Les Colons de Catane

Introduction

Le projet Colons de Catane réalisé par l'équipe numéro 75, formée par Paris Mollo et Jade Cortial, représente une version simplifiée mais loyale au célèbre jeu de plateau Catane.

Catane est un jeu de stratégie et de négociation, où les joueurs assument les rôles des colons, chacun essayant de construire et de développer des actifs tout en échangeant et en acquérant des ressources. Les joueurs gagnent des points de victoire au fur et à mesure que leurs établissements grandissent; les premiers à atteindre un nombre déterminé de points de victoire, généralement 10, gagnent la partie.

Notre version du jeu Catan, permet à 3 ou 4 joueurs de jouer un match complet de Catane par une interface graphique interactive, ou par une interface textuelle. Le jeu offre aux joueurs des options de configuration d'un match de catan, comme la taille du tableau, le nombre de joueurs, entre autres. En plus de permettre à chaque joueur, par les différentes fonctionnalités du jeu, de choisir la stratégie qui leur convient le plus pour gagner la partie.

Comme il s'agit d'un travail relativement complexe, qui a une échéance et un grand nombre de tâches à accomplir, nous avons décidé de nous organiser et de travailler en utilisant certaines pratiques de développement qui nous ont aidés durant cette période de travail.

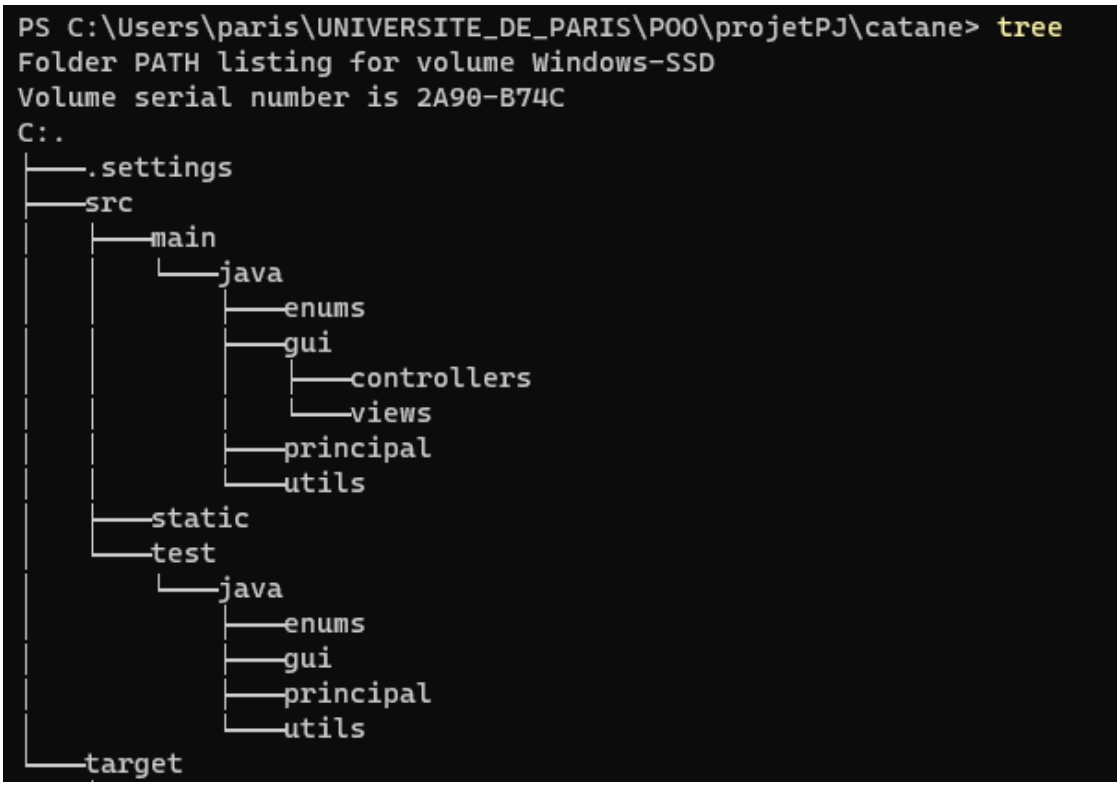
Dès le début du développement, nous avions souhaité pouvoir produire un logiciel de qualité, qui satisfasse à certaines exigences de base d'un bon software. C'est-à-dire un logiciel maintenable (capable d'évoluer), sûr (fonctionne comme prévu) et acceptable (adapté aux utilisateurs) et c'est pourquoi nous avons décidé d'utiliser quelques pratiques du type:

