ZEYNALI Kyan
ZHOU SEBASTIEN

Projet POO3 Catane

I – Introduction

Pour réaliser notre projet, nous nous sommes dans un premier temps réparti les tâches importantes. Pour cela nous nous sommes basés sur nos préférences dans le programme de l'année en POO. En effet, Kyan a tout de suite était intéressé par l'implémentation des fonctionnalités de base, comme le plateau et l'interface homme-machine pour la version sur terminal. Quant à Sebastien, il a bien plus aimé les interfaces graphiques et s'est donc concentré sur ça.

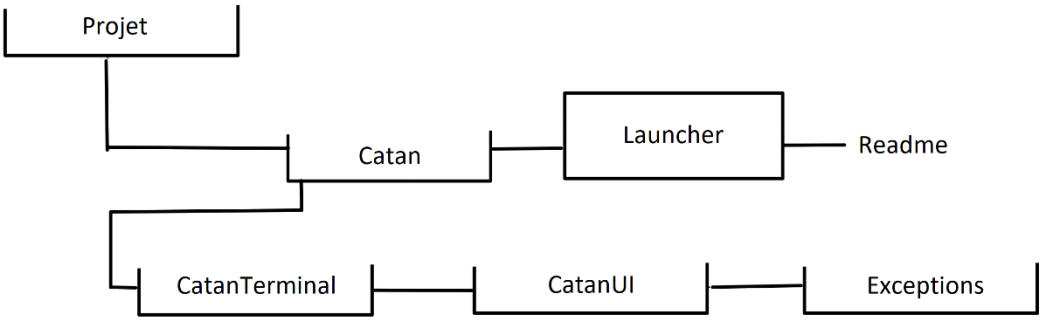
Pour travailler en binôme de manière efficace, nous avons utilisé git pour coordonner nos travaux et éviter qu'il y ait des conflits entre les versions de chacun.
En effet, nous avons utilisé le site : https://gaufre.informatique.univ-paris-diderot.fr qui est un domaine de GitLab qu'on a également utilisé dans la matière conduite de projet. De ce fait, on a décidé d'utiliser cet outil comme la plupart des développeurs professionnels pour se mettre dans des conditions de travail réalistes.
L'historique des commits de notre dépôt est disponible à la page 3 de ce rapport.
Ce dernier était comme un rapport temporaire de notre travail, car nous avons écrit ce rapport uniquement une fois que notre projet était complétement terminé.

En ce qui concerne le cahier des charges, nous l'avons respecté dans son intégralité.
A la base, on avait même prévu d'implémenter la négoce (commerce intérieur). Mais on n'a pas pu programmer les échanges entre joueurs par manque de temps.
Cependant concernant les règles du jeu, nous avons décidé de tout implémenter sauf une règle : la règle de la distance entre deux colonies. En effet, nous avons codé le plateau 4x4 qui est fourni dans le sujet (figure 2). Nous avons un moment même songé à laisser le joueur choisir entre un plateau 4x4 et 5x5, avec case aléatoires. Cependant, pour éviter de créer de nouveaux bugs et par manque de temps, nous avons abandonné cette idée.
Par conséquent, sur un plateau 4x4, la règle de distance entre deux colonies est irréaliste d'un point de vue gameplay, car avec cette taille ce sera donc presque impossible pour le joueur de construire de nouvelles colonies, surtout s'il y a 4 joueurs qui jouent. Il s'agit vraiment de la seule règle qu'on n'a pas implémenter dans notre version du Catane.

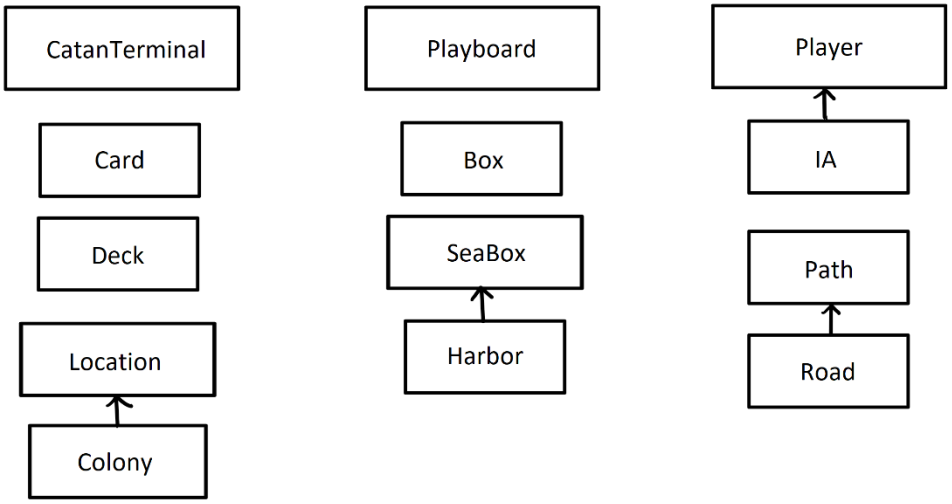
Ensuite, il y a trois autres règles qu'on a un peu modifier, dans le but d'améliorer le gameplay de notre version :

- Comme avec des tuiles carrées, une intersection peut être en contact avec 4 cases au maximum, on a décidé que lorsque que le joueur place sa deuxième colonie, durant la phase initiale du jeu, il n'obtiendra pas 4, mais seulement 3 ressources des terrains adjacent à sa colonie. En effet, on a décidé de « nerfer » cette règle, pour éviter que ce soit trop facile pour le joueur de construire dès le début.
- En raison de la complexité algorithmique du processus de la route la plus longue, nous avons décidé de la calculer dès le début pour mieux tester notre programme. Bien évidemment, le joueur ayant la route la plus longue sera actualisé en temps réel (la fonction est appelée au tour de chaque joueur).
- Enfin, pour la carte développement point de victoire, on a décidé de ne pas cacher les points du joueur courant gagnés avec cette carte, car c'est complétement inutile dans une version multijoueur locale (sur une même machine).

