Projet de Programmation Orientée Objet
ABIGNOLI Léopold
DEBBAH Martin

# Les colons de Catane

## Page de Settings :

Sur la page d'accueil, lorsqu'on appuie sur « Jouer », cela nous redirige vers la page de configuration de la partie. On peut choisir la taille du plateau (4x4 ou 6x6), le nombre de joueurs (en comptant les IA), le nombre d'IA et les noms des joueurs non-IA.
Dans le terminal, il suffit de se laisser guider par les instructions pour effectuer les mêmes paramètres que précédemment.

## Les routes, villes et colonies :

**Plateau :**
Nous avons créé une classe Board.java dans la package model. L'attribut principal du plateau est un tableau 2D de Case.

**Case :**
Il s'agit d'une classe abstraite. On l'utilise pour pouvoir regrouper tous les types de cases dans un même tableau (MovableCase, Road, ResourceCase…). Toutes ces classes héritent de Case.

**MovableCase :**
Cette classe représente toutes les cases qui peuvent être placées par un joueur (Colony, Town et Road héritent de cette classe). Elle a en particulier un argument Player qui permet de savoir à qui appartient la case. Si l'attribut player est null on considère que la case est vide.

Pour implémenter ces trois éléments, nous avons créé les classes Road, Colony qui héritent donc de MovableCase. Nous avons aussi considéré qu'une ville (Town) est en particulier une colonie. C'est pourquoi Town hérite de Colony.

Pour pouvoir poser un élément, nous allons d'abord vérifier si le joueur est autorisé à le faire. Pour cela on utilise les méthodes canBuildOn() (CanBuildColonyOn(), canBuildRoadOn()…). Ces méthodes sont définies dans la classe Player et prennent en paramètres un plateau, les coordonnées d'une case du plateau ainsi qu'un boolean (early). Ce boolean permet d'autoriser les joueurs à poser un élément gratuitement en début de partie (2 routes et 2 colonies). De même, on réutilise ce boolean pour pouvoir utiliser la carte de développement ***Construction de routes.*** Ces méthodes vérifient plusieurs paramètres. Tout d'abord, pour chaque élément elles vérifient que si le joueur à assez d'argent. Ensuite, on vérifie également que :

**Route :**
- les coordonnées précisées correspondent bien à une case Road vide

**Colonie :**
- les coordonnées précisées correspondent bien à une case Colony vide
- il n'y a pas d'autres colonies aux alentours.

**Ville :**

- les coordonnées précisées correspondent bien à une case Colony
- la case Colony appartient au joueur actuel

Après vérification, on peut donc poser notre élément. Pour cela, on vient exécuter les méthodes putColony(), putTown(), putRoad() de notre plateau (Board). Pour les colonies et les routes, on vient récupérer la case correspondant aux coordonnées passées en paramètre puis on utilise la méthode setPlayer pour préciser que la case est désormais attribuée à un joueur. Pour putTown(), c'est un peu plus subtile. On vient remplacer la case Colony dans le tableau 2D de cases du plateau par une nouvelle case Town (relié directement au joueur qui avait cette case Colony).

CLI → Le joueur sélectionne une option : 'c' (colonie), 'r'(route), 'v'(ville)… On vérifie dans un premier temps si le joueur à l'argent pour pour placer l'élément. Ensuite, on lui demande les coordonnées de la case puis on appelle la méthode canBuildOn correspondante. Des messages d'erreurs sont alors affichés (si besoin) selon la valeur renvoyée par canBuildOn (1, 2…). Si elle renvoie 0, alors on appelle la méthode put (putColony, putRoad…) pour placer l'élément dans le plateau.

GUI → Tout d'abord, nous avons un JPanel GameView qui contient tous les éléments importants d'une partie de catane. Parmi ces éléments, nous avons le BoardView et un ActionPanel (classe interne de GameView).

**BoardView :** Cette classe est un JPanel avec un GridLayout. Elle contient un lien vers le model du plateau (Board). On crée alors un tableau de CaseView correspondant aux cases défini dans Board (model).

