Projet de Programmation Orientée Objet : Le Lightbot

Phase de Conception

Lilian Gallon & Tristan Renaudon Groupe B

Rapport de conception : Lightbot

I. Actions élémentaires

Voici la liste des actions élémentaires que peut faire le joueur/programmateur:

Le joueur/programmateur peut dans la partie menu (page principale et sélection niveau):

- Voir les crédits, c'est-à-dire voir la licence et le nom des professeurs responsables,
- Voir la liste des niveaux, si ceux-ci sont complétés,
- Créer un nouveaux niveau en passant en mode édition,
- Jouer un niveau sélectionné dans la liste des niveaux,
- Quitter le programme.

Le joueur/programmateur peut dans la partie édition:

- Déplacer les actions (avancer, changer l'orientation du robot dans le sens antihoraire, changer l'orientation du robot dans le sens horaire, saut, lumière) dans programme principal, P1, P2 en glisser/déposer ou par clic sur l'action après avoir cliqué sur la fenêtre choisie,
- Déplacer les actions (P1, P2) dans le programme principal en glisser/déposer ou par clic sur l'action après avoir cliqué sur la fenêtre choisie,
- Exécuter le programme principal,
- Supprimer une action,
- Réinitialiser l'ensemble des actions,
- Revenir à la liste des programmes,
- Quitter le programme (sauvegarde le programme).

Le joueur/programmateur peut dans la partie jeu:

- Déplacer les actions (avancer, changer l'orientation du robot dans le sens antihoraire, changer l'orientation du robot dans le sens horaire , saut, lumière) dans le programme principal, P1, P2 en glisser/déposer ou par clic sur l'action après avoir cliqué sur la fenêtre choisie,
- Déplacer les actions (P1, P2) dans programme principal en glisser/déposer ou par clic sur l'action après avoir cliqué sur la fenêtre choisie,
- Exécuter le programme principal,
- Choisir entre retourner au menu, passer au niveau suivant ou recommencer le niveau à la réussite du niveau en cours,
- Supprimer une action,
- Réinitialiser l'ensemble des programmes,
- Revenir à la liste des niveaux,
- Quitter le programme.

II. Contraintes des actions

Dans le programme principal, le joueur/programmateur peut effectuer 15 actions maximum.

Dans le PROC1, le joueur/programmateur peut effectuer 10 actions maximum.

Dans le PROC2, le joueur/programmateur peut effectuer 10 actions maximum.

III. Aspect visuel

Les hexagones ont deux états possibles; allumé (de couleur jaune si hauteur de 0 sinon jaune foncé), et éteint (de couleur gris si hauteur de 0 sinon gris foncé), le joueur/programmateur peut donc identifier les cases éteintes et tenter de les allumer pour compléter le niveau, tout en prenant compte de la hauteur de celles-ci. L'orientation du robot est représentée par un trait, et le robot par un rond tandis que le niveau est vu de dessus.

IV. Fonctions de contrôle du jeu

Le joueur/programmateur peut effectuer plusieurs actions pour contrôler le jeu, en voici la liste :

- Reset : Permet de réinitialiser les programmes (P1, P2 et Main).
- Retour : Permet de revenir à la liste des niveaux
- Quit : Quitte le jeu
- Play : Exécute l'ensemble de la séquence des actions présentes dans le programme principal, ainsi que dans PROC1 et PROC2 si ceux-ci sont appelés. Le robot se déplace en fonction de cette séquence et effectue les actions nécessaires (allumer la lumière, sauter, se tourner, avancer).

V. Jouabilité

1) Le score du joueur

Le score du joueur est calculé en fonction du nombre d'actions qu'il effectue afin de compléter le niveau. En conséquence, plus il est faible, meilleur est le score. Par ailleurs, les actions effectués dans le PROC1 ou le PROC2 comptent moins que dans le MAIN pour inciter le joueur/programmateur à diviser au maximum son problème en plusieurs sous problèmes à travers les sous programmes.

2) Les niveaux

On souhaite implémenter 5 niveaux dont 4 niveaux “tutoriel”:

1. Apprend l’utilisateur à avancer et allumer une case.
2. Apprend l’utilisateur à tourner.
3. Apprend l’utilisateur à sauter.
4. Apprend l’utilisateur à utiliser les PROC
5. Niveau utilisant l’ensemble des capacités utilisées dans les quatres niveaux précédents.

VI. Mode édition

Le joueur/programmateur enregistre une séquence dans le programme principal (PROC1 et PROC2 compris s’il le souhaite), puis à l'exécution, le robot effectue la séquence, puis le niveau est enregistré en fonction des cases qu’il a parcouru, et on passe à la liste des niveaux.

Le robot se déplace sur un niveau remplit de cases éteintes.

VII. Enregistrement / Chargement des terrains

On enregistre l’ensemble des cellules du niveau au fur et à mesure du déplacement du robot dans un fichier csv. Les cases contiennent un certain nombre d’informations comme la hauteur, si la case est allumé et la position.

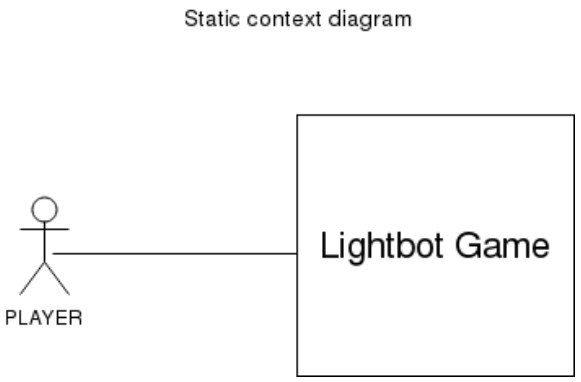
On charge un niveau à l’aide d’un fichier csv du type :

id;nom;posXRobot;posYRobot;{posXCase,posYCase,hauteur,lumiere};

Avec {...} contenant les informations d’une case. il peut y en avoir 7*7=42 maximum. Etant donné que la taille maxe d’un niveau est de 7*7.