- Agile (Plan > Design > Develop > Review).
- Gestion de version (Github et le système d'issues).
- Test Driven Development.
- Maven pour l'organisation structurale du projet.

Malheureusement, nous n'avons pas complètement atteint les objectifs que nous avions prévus. Cependant, avec ces défis, nous avons beaucoup appris, nous allons donc partager en détail toutes les difficultés et les solutions possibles, et pourquoi nous n'avons pas eu le temps de les mettre en place.

Architecture du projet

Nous allons maintenant vous montrer la structure générale du projet, en d'autres termes son arborescence.



Pour mieux comprendre le fonctionnement et le rôle de chaque package du projet, nous allons expliquer en quelques mots leur importance dans l'ensemble du projet.

Dossiers:

1. *Main*
2. *Static*
3. *Test*

Dans le dossier "**main**" nous trouvons tous les composants principaux du projet, le package "**gui**", "**enum**", "**principal**", et "**utils**". Dans le dossier "**static**", nous avons tous les fichiers statistiques, comme les images et les polices pour le texte de l'interface. Dernier point, mais non des moindres, nous avons le dossier "**test**", où tous les tests unitaires sont effectués pour s'assurer que le backend du projet fonctionne de la manière souhaitée.

Packages:

1. *enums*
2. *gui*
3. *principal*
4. *utils*

Les packages contiennent la partie principale du projet, là, vous trouverez toutes les classes du projet et leurs méthodes respectives. Nous allons maintenant faire une description générale de leur rôle dans le projet.

‹ package: enums ›

Afin d'éviter une création exhaustive de classes pour définir tout type d'action, terrain ou type de production, qui, à leur tour, n'aurait pas eu beaucoup de contenu ou méthodes. Nous avons décidé d'utiliser les enums de Java, cette classe spéciale qui nous permet de représenter des constantes que nous utiliserons tout au long du développement du jeu. Ces constantes sont utilisées dans la création du tableau, tuiles, croisement, cartes de développement et beaucoup d'autres exemples.

‹ package: utils›

Le package "utils" à son tour, contient tout type de classe qui est utilisé comme outil auxiliaire pour les autres classes, c'est-à-dire tout type de message sur la console qui doit être montré de façon colorée, ou calculs qui doivent être réalisés pour connaître un résultat, entre autres exemples qui ne sont pas nécessairement liés au

jeu de catan mais qui sont importants pour le fonctionnement de notre projet dans son ensemble.

‹ package: gui›

Le package "gui" contient tous les composants graphiques de notre projet. Dans ce package, nous avons les classes qui représentent les frames, les panels et leurs composants. En outre, chaque classe possède un controller correspondant, responsable de la communication du modèle (package: principals) avec l'interface. Dans ce package, il y a une classe principale qui représente la fenêtre principale de l'interface graphique, cette fenêtre et son contenu est régulièrement mis à jour, en fonction du déroulement du jeu.

‹ package: principal›

Le package principal est là où se trouvent les définitions conceptuelles et fondamentales du jeu de catan. Toutes les classes concernant les principaux composants du jeu, comme le joueur, le tableau, les croisement et les tuiles sont représentées dans ce package. Dans ce package nous trouvons les définitions de toutes les règles du jeu que nous avons définies et nous trouvons aussi les fonctions relatives aux impressions sur la console pour la version du jeu textuelle.