**CaseView :** Il s'agit d'un JPanel permettant d'afficher une case à l'écran. Il existe plusieurs dérivés de cette classe (ex : ResourceCaseView, MovableCaseView…). Chaque case contient une référence vers son model. Ces cases sont cliquables.

**ActionPanel :** Nous avons créé une classe ActionPanel situé sur le côté droit de l'écran durant une partie. On peut alors cliquer sur plusieurs boutons radios (colonie, ville, route). Après avoir sélectionné un bouton, seules les cases voulues sont sélectionnables.

Finalement, on peut alors cliquer sur les cases voulues pour poser des éléments. Lors d'un clic sur une case, on vérifie d'abord que le joueur est autorisé à poser un élément grâce à la méthode canBuild(). En cas de problème, rien ne se passe. Sinon, on appelle la méthode put() correspondante dans boardView et on modifie l'apparence de la case. Les méthodes put() dans BoardView rappelle ensuite les méthodes put() du Board pour modifier les cases du model.

**Echange des ressources via les ports :**

Les ports sont représentés sous la forme d'une classe Port qui hérite de Case. On a alors 2 attributs :
- Ressource ressource : C'est la ressource qu'on va devoir fournir pour obtenir une autre ressource.
- int resourcesToGive : C'est le nombre de ressources qu'on va devoir donner.

Si l'attribut ressource est null, alors on peut échanger la ressource de notre choix en X exemplaires contre une autre ressource de notre choix (Ex : port 3:1).

trade() (Player) : Une méthode trade est présente dans la classe Player et permet de proceder à un échange. On précise la ressource à payer, le nombre de ressources à payer ainsi que la ressource voulue en échange.

CLI→ Pendant une partie, il faut sélectionner l'option 'e'. On appelle alors la methode portAction() de CLI. Cette méthode nous demande de choisir parmi la liste de ports. De plus, on a l'option qui permet de faire un échange 4:1 dans tous les cas. Après avoir choisi un port (pour l'option 4:1, un faux port est crée avec une ressource null et un int resourcesToGive valant 4), on appelle la methode makeTrade. Cette methode demande alors :
Si resource == null : de préciser la ressource qu'il veut échanger en X exemplaires
Sinon : on ne demande rien.
Puis on demande la ressource voulu en échange (uniquement si le joueur à assez de ressources pour payer). Ensuite, la méthode trade de Player est appelée sur le joueur actuelle.

GUI → Des cases PortView sont présentent sur le plateau (boardView). Chaque PortView contient un attribut Port vers son model. Lors d'un clic sur un PortView, on ouvre une TradeFrame (classe interne de PortView). Si l'utilisateur n'a pas de colonie à côté de ce port, alors un message d'erreur s'affiche dans la fenêtre. Sinon, on ouvre un ResourceChoicePanel qui permet d'afficher des cartes Resource à l'écran (CardView). On peut alors sélectionner les ressources (en cliquant sur les cartes) puis cliquer sur le bouton « échanger » pour procéder à l'échange.

**Gestion du Score :**

Une classe Score est présente dans le package model. Elle prend un Game et un Player en paramètre. Elle a également une fonction getScore() qui calcule le score du joueur en fonction du nombre de colonies, de villes et de cartes points de victoire. On verifie aussi si le joueur à la route la plus et l'armée la plus puissante grâce aux methodes de Game suivantes : game.longestRoadOwner() et game.mostPowerfulArmyOwner().

**Gestion de l'historique :**

Une classe History est présente dans le package model. Elle hérite d'une ArrayList<String>. Game à un attribut History. A chaque fois qu'une fonction du model est executé, on vient ajouté un nouveau String dans l'historique.

CLI → une fonction cutHistory est présente dans History. Elle permet de créer un nouvelle historique avec uniquement les lignes de « int beginIndex » jusqu'à la fin. On s'en sert notamment pour afficher les actions effectuées par l'IA dans la CLI.