VIII. Diagrammes

1) Diagramme de contexte statique



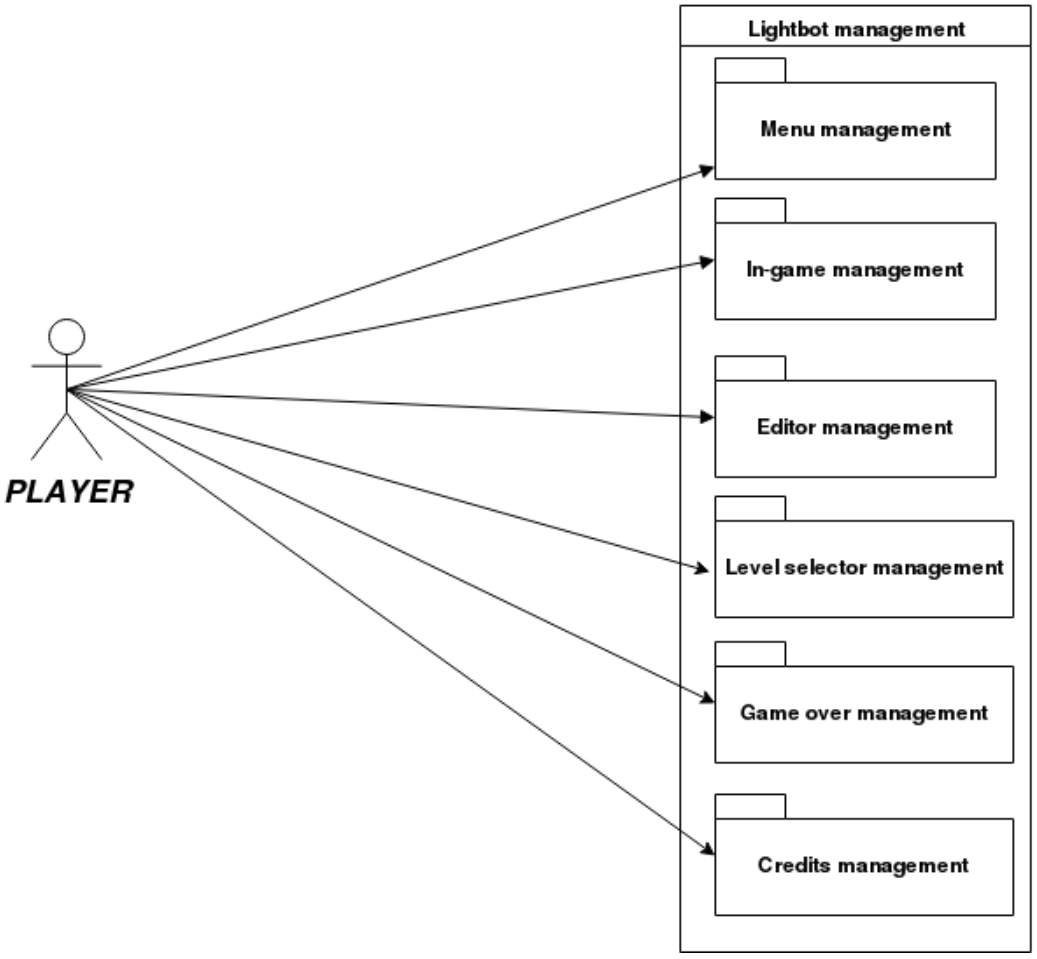
2) Liste des événements externes et résultats

Evenements		
Extern	Intern	
ARR_addActionMain	ENV_updateMain	In-game management
ARR_addActionP1	ENV_updateP1	
ARR_addActionP2	ENV_updateP2	
ARR_resetProg		
ARR_deleteAction		
ARR_runProg	ENV_programResult	
ARR_backToMenu	ENV_updateGameStatus	
ARR_exit	ENV_exitProgram	
ARR_addActionMain	ENV_updateMain	Editor management
ARR_addActionP1	ENV_updateP1	
ARR_addActionP2	ENV_updateP2	
ARR_resetProg		
ARR_deleteAction		
ARR_runProg	ENV_programResult	
	ENV_createLevel	
ARR_backToMenu	ENV_updateGameStatus	
ARR_exit	ENV_exitProgram	

Evenements		
Extern	Intern	
ARR_goMenu	ENV_changeGameStatus	Credits management
ARR_Exit	ENV_exitProgram	
ARR_resetLevel	ENV_resetProgram	Game over management
ARR_goLevelSelector	ENV_changeGameStatus	
ARR_Exit	ENV_exitProgram	
ARR_goMenu	ENV_changeGameStatus	Level selector management
ARR_goEdit		
ARR_goGame		
ARR_exit	ENV_exitProgram	
ARR_goCredits	ENV_changeGameStatus	Menu management
ARR_goLevelSelector		
ARR_exit	ENV_exitProgram	

