Projet de reprise
Programmation Orientée Objet en langage Java
DUT Informatique
version du 1er septembre 2021

2021-2022

Table des matières

Informatique iut de BORDEAUX

université de BORDEAUX

Ce projet est consacré à la réalisation d'une application "labyrinthe", par ajouts de fonctionnalités mettant en œuvre progressivement différentes notions vues en première année. Il est structuré en six objectifs successifs (cf. table des matières), dont les quatre premiers ont été définis pour être à la portée de chacun dans les temps impartis. L'ordre de ces objectifs doit être strictement respecté.

La réalisation des différents objectifs est guidée par une série d'exercices, dont le niveau de difficulté (subjective) est indiqué par un score d'une à trois étoiles.

1 Modalités

1.1 Réalisation

- le projet est à réaliser seul ;
- les interfaces Java fournies dans le code de départ ne doivent pas être modifiées, ni enrichies (y compris par d'autres interfaces) ;
- 26 h dédiées de travail dans les emplois du temps, sur deux semaines, dont 14 h encadrées.

1.2 Évaluation

- simplicité du code et qualité de sa documentation
- test sous `NetBeans 12` de la machine `INFO-DEV` (VDI) / absence de message d'erreur et d'avertissement
  - ouverture du projet ;
  - compilation du projet (sans intervenir sur les paramètres) ;
  - exécution.
- investissement personnel dans la réalisation du projet
  - qualité et régularité des commits dans `gitlab`, avec prise en compte de la pertinence des messages associés ;
  - attitude pendant les séances encadrées : demande d'assistance sur le code pour ne pas rester bloqué, pour des explications de concept...
- note de type A,B,C,D,E prise en compte dans le contrôle continu du module M3105 (Conception et Programmation Objet)

1.3 Remise

- la remise sera faite sous la forme d'un dépôt `gitlab` de nom `2021-ProjetReprise-VotreNomPrenom` (pour lequel votre enseignant sera membre avec le statut `Reporter`), constitué d'un projet `NetBeans` fonctionnel (cf. sous-section Évaluation) comportant un dossier `livrables` (à la racine du projet) contenant
  - le projet `NetBeans` sous la forme d'une archive `zip` dans le fichier `VotreNomPrenom.zip` (créée à partir de `NetBeans` par exportation) ;
  - les trois diagrammes de classes (les deux diagrammes de l'exercice deux et le diagramme final de votre application) sous la forme d'un seul fichier `diagrammes.pdf` (une page par diagramme) au format `pdf` ;
  - un tableau de bord au format `markdown` dans le fichier `rapport.md` décrivant ce que vous avez réalisé pour chaque session de travail ainsi que les difficultés rencontrées.
- le fichier `README.md` de votre projet `gitlab` indiquera simplement la liste des objectifs atteints avec leurs dates.

2 NetBeans, JavaFX et INFO-DEV

Le code de départ fourni est déjà configuré : il suffit de l'importer dans `NetBeans 12` et à la première exécution ou demande de compilation, les dépendances seront téléchargées automatiquement.

Pour la réalisation d'autres projets de ce type, voici les instructions pour créer un projet utilisant JavaFX sur NetBeans 12, installé sur INFO-DEV :

— créer un projet de type `Java with Maven -> Simple JavaFX Maven Archetype`
— lors de la première exécution, les bibliothèques de JavaFX (version 13) seront téléchargées et configurées par le système de gestion des dépendances `maven`.

3 Objectif 1 : dessiner un labyrinthe

Le premier objectif est de dessiner le plan d'un labyrinthe (donné dans un fichier texte) dont les salles sont carrées. Le plan peut donc être dessiné dans une grille, sous la forme d'une case blanche pour une salle (par exemple). Le dessin des murs n'est pas requis pour cet objectif.

Le format utilisé pour stocker un labyrinthe dans un fichier texte est le suivant :

```
37 37 // nombre de lignes et de colonnes de la grille
1 1 // coordonnees de l'entree (dans la grille)
11 35 // coordonnees de la sortie
1 3 // coordonnees d'une autre salle
...
```

L'archive `depart.zip` contient le code de départ sous la forme d'un projet pour NetBeans. Le dossier `labys` contient des exemples de fichiers de labyrinthe.

