Dossier du projet (20-21)

Groupe : BAKILI Shaïna, LOISY Nicolas , ROY Rudra

Ce fichier reprend la description des documents demandés pour le projet, ainsi que quelques renseignements que vous devez fournir.

Vous devez le compléter et le mettre à disposition sur git.

Tous les diagrammes devront être fournis avec un texte de présentation et des explications.

1 Documents pour CPOO : sur 19,5

Chaque diagramme utilisé à des fins de documentation doit être focalisé sur un objectif de communication. Il ne doit par exemple pas forcément montrer toutes les méthodes et dépendances, mais juste ce qui est nécessaire pour montrer ce que le diagramme veut montrer. Chaque diagramme doit être commenté.

1.1 Diagrammes de classe (8,5 points)

1 : Vous donnerez une vision d'ensemble de la partie modèle de votre programme à l'aide d'un ou de plusieurs diagrammes de classe commentés.

2 : Vous choisirez des parties du modèle que vous considérez particulièrement intéressantes du point de vue de la programmation à objets (héritage, composition, polymorphisme…).

Vous expliquerez vos choix et les illustrerez par des diagrammes de classe.

Diagramme de classe Amnesiacor

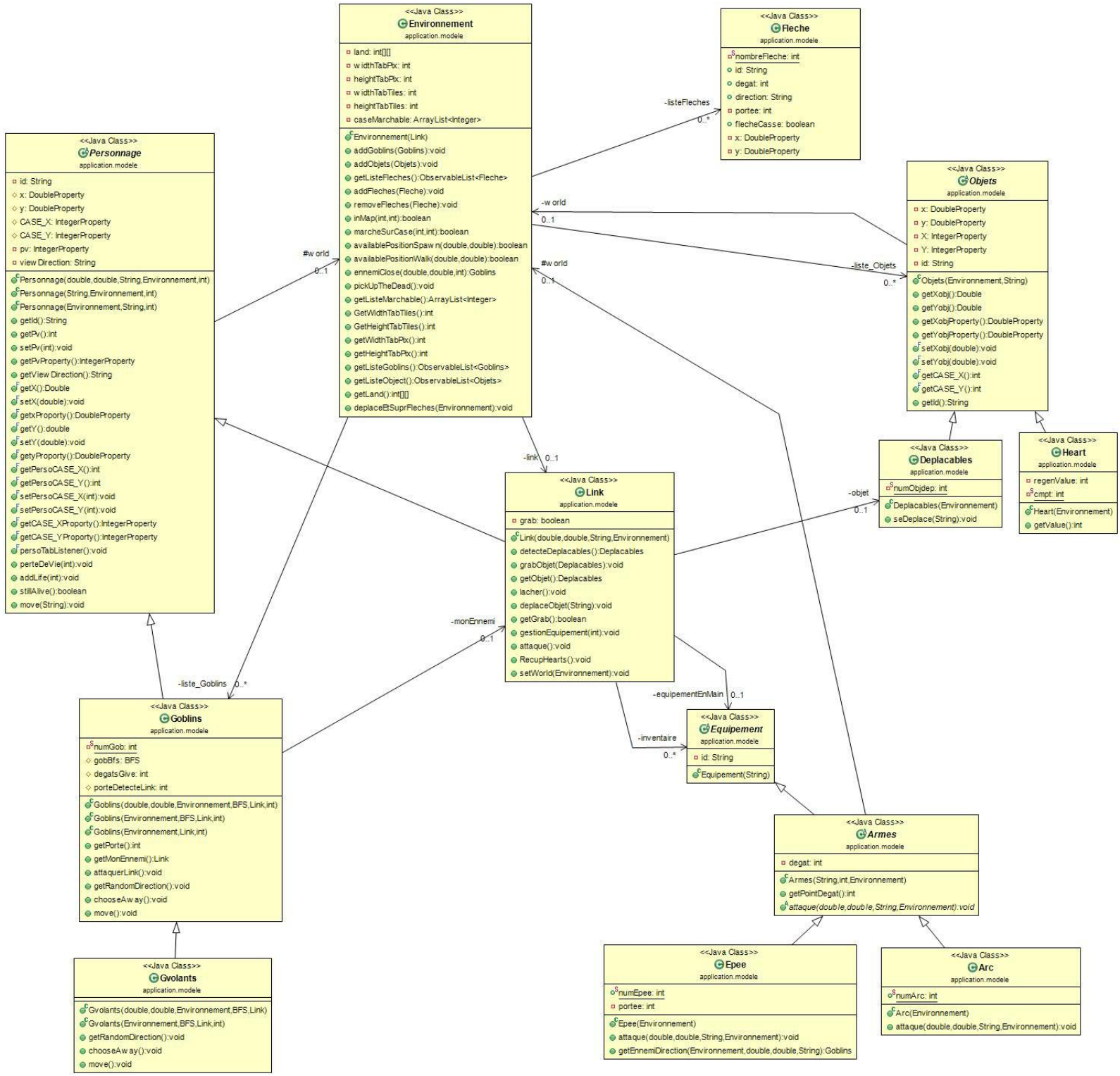
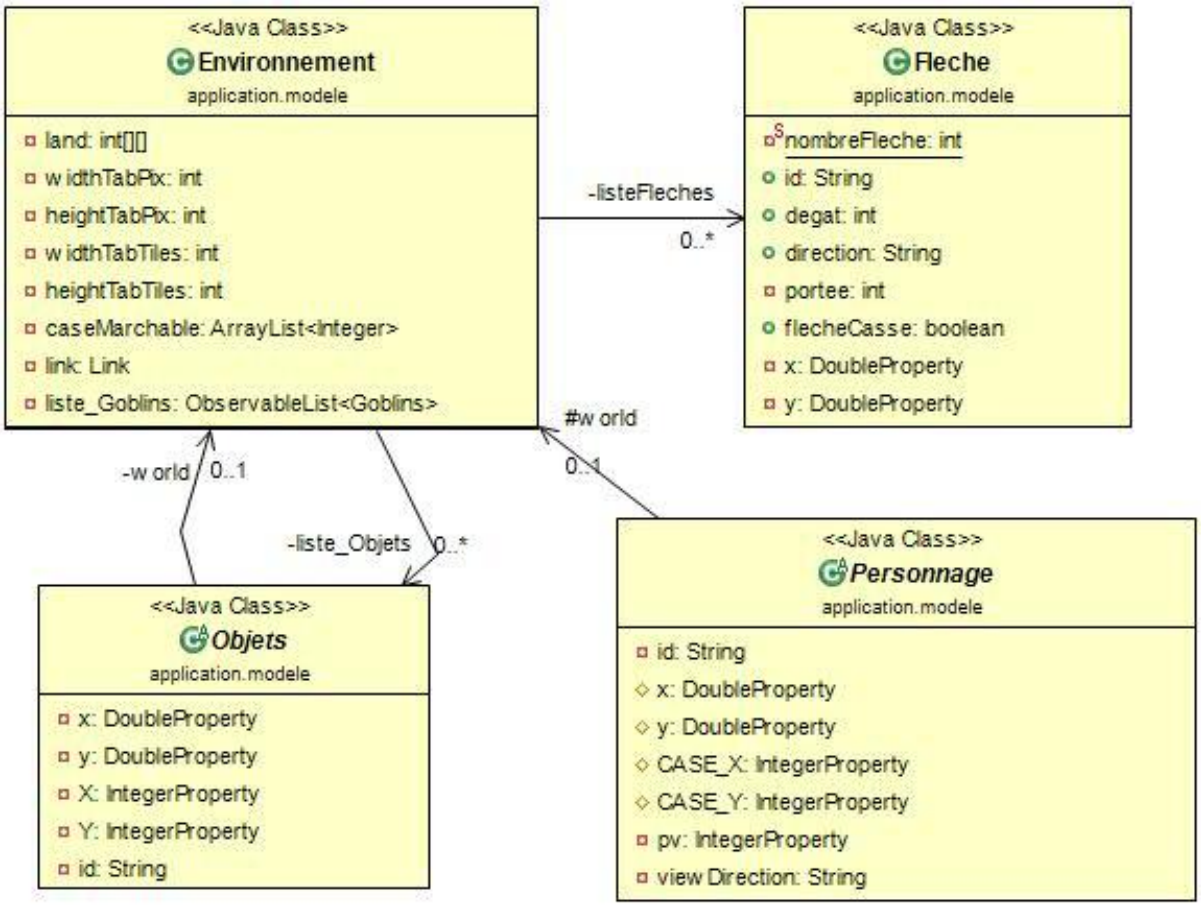
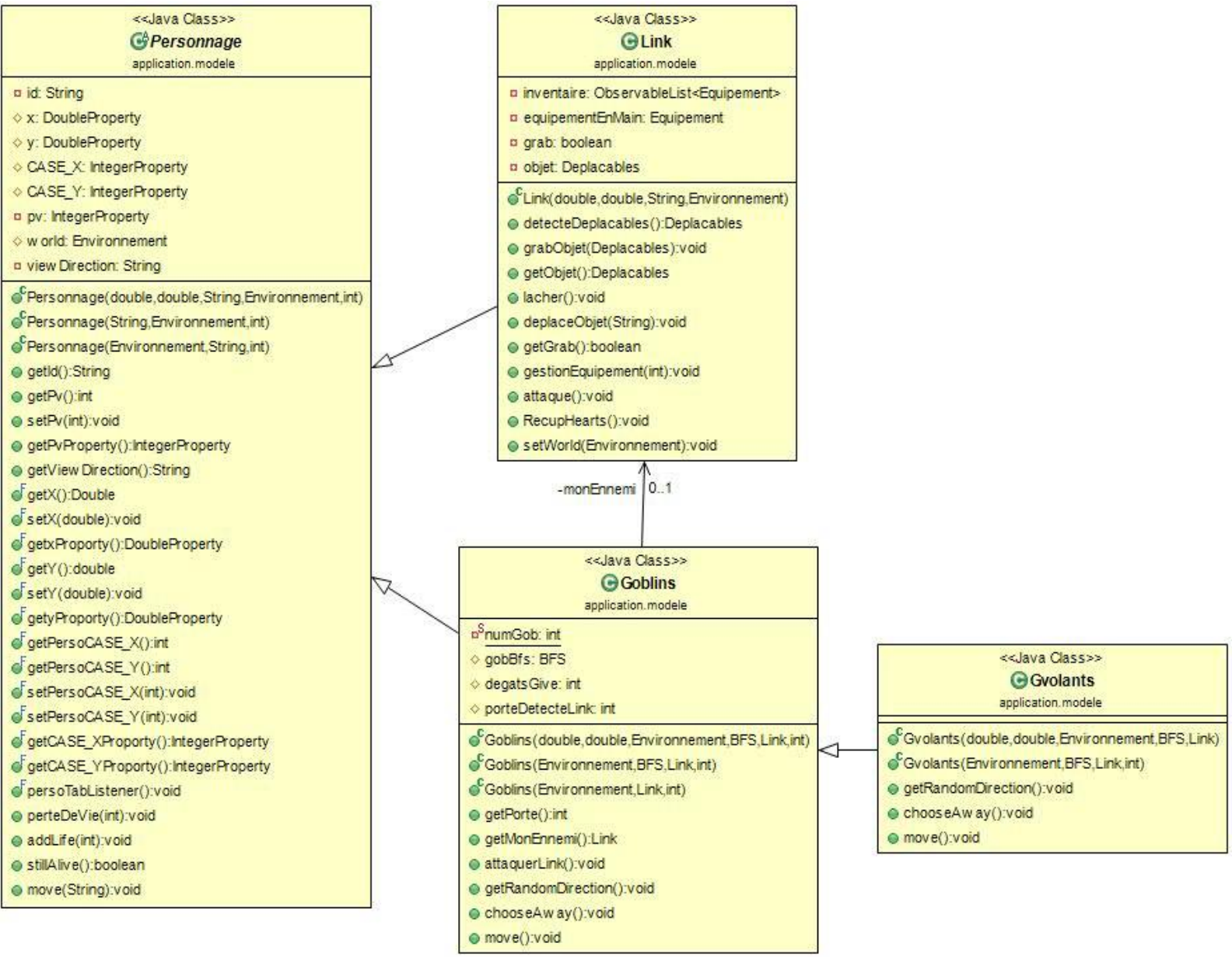


Diagramme de classe : Environnement



La classe “Environnement” est le terrain du jeu. Cette classe possède une dimension, un héros, une liste d'ennemis, une liste d’objets et une liste de projectiles(Flèches). Le rôle de cette classe est la gestion des éléments se trouvant dans le jeu.
Elle a aussi des fonctions se consacrant à la vérification de la map, comme par exemple la fonction “**availablePositionSpawn()**” permet de vérifier si la case sélectionnée permet l'apparition d’un personnage.
La fonction “**availablePositionWalk()**” vérifie si un personnage peut marcher sur la case sélectionnée.