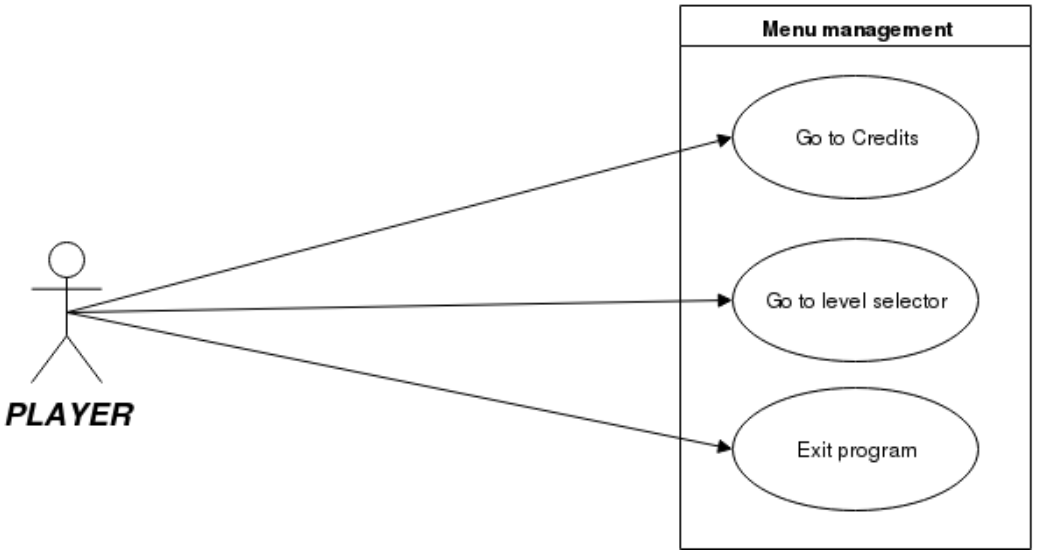
3) Diagramme des cas d'utilisation

Use case diagram : Lightbot management



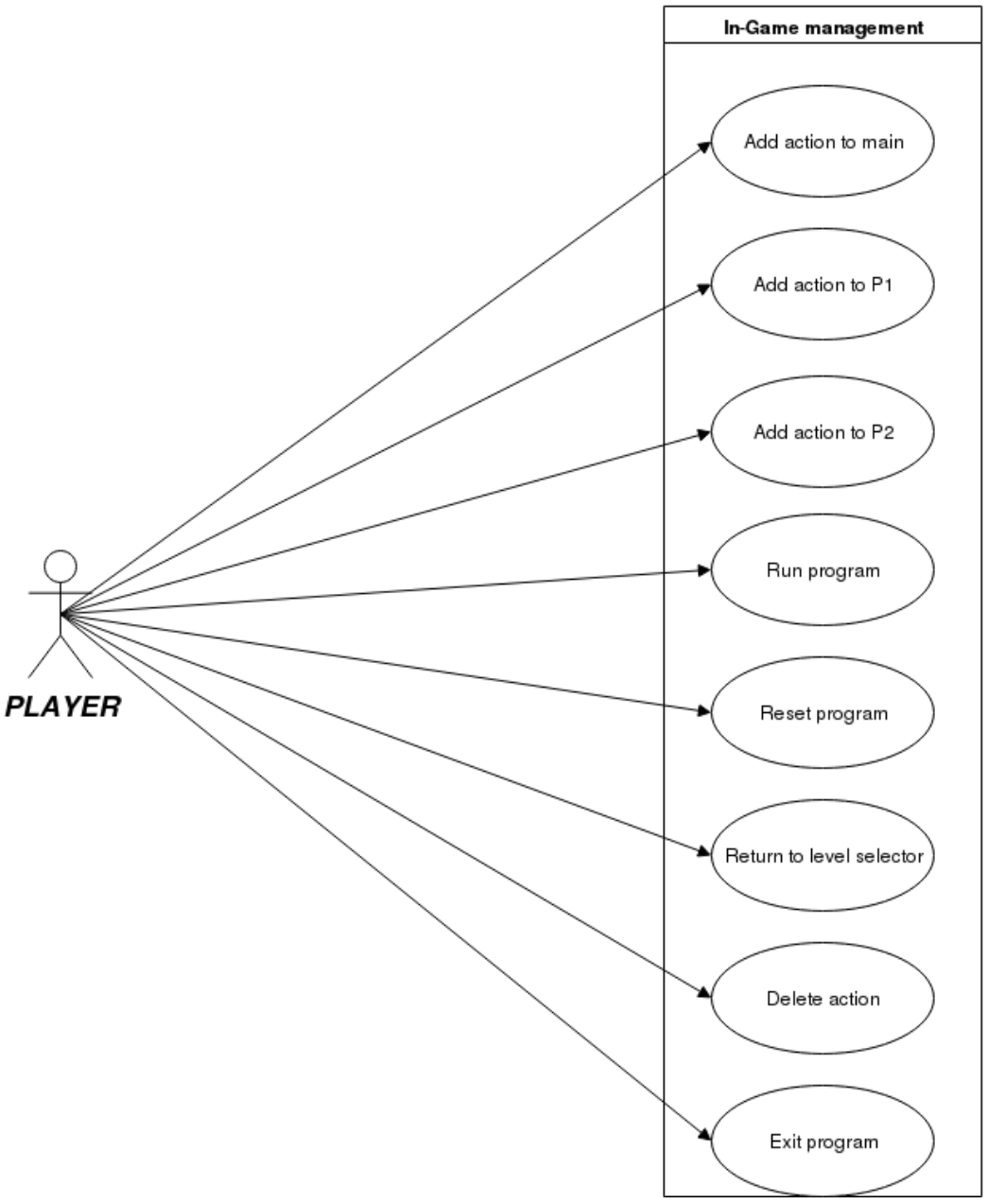
Voici l'ensemble des actions que peut réaliser le joueur/programmateur dans le programme