Exercice 1 (⋆) Pour vous familiariser avec le format ci-dessus, ouvrez un fichier de labyrinthe et dessinez sur une feuille la partie du labyrinthe formée des salles dont les coordonnées sont inférieures ou égales à 7. Ce dessin artistique n'est pas à rendre.

Exercice 2 (⋆⋆) Avant de passer à la suite, prenez le temps de prendre connaissance des interfaces fournies en annexe ainsi que de l'extrait de code de la boucle principale : ces interfaces et cette boucle ne devront pas être modifiées, ni étendues dans le cadre de votre réalisation. Faites le diagramme de classes correspondant (à rendre). Réalisez maintenant le diagramme de classes (à rendre également) induit par le code initial fourni, sans la partie "tests".

Exercice 3 (⋆⋆) Créer une classe *Salle* implémentant l'interface *ISalle*, sachant qu'une salle possède des coordonnées x et y. Deux salles sont considérées comme identiques si elles ont les mêmes coordonnées (spoiler : c'est important pour les méthodes *equals* et *hashcode*, bien sûr). À ce stade, considérez qu'une salle n'est jamais adjacente à une autre.

Exercice 4 (⋆⋆⋆) Compléter la classe *Labyrinthe*, la méthode *chemin* se contentant de retourner l'objet *null* pour le moment : il suffit donc d'écrire le contenu de la méthode *creerLabyrinthe*. Indication : la classe *outils.Fichier* (fournie) contient le nécessaire pour lire les nombres enregistrés dans un fichier texte, un par un.

Exercice 5 (⋆⋆) Modifier la classe *Dessin* pour qu'elle puisse dessiner le labyrinthe (en dessinant au minimum les salles). Puis compléter la classe *Vue* composée d'un dessin et d'un labyrinthe (le dossier *icons* fournit des images que vous êtes libres d'utiliser pour ce faire), et implémentant l'interface *IVue*. Au lancement du programme, le plan du labyrinthe s'affiche maintenant : félicitations !

4 Objectif 2 : tests (tests unitaires sur les formats de fichiers)

Ce second objectif consiste à gérer les problèmes posés par des fichiers de labyrinthe non conformes. Le but est de réaliser des tests unitaires pour pouvoir garantir facilement que les fichiers de labyrinthes de l'application sont valides.

L'ensemble des tests unitaires devront être réalisés dans un paquet `tests` dédié.

Exercice 6 (⋆⋆) Écrire une méthode auxiliaire *boolean testCoordonneesSallesFichier(File f)* qui teste si les salles du fichier *f* ont toutes des coordonnées valides i.e. abcisse (resp. ordonnée) comprise entre 0 et la largeur-1 (resp. hauteur-1) du labyrinthe. Compléter ensuite le test unitaire *testCoordonneesSalles*, testant chacun des fichiers contenus dans le répertoire *labys/*.

Exercice 7 (⋆⋆) Compléter de manière similaire le test unitaire *testPasDeDoublon* qui teste, pour chacun des fichiers contenus dans le répertoire *labys/*, que chacune des salles n'est présente qu'une fois (i.e. il n'y a pas deux salles avec les mêmes coordonnées).

Exception **ExceptionInvalidFile** (labyrinthe de secours)

On désire maintenant protéger l'application contre un fichier de labyrinthe invalide (ce qui permettra par exemple de laisser un utilisateur concevoir ses propres labyrinthes et les utiliser dans l'application).

Le principe est, lorsqu'un fichier est détecté comme non valide, de passer l'application dans un mode de secours, consistant à charger un fichier de labyrinthe donné, à la place du fichier fautif.

Nous allons procéder en deux étapes.

Exercice 8 (⋆) Écrire une méthode statique *boolean testValide(String nomFichier)* dans la classe *Fichier* qui effectue les tests décrits dans la partie précédente sur les tests unitaires, sur un nom de fichier donné.