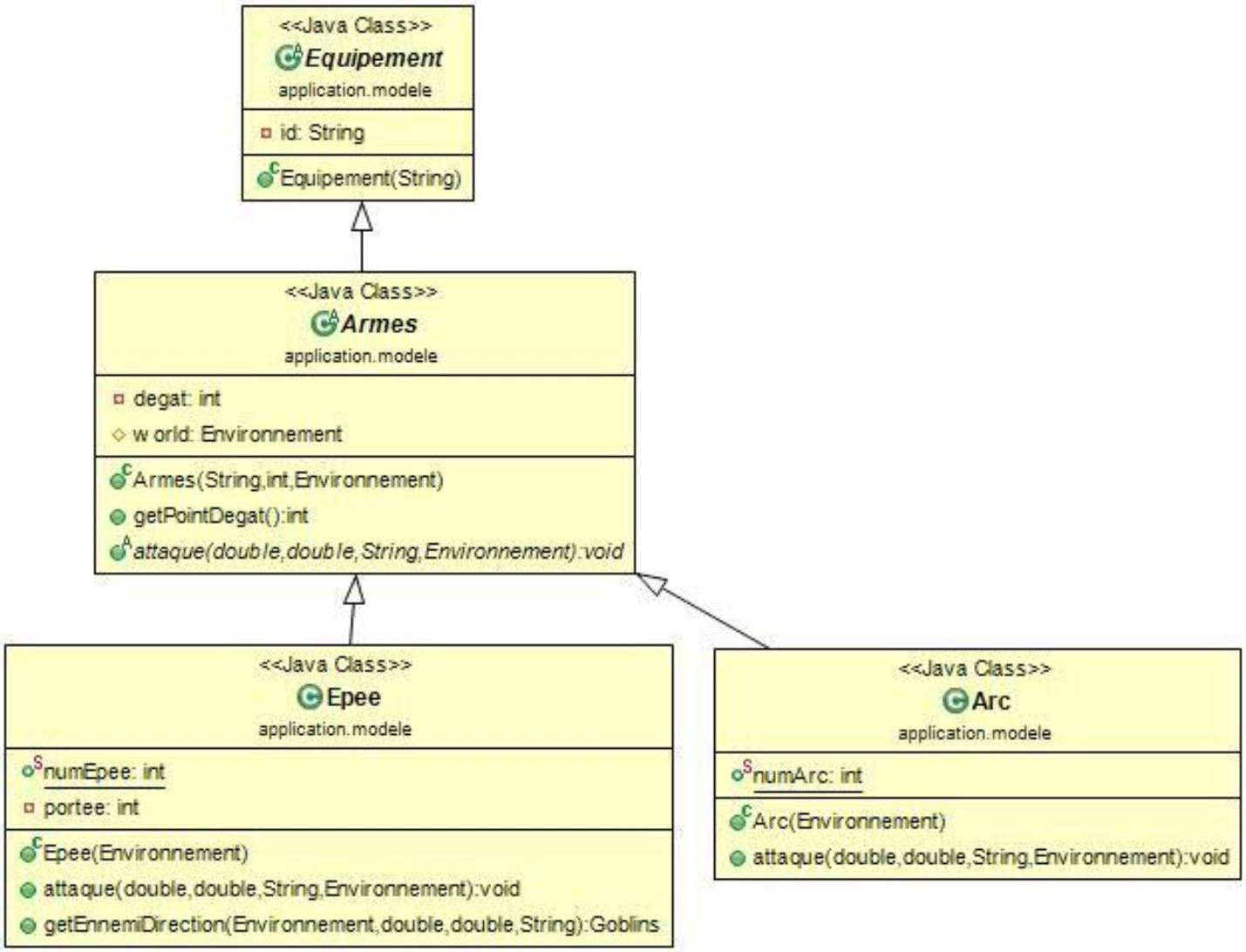
Diagramme de classe : Personnage



Ci-dessus la super classe “Personnage” et ses sous classes “Link”, “Goblins”, “Goblins volants” (sous classe de Goblins). “Personnage” est la classe regroupant toutes les caractéristiques et statuts (position dans le terrain, points de vie, statuts en vie ou pas) des différents personnages du jeu, pour notre part le “Link” et les goblins (ses ennemis). Un personnage peut donc se déplacer dans le monde, attaquer un autre personnage et même mourir.

Mais pourquoi ces sous classes ? Les grandes différences entre le “Link” et les “Goblins” sont leur manière d’attaquer, de se déplacer et le fait de pouvoir ou pas récupérer/déplacer des objets. Pour se déplacer, les “Goblins” utilisent le BFS via **chooseAway()** ou leurs randomiseur de position **getRandomDirection()** (**move(string)** de *Personnage* appelé dans chacune de ces méthodes) pour se déplacer.La classe de Link font enfin appel à la méthode **move()** de leur classe pour choisir la méthode de déplacement. Le “Link” utilise également la méthode **move(string)** *(de “Personnage”)* pour se déplacer mais sans BFS ni randomiseur puisque c’est l’utilisateur qui le déplace.
Du côté des attaques, les goblins attaquent automatiquement le link s’ils sont a une case directe de ce dernier (***attaqueLink()***). Le “Link” appel la méthode attaque de ses armes via sa méthode ***attaque()*** mais nous verrons cela avec détaillés plus loins. Enfin contrairement à ses ennemis le Link peut récupérer des coeur ou bien déplacer des objets dit "Deplacables"

