Manuel développeur

Anna BRUEL - David BUI - Quentin FAURE
Edwin NIOGRET - Florian POPEK - 赵子龙 (Zilong ZHAO)

Sommaire

Sommaire 2

Présentation générale 3

Architecture 4

Diagramme de classe 5

Ajout de fonctionnalités 8

- Nouveau bloc 8
- Nouvelle action 8
- Nouveau niveau 8
- Autres fonctionnalités 8

Extrait de code 9

Présentation générale

Lightbot est un jeu éducatif qui a pour but d'enseigner la programmation de manière ludique aux enfants. Pour ce faire, Lightbot confronte les joueurs à différents puzzles dans lesquels il faut allumer des lumières. Pour les résoudre, une suite d'actions est donnée au personnage, qui sera exécutée par la suite.

Chaque série de puzzle permet de découvrir un nouveau concept de la programmation, de plus en plus « technique ». Chaque concept est introduit dans un premier niveau simple, puis montre ses possibilités au travers de niveaux plus poussés. Pour clôturer chaque série de puzzle, un niveau, qui reprend tous les concepts précédents ainsi que le nouveau concept, est à résoudre.

Les mécanismes utilisés par Lightbot permettent donc d'appréhender la programmation sans avoir à se lancer dans l'apprentissage d'un langage spécifique. Beaucoup de concepts de programmation pourraient être introduits avec Lightbot, même les plus complexes, mais ce n'est pas son but. Le but principal étant de montrer des concepts avancés de manière abordable.

Architecture

L'application est divisée en plusieurs packages, chacun contenant les classes utiles pour une utilisation précise. Ainsi, l'application est actuellement répartie en 5 package :

- Entities : Contient tout ce qui est représentable dans le monde
- Game : Contient ce qui est propre a une partie du jeu
- Levels : Contient les classes permettant de créer un niveau (depuis un XML)
- Prog : Contient toutes les classes d'actions et de procédures
- UI : contient toute la gestion de l'interface graphique et les interactions avec l'utilisateur

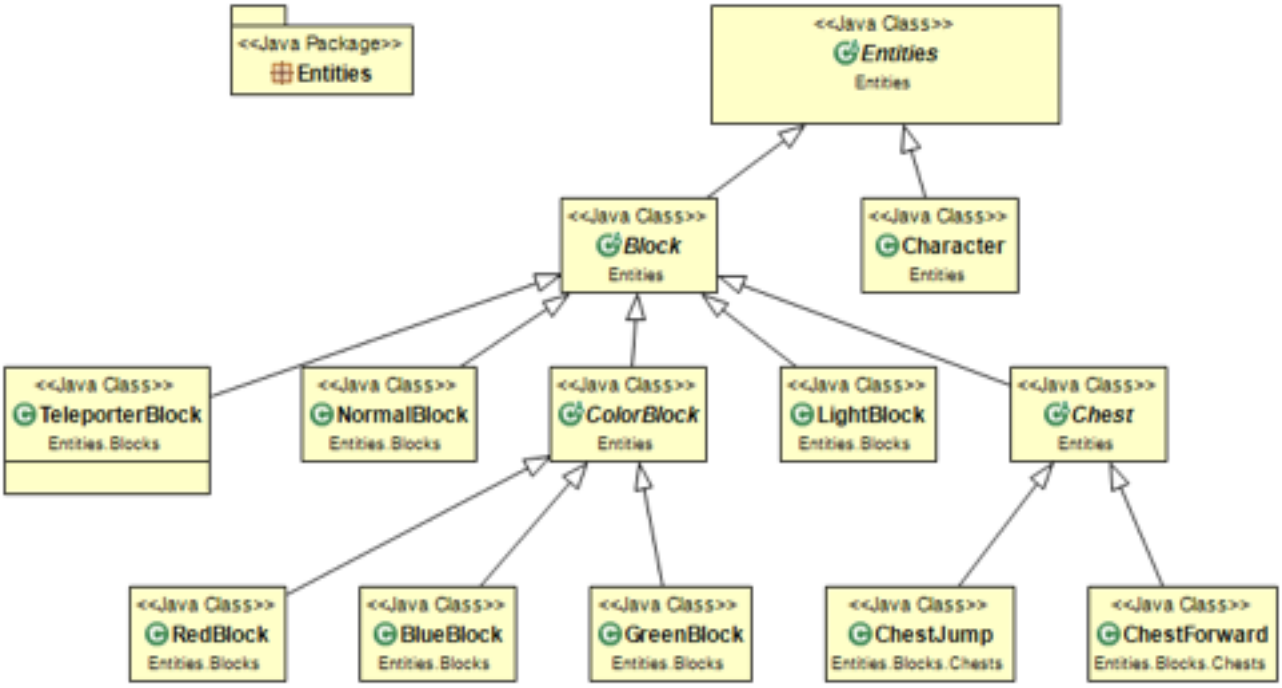
De plus, certains packages ont été découpés en sous packages, afin de regrouper les classes similaires et d'alléger les packages principaux. En voici la liste :

