

Jeu FlowFree

Rapport de projet

Ce document présente les différentes phases du projet FlowFree réalisé dans le cadre de l’électif Génie logiciel orienté objet

Dossier rédigé par *Hugo BORSONI et Vincent FLATTOT.*

17*/01/2021*

Table des matières

[01. Cahier des charges 2](#_Toc61805974)

[A. Présentation du jeu Flow Free 2](#_Toc61805975)

[B. Présentation d’une phase de jeu (version mobile) 2](#_Toc61805976)

[C. Notion de niveaux 3](#_Toc61805977)

[02. Expression des besoins 3](#_Toc61805978)

[03. Conception 3](#_Toc61805979)

[04. Réalisation 3](#_Toc61805980)

[05. Conclusion 3](#_Toc61805981)

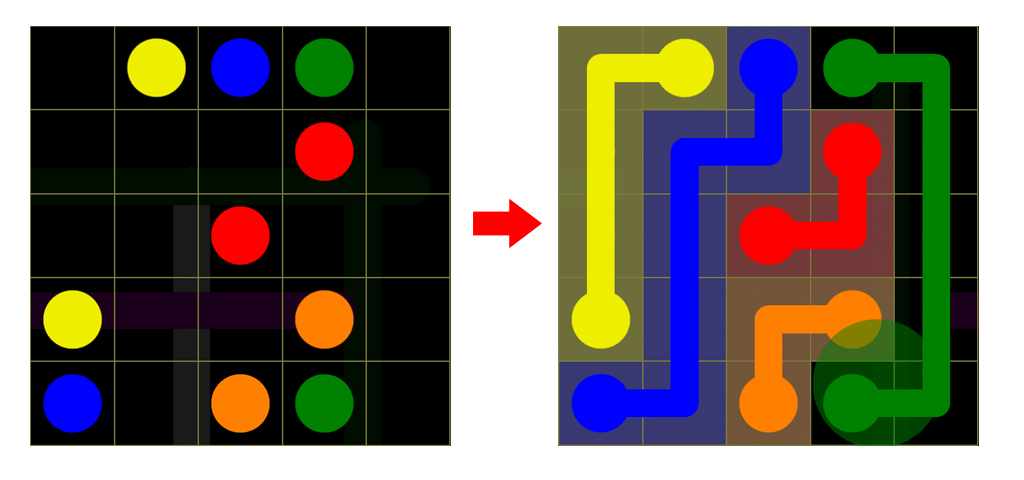
NB : les codes utilisés pour dessiner les différents diagrammes sont fournis dans le dossier code\_dossier rendu avec ce fichier. On pourra par exemple les entrer dans le site web plantuml.com afin d’obtenir les résultats graphiques.

# Cahier des charges

## Présentation du jeu Flow Free®

Le jeu dont nous nous inspirons est Flow Free®. Il s’agit d’un jeu de puzzle développé par le studio Américain Big Duck Games et sortit en juin 2012. Il s’agit initialement d’un jeu pour mobile et est disponible sur IOS et Android.

Le concept du jeu est assez simple. On part d’un plateau carré découpé en NxN cases. On dispose sur 2xn cases des « Plots » en forme de ronds. Il faut ensuite à l’aide de « Tuyaux colorés » réussir à relier chaque plot de la même couleur tout en s’assurant que l’ensemble {Tuyau+Plot} couvre bien l’ensemble des cases du Plateau et qu’aucun Tuyau ne croise un autre Tuyau .



*Une illustration d’une partie au début et à a fin*

## Présentation d’une phase de jeu (version mobile)

Dans la phase de jeu, l’utilisateur peut cliquer sur un plot puis tracer avec son doigt le chemin qu’il souhaite lui faire prendre. **Dans notre version, l’utilisateur cliquera avec sa souris sur un plot puis déplacera le tuyau grâce aux flèches de son clavier**

Si l’utilisateur lâche son doigt, le tuyau s’arrête sur la dernière case balayée par l’utilisateur. **De même, dans notre version, nous utiliserons ce procédé**

Si l’utilisateur fait croiser le nouveau tuyau avec un ancien, l’ancien tuyau est effacé jusqu’à la case précédant le croisement, le tuyau en construction reste celui que l’utilisateur était en train de construire. **Dans notre version du jeu, nous supprimerons le tuyau courant entièrement et l’utilisateur pourra recliquer sur un plot pour construire un nouveau tuyau**.