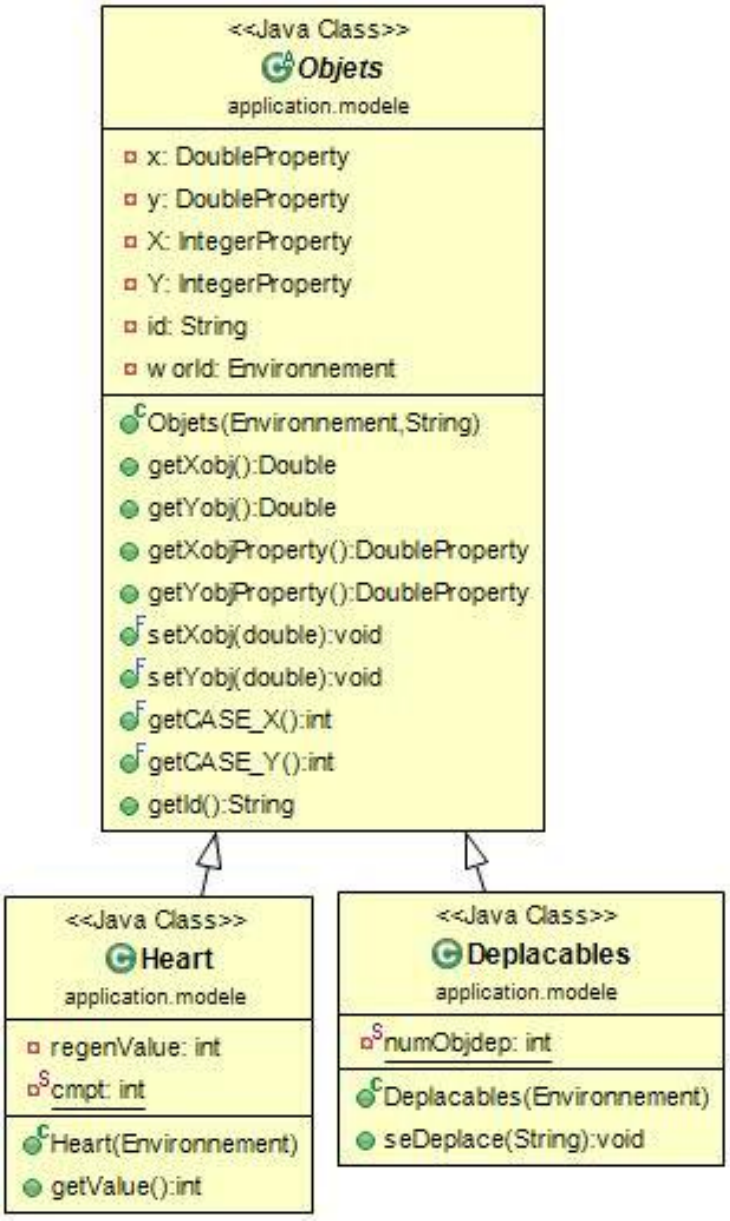
Diagramme de classe : Équipement



Voici, ci-dessus la super classe “Équipement” qui nous servira pour l’inventaire du jeu.
L’inventaire est est conçu pour accueillir des Armes et d'autres types d’équipement au futur.
La sous-classe “Armes” est la superclasse de toutes les armes, on y retrouve donc les sous-classes “Épée” et “Arc”.

Chacune des armes comporte une fonction “**attaque()**” qui a un fonctionnement radicalement différent selon l’arme.
Les Épées peuvent avoir une caractéristique différente qui est la portée.
La fonction attaque des armes a comme paramètre la position de Link ainsi que sa direction.
Cela permet à l'Épée de récupérer l’ennemi en face de Link (avec “**getEnnemiDirection()**”) pour lui retirer des PVs.
Pour l’Arc, les paramètres sont utilisés pour la création des flèches qui seront tirés dans la direction indiquée.

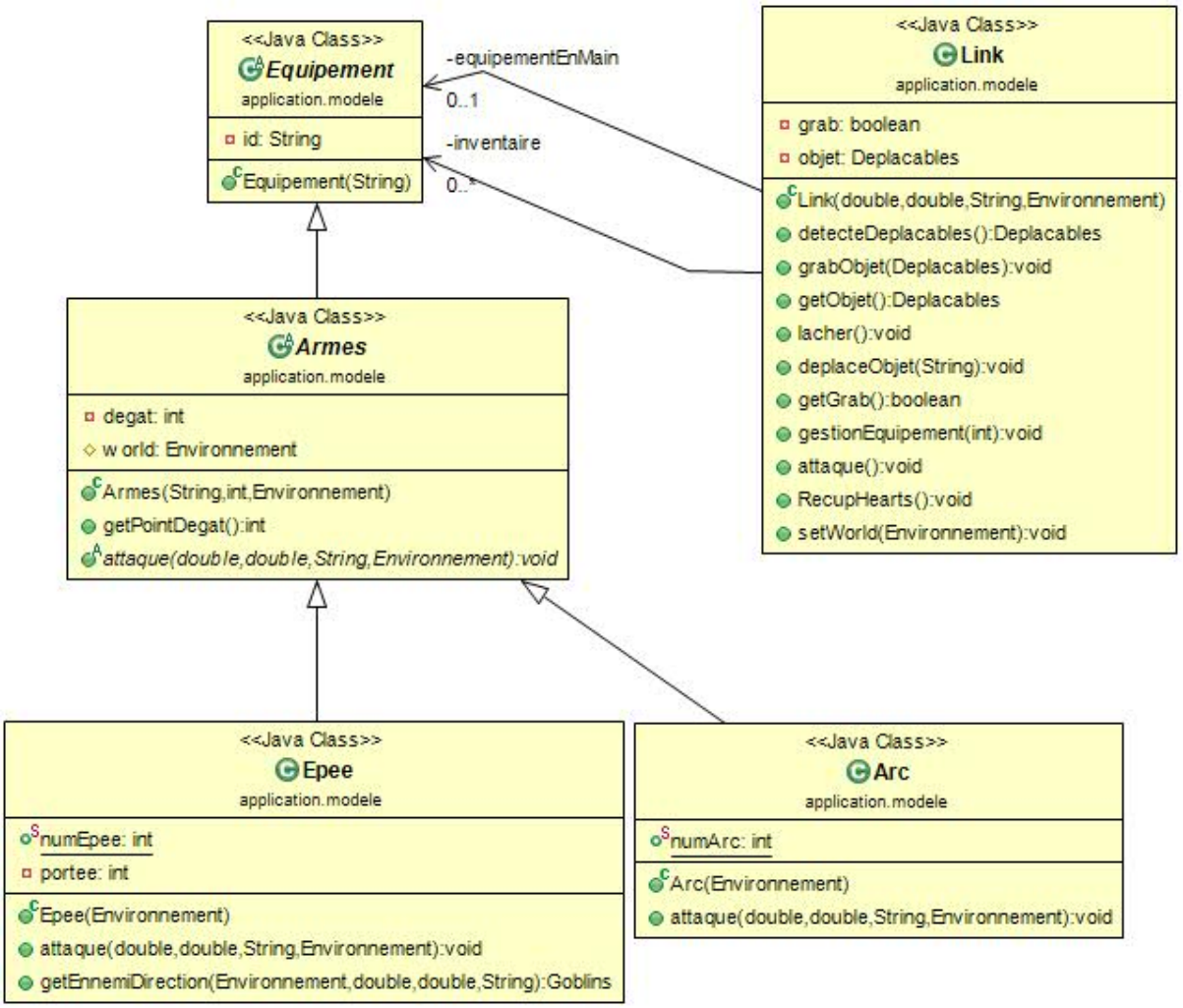
Diagramme de classe : Objets



Ci-dessus se trouve la classe “Objet” avec ses sous-classes “Heart” et "Deplacables".
La classe “Objets” correspond à tous les éléments de la map où Link peut interagir.

