CONCEPTION TECHNIQUE

CRYPT OF THE JAVADANCER

OUT OF BOUNDS
IUT DIJON-AUXERRE

TABLE DES MATIERES



I - CONTEXTE

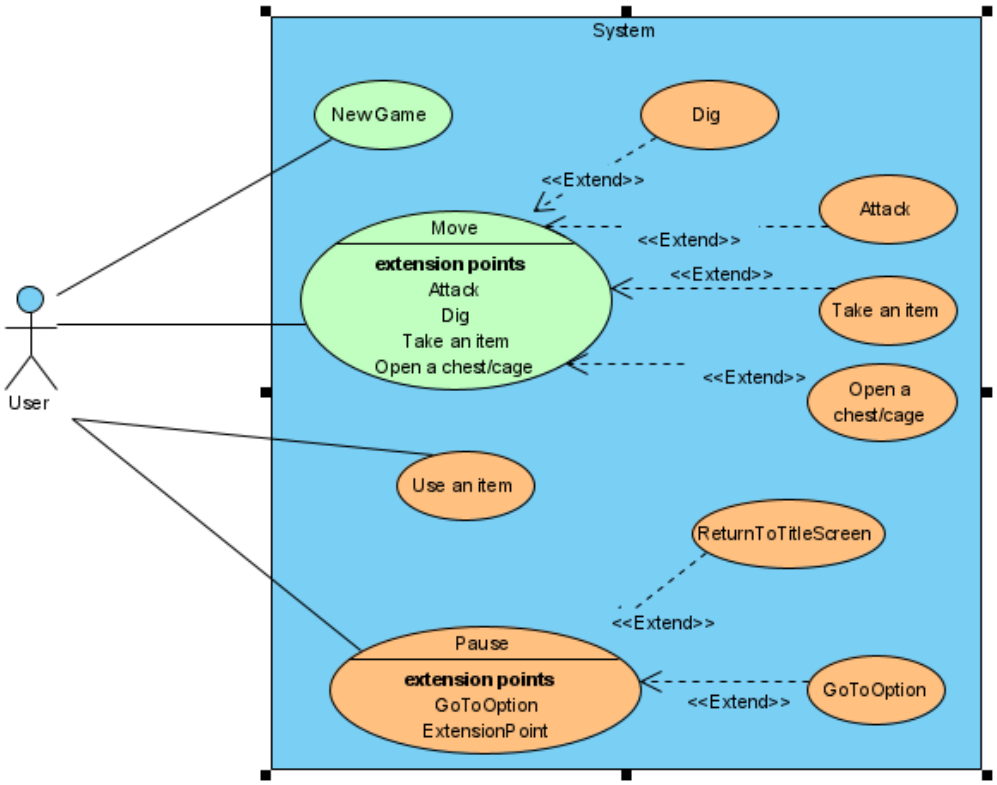
Notre projet consiste à reproduire le moteur du jeu de rythme indépendant *Crypt of the Necrodancer* (copycat). Nous devrons tout d'abord intégrer les fonctionnalités mères du jeu : l'écran titre, démarrer une nouvelle partie, générer une carte de manière procédurale, se déplacer et gagner.

Ensuite, le client attendra de nous une intégration des différentes fonctionnalités secondaires qui composent la base du gameplay de *Crypt of the Necrodancer* : intégration d'ennemis, points de vie, attaquer, creuser, ramasser et utiliser des objets, s'équiper d'armes et d'armures, ramasser de l'or et ouvrir des coffres. Il devra également être possible de mettre le jeu en pause pour quitter la partie ou ouvrir l'écran des options.

Pour effectuer ce dossier d'analyse nous avons suivi les demandes de notre client qui nous as demandé de faire un diagramme de cas d'utilisation ainsi que des diagrammes de séquence et un diagramme de classe seulement sur la partie la plus importante pour le bon déroulement d'une partie du jeu *Crypt of the NecroDancer*.

II - DIAGRAMME DE CAS D'UTILISATION

Concernant le diagramme de cas d'utilisation nous avons représenté les différentes actions que le joueur pourra effectuer dans le jeu. Nous avons représenté les fonctionnalités mères du moteur (en vert), à savoir « créer une nouvelle partie » et « se déplacer », mais également quelques fonctionnalités secondaires (en orange). Nous allons vous expliquer pourquoi.



