Version : 17 mars 2025

IUT Charlemagne - Dept info
Qualité de Développement

2024-2025
BUT 1A

TD 4 - Sokoban

1 Description du sujet

`Sokoban` (cf figure 1) est un jeu de puzzle publié en 1982 au Japon dont les règles sont simples mais la solution complexe [1].

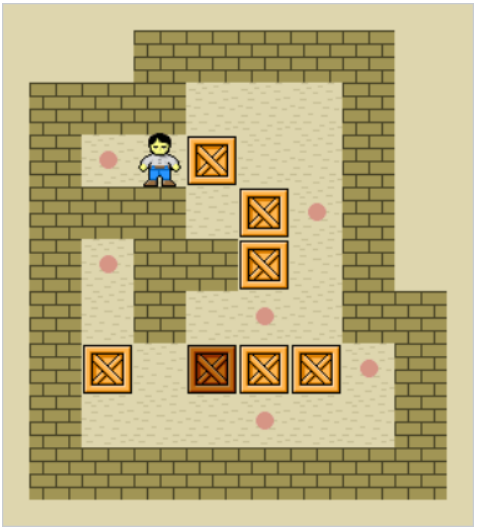

FIGURE 1 – Sokoban (Source Wikipedia)

Dans ce jeu, on contrôle le gardien d'un entrepôt devant pousser des caisses pour amener chaque caisse sur un lieu de dépôt différent. La difficulté repose sur le fait qu'on ne peut pousser une caisse que si la case derrière elle est vide (cf figures 2, 3 [2]).

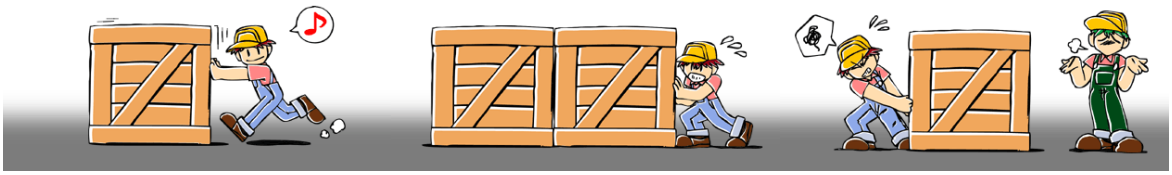

FIGURE 2 – Règles du jeu : (a) le gardien peut pousser des caisses isolées ; (b) il ne peut pas pousser une caisse si une autre caisse se trouve derrière ; et (c) il ne peut pas tirer les caisses.

1. `http://www.game-sokoban.com/`
2. Sources : `https://www.sokoban.jp/rule.html`

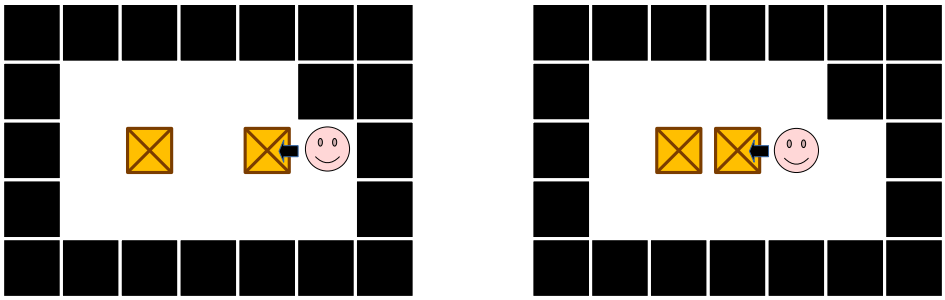

FIGURE 3 – Pousser des caisses dans Sokoban. La première action (à gauche) pousse la caisse. Par contre la caisse ne peut ensuite plus être poussée dans la même direction car une autre caisse la bloque (à droite).

Ce TP a pour objectif de mettre en place un jeu de type `Sokoban` qui servira de base à la SAÉ *S2.01 - Développement d'une application JAVA*.

Ce projet est structuré en plusieurs parties :

- la section 3 présente les données de votre application et la structure des classes attendues ;
- la section 4 présente les traitements attendus de votre application ;
- la section 5 présente les rendus attendus et des éléments complémentaires (tests, interface graphique, ...)