GUI→ On utilise une class HistoryView. Cette classe est un JtextArea qui affiche l'historique à l'écran. On récupère à chaque nouvelle action l'historique enregistré dans la classe Game (model) puis on change le texte de HistoryView.

**Route la plus longue :**

Pour calculer la route la plus longue on utilise la méthode du backtracking (retour sur trace). On utilise un tableau de boolean visited pour faire cela. A chaque fois qu'on visite une route ou une colonie, on met les coordonnées de cet élément dans le tableau visited à true, pour signifier qu'on est déjà passé dessus. Cependant, quand on a fini de calculer un chemin, on remet les booleans des différentes routes et colonies à false.
GetLongestRoadAux() (Board) :
On parcourt le plateau :

Si la route est verticale, on appelle recursivement la fonction sur les trois routes en haut ainsi que les trois routes en bas (seulement si elles n'ont pas déjà été visitées). On prend alors le maximum entre toutes ces valeurs et on ajoute 1 (on compte la case actuelle).
De même si la route est horizontale, on effectue recursivement la fonction sur les trois routes à gauche et les trois routes à droite.
Cependant, si une colonie qui n'appartient pas au joueur est situé sur le chemin alors on ne compte pas ce morceau de route.
Finalement, on a la fonction finale : getLongestRoad(Player player)
On applique alors la fonction getLongestRoadAux sur tous les morceaux de routes du joueur et on prend la valeur maximale. C'est la route la plus longue de player.

**Gérer le voleur en cas de 7 aux dés :**

Chaque case ressource (ResourceCase) contient un attribut « boolean » qui indique si il possède le voleur. Lorsque les dés font sept, cela appelle différentes fonctions. Tout d'abord, une fonction discard() qui permet aux utilisateurs de se défausser de la moitié de leurs cartes ressources, si ils en ont plus de sept.
discard() en CLI -> Une boucle « for » sur la moitié des cartes ressources demande au joueur de se défausser des cartes nécessaires. S'il ne possède pas la ressource, un message d'erreur est affiché.
discard() en GUI -> On crée une nouvelle JFrame avec une liste des joueurs devant se défausser. Pour chaque joueur dans la liste, on ouvre des JPanel permettant de se défausser des cartes jusqu'à que la liste soit vide. On empêche la fermeture de la JFrame pour que le joueur ne puisse pas continuer sans se défausser de ses cartes.

Ensuite, une méthode thiefAction() qui se décompose en deux parties est appelée. Dans un premier temps, elle demande de nouvelles coordonnées pour déplacer le voleur. Cette méthode vérifie que ces coordonnées correspondent à une ResourceCase, puis que cette nouvelle case ne comprend pas déjà le voleur. Si ces deux conditions ne sont pas respectées, on redemande les coordonnées jusqu'à qu'elles soient valides.
En CLI -> Vérifie que les nouvelles coordonnées correspondent à une ResourceCase, puis que cette nouvelle case ne contient pas déjà le voleur. Si ces deux conditions ne sont pas respectées, on redemande les coordonnées jusqu'à qu'elles soient valides.
En GUI -> On utilise des ResourceCaseView qui ont comme attribut une ResourceCase. Seulement les ResourceCaseView sont cliquables, ce qui empêche de cliquer sur une case d'un autre type (RoadView / ColonyView / autres). De plus, la ResourceCaseView contenant le voleur n'est pas cliquable.

Ensuite le voleur est déplacer à l'aide d'une méthode switchThief(int[] coord) dans Board. Cette méthode échange le voleur avec la case de coordonnées coord en supprimant le voleur de l'ancienne case et le mettant sur la nouvelle.