- Entities.Block : Contient tous les types de blocs
- Entities.Block.Chest : Contient un type particulier de blocs : les coffres
- Prog.NormalActions : Contient les actions de bases du robot (séparation avec les procédures et autre)

Pour plus de modularité, tout ce qui peut être affiché hérite de la même classe abstraite, et est caractérisé par des Coordonnées x, y et z au minimum. Ce mécanisme est aussi utilisé pour les Actions et procédures.

Diagramme de classe

Notre version de Lightbot est conçue autour de son moteur graphique, la séparation en package et la hiérarchisation des classes sont faites pour optimiser le rendu graphique.

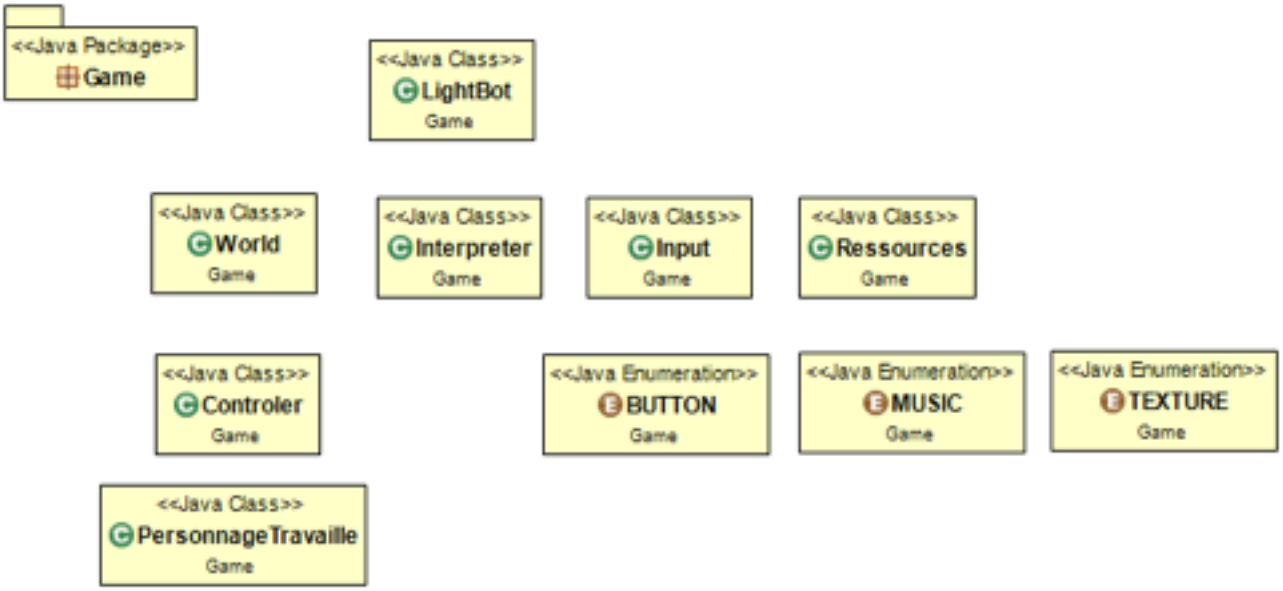


Les packages Entites, Entites.Blocks et Entites.Blocks.Chest permettent de créer toutes les composantes **affichables** par le jeu. Dans ces packages, on a l'élément **le plus basique** du jeu qui est un « entity » ; en découle de cet élément tous les autres blocks particuliers tels que les personnages, les blocks de couleurs, les blocks allumables, les coffres utilisables etc. Des fonctions de comparaison sont disponibles pour trier les éléments afin de les **afficher** dans le bon ordre. Ces éléments sont caractérisés par un **« sprite »** (l'image lui correspondant) et des **coordonnées**.