MVC

Comme indiqué dans les cours de *POO3* et dans les *TP 10* et *11* de POO3, nous essayons au maximum de respecter le design pattern MVC (model-view-controller), c'est-à-dire toute la logique du jeu, ses définitions et ses modèles sont complètement indépendant de l'interface graphique. Nous utilisons les "controllers" qui se trouvent dans le package "gui" pour "modifier" les modèles (en fonction de l'action/input de l'utilisateur) et donc ensuite chaque controller indique sa "view" respective de se mettre à jour en fonction du nouveau contenu des modèles.

Fonctionnement et Fonctionnalités

Maintenant que nous connaissons la structure générale du projet, nous pouvons entrer dans quelques détails du fonctionnement et comprendre le rôle de chaque classe dans le jeu. Nous utiliserons des représentations graphiques pour simplifier la

compréhension de quelques classes, mais nous couvrirons aussi certaines de ses fonctions ici.

Dans notre version du catane vous pouvez :

1. Créer un plateau de différentes tailles.
2. Jouer avec 3 ou 4 joueurs.
3. Choisir les noms, les types (le type IA ne joue pas encore tout seul) et les couleurs de chaque joueur.
4. En fonction de nos ressources on a un menu d'actions qui s'adapte pour nous proposer que ce qu'on peut faire (et non pas des actions impossibles)
5. Construire des "routes".
6. Construire des "colonies".
7. Construire des "villes".
8. Jouer le jeu à travers le terminal de votre ordinateur.
9. Jouer le jeu via une interface graphique.
10. Lancer les dés.
11. Passer votre tour.
12. Positionner le voleur sur le plateau.
13. Acheter et utiliser la carte du chevalier.
14. Faire du commerce sans port.
15. Visualiser vos points et fonctionnalités tout au long de la partie.

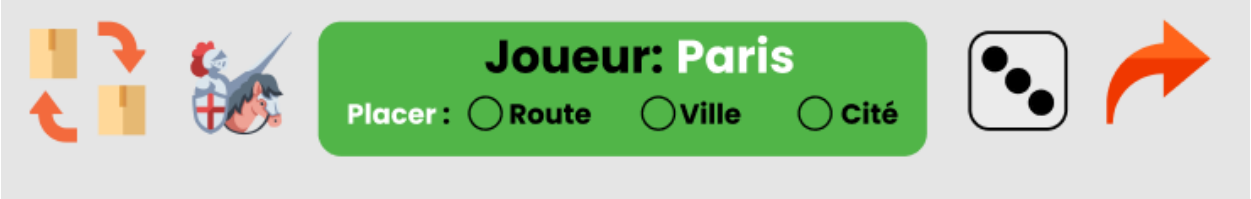
Package principal:

Pour le "package principal" il se décompose en 2 classes fondamentales :
Jeu et Joueur.
D'un côté il y a la classe Jeu qui contient l'Aire de jeu (le plateau) qui contient lui-même les tuiles et les croisements.
D'un autre côté, il y a la classe joueur qui permet d'utiliser les règles du jeu et de jouer.
Ce modèle représente une partie dans la vraie vie avec sur une table le jeu avec ses objets qui lui sont reliés (Tuile, Croisement, Route, ...) et les joueurs autour de la table ce qui nous a permis de ne pas se triturer l'esprit et de suivre ce modèle logique pour la conception de nos classes.

