Arnaud Curie | Pascaline Guichard | Juliette Gagnepain |

Guillaume Clerc | Johann Fouchard | Noa Barbosa

IUT DIJON | DEPARTEMEnt informatique

Livrable n°2

Dossier de conception



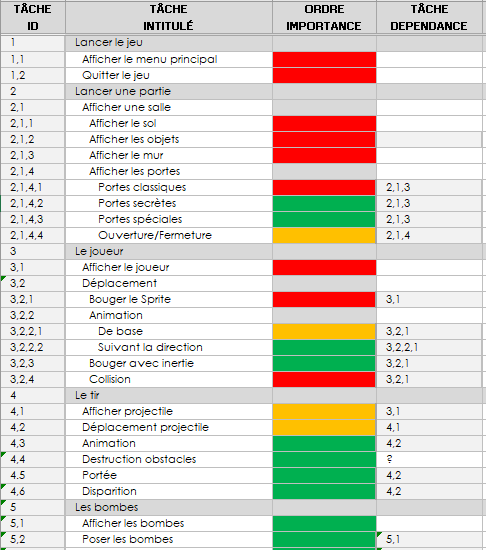
# Introduction

Dans ce dossier de conception, vous verrez ce que nous avons fait pour la partie conception de notre projet. Mais avant tout, il nous faut rappeler quel est notre projet :

Notre projet consiste en la réalisation d’un clone du jeu *The Binding of Isaac*. C’est un jeu indépendant d’action-aventure en 2D isométrique. Le style du jeu est volontaire gore et effrayant mais est adouci par des designs cartoons. Nous avons décidé pour notre clone de reprendre un style similaire dans l’univers d’Alice au pays des merveilles. L’objectif du projet est de réaliser la version la plus abouti possible du jeu.

Le projet étant d’envergure et nos idées pouvant changer, il serait irréaliste d’essayer de faire une conception complète. En effet, d’une part nous oublierons certainement des éléments mais aussi nous comptons rajouter des éléments au fur et à mesure de notre progression. Nous avons donc fait une conception très large pouvant accueillir beaucoup d’élément.

# Backlog produit

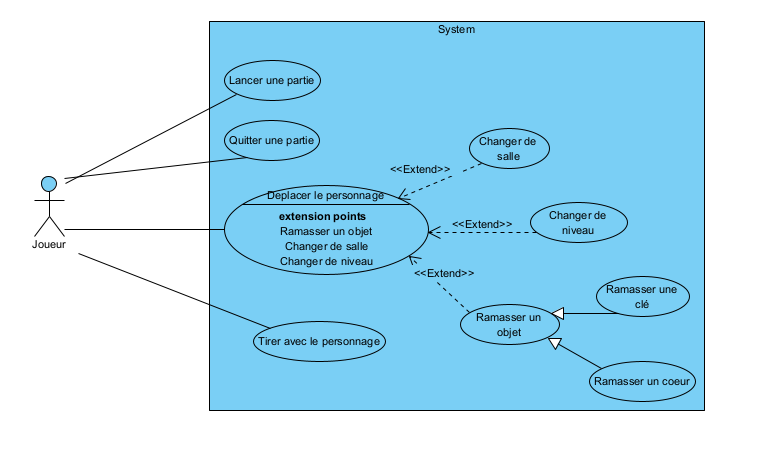
Nous avons commencé par réaliser un backlog produit. Un backlog produit a la même utilité qu’un Use-Case mais sous une forme différente. Nous allons lister les fonctionnalités qu’attend notre client en les rangeant par grande catégories et par importance. De plus nous rajoutons les dépendances entre chaque fonctionnalité. Rouge = Vitale | Orange = Important | Vert = Annexe.

