# 1 PO&GL, DM Colt Express

Ce projet est à réaliser en groupes de un à trois, et doit être hébergé sur le GitLab de l'université (https ://gitlab.u-psud.fr/) même si réalisé en monôme. Soutenance la semaine du 10 mai, en présentiel, a votre horaire habituel de TP, dans la salle ou vous auriez du être si on avait été en présentiel toute l'année.

La séance de TP juste avant les vacances de printemps sera dédiée à ce projet. Vous en profiterez d'autant que vous aurez déjà travaillé sur le sujet !

Ce DM vous invite à construire une version électronique et un peu simplifiée du jeu *Colt Express*[1].

*Les indications de rendu et consignes générales sont groupées à la fin du sujet.*

## 1.1 Aperçu des règles du jeu

Le jeu se déroule à bord d'un train, composé d'une locomotive et d'un certain nombre de wagons. Les joueurs incarnent des bandits qui ont sauté à bord pour détrousser les passagers. Objectif : récupérer le plus de butin possible, chacun pour soi. Il s'agit d'un jeu de programmation, dans lequel on alterne entre deux phases :

**Planification :** chaque joueur décide secrètement un certain nombres d'actions, que son personnage va effectuer dans l'ordre.

**Action :** on effectue toutes les actions numéro 1, puis toutes les numéro 2, et ce jusqu'au bout.

Les bandits peuvent se trouver dans les wagons ou la locomotive, et pour chacun de ces éléments soit à l'intérieur soit sur le toit. Dans cet énoncé, par abus de langage on désignera par « wagon » un élément quelconque du train, qui peut être la locomotive. Les actions possibles pour les bandits sont :

- Se déplacer d'un wagon en avant ou en arrière, en restant au même étage.
- Aller à l'intérieur ou grimper sur le toit de leur wagon actuel.
- Braquer un voyageur pour récupérer du butin (ou simplement récupérer un butin qui a été abandonné là).
- Tirer sur un autre bandit proche pour lui faire lâcher son butin.

Les butins récupérables à bord du train sont :

- Des bourses valant entre 0 et 500$, auprès des passagers, à l'intérieur des wagons.
- Des bijoux valant 500$, auprès des passagers, à l'intérieur des wagons.
- Un magot valant 1000$, à l'intérieur de la locomotive, sous la garde du Marshall.

Un Marshall est présent à bord du train et peut se déplacer entre la locomotive et les wagons, en restant toujours à l'intérieur. Il tire sur tous les bandits qui se trouvent à la même position que lui et les force à se retrancher sur le toit.

1. De Christophe Raimbault, chez Ludonaute.

## 1.2 Organisation

L'application sera organisée selon une architecture Modèle-Vue-Contrôleur (MVC), et vous pouvez vous inspirer du fichier Conway.java disponible sur la page du cours. Les différentes sections du sujet vous proposent de mettre progressivement en place les différents éléments du jeu, et suggèrentun ordre dans lequel les traiter. Précisons toutefois qu'il est toujours utile de réfléchir en amont à la manière dont les éléments suivants pourront s'insérer dans votre code.

## 1.3 Un modèle réduit

Un élément central du modèle est le train lui-même, qui sert de plateau de jeu. Déterminez les différents éléments qui le composent (il pourra être utile de les organiser en diagramme de classes) et écrivez les classes correspondantes. Ajoutez une classe pour les bandits et une action de déplacement.

*Note : pour l'instant, on ne s'intéresse pas aux autres actions (braquage, tir), ni aux butins ni au Marshall.*

Quelques précisions :

- Le nombre des wagons est donné par une constante `NB_WAGONS` que vous devez définir dans la classe principale. Vous pourrez par exemple la fixer à `4`.
- On placera pour l'instant un unique bandit à bord du train, sur le toit du dernier wagon. Le nom de ce bandit est donné par une constante `NOM_BANDIT_1`.
- Un déplacement vers l'arrière lorsque l'on est dans le dernier wagon ou en avant lorsque l'on est dans la locomotive n'a pas d'effet. De même pour un déplacement vers le toit ou vers l'intérieur lorsque l'on y est déjà.
- Vous devez écrire des tests unitaires pour les actions des bandits.
- À chaque action du bandit, affichez un compte rendu sur la sortie standard. Par exemple :

```
Wyatt grimpe sur le toit.
Wyatt est déjà sur le dernier wagon.
```

*Indication :* pour définir les directions, vous pouvez utiliser une énumération Java, par exemple

```
enum Direction { AVANT,
                 ARRIERE,
                 HAUT,
                 BAS }
```

On peut alors désigner une direction sous la forme `Direction.AVANT`.