Use case diagram : Menu management



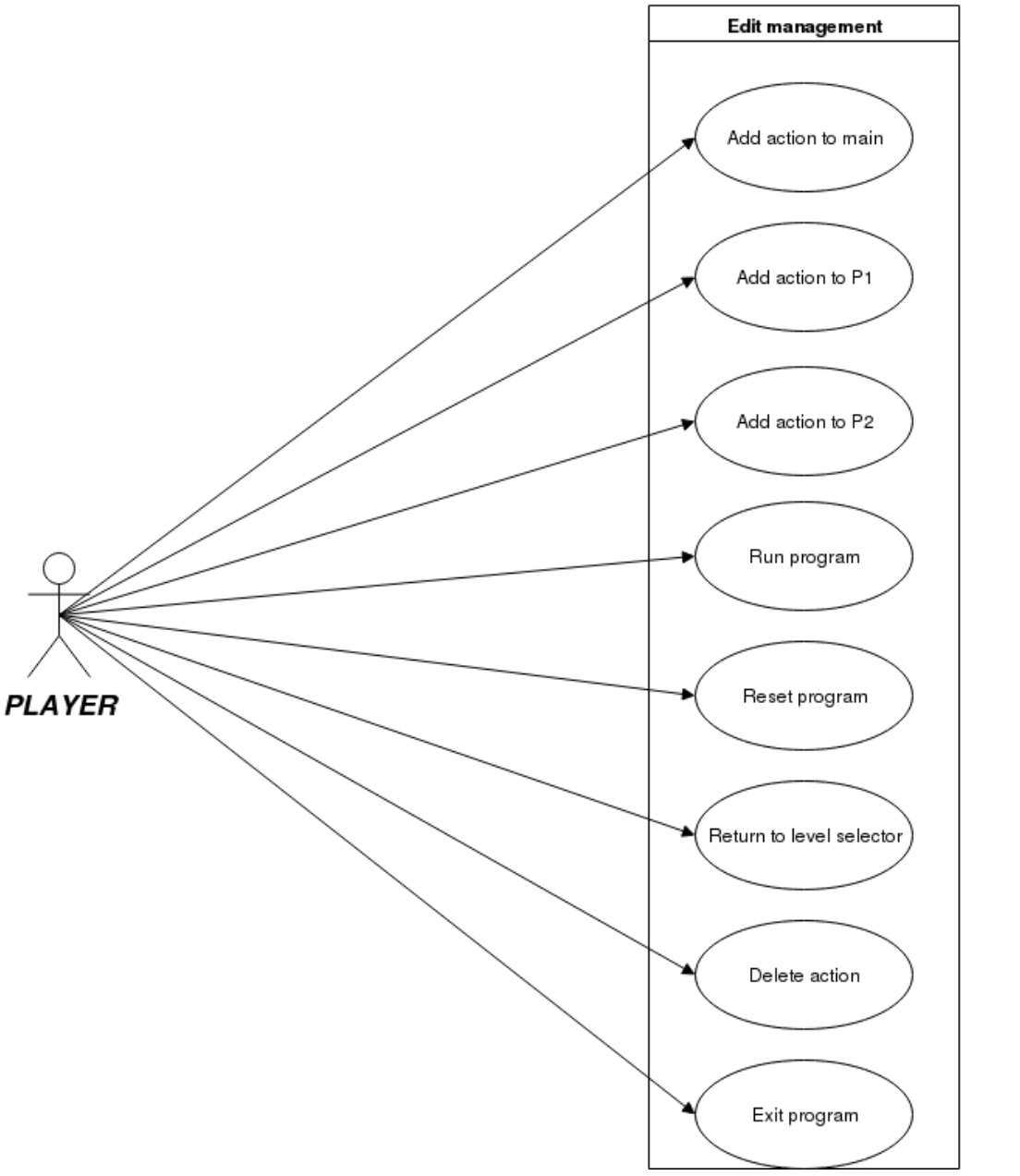
Voici l'ensemble des actions que peut réaliser le joueur/programmateur dans le menu

Use case diagram : In-Game management



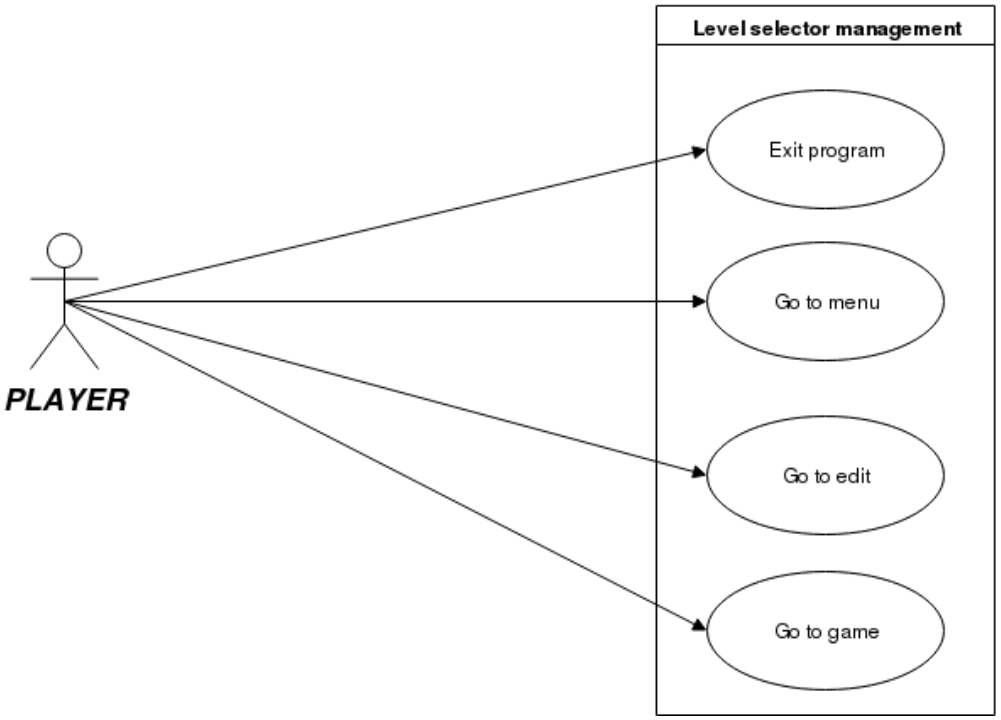
Voici l'ensemble des actions que peut réaliser le joueur/programmateur dans le jeu