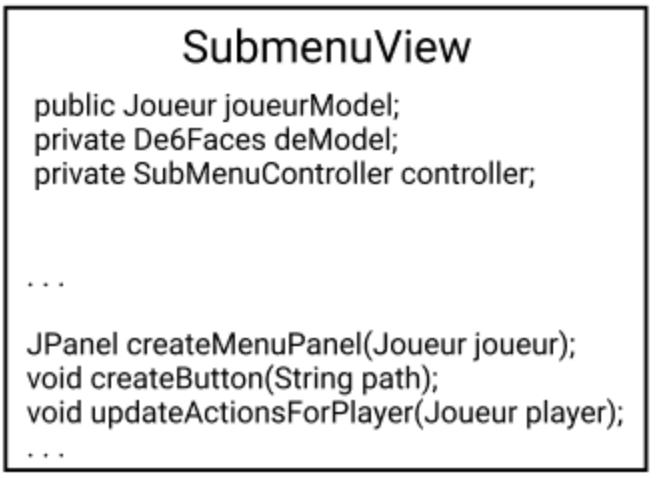
Package gui:

Toutes les classes du "package gui" suivent la même nomenclature et la même structure. Les classes responsables de la création de composants graphiques sont les classes qui se trouvent dans le dossier "gui.views", tous les fichiers sont appelés "[nom_classe]View.java", à l'exception de la classe Window.java, tous représentent "JPanels" dont le contenu est spécifique en fonction de son rôle dans l'interface graphique. La classe Window.java (Jframe) est qui décide quel JPanel ajouter à son contentPane et à quel moment.

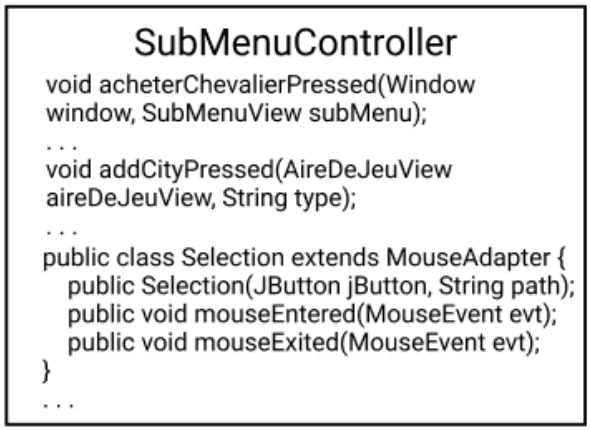
Pour illustrer le fonctionnement et les méthodes d'une classe, nous prenons par exemple la view "Submenuview.java". Cette classe est responsable pour définir tous les buttons et leurs dispositions respectives dans l'interface de notre jeu. Dans cette classe, nous avons 4 buttons, un autre JPanel qui possède lui-même 4 autres buttons.



La classe SubMenuView.java possède un ensemble de méthodes qui définissent la création des chaque composant, leurs layouts, paramètres, listeners, entre autres, que vous trouvez dans l'image ci-dessus. La classe a comme attributs des constantes utilisés dans la création de ses plusieurs composants et aussi des attributs relatifs aux modèles nécessaires pour son affichage, comme le modèle Joueur et aussi un attribut qui fait référence à son controller, par exemple:



Les controllers sont responsables pour contenir les définitions des Actionlisteners et MouseListeners des composants qui se trouvent dans les views, en plus de tout type d'action qui demande une mise à jour de la part du modèle et ensuite par conséquent de la vue correspondant. Par exemple:



Problèmes connus et pistes de solutions

Lors du développement de notre projet, nous avons rencontré quelques défis liés à la mise en œuvre de certaines fonctionnalités, autant que dans la partie des règles du jeu, à la fois dans l'interface graphique.

Cependant, si cela n'était pas limité dans le temps, nous pensons qu'en raison de la modularité et de l'architecture de notre projet et de nos classes, nous n'aurions aucun problème à mettre en œuvre les fonctionnalités que nous n'avons pas eu le temps de mettre en œuvre complètement. Certaines d'entre elles sont disponibles dans le code source mais n'ont pas été ajoutées au jeu car nous n'avons pas réussi à résoudre certains bugs ou à les tester suffisamment pour éviter tout problème pendant le départ d'une partie.

