AP3 - PROJET BATTLEROBOT

02 – Labyrinthe piégé

Objectifs à réaliser :

- Ajouter des salles piégées dans le labyrinthe
- Adapter les déplacements du Personnage dans ce nouvel environnement

- Mise en place des Salles piégées

Ajouts et modifications des classes :

Nous avons procédé à un certains nombre de changements dans l'organisation des classes. Pour mettre en place des salles piégées, nous avons créé un héritage de la classe « Salle » que nous avons appelé « SallePiege », cela permet à partir d'un transtypage de connaître parmi la collection de salles, lesquelles sont des salles piégées.

Évidemment cette héritage ne suffit pas, le Personnage doit prendre en compte cette nouveauté. Cette fois notre héros possède des vies, chaque passage dans une salle piégés lui fait perdre une vie, il continue son parcours dans le labyrinthe tant qu'il lui reste des vies. On a fait un héritage de la classe « PersonnageDefaut » pour obtenir la classe « PersonnageLife ». Cette classe possède un nouvel attribut « vie » qui est une classe « Life » gérant des points de vie ainsi qu'une nouvelle méthode « estVivant » pour savoir si le personnage possède encore des vies. Les collisions entre les pièges et le héro se fait dans la méthode « aller » : si la prochaine salle où il va est piégée (à l'aide d'un instanceof), alors on appel la méthode vie.perdreUneVie( ).

Il ne reste plus qu'à apporter des modification sur le labyrinthe : on a fait « LabyrinthePiege », un héritage de la classe « LabyrintheDefaut » et qui fait une surcharge de la méthode « creerLabyrinthe » pour enregistrer les salles piégés depuis le fichier « level_trapped.txt ». Du coup, on a dû changer l'écriture du fichier niveau de la manière suivante :

```
// level_trapped.txt

40 40          // définit la taille du labyrinthe
1 1            // coordonnée de l'entrée
38 38          // coordonnée de la sortie
3              // nombre de pièges dans le labyrinthe
1 15           // coordonnée du premier piège
20 13          // coordonnée du deuxième piège
14 30          // coordonnée du dernier piège
```

```
1 2                 // coordonnée de la premiere salle normale
… etc
```

outil supplémentaire :

Nous avons réalisé un petit script pour simplifier l'ajout des salles piégées à un « level.txt ». Il prend en paramètre deux arguments : le nombre de salle piégées voulue et le fichier level.txt à modifier, à la fin de l'exécution, il affiche une fenêtre pour confirmer la création du fichier « level_trapped.txt ». Ce script vérifie si les nouvelles salles piégées ajoutées ne dépassent pas les limites de la carte et si elles ne correspondent pas à la salle d'entrée ou à la salle de sortie. Si la nouvelle salle piégée est déjà déclarée dans le fichier level,txt, on supprime cette salle déjà existante pour la remplacer par la salle piégée voulue, et ainsi éviter une répétition, donc une erreur lors la création du labyrinthe.