Durant tout le jeu, l’utilisateur est informé du nombre de tuyaux qu’il a réussi à finir, le pourcentage du plateau couvert par les tuyaux et le nombre de coup réalisé. **Nous allons essayer de reproduire ce comportement**

Lorsque l’utilisateur à réalisé le dernier mouvement avant la victoire, le jeu l’informe de la réussite et indique le nombre de tentatives qu’il a fallu pour réussir le niveau**. Nous allons essayer de reproduire ce comportement**

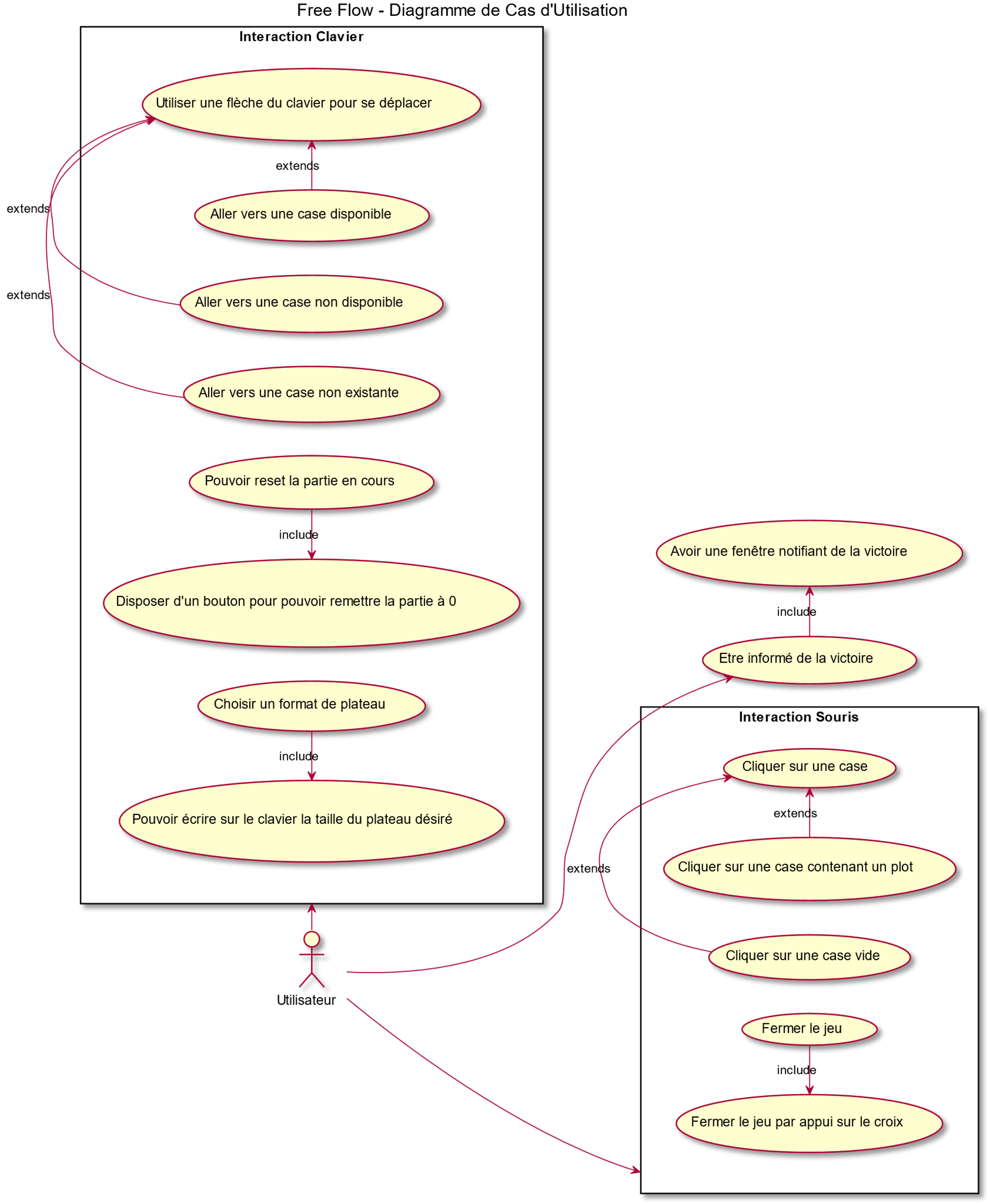
## Notion de niveaux

Dans le jeu, il existe différents niveaux auxquels on peut accéder lorsqu’on a réussi les niveaux précédents. **On essaiera dans ce projet de donner la possibilité à l’utilisateur de changer de « niveau » en changeant le nombre de cases du plateau**. **Ce choix s’effectuera au début via un sélecteur.**

# Expression des besoins

L’utilisateur du jeu a plusieurs attentes que nous récapitulons ici sous forme d’une liste et que nous illustrons via un diagramme de cas d’utilisation et de différents scénarios.