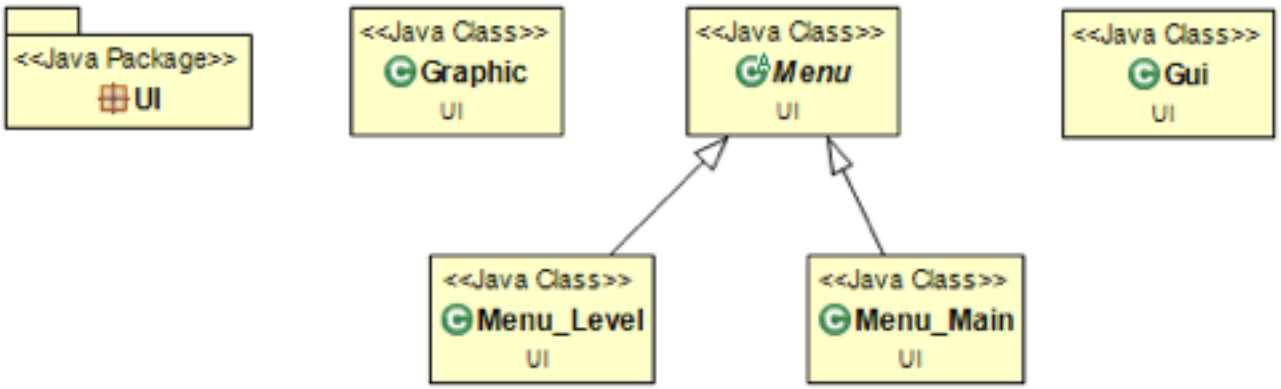
Pour ajouter un nouveau type de block, il faut l'ajouter dans Entites.Block ; quant à l'ajout d'un nouveau coffre offrant une nouvelle action, il faut l'ajouter dans Entites.Block.Chest

Le package Levels permet de créer des niveaux depuis un fichier XML et inversement. On retrouve dans ce package les classes Writer et Reader qui permettent de faire le lien entre le moteur graphique et la création du niveau en XML. L'ajout de nouvelles balises à parser se fait dans la classe *BeaconXML.java*. *LevelPreview.java* permet de faciliter l'édition de niveaux en les chargeant directement sans passer par le menu de sélection de niveau. Attention, les niveaux ne sont cependant pas jouables en passant par *LevelPreview.java*.

Les packages Prog, Prog.NormalActions et Prog.SpecialActions regroupent les actions que l'on donne aux personnages. Chaque personnage a une liste d'actions qui lui est propre. Toutes les différentes actions possibles sont décrites dans le package Prog.NormalAction, et l'ajout d'actions spéciales pour les coffres se fait dans Prog.SpecialActions.



