***Projet d’informatique :***

***Baba Is You***

******

**(Arnaud De Lena)**

**Romain Eloy**

**Julien Ladeuze**

**2022**

**Bachelier année 1, sciences de l’informatiques**

**Tables des matières**

***1.Introduction (page 3)***

-présentation du jeu (page 3)

-répartition des tâches (page 3)

***2.Structucture du projet (page 4)***

**2.1.Model (page 4)**

-fonctionnalités de bases (page 4)

- principe OO\* utilisé (page 5)

- structure de données utilisées

- complexité globale

**2.2.View**

- système de fenêtre utilisé

- ???

**2.3.Presenter**

- liens effectués entre le Model et le Présenter

**2.4.Ressources**

Hiérarchie des fichiers dans ressources

***3.Ajouts apportés***

***4.Difficultés rencontrées***

***5.Points forts et Points faibles***

***6.Erreurs connues***

***7.Mini guide***

***8.Conclusion***

***9.Bibliographie***

***10.Remerciements***

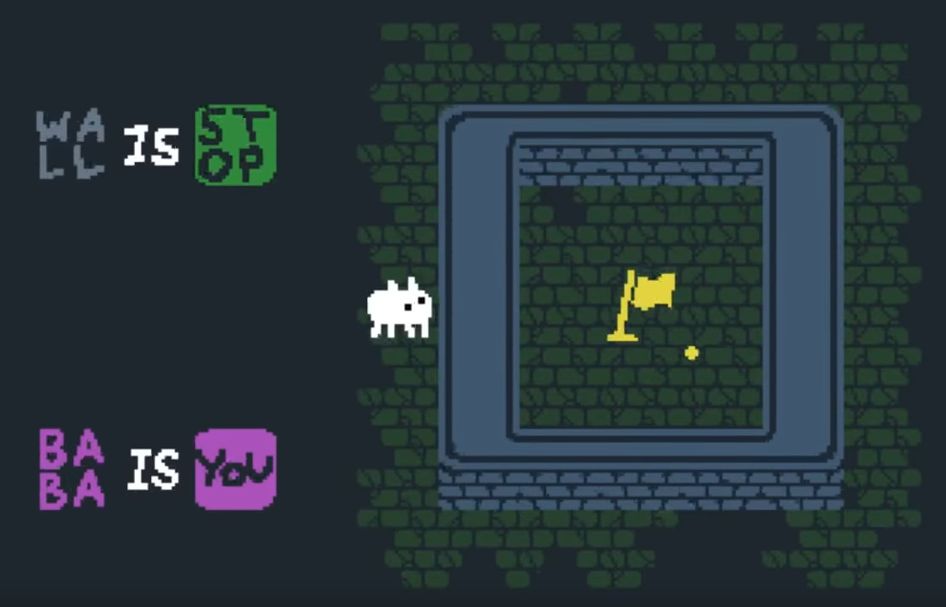
\*Orienté Objet

*1.INTRODUCTION*

Présentation du jeu

Dans le cadre de notre cours « projet d’informatique », il nous est demandé de reproduire un jeu vidéo. Ce jeu a été fait pour la première fois durant le Nordic Games Jam (Un endroit où les développeurs peuvent exprimer leur talent sur des jeux vidéo) de 2017 où il a d’ailleurs remporté le prix. Ce jeu est « Baba Is You ». Un jeu vidéo de style « Sokoban\*» puzzle/casse-tête. La particularité de ce jeu est que les règles se trouve elle-même dans le niveau et peuvent être déplacés afin de réussir le niveau. Ceci est l’essence même du jeu.

A noter que des fonctionnalités de bases devaient être implémentés : chargement d’une carte à partir d’un fichier texte, système de sauvegarde, implémentation des 4 premiers niveaux de la version jam\*



Répartition des tâches

-Romain : partie interface graphique

-Julien : partie logique + rédaction

-Arnaud : absent (dernier commit sur github : 01/03/2022)

\*https://fr.wikipedia.org/wiki/Sokoban

\*https://hempuli.itch.io/baba-is-you

*2.Structure du projet*

Lors du cadre de notre projet, nous avons eu un cours concernant l’architecture MVP

(Model View Presenter). Nous avons décidé d’adopter cette architecture car elle convenait bien pour l’implémentation de l’application. Les parties du MVP seront présentés dans les points suivants.