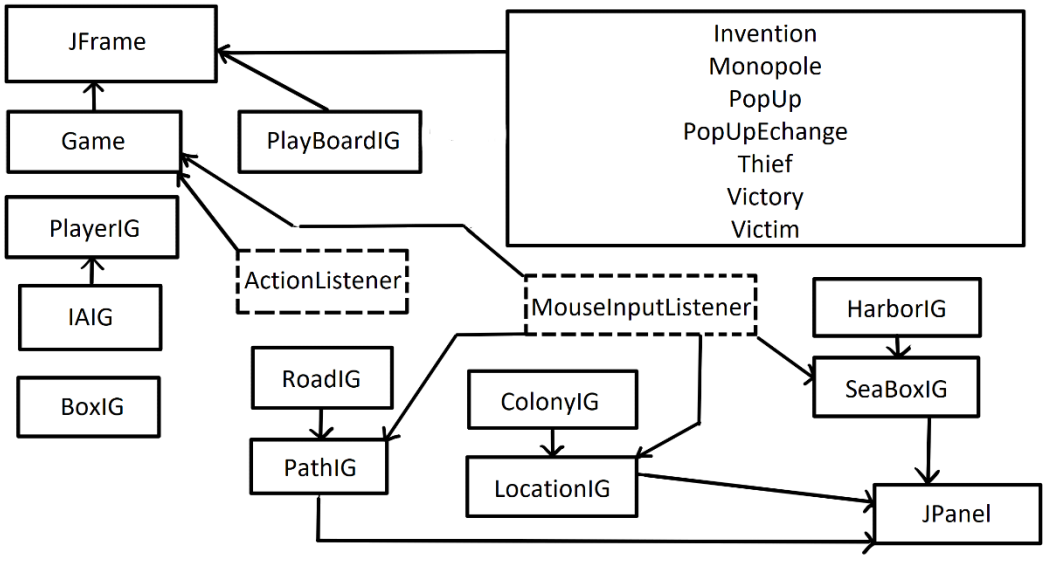
Hiérarchie des packages et des classes

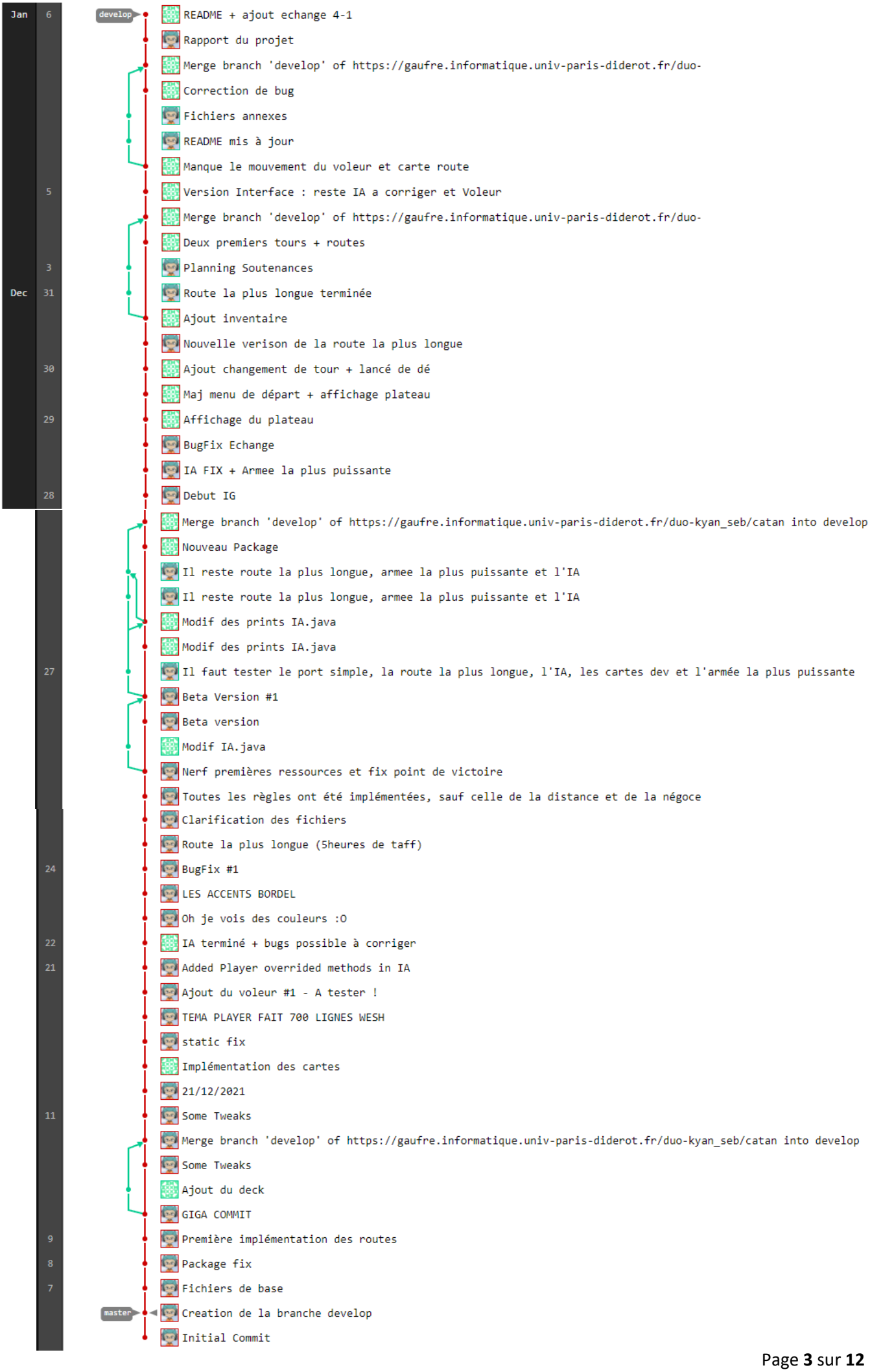


Package CatanTerminal



Package CatanUI




Jan 6
develop
README + ajout echange 4-1
Rapport du projet
Merge branch 'develop' of https://gaufre.informatique.univ-paris-diderot.fr/duo-
Correction de bug
Fichiers annexes
README mis à jour
Manque le mouvement du voleur et carte route
5
Version Interface : reste IA a corriger et Voleur
Merge branch 'develop' of https://gaufre.informatique.univ-paris-diderot.fr/duo-
Deux premiers tours + routes
3
Planning Soutenances
Dec 31
Route la plus longue terminée
Ajout inventaire
Nouvelle verison de la route la plus longue
30
Ajout changement de tour + lancé de dé
Maj menu de départ + affichage plateau
29
Affichage du plateau
BugFix Echange
IA FIX + Armee la plus puissante
28
Debut IG
Merge branch 'develop' of https://gaufre.informatique.univ-paris-diderot.fr/duo-kyan_seb/catan into develop
Nouveau Package
Il reste route la plus longue, armee la plus puissante et l'IA
Il reste route la plus longue, armee la plus puissante et l'IA
Modif des prints IA.java
Modif des prints IA.java
27
Il faut tester le port simple, la route la plus longue, l'IA, les cartes dev et l'armée la plus puissante
Beta Version #1
Beta version
Modif IA.java
Nerf premières ressources et fix point de victoire
Toutes les règles ont été implémentées, sauf celle de la distance et de la négoce
Clarification des fichiers
Route la plus longue (5heures de taff)
24
BugFix #1
LES ACCENTS BORDEL
Oh je vois des couleurs :O
22
IA terminé + bugs possible à corriger
21
