Projet : Le Tic-Tac-Toe et ses variantes.

Module : Bases de la programmation orientée objet.

IUT de l'Université Paris Descartes

DUT Informatique - Année Spéciale

Enseignant : Denis Poitrenaud

Auteur : Adrien Jallais

Date d'échéance : 12/01/20

Table des Matières

Introduction

La programmation orientée objet (*POO*), permet une programmation plus sure (en encapsulant des données) et plus rapide (en promouvant la réutilisation du code existant).

Afin de nous en approprier les fondamentaux, une application de parties de TicTacToe (*TTT*, n°**1**) et ses dérivés ont été codées en Java. Les objectifs principaux étaient leurs fonctionnalités et la qualité de leur hiérarchie.

Le principe du TTT est le placement de symboles alignés au sein d'une grille initialement vide.

Les extensions réalisées sont :

- le TTT avec plusieurs alignements : le Morpion (*M*, n°**2**),
- le TTT avec une forme particulière au choix (*TF*, n°**3**),
- le Morpion avec la permutation des symboles d'une grille remplie aléatoirement (*PM*, n°**4**).

Pour jouer à celles-ci, il faut indiquer leur **numéro** au sein des paramètres d'exécution de l'*Appli.java*. Un menu interactif a été mis en place pour personnaliser la taille de la grille ou celle des alignements (n°**0**).

Ce dossier présente cette application avec les points suivants :

I. les diagrammes UML des classes (brute puis simplifié),

II. les difficultés rencontrées et les réflexions associées,

III. les améliorations possibles,

IV. le respect de la POO par cette application,

V. les extensions possibles,

VI. le code de celle-ci et ses tests.

I. Diagramme UML des classes

Les diagrammes page suivante, présentent une organisation ascendante des différentes parties de cette application. On y observe en amont des interfaces, permettant le lancement des applications par l'*Appli.java*. Celles-ci sont implémentées par des classes abstraites dont *CA_Grille_Partie* définissant le déroulement commun des parties. C'est cette dernière que les parties concrétisent.

Ces diagramme ont été générés avec *ObjectAid UML Explorer*. Les relations entre ses entités étant ajoutées de manière automatique, il est rapidement devenue surchargé et illisible (*Illustration 1*). Afin d'améliorer sa visibilité, les relations entre les packages au profit de celles des classes ont été favorisées (*Illustration 2*).

Par exemple, comme les classes concrètes du package *partie* utilisent toutes les classes des packages *jetons*, *direction*, il a été résumé ces utilisations par une relation entre *partie* et ceux-ci.Les messages du package *interaction* ont été crées afin d'aider l'utilisateur. Cependant les consignes du projet contraignant la diversité des messages, leurs sorties ont donc été limitées, mais ceux-ci sont toujours utilisables pour de possibles extensions.

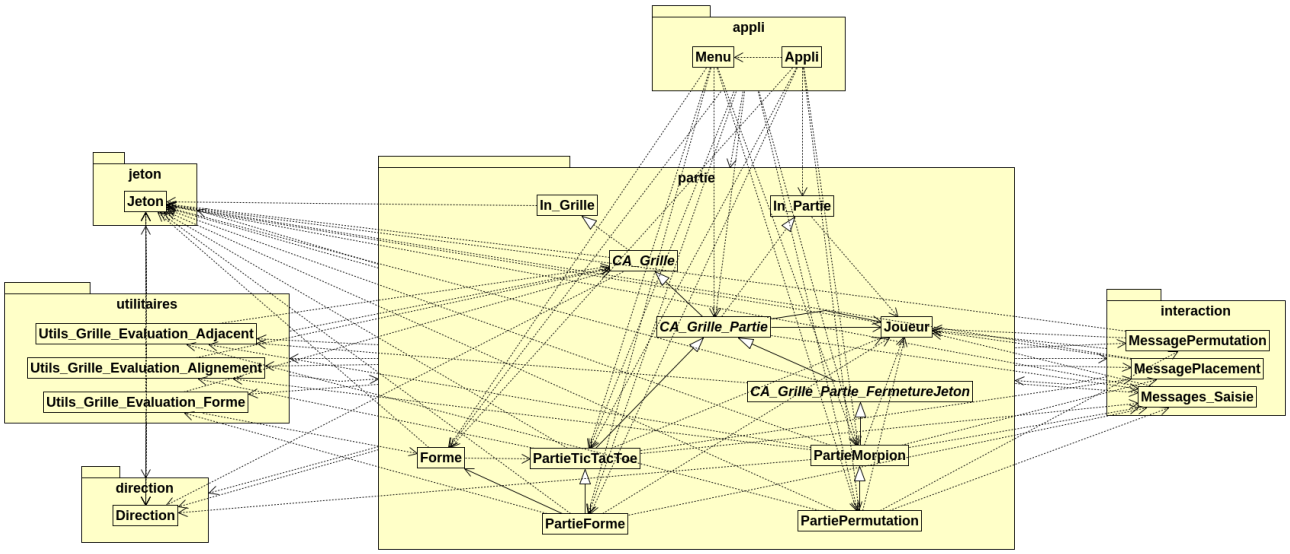

Illustration 1: Diagramme UML des classes, version brute.