* Pouvoir sélectionner un plot pour créer un tuyau
* Pouvoir se déplacer dans le plateau avec les tuyaux
* Pouvoir connaitre son avancée dans le jeu
* Pouvoir savoir qu’on a gagné
* Pouvoir quitter le jeu

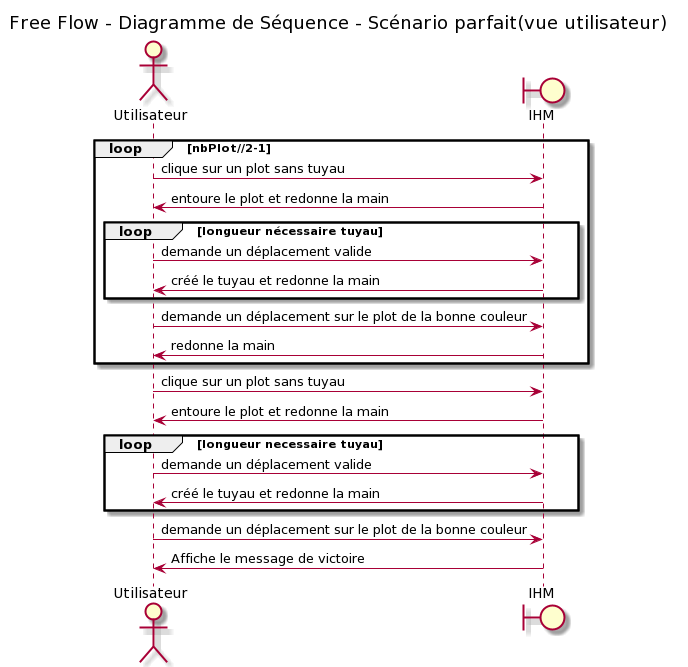


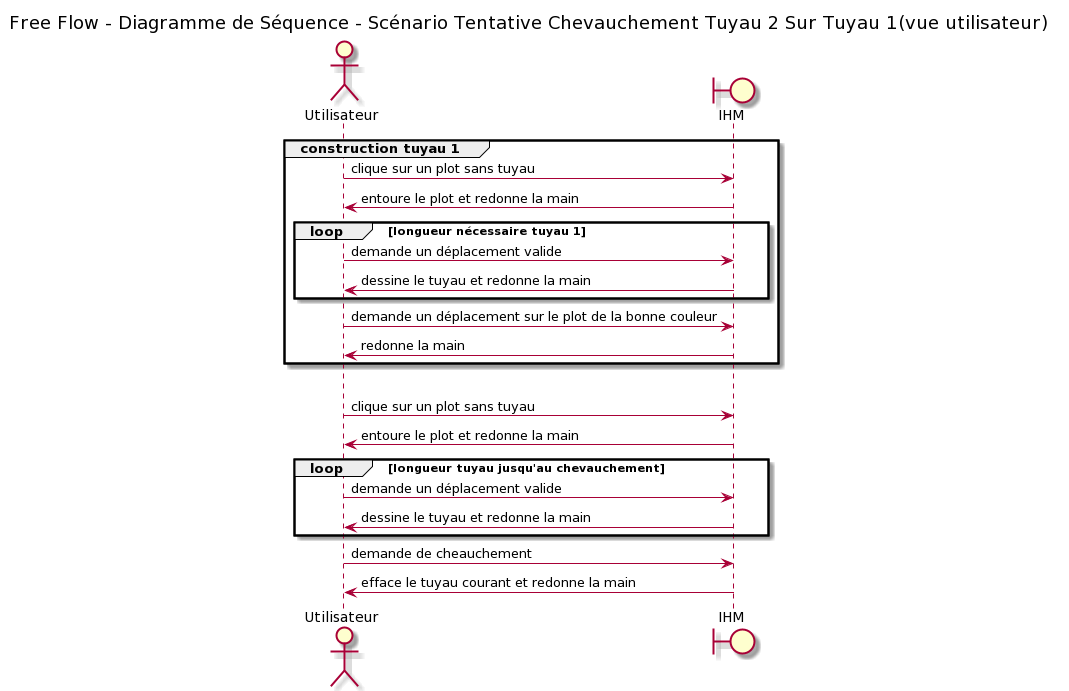
Ces cas d’utilisations peuvent être mis en parallèle avec les actions qu’ils vont nécessiter de la part de l’Interface Homme-Machine :