Elle contient les informations générales (positions, id,...) permettant d’identifier chaque objet.
La classe “Heart” représente les cœurs ramassables sur la map.
La fonction **getValue()** permet de récupérer le nombre de PVs à régénérer, qui sera appelé dans la fonction **RecupHearts()** du Link pour augmenter les points de vie de notre héros. La classe “Deplacables” définit tous les objets pouvant se mouvoir grâce au héros. La fonction **seDeplace()** permet de déplacer l’objet, suivant des mécanismes similaires aux mouvements du personnage.

Diagramme de classe : Link (attaque)

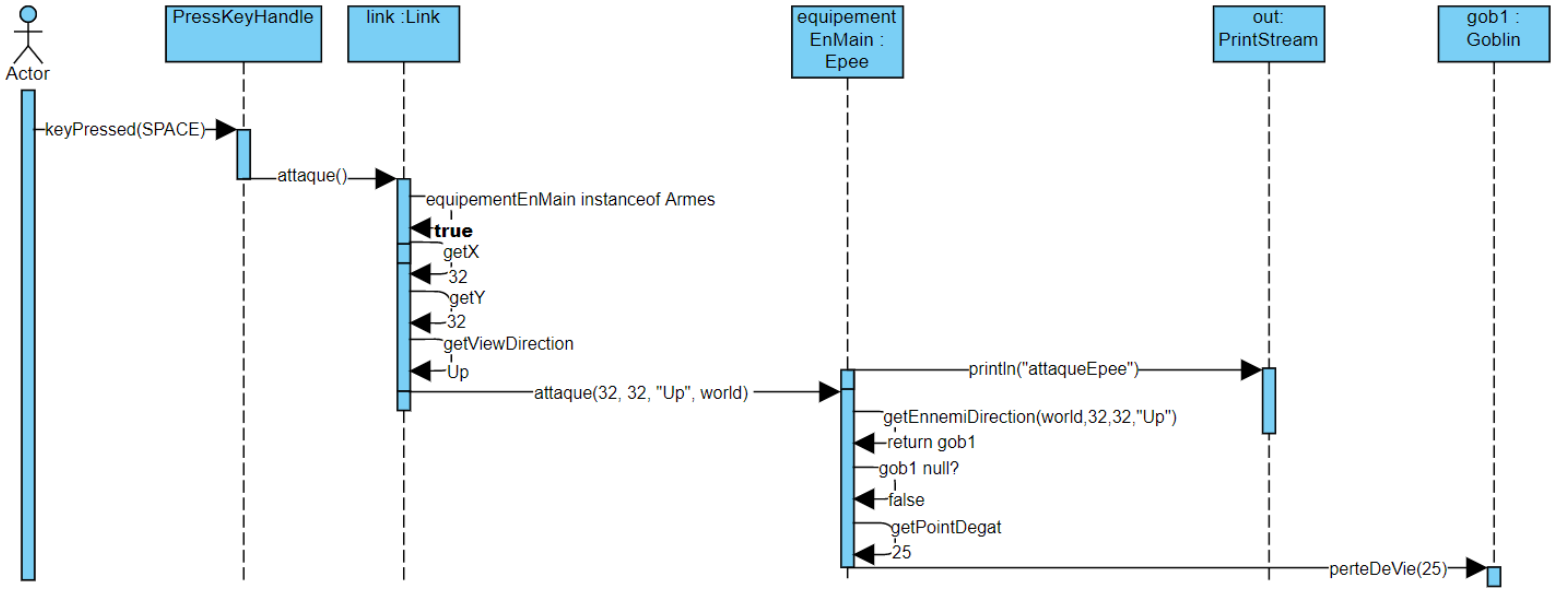


Voici, ci-dessus la classe “Link”.
Cette classe comporte un inventaire qui est une liste d’Equipement. Cette liste permet de gérer les items que Link possède, principalement des Armes.
La fonction **gestionEquipement()** permet d’équiper ou de déséquiper un “Equipement” en mettant ce dernier dans l’attribut “equipementEnMain”.
La fonction “**attaque()**” de Link vérifie si l’objet “Equipement” est bien une sous classe “Arme” puis appelle la fonction **attaque()** de l’arme en Main. Par polymorphisme, la fonction attaque sera différente en fonction de la sous-classe d’Arme en main.

1.2 Diagrammes de séquence (6,5 points)

Vous choisirez une ou deux méthodes intéressantes du point de vue de la répartition des responsabilités entre différentes entités du programme. Vous utiliserez des diagrammes de séquence pour expliquer l'exécution de ces méthodes. Ces diagrammes doivent être faits à la main.

Diagramme de séquence : Attaque (Link)



Ci-dessus, nous pouvons voir le diagramme de séquence d'attaque du Link. Cela nous permet de voir comment se déroule le programme après une action de l'utilisateur (ici la touche SPACE).

