**Projet POO3 Catane**

ZEYNALI Kyan ZHOU SEBASTIEN

**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. 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 fournit 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.

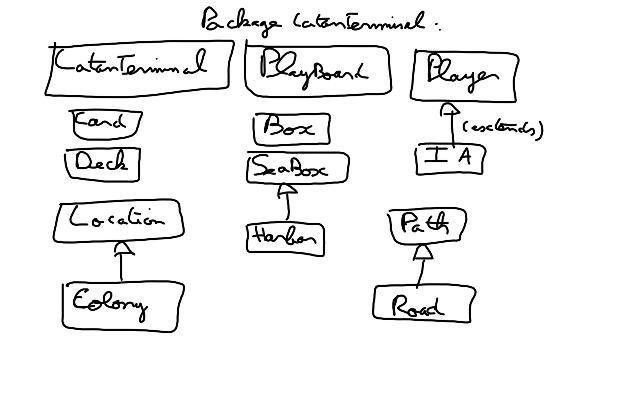
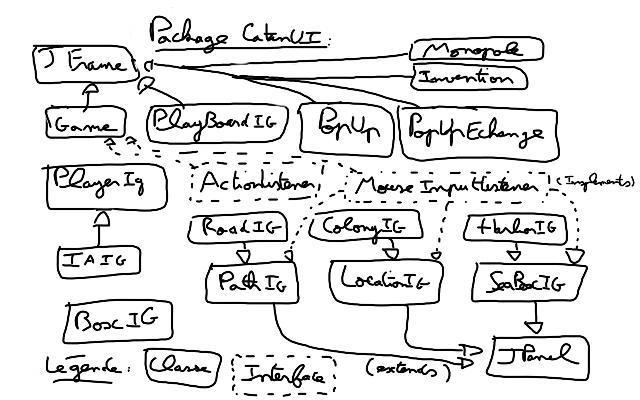
Ensuite, il y a trois autres règles qu’on a un peu modifier, dans le but d’améliorer le gameplay de notre version :

* 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).

Une image contenant texte

Description générée automatiquement

**Hiérarchie des packages et des classes :**



**Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquementRépartition 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 – Le Plateau et les 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. 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 simples 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 même ressources spécifiques contre une et les ports simples de trois même 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 – La fonction principale du jeu et exceptions**

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 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 de base 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 sur play(). Voir la section IV pour plus d’informations sur ces fonctions importantes.

Une fois qu’un joueur à 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 entrée de l’utilisateur.

Bien évidemment, toutes les exceptions personnalisées qu’on a implémenté hérite 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 – La classe Player**

**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**

**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 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.

Choses à mettre :

* Déroulement du dév
* Idées non implémentées
* Problèmes/difficultés