Rapport de Projet Java

« Perdu dans les corridors ! »

Cosialls Thomas / Berrada Salim – projet 203 – groupe TR3

Le 26/01/2012

Résumé

Ce document a pour but de présenter le travail effectué dans le cadre du projet d'algorithme et programmation utilisant le langage Java.

Seront principalement abordés les choix effectués pour mettre en œuvre l'application, les différents composants de l'architecture de l'application, les principaux algorithmes ainsi que les problèmes et améliorations rencontrés.

Table des matières

1 Introduction **2**

2 Architecture de l'application **3**

3 Principaux choix réalisés **3**
3.1 Classe Salle 3
3.2 Classe Pousse en abstract 3
3.3 Paramétrage mode aléatoire 3

4 Principaux algorithmes **4**
4.1 Développement des ronces 4
4.2 Méthodes possibilite et choisirPilier 5
4.3 Trouver le chemin 5

5 Conclusion **6**

1 Introduction :

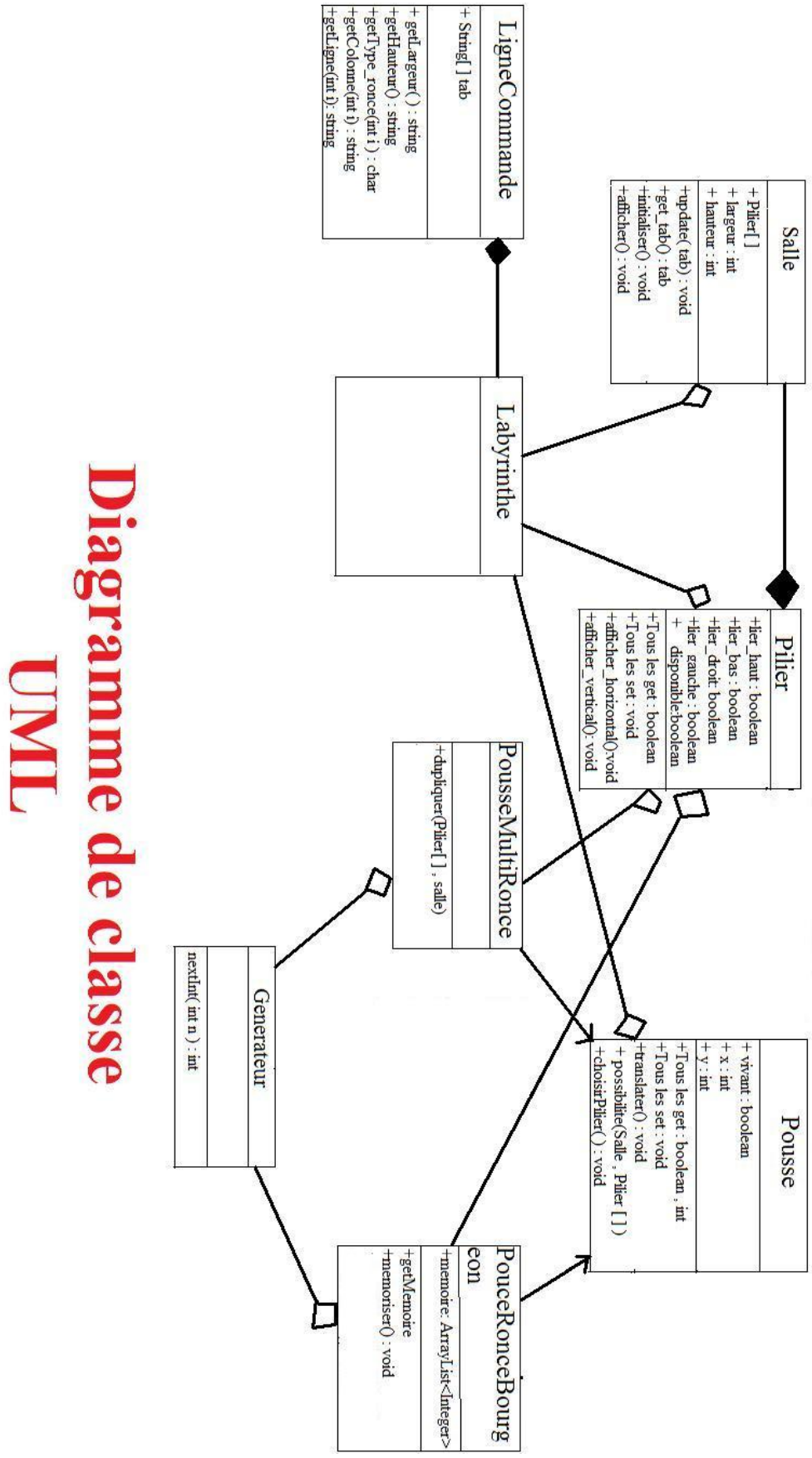
Le projet Java intitulé « Perdu dans les corridors ! » propose de créer une application ayant deux buts principaux : générer un labyrinthe et donner les instructions pour en sortir.