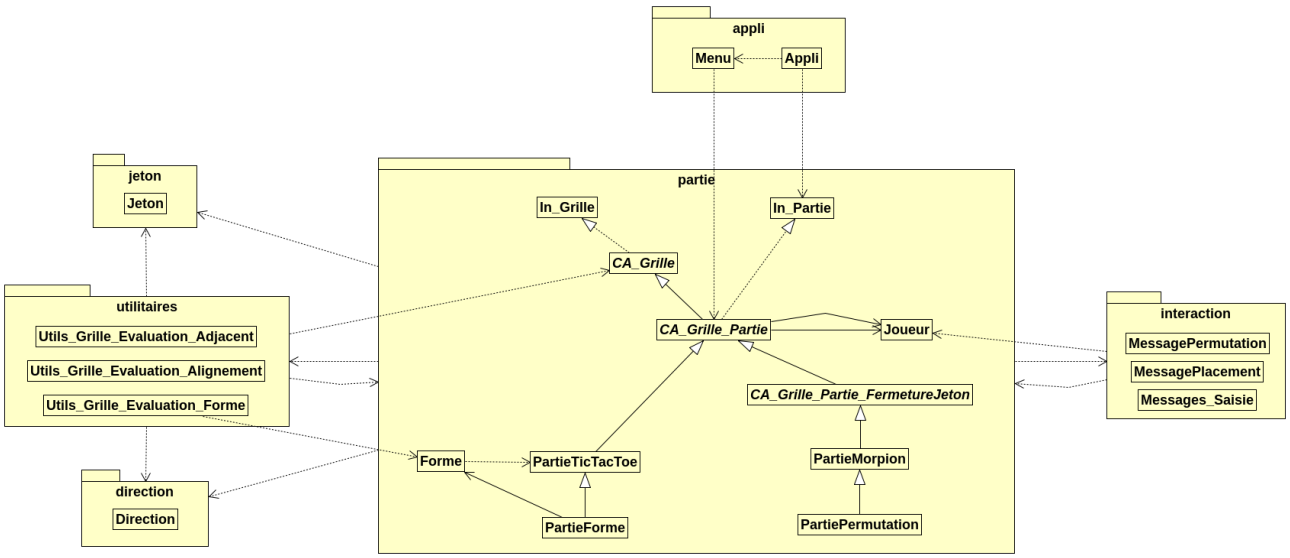

Illustration 2: Diagramme UML des classes, version simplifiée.

II. Difficultés rencontrées

Lors du développement de l’application, quelques difficultés ont été rencontrées :

1. Conceptualiser l’application :
   - Doit on avoir une classe grille ? Les données étant dans ce cas moins encapsulées, une classe abstraite contenant une table de jeton a été finalement mise en place.
   - Doit on avoir une fabrique de tours utilisant *CA_Grille_Partie* ? Dans ce cas, si de nouvelles méthodes étaient développées au sein des sous-classes concrètes des parties des nombreux casts de *CA_Grille_Partie* auraient eu lieu ; ainsi cette fabrique n’a pas été développée.

2. Utiliser les classes d’énumération :
   - Le TTT ne doit pas avoir accès à des données qu'il ne doit pas utiliser. Il a fallu limiter les jetons a deux valeurs (X,O) et non 4 (X,x,O,O). Ainsi il a été mise en place une table de booléens pour définir l'état d'ouverture des jetons.
   - La classe *Forme* devrait être une énumération. Cependant ceci est non possible avec la méthode *transForme(n)*, qui décale les déplacements relatifs à réaliser pour évaluer si une forme est complète.

3. La partie *Forme* :
   - Elle ne se termine pas quand un joueur a complété une forme donnée (*i.e* marque un point), alors qu'au vue des méthodes de TTT dont elle hérite (ex. *estFinie()*) ce devrait être le cas puisque celles-ci fonctionnent pour le TTT. De plus les méthodes qui reconnaissent si un jeton complète une forme donnée sont opérationnelles comme le montre les tests. Après une observation attentive lorsque *evaluerCoup()* evalue qu’une forme est complète, elle augmente le score d'un joueur, mais celui-ci ne semble pas être celui de *CA_Grille_Partie* car le score de ce dernier reste à 0.

4. Limiter l’héritage de méthodes non utilisées par les applications :
   - Il a donc été définies des classes vides dans le package u*tilitaires*.

5. Factoriser du code :
   - Pour le M, qui comprenait des jetons à ouvertures variables, il a fallu reprendre les méthodes d'alignement du package *Utilitaires* élaborer pour l'appli TTT mais en remplaçant la *getJeton*(X,O) par *getJetonOouF*(X,x,O,O). Comme le M utilise la classe Jeton, mais n'hérite pas de la classe Jeton une surcharge de la méthode n'est à ma connaissance pas utilisable.