Added Player overrided methods in IA
Ajout du voleur #1 - A tester !
TEMA PLAYER FAIT 700 LIGNES WESH
static fix
Implémentation des cartes
21/12/2021
11
Some Tweaks
Merge branch 'develop' of https://gaufre.informatique.univ-paris-diderot.fr/duo-kyan_seb/catan into develop
Some Tweaks
Ajout du deck
GIGA COMMIT
9
Première implémentation des routes
8
Package fix
7
Fichiers de base
master
Creation de la branche develop
Initial Commit

Répartition des tâches :

Kyan :

- Launcher.java -> main
- CatanTerminal.java -> éléments principaux de la version terminal
- PlayBoard.java -> plateau
- Box.java -> Tuile terrain
- SeaBox.java -> Case Maritime
- Harbor.java -> Port
- Location.java -> Intersection du plateau
- Colony.java -> Colonie d'un joueur
- Path.java -> Chemins du plateau
- Road.java -> Route d'un joueur
- Player.java (sauf les fonctions des cartes développement) -> Joueur
- Tous les fichiers du package Exceptions

Sebastien :

- Player.java (fonctions des cartes développement)
- IA .java -> Comportement de l'IA
- Card.java -> Carte développement
- Deck.java -> Tas de cartes du jeu
- Tous les fichiers du package CatanUI

II – Plateau et éléments du jeu

Plateau :

En toute évidence, nous avons modélisé le plateau en premier. Au début, le plateau (classe PlayBoard) contenait un tableau de cases (classe Box), un tableau de tableaux d'intersections (classe Location) et deux tableaux de tableaux de chemins (classe Path), un pour les chemins verticaux et un pour les chemins horizontaux. On a par la suite ajouté un tableau de cases maritimes pour les ports et une case contenant le voleur.