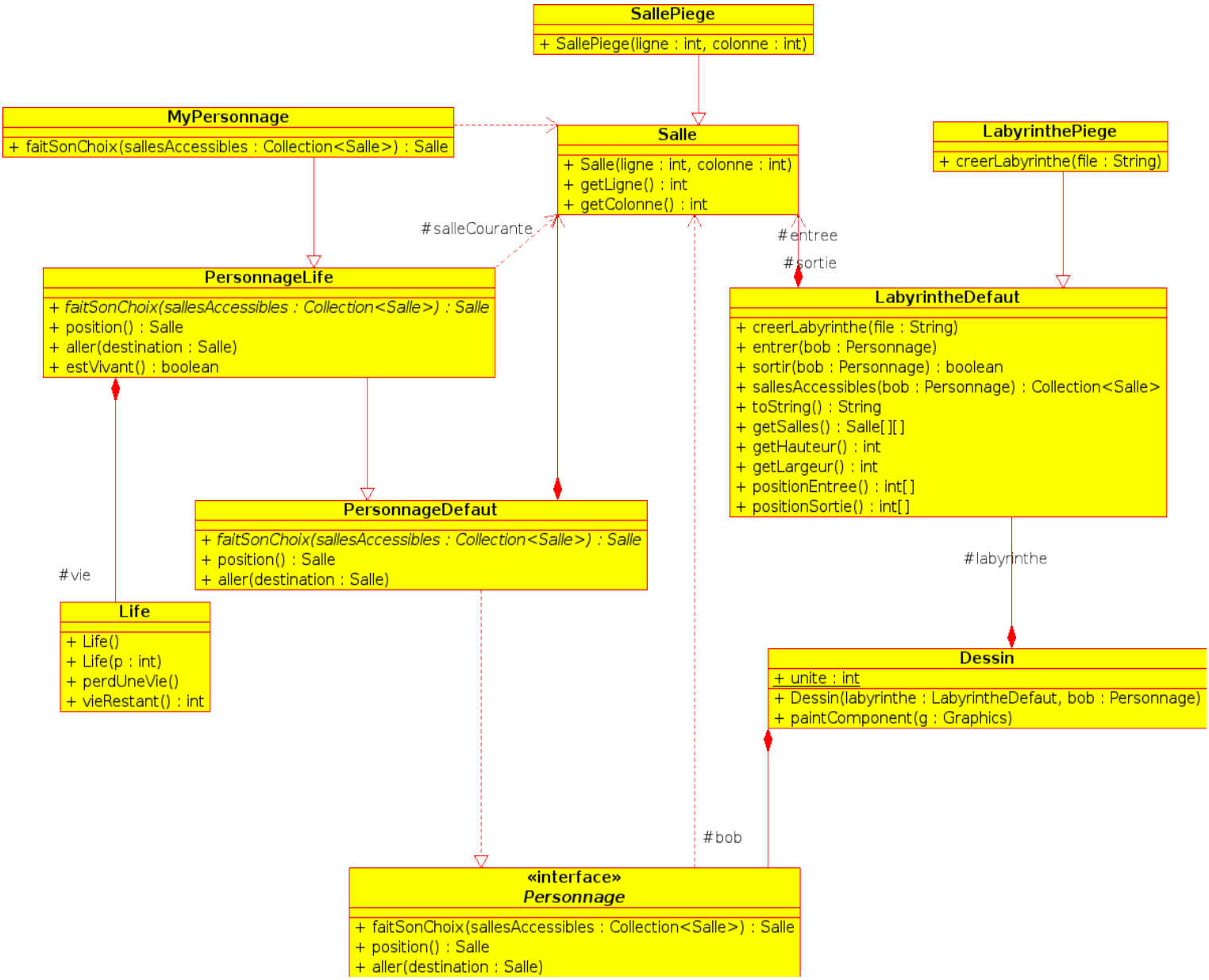
Pour l'utiliser, depuis la console placez vous dans le répertoire « outil », entrer la commande suivante : ./addtrap [nombre_piège] [chemin_du_level.txt]
Il ne vous reste plus qu'à suivre les instructions indiquées par le script.

mise à jour dans l'organisation des classes :

Nous avons gardé la même organisation du programme que la version précédente, nous avons juste ajouté quelques classes pour essayer de respecter l'esprit de développement d'un programme JAVA : effectuer des modifications sans changer le code source déjà présent à l'aide de surcharges, d'héritages et de polymorphisme.

Ci-dessous la nouvelle version du diagramme de classe (Voir Illustration 1: Diagramme de classe de la page 3)

Illustration 1: Diagramme de classe



- Adaptation de l'algorithme

Nouvelle organisation du héros :

Notre ancien algorithme dît avec « le marquage craie » n'était pas vraiment correct, d'une part nous n'avions pas traité l'exception qui faisait que notre héros ne sortait pas toujours du labyrinthe, et d'autre part la compréhension de l'algorithme était quelque que difficile au premier abord. Nous avons adopté une nouvelle démarche, en effet nous l'avons implémenté avec deux collections de salles :

« map » qui enregistre toute les salles traversée

« backUp » retient le chemin pour remonter aux prochaines salles non visitées.

Pour chaque passage dans une salle, notre personnage réalise les procédures suivantes :

- il regarde s'il est sur un bon chemin : il regarde si au moins une salle accessible n'est pas inscrit dans le « map ». Cet état est enregistré dans un booléen « wrongWay ».
- est-il sur un mauvais chemin, ou dans un cul-de-sac ? Dans ce cas il doit revenir sur ses pas, nous l'avons sauvegardé grâce au booléen « back ». Il enregistre la salle ou il se situe dans les listes « map » et « backUp ».
- lors qu'il effectue un retour, il suit le chemin contenu dans « backUp ». S'il termine son retour il continue normalement son parcours.
- S'il fait un parcours normal, il enregistre juste sa position dans le « map » et dans « backUp », en vérifiant si sa position n'est pas déjà marqué dans ces derniers.

Contournement de salles piégés :

Bob doit finir le labyrinthe en gardant toute ses vies, en effet il n'en possède seulement trois et ne connaît la position des pièges que si il met le pied dedans.