2 Organisation du travail

Consigne

Le rendu constituant une base pour la suite, une vérification de plagiat sera effectuée -comme pour tout rendu- et aucun plagiat ne sera toléré.

Consigne

Le travail est à faire en binome. Un seul membre du binôme devra déposer sur arche. Vous ferez attention à bien préciser les membres du binômes dans votre dépot (rapport et fichier texte `README`).

Pour fonctionner, votre projet `IntellJ` devra être structuré de la manière suivante (cf. figure 4) :

- un répertoire `src` contenant :
  - un package `graphisme` avec les classes fournies sur arche ;
  - un package `jeu` avec les classes à écrire ;
- un répertoire `out` contenant les .class générés par `IntellJ` ;
- un répertoire `laby` contenant les fichiers labyrinthe ;
- un répertoire `test` contenant les test `JUnit` à écrire.

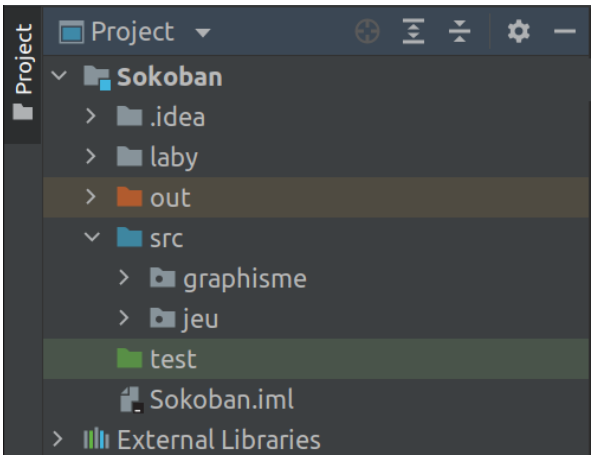

FIGURE 4 – Structure du projet `IntellJ`.

3 Données et structure de l'application

L'application proposée sera structurée autour des différentes classes suivantes décrites de manière plus précise dans la suite du sujet :

- une classe `Element` décrivant les objets du jeu ;
- des sous-classes `Perso`, `Caisse` et `Depot` qui héritent de `Element` et qui décrivent les éléments de jeu ;
  - `Perso` représente le gardien que le joueur peut déplacer ;
  - `Caisse` représente les caisses qu'il est possible de pousser ;
  - `Depot` représente les lieux de dépôt des caisses (pour que le jeu soit un succés, il faut que toutes les caisses soient déplacées sur des depôts) ;
- une classe `ListeElements` destinée à contenir une liste de plusieurs éléments en jeu (on utilisera deux objets `ListeElements`, un pour stocker la liste des depôts et un pour stocker la liste des caisses - cf classe `Jeu` en section 3.6) ;
- une classe `Labyrinthe` représentant le labyrinthe et décrivant les murs ;
- une classe `Jeu` représentant le jeu et décrivant le labyrinthe, le personnage et l'ensemble des caisses et des lieux de dépôt.

Ces classes peuvent être représentées sous la forme du **diagramme de classes** de la figure 5 :

- Les flèches avec un triangle (par exemple entre `Caisse` et `Element`) représentent une relation d'héritage.
- les flèches finissant par une pointe (par exemple entre `Jeu` et `Labyrinthe`) représentent une relation d'association, c'est-à-dire, le fait de possèder un objet en attribut (ici `Jeu` possède un objet de type `Labyrinthe` en attribut).

3.1 Système de coordonnées

Pour avoir les mêmes comportements sur tous les projets, on considérera que la première dimension (habituellement nommée `x`) correspond au numéro de colonne et que la