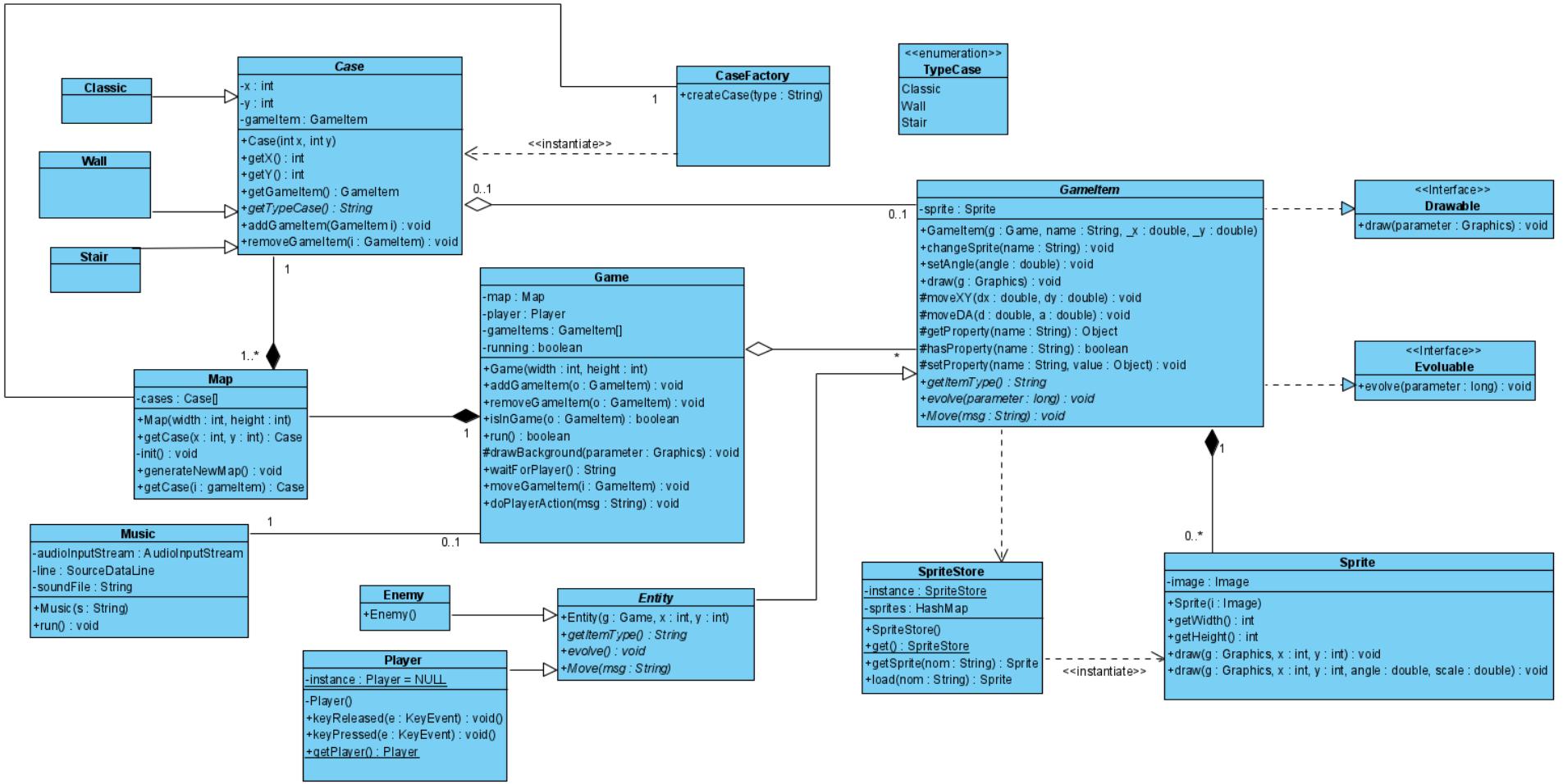
Comme vous pouvez le voir sur l'image ci-dessus, le joueur pourra lancer une nouvelle partie. Une fois le jeu lancé, le joueur pourra déplacer son personnage de case en case. C'est la seule action que le joueur pourra faire si on s'en tient à la partie vitale de notre *Backlog Product*. Cependant, nous avons trouvé pertinent de vous expliquer comment les autres actions exécutables par le joueur seront intégrées au moteur. Comme vous pouvez le constater avec le diagramme ci-dessus, les actions comme « Creuser », « Attaquer », « Ramasser un objet » ou « Ouvrir un coffre/une cage » sont des extensions de la fonction « Se déplacer ». En effet, le joueur n'aura pas à ordonner directement à son personnage d'attaquer par exemple. Il lui suffira de se déplacer sur un ennemi, le jeu s'occupera de faire attaquer le joueur si cela est possible.