Exercice 9 (⋆⋆) Dans le cas d'une tentative de chargement d'un fichier invalide (via la méthode *creerLabyrinthe*), levez une exception *ExceptionInvalidFile*. De quelle classe doit hériter cette exception ?

Faites en sorte que le traitement de cette exception dans le programme principal crée le labyrinthe à partir du fichier *labys/level7.txt*.

Si le fichier de secours est lui-même invalide, traitez l'exception en mettant fin au programme.

5 Objectif 3 : déplacer des personnages

Il est temps d'insérer un (gentil) héros dans notre labyrinthe. Cependant, si l'application est capable de dessiner le plan du labyrinthe, on ne peut pas utiliser celui-ci pour se déplacer car la structure du labyrinthe n'est (pour l'instant) pas gérée.

Exercice 10 (⋆⋆) Modifier la classe *Salle* : une salle peut indiquer si elle est adjacente à une autre. Les salles accessibles à partir d'une position donnée sont donc les salles adjacentes à cette position. Adapter la classe *Labyrinthe* conséquemment.

Exercice 11 (⋆) Écrire une classe abstraite *APersonnage* implémentant l'interface *IPersonnage* en procurant une implémentation à toutes les méthodes à l'exception de la méthode *faitSonChoix*.

Exercice 12 (⋆) Écrire la classe *Heros* héritant de *APersonnage* : celle-ci a un attribut *salleChoisie* qui est retourné par la méthode *faitSonChoix* si celle-ci fait bien partie des salles accessibles (sinon cette méthode retourne la position actuelle). L'attribut *salleChoisie* sera mis à jour par le gestionnaire graphique (lors de l'exercice 14) en fonction des actions du joueur sur la fenêtre graphique : quelle doit être la visibilité de cet attribut ?

Il faut maintenant dessiner le héros et réagir aux actions sur le clavier.

Exercice 13 (⋆⋆⋆) Écrire une classe abstraite *ASprite*, composée d'un *IPersonnage*, qui propose une implémentation de toutes les méthodes de l'interface *ISprite* : toutes les méthodes doivent donc être concrètes.
Cette classe est abstraite car l'image à dessiner dépend du type de personnage (héros, monstre…) et l'initialisation de l'image sera déléguée à des classes filles.
Indication : la méthode *dessiner* doit dessiner une image positionnée à un certain endroit dans la fenêtre graphique en fonction de la localisation (salle) du personnage pris en charge : doter la classe *ASPrite* d'attributs pour stocker les coordonnées graphiques du sprite ainsi que l'image est donc raisonnable.
Munir cette classe d'un constructeur paramétré avec un paramètre de type *IPersonnage* : la classe

étant abstraite, on ne pourra pas créer directement des objets de la classe *ASprite* même si celle-ci a un constructeur et uniquement des méthodes concrètes. Ce constructeur sera appelé (cf. exercice suivant notamment) par les classes filles.

Comme le constructeur paramétré de `ASprite` prend un paramètre de type `IPersonnage`, on peut créer un personnage graphique (Sprite) à partir de n'importe quelle implémentation de `IPersonnage` : l'exercice suivant concerne la cas de l'implémentation `Heros`.

Indication : pour initialiser un objet `Image` à partir du fichier `icons/link/LinkRunShieldL1.gif` (fourni dans le projet), par exemple, on peut faire simplement

```
Image image = new Image("file:icons/link/LinkRunShieldL1.gif");
```

Exercice 14 (⋆⋆⋆) Écrire une classe *HerosSprite* héritant de *ASprite* pour le personnage géré par un joueur : dans son constructeur, pensez à appeler le constructeur de *ASprite* avec l'instruction *super*. Le constructeur doit notamment charger l'image correspondant au héros.
La gestion des déplacements du personnage sera faite avec les touches fléchées : pour ce faire, déclarer que la classe *HerosSprite* implémente *EventHandler<KeyEvent>* et donner une implémentation de la méthode *handle(KeyEvent event)* adaptée, sachant que *event.getCode()* contiendra le code de la touche appuyée (*LEFT* pour la flèche gauche etc.). Naturellement, il faut mettre à jour l'attribut *salleChoisie* du héros en fonction de la touche appuyée.