Use case diagram : Edit management



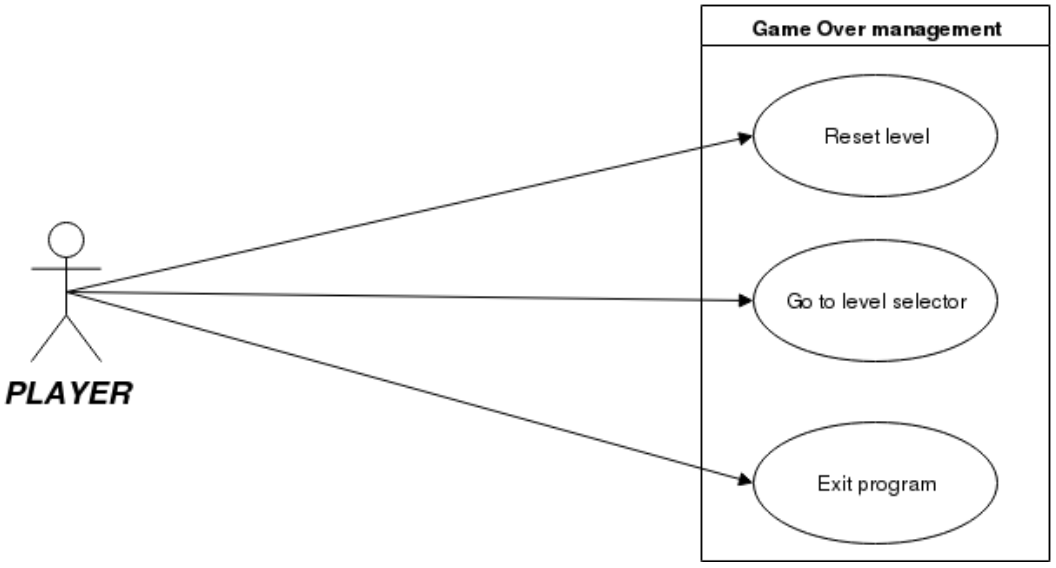
Voici l'ensemble des actions que peut réaliser le joueur/programmateur dans l'editeur

Use case diagram : Level selector management



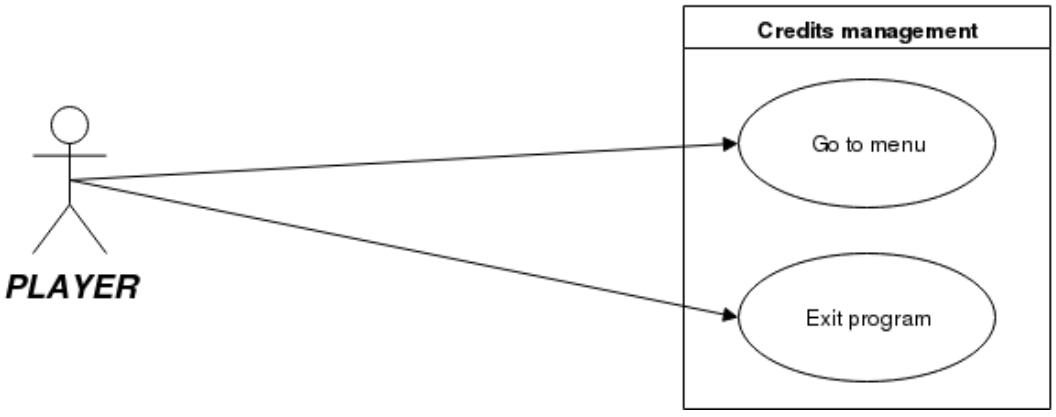
Voici l'ensemble des actions que peut réaliser le joueur/programmateur dans la liste des niveaux

Use case diagram : Game Over management



Voici l'ensemble des actions que peut réaliser le joueur/programmateur lors de la fin du jeu

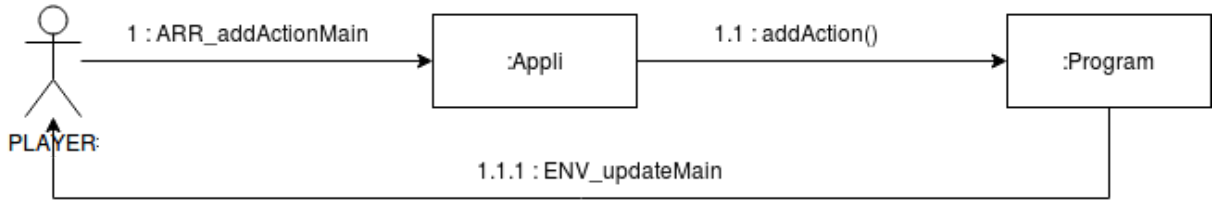
Use case diagram : Credits management



Voici l'ensemble des actions que peut réaliser le joueur/programmateur lors de la visualisation des crédits.