Une image contenant texte

Description générée automatiquement

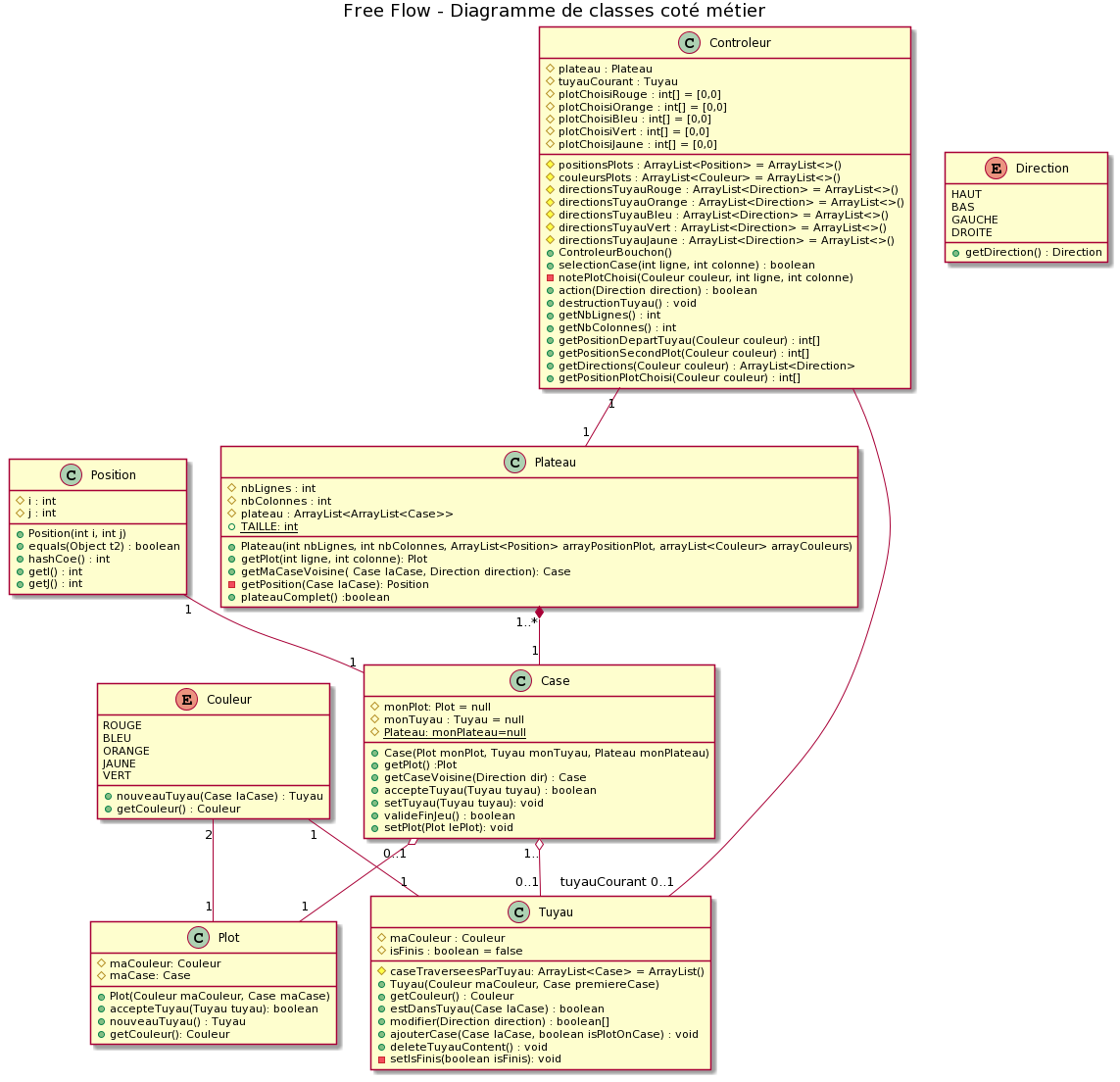
Afin de permettre une meilleure compréhension de ces éléments, on propose la modélisation d’un scénario « parfait » et d’un scénario avec une erreur du point de vue du joueur





# Conception

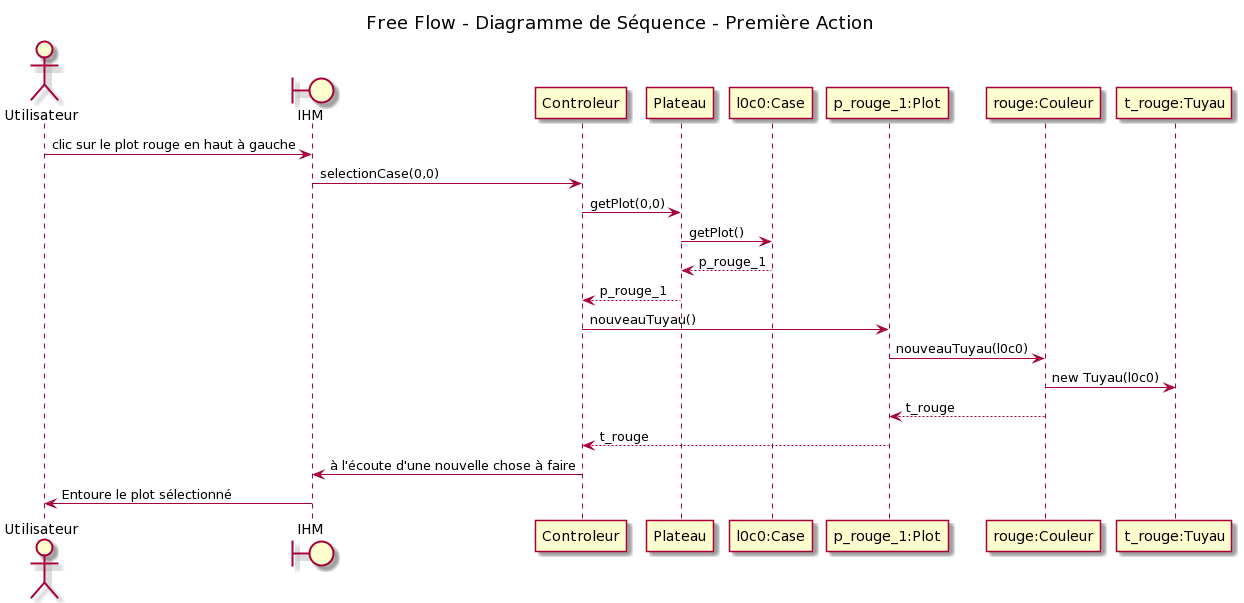
Pour commencer cette partie, on présente ci-dessous le diagramme de classe de notre application (coté machine) afin de pouvoir comprendre plus facilement les éléments en jeu.