Une image contenant table

Description générée automatiquementUne image contenant table

Description générée automatiquement

# Diagramme de cas d’utilisation

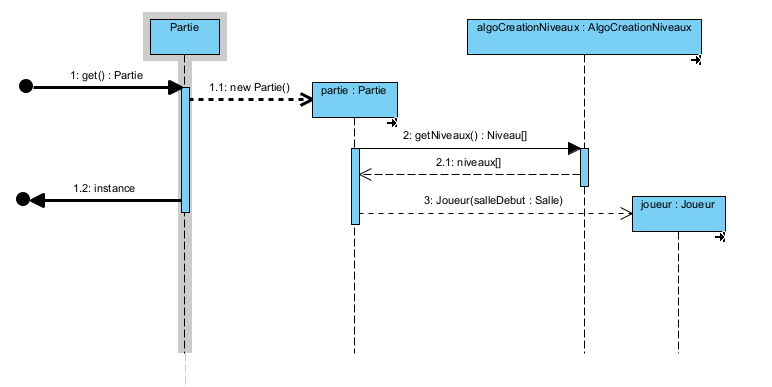


Voici le diagramme de cas d’utilisation. Étant donnée la taille de notre projet il ne représente que les fonctions « vitales » (voir BackLog produit) du jeu. Nous avons Lancer une partie et Quitter qui sont des fonctionnalités classiques. Tirer avec le personnage est aussi très fréquent dans un jeu vidéo. Le point intéressant est sur le déplacement du personnage. Le jeu est en 2D isométrique, et quand le joueur veut interagir avec un objet il doit marcher dessus. Donc ramasser un cœur ou une clé fait en fait partie de se déplacer. Comme Changer de salle ou de niveau. Pour ces derniers, le joueur doit soit aller vers une porte ou vers une trappe. Nous pouvons donc considérer que cela fait partie de se déplacer.

# Diagrammes de séquence

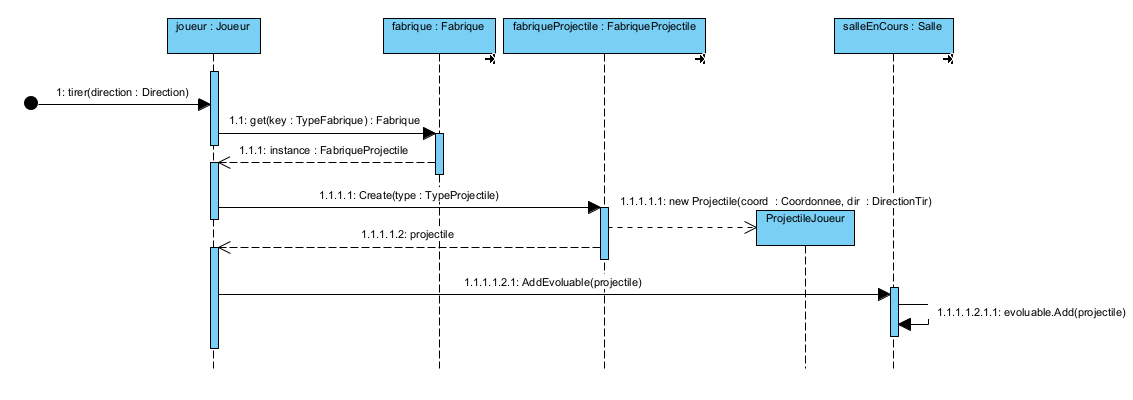
Les cas d’utilisations étant fait nous avons réalisé les diagrammes de séquences pour ces mêmes cas d’utilisation.

## Lancer une partie



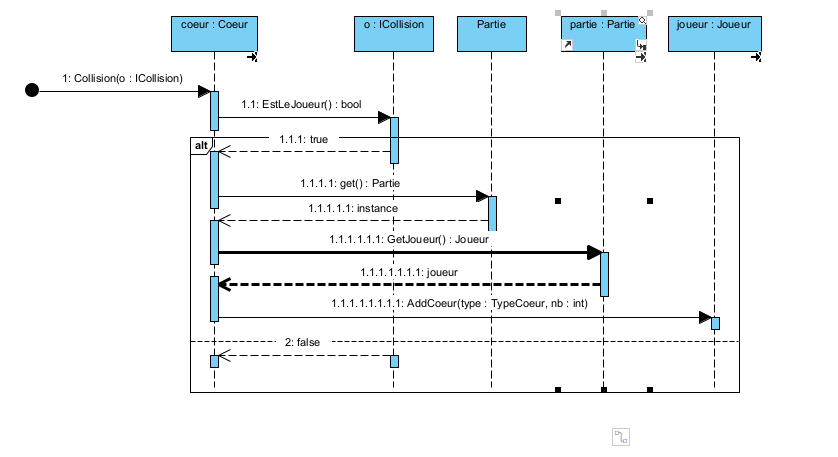
Dans ce diagramme nous souhaitons lancer une partie. Nous avons mis la classe Partie en singleton car nous allons avoir besoin de l’appeler régulièrement dans notre code. Nous l’appelons donc et à sa création elle demande un tableau de Niveau à une interface nommé AlgoCreationNiveaux. Pour notre clone d’Isaac, il nous faudra des algorithmes de créations de niveaux, de cartes, de salles… C’est à cela que sert cette classe. N’ayant pas encore les compétences pour ces algorithmes, nous les modélisons sous la forme d’une classe qui contient simplement un tableau de Niveaux qu’elle nous renvoie.