Vous remarquerez également que nous avons séparer le déplacement de la fonction « Utiliser un objet ». Il nous a semblé important de mettre en évidence cette séparation.

Vous pouvez constatez qu'il y aura une fonction pause dans le jeu. Lorsque le joueur met le jeu en pause il pourra retourner à l'écran-titre et il pourra ouvrir l'écran des options. Ici aussi, vous pouvez constater que ces fonctionnalités sont des extensions de la fonction pause.

III - DIAGRAMMES DES CLASSES

Nous allons désormais vous présenter le diagramme de classe représentant les différentes classes ainsi que les fonctionnalités qui devront être présentes pour que le jeu fonctionne correctement :


Classic
Wall
Stair
Case
-x : int
-y : int
-gameItem : GameItem
+Case(int x, int y)
+getX() : int
+getY() : int
+getGameItem() : GameItem
+getTypeCase() : String
+addGameItem(GameItem i) : void
+removeGameItem(i : GameItem) : void
CaseFactory
+createCase(type : String)
<<instantiate>>
1
<<enumeration>>
TypeCase
Classic
Wall
Stair
0..1
0..1
GameItem
-sprite : Sprite
+GameItem(g : Game, name : String, _x : double, _y : double)
+changeSprite(name : String) : void
+setAngle(angle : double) : void
+draw(g : Graphics) : void
#moveXY(dx : double, dy : double) : void
#moveDA(d : double, a : double) : void
#getProperty(name : String) : Object
#hasProperty(name : String) : boolean
#setProperty(name : String, value : Object) : void
+getItemType() : String
+evolve(parameter : long) : void
+Move(msg : String) : void
<<Interface>>
Drawable
+draw(parameter : Graphics) : void
<<Interface>>
Evoluable
+evolve(parameter : long) : void
1
1..*
Map
-cases : Case[]
+Map(width : int, height : int)
+getCase(x : int, y : int) : Case
-init() : void
+generateNewMap() : void
+getCase(i : gameItem) : Case
1
Game
-map : Map
-player : Player
-gameItems : GameItem[]
-running : boolean
+Game(width : int, height : int)
+addGameItem(o : GameItem) : void
+removeGameItem(o : GameItem) : void
+isInGame(o : GameItem) : boolean
+run() : boolean
#drawBackground(parameter : Graphics) : void
+waitForPlayer() : String
+moveGameItem(i : GameItem) : void
+doPlayerAction(msg : String) : void
*
1
0..*
Music
-audioInputStream : AudioInputStream
-line : SourceDataLine
-soundFile : String
+Music(s : String)
+run() : void
1
0..1
SpriteStore
-instance : SpriteStore
-sprites : HashMap
+SpriteStore()
+get() : SpriteStore
+getSprite(nom : String) : Sprite
+load(nom : String) : Sprite
<<instantiate>>
Sprite
-image : Image
+Sprite(i : Image)
+getWidth() : int
+getHeight() : int
+draw(g : Graphics, x : int, y : int) : void
+draw(g : Graphics, x : int, y : int, angle : double, scale : double) : void
Enemy
+Enemy()
Entity
+Entity(g : Game, x : int, y : int)
+getItemType() : String
+evolve() : void
+Move(msg : String)
Player
-instance : Player = NULL
-Player()
+keyReleased(e : KeyEvent) : void()
+keyPressed(e : KeyEvent) : void()
+getPlayer() : Player

III.1. LA CLASSE GAMEITEM

GameItem est une classe *abstraite*. Elle représente un quelconque élément du jeu (sauf les murs). Un GameItem possède un Sprite, il peut changer ce sprite grâce à la fonction changeSprite() (nous verrons comment sont gérés les sprites plus tard). Les fonctions getItemType(), evolve() et move() sont des fonctions *abstraites* qui devront être redéfinies par les classes filles.