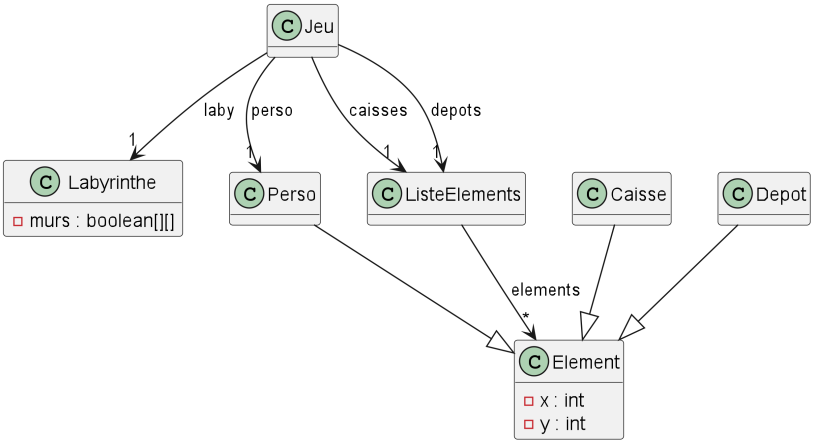

FIGURE 5 – Diagramme de classes décrivant les relations entre toutes les classes du projet.

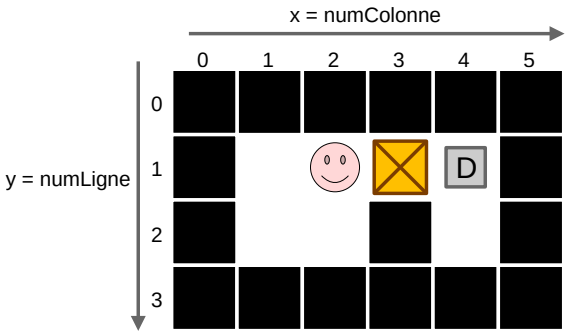

FIGURE 6 – Système de coordonnées utilisé. Dans tous les cas (tableau de murs, coordonnées, déplacements du personnage), la première dimension est x (le numéro de colonne) et la seconde y (le numéro de ligne). Dans cet exemple, le personnage se trouve en coordonnées `(2,1)`, la caisse en `(3,1)`, le lieu de dépôt en `(4,1)` et le mur intérieur en `(3,2)`.

seconde dimension (habituellement nommée `y`) correspond au numéro de ligne. La manière dont les coordonnées sont représentées est illustré par la figure 6.

Vous ferez particulièrement **attention aux axes x et y et à leurs orientations** (l'axe y va du haut vers le bas) que ce soit pour le chargement des fichiers, les déplacements du personnage ou l'affichage textuel.

3.2 Classe `Element`

La classe `Element` représente un élément de jeu. Un `Element` est caractérisé par deux attributs représentant la position de l'élément dans le labyrinthe :

- un attribut `x` de type `int` ;
- un attribut `y` de type `int`.

3.3 Classes `Perso`, `Depot` et `Caisse`

Les classes `Perso`, `Depot` et `Caisse` sont simplement des sous-classes de la classe `Element`. Elles ne possèdent pas d'attributs ni de méthodes spécifiques.

Comme indiqué auparavant,

- la classe `Perso` représente le gardien ;
- la classe `Caisse` permet de représenter les caisses en jeu ;
- la classe `Depot` premet de représenter les lieux où déposer les caisses.

Concernant les lieux de dépôt et les caisses, une caisse posée sur un lieu de dépôt reste en jeu. Elle constitue toujours un obstacle. Il est toujours possible de la déplacer tant qu'on respecte les règles du jeu. Le jeu s'arrête lorsque toutes les caisses sont positionnées sur des lieux de dépôt.

3.4 Classe `ListeElements`

La classe `ListeElements` contient plusieurs éléments qui seront soit des `Caisse` soit des `Depot`. De manière précise, la classe `ListeElements` possède comme seul attribut une `ArrayList` d'objets de type `Element`.

On utilisera deux objets `ListeElements` dans `Jeu` (cf descriptif de `Jeu`) : la liste des caisses présentes en jeu et la liste des lieux de dépôt.

3.5 Classe `Labyrinthe`

La classe `Labyrinthe` décrit le labyrinthe dans lequel le personnage va évoluer.

Cette classe ne contient qu'un attribut nommé `murs` de type `boolean[][]`. Pour un `x` et un `y` donnés, `murs[x][y]` vaut `true` si et seulement si la case `(x,y)` est un mur.

3.6 Classe `Jeu`

La classe `Jeu` contient toutes les informations de la partie en cours.

Conformément au diagramme de classes, la classe `Jeu` possède 4 attributs :

- un attribut `perso` de type `Perso` qui représente le personnage contrôlé par le joueur et sa position ;
- un attribut `caisses` de type `ListeElements` qui contient tous les objets `Caisse` du jeu ;
- un attribut `depots` de type `ListeElements` qui contient tous les objets `Depot` du jeu ;
- un attribut `laby` de type `Labyrinthe` qui décrit les murs de l'environnement.