## Le joueur tire

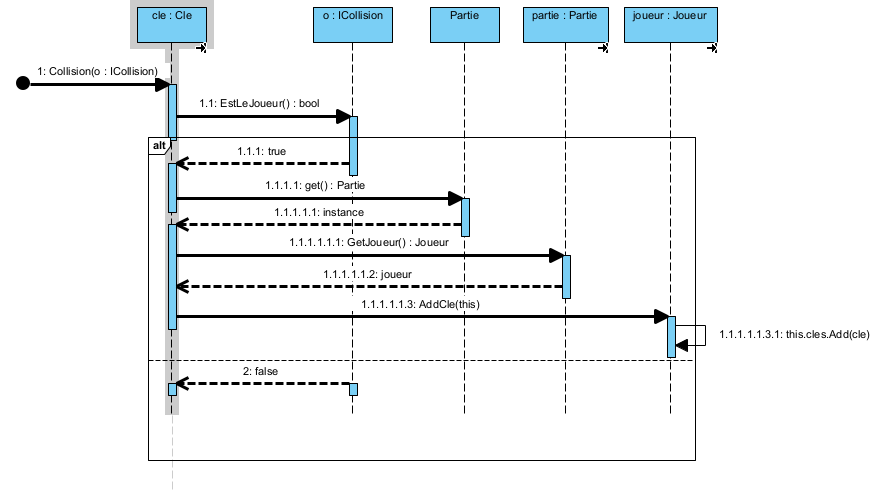


Voici le diagramme de séquence du joueur qui tire. Au moment où on demande au joueur de tirer, il appel l’instance de la fabrique correspondante pour instancier un projectile. Puis le joueur ajoute à la salle en cours le projectile. Le joueur connait ça salle en cours et la salle contient plusieurs interface nommé Evoluable.

## Collision du joueur avec un cœur

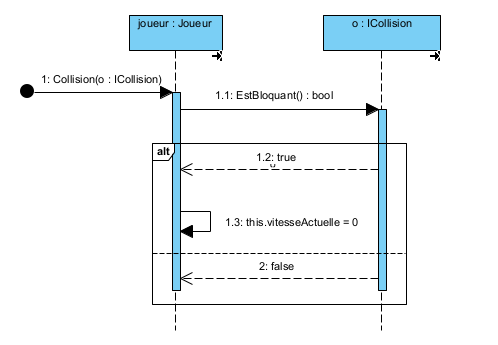
Dans ce diagramme, l’IHM va appeler le cœur pour lui dire qu’il est actuellement en collision avec un autre objet. Le cœur va donc demander si cet objet est le joueur. Si non, alors la fonction s’arrête. Mais si oui, alors le cœur récupère l’instance du singleton Partie pour ainsi récupérer l’instance du joueur. Le cœur pourra donc ajouter son type et le nombre de cœur qu’il est. Le joueur ne stocke pas directement une instance de cœur car nous prévoyons qu’il y est plusieurs types de cœur et qu’il soit géré en demi-cœurs comme sur le jeu de base. Cette modélisation rendra les algorithmes plus faciles.

## Collision avec une clé



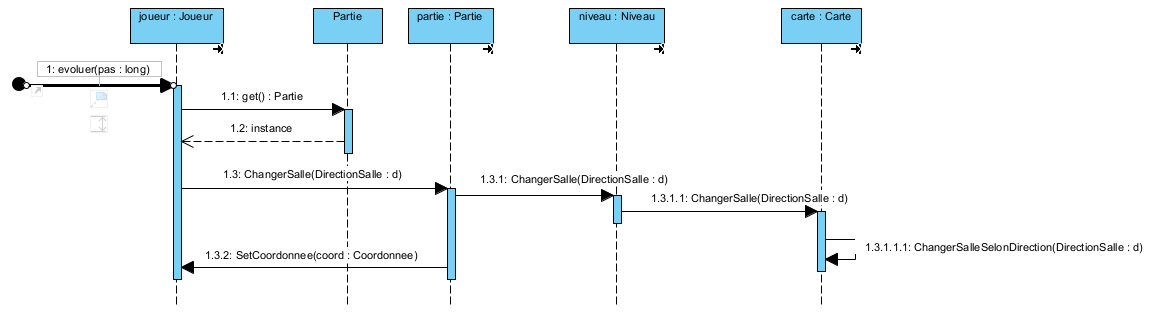
Ce diagramme est très similaire au précédent. La clé reçoit l’information qu’elle touche un objet. Elle lui demande s’il est le joueur. Si nan, il ne se passe rien. Si oui, alors elle récupère l’instance du joueur par le biais du singleton partie et ainsi elle s’ajoute elle-même aux joueurs. Remarquez que cette fois le joueur contient directement les instances des clés. Cela nous permettra un code plus simple étant donné que nous n’avons pas à différencier différents types de clé.