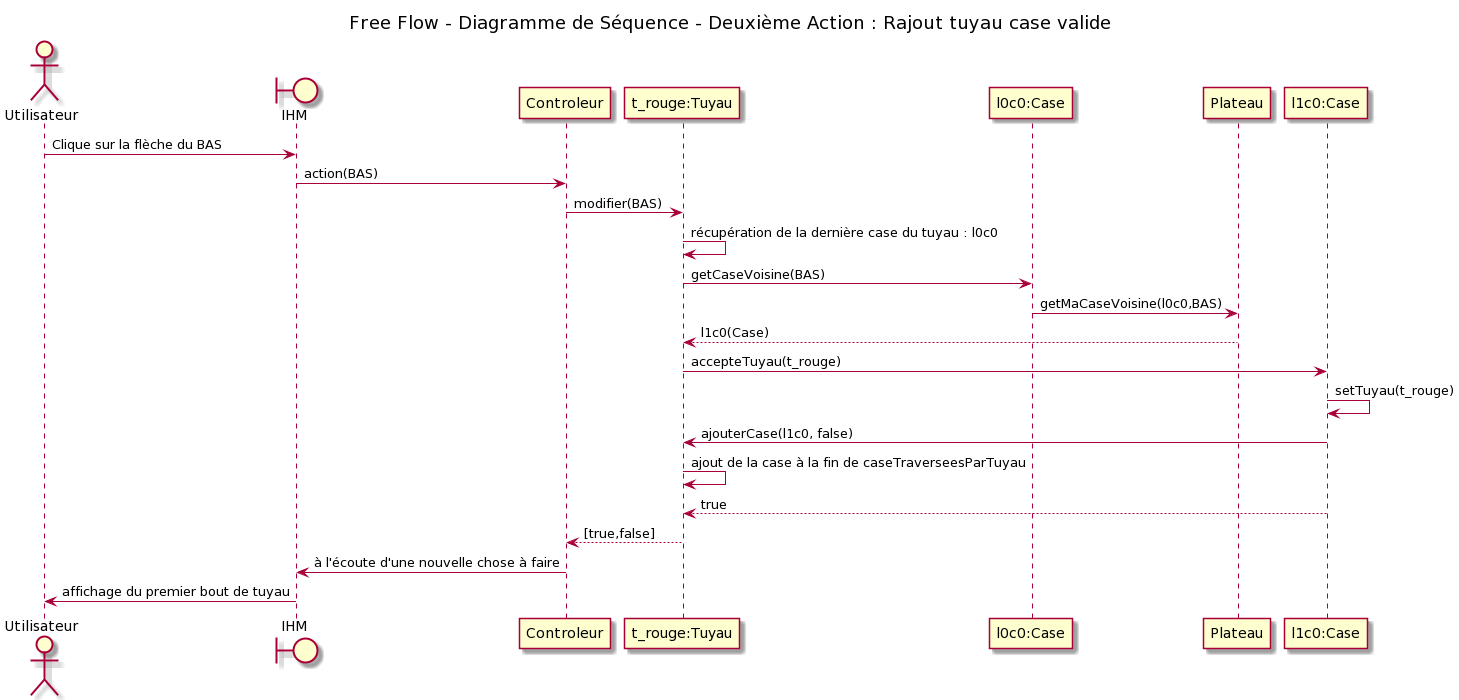


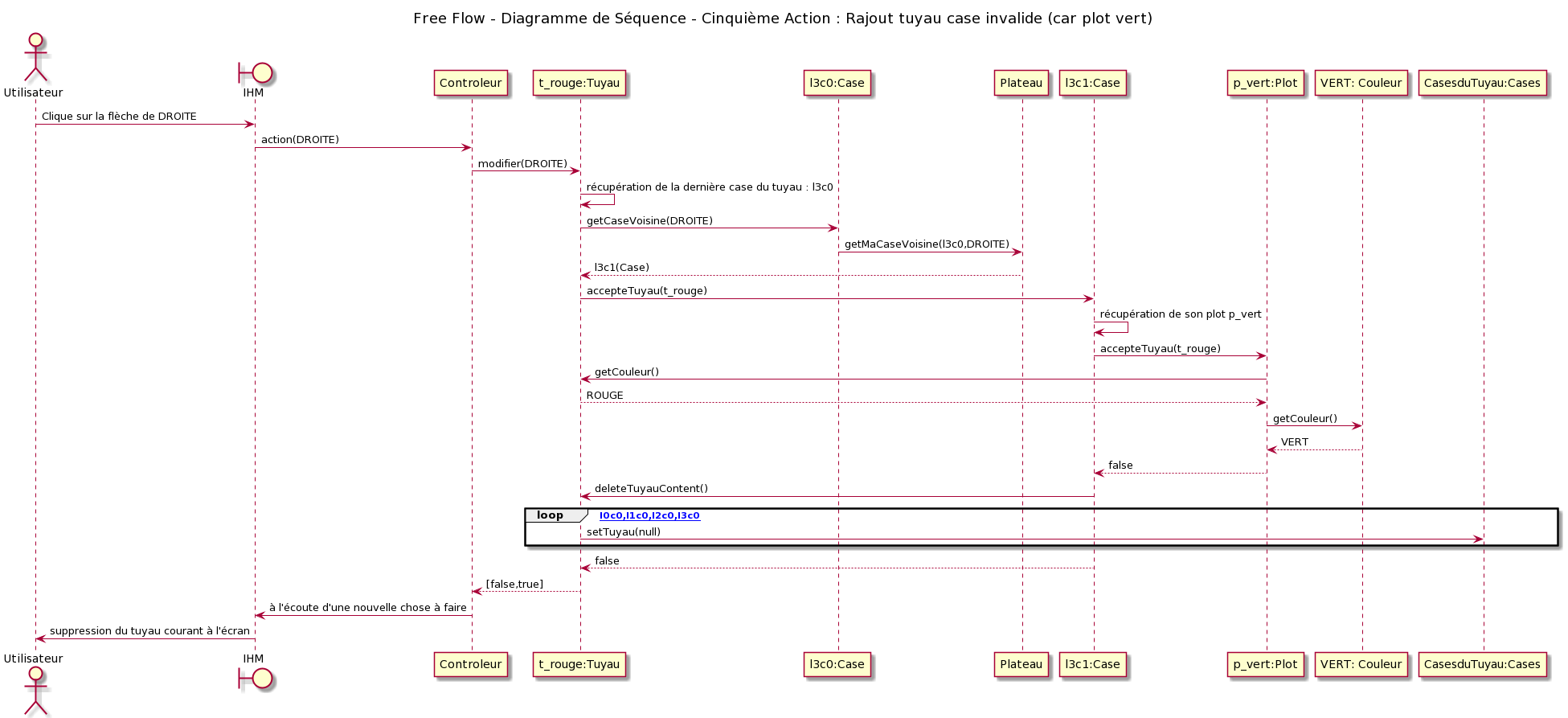
En ayant repris en grande partie le modèle développé lors du TD, nous allons développer uniquement les éléments que nous avons rajoutés et qui ne sont pas explicites.