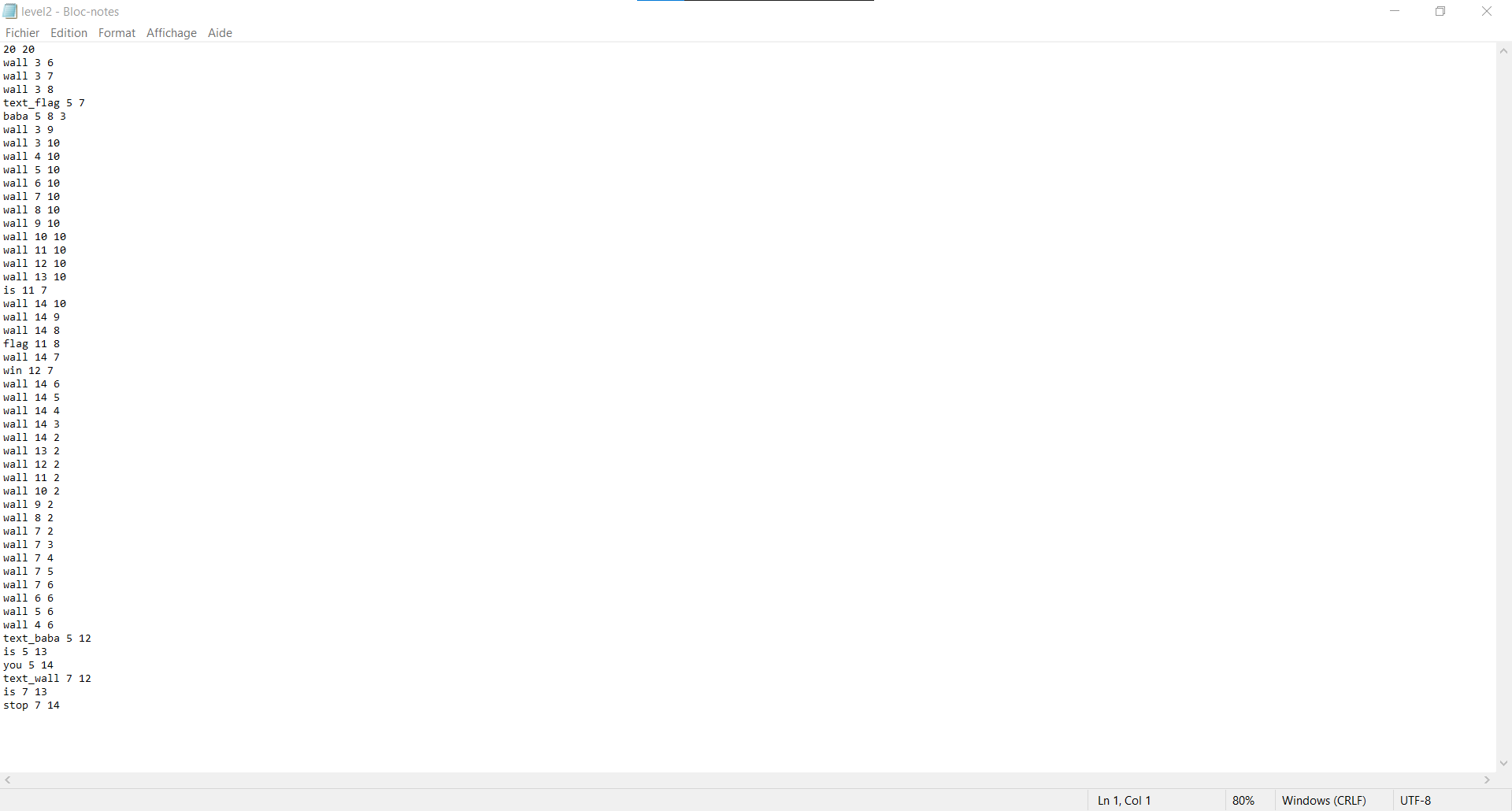
2.1.Model

La partie Model correspond à la « machinerie » de l’application. C’est ici que le travail de données va se jouer.

Fonctionnalités de bases

Niveau = fichier texte :

Comme dit dans l’introduction, nous avons dû implémenter les niveaux comme des fichiers texte d’un format bien précis voir ci-dessous.



Ce fichier représente le 2eme niveau du jeu

La première ligne représente la taille de la carte. Dans ce cas-ci, 20 en longueur et 20 en largeur. Pour le reste, on y trouve des éléments du jeu comme baba. A côté de leur nom, on y trouve les coordonnés associées en x et y. La quatrième valeur possible est un chiffre compris entre 0 et 3 représentant la direction initiale de l’objet (dans notre cas, on ne l’a pas utilisé car on y a pas trouvé d’utilité).

Pour implémenter tout ça dans notre jeu, nous avons eu recours à un tableau à double dimensions de type String. Nous avons fait ce choix car le développement de cette partie s’est faite aux alentours du cours de programmations et algorithmiques 2 sur les tableaux.