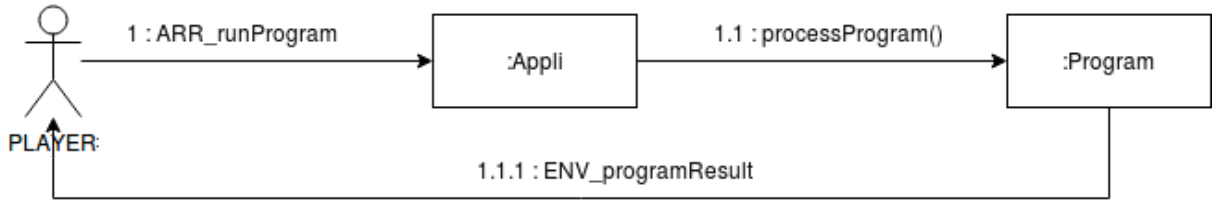
4) Diagrammes de communication

Communication diagram - Main



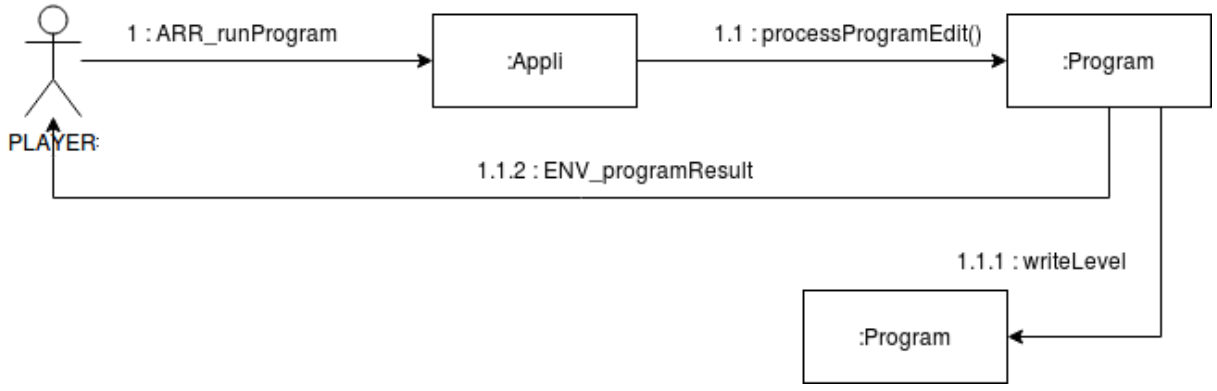
On voit ici que le glisser/déposer n'est pas géré par la classe Program directement mais bien par la classe Appli. La partie graphique de l'ajout d'une action se fait dans Appli.

Communication diagram - Play



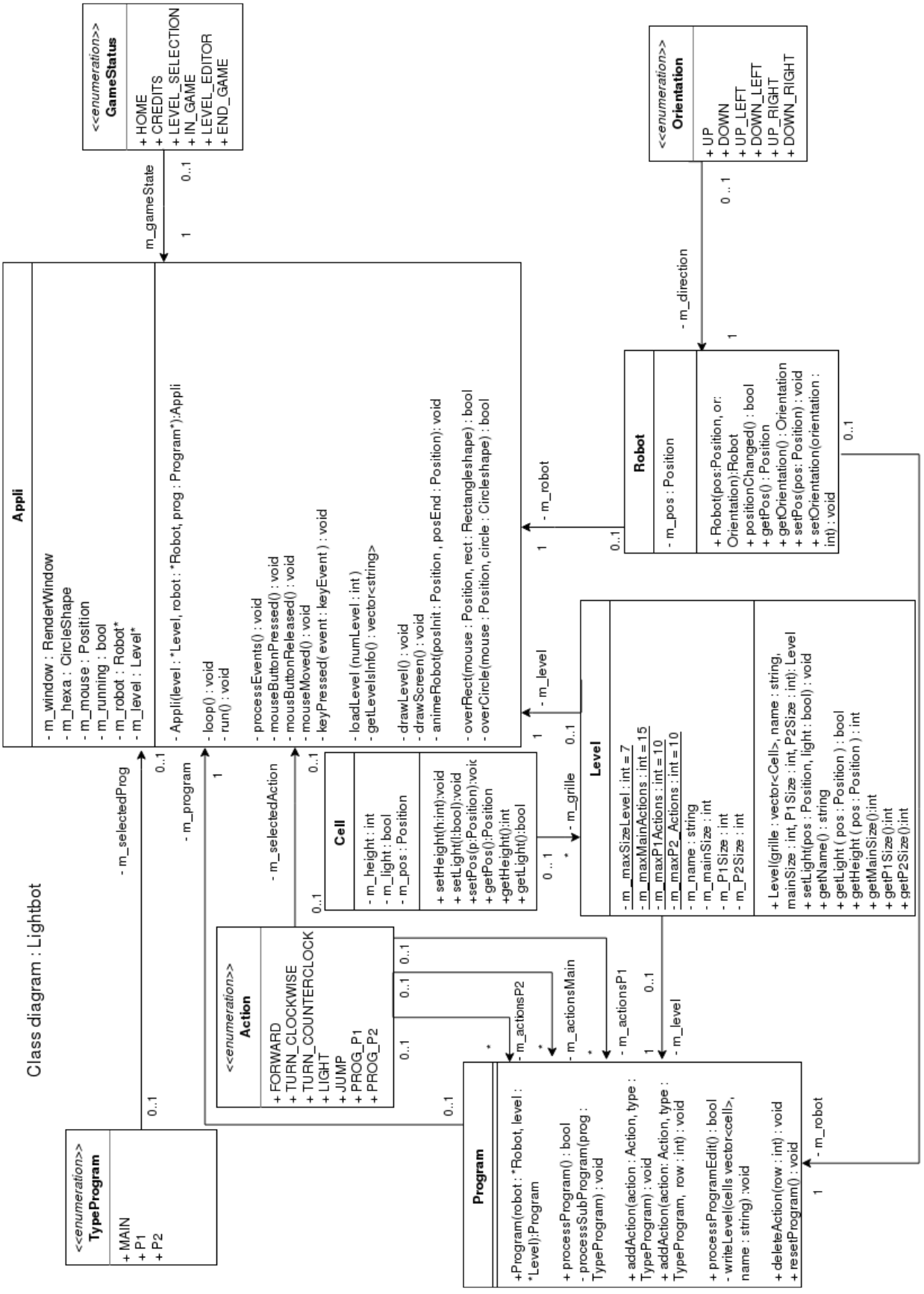
Ici c'est la classe Appli qui demande l'exécution du programme et non l'utilisateur directement

Communication diagram - Edit



La classe Program génère un nouveau programme a la fin de l'exécution de la séquence d'action.