Le package Game est le cœur du jeu. C'est *World.java* qui représente les mondes. L'application fait appel au Reader XML afin de créer le niveau et créer la liste d'action dont le joueur disposera afin de le résoudre. Puis c'est *Controller.java* qui gère si un personnage peut exécuter sa prochaine action : en effet, dans le cas où l'on a plusieurs personnages sur le niveau, chaque personnage doit exécuter le même nombre d'actions que les autres, or l'animation de certaines actions peut être plus rapide que d'autre. Il faut donc que *Controller.java* régule l'exécution des actions.
Interpreter.java est la classe qui **interprète** la liste d'action de chaque personnage. Il analyse la suite d'action et retourne la prochaine action que le personnage doit exécuter : par exemple si la prochaine action est « Avancer 5 fois » il faut que l'interpréteur renvoie « Avancer » et non « 5 fois ».



Le package UI gère l'interface graphique et les menus des niveaux, l'utilisation des ressources audios et images. La classe **Gui.java** est essentielle dans notre jeu : elle fait le lien entre les animations (celles des personnages, des blocks, de l'arrière-plan) et l'interpréteur. Quand une action est exécutée, Gui.java joue l'animation associée.

Ajout de fonctionnalités

Nouveau bloc

La hiérarchie des blocs rend très simple l'ajout de nouveaux blocs. Pour ce faire, il suffit de créer une nouvelle classe héritant de Block, et de l'implémenter. Afin qu'un bloc soit correct, il faut au minimum implémenter la méthode « perform », qui correspond à une interaction entre le personnage et le bloc. Afin d'interagir avec ce nouveau bloc, il est nécessaire de créer une nouvelle action.

Pour ce qui est des blocs de couleurs, ces derniers héritent d'une classe particulière : ColorBlock. Cette classe permet de regrouper les blocs de couleur et de diminuer le code à ajouter pour un nouveau bloc de couleur. L'ajout d'un nouveau bloc de couleur est similaire à un bloc normal, sauf qu'il n'y a pas à implémenter de méthode « perform ». La seule chose à faire est de créer un constructeur dans lequel est indiqué la couleur du bloc (classe Color) et son image à afficher).

Nouvelle action

Etant donné que la hiérarchie des actions est similaire à celle des blocs, l'ajout d'une nouvelle action est aussi simple. En effet, pour une action « basique », il suffit de créer une nouvelle classe héritant de la classe abstraite Action. Ensuite, il ne reste plus qu'à créer un constructeur (basé sur celui de la classe mère) et à implémenter la méthode « perform ».

S'il s'agit d'une action se basant sur d'autres actions (comme la classe For par exemple), il est nécessaire de modifier « Interpreter » afin d'y intégrer correctement la nouvelle action.

Nouveau niveau

Ajouter un nouveau niveau revient à créer un nouveau document XML. Ce document XML doit respecter une certaine syntaxe afin de créer un niveau valide. Cette syntaxe est explicitée dans le niveau « syntax.xml » dans le répertoire niveau.

Pour lancer ce niveau, il faut le lier dans le menu de choix des niveaux. Ceci se fait dans la classe « Menu_level » du package UI.

Autres fonctionnalités

Cette liste n'est pas une liste exhaustive de toutes les fonctionnalités qui peuvent être ajoutée. Il serait par exemple possible de passer à un monde sous forme de cube en 3D dans lequel il y aurait un personnage sur chaque face. Les changements à effectuer seraient au niveau de la classe « World », « Character » et sur les classes des actions pour prendre le nouveau monde en compte.

De plus, pour pouvoir stocker les niveaux, la syntaxe du XML serait aussi à modifier ainsi que la classe « Reader ». Les nouvelles balises seraient à ajouter dans la classe « BeaconXML ».