III.2. LA CLASSE ENTITY

Entity est une classe *abstraite* qui hérite de la classe GameItem. Elle va nous permettre de différencier les entités et les objets (une fois que ces derniers devront être intégrés au moteur).

III.3. LA CLASSE ENEMY

Enemy est une classe qui hérite de Entity et qui représente les ennemis que l'on rencontrera dans le jeu.

III.4. LA CLASSE PLAYER

Player hérite aussi de la classe Entity et permet la création du joueur. Player sera le lien entre l'utilisateur et le jeu. Ainsi, KeyReleased() et KeyPressed() serviront à récupérer les *inputs* de l'utilisateur afin que le joueur puisse agir en conséquence.

Il est important de noter que Player est un *Singleton* : une fois qu'une instance de Player est crée, elle reste la même indéfiniment. C'est la seule et unique instance que l'on récupérera avec getPlayer().

III.5. L'INTERFACE DRAWABLE

Cette interface servira à afficher les GameItem présents dans le jeu.

III.6. L'INTERFACE EVOLUABLE

Cette interface servira à mettre à jour l'états des GameItem présents dans le jeu.

III.7. L'ENUMERATION TYPECASE

Cette énumération représente les différents types de cases qui seront présentes dans le jeu.

III.8. LA CLASSE CASE

Case est une classe *abstraite* qui représente une case de la carte. Une case contient deux entiers x et y ainsi qu'un potentiel GameItem. Case possède une fonction *abstraite* getTypeCase(), chaque classe fille retournera son type.

III.9. LA FABRIQUE CASEFACTORY

La classe CaseFactory est une *fabrique* de Case qui permettra de créer une case en fonction d'un message passé en paramètre (« Classic », « Wall » ou « Stair »).

III.10. LA CLASSE MAP

Cette classe contiendra toute les cases du jeu et représentera la carte du jeu. Elle générera une carte de manière procédurale grâce à la fonction generateNewMap(). Map pourra également renvoyer une Case contenant tels coordonnés ou tel GameItem avec la fonction getCase().

III.11. LA CLASSE MUSIC

Music est la classe qui chargera les musiques du jeu et qui les jouera. Cette classe est importante car sans celle-ci le jeu n'aurait aucun intérêt puisqu'il s'agit d'un jeu ou l'on doit se déplacer au rythme de la musique. Elle contient un AudioInputStream afin de pouvoir jouer de la musique.

III.12. LA CLASSE GAME

C'est la classe principale du moteur, c'est Game qui va créer les éléments du jeu, cela inclut le joueur, la carte et les GameItem. De plus, la classe Game gérera le déroulement d'un tour, les actions des GameItem présents dans le jeu, et c'est elle qui vérifiera les conditions de victoire et de défaite.

III.13. LA CLASSE SPRITE

La classe Sprite permet de charger une image et de définir ses dimensions.