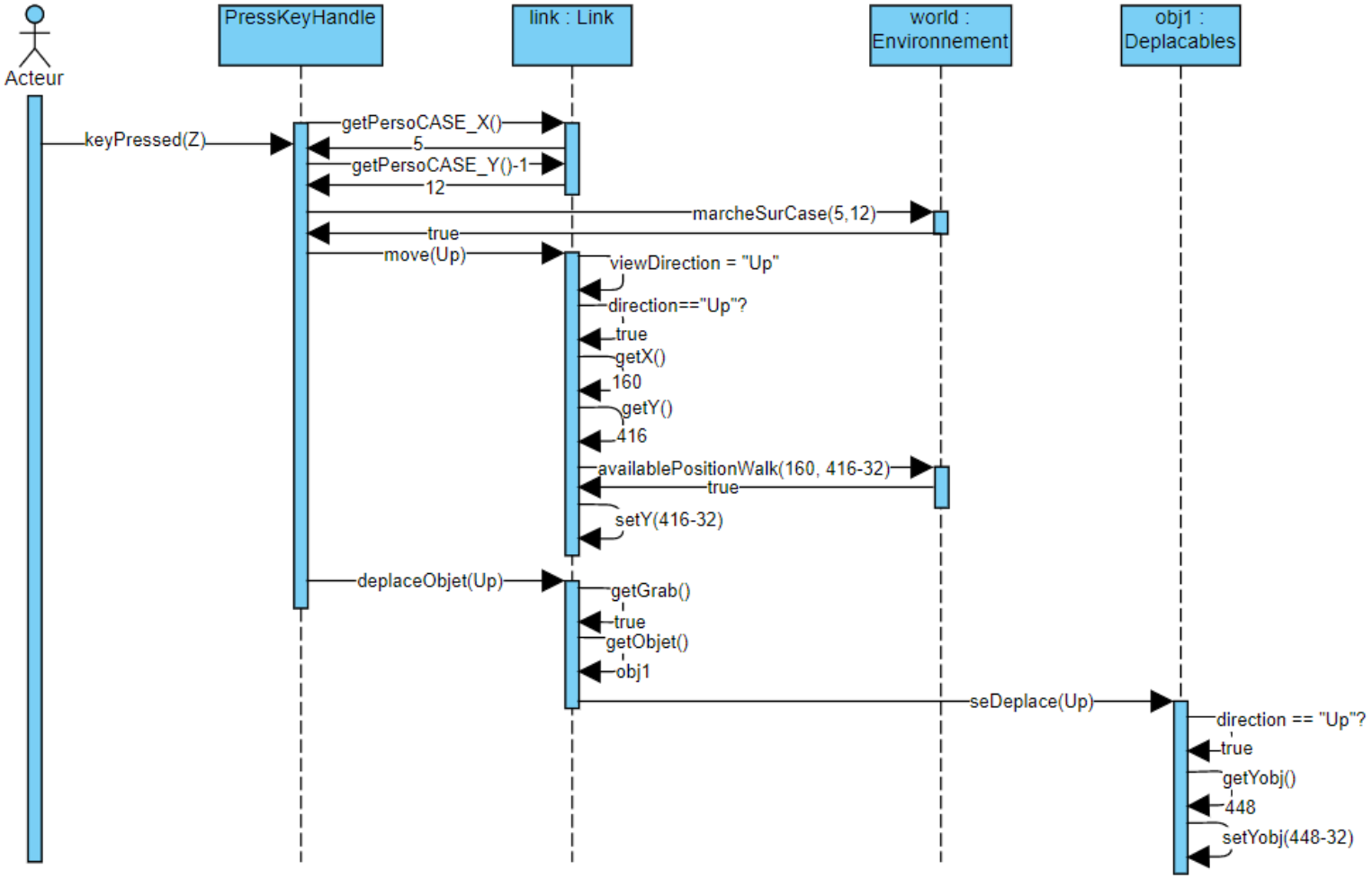
Dès que la touche SPACE est pressée, la classe PressKeyHandle (classe qui écoute toutes les saisies claviers) appelle la fonction **attaque()** du Link.

Si l'équipement en main est bien une arme, la classe appelle la fonction **attaque()** de l'arme en main avec les positions du Link et sa direction.

Dans cet exemple, c'est l'Epee qui est en main. Ainsi, l'Epee va vérifier s'il y a un ennemi devant Link, si c'est le cas et donc que l'ennemi n'est pas null, l'Epee appelle la fonction **perteDeVie()** de l'ennemi (classe Goblin) avec le nombre de dégâts de l'arme en main en paramètre.

Le fait que ce soit les Armes qui "attaquent" nous permet de faire du polymorphisme puisque si un Arc était l'équipement en main, le diagramme de séquence aurait complètement changé. La fonction attaque de l'Arc ne fonctionne pas comme celle de l'Epee ("Arc" et "Epee" sont des sous-classes de "Arme").

Diagramme de séquence : Mouvement (Link)



Ci-dessus, nous pouvons voir le diagramme de séquence sur le mouvement de notre personnage. Lorsque la touche “Z” est pressée, La classe “PressKeyHandle” vérifie si la case de la direction regardée est une case où l’on peut marcher dessus.

Cette vérification fait appel à la fonction **marcheSurCase()** de la classe “Environnement”, cette fonction prend en paramètres les fonctions **getPersoCASE_X()** et **getPersoCASE_Y()**, ces fonctions de la classe “Personnage” (super classe de “Link”) permettent de récupérer les coordonnées de la case en question. Après la vérification, la fonction **move()** de la classe “Link” est appelée avec comme paramètre “Up”. La fonction **move()** procède à la vérification du paramètre et la présence d’un ennemi ou d’un obstacle grâce à la fonction **availablePositionWalk()** de la classe "Environnement". Si la vérification renvoie une réponse “true”, les positions de notre personnage sont alors modifiées.

En ce qui concerne le déplacement d’un objet, nous appelons la fonction **deplaceObjet()** de “Link” avec le même paramètre que dans la fonction **move()**. Dans la fonction **deplaceObjet()** nous vérifions si le héros est en possessions d’un objet grâce à sa fonction **getGrab()**, pour ensuite déplacer sa “caisse” avec la fonction **seDeplace()** de l’objet, avec le

paramètre “Up”. La fonction **seDeplace()** vérifie la direction proposée et modifie ainsi la position de l’objet.

1.3 Présentation d’algorithmes : (4,5 points)

Un texte présentant la liste des algorithmes intéressants que vous avez mis en œuvre et comment vous les avez implémenté (choix des structures de données etc...)

I. Le BFS

Pour notre BFS nous avons utilisé la méthode A* (A star) qui est très répandu dans les jeux vidéos. Elle consiste à mettre depuis un point de départ des “poids” de distance sur chaques cases de la map.

Pour procéder à tout cela nous avons décidé de créer une classe à part nommée BFS.

```
3  public class BFS {
4      private Environnement world;
5      private Link link;
6      private ArrayList<Integer>ListeWorkable;
7      HashMap<Integer, Integer> sizeWaysGobT;
8      HashMap<Integer, Integer> sizeWaysGobV;
9
0      public BFS(Environnement e, Link link) {
1          world = e;
2          this.link = link;
3          ListeWorkable = world.getListeMarchable();
4          sizeWaysGobT = new HashMap<Integer, Integer>();
5          sizeWaysGobV = new HashMap<Integer, Integer>();
6
7          // TODO Auto-generated constructor stub
8      }
```

Un BFS possède donc:

- un Environnement : celui dans lequel il est censé s’activer.
- Le link : qui est son point de départ.
- La liste des cases marchables : important pour les cases non marchables donc sans “poid”.
- La map de poids pour les goblins Terrestres.
- La map pour les goblins volants: différents car ne prend pas en compte les case marchable ou pas .

Le BFS (étant forcément créé puisque objet) s’active depuis tout autre classe ou il est créé grâce à ses méthodes **findAWayGobT()** ou **findAWayGobV()**.