## Collision du joueur



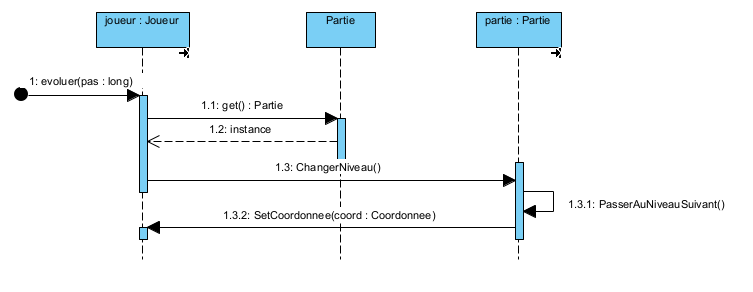
Ce diagramme de séquence nous présente ce qu’il se passe lorsque le joueur est en collision avec un autre objet. Il lui demande simplement s’il est bloquant. Si oui, alors nous mettons sa vitesse à 0. Si non, il ne se passe rien.

## Changer de salle



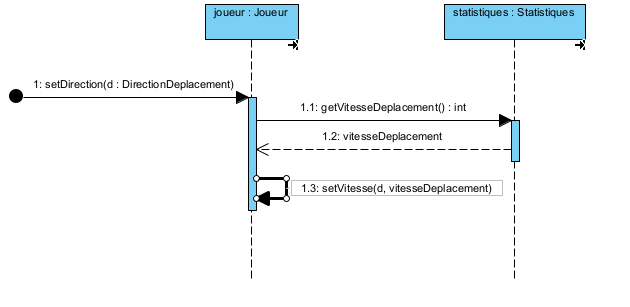
Ce diagramme de séquence commence par l’appel à la méthode évoluer du joueur. Pour que le joueur change de salle, il doit marcher sur une porte et donc dépasser une certaine hauteur ou largeur. Dans la méthode évoluer ce dépassement sera détecté et une direction sera donnée pour changer de salle. Cette direction représente l’emplacement de la porte que le joueur à pris par rapport à la salle. La méthode Evoluer va tout d’abord récupérer l’instance de la Partie (Rappelons que Partie est un singleton). Ensuite elle va appeler la méthode ChangerSalle de Partie en lui donnant la direction. Cette dernière va faire passer l’information de Niveau jusqu’à Carte. Et la carte, qui connait la salle en cours, va la changer. Nous n’avons volontairement pas détaillé l’algorithme pour changer de salle car, dans un premier temps, ce sera fait statiquement, puis, dans une amélioration future, nous implémenterons un algorithme plus complexe. Enfin la partie doit replacer le joueur dans la salle nouvellement chargé. C’est pourquoi il va changer les coordonnées du joueur.

## Changer de niveau



Ce diagramme détaille le changement de niveau. Tout d’abord, le joueur récupère l’instance du singleton Partie. Ensuite, le joueur appel la méthode changerNiveau de la classe Partie. Cette dernière connait les niveaux car fixé statiquement dans un premier temps (voir diagramme de séquence pour lancer une partie), elle va donc simplement passer au niveau suivant. La partie doit après changer la position du joueur par rapport au niveau et à la salle nouvellement chargés. Nous n’avons volontairement pas détaillé l’algorithme pour le passage au niveau suivant car il est destiné à changer plus tard.