Ainsi, on a créé dans un premier temps la classe Box, définie par son type (Colline, Champs, Montagne, …), la ressource qu'elle produit (Argile, Blé, Roche …), par son numéro pour le lancer de dés et ses index dans le tableau de PlayBoard. Plus tard on a ajouté un booléen dans le constructeur, qui est true si la case contient le voleur (cf. Classe Player).
Ensuite, on a créé la classe Location, qui modélise chaque intersection du plateau. Chaque intersection est adjacente à 1, 2 ou 4 cases ; ces dernières sont stockées dans un tableau de cases. Elles sont également collées à 2, 3 ou 4 chemins (arrêtes des cases), stockés aussi dans un tableau. Elles ont enfin des index uniques. Plus tard, on a ajouté une fonction qui permet de déterminer s'il existe un port à proximité d'une intersection et ainsi le récupérer si ce dernier existe. On a rajouté également des fonctions pour le processus de la route la plus longue. Cette classe est une base pour les colonies.
Après on a implémenté les chemins dans la classe Path, qui par définition relie deux intersections et possède une direction (verticale ou horizontale). Cette classe sera très utile pour la conception des routes placées par les joueurs.
Enfin on a modélisé les cases maritimes dans la classe SeaBox, qui sont chacune liée à deux emplacements (intersections). Cette classe va nous servir de base pour les ports.

Chacune des classes citées précédemment possède une méthode toString() bien particulière, qui permet d'afficher le plateau de manière claire dans le terminal.
En effet, la plupart des fonctions du plateau sont appelées dans le constructeur, et vont construire le plateau (basé sur celui du sujet). On construit d'abord les cases, puis les intersections, ensuite les cases maritimes, ensuite les ports et enfin les chemins.
Sinon la fonction importante de cette classe est display() : l'affichage du plateau sur le terminal. C'est probablement la fonction qu'on a le plus souvent testé. Elle était longue à écrire, mais pas difficile non plus. Dans un premier temps on l'affichait à l'état vide (sans pions dessus). Ensuite on a rajouté les ports, les cases maritimes les pions des joueurs, ainsi que des couleurs ASCII pour rendre l'affichage plus beau et compréhensible par l'utilisateur. Les constantes String des couleurs ASCII sont définies dans la classe principale CatanTerminal et fonctionnent en plus sur n'importe quel système d'exploitation.

Colonies/Villes/Routes/Ports :

Les classes Colony (colonie), Road (route) et Harbor (port) héritent respectivement de Location, Path et SeaBox. La particularité de ces éléments, c'est qu'ils appartiennent tous à un joueur spécifique. Dans les deux versions, ces derniers auront une couleur spécifique correspondant à celle du joueur (sauf les ports qui sont un cas particulier car ils peuvent appartenir à deux joueurs, vu qu'on n'a pas inclus la règle de distance entre deux colonies).

Colony est une sous-classe de Location, car le joueur va construire une colonie sur une intersection. En plus de son joueur, une colonie a un booléen qui définit si c'est une ville. En effet il est inutile de créer une classe pour les villes, car elles ont le même comportement que les colonies. Le point de victoire supplémentaire obtenu lors de la construction d'une ville sera donné dans sa fonction de construction dans la classe Player.

Road est une sous-classe de Path, car le joueur va construire une route sur un chemin. Une route a une intersection de départ et une intersection d'arrivée. En effet ces deux paramètres sont fondamentaux car le joueur peut seulement construire une route à côté d'une de ses colonies ou d'une de ses routes. Des fonctions ont par la suite été rajoutées pour le processus de la route la plus longue, notamment une qui détermine si la route courante est liée à une autre route du même joueur.

La classe Harbor hérite de SeaBox, car chaque port est lié à deux intersections. De plus chaque port peut être simple ou spécial. Chaque port spécial a pour paramètre une ressource spéciale. La difficulté a été de différencier les ports du plateau. Pour faciliter la distinction entre les différents ports, nous avons rajouté un id pour éviter toute confusion. Ainsi chaque port a un prix, les ports spéciaux ont un rapport d'échange de deux mêmes ressources spécifiques contre une et les ports simples de trois mêmes ressources quelconques contre une. On note que sur le terminal, les ports simples sont notés : PS ? et le port spécial de bois par exemple sera noté : PBo. Les cases maritimes sont notées par des vagues ~~~ et ne possède rien de spécial.

Cartes développement et tas de cartes :

L'implémentation des cartes a été faite en deux parties : d'abord la classe Card qui contient les cartes qui possèdent un nom et un id pour les identifier facilement puis une classe Deck qui est une liste de carte. Le deck contient le même nombre de cartes que dans le jeu classique et cette liste est mélangée grâce à une fonction de la bibliothèque « Collection ».

Ce deck est initialisé sur le plateau (PlayBoard) afin qu'il soit commun à tous les joueurs. Ainsi chaque joueur peut acheter une carte à partir de ce deck.

Toutes les méthodes liées aux cartes sont contenues dans la classe Player et ces cartes ont les mêmes propriétés que celles du jeu classique. Cependant, pour la carte « Point de victoire » nous avons décidé d'omettre le fait qu'elle soit cachée car cela n'a pas d'intérêt en local.

De plus on ne peut pas utiliser une carte qu'on vient d'acheter, elle est donc stockée dans une liste d'attente puis est déplacée dans l'inventaire des cartes à la fin du tour.