Ces méthodes sont basiquement les mêmes, leur unique point de divergence est la vérification de case marchable ou pas (**world.marchesurcase(x,y)**) qui remplacer dans **findAWayGobV()** une vérification d'hors map ou pas (**world.inMap(x,y)**).

```
public void findAWayGobT(){//BFS s'active uniquement quand le link sera a un certain emplacement
    ArrayList<Integer>Queue = new ArrayList<>();
    ArrayList<Integer>Past = new ArrayList<>();//d?ja pass?
    sizeWaysGobT.clear();
    int size = 0;//map's value

    Queue.add(calculCase(link.getPersoCASE_X(), link.getPersoCASE_Y()));
    Past.add(calculCase(link.getPersoCASE_X(), link.getPersoCASE_Y()));

    int[][]currentCase ={{link.getPersoCASE_X()},{link.getPersoCASE_Y()}};

    while(Queue.size() != 0){

        if(size>1) {
            size = sizeWaysGobT.get(calculCase(currentCase[0][0], currentCase[1][0]))+1;

        }
        else {
            size++;
        }
        if(world.marcheSurCase(currentCase[0][0], (currentCase[1][0]+1)) && Past.contains(calculCase(currentCase[0][0], currentCase[1][0]+1))==false ){//UP
            Queue.add(calculCase(currentCase[0][0], (currentCase[1][0]+1)));
            Past.add(calculCase(currentCase[0][0], (currentCase[1][0]+1)));
            sizeWaysGobT.put(calculCase(currentCase[0][0], (currentCase[1][0]+1)), size);
        }
        if(world.marcheSurCase(currentCase[0][0], (currentCase[1][0]-1)) && Past.contains(calculCase(currentCase[0][0], currentCase[1][0]-1))==false ){//DOWN
            Queue.add(calculCase(currentCase[0][0], (currentCase[1][0]-1)));
            Past.add(calculCase(currentCase[0][0], (currentCase[1][0]-1)));
            sizeWaysGobT.put(calculCase(currentCase[0][0], (currentCase[1][0]-1)), size);

        }
        if(world.marcheSurCase((currentCase[0][0]+1), currentCase[1][0]) && Past.contains(calculCase(currentCase[0][0]+1, currentCase[1][0]))==false ){//LEFT
            Queue.add(calculCase((currentCase[0][0]+1), currentCase[1][0]));
            Past.add(calculCase((currentCase[0][0]+1), currentCase[1][0]));
            sizeWaysGobT.put(calculCase((currentCase[0][0]+1), currentCase[1][0]), size);
        }
        if(world.marcheSurCase((currentCase[0][0]-1), currentCase[1][0]) && Past.contains(calculCase(currentCase[0][0]-1, currentCase[1][0]))==false ){//LEFT
            Queue.add(calculCase((currentCase[0][0]-1), currentCase[1][0]));
            Past.add(calculCase((currentCase[0][0]-1), currentCase[1][0]));
            sizeWaysGobT.put(calculCase((currentCase[0][0]-1), currentCase[1][0]), size);
        }

        //enl?ve la case en tete de file
        Queue.remove(0);
        //change la case case observ?e
        if(Queue.size()>0){
        currentCase[0][0] = backToX(Queue.get(0));
        currentCase[1][0] = backToY(Queue.get(0));
        }
    }
}
```

Ci-dessus donc **findAWayGobT().**

Notre méthode pour fonctionner est composée de:

- **La Current Case** : position du link a l'initialisation puis position de la case qu'on traite, sous forme d'un tableau à deux dimensions.
- de variable "**size**" qui représente le poids.
- Une "**Queue"** liste : la liste des cases non traitées ou en cours de traitement, composé à l'initialisation uniquement de la position du link.
- Une "**Past**" liste : la liste de toutes les cases déjà traitées, composé à l'initialisation uniquement de la position du link.
- Enfin de la map **sizeWay** qu'elle remplit au fur et à mesure.

Nous vérifiant (pour l'algorithme principale) que la "**Queue**" soit non vide (et donc qu'il reste des cases à traiter) avant de rentrer dans la boucle de traitement. Puis ensuite pour chaque case valide se trouvant au tour de la "**Current Case**" nous les ajoutant dans la **Queue liste**, **la Past liste** et **sizeWay** map (avec son poids/distance au link) via le numéro de leur case en utilisant à la méthode **calculcase() ,** qui nous permet d'avoir le numéro d'une case d'un tableau grâce à sa position x et y.

```
public int calculCase(int x, int y){//return le numero de la case
    return (y*(world.GetWidthTabTiles())+x);
}
```

Après avoir check toutes les cases relatives à la “**Current case**”, nous la retirons de la “**Queue**” (étant traitée), et nous remplaçons la valeur de la “**Current case**” par la “nouvelle” première valeur de la “**Queue**” (méthode FIFO). Cette nouvelle valeur est sous forme de numéro de case puisqu’elle était dans la Queue, elle est donc traduite en position X puis Y avant d'être introduite dans la “**Current Case**” et cela à l’aide de nos méthodes **backToX()** & **backToY()**.

```
public int backToX(int numCase){
    return (numCase%(world.GetWidthTabTiles()));
}
public int backToY(int numCase) {
    return ((numCase-(backToX(numCase)))/(world.GetWidthTabTiles()));
}
```

L'algorithme se termine donc quand la Queue est vide. Pour visualiser les map nous utilisons **displaySizeWay()**.

```
public void displaySizeWay() {
    System.out.println(sizeWaysGobV);
    System.out.println(sizeWaysGobT);
}
```

II. Création de la MAP (JsonReader)

Pour la création de la Map, nous avons utilisé le logiciel “Tiled”, qui génère un fichier sous le format Json.

Après des recherches et des essais concluant sur la lecture des fichiers avec Java, nous avons cherché un moyen de lire un fichier Json. La solution la plus efficace que nous avons trouvé a été l’utilisation d’une librairie spécifique à la lecture des fichiers au format Json. C’est pour cette raison que nous avons introduit la librairie “Json-simple-1.1.1” dans notre projet.

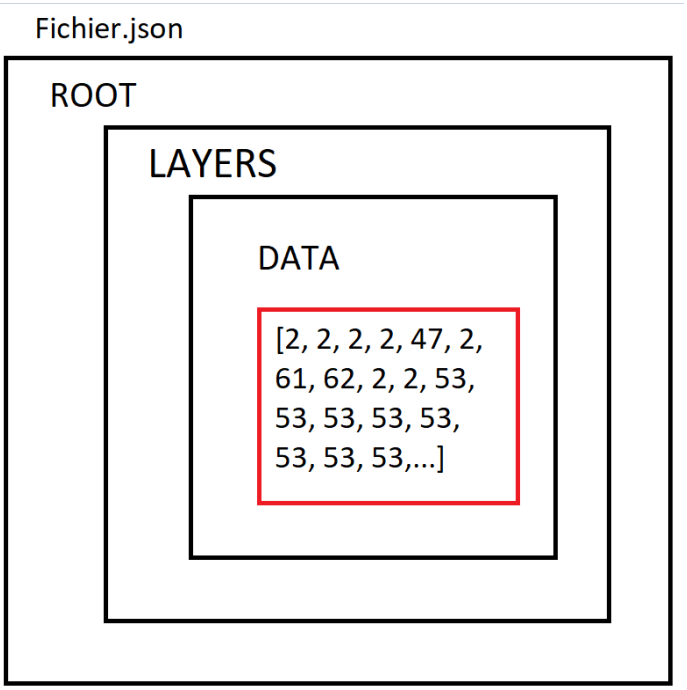
Les fichiers Json ont une structure ressemblant au langage Web avec des balises et des sous-balises.