Pour générer le labyrinthe, on considère des graines que l'on plante initialement dans le labyrinthe et qui vont se propager aléatoirement à l'intérieur de celui-ci pour créer des murs vivants, transformant une simple salle vide en un véritable dédale ! Plusieurs contraintes et extensions doivent être prises en compte :

- On doit pouvoir planter les graines de deux manières : soit aléatoirement si l'utilisateur tape seulement `java Labyrinthe` en ligne de commande, soit c'est l'utilisateur qui choisit où planter les graines dans le labyrinthe, il doit alors le faire sous la forme :
  `java Labyrinthe largeur hauteur type_ronce@colonne-ligne`
  où type_ronce est un caractère, largeur, hauteur colonne et ligne des entiers.
- On doit pouvoir planter trois types de graine différents, donnant des ronces se dévellopant de manière différentes. Il y a tout d'abord la graine « normal » dont la pousse meurt dès lors qu'elle ne peut plus évoluer, la graine « multi-ronce » qui peut se dédoubler en 1,2,3 ou 4 pousses quand on l'arrose et enfin la graine à ronce bourgeonnante, qui essaie de se redévelopper le long de sa tige quand son pousse est mort.

Le labyrinthe crée, on doit ensuite indiquer le chemin à suivre pour sortir de ce dédale.
La méthode du sabot droit s'avère être un précieux fil d'Ariane…

2 Architecture de l'application :



3 Principaux choix réalisés :

3.1 Classe Salle :

Cette classe contient un **constructeur** contenant un tableau de deux dimensions de type Pilier ainsi que deux méthodes : **initialiser()** qui crée le labyrinthe sans les murs au départ et **afficher()** qui permet l'affichage graphique du labyrinthe en s'appuyant sur des méthodes de la classe Pilier.

Il aurait été tout à fait possible de traiter le sujet en ne créant pas le constructeur Salle et en faisant directement les opérations sur le tableau à deux dimensions de type Pilier. Cependant, la longueur des méthodes contenues dans la classe Salle le prouve, la classe Labyrinthe aurait été surchargée sans. Cela offre de la clarté et une facilite la programmation car il n'est plus nécessaire de jongler avec des expressions du type `tableau[i][j]` dans la classe Labyrinthe.

3.2 Classe Pousse en abstract :

Le choix a été fait de ne pas définir les méthodes **choisirPilier(Pilier[] pilier_possible, Salle salle)** et **possibilite(Salle salle, Pilier[] pilier_possible)** dans la super-classe Pousse mais seulement dans ses sous-classes (PousseRonce, PousseMultiRonce, PousseRonceBourgeon, Coccinelle). En effet, si pour les trois types de pousse, le code dans les deux méthodes est identique, il diffère dans la classe Coccinelle (qui sert pour trouver le chemin de la sortie). Dans cette classe au contraire, si un pilier n'est pas disponible, la coccinelle peut se déplacer vers lui.

3.3 Paramétrage du mode aléatoire :

Le mode aléatoire est choisi automatiquement si l'utilisateur n'entre rien en ligne de commande ie si le tableau `args[]` est vide. L'utilisateur est alors invité à saisir une hauteur et largeur correcte pour le labyrinthe.

Pour la plantation aléatoire des graines par l'ordinateur, une boucle for est utilisé et s'arrête quand le compteur a atteint la valeur hauteur*largeur/8. Pourquoi cette valeur ?
Le nombre hauteur*largeur correspond en fait au nombre de pilier du labyrinthe, c'est-à-dire au nombre maximal de graine que l'on peut planter au départ. Planter hauteur*largeur/8 graines nous parait être une valeur très convenable : il n'y a pas trop de graine plantée dès le départ, donc on peut voir chaque type de graine s'exprimer (on peut bien observer les duplications pour les multi-ronce, retour en arrière pour les bourgeonnantes), tout en obtenant un labyrinthe complexe au final.