La structure du tableau se fait de cette manière Un sous-tableau représente une ligne du fichier. Celui-ci est de taille 4 pour accueillir le maximum d’information et toujours dans le même ordre (sauf la premier sous tableaux <-> 1 ere ligne du fichier): indice 0 : élément du jeu, indice 1 : coordonné en x, indice 2 : coordonné en y (et indice 3 : direction initiale).

L’initialisation se fait dans la classe « Extract » qui est indépendante des autres classes. Après que le tableau aura été initialisé, on l’appliquera dans la classe « Map » (à voir par la suite).

L’emplacement des ces fichiers se trouvent dans le répertoire « default»(resources/level/default).

Sauvegarde = l’inverse de la mise en place d’un niveau

Dans le cas d’une sauvegarde, on va devoir enregistrer la carte du jeu à partir du niveau en cours ( modifié ou non). Étant donné que notre carte est faite à partir d’un tableau ( voir plus bas), on va tout simplement prendre le nom de l’objet dans le tableau et ses coordonnées (x,y). On va placer tout ça dans un fichier texte de la même manière que le fichier représenté au dessus.

Toute ces sauvegardes se situent dans le dossier « save » (resources/level/save) dans lequel on peut retrouver un fichier « history ». Ce fichier contient le nom des sauvegardes et la date à laquelle elles ont été prises . Ceci permettra de récupérer la dernière sauvegarde en priorité car elle se trouve à la fin du fichier(pour reprendre une partie).

Les 4 niveaux obligatoires

Les 4 premiers niveaux se situant dans le dossier « default » sont ceux imposés par l’énoncé.

Principes OO utilisés

Voici une liste exhaustive des concepts orientés objets utilisés :Classe, classe interne ,encapsulation, héritage, interface, polymorphisme et l’énumération.

Où ?

-Classe : partout

-Classe interne : dans la classe BlockRules

-encapsulation : partout où il y a des attributs

-héritage : de la classe Environment aux classes enfantes de Item

-interface : les classes qui l’implémente sont Item et BlockRules

-polymorphisme : Environment

- énumération : dans la classe Rules

Pourquoi ?

Le concept de classe est l’essence même du langage Java. En plus de ça, l’orienté objet est un paradigme plus simple pour faire un jeu que le procédural.

Pour rappel, une classe interne est une classe se trouvant dans une autre classe. Ici, la notion de classe interne a été utilisé pour une meilleure gestion de classe et de lisibilité au niveau du code. Comme tous les objets de BlockRules sont tous les mêmes (sauf l’apparence), aucune raison de les mettre dans des fichiers différents.

L’encapsulation des données est un moyen de protéger son programme des manipulations de l’utilisateur. Dans notre cas, l’utilisateur de ne pourra interagir avec le jeu que par les entrées au clavier qu’il met. Il ne pourra en aucun cas, modifier l’apparence, la position, l’élément d’un jeu sans avoir fait une commande spécifique au préalable.

L’héritage est un moyen de factoriser du code et de rendre de plus en plus spécifique les classes via l’ajout de méthode, d’attribut, ect… Ceci pour nous faire gagner un temps considérable dans la réalisation du projet.

Une interface est une sorte de caractéristique attribuée à différents objets. Plus précisément, une interface est un contrat régi par un ensemble de méthode à implémenter dans les classes qui utilisent cette interface. Ceci nous a permit de cataloguer nos classes et d’utiliser le concept de polymorphisme.

Le polymorphisme peut rendre plus accessible certaines classes. Dans notre cas, le polymorphisme a été utilisé pour représenter tous les objets dans une même catégorie (voir plus bas).

L’énumération est un moyen de représenter quelque chose sans lui attribuer aucune caractéristique et méthode. Cela nous a permis de manipuler nos règles de la manière la plus simple possible.

Comment ?(interface,heritage,polymorphisme,enumeration)

La structure du model se présente comme ceci :

On retrouve en amont une interface (Entity) représentant tous les objets disponibles dans le jeu. Cette interface comprend en majorité des méthodes booléennes qui font office de caractéristique pour les objets les implémentant.

A côté d’Entity, on retrouve Environment. Une classe qui crée la carte du jeu. Cette carte est un tableau de type Entity. On y retrouvera à l’intérieur des classes implémentant cette interface. C’est comme ça que le concept de polymorphisme est utilisé.

Ensuite, on retrouve deux grandes classes : Item et BlockRules. Comme son nom l’indique, BlockRules est une classe représentant tous les blocs de règles du jeu. Quant à Item, c’est tout le reste. Autrement dit, tous les objets représentés par les règles.