III – Fonction principale du jeu

Quand dans le main du launcher l'utilisateur choisi de joueur à la version terminal, java va exécuter la fonction principale du jeu catan() dans la classe CatanTerminal, ou la fonction catan() dans la classe Game si le joueur a choisi la version avec interface graphique.
Dans les deux cas, la vue va demander le nombre de joueurs, si ces derniers seront des humains ou des robots, ainsi que le nom de chaque joueur humain.
Une fois les joueurs configurés, notre programme va créer un nouveau plateau vierge et une nouvelle pile de cartes développements. Au début du jeu, personne n'a la route la plus longue ou l'armée la plus puissante.
Ici, la difficulté a été l'organisation de notre code. En effet, à la base, les variables fondamentales de la partie, à savoir le tableau de joueurs, le plateau et la pile de cartes étaient des variables locales de la fonction catan(). Cependant cela n'était pas pratique pour leur visibilité dans les autres classes. C'est pour cette raison qu'on a mis ces variables très importantes en champs statiques et final.

Une fois que java a initialisé les variables de base du jeu, il exécute la fonction principale catan(). Au début on commence par la phase initiale, avec le placement des colonies et routes de base. Ensuite on entre dans le jeu, qui n'est en fait rien de plus qu'une boucle qui s'arrête quand un joueur a gagné au moins 10 points de victoire. Cette boucle va répéter une autre boucle qui fait jouer chaque joueur et met à jour la route la plus longue ainsi que l'armée la plus puissante. La fonction principale fait des appels à 7 fonctions de la classe Player (qui modélise les joueurs) :
buildColony (true), buildRoad (true, true), gainInitialRessources(), longestRoad(), isWinner(), toString() et bien sûr play() (cf. Classe Player).

Une fois qu'un joueur a gagné la partie, la boucle s'arrête et le gagnant est félicité et un classement est affiché. De plus, le nombre de tours est également affiché pour donner une estimation de la longueur de la partie.

Lorsque la vue va demander une action au joueur, notre programme va s'attendre à un certain type d'action ou de réponse de sa part (le tout dans un try and catch). Si par exemple ce dernier ne tape pas une entrée valide dans la version terminale (qui utilise des scanners) alors notre code va envoyer des exceptions, et tant que ce dernier ne donnera pas une entrée valide, alors la vue va refaire sa requête sans qu'il n'y ait de conséquence dans le modèle. En effet, on a tout simplement mis un try and catch dans une boucle infinie (while (true)), qui s'arrête quand aucune exception n'a été capturée.

Ainsi nous avons créé des exceptions dans le package exception pour prévenir certains cas :

- InexistantColonyException -> Colonie inexistante dans un cas où on en a besoin, exemple : lorsque le joueur veut construire une route
- InexsitantRoadException -> Route inexistante dans un cas où elle est nécessaire, exemple : lorsque le joueur veut construire une colonie
- InvalidNameException -> Apparaît quand le joueur n'a pas entré de nom, tous les noms des joueurs doivent contenir au moins un caractère
- NotEnoughRessourcesException -> Apparaît quand le joueur veut utiliser un port, mais que ce dernier n'a pas les ressources nécessaires pour procéder à l'échange.
- WrongInputException -> Concerne les mauvaises actions ou entrées de l'utilisateur.

Bien évidemment, toutes ces exceptions personnalisées héritent de la classe Exception de l'API java. On a également utilisé les exceptions de base de l'API java, comme IndexOutOfBoundsException pour les index incorrects.

IV – Classe Player

Dans la classe Player, nous avons modélisé le joueur, en y implémentant tous ses éléments en guise d'attribut : nom, pion, couleur, points de victoire, inventaire de ressources, routes construites, cartes développement achetées, colonies construites, ports colonisés et nombre de chevaliers joués.
Nous avons décidé de notre plein gré de mettre la plupart des méthodes importantes dans cette classe, car dans la conception du jeu, on est parti du principe que c'est le joueur qui joue quand c'est à son tour, que c'est lui qui lance les dés, et que c'est lui qui va potentiellement construire une route, etc. Ainsi par exemple la fonction pour construire une route sera dans la classe Player, et non dans la classe Road.
En partant de ce constat-là, Player contient la majorité du fonctionnement du jeu, mais malgré le fait que cette classe contient plus de 1000 lignes, elle est basée sur seulement deux fonctions, qui sont appelées dans la fonctions principale catan() de la classe CatanTerminal ou Game . Ces deux fonctions sont isWinner() et play().
isWinner() va juste vérifier si le joueur courant a gagné la partie. On tient à dire qu'au début de notre projet, on avait créé une interface PlayerAction, qui contenait ces deux méthodes. Cependant on a finalement décidé de la supprimer à cause de la visibilité des méthodes ; on ne voulait pas qu'elles soient publiques, mais de visibilité package.

Dans les premières versions de notre code, quasiment toutes les fonctions de la classe Player avait en argument un plateau p. En effet, la majorité des fonctions (et donc actions du joueur) vont modifier le plateau, et donc ce dernier doit être mis à jour régulièrement.