La librairie nous permet de créer des objets Json (JsonParser, JsonObject, JsonArray).

```
JSONParser parser = new JSONParser();
JSONObject root = (JSONObject) parser.parse(fileContent);
JSONArray layers = (JSONArray) root.get("layers");
JSONArray data = (JSONArray) ((JSONObject) layers.get(0)).get("data");
```

L’utilisation de ses objets dans notre code est similaire au concept des poupées russes.

Une balise comprend une autre balise, qui peut comprendre une autre balise qui contient des données.



```
{ "compressionlevel":-1,
 "editorsettings":
    {
     "export":
        {
         "format":"json",
         "target":"minish.json"
        }
    },
 "height":20,
 "infinite":false,
 "layers":[
        {
         "data":[2, 2, 2, 2, 47, 2, 61, 62, 2, 2, 53, 53, 53, 53, 53, 53,
         "height":20,
         "id":1,
         "name":"Calque de Tuiles 1",
         "opacity":1,
         "type":"tilelayer",
         "visible":true,
         "width":20,
         "x":0,
         "y":0
        }],
 "nextlayerid":2,
 "nextobjectid":1,
 "orientation":"orthogonal",
 "renderorder":"left-up",
 "tiledversion":"1.6.0",
 "tileheight":32,
 "tilesets":[
        {
         "firstgid":1,
         "source":"zelda_oot v2.0.tsx"
        }],
 "tilewidth":32,
 "type":"map",
 "version":"1.6",
 "width":20
}
```


Ici, Root (le fichier dans sa globalité) comprend : compressionlevel, editorsettings, height, infinite, layers, ...

Ce qui nous intéresse dans le fichier, c'est la taille du tableau (height et width) ainsi que les données du tableau (layers).

Pour avoir accès au données de layers, on va donc :

1- Créer un JSONParser, qui va nous permettre de lire le fichier.

2- Créer un JSONObject que l'on va appeler root, qui va prendre l'entièreté du contenu du fichier.

3- Créer une liste JSONArray que l'on va appeler layers, qui va prendre dans root le contenu de layers.

4- Créer une liste JSONArray que l'on va appeler data, qui va prendre les données dans layers.

Lors de la définition du tableau, nous y introduisons les dimensions prises dans le fichier json de la même manière que vu précédemment. Ensuite, nous créons le tableau en parcourant les valeurs de "data".

```
int[][] tab = new int [Integer.parseInt(root.get("height").toString())][ Integer.parseInt(root.get("width").toString())];


int i = 0;
for(int j = 0; j < tab.length; j++) {
    for(int k = 0; k < tab[j].length; k++) {
        tab[j][k] =  Integer.parseInt(data.get(i).toString());
        i++;
    }
}
return tab;
```

Maintenant que nous avons le tableau avec les numéros des tuiles à afficher, un problème survient.

Comment avoir les coordonnées (x et y) de la tuile à afficher avec le numéro de la tuile?

On a trouvé les formules suivantes pour retrouver ses coordonnées à partir du numéro de la case.

Pour x = ((numCase-1)*modulo5)*32

Pour y = (arrSup(numCase/5)-1)*32

Exemple : Pour la case 8.

Pour x = ((8-1)modulo5)*32 = 64

Pour y = (arrSup(8/5)-1)*32 = 32

2 Informations à fournir pour le code :

2.1 Structures de données:

indiquez ici quelles structures de données avancées vous utilisez et dans quelles classes.

Dans notre projet nous avons utilisé les structures de données suivantes:

- ➢ Hashmap
- ➢ Filepath

2.2 Exception :

indiquez ici quelle(s) classe(s) contiennent une gestion d'exception intéressante.

La classe **Personnage** est la classe gérée avec des exceptions.

2.3 Junits :

indiquez ici quels classes sont couvertes par vos Junits.

La classe **Environnement** est couverte par des tests Junits.

3 Gestion de projet : 1 de coefficient

(1,5 de coefficient PPN pour le module : 1 pour le projet et 0,5 pour le contrôle)

Document utilisateur (8 points) :

- Description du jeu (son objectif, son univers…)
- Description des armes et accessoires de Link, des types d'éléments de la map, des ennemis (tableau de relation ...).
- Toutes fonctionnalité définie et utilisable doit être documentée. A l'inverse il est hors de question (et donc pénalisant) de documenter une fonctionnalité non encore utilisable.

Ce monde est inspiré d'une histoire fausse.
Notre héros vient tout juste de se réveiller d'un sommeil de plusieurs décennies.
En sortant du temple des sables qui lui servit de refuge lors de son long sommeil, rien, plus aucun souvenir !
Sa mémoire... complètement effacée.
Partez à l'aventure pour chercher des réponses et vous battre pour rétablir le calme sur ce monde obscurci par les ténèbres !

Les touches:

Touches de déplacement : Z/Q/S/D
Touche Inventaire : 1/2
Touche Attaque : ESPACE
Touche Prendre : E
Touche Lâcher : R

L'objectif du jeu est de combattre les gobelins pour sauver le monde. L'univers ressemble beaucoup à ceux des jeux Zelda.

Les armes que Link peut utiliser sont une épée et un arc.

Avec l'épée, Link attaque sur son chemin pour ravager ses ennemis.

Utilisez votre Arc pour terrasser à distance vos ennemies et transpercer les avec vos flèches.

Vos plus grands alliés lors de votre aventure seront les cœurs, chercher les pour augmenter les PV de Link.

Les ennemis vous poursuivent, vous traquent, vous attaquent ?

Utilise les objets déplaçables tels que les caisses. Link peut faire bouger les caisses pour bloquer le chemin de ses ennemis.

Les ennemis se résument en deux catégories, les ennemis terrestres et aériens. Les ennemis terrestres prennent en compte la collision avec les obstacles, ce qui n'est pas le cas pour les ennemis aériens.

Le seul moyen d'arrêter les ennemis est de placer une caisse (objet déplaçables) devant.

---INFO GAME---

RESOLUTION: 1920x1080
DIM TABLEAU : 60/33
NB GOBLINS: 5

Utilisation de Trello et git (12 points): bonne utilisation des outils Trello et Git.r. I