Dans un second temps, thiefAction() récupère une liste contenant les joueurs qui possèdent une colonie autour de la nouvelle case. Puis le joueur actuel peut choisir un joueur parmi la liste (si celle-ci n'est pas vide) pour lui dérober au hasard une de ses ressources.
En CLI -> On demande un joueur parmi la liste tant que le joueur entré par les joueur actuel n'en fait pas partie. Ensuite on appelle player.stealResource(Player p)

En GUI -> On affiche une JFrame avec plusieurs JPanel qui contiennent les différents joueurs de la liste. Le joueur actuel doit cliquer sur l'un d'entre eux pour ensuite activer la méthode player.stealResource(Player p)
player.stealResource(Player p) -> player vole une ressource au hasard à p. Si p n'a pas de ressources, ne fait rien.

Finalement, le jeu reprend son cours.

**Acheter, stocker et utiliser des cartes de développement**

Les cartes de développement sont séparées en trois classes, implémentant DevelopmentCards (pour que chacune puisse être reconnue en tant que carte de développement): Knight pour les cartes chevaliers, Progress pour les cartes progrès et VictoryPoints pour les cartes points de victoire.
Ces différentes classes ne contiennent pas de méthodes, en effet, on effectuera les actions nécessaires en fonction du type effectif de la carte utilisée.
Il existe plusieurs types de cartes progrès et de cartes points de victoire, nous avons donc décidé de faire de Progress et VictoryPoints des Enum.
Progress -> Contient Progress.MONOPOLY pour la carte Monopole, Progress.ROAD_CONSTRUCT pour la carte Construction de routes et Progress.INVENTION pour la carte Invention.
VictoryPoints -> Contient diverses valeurs pour plusieurs constructions (ex : des églises, des universités, etc.)

Stocker les cartes : Chaque objet Player a en attribut une liste de DevelopmentCards à laquelle on ajoute les cartes qu'il pioche.

Acheter les cartes : A la création de la partie, on crée un objet de type DevelopmentCardsDeck qui contient toutes les cartes de développement qui pourront être piochées durant celle-ci. On y ajoute deux cartes de chaque cartes points de victoire, deux cartes de chaque cartes progrès et quatorze cartes chevalier. Le deck est ensuite mélangé, puis lorsqu'un joueur achète une carte de développement il prend la première dans la liste. On vérifie évidemment qu'il a les ressources nécessaires et que le deck n'est pas vide.

Utiliser les cartes :
Les cartes de type VictoryPoints ne sont pas utilisables. Elles ne sont utiles que lors du calcul du score des joueurs.

Les cartes Knight appellent la fonction thiefAction() qui permet de déplacer le voleur et de voler une carte ressource au hasard à un des joueurs possédant une colonie autour de la nouvelle case contenant le voleur. (voir Gérer le voleur en cas de 7 aux dés)
Ensuite la méthode game.refreshMostPowerfulArmy() est appelée. Elle augmente le nombre de chevalier du joueur actuel de un et, si il a au moins trois chevaliers et plus de chevaliers que celui qui détient l'armée la plus puissante (ou si aucun ne détient ce titre), le désigne en tant que tel.

Les cartes Progress.MONOPOLY permettent d'avoir le monopole sur une ressource choisie.

CLI -> On demande une ressource au joueur qui la joue, puis on appelle une fonction game.monopoly(Resource r) qui prend toutes les ressources r de tous les joueurs et les donne au joueur actuel.
GUI -> On ouvre une JFrame contenant un JPanel ou l'utilisateur peut choisir une ressource. Ensuite la même méthode game.monopoly(Resource r) est appelée.

Les cartes Progress.ROAD_CONSTRUCT donnent de constructions de routes gratuites au joueur.
CLI -> Une boucle « for » permet de demander des coordonnées et de poser deux routes.
GUI -> Lorsque la carte est utilisée une fonction constructRoad() de GameView est appelée. Un « int » constructRoad est alors initialisé à un, puis à chaque appel de constructRoad() il est incrémenté de un. S'il est inférieur à deux, il nous reste des routes à poser. Tous les boutons sont donc désactivés pour empêcher le joueur d'effectuer d'autres actions que poser des routes. S'il est supérieur à deux, alors les boutons sont réactivés et construcRoad est réinitialisé à zéro, ce qui permet de sortir de la boucle.
Dans RoadView, si gameView.getConstructRoad() renvoie un chiffre supérieur ou égal à 1, on rappelle la fonction constructRoad() de GameView (si c'est égal à zéro, on est pas en train d'utiliser une carte de développement).
Que ce soit en CLI ou en GUI, les fonctions putRoad() prennent le boolean early en true pour que le joueur n'ait pas à payer.