4 Traitements attendus

4.1 Constantes utilisées

4.1.1 Constantes Caractères

Pour représenter le labyrinthe à l'affichage et à la lecture, on utilisera les **constantes** suivantes de type `char` à déclarer dans la classe `Labyrinthe`

- `'#'` représente un mur et sera nommée `MUR` ;
- `'$'` représente une caisse et sera nommée `CAISSE` ;
- `'@'` représente le personnage et sera nommée `PJ` ;
- `'.'` représente un lieu de dépôt et sera nommée `DEPOT` ;
- `' '` représente une case vide et sera nommée `VIDE`.

Ces caractères correspondent au standard des fichiers `.xsb` qui sont le standard de représentation d'un niveau Sokoban [3].

4.1.2 Constantes Actions

De la meme manière, on va représenter les actions du joueur par les chaînes **constantes** suivantes à déclarer dans la classe `Jeu` :

- `"Haut"` représenté par la constante nommée `HAUT` ;
- `"Bas"` représenté par la constante nommée `BAS` ;
- `"Gauche"` représenté par la constante nommée `GAUCHE` ;
- `"Droite"` représenté par la constante nommée `DROITE`.

L'intérêt d'utiliser des constantes est (1) d'éviter les fautes de frappe dans votre code et (2) de pouvoir facilement changer les valeurs des constantes sans impacter le comportement de votre application.

4.2 Chargement de labyrinthe

Les fichiers décrivant un labyrinthe sont des fichiers textes structurés conformément à la figure 7. Chaque ligne du fichier contient le descriptif de chaque ligne du labyrinthe caractère par caractère. Chacun de ces caractères correspond à une des constantes déclarées préalablement (cf section 4.1.1) :

- `'#'` pour un mur ;
- `'$'` pour une caisse ;
- `'.'` pour un lieu de dépôt ;
- `'@'` pour la position initiale du personnage ;
- `' '` pour une case vide.

On notera que la taille du fichier n'est pas indiquée initialement. Pour simplifier les opérations et ne pas lire deux fois le fichier, on vous propose de faire l'opération en plusieurs étapes :

3. `https://fr.wikipedia.org/wiki/Sokoban` et `http://grigr.narod.ru/` pour charger des niveaux.

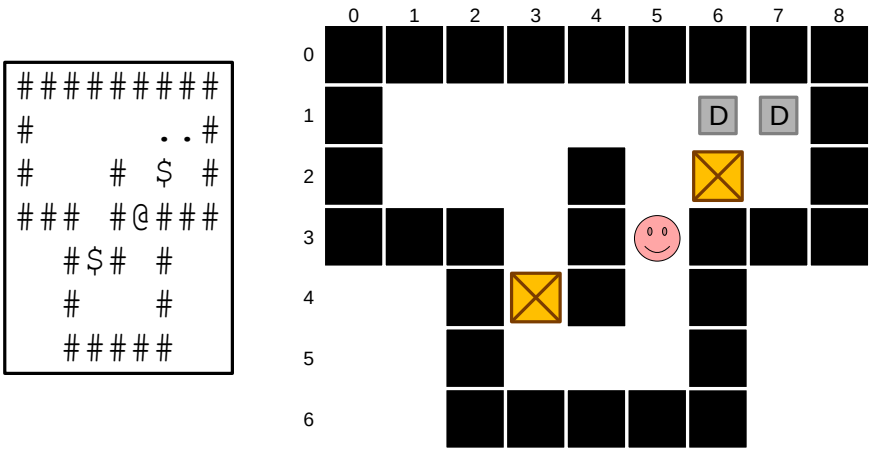

FIGURE 7 – Représentation graphique du fichier `"laby1.txt"` fourni sur arche. Le labyrinthe a 7 lignes et 9 colonnes. La position initiale du gardien (caractère `'@'`) est `(5,3)`.

1. lecture du fichier stocké dans une liste de `String` ;
2. déduction des dimensions du labyrinthe pour construire les tableaux utiles (attention toutes les lignes n'ont pas la même dimension) ;
3. construction du `Jeu` en fonction des informations stockées.