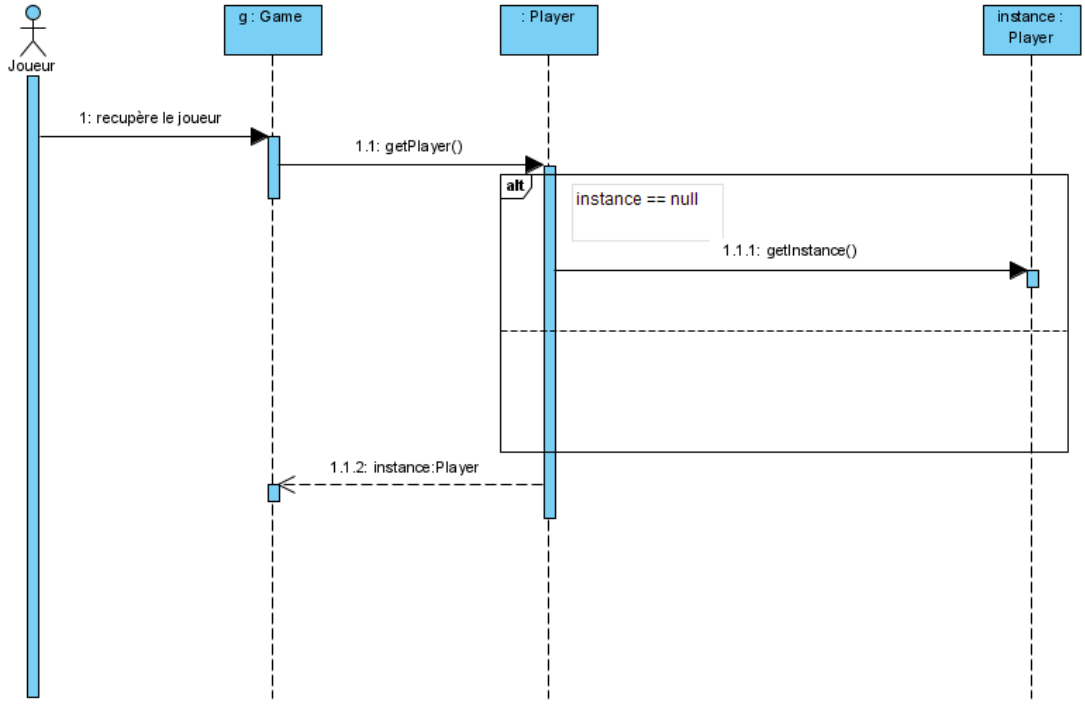
III.14. LA CLASSE SPRITESTORE

Cette classe permet de stocker les sprites et de les afficher grâce à la fonction load().

IV - DIAGRAMMES DE SEQUENCE

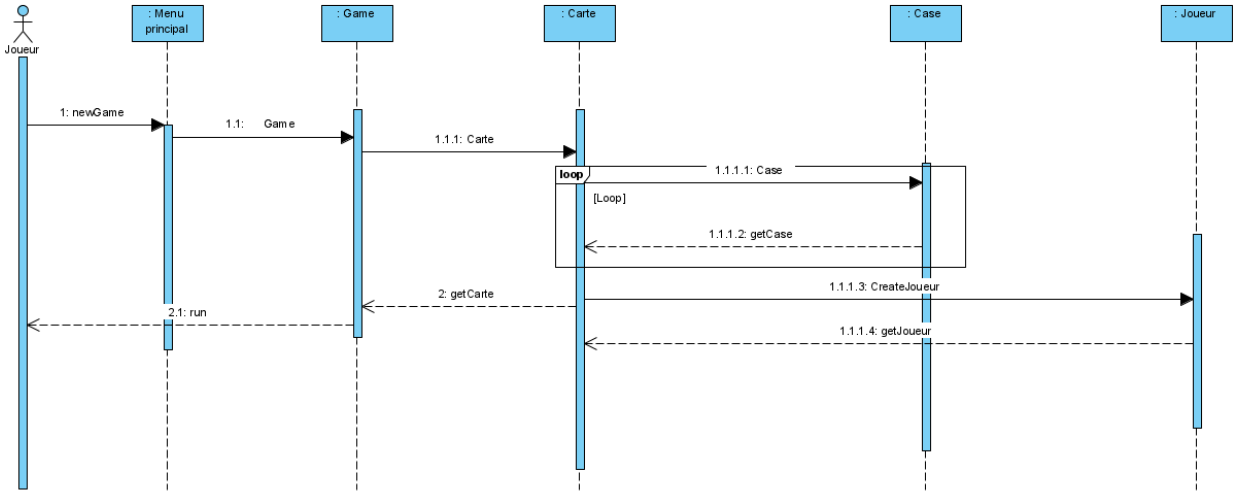
Grâce à ces diagrammes de séquence, nous allons pouvoir décrire plus en détail ce qu'il se passe lorsque que l'utilisateur veut utiliser telle ou telle fonctionnalité.

