont toutes protected car cette classe doit seulement être utilisée par les classes du même package

Classe DatabaseBuilder: Cette classe étend BoardBuilder. Elle permet de gérer la création des différents plateaux contenus dans la base de donnée en utilisant la classe TextBoardBuilder.

f. Choix de collections

Dans la classe TextBoardBuilder j'utilise un ArrayList car j'ai besoin d'un tableau dynamique étant donné que je ne sais pas d'avance combien il y aura de lignes à insérer. Ce tableau doit bien évidemment accepter les doublons mais de plus l'ordre est important étant donné que quand le programme devra reformer le plateau il devra restituer ce dernier dans l'ordre.

Dans la classe Board, j'utilise deux ArrayList permettant d'enregistrer les positions X et Y de mes cibles. J'ai besoin de quelque chose de dynamique car je ne sais pas d'avance de combien sera composée de cibles le plateau. J'ai choisi d'utiliser une ArrayList car les entrées sont dans l'ordre donc je sais que la position X et ma position Y sont au même index ce qui fait qu'elles sont indissociables. Pour éviter les doublons, j'ai séparé ma méthode de pose en 2 méthodes distinctes. La première permet de changer la case en cible tandis que la deuxième permet d'enregistrer dans l'ArrayList. Ce choix à pour effet que quand un tableau est crée j'ajoute mes cibles

sur le plateau et dans l'ArrayList. Par contre, quand je suis dans une partie sur un plateau crée je fais seulement la pose du pion.

g. Choix divers

Pour la création de son propre plateau, j'ai laissé au joueur le libre droit, à ses risques et périls, de faire un plateau à plusieurs joueurs pour la simple est bonne raison qu'un niveau comportant plusieurs joueurs peut être très challengeant et intéressant si bien pensé. J'ai aussi décidé de faire un contrôle au niveau de la largeur rentrée mais pas au niveau de la hauteur, ce qui veut dire que l'utilisateur peut faire un plateau vide,

qui pour autant n'implémentera pas d'erreur et qui fera juste gagner le joueur dès

premier mouvement étant donné qu'il n'y a pas de cible. Enfin, si l'utilisateur ne rentre pas de joueur, un joueur pas défaut sera présent en 0,0 et n'implémentera pas d'erreur. J'ai donc laissé au bon vouloir du joueur de créer son propre plateau qui fonctionne, ou non mais qui ne relèvera pas d'erreur dans mon programme.

Je n'ai pas fait beaucoup de tests unitaires car j'ai testé de moi même à maintes reprises le jeu pour voir si d'éventuels bugs pouvaient apparaître. Comme il y a une ''interface graphique'' et que les méthodes que j'ai créées ne sont pas vraiment mathématiques et sont directement applicables. Je pense qu'essayer directement son programme est à peu de choses prêt tout aussi efficace, dans ce cas-ci, que des tests unitaires notamment dans le cas des builders et des bases de données.

Pour ce qui est des tests unitaires sur les mouvements, la fonction playerMvt était de base en private mais j'ai du la mettre en protected pour pouvoir faire des tests dessus.

Pour ce qui est de l'utilisation des bases de données, on a tout d'abord l'affichage du contenu de la base de donnée, puis une demande à l'utilisateur pour savoir ce qu'il veut prendre. Et en fonction de l'ID rentrée va prévisualiser le plateau qui va être construit, l'utilisateur doit alors valider oui ou non. Seulement, j'ai laissé à

l'utilisateur de rentrer un ID ne se trouvant pas dedans car ça affichera juste un plateau vide, ce qui n'est pas problématique mais juste inutile. C'est aussi pour cela

qu'il y a la confirmation et la prévisualisation pour bien montrer au joueur si c'est bien le bon plateau qui va être construit.

J’ai aussi utilisé une surcharge de la méthode showBoard car j’en avais besoin dans la classe DatabaseBuilder pour faire la prévisualisation. Je surcharge cette méthode pour l'adapter à cette nouvelle situation, cela permet aussi de ne pas alourdir la la méthode de base avec de nombreux paramètres génériques que je ne vais pas utiliser souvent.

En effet j’ai besoin que chaque row de la base de donnée soit stockée dans l’ArrayList de mon TextBoardBuilder pour reconstituer le plateau. Et pour cela je dois donc retourner un TextBoardBuilder d’où une nouvelle fois mon choix de faire une surcharge.