6. Tester des méthodes privées :
   - Il a été décide de modifier leur attribut pour la réalisation des tests. Ce n'est pas la bonne méthode car tous les test devraient être opérationnels en tout temps *t*.

7. Limiter l’accès aux donner
   - En C, nous avions pris l'habitude de mettre les données dont nous avions besoin en paramètre des nos fonctions. Ainsi au début du projet, lors de l'écriture de la classe abstraire *CA_Grille_Partie* il y avait la méthode suivante *evaluerCoup(Joueur j1, Joueur j2)*, or cela

contraignant la visibilité de *j1* et *j2* à *protected*. Ainsi finalement il a été décidé le prototype suivant : *evaluerCoup()* et la mise en place de *getteurs*.

III. Améliorations possibles

Des améliorations de l'application sont possibles :

1. Pour le point 4. des difficultés rencontrées,

- ce qui aurait pu être fait c'est :
  - incorporer ces méthodes au sein de la classe TTT,
  - faire du M une classe fille du TTT,
  - obtenir les caractères des jetons non pas avec la méthode *getCellule(i,j).getJeton()* mais avec une méthode *getSymbole(i,j)* (renvoyant X ou O dans TTT) qui aurait ensuite été surchargée (renvoyant X,x,O ou O dans M).
- la réécriture du code est contraire à la POO, ainsi l'externalisation de l'évaluation de la grille ne semble finalement pas une bonne idée.

2. La classe *forme* devrait être une classe d'énumération. La classe *PartieForme* devrait être améliorer pour fonctionner.
3. Avec la méthode de déplacement relatif toutes les formes ne peuvent être réalisées. En effet il y a des angles morts : par exemple d'une case en 3-3 ne peut pas rejoindre une case en 1-2 directement.

IV. Respect de la POO

Cette application respecte et tire profit des fondamentaux de la POO, dont voici quelques exemples :

1. Protection des données :
   - Les objets contenus dans la grille sont issus d'une classe d'énumération, limitant ainsi la diversité des données.
2. Promotion du code existant :
   - La définition d'un package *utilitaires*, et notamment les méthodes telles que *getCoordForme()* permet la réutilisation de ce code et une évaluation a posteriori des jetons obtenus.
3. Utilisation du polymorphisme :
   - La subsomption des méthodes *utilitaires* qui évaluent une *CA_Grille* mais qui sont utilisées avec une de ses classes concrètes.
   - La surcharge de certaines méthodes définies dans *CA_Grille_Partie* au sein des classes concrètes directe (TTT et M) et indirectes (MP et TF) a permis la définition spécifique de placement de jeton, d'évaluation de coup et de conditions de fin de match.
4. Hiérarchisation des classes, que l'on peut comparer à un arbre :
   - Une interface à la racine,

- Des classes abstraites au niveau des nœuds,
- Des classes concrètes au niveaux des feuilles.

V. Extensions possibles

Comme notre application respecte les objectifs de la POO, des extensions sont possibles :

1. Il y a trois classes abstraites qui permettent à de futures extensions d'y prendre racines. Celles ci pourront ré-utiliser la possibilité de fermer les jetons (à partir de *CA_Grille_Partie_Fermeture*) ou non (*CA_Grille_Partie*).
2. Les extensions pourront être lancer à partir de l'*Appli* du moment qu'elles implémentent l'interface *In_Partie* qui comprend uniquement la méthode *lancerPartie()*.
3. La réutilisation des méthodes d'évaluation (package *utilitaires*) et celles du choix sécurisé de cellule (package *interaction*) de la grille est possible dans d'autres extensions si elles sont issues de *CA_Grille*.
4. Si l'on souhaite jouer avec un 3ème joueur il faudrait :

- dans la *CA_Grille_Partie :*
  - utiliser la méthode `public Joueur aQuiLeTour() {return ((tour % 2 == 0) ? joueur2 : joueur1);}` pour la définition `Joueur joueurActuel = aQuiLeTour();`.
- dans *Partie3J.java :*
  - surcharger la méthode *aQuiLeTour()*,
  - avoir dans les attributs : `` `private Joueur joueur3` ``,
- dans l'énumération *Jeton* :
  - rajouter un symbole d'un autre caractère

5. Comme la *CA_Grille_Partie*, comporte la méthode *jouerCoup(Joueur joueurActuel)*, si l'on souhaite faire une extension de chacune des partes avec une *IA*, il faudrait :

- qu'elles étendent leur partie 2 joueurs physiques respectives
- qu'elles surchargent la méthode *jouerCoup(Joueur joueurActuel)* en choisissant la méthode *super.jouerCoup(joueurActuel)* ou celle de l'*IA* en fonction de *joueurActuel.getJeton()*.

VI. Le script

Le script de ce projet est disponible en page suivante, les packages sont dans l'ordre suivant :

1. appli
2. direction
3. interaction *(onglet bleu)*
4. jeton
5. partie *(onglet orange)*
6. tests *(onglet vert)*
7. utilitaires