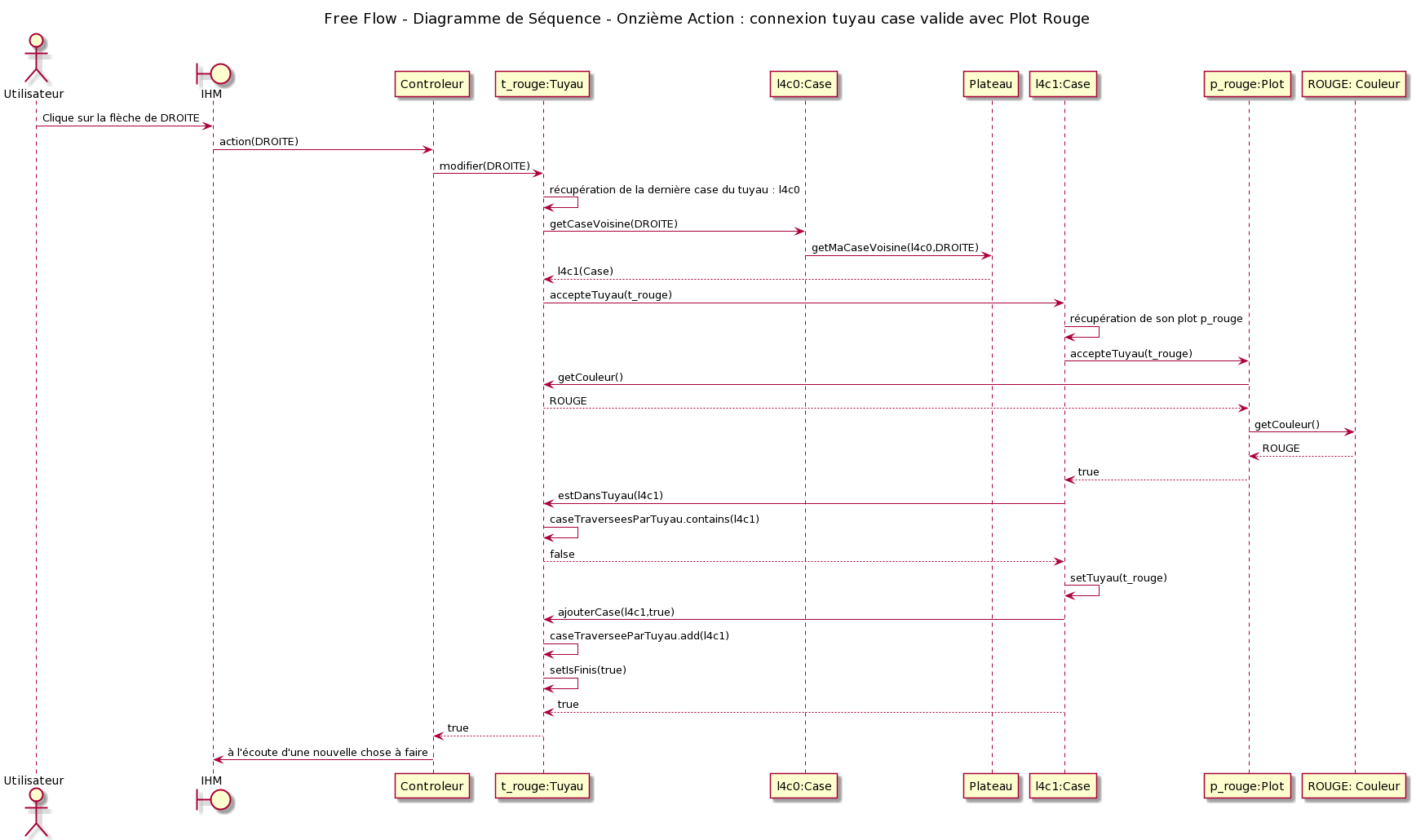
* La classe Controleur:
* La classe Case :
  + Elle possède le plateau qui la contient comme attribut
  + Elle possède la méthode accepteTuyau(Tuyau tuyau) :boolean[] qui renvoit deux booléens, un pour dire que on a rajouté une case à un tuyau et un pour dire si le tuyau est fini, A COMMENTER DANS LE CODE PARCE QUE C4EST PAS COMPREHENSIBLE
  + Elle possède la méthode deleteTuyau() : boolean qui supprime le tuyau qui a voulu se mettre sur la case ou qui était présent sur la case et retourne si ce tuyau était terminé afin de pouvoir faire les statistiques sur le nombre de tuyaux terminés
* La classe Plot :
  + Il possède des attributs qui indiquent sa couleur et la case sur laquelle il se situe
  + Il possède une méthode accepteTuyau(Tuyau tuyau) :boolean qui lui permet de dire à un tuyau s’il peut se raccorder à ce plot.
* La classe tuyau
  + Un tuyau possède un attribut isFinis qui lui permet de savoir qu’il a ses deux bords qui touchent un plot et donc qu’il ne doit pas pouvoir continuer à s’étendre
  + Il possède aussi un attribut caseTraverseesParTuyau qui est un ArrayList de case qui contient l’ensemble des cases sur lesquelles est le tuyau.
  + Il possède une méthode estDansTuyau(Case laCase) : boolean qui lui permet de dire à une case si elle parmi les cases qu’il traverse
  + Il possède une méthode modifier(Direction direction) : boolean[] qui retourne un tableau de boolean. Le premier correspondant au fait que le mouvement est possible et le deuxième correspondant au fait que le tuyau doit se détruire car l’utilisateur a fait une action « interdite »(ex : vouloir couper un autre tuyau ou rentrer dans un plot de la mauvaise couleur ) et le troisième correspondant au fait que le tuyau était fini afin de modifier les statistiques de jeu