Il reste quelques actions à effectuer dans le code afin de voir notre héros dans le labyrinthe et pouvoir le déplacer.

Exercice 15 (⋆) Tout d'abord, initialiser un héros dans l'application, en dé-commentant les lignes aimablement fournies dans la méthode *initSprites* de la classe *Core*.

Compléter ensuite la méthode *dessiner* de la classe *vue2D.javaFX.Vue* pour qu'elle dessine aussi ses sprites (rappel : *Vue* implémente *IVue* qui est une collection de *ISprite*).
Enfin, prévenir la scène (fenêtre graphique) qu'il y a un écouteur sur le clavier : le bon moment est de le faire lorsqu'on ajoute le sprite du héros (l'objet *HeroSprite*) à la vue. Pour cela redéfinir la méthode *add* de la classe *Vue* ainsi :

```
@Override
public boolean add(ISprite sprite){
        super.add(sprite);
        // si le sprite est controle par le clavier
        if (sprite instanceof EventHandler){
                System.out.println("registering keylistener");
                // association de l'ecouteur sur le clavier avec le
                   composant graphique principal
                this.scene.setOnKeyPressed((EventHandler) sprite);
        }
        return true;
}
```

Une nouvelle étape a été franchie : le labyrinthe se dessine, le héros aussi et on peut même le déplacer !

6 Objectif 4 : monstres (polymorphisme sur les personnages)

Notre héros peut désormais parcourir moult labyrinthes, mais il se sent un peu seul. Nous allons maintenant rajouter des monstres qui se déplaceront de manière aléatoire, i.e. pour se déplacer leur méthode `faitSonChoix` renverra une salle au hasard parmi les salles accessibles. Notre application contiendra donc une collection de personnages, l'un d'entre eux étant le héros, les autres étant des monstres. C'est une occasion rêvée d'exploiter le polymorphisme... Un monstre ne pourra pas sortir d'un labyrinthe.

Exercice 16 (⋆⋆) Créer une classe *Monstre* qui se déplace de manière aléatoire et la classe graphique *MonstreSprite* associée.

Exercice 17 (⋆) Créer une bonne dizaine de monstres et les placer à la sortie du labyrinthe.

Il faut prendre garde à une difficulté algorithmique : un processus séparé (provenant du moteur de rendu graphique) prend en charge l'affichage de la vue. Or il est nécessaire de parcourir la collection des personnages pour tous les redessiner. Si on modifie simultanément (dans la boucle principale) cette collection en enlevant des personnages (par exemple s'ils ont perdu), alors cela pose un problème d'accès concurrent. Pour gérer ce problème, il faut prendre un type de collection qui est protégé contre les accès concurrents, comme `CopyOnWriteArrayList`.

Exercice 18 (⋆) Modifier la vue pour qu'elle hérite de *CopyOnWriteArrayList* au lieu de *ArrayList*.

7 Objectif 5 : améliorations simples (animations, éclairage localisé)

Le héros se déplace dans le labyrinthe mais son déplacement n'est pas fluide d'un point de vue graphique.

Exercice 19 (⋆⋆⋆) Modifier la gestion du déplacement pour que le déplacement soit fluide. Pour cela, il faut découpler le déplacement dans le labyrinthe (salle par salle) du déplacement graphique, pour que ce dernier se fasse pixel par pixel. Indication : avant que le héros ne puisse à nouveau se déplacer dans le labyrinthe, il faut que le déplacement graphique soit achevé.

Exercice 20 (⋆⋆)
Mettre en place (dans la classe *Dessin*) un éclairage autour du héros : l'affichage des salles sera de plus en plus sombre en fonction de leur distance (euclidienne ou Manhattan, au choix) au héros.

8 Objectif 6 : améliorations plus complexes basées sur les plus courts chemins

L'implémentation du labyrinthe est assez rudimentaire et utilise comme structure de données une collection de salles. En utilisant un graphe comme structure de données, il est possible de faire facilement des améliorations plus "sophistiquées".
L'objectif est de tirer partie de l'abstraction offerte par les interfaces pour inclure du code développé par des tiers, offrant des fonctionnalités avancées, que nous n'aurions pas pu développer dans un temps raisonnable et/ou avec un niveau de performance satisfaisant.