En plus des exceptions de type `IOException`, plusieurs types d'erreurs peuvent apparaître à la lecture du fichier. Pour chacun de ces cas, on lèvera une exception de type `FichierIncorrectException` (à définir) avec le message correspondant :

- un caractère est inconnu dans le descriptif du labyrinthe (message : `"caractere inconnu <X>"` avec `<X>` le caractère correspondant) ;
- il manque le personnage (message : `"personnage inconnu"`) ;
- il n'y a aucune caisse dans le jeu (message : `"caisses inconnues"`) ;
- le nombre de lieux de dépôt ne correspond pas au nombre de caisses (message : `"Caisses(<C>) Depots(<D>)"` où `<C>` désigne le nombre de caisses et `<D>` le nombre de dépôts).

La classe `Chargement` (à écrire) devra possèder une méthode `static Jeu chargerJeu (String nomFichier)` en charge d'effectuer le chargement du jeu stocké dans le fichier nommé `nomFichier` et de retourner un objet `Jeu` complet. Cette méthode devra faire remonter les exceptions que vous jugerez utiles. Vous pouvez écrire les méthodes et les constructeurs que vous jugez utile pour écrire la classe `Chargement`.

Question 4.1

Écrire la classe `Chargement` et la méthode `chargerJeu`.

4.3 Sortie textuelle

La classe `Jeu` doit disposer d'une méthode `String jeuToString()` générant le descriptif de l'état du jeu conformément à la manière dont les caractères sont représentés. Vous pourrez utiliser une méthode `char getChar(int x,int y)` retournant le caractère correspondant à la case `(x,y)` et utilisant les constantes proposées.

Consigne

Attention, les tests unitaires enseignant et l'interface graphique proposée reposeront sur cette méthode. Cette méthode doit être **correctement codée** sous peine de ne pas pouvoir tester votre application ou lancer l'interface graphique.

Question 4.2

Écrire la méthode `jeuToString` dans la classe `Jeu`.

4.4 Déplacement du personnage

La classe `Jeu` doit possèder une méthode `void deplacerPerso(String action)` en charge de déplacer le personnage dans la direction choisie par l'utilisateur tout en respectant les règles du jeu. Les directions possibles seront les constantes `HAUT`, `BAS`, `GAUCHE`, `DROITE` définies dans la classe `Jeu`.

Afin de simplifier l'écriture du code et de **ne pas faire de copier-coller**, la méthode `deplacerPerso` pourra utiliser différentes méthodes à définir :

- une méthode `static int[] getSuivant(int x, int y, String action)` dans la classe `Jeu`. Cette méthode permet de calculer les coordonnées de la case suivante en fonction de l'action. Cela permet de séparer le calcul des cases suivantes des tests de déplacements ;
- une méthode `boolean etreMur(int x,int y)` dans la classe `Labyrinthe` qui permet de savoir si un mur se trouve sur la case `(x,y)` ;
- une méthode `Element getElement(int x,int y)` dans la classe `ListeElements` qui retourne l'élément situé à la case `(x,y)` s'il y en a un ou `null` sinon. Cela permet facilement de tester la présence d'une caisse ou d'un lieu de dépôt et de récupérer l'objet correspondant.

Une exception de type `ActionInconnueException` (à définir) devra être lancée lorsque la chaîne passée en paramètre de la méthode ne correspond pas à une des actions proposées.

Question 4.3

Ecrire la méthode `deplacerPerso` dans la classe `Jeu`.

4.5 Fin du jeu

La méthode `boolean etreFini()` de la classe `Jeu` a pour objectif de vérifier si le jeu est fini, c'est-à-dire si le joueur a réussi à pousser les caisses sur les lieux de stockage

(dépôts).

Cela se caractérise par le fait que toutes les caisses se trouvent sur des lieux de dépôt (ou inversement que les lieux de dépôt contiennent tous une caisse).

Question 4.4
Écrire la méthode `etreFini` dans la classe `Jeu`.

4.6 Classe `MainJeu`

Enfin, en utilisant vos méthodes et les constantes, écrire une classe `MainJeu` prenant au lancement le nom d'un fichier et lançant l'exécution du jeu en mode texte.