## 1.4 Une belle vue

Pour commencer à voir le modèle s'animer, ajoutez une vue comportant deux parties :

- Un panneau d'affichage dans lequel on voit les différents éléments du train et la position du bandit.
- Un bouton `Action !` qui effectue la prochaine action du bandit.

Quelques précisions :

- Pour l'instant, le bandit n'effectue qu'une suite d'actions prédéterminées (écrites en dur dans le modèle).

— L'affichage doit être mis à jour à chaque déplacement du bandit.
— Continuez à afficher les compte-rendus sur la sortie standard.

*Indications :*

— Vous pouvez soit récupérer les définitions de `Observer` et `Observable` vues en cours soit utiliser celles de la bibliothèque standard. Dans ce deuxième cas : les appels à la méthode `notifyObservers` devront être précédés d'un appel à `setChanged()`.
— Vous pourrez avoir l'usage de nouvelles fonctions de la classe `Graphics`, comme `drawString` ou `drawRect`. Consultez leur documentation.

# 2 Une poignée de dollars

Ajoutez dans le modèle les butins et le Marshall.

Quelques précisions :

— À l'intérieur de chaque wagon se trouvent initialement entre 1 et 4 butins (de type bourse de valeur aléatoire, ou bijou). La nature des butins présents à chaque position doit apparaître sur la vue, mais pas leur valeur.
— Le magot et le Marshall sont placés à l'intérieur de la locomotive.
— Avant chaque action du bandit, le Marshall se déplace avec une probabilité $p$ dans une direction aléatoire (mais toujours à l'étage inférieur, c'est-à-dire à l'intérieur des wagons). La probabilité $p$ est définie par une constant `NERVOSITE_MARSHALL`, qu'on pourra fixer à `0.3`.
— Le bandit a maintenant accès à une nouvelle action : braquage. Cette action lui fait récupérer un butin au hasard parmi ceux présents sur sa position.
— Si le Marshall atteint ainsi la position du bandit, le bandit lâche un des butins qu'il a ramassés tiré au hasard (s'il en a) et se déplace immédiatement vers le toit. Le butin est ajouté à l'ensemble des butins de la position dont le bandit vient d'être chassé, et peut être récupéré à nouveau. Continuez à afficher sur la sortie standard un compte-rendu de tous ces événements.
— Une fois toutes les actions effectuées, affichez le montant total des butins possédés par le bandit.

## 2.1 Tout est sous contrôle

Toujours en considérant un seul bandit, ajoutez maintenant de nouveaux boutons permettant au joueur de planifier les actions de son personnage. Le contrôleur doit alterner entre deux états :

1. Un état de planification, dans lequel le joueur doit donner un certain nombre d'ordres en cliquant sur les boutons correspondants. Le bouton `Action !` est inopérant pendant cette phase.
2. Un état d'action, dans lequel chaque clic sur le bouton `Action !` effectue la prochaine action, les autres boutons étant inopérants.

Quelques précisions :

— Le panneau de commande doit afficher une ligne de texte indiquant quelle est la phase en cours (voir par exemple la classe `JLabel`).
— Le nombre d'actions que doit indiquer le joueur lors d'une phase de planification est une constante `NB_ACTIONS` que vous pouvez fixer par exemple à `4`.

*Indication : le prochain cours donnera quelques éléments pour cette gestion avancée du contrôleur.*

## 2.2 Echanges de plombs entre amis

Étendez maintenant le jeu pour pouvoir y jouer à plusieurs. Une vue supplémentaire devra montrer l'état de chaque bandit, en particulier les butins qu'il porte et le nombre de balles qu'il a encore à sa disposition. C'est le moment également d'ajouter aux bandits l'action de tir.