  + Il possède une méthode ajouterCase(Case laCase, boolean isPlotOnCase) : boolean qui ajoute la case au tuyau et retourne si le tuyau viens d’être finis ou non
  + Il possède une méthode deleteTuyauContent() :void qui lui permet de vider son arrayList de cases traversées si on lui demande de se détruire.
* La classe Position :
  + Il s’agit d’une classe qui permet de repérer les cases sur le plateau
  + Elle possède deux attributs : une ligne i et une colonne j
  + Elle permet surtout de pouvoir redéfinir la méthode equals(Object t2) : boolean utile pour faire de la comparaison sur notre ArrayList de Plateau contenant les cases.
* La classe plateau :
  + Le plateau contient un ArrayList 2D des cases. Ce plateau est créé grâce à deux ArrayList, un de Position et un de Couleur. On met dans l’ArrayList de position toutes les positions des différents Plots et on leur associe leur Couleur dans l’autre ArrayList. Il faut donc un nombre pair d’éléments dans chacun de ces deux ArrayList vu que les Plots vont par deux.
  + Il possède aussi une constante TAILLE qui indique sa taille afin de pouvoir redéfinir hashCode() :int dans Position().
  + Le plateau possède une méthode getPosition(Case laCase) :Position qui permet de renvoyer la position d’une case connaissant cette case.
  + Le plateau possède une méthode getPlot (int ligne, int colonne) : Plot qui permet d’aller chercher le plot sur une case à une certaine position et ça via la méthode getPlot de la classe Case.
  + Il possède aussi la méthode getCaseAutrePlotByCouleur(Plot plotCourant, ArrayList<Couleur> arrayCouleurs, ArrayList<Position> arrayPositionPlot) : Case qui retourne la case sur laquelle est située l’autre plot de la même couleur que plotCourant