Extrait de code

Classe Block, utilisée comme classe mère de tout les blocs :

```
package Entities;

import Prog.Coordonnees;

/**
 * Classe identifiant les blocs affichable a l'ecran.
 *
 */
public abstract class Block extends Entities implements Cloneable
{
    public Block(Coordonnees coord)
    {
        super(coord);
    }

    public Object clone() throws CloneNotSupportedException {return super.clone();}

    /**
     * Préparation de la texture
     */
    public void initialiser() {}

    /**
     * Action effectuée  lorsqu'un personnage interagie avec un block
     * @param p : Le personnage se trouvant sur le bloc
     */
    public abstract void perform(Character p);
}
```

Une implémentation de la classe Block, le LightBlock :

```
package Entities.Blocks;

import org.jsfml.graphics.IntRect;

/**
 * Bloc de lumiere. Il faut allumer tout les blocks de lumieres pour terminer un niveau.
 *
 */
public class LightBlock extends Block
{
    /**
     * Cree un bloc lumiere eteint
     * @param pos Coordonnees de ce bloc
     */
    public LightBlock(Coordonnees coord)
    {
        super(coord);
        sprite.setTexture(Ressources.TEXTURE.getTexture(TEXTURE.BLOCK));
        isOn = false;
        initialiser();
    }

    private boolean isOn;

    private int anim = 0;
    private int one_frame = 8;

    /**
     * Gestion graphique
     */
    public void gerer()
    {
        if (isOn)
        {
            if (anim % one_frame == 0)
                sprite.setTextureRect(new IntRect(83+anim/one_frame*82, 83, 81, 81));

            anim++;

            if (anim == one_frame*5)
                anim = 0;
        }
    }
public void afficher()
{
    gerer();
    Graphic.SFML.draw(sprite);
}

public void initialiser() {sprite.setTextureRect(new IntRect(1, 83, 81, 81));}

/**
 * Permet de connaitre l'état du bloc (allumé ou non)
 * @return
 */
public boolean getLight() {return isOn;}

public void reverseLight()
{
    isOn = !isOn;
    if (isOn)
        sprite.setTextureRect(new IntRect(83, 83, 81, 81));
    else
        sprite.setTextureRect(new IntRect(1, 83, 81, 81));
}
```

```
    /**
     * Appele par l'action Light. Allume le bloc si il est eteint ou l'inverse dans le cas contraire.
     */
    public void perform(Character p) {reverseLight();}
}
```

Création des blocks à partir du XML dans le Reader :

```
        // Récupération des valeurs carctérisant le bloc
        String t = block.getAttribute(BeaconXML.B_BLOCK_TYPE).getValue();
        Attribute begin = block.getAttribute(BeaconXML.B_STARTING_BLOCK);
        int x = Integer.valueOf(block.getChild(BeaconXML.B_X).getValue());
        int y = Integer.valueOf(block.getChild(BeaconXML.B_Y).getValue());

        // Si c'est la position de départ
        if (begin != null)
            bng = new Coordonnees(x, y, z + 1);

        try
        {
            Class<?> c = Class.forName(PACKAGE_BLOCK + t);
            if(t.equals(CLASS_TP))
            {
                // Si c'est un téléporteur (données supplémentaires)
                String d = block.getAttribute(BeaconXML.B_TP_DEST).getValue();
                Constructor<?> constructor =
                        c.getConstructor(Coordonnees.class, TeleporterBlock.class);
                Constructor<?> constructor2 = c.getConstructor(Coordonnees.class);
                int xx = Integer.valueOf(d.split(",")[0]),
                        yy = Integer.valueOf(d.split(",")[1]),
                        zz = Integer.valueOf(d.split(",")[2]);
                Block tp = (Block) constructor.newInstance(new Coordonnees(x, y, z),
                        constructor2.newInstance(new Coordonnees(xx, yy, zz)));
                lb.add(tp);
                lb.add(((TeleporterBlock) tp).getDest());
                ((TeleporterBlock) tp).lier();
            }
            else
            {
                // Sinon : bloc normal
                Constructor<?> constructor = c.getConstructor(Coordonnees.class);
                lb.add((Block) constructor.newInstance(new Coordonnees(x, y, z)));
            }
        }
        catch (SecurityException | IllegalArgumentException
                | NoSuchMethodException | InstantiationException | IllegalAccessException
                | InvocationTargetException e1)
        {
            System.out.println("Invalid XML format :\n\t" + e1.toString());
            return;
        }
```

Note : En cas de « ClassNotFoundException », c'est que le block est un coffre, il est donc créé par la suite.