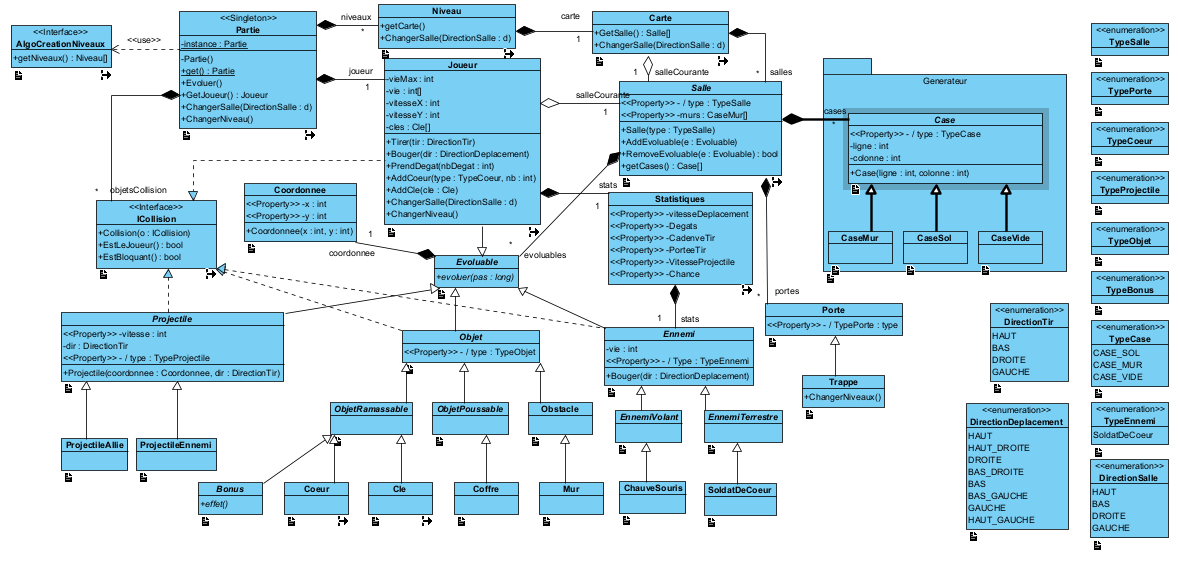
## Déplacement du joueur



Dans ce diagramme de séquence nous allons voir le déplacement du joueur. Tout d’abord, le joueur reçoit une direction du déplacement. Puis il va récupérer, dans la classe statistiques, sa vitesse de déplacement. Ensuite, il va déterminer la vitesse de ses deux attributs vitesseX et vitesseY dans sa méthode SetVitesse.

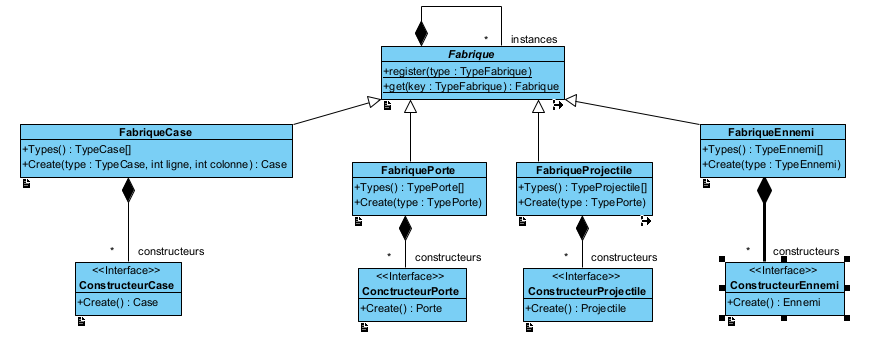
# Diagrammes de classe

## Diagramme métier



Nous voici donc au diagramme de classe illustrant la partie métier de notre application. Comme dit précédemment, il ne représente que la partie « vitale » du jeu mais il reste quand même assez imposant par sa taille. Nous savons d’ores et déjà qu’il est incomplet et qu’il va évoluer au fil du temps. Nous l’avons pensé pour pouvoir accueillir de nombreux autres éléments.

## Multiton de fabriques abstraites



Nous avons décidé d’utiliser le design pattern de la fabrique abstraite, le trouvant particulièrement adapté à notre situation. Nous avons remarqué que nous aurons besoin de nombreuses fabriques pour notre projet. Et en sachant que la fabrique abstraite est elle-même un singleton, nous avons décidé de combiner la fabrique abstraite avec un autre design pattern : le multiton. Cela nous permet d’appeler depuis n’importe où dans notre code nos fabriques abstraites et de les regrouper sous la même structure. Il est lui aussi destiné à évoluer. Nous aurons surement d’autres fabriques à rajouter.