TP C#
Yu-Gi-Gi

1 Introduction

1.1 Avertissement

Certaines notions abordées dans ce tp peuvent être un peu délicates à comprendre. Soyez sûrs d'avoir bien compris les concepts de bases avant de vous y atteler.
N'hésitez pas à solliciter les assistants présents durant toute la durée du tp !

1.2 Objectifs

Le but de ce TP est de vous faire découvrir les bases de la programmation orientée objet en C#. Vous allez devoir implémenter :

- une classe abstraite,
- de l'hérédité,
- des vector<T>

Pour cela, vous allez créer un programme qui fait un jeu de carte Yu-Gi-Oh simplifié.

Nous vous conseillons de