5) Diagramme de classe



IX. Dictionnaire de données

1) Classe Appli

Pour afficher le jeu, nous utilisons une variable RenderWindow *(m_window)* qui sera notre fenêtre de jeu. De plus, pour afficher les hexagones, on utilise un cercle à 6 côtés *(m_hexa)*.

Il nous faut récupérer l'interaction du joueur. Pour ceci, nous utilisons une variable de type Position qui contient la position de la souris *(m_mouse)*.

Il y a un booléen *(m_running)* qui détermine s'il faut continuer la boucle de jeu *(run()).*

La classe Appli instancie un Robot et un Level (lors du chargement du niveau) et pointe vers eux. L'intérêt pour Appli de pointer sur un Robot *(m_robot)* et un Level *(m_level)* est d'afficher par la suite le Level, et le Robot, et de l'animer en récupérant ses positions.

De plus, la classe Appli pointe le programme *(m_program)* afin de :

- demander sa réinitialisation *(resetProgram())*
- ajouter une action *(addAction(..))*
- démarrer le programme *(processProgram())*
- démarrer le programme en mode édition *(processProgramEdit())*

L'instanciation du programme sera fait lors du passage à l'état jeu : "IN_GAME"

Nous avons décidé qu'en plus du glisser-déposer, il est possible d'ajouter une action en appuyant sur un programme, ce qui le sélectionne *(m_selectedProg)* puis sur une action ce qui la sélectionne *(m_selectedAction)* et l'ajoute au programme précédemment sélectionné. Par défaut, le programme sélectionné est "Main".

Pour les événements souris, trois méthodes sont utilisées, *mouseButtonPressed()*, *mouseButtonReleased()* et *mouseMoved()*.

Concernant les événements du clavier, une méthode est utilisée, keyPressed(..) qui capture la touche pressée par le joueur. Pour le moment la seule touche qui est utile est echap, qui quitte le jeu.

Dans la boucle de jeu, *loop()* s'occupera plutôt de l'affichage, alors que *processEvents()* s'occupera des événements souris et clavier. Ce comportement changera en fonction de l'état du jeu déterminé par *m_gameState.*

drawLevel() est appelé pour l'affichage du niveau seulement, alors que *drawScreen()*, affiche le reste, en fonction de l'état du jeu.

Pour l'interaction du joueur avec les boutons, il y a deux méthodes, *overRect(..)* et *overCircle(..)* qui renvoient true si la position entrée en paramètre est sur un rectangle ou sur un cercle, respectivement. A noter, que la méthode overCircle prend en compte le nombre de côtés du cercle.

La méthode *getLevelsInfo()* permet de récupérer les noms des niveaux. En effet, cette méthode est utile pour l'écran de sélection de niveau. Elle renvoie un conteneur de type vector, contenant des chaînes de caractère. Ces chaînes correspondent au nom des niveaux, qui seront affichés dans l'écran de sélection.

La méthode *loadLevel()* charge le level définit selon son id entré en paramètre. Pour le charger, il appel son constructeur. L'instanciation du level est faite ici.

La méthode *removeLevel()* permet de supprimer un niveau si l'utilisateur en a décidé ainsi. Il n'est possible de supprimer que des niveaux édités.

La méthode *animeRobot(...)* permet l'animation du robot à partir de sa position initiale et sa position finale. Exemple : Le programme s'exécute est a les positions initiales du robot. Il change ses positions (temporaires) au fur et à mesure qu'il parcourt les actions. Dès qu'il y a un changement de direction, il envoie une demande d'animation par l'intermédiaire de positionChanged, qu'il passe à true. Donc dès que Appli détecte que positionChanged est passé à true, il appelle *animeRobot()* avec les positions du robot initiales et finales. Une fois l'animation finie, le programme continue son exécution.

Si le joueur est en mode édition, cela affiche un niveau sans cases, donc vide.

Enfin le constructeur initialise les éléments graphiques

2) Classe Robot

Le robot a une variable *m_pos* qui contient sa position et *m_direction* qui contient sa direction. Il y a un getteur sur ces deux attributs. Respectivement *getPos()* et *getOrientation()*. Il est possible de changer sa position et son orientation par les setteurs *setPos()* et *setOrientation()*.

La méthode PositionChanged est utile à l'animation comme indiqué précedemment.

Le constructeur de la classe robot initialise sa position et son orientation.

3) Classe Level

Le level a un nom *(m_name)*, une taille max *(m_maxSizeLevel)*, un nombre max d'actions dans main, P1 et P2 (*m_maxMainAction*, *m_maxP1Actions*, *m_maxP2Actions)* . La taille max permet de limiter le nombre de cases en largeur et hauteur à l'écran pour éviter de dépasser l'affichage prévu pour le niveau. Le nom est utile pour la sélection du niveau. De plus celui-ci sera affiché au-dessus du niveau en jeu. Tandis que le nombre max d'actions permet de limiter le nombre d'actions à mettre dans chaque programme (main, p1 et p2) afin d'augmenter la difficulté.

La méthode *getName()* permet de récupérer le nom du niveau courant afin de l'afficher en haut de l'écran en jeu.

La méthode *getLight()* permet de savoir si la case dont la position est entrée en paramètre est allumée.

La méthode *getHeight()* permet de connaître la hauteur de la case dont la position est entrée en paramètre.

La méthode *setLight()* permet de changer la lumière de la case dont la position est entrée en paramètre. C'est utile pour le programme lorsqu'il parcourt l'action qui consiste à allumer la case.

En mode editeur, le niveau est chargé sans cases et avec le nombre d'actions maximum paramétré au maximum.

Le constructeur de Level prend un vecteur de Cell, qui sont les cases, et le nombre max d'actions du main, de p1 et de p2.