Les cartes Progress.INVENTION permettent au joueur de piocher deux cartes ressource de son choix.
CLI -> Une boucle « for » qui lui demande deux ressources et les lui fait piocher.
GUI -> Une JFrame avec deux JPanel permettant chacun de choisir une ressource.
Ensuite, dans les deux cas on utilise la fonction player.gainResource(Resource r), qui fait piocher la ressource au joueur, pour les deux ressources choisies.

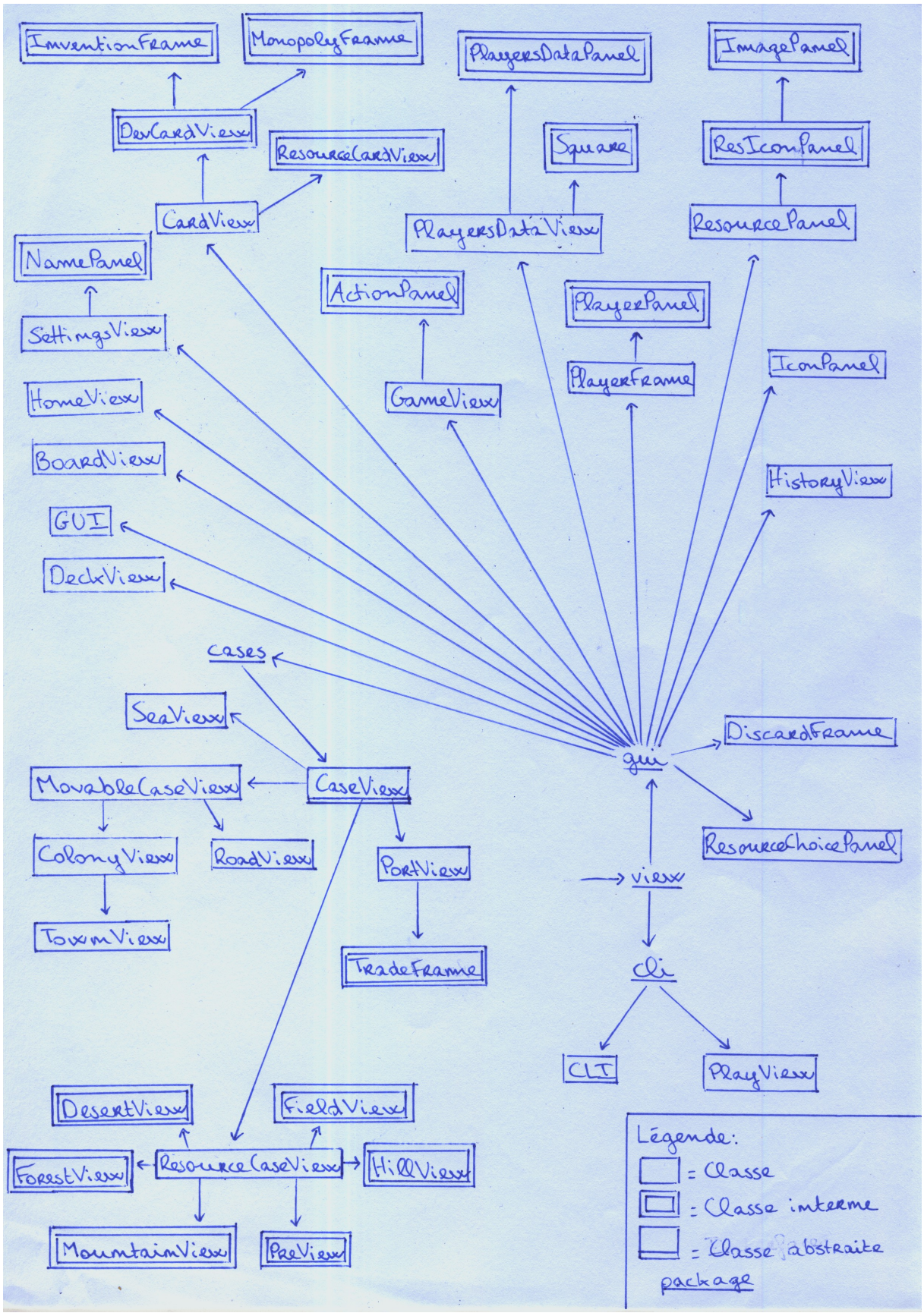
## Intelligence Artificielle

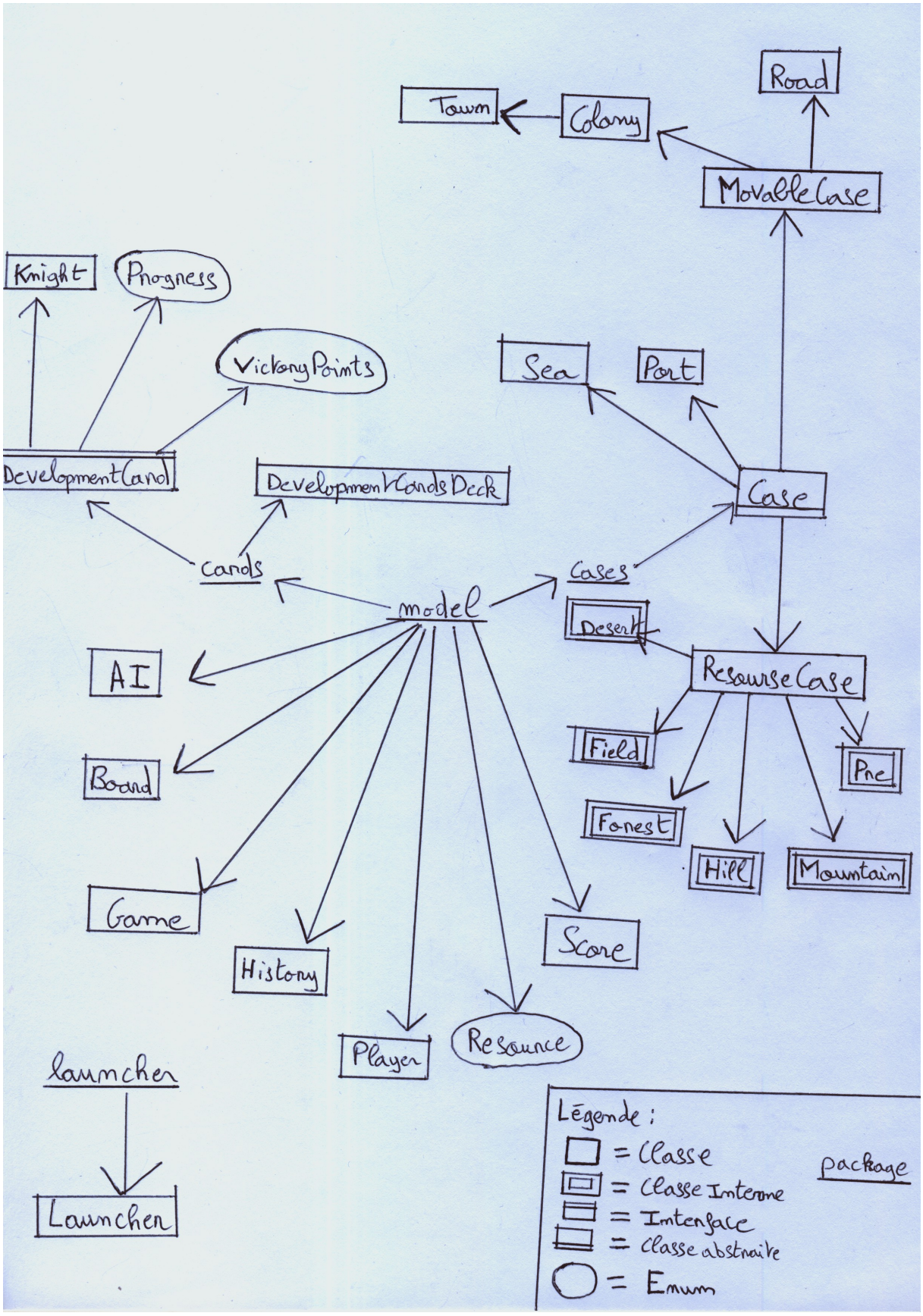
Au début de la partie, la méthode earlyGame(game) est appelée pour que l'IA pose deux colonies et deux routes.