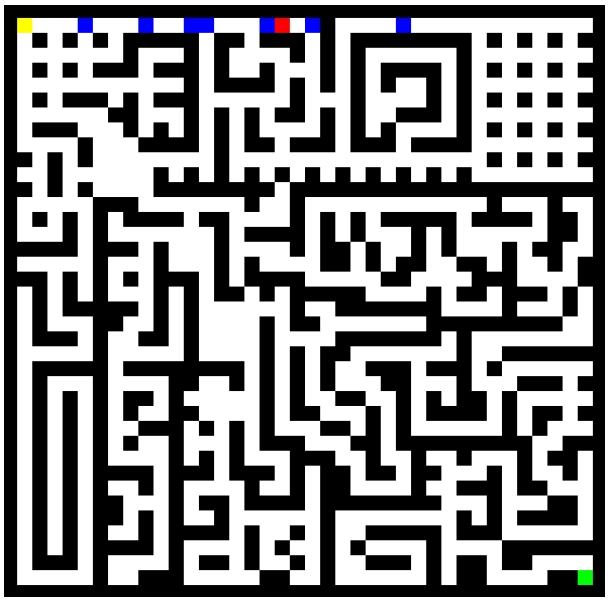
Cette Partie de la conception est celle qui nous a posé le plus de soucis, en effet tant sur le plan conceptuel que sur le plan technique la conception d'un algorithme permettant d'éviter les salle piégées déjà visitées lorsque c'était possible représente une quantité de travail et de reflexion tres importante.

De plus il faut souligner le fait que nous avons du changer totalement la logique de notre algorithme, cette tache a occuper la plus grande partie du temps qu'il nous était imparti afin ce réaliser notre projet.

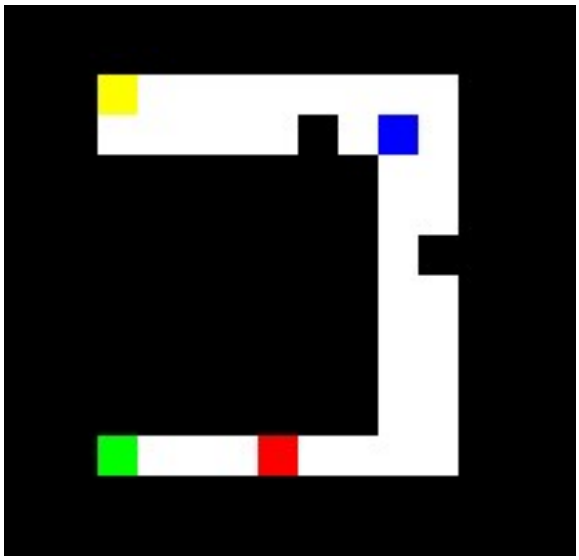
- Détails sur les niveaux piégés

On a réalisé 3 labyrinthes pour constater des différentes attitudes de notre nouvelle algorithme. Nous avons coloré toutes les salles piégées en bleu.

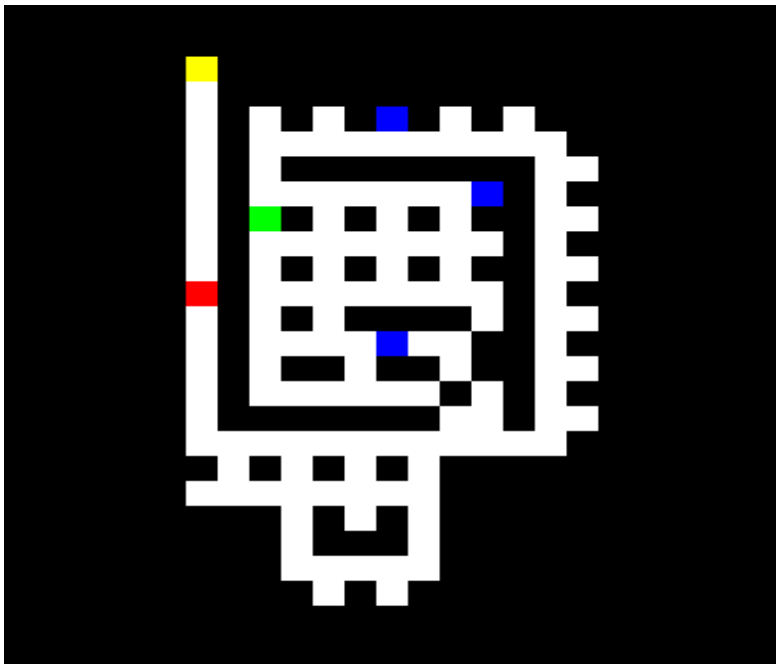
- Level1.txt : ce labyrinthe montre simplement le cas où notre héros ne peut plus continuer son exploration du labyrinthe faute de vie.



- level2.txt : Ce labyrinthe montre le défaut de notre algorithme, en effet on remarque que notre héros pourrait éviter de passer deux fois dans le piège.



- level3.txt : Ce labyrinthe montre que notre héros sort du labyrinthe vivant.



- Conclusion:

Malgré les problèmes rencontrés, au cours de notre projet, nous avons tout de même su concevoir un algorithme qui a défaut d'éviter intelligemment les salles piégées déjà visitées, va dans le pire des cas passer seulement 2 fois sur chacune d'elles.

Ce projet a été très formateur, notamment concernant la remise en question de nos décisions. Notre premier échec quand a faire sortir notre héros du n'importe quel labyrinthe nous a forcer a revoir notre perception du problème, a changer de point de vu et a a s'adapter a cette nouvelle vision du problème.