Certains des changements qui pourraient être faits pour améliorer notre code, serait la simplification de certaines classes, et peut-être créer des fonctions encore plus "génériques" qui ont moins de dépendances avec les types de variables et les classes. (Utilisation potentielle de classes génériques et interfaces).

Un exemple de problème que nous n'avons pas eu le temps de résoudre, mais je pense que nous pouvons le résoudre, serait d'annuler une action faite par le joueur via l'interface. En d'autres termes, avant de passer le tour, le joueur a la possibilité d'annuler sa dernière action, comme par exemple placer une ville dans un certain emplacement dans le tableau. Une solution possible serait d' avoir une variable qui garde la référence de la dernière tuile sélectionnée et le type d'action qui a été lancé (cette type de référence est déjà présent dans le code), à partir de là, il suffirait de mettre à jour le modèle, de sorte que nous supprimons cette ville du "inventaire" du joueur courant, en utilisant la référence stockée, et mettrons à jour à nouveau la vue de la tuile, cette fois, sans la ville, qu'on avait ajouté mais est désormais supprimé car l'action a été annulée.
Cependant cela n'a pas été explicitement demandé, c'est une fonction qui apparaît évidente quand on joue au jeu. En mode console nous aurions pu implémenter une confirmation de l'action mais cela aurait été plus lourd lors du jeu.

Conclusion

Pour finir nous allons revenir sur deux points importants :

- En raison de la sophistication du projet nous avons utilisé les techniques du Test Driven Development. A chaque création d'objet et nouvelle fonctionnalité on commençait d'abord par écrire les tests (qui ne marchaient pas car il n'y avait pas de code écrit) puis écrire le code jusqu'à ce que les tests deviennent verts puis à refactorer si besoin (refactorer du code rend parfois les tests rouges donc il faut par la suite éventuellement les modifier etc) .

TEST ÉCRIT SANS CODE (méthodes vides) -> CODE ÉCRIT POUR QUE LES TESTS SOIT VALIDE -> REFACTO

Utiliser le Test Driven Development nous a permis d'avancer au début certes de manière un peu lente car la confection des tests prenaient autant de temps voire davantage que la confection du code en lui-même mais de manière sûre car pas de régression. Utiliser cette pratique permet au fur et à mesure de l'avancée du projet de ne pas se retrouver bloqué et d'avoir peur de tester de peur que ça casse le projet et que l'on ne sache pas déceler pourquoi ça ne marche pas.
Les tests sont une partie vivante de notre projet qui prend une place importante car elle nous sécurise de toute casse éventuelle.

- Pour ce qui est de l'IA nous estimons avoir couvert 50% de sa conception :
  En effet nous avons pensé tout notre projet dans la vision qu'un ordinateur puisse y jouer de la manière suivante :
  Quand un utilisateur joue il lance les dés qui permettent de mettre à jour les ressources de chaque joueurs. Puis vient le moment de faire des actions et c'est là que vient notre réflexion pour concevoir l'IA. Les actions qui nous sont proposées sont dynamiques. C'est-à-dire qu'elles s'adaptent en fonction de nos ressources pour nous proposer des actions que l'on peut forcément faire. De plus, ces actions se présentent sous la forme d'une liste d'entiers, ce qui permettrait à l'IA de pouvoir sélectionner au hasard une action qu'elle pourrait forcément faire. Or la véritable complexité est plutôt de déterminer comment elle doit agir à l'intérieur de l'action. Comme exemple si on avait fait une IA bête, dès le début de la partie elle aurait pu se retrouver bloquée car il y aurait

pû y avoir la possibilité qu'elle réussisse à positionner une route qui est valide mais trop proche de sa première colonie et n'aurait donc jamais pû poser sa deuxième colonie. La conduite de notre projet est dans l'optique d'y intégrer une IA mais par manque de temps nous n'avons pas voulu livrer une IA incomplète qui peut se retrouver en incapacité de continuer la partie dû à ses actions.