Exemple : void constructColony(PlayBoard p)

Au début on a modélisé comme ça, car pour nous le joueur (this) va construire une colonie (new Colony) sur la plateau du jeu, (qui sera toujours mis en argument). Cependant, il y a une ambiguïté avec cette méthode. En effet, on peut en théorie mettre n'importe quel autre plateau en argument, ce qui pourrait rendre le fonctionnement du jeu instable.
Dans l'objectif de rendre notre code plus sécurisé, nous avons créé un champ statique constant dans la classe CatanTerminal ou Game, qui fait référence au seul et unique plateau du jeu : static final PlayBoard PLAYBOARD = new PlayBoard() ;

Ce plateau unique et constant nous a permis de lever cette ambiguïté concernant le plateau et nous a aussi permis de retirer le plateau en argument des fonctions de la classe Player, et ainsi rendre nos signatures de méthodes plus concises. Ainsi dès qu'une méthode a besoin d'invoquer le plateau de la partie, on écrit : CatanTerminal.PLAYBOARD ou GAME.PLAYBOARD (selon la version).

A) Fonctions du jeu

La fonction principale du joueur est play(). Cette dernière va annoncer le tour du joueur, et attendre une action du joueur pour lancer les dés. Ensuite le résultat du lancer est affiché et sauf si le résultat est 7, la distribution de ressources sera effectuée selon le numéro des tuiles de terrain. Si c'est le 7 qui tombe aux dés, alors les fonctions du voleur sont appelées (cf. Fonctions du voleur). Ensuite, si le joueur possède assez de ressources, il pourra faire une action spéciale, comme acheter une carte développement par exemple. S'il ne peut rien faire, alors il est obligé de passer son tour. Si le joueur peut faire quelque chose, alors il sera entrainé dans un menu d'actions dans la version sur terminal. Dans l'interface graphique, il pourra faire ce qu'il souhaite et ce qu'il peut en cliquant sur les bons boutons. Pour résumer, c'est cette fonction qui va appeler d'autres fonctions, qui vont elles aussi appeler d'autres fonctions, etc. C'est la première fonction à être appelée de la classe.

La fonction statique throwDices() va simuler un lancer de dés, en renvoyant un entier aléatoire entre 2 et 12.

La fonctions statique earnRessources(int dice) va répartir les ressources selon le résultat du lancer des dés (cf. commentaires du code source).

B) Fonctions du voleur

Comme beaucoup de nos camarades, le voleur a été l'un des aspects les plus complexes à implémenter du projet. En effet, il s'agit déjà d'un des seuls éléments qui n'a pas été traité de la même manière dans la version sur terminal et celle sur fenêtre. En premier lieu, il nous paraissait évident de créer une nouvelle classe pour le voleur, qu'on a gardé pour l'interface graphique (cf. Interface Graphique). Cependant pour la version sur terminal, on l'a finalement supprimé pour au final baser le concept du voleur sur quatre fonctions dans la classe Player, un attribut pour la case contenant le voleur dans l'implémentation du plateau, et un booléen dans la classe box. On note que la fonction statique earnRessources(int dice) n'est pas appelée dans le cas du 7 aux dés, et c'est la fonction thief() qui est appelée (cf. commentaires du fichier Player.java).

Parmi ces quatre fonctions on a :

- void thief() : fonction qui va appeler les trois autres selon certains conditions
- void giveRessources(int n) : fonction qui va demander la moitié des ressources du joueur, quand ce dernier a au moins 8 ressources en tout dans son inventaire
- Box moveThief() : fonction qui va déplacer le voleur sur une autre case du plateau et renvoyer la nouvelle case choisie par le joueur
- Player selectPlayerToStealFrom(Box b) : si la case b en argument possède des colonies autour d'elles, alors le joueur va choisir chez qui il va voler une ressource au hasard. Elle renvoie la victime choisi par le joueur.
- void steal(Player Victim) : fonction qui va voler une ressource au hasard au joueur en argument, et la donner au joueur courant (this).

C) Menu principal

Lorsque le joueur a la possibilité de construire, acheter ou utiliser des cartes, il pourra alors effectuer l'action possible de son choix. Sur terminal, il va devoir taper explicitement au clavier en majuscules une des actions proposée dans le menu. Sur l'interface graphique, il aura juste à cliquer sur le bouton correspondant à l'action qu'il souhaite faire.
En ce qui concerne l'implémentation des actions, on a dû faire beaucoup de modifications dans la fonction play(). En effet dans la version sur terminal, la fonction playerMenu() n'existait pas au début. On avait d'abord écrit les fonctions de proposition (cf. ligne 425 de Player.java), qui propose au joueur de faire telle action quand cette dernière est faisable, en répondant par oui ou par non.
Chacune des actions ci-dessous possède sa fonction de proposition dans la classe Player :

- Faire un échange 4 :1 avec la banque (commerce maritime)
- Utiliser un port (commerce maritime)
- Construire une colonie (construction)
- Construire ville (construction)
- Construire route (construction)
- Utiliser une carte (carte développement)
- Acheter une carte (carte développement)

Ainsi dans la fonction play(), on avait appelé successivement chaque fonction de proposition dans le même ordre que la liste ci-dessus. Cependant, on s'est rendu compte plus tard que notre conception n'était pas bonne. En effet avec cet algorithme, le joueur ne pouvait pas effectuer les actions dans l'ordre de son choix ; l'ordre était imposé par celui des appels des fonctions de proposition. Pour résoudre ce problème, nous avons décidé de créer un menu pour la version sur terminale, où le joueur peut jouer tout simplement comme il le souhaite. Ce menu est modélisé par la fonction void playerMenu(). De plus nous avons tout de même gardé les fonctions de propositions. En effet malgré l'existence du menu, ces dernières s'avèrent utiles pour permettre à l'utilisateur de revenir en arrière si ce dernier change d'avis, car elles demandent la confirmation du joueur avant d'effectuer l'action sélectionnée.

