# Le labyrinthe, partie II.

POnGL, TP5, gagner en autonomie.

## Introduction

Dans ce TP, nous poursuivons le travail de la semaine dernière. Vous pouvez soit retravailler directement à partir de vos fichiers du TP précédent, soit repartir du corrigé fourni sur la page du cours, qui traite les sections 1 à 4 de la première partie :
`http://www.lri.fr/~blsk/POnGL/Laby1.java`
Comme aux séances précédentes, vous aurez parfois besoin d'aller consulter la documentation Java pour vous renseigner sur les méthodes de certaines classes :
`http://docs.oracle.com/javase/8/docs/api/index.html`
Aujourd'hui, vous ferez un pas de plus vers l'indépendance, puisqu'il vous faudra aussi aller chercher des informations sur des modèles de conception qui n'ont pas été vus en cours.

## 1 Ajouter des portes (repris de la semaine dernière)

On veut que le labyrinthe contienne maintenant des cases spéciales qui seront des portes, qui peuvent être soit ouvertes soit fermées, et que le héros peut ouvrir et fermer. Une porte ouverte s'affichera en vert et sera une case traversable, alors qu'une porte fermée s'affichera en rouge et ne sera pas traversable.

Le travail à effectuer est le suivant :

1. Déterminer la stratégie de conception à utiliser pour gérer de bonne façon l'état ouvert ou fermé de vos portes. Indice : il s'agit d'un modèle qui est présent dans les notes de cours !
2. Créer une classe `Door`, qui devra fournir deux méthodes de construction `openDoorFactory` et `closedDoorFactory` créant respectivement une porte ouverte et une porte fermée, et une méthode `changeState()` qui fait passer une porte de l'état ouvert à l'état fermé ou de l'état fermé à l'état ouvert.
3. Faire que les touches `y`, `n`, `j` et `g` déclenchent une tentative d'ouverture ou de fermeture de porte sur une case voisine du héros (respectivement au-dessus, en-dessous, à droite et à gauche). Comme aux parties précédentes, répartissez le travail de la bonne manière entre le modèle, la vue et le contrôleur.

Pour tester cette partie, décommentez les méthodes `putOpenDoor` et `putClosedDoor` dans la classe `LModel`, et la quatrième section de configuration dans la classe `Laby`.

## 2 Améliorer les portes

Si vous avez suivi à la lettre la recette du modèle de conception du cours, vous avez probablement obtenu un comportement qui n'est pas le plus satisfaisant. Par exemple, à chaque fois qu'une porte est ouverte, un nouvel objet représentant l'état d'une porte ouverte doit être créé par une instruction `new`. Pourtant, ce nouvel objet n'a pas d'attribut ni d'état qui le distingue véritablement du dernier état où la porte était déjà ouverte. Une porte a une seule manière d'être ouverte. Pour réutiliser toujours la même instance d'un objet représentant l'état ouvert d'une porte, on peur utiliser le modèle de conception "singleton", qui permet de construire une classe `A` telle qu'on ne construira pas plus que un objet de cette classe. Quand on a besoin de cet objet, plutôt que d'en créer un nouveau, on appelle une méthode qui nous retourne cet objet unique.

Le travail à effectuer est le suivant :

1. Chercher des informations sur le modèle de conception singleton. Vous pourrez entrer dans un moteur de recherche des mots-clés comme `java singleton pattern`.
2. Modifier votre implémentation des portes pour qu'elle utilise ce modèle.

## 3 Plusieurs vues

Soulignons deux raison pour lesquelles l'architecture MVC sépare le modèle (le labyrinthe) de la vue (l'affichage) :

— Une vue peut ne pas donner toutes les informations sur le modèle.

— Nous pouvons avoir simultanément plusieurs vues du même modèle.

Le but de cette section est de modifier le code du labyrinthe pour que la fenêtre graphique contienne deux vues différentes, qui joueront les rôles suivants :

— La vue principale affiche uniquement un carré de 5 cases de côté, centré sur le héros, qui montre l'état du labyrinthe (murs, portes, monstres...) à proximité du héros.

— Une deuxième vue, avec une échelle plus petite (par exemple `SCALE=10` au lieu de `SCALE=40`), montre l'ensemble du labyrinthe, mais n'affiche pas les créatures.

Le travail à effectuer est le suivant :

1. S'inspirer de la classe `LView` pour créer deux classes `LLocalView` et `LMiniMap`, dont la première n'affiche le labyrinthe que sur un carré de cinq cases de côté, et dont la deuxième n'affiche pas les créatures.
2. Modifier le reste du programme pour que l'affichage principal (la variable `frame` de la méthode `Laby:main`), au lieu d'un seul composant de classe `LView`, contienne maintenant deux composants de classes `LLocalView` et `LMiniMap`. Normalement, l'architecture MVC fait que vous n'avez pas à changer plus de deux lignes pour cela !

## 4 Plusieurs contrôleurs

On veut pour finir ajouter un bouton à notre interface graphique, que le joueur peut utiliser une fois dans la partie pour éliminer les monstres distants de moins de deux cases (c'est-à-dire les montres visibles dans la vue locale). Le bouton en question sera affiché et sera cliquable dans une troisième vue, et ses actions seront gérées par un deuxième contrôleur spécialement dédié (l'architecture MVC permet bien d'avoir plusieurs vues, mais aussi plusieurs modèles et plusieurs contrôleurs, chacun dédié à un aspect particulier).

Le travail à effectuer est le suivant :

1. Aller chercher des informations sur les boutons et leur utilisation en Java (on s'intéressera en particulier à la classe `JButton`).
2. Créer une nouvelle vue contenant ce bouton.
3. Créer une nouvelle classe `LButtonController` qui, lors d'un clic de notre nouveau bouton, appelle une méthode `LModel:killMonsters`. Il vous faudra au passage définir cette méthode du modèle.

## 5 Extensions

Quelques suggestions pour rendre le jeu plus complet/complexe :

— Faire que la mini-carte n'affiche que les parties du labyrinthe qui ont effectivement été découvertes par le héros, et laisse les autres en gris.

— Rendre le déplacement des monstres moins aléatoire, par exemple en les faisant se diriger de préférence dans la direction du héros.

— Plutôt que de multiplier les touches (e,c,f,s,d,y,j,n,g), faire en sorte qu'on ne se serve que de quatre touches de direction, et qu'un bouton permette de commuter entre deux utilisations : déplacement ou action.