Quelques précisions :

— Le nombre de joueurs est donné par une constante `NB_JOUEURS`.
— Les joueurs jouent toujours dans le même ordre, laissé à votre discrétion (dans la vue montrant les états des bandits, respecter cet ordre).
— Lors de la phase de planification, chaque joueur à son tour donne tous ses ordres à la fois.
— Lors de la phase d'action, on effectue d'abord toutes les premières actions (dans l'ordre des joueurs), puis toutes les deuxièmes actions (dans le même ordre), et ainsi de suite.
— Un tir est fait dans l'une des quatre directions, qu'on interprète de la même manière que pour les déplacements : en avant ou en arrière au même étage, ou vers le haut ou vers le bas dans le même wagon. Un tir vers le haut (resp. vers la bas) lorsqu'un bandit se trouve sur le toit (resp. à l'intérieur) vise la position occupée par le bandit.
— Le tir touche un bandit tiré au hasard parmi les occupants de la position visée, excepté le tireur lui-même.
— Un bandit touché par un tir lâche l'un de ses butins tiré au hasard. Le butin est ajouté à ceux présents à la position du bandit touché et pourra être récupéré à nouveau.
— Chaque tir utilise une balle. Chaque bandit possède à l'origine un nombre de balles donné par une constante `NB_BALLES`, que vous pourrez par exemple fixer à `6`.
— Le compte-rendu textuel doit également mentionner ces actions de tir et leurs conséquences.

### 2.3 Expression libre

Ajoutez pour finir au moins une fonctionnalité supplémentaire à votre application.
Voici une mini-liste de suggestions d'ajouts. La liste n'est évidemment pas limitative et a simplement pour rôle de vous donner un point de départ.

- Récupérer tous les paramètres du jeu sur la ligne de commande plutôt que d'utiliser des valeurs arbitraires, ou les récupérer dans une fenêtre de dialogue s'ouvrant au début de la partie.
- Gérer plus finement les vues pour ne redessiner que les positions qui ont effectivement été modifiées.
- Lors de la planification, afficher la séquence d'ordres donnée par le joueur courant et lui donner la possibilité de revenir sur ses choix tant qu'il n'a pas cliqué sur un bouton de validation (le bouton `Action !` peut être utilisé pour la validation).
- Introduire des raccourcis clavier pour les différentes commandes du jeu.
- Afficher le compte-rendu de la partie non sur la sortie standard mais dans une zone dédiée de la fenêtre graphique.
- Ajouter une intelligence artificielle pour faire jouer certains bandits par l'application.
- Transformer votre application en une application Android.

## Consignes

Le `README` détaille :

1. Les parties du sujet que vous avez traitées.
2. Vos choix d'architecture.
   *Les choix d'architecture font partie de ce que nous allons évaluer : donnez un peu de détail sur l'organisation de vos classes et le partage des rôles. N'hésitez pas à utiliser massivement les différentes techniques vues en cours et en TP.*
3. Les problèmes qui sont présents et que vous n'avez pas pu éliminer.
4. Les morceaux de code écrits à plusieurs ou empruntés ailleurs.
   *Sans cette information, cela s'appelle du plagiat, et est considéré comme une fraude par le réglement de l'université. On fera une exception pour ce qui est issu des notes de cours ou des corrections de TP.*

Nous allons tester votre application : le code source doit impérativement compiler sans erreur, et au minimum afficher une fenêtre à l'exécution. *Si des morceaux de code ne sont pas finis, vous pouvez les mettre en commentaire. Une méthode non implémentée peut avoir un code vide, ou renvoyer une valeur arbitraire comme* `null`.

Ces deux consignes ( format et contenu du `README`, code source qui compile) sont impératives ; leur non-respect équivaut à un DM non rendu.

L'évaluation sera basée sur la lecture de votre `README`, le test de votre application, et la lecture de passages de votre code choisis par votre encadrant (liés aux points clés du sujet et/ou à ce qu'il/elle aura vu dans le `README` et le test).