Dans BlockRules, on y retrouve des classes internes. Ces classes internes descendent toutes de la classe dans laquelle elles sont contenues. Alors pourquoi avoir fait des classes internes ? Car dans ce cas-ci, comme les règles ont les mêmes caractéristiques cad elles peuvent être poussés, elles ne permettent pas la victoire (directement), ect… Elles sont quasiment toutes les mêmes à une différence près : leur apparence. Cela permet donc d’avoir moins de fichier .java inutile.

Du coté de Item, on va retrouver dans cette classe trois choses importantes : l’implémentation des méthodes issus de l’interface, la gestion d’une carte de jeu temporaire et la méthode « move(String input) ».

Les méthodes d’Entity

Ces méthodes vont discuter avec la classe « BigAlgorithm ». Cette classe s’occupe de la gestion des règles du jeu. Ces méthodes ont pratiquement toutes le même moyen d’implémentation. Elles vont vérifier la condition de l’objet en fonction des permissions issues de « BigAlgorithm .

La carte temporaire

La carte temporaire est comme son nom l’indique une carte qui ne dure qu’un temps. On va y trouver des éléments qui ont toutes des règles permettant à certain objets de passer au-dessus d’elles. Cette carte permet dont des les stocker en attendant qu’il n’y ait plus rien à leur emplacement sur la carte principale.

La méthode : move(String input)

Cette méthode est le point central du jeu. Cette méthode va implémenter TOUTES les autres méthodes d’Item. À l’aide d’un input (entrée représentant un mouvement), la méthode va effectuer tous les changements nécessaires à la carte principale(en plus d’établir et d’actualiser la carte temporaire).

Au plus bas de la structure, on retrouve les classes enfants d’Item comme Baba, Wall , ect… Ces classes ont toutes le même moyen d’implémentation à deux exceptions près :

Leur apparence et la règle qui représente.

À l’extérieur de ces classes, on retrouve la classe Rules est de type « Enum ». Elle permet de représenter de les règles du jeu.

Voici la structure du Model :

Extract,

Rules,

Action

Environment

Entity

BlockRules :

Sink,TextGoop,Is,ect

BigAlgorithm

Item

ect

Wall

Flag

Baba

« Is-a »

classe

classe

La classe BigAlgorithm sert à créer un tableau à deux dimensions de type Enum contentant les règles. Il est présenté de cette manière : {{Rules.BABA, Rules.YOU}}.

Structure de données utilisées

Les structures de donnés utilisés sont : les tableaux, les Arraylists(tableaux dynamiques) et les dictionnaires.

Les tableaux sont utilisés dans de nombreux cas mais on peut en retenir un qui est très important : la carte du jeu. Alors pourquoi l’utiliser pour ça ? Tout simplement que la taille de notre carte est fixe et d’un type bien particulier. En plus ce ça, ce tableau est à double dimension. Les éléments qui composent le tableau de base sont des sous tableaux (représente la coordonné y). Tandis que les éléments des sous tableaux sont des objets (représente la coordonné en x).

L’arraylist est utilisé pour stocker les différentes coordonnés des objets qui sont « win ». Autrement dit, les objets qu’il faut atteindre pour gagner la partie. Cette structure de donné nous permet d’ajouter et de supprimer des objets plus facilement qu’un tableau.

Le dictionnaire est utilisé dans le cadre de faire une correspondance entre deux objets. Dans notre cas, elle est faite pour faire le lien entre un objet de type entité et un objet de type rules. Cela permet d’éviter des « switch() » un peu trop long.

Complexité globale

Notre utiliserons la complexité dans le pire des cas pour cette partie. Comme vous vous en douterez, la méthode qui a la complexité la plus « lourde » est « move() ». Pour des raisons de simplification, les méthodes implémentés dans « move() » sont déjà évalués et on évalue qu’un « case » du switch vu que c’est la même manière pour tous.

Move :

switchObject() T(n\*2)

setTempObjectMap() T(n\*2)

searchWin()  T(n\*2)

if(thingIsYou()) T(n)

switch() T(1)

for(int i = 0 ; i <= map.length – 1 ; i++) T(n)

for(int j = 0 ; j <= mapO[i].length – 1 ; j++) T(n)

if(this.getClass().isInstance(maO[i][j])) T(1)

if(canMove()) T(1)

Action.up() T(1)

Else if(thingIsPushing()) T(1)

Action.pushY() T(n)

If(thingHasWin()) T(n)

winStatus = true  T(1)

break T(1)

BigAlgorithm.actualise() T(n\*2)

actualiseObjectMap()  T(n\*2)

Nous avons donc T(n\*2 + n \*2 + n\* 2 + n + n x n x (1 + 1 + n + n + 1)).

Après simplification, nous avons un temps en T(6n\*2 + 2n\*3 + n).

Ce qui nous donne une complexité en O(n\*3).

A noter que la complexité globale est en O(n\*2). O(n\*3) est la plus « lourde ».

2.2.View