4) Classe Cell

La Classe Cell a trois attributs, sa hauteur *(m_height)*, sa lumière *(m_light)*, sa position *(m_pos)*. Il y a des setteurs et accesseurs sur chacun de ces attributs.

5) Classe Program

Les programmes main, p1 et p2 sont représentés par un conteneur (array) d'actions et de taille définie par le niveau (*m_actionsMain, m_actionsP1, m_actionsP2*).

La méthode *addAction()* est surchargée. En effet, il est possible d'ajouter une action à la fin, ou à un indice bien précis (glisser-déposer) entre deux actions. Cette action vérifie s'il y a de la place dans le conteneur par ajouter une action et que l'action est valide (ex: interdit de mettre P1 dans P1)

Pour supprimer une action spécifique, *deleteAction()* est appelée avec en paramètre le rang de l'action à supprimer. Pour supprimer toutes les actions, alors la méthode *resetProgram()* est appelée, ce qui reset les 3 programmes. Le principe est simple, on applique *deleteAction()* à tous les rangs des tableaux contenant les actions.

Lorsque le programme est exécuté, la méthode *processProgram()* est appelée. Elle parcourt le tableau *m_actionsMain*. Dès que la méthode détecte une action P1 ou P2, elle appelle *processSubProgram()*, qui parcourt PX, où X est le programme détecté précédemment. L'appel à cette méthode permet au joueur de faire de la récursivité. Or, si le programme fait une boucle sans fin, il faut le savoir. Pour cela il y a une variable compteur, qui compte le nombre de fois que l'appel à *processSubProgram()* est fait. Une fois que cette variable dépasse un numéro max, l'appelle à *processSubProgram()* est interdit et *processSubProgram()* renvoie faux. En effet, *processSubProgram()* renvoie un booléen indiquant si le programme a résolu le niveau.

La méthode *processProgramEdit()* est appelée lorsque le joueur est en mode éditeur. Vu que le niveau est édité en fonction des actions du joueur, la méthode vérifie que le programme n'a pas de récursivité infinie, et ne dépasse pas la taille max. Les cases seront définies par les cases parcourues par le robot. Le nombre d'actions réalisées défini le nombre d'actions max du niveau et le niveau est nommé "E-x", où x est le numéro id du niveau (exemple: s'il y a 5 niveaux, alors ce sera 6). Pour l'écriture de ce niveau, *writeLevel(...)* s'en charge.

Le constructeur de Program prend un pointeur du robot (*m_robot*) et du level (*m_level*) qui ont été instanciés dans Appli auparavant.

6) Les différentes classes énumération

Les classes d'énumération sont en fait des classes statiques ayant des attributs constants publics.

a) TypeProgram

TypeProgram permet de savoir sur quel programme on travaille, pour ensuite ajouter des actions, ou les supprimer par exemple.

b) Action

Action contient toutes les actions élémentaires que peut contenir un programme.

c) GameStatus

GameStatus permet à Appli de changer son comportement en fonction de l'état du jeu.

d) Orientation

L'orientation correspond à l'orientation du Robot. Cette orientation sera changée avec l'action TURN_CLOCKWISE (tourner dans le sens horaire) ou TURN_COUNTERCLOCK (tourner dans le sens anti horaire).

X. Signature des méthodes

1) Classe Program

```
public Program(Robot* robot, Level* level);
public bool processProgram();
private void processSubProgram(TypeProgram prog);
public void addAction(Action action, TypeProgram type);
public void addAction(Action action, TypeProgram type, int row);
public bool processProgramEdit();
private void writeLevel(cells vector<cell>; string name);
void deleteAction(int row);
void resetProgram();
```

2) Classe Level

```
public Level(vector<Cell> grille, string name ,int mainSize, int P1Size, int P2Size);
public void setLight(Position pos, bool light);
public string getName();
public bool getLight ( Position pos );
public int getHeight ( Position pos );
public int getMainSize();
public int getP1Size();
public int getP2Size();
```

3) Classe Appli

```
public Appli(Level* level, Robot* robot, Program* prog);
private void loop();
private void run();
private void processEvents();
private void mouseButtonPressed();
private void mousButtonReleased();
private void mouseMoved();
private void keyPressed( event : keyEvent );
private void loadLevel (int numLevel);
private vector<string> getLevelsInfo();
private void drawLevel();
private void drawScreen();
private void animeRobot(Position posInit, Position posEnd)
private bool overRect(Position mouse, Rectangleshape rect);
private bool overCircle(Position mouse, Circleshape circle);
```

4) Classe Robot

```
public Robot(Position pos,Orientation or)
public bool positionChanged()
public Position getPos();
public Orientation getOrientation();
public void setPos(Position pos);
public void setOrientation(int orientation);
```

5) Énumérations

Classe Robot:

- enum class Orientation {UP, DOWN, UP_LEFT, DOWN_LEFT, UP_RIGHT, DOWN_RIGHT } m_direction;

Classe Appli:

- enum class GameStatus {HOME,CREDITS,LEVEL_SELECTION,IN_GAME,LEVEL_EDITOR,END_GAME} m_gameState;
- enum class TypeProgram {MAIN, P1, P2} m_selectedProg;
- enum class Actions {FORWARD, TURN_CLOCKWISE, TURN_COUNTERCLOCK, LIGHT, JUMP, PROG_P1, PROG_P2} m_selectedAction;

XI. Avancement du projet

Tout le processus de conception a été très difficile. En ce qui concerne les lacunes, au niveau des énumérations par exemple, on ne sait pas s'il faut plutôt créer une classe statique, ou plutôt déclarer les énumérations comme fait ci-dessus. Un autre problème a été de savoir comment, avec le diagramme de classe on retranscrit le fait qu'on travaille sur le même objet (pointeur). C'est la première fois que l'on réalise un travail de conception d'une telle ampleur.