8.1 Affichage du plus court chemin dans le labyrinthe, via une bibliothèque externe (JGraphT)

Nous allons utiliser une bibliothèque dédiée à la modélisation des graphes et à leurs algorithmes : `JGraphT`. Le site officiel de cette bibliothèque est `http://jgrapht.org/`.

Exercice 21 (⋆⋆) Consultez le site de JGraphT. En particulier, analysez le code de démonstration pour un graphe dirigé : https ://github.com/jgrapht/jgrapht/wiki/Demos-DirectedGraphDemo. Puis, parcourez la *Javadoc* disponible à *https://jgrapht.org/javadoc/* pour vous faire une idée des classes et des méthodes disponibles. Quelles sont les méthodes dédiées au calcul de plus courts chemins ?

Exercice 22 (⋆) Suivre les instructions officielles pour ajouter la bibliothèque *JGraphT* à votre projet, en utilisant la résolution de dépendances de *Maven* : voir *https://github.com/jgrapht/jgrapht/wiki/Users:-How-to-use-JGraphT-as-a-dependency-in-your-projects*.

Maintenant que nous utilisons la bibliothèque `JGraphT`, nous pouvons exploiter des fonctionnalités plus avancées de celle-ci. Ainsi, on peut calculer un plus court chemin entre deux sommets `i` et `c` avec

```
DijkstraShortestPath dsp = new  DijkstraShortestPath(g, i, c);
List path = Graphs.getPathVertexList(dsp.getPath());
```

Exercice 23 (⋆⋆⋆) Implémentez maintenant l'interface *ILabyrinthe* dans la classe *LabyrintheGraphe* composée d'un *DirectedGraph* ou d'un *SimpleGraph* (au choix). Vous pouvez naturellement étendre la classe *Labyrinthe*. La construction du graphe peut être effectuée dans la méthode *creerLabyrinthe*.

Testez le bon fonctionnement de cette nouvelle version, en mettant dans `Application.Core` :

```
labyrinthe = new labyrinthe.LabyrintheGraphe();
```

Exercice 24 (⋆⋆) Dans la méthode *creerLabyrinthe*, calculer un plus court chemin entre l'entrée et la sortie, et le dessiner.

Impressionnant, non ? Soyons plus fou encore... Notre héros est particulièrement maladroit : il n'est pas capable de suivre le plus court chemin... Pour l'aider, nous allons lui indiquer en permanence un plus court chemin en jaune entre sa position et la sortie. Pour cela, la vue a besoin de pouvoir connaître un plus court chemin entre la position du héros et la sortie : c'est le rôle de la méthode `public Collection<ISalle> chemin(ISalle u, ISalle v);` dans l'interface `ILabyrinthe`.

Exercice 25 (⋆⋆⋆) Écrire une méthode *dessinPlusCourtChemin(ISprite heros)* dans la classe *Dessin* qui dessine un plus court chemin entre la position du héros et la sortie du labyrinthe.

Un problème de performances potentiel est que la classe *Dessin* redessine très fréquemment le labyrinthe (il faut 20 images par seconde pour qu'une animation soit fluide...). Par conséquent, il est important de limiter les calculs dans les méthodes de dessin. Or dans l'exercice précédent, on demande un plus court chemin systématiquement.

Pour éviter des calculs inutiles lorsque le héros ne bouge pas, modifier l'implémentation de la méthode *chemin* pour qu'elle ne recalcule un plus court chemin vers la sortie que lorsque le héros a bougé.

Exercice 26 (⋆⋆) Ajouter un terrible dragon qui poursuit par un chemin le plus court le héros. Naturellement, le chemin doit être recalculé en fonction des déplacements du héros.

Exercice 27 (⋆⋆) Compléter le test unitaire *testChemin* qui teste, pour chacun des fichiers contenus dans le répertoire *labys/*, qu'il existe bien un chemin entre l'entrée et la sortie.

8.2 Éclairage

On voudrait maintenant que l'éclairage ne traverse plus les murs : pour cela, nous allons nous baser sur la distance entre deux salles dans le graphe.