En cours de partie, lorsque c'est au tour d'une IA, la méthode midGame(game) est appelée. Une fonction wantsTo() lui permet de faire un choix binaire. Grâce à cette fonction on lui demande d'abord s'il veut piocher une carte de développement. Puis s'il veut en utiliser une. Les dés sont ensuite lancés. S'il a les ressources nécessaires il essaie de poser une colonie. Ensuite, il a le choix entre poser une route, poser une ville, piocher une carte de développement, utiliser une carte de développement ou faire un échange. Pour effectuer ce choix, on utilise un Random. Si l'IA ne peut pas effectuer l'action choisit, il passe son tour.

Pour choisir quelle case choisir, une fonction pour chaque type de case fait appel à la fonction findFreeCase(MovableCase c, Player p) de Board. Cette fonction retourne une liste des cases disponibles de même type que c. (pour Colony et Road les cases doivent être libre, pour Town elles doivent avoir le même joueur que celui passé en paramètres et pour ResourceCase elles ne doivent pas avoir le voleur). Ensuite la liste est mélangée puis l'IA prend la première case de la liste.
Pour les choix de ressources (par exemple les ports ou Progress.MONOPOLY) une fonction askResourceMin() permet de retourner la ressource que l'IA a en le moins

d’exemplaires. Cette fonction appelle askResourceMin(Resource r) avec comme paramètre null. askResourceMin(Resource r) a le même comportement que la première, seulement si une ressource est entrée en paramètre, il ne peut pas la retourner. (c’est cette fonction qui est utilisée dans les ports pour éviter d’échanger une ressource contre la même) Il existe aussi une fonction askResourceMax() qui retourne la ressource que l’IA a en le plus d’exemplaires. Cette fonction est utilisée lorsque l’IA doit se défausser de cartes ressources. (par exemple dans la fonction discard() du voleur)


InventionFrame
MonopolyFrame
PlayersDataPanel
ImagePanel
DevCardView
ResourceCardView
Square
ResIconPanel
CardView
PlayersDataView
ResourcePanel
NamePanel
ActionPanel
PlayerPanel
SettingsView
PlayerFrame
IconPanel
HomeView
GameView
BoardView
HistoryView
GUI
DeckView
cases
SeaView
DiscardFrame
gui
MovableCaseView
CaseView
ResourceChoicePanel
ColonyView
RoadView
PortView
view
TownView
TradeFrame
cli
CLI
PlayView
DesertView
FieldView
ForestView
ResourceCaseView
HillView
MountainView
PreView
Légende:
= Classe
= Classe interne
= Classe abstraite
package


Road
Town
Colony
MovableCase
Knight
Progress
VictoryPoints
Sea
Port
DevelopmentCard
DevelopmentCardsDeck
Case
cards
cases
model
Desert
AI
ResourseCase
Field
Board
Pre
Forest
Hill
Mountain
Game
Score
History
Player
Resource
launcher
Launcher
Légende :
= Classe
= Classe Interne
= Interface
= classe abstraite
= Enum
package