Dans l'Interface Graphique, on a eu moins de problèmes grâce aux boutons, notamment le bouton annuler pour permettre au joueur de revenir sur ces pas. L'avantage de l'IG est le fait qu'on puisse afficher le plateau, les données du joueur et le menu des actions en même temps, le tout sur une même fenêtre (cf. Interface Graphique).

D) Fonctions de construction

Hormis la construction de ville, qui va juste mettre à jour un booléen, les fonctions de construction sont basées sur les attributs du plateau. On va donc effectivement jouer sur les index du tableau d'intersections de la classe PlayBoard pour placer une colonie. De même, pour les routes, on va placer cette dernière dans le tableau de routes du plateau correspondant à sa direction (horizontale ou verticale).

Pour la construction de colonies ou de villes, nous n'avons pas eu de difficultés notables. Cependant, on a dû rajouter un booléen en paramètre pour faire comprendre à notre programme si la colonie sur le point d'être construite sera « gratuite ». En effet les deux colonies que placent les joueurs sur le plateau au début du jeu durant la phase initiale ne leur coûte rien en termes de ressources.

Cependant, la construction de route nous a bien plus causé de problèmes.
Tout d'abord on a essayé de modéliser la construction dans une seule fonction, mais on

s'est rendu compte que cela était trop compliqué. En effet il y a deux cas de figure en ce qui concerne la construction de routes : le joueur peut construire une route à côté d'une de ses colonies ou à côté d'une de ses autres routes.

Donc une fois que le joueur aura tapez des coordonnées valides dans la version sur terminal ou quand il aura cliqué sur un endroit valide dans l'interface graphique, notre programme va d'abord essayer de construire cette route à côté d'une colonie. S'il s'avère que ce n'est pas la bonne situation, alors il va ensuite essayer de placer la route à côté d'une autre route du joueur. On a évidemment réussi à implémenter cet algorithme grâce à une quantité colossale de try-catch, if-else et d'exceptions.

En résumé, la construction de route est modélisée en trois fonctions (cf. code source de Player.java et lignes 46 à 50 de CatanTerminal.java).

- void buildRoad(boolean isFree, boolean beginning)
- Road buildRoadNextToColony(char c, Path selectedPath, boolean beginning)
- Road buildRoadNextToRoad(char c, Path selectedPath)

E) Route la plus longue

Après les nombreuses heures passé dessus, on peut déclarer avec certitude que la route la plus longue est le problème le plus intéressant d'un point de vue algorithmique.
En effet, après chaque tour la fonction principale du jeu va appeler la fonction longestRoad() de sa classe (CatanTerminal ou Game). Cette méthode met à jour les points de victoire et renvoie le joueur qui possède la route la plus longue. Elle renvoie null en cas d'égalité. Ainsi, chaque joueur possède également une méthode longestRoad(), qui renvoie la longueur de leur plus grande combinaison de routes sur le plateau.

On rappelle que chaque joueur possède une liste de routes, qui contient celles qui ont construite sur le plateau. Ainsi la fonction longestRoad() de la classe Player va exécuter sur chaque route construite par le joueur la fonction récursive suivante :
calculateLongestRoad(Road road, ArrayList<Road> crossedRoads)

Explication des paramètres :
La route road, est la route de départ de notre calcul. Avant le premier appel de la fonction, elle est ajoutée dans la liste préalablement initialisée crossedRoads.

La fonction getLinkedRoads() va renvoyer une liste contenant les routes du joueur qui sont collées à la route road. Si aucune route n'a été trouvée, ou si une colonie adverse bloque le passage, alors elle renvoie null et on return 1 (cf. Road.java).

Si une seule route a été trouvée, alors on ajoute 1 et on recommence le processus pour la route suivante (récursion classique). Pour éviter que la fonction revienne sur des routes déjà parcourues, on ajoute chaque nouvelle route parcourue dans la liste crossedRoads.
Si jamais avec getLinkedRoads() on revient sur une route qu'on a déjà parcouru (et qui est donc dans la liste) alors on return, sinon on aura une erreur de stack overflow.
Si plusieurs routes ont été détectées, alors on va rajouter 1 et réappeler la fonction pour chaque route non-parcourues, le tout dans un tableau d'entiers (la fonction renvoie un int).
Une fois que toutes les possibilités auront été calculées, on a qu'à retourner la plus grande valeur du tableau, grâce à la fonction statique getMax(int[] tab) : qui renvoie la valeur maximale d'un tableau d'entiers.

Pour résumer, on va prendre chaque route construite par le joueur comme route de départ de la fonction récursive, pour ainsi parcourir toutes les combinaisons de routes possibles, et ainsi renvoyer celle qui a la plus grande longueur.

V – L'IA

L'IA se trouve dans la classe IA qui hérite de la classe Player vue avant. Son fonctionnement est simple, elle a les mêmes méthodes et attributs qu'un vrai joueur cependant, toutes les méthodes sont réécrites selon un système d'aléatoire. Tout le système se trouve dans les méthodes play() et playerMenu().