Afin d’illustrer les utilisations de ces différentes classes, on propose les séquences suivantes expliquant le comportement du système vis-à-vis d’actions de l’utilisateur :









# Réalisation

Pour coder cette application nous avons décidé de nous repartir les taches de la manière suivante :

Hugo s’est occupé de la partie interface graphique et contrôleur

Vincent s’est occupé de la partie métier appart le contrôleur

Nous avons ainsi pu avancer en parallèle sur chacune des parties sans trop dépendre de l’autre. En effet Hugo a pu coder son interface graphique en partant de la base donnée dans le sujet du projet. Ainsi il avait la possibilité de créer des phases de jeu artificiellement sans avoir réellement besoin des objets métiers. De la même manière, Vincent a pu coder le coté métier de l’application sans les actions de l’utilisateur (utilisation d’une classe Main où on entrait une suite d’action de l’utilisateur via des appels de méthodes et qui se déroulait alors séquentiellement lors du run, classe Main en annexe AVORIIIRRIRIRIRI). Nous pensons que cette manière de faire était vraiment intéressante car elle permettait vraiment une parallélisation des tâches et un besoin de communication uniquement pour raccorder les deux morceaux. (ce qui n’aurait pas été le cas si nous nous étions séparé l’ensemble des classes en deux aléatoirement et que nous avions dû attendre le nom d’une méthode d’une classe que l’un ne code pas pour que l’autre puisse l’utiliser dans sa classe). Nous avons parlé d’un besoin de communication restreint mais nous tenons à mentionner que par le fait que nous sommes dans la même collocation, nous avons pu travailler ensemble et nous avons en réalité décidé de ne faire que des sessions de code communes pour pouvoir avancer plus vite en s’aidant mutuellement sur des points de blocage.

Lorsque nous avons tous les deux finis nos parties, nous avons dû recoller ces deux morceaux, via les différentes fonctions appelées par le controleurBouchon. Cette étape n’a pas été très compliqué et nous avons alors pu tester des cas non envisagés par l’un ou l’autre car il est vrai que lorsqu’on code une partie, on a tendance a faire toujours les même test et on n’imagine pas forcément une action bizarre que pourrait faire l’utilisateur. Par exemple lors de la phase de conception du modèle métier, Vincent avait oublié de prévoir le cas où l’utilisateur décidait de créer un tuyau et de le faire boucler sur le plot de départ. C’est Hugo qui a eu cette idée (Hugo a eu la plupart des idées bizarres que les utilisateurs pourraient avoir en fait, il faut l’admettre*)*

On est alors arrivé dans une phase itérative ou l’une ou l’autre des parties rencontrait un problème, on cherchait sa source via un système de print dans la plupart des méthodes appelées et qui permettait de voir les appels de méthodes au cours du temps. On résolvait cette erreur et on se remettait à tester l’application sur un autre cas limite afin de la rende la plus robuste possible.

Liste des tests mis en place :

* Partie métier hors Controleur :
  + Tests unitaires sur nos différentes méthodes avec un système de print et une réflexion sur les valeurs attendues lors de nos appels
  + Tests de chaine via l’analyse des séquences d’appel qui sont réalisées dans le jeu et grâce à un système de print
  + Test de scénario via la clase Main qui permettait de simuler des cas d’utilisation précis
  + Test d’intégration via l’interface graphique, correction des cas non pris en compte(Monkey Testing) en réutilisant le test de scénarios puis en réessayant depuis l’interface graphique
* Partie IHM + Controleur  :
* Tests utilisateurs : nous avons fait tester notre jeu à nos 4 autres colocataires

# Conclusion

Limites : lorsqu’on spamme le clavier ça chibre

# Annexe

Code pour le diagramme des cas d’utilisation :

diag\_usecase.txt

Code pour le diagramme de séquence des actions utilisateur

diag\_sequence\_useractions.txt

Code pour le diagramme de séquence scénario « parfait »

diag\_sequence\_perfectcase.txt

Code pour le diagramme de séquence scénario « erreur »

diag\_sequence\_errorcase.txt

Code pour le diagramme de classes :

diag\_classe.txt

Code pour les différents diagrammes de séquences de la partie conception :

Diag\_sequence\_NUMEROACTION.txt