Exercice 28 (⋆⋆) Écrire une méthode *int distanceGraphe(Salle s, Salle t)* qui retourne la distance entre deux sommets dans le graphe (cette valeur peut être pré-calculée lors de la création du graphe), puis adapter la méthode de dessin des salles en fonction.

8.3 Affichage des murs

Exercice 29 (⋆⋆) Ajouter l'affichage des murs : un mur n'est visible que lorsqu'il est contigu à une salle visitée. Afficher également les salles visitées.

9 Annexes

Code des interfaces (ne pas modifier)

```
package labyrinthe;
public interface ISalle {
    public int getX(); // abcisse
    public int getY(); // ordonnee
    public boolean estAdjacente( ISalle autre);
}

package personnages;
public interface IPersonnage {
    public ISalle faitSonChoix(Collection<ISalle> sallesAccessibles); //
       renvoie une salle parmi sallesAccesibles
    public ISalle getPosition();  // renvoie sa position courante
    public void setPosition( ISalle s); // definit sa position courante
}

package labyrinthe;
public interface ILabyrinthe extends Collection<ISalle>{
    public void creerLabyrinthe(String file) throws IOException; // cree
       le labyrinthe a partir d'un fichier
    public Collection<ISalle> sallesAccessibles(IPersonnage heros); //
       renvoie les salles accessibles pour le heros
    public ISalle getEntree(); // accesseur sur l'entree
    public ISalle getSortie(); // accesseur sur la sortie
    public Collection<ISalle> chemin(ISalle u, ISalle v); // un plus
       court chemin entre u et v
    // dimensions de la grille
    public int getLargeur();
    public int getHauteur();
}

package vue2D;
public interface IVue extends Collection<ISprite>{
    public void dessiner(); // dessiner l'ensemble des elements
       graphiques
}

package vue2D.sprites;
public interface ISprite extends IPersonnage{
        public void dessiner(GraphicsContext g);
        public void setCoordonnees(int xpix, int ypix);
}
```

Code de la boucle principale (méthode **jeu**) (ne pas modifier)

```
package application;
public class Core {
    ISprite heros;
    ILabyrinthe labyrinthe;

    protected void initLabyrinthe() {
```

```
        // creation du labyrinthe
        labyrinthe = new labyrinthe.Labyrinthe();
        chargementLaby("labys/level3.txt");
    }

    protected void initSprites(IVue vue) {
        // creation du heros
        //IPersonnage h = new personnages.Heros(labyrinthe.getEntree());
        //this.heros = new HerosSprite(h, labyrinthe);
        //vue.add(this.heros);
    }

    protected void jeu(IVue vue) {
        // boucle principale
        ISalle destination = null;
        while (!labyrinthe.getSortie().equals(heros.getPosition())) {
            // choix et deplacement
            for (IPersonnage p : vue) {
                Collection<ISalle> sallesAccessibles = labyrinthe.
                    sallesAccessibles(p);
                destination = p.faitSonChoix(sallesAccessibles); // on
                    demande au personnage de faire son choix de salle
                p.setPosition(destination); // deplacement
            }
            // detection des collisions
            boolean collision = false;
            ISprite monstre = null;
            for (ISprite p : vue) {
                if (p != heros) {
                    if (p.getPosition() == heros.getPosition()) {
                        System.out.println("Collision !!");
                        collision = true;
                        monstre = p;
                    }
                }
            }
            if (collision) {
                vue.remove(monstre);
                vue.remove(heros);
                System.out.println("Perdu !");
                System.out.println("Plus que " + vue.size() + "
                    personnages ...");
            }

            temporisation(50);
        }
        System.out.println("Bravo!");
    }

    private void chargementLaby(String fic) {
         try {
                labyrinthe.creerLabyrinthe(fic);
        } catch (IOException ex) {
```

```
                ex.printStackTrace();
            }
        }

        protected void temporisation(int nb) {
            try {
                Thread.sleep(nb); // pause de nb millisecondes
            } catch (InterruptedException ie) {
            };
        }
    }
```