La méthode play() lance le tour de l'IA et fait tous les actions qu'un joueur et censé faire puis le choix des actions sont fait dans playerMenu() .

La méthode playerMenu() ajoute tous les actions possibles de l'IA en fonction de ses ressources dans une liste d'actions. On peut voir si ces actions sont possibles grâce aux méthodes booléennes héritées de la classe Player. Si la liste est vide, l'IA passe son tour. Sinon, une des actions est choisie aléatoirement puis appelle la méthode associée à l'action choisie. Cependant, l'IA a également une chance sur trois de passer son tour. Enfin, si l'action est bien exécutée, la fonction est rappelée récursivement pour que l'IA puisse possiblement faire plusieurs actions dans son tour.

Ainsi, l'IA ne peut pas faire d'action impossible ce qui permet d'éviter des bugs.

VI – Commerce maritime

Durant la partie, le joueur va pourvoir faire plusieurs type d'échanges.
En effet, les échanges 4 :1 avec la banque et les ports simples (échange 3 :1) sont similaires ; ils vont demander 3 ou 4 ressources quelconques de même nature au joueur en échange d'une ressource de son choix. Cependant, les ports spéciaux vont demander 2 ressources spécifiques de même nature (2 bois par exemple) au joueur en échange d'une ressource de son choix. Rien qu'en relisant ce paragraphe, on se rend compte que tous les échanges se ressemblent et peuvent être implémentés dans un seule fonction :

void exchange(int n, String ressource)

Explication des paramètres :
n désigne le prix de l'échange (n est donc forcément égal à 2, 3 ou 4).
Si l'échange ne concerne pas une ressource en particulier (comme dans le cas du port simple ou de l'échange 4 :1), alors ressource sera null en argument.
Par contre dans le cas des ports spéciaux, la ressource du port sera mise en argument.

Quand on appelle cette fonction avec ressource=null, le joueur devra choisir l'une de ses ressources dont le nombre est supérieur au prix de l'échange. Ensuite, la fonction va se réappeler (sans pour autant qu'il y ait de récursion) en mettant la ressource choisie par le joueur en argument.

Si une ressource est spécifiée lors de l'appel, alors notre programme enlève automatiquement n ressource, avec ressource = le type de ressource en entrée.

Dans tous les cas, le joueur doit indiquer en dernier lieu, la ressource qu'il souhaite obtenir en échange.

VII – L'interface graphique

L'implémentation de l'interface graphique s'est faite en plusieurs étapes. Tout d'abord, l'affichage du plateau sans aucune action qui s'est avérée très longue à faire car il y a beaucoup d'élément à prendre en charge. C'est pourquoi l'utilisation d'une extension d'Eclipse appelée « WindowBuilder » qui permet de mettre les éléments directement sur une interface et qui crée le code automatiquement. Grâce à cela, il était plus simple d'avoir une vue d'ensemble de la vue du plateau. La majorité des classes du package CatanUI sont les mêmes classes que celles de la version terminale et reprennent le même fonctionnement que celles-ci. Les classes qui sont totalement identiques telles que les classes Card et Deck sont importés du package CatanTerminal mais les autres sont réécrites ou légèrement modifiées selon les besoins pour le fonctionnement de l'interface graphique. Le code de l'interface est réparti en deux grosses classes principales : Game et PlayBoardIG.

- La classe Game sert de contrôleur et presque toutes les actions possibles sont gérées dans cette classe. Il y a également le menu principal qui se trouve dans cette classe.
- La classe PlayBoardIG sert de vue et de plateau de jeu, la majorité de ce qui se passe dans le jeu sera pris en charge dans cette classe.

Tout d'abord l'affichage des cases, des routes et des ports qui sont créés en même temps et dans les mêmes méthodes que les cases, les routes et les ports dans la version terminale. Ainsi ces éléments héritent désormais de JPanel afin de pouvoir interagir avec plus tard.

Puis, l'affichage d'un menu principal afin de pouvoir décider du nombre de joueur, d'IA et du nom des joueurs. Lorsque les données sont récupérées le menu se ferme pour laisser la place au plateau du jeu.

Ainsi pour la plupart des actions possibles du jeu il y a un bouton attribué directement disponible sur le plateau. Sauf pour les cartes qui a un système de pop-up pour voir ses cartes, acheter et utiliser.

A cause de notre petit plateau, la place était limitée c'est pourquoi les pop-ups ont été d'une grande aide pour gérer la place. De nombreuses fonctionnalités les utilisent :

- Les cartes comme dit précédemment qui fait appel à la classe PopUp
- Les échanges 4-1, les échanges avec les ports qui font appel à la classe PopUpEchange
- La carte Monopole qui fait appel à la classe Monopole
- La carte Invention qui fait appel à la classe Invention
- Le voleur qui fait appel à la classe Thief

Le voleur qui n'est pas implémenté à 100% dû au manque de temps, sur l'interface graphique car on ne peut pas le déplacer mais il vole les ressources aux joueurs ayant au moins huit ressources au total, lorsque le dé tombe sur 7.

Le chevalier ne peut donc pas déplacer le voleur mais l'Armée la plus puissante fonctionne tout de même. Néanmoins l'IA peut déplacer le voleur mais il n'est pas affiché.