4 Principaux algorithmes :

4.1 Développement des ronces :

Présentons ici l'algorithme le plus complexe du projet. Son utilité est simple : faire évoluer les ronces dans le labyrinthe jusqu'à ce qu'elles soient toutes mortes.

```
Tantque repneg != stock_pousse.length-1
      Tantque on est pas arrive au bout du tableau stock_pousse
            Si Pousse Vivante
                  Si Ronce normale …

                  Si Ronce Bourgeonnante …

                  Si Multi ronce …
                                    Fin Si
            FinSi

                     Si Pousse Morte
                                 Si Ronce bourgeonnante non ressucite…

                               Sinon…
                     FinSi


      Fin Tantque

Fin Tantque
```

Structure globale de l'algorithme

Pour faire évoluer les pousses, on utilise une boucle while qui parcoure le tableau stock_pousse (crée lors de la plantation des graines) jusqu'à sa taille maximale (égale aux nombres de piliers dans le labyrinthe moins un). Des cases seront nécessairement vide, sauf si on plante le maximum de graine dès le début.

On a ainsi utilisé un compteur repneg qui permet de sortir de la boucle lorsqu'il atteint la taille du tableau stock_pousse-1. Plus précisement, il s'incrémente quand une pousse morte est traité ou lorsqu'une case de stock_pousse est vide.

Quand la pousse est vivante, on distingue les trois types de graines pour avoir une évolution adaptée à la graine « que l'on est en train d'arroser ». Par contre, quand elle est morte, on doit juste vérifier que ce n'est pas une ronce bourgeonnante, auquel cas on essaie de la ressuscité le long de sa tige en faisant appel à la mémoire de la pousse bourgeonnante.

4.1 Méthode possibilite et choisirPilier :

Ces deux méthodes, définis dans les sous classe du type Pousse, permettent respectivement à la graine d'évaluer les possibilités d'évolution qui sont offertes autour d'elle et de choisir en fonction de l'étude précédente un pilier vers lequel se déplacer.

La **méthode possibilite** prend en paramètre l'objet salle contenant le tableau à deux dimensions de piliers ainsi qu'un tableau simple (non vide en principe, de taille 4). Elle consiste simplement à déterminer un par un si les piliers se trouvant autour de la ronce sont disponibles (utilisation de pilier.estDisponible()) et à engendrer un tableau simple de taille 4 qui stocke le pilier dans une case si celui-ci est disponible ; stocke la valeur `null` sinon.

La **méthode choisirPilier** prend en paramètre les mêmes paramètres que possibilite, le tableau simple en paramètre étant évidemment le tableau engendré par la méthode précédente. Si le tableau ne contient pas que des valeurs null (cas où au moins un piler est disponible), la ronce peut évoluer vers un pilier en modifiant ses coordonnées et en modifiant les liaisons du pilier vers lequel elle se déplace mais aussi celles de celui d'où elle provient.

4.1 Trouver le chemin :

Le code complet ne figure pas dans les sources rendues. Cependant, voilà les points de notre démarche pour parvenir à indiquer la sortie du labyrinthe :

On crée tout d'abord un objet Coccinelle, sous classe de Pousse.
On initialise ses coordonnées au pilier inférieur de la porte d'entrée gauche du labyrinthe.
Ensuite, on applique une **méthode choisir pilier spécifique à la Cocinelle** : elle doit choisir les piliers Qui ne sont pas disponibles. ➔ null est indiqué quand le pilier est disponible, dans le tableau renvoyé par la méthode possibilite.
Après, pour choisir le pilier, la cocinelle doit privilégier le pilier non disponible le plus proche d'elle en tournant dans le sens horaire (méthode du sabot droit).

- ⇨ Problème survient lorsque l'on arrive à des intersections.

5 Conclusion

La partie du projet s'intéressant à la génération du labyrinthe a été traitée entièrement, le labyrinthe est généré avec tous les types de graines, sous les deux modes, sans erreurs.
Cependant, des erreurs peuvent apparaitre si l'utilisateur ne saisit pas correctement les informations dans la ligne de commande : il resterait à **traiter les exceptions** (largeur négative, mauvaise structure pour planter une graine…). On pourrait aussi optimiser la plantation en interdisant tout doublet dans le tableau stock_pousse (utilisation d'un hashSet).

Concernant la partie devant indiquer la marche à suivre pour sortir du labyrinthe, elle n'a pu être achevée pour cause d'erreur au niveau des intersections qui nous ont fait perdre un temps précieux. De ce fait, aucune méthode choisirPilier n'ayant été redéfinie dans la classe Coccinelle, le fait de mettre cette méthode en abstract dans la classe Pousse parait inappropriée et engendre du code inutile.