IV.1 - DIAGRAMME DE SEQUENCE DE LA FONCTION GETJOUEUR



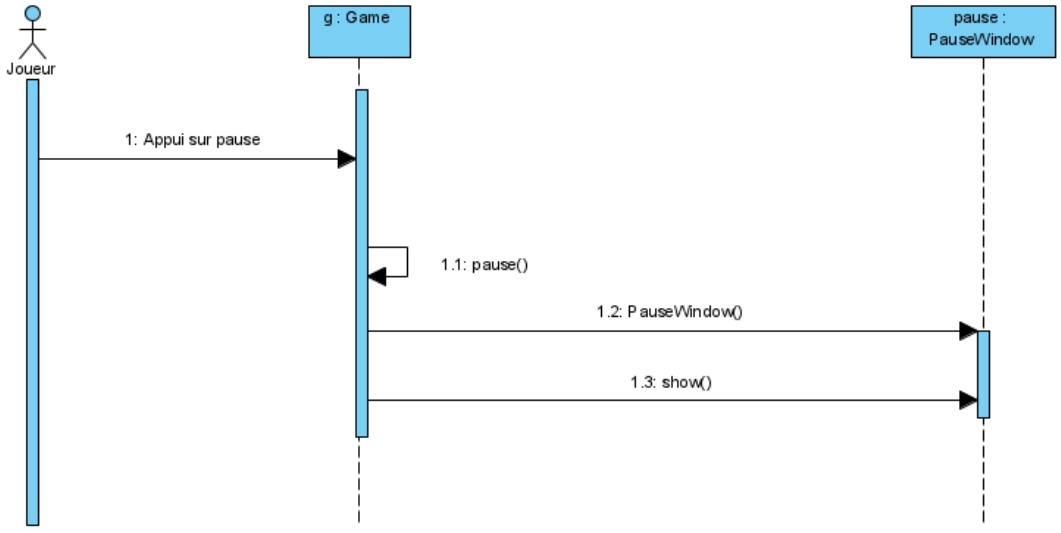
Player étant un *Singleton*, récupérer le joueur se passe d'une manière un peu particulière. En effet, lorsque le jeu demande à Player qu'on lui renvoie le joueur, Player va vérifier s'il possède une instance de joueur. Si ce n'est pas le cas, une instance est crée puis est envoyé au jeu.

IV.2 - DIAGRAMME DE SEQUENCE DE LA FONCTION NEWGAME



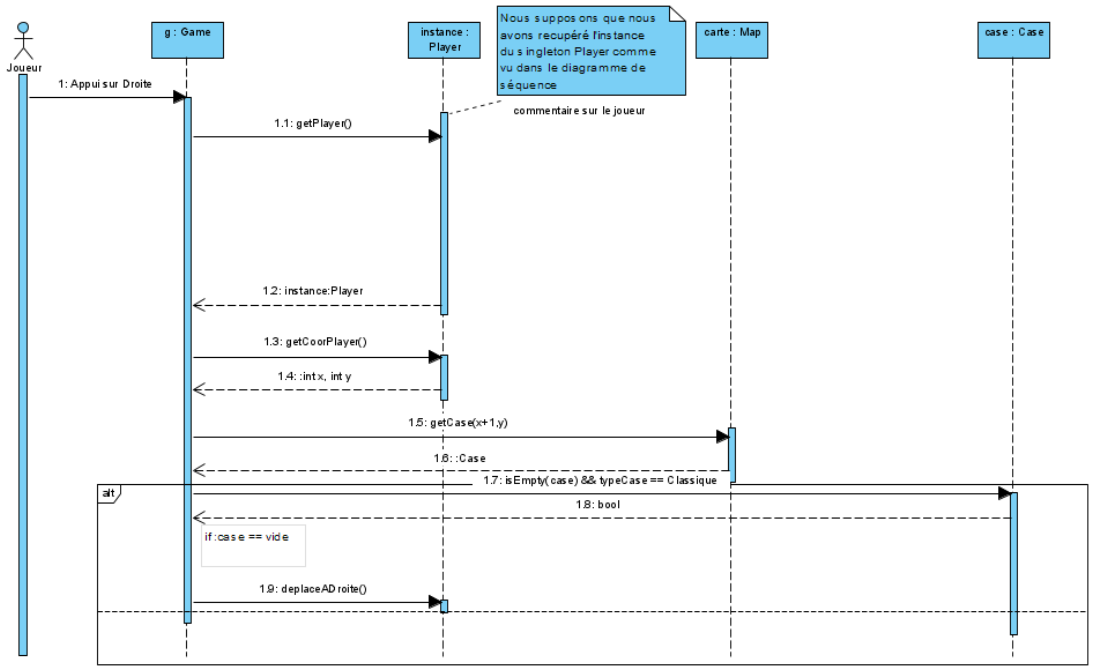
L'utilisateur va lancer la fonction newGame() depuis le menu principal. Ce dernier va demander à ce qu'un jeu soit crée, ce qui entraînera la génération d'une carte composée de plusieurs cases crées aléatoirement. Enfin, un joueur sera crée puis le jeu se lancera grâce à la fonction run().