World, vérification de la fin d'un niveau (et méthodes utilisées) :

```
/**
 * Recupere la liste des blocks d'un type</br>
 * Exemple d'utilisation : getBlocksT(Block.class)
 * @param <T> Le type de block à récupérer
 * @param blockType Le type de blocs a recuperer
 * @return La liste des blocks du type blockType
 */
@SuppressWarnings("unchecked") // TypeSafe : Check c'est un block de type T puis cast en T
public <T> List<T> getBlocksByType(Class<? extends Block> blockType)
{
    List<T> l = new ArrayList<T>();

    for(Block b : blockList)
        if(b.getClass().equals(blockType))
            l.add((T) b);

    return l;
}

/**
 * Verifie si un monde est gagné (tous les blocs lumiere allumes)
 */
public boolean isComplete()
{
    List<LightBlock> llb = this.getBlocksByType(LightBlock.class);
    boolean win = true;

    for(LightBlock lb : llb)
        win = win && lb.getLight();

    return win;
}
```

World, gestion des clones :

```
/**
 * Remplit la liste des coordonnee des clones
 */
public void setClone(Coordonnees xyz, int o)
{cloneList.add(new Coordation(xyz, o));}

/**
 * Gere la creation des clones
 */
public void popClone()
{
    for (int i=0; i < cloneList.size(); i++)
        if (isValidPosition(cloneList.get(i).getCoord()))
        {
            Character p = new Character(new Coordonnees(cloneList.get(i).getCoord()),
                    cloneList.get(i).getOrientation());
            p.setPosSprite(placeMe(p.getCoord()));

            addCharacter(p);
            p.setMain();

            cloneList.remove(i);
            i--;
        }
}
```

Interpreteur, gestion du « For » :

```
//Si l'action recupere est un For.
if (act instanceof For)
{
    For actFor = (For) act;
    //On decremente de 1 sa valeur
    actFor.decrementer();

    //Si il n'y a rien apres, on ignore le For et on sort de la procedure.
    if (!it.hasNext())
        return eval(p);
    //Sinon on regarde l'action apres le For.
    act = it.next();

    //Si elle n'est pas de la bonne couleur, on la passe et on ignore le for aussi.
    if ( !(act.getColor() == Color.DEFAUT || act.getColor() == p.getColor()) )
    {
        if (it.hasNext())
            pile.push(it);
        return eval(p);
    }

    //Si il reste des tours de for a faire, on replace l'iterateur sur le for pour la prochaine evaluation.
    if (!actFor.isZero())
    {
        it.previous();
        it.previous();
    }

    //Si l'action recupere est un Break, on sort de la procedure mais on ne fait rien. Le Break est considerer comme une action.
    if (act instanceof Break)
    {
        p.incrementNbActions();
        return null;
    }

    //Si il reste des actions dans la procedure ou l'on se trouve, on remet l'iteratorr dans la pile.
    if (it.hasNext())
        pile.push(it);

    //Si l'action recuperer est une procedure, on evalue la premiere action de cette procedure.
    if (act instanceof Procedure)
    {
        List<Prog> l = ((Procedure) act).getListProcedure();
        //Creation de l'iterator pour cette nouvelle procedure.
        ListIterator <Prog> it2 = l.listIterator();
        //On empile au sommet de la pile l'iterator pour la nouvelle procedure et on evalue la premiere action de celle-ci.
        pile.push(it2);
        return eval(p);
    }

    p.incrementNbActions();
    return (Action) act;
}
```