À chaque pas de temps, la méthode `main` doit afficher le labyrinthe, demander au joueur son action en affichant les actions possibles, déplacer le personnage et afficher le nouveau statut du jeu. Le jeu s'arrêtera lorsque la condition de fin sera atteinte.

Le nombre de déplacements effectués pour atteindre la victoire sera alors affiché. Il représente le score du joueur (l'objectif étant de résoudre le problème en un minimum de déplacements). Si vous le souhaitez, il est possible d'afficher le nombre de déplacements effectués au fur et à mesure de l'avancement du jeu.

Enfin, votre application doit ête robuste et capable de s'adapter aux différentes situations exceptionnelles qui pourraient se produire à l'exécution.

Question 4.5
Écrire la classe `MainJeu`.

4.7 Tests unitaires

Question 4.6
En utilisant `IntelliJ`, ajouter l'ensemble des `getter` pour toutes les classes (cela permettra de lancer facilement les tests).

Votre projet devra proposer des tests unitaires. Ces tests devront permettre :

- de vérifier que le chargement de fichier fonctionne comme décrit dans le sujet (dans la classe de test `JUnit` `TestChargement`) ;
- de vérifier que le déplacement de personnage fonctionne conformément aux règles (dans la classe de test `JUnit` `TestJeu`).

Question 4.7
Développer en `JUnit` 5, les tests unitaires correspondants.

5 Rendu

5.1 Forme des méthodes

Afin de pouvoir tester votre code, vous devrez impérativement respecter les noms des exceptions et des méthodes suivantes :

- exception `ActionInconnueException` ;
- exception `FichierIncorrectException` ;
- méthode `void deplacerPerso(String action)` dans la classe `Jeu` ;
- méthode `String JeuToString()` dans la classe `Jeu` ;
- méthode `boolean etreFini()` dans la classe `Jeu`.

Chacune de ces méthodes devra respecter les contraintes de l'énoncé. Vous respecterez aussi les constantes proposées.

5.2 Interface Graphique

Une interface graphique est fournie pour tester vos classes. Cette interface graphique se trouve dans le package `graphisme` et est à copier dans votre projet `IntelliJ`.

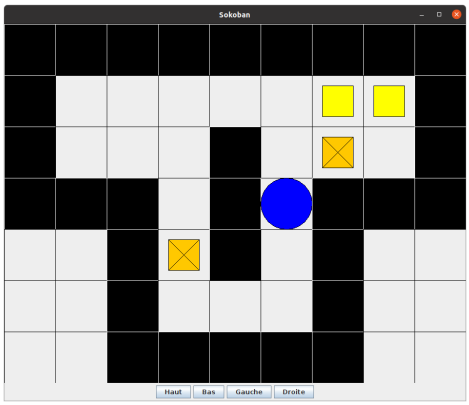

FIGURE 8 – Rendu graphique de l'application fournie.

Le lancement d'un jeu se fait simplement en écrivant une classe `Main` appelant le constructeur `Application(Jeu jeu)` de la classe `Application`. Les classes graphiques reposent uniquement sur les méthodes demandées `deplacerPerso`, `JeutoString` et `etreFini`. Il est nécessaire que vos méthodes fonctionnent correctement pour que l'interface graphique puisse s'exécuter.

L'interface graphique obtenue aura alors la forme présentée figure 8 (à partir du fichier fourni en exemple).

Le personnage se contrôle alors en cliquant sur les différents boutons de l'application ou en utilisant les touches du clavier `Z`, `Q`, `S` et `D` une fois l'application activée.

5.3 Compte-rendu

Votre projet devra être accompagné d'un compte-rendu en pdf (environ une page) précisant les méthodes que vous avez écrites et les difficultés rencontrées au cours de votre développement (quel problème, quelle solution, ...).

5.4 Rendu final

Votre rendu final se fera sur arche et devra inclure :

- l'ensemble de vos sources (tests compris) ;
- vos fichiers compilés (dans le sous-repertoire `out`) ;
- vos fichiers labyrinthe de test (permettant de reproduire vos tests ou de lancer l'application) ;
- votre compte-rendu d'une page.

Si tous les documents y sont présents, vous pouvez simplement compresser votre projet `IntelliJ` et le déposer sur arche.