IV.3 - DIAGRAMME DE SEQUENCE DE LA FONCTION PAUSE



Lorsque le joueur appuiera sur la touche censée mettre le jeu en pause, le jeu lancera la fonction pause() qui va *freeze* la partie. Ensuite, il va demander à ce qu'une PauseWindow soit créée et lui ordonnera de s'afficher (show()).

IV.4 - DIAGRAMME DE SEQUENCE DE LA FONCTION MOVE



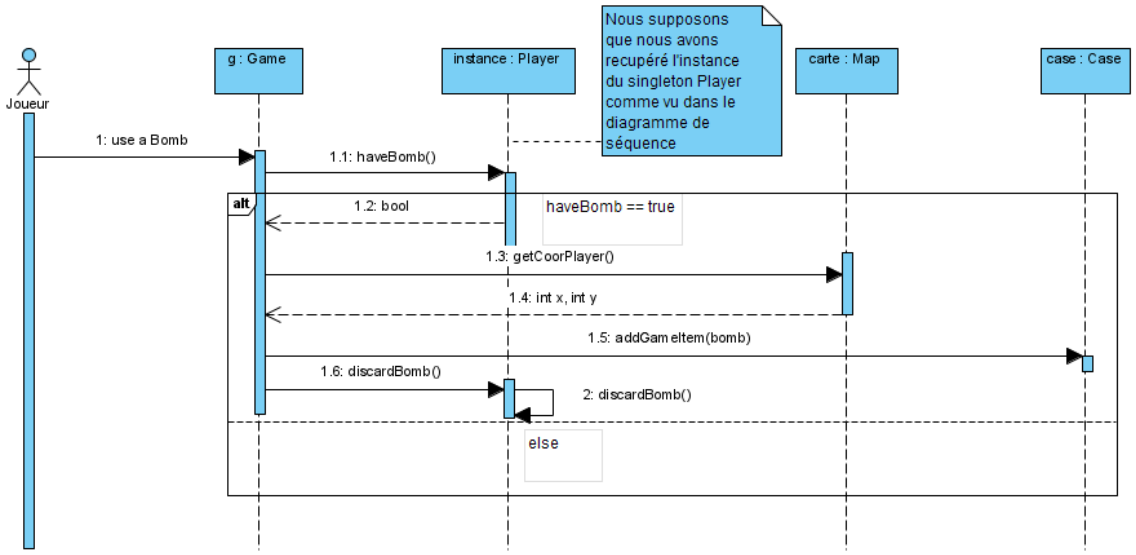
Sur ce diagramme de séquence on peut voir ce qu'il se passe lorsque le joueur veut se déplacer à droite. Nous avons décidé de ne le faire que le déplacement à droite car le diagramme de séquence sera exactement le même pour les autres déplacements.

Lorsque l'utilisateur appuiera sur la touche pour déplacer le joueur à droite cela enverra un message à Game qui va demander à récupérer le joueur à la classe Joueur.

Une fois que le jeu aura appelé le joueur, il récupérera ses coordonnées et retournera deux entiers correspondants aux coordonnées du joueur. Une fois ces coordonnées obtenues on demandera les coordonnées de la case se trouvant à droite du joueur à la classe Carte les demandera à la classe Case et retournera les coordonnées de la case à droite du joueur.

Nous avons ensuite fait une condition pour que le jeu vérifie si la case à coté du joueur est vide, si celle-ci est vide alors le joueur pourra se déplacer dessus sinon il ne se passera rien du tout.

IV.5 - DIAGRAMME DE SEQUENCE DE LA FONCTION USEABOMB



Ce diagramme de séquence représente la façon dont le jeu va réagir lorsqu'il va interagir avec un objet. Nous avons pris comme exemple, l'utilisation d'une bombe.

Lorsque le joueur appuiera sur la touche qui permet d'utiliser une bombe, le jeu vérifiera tout d'abord si le joueur possède une bombe. S'il n'en a pas, rien ne se passe.

Si le joueur possède une bombe, le jeu récupèrera les coordonnés du joueur puis créera une bombe sur la case où se trouve le joueur. Enfin, le jeu décrémentera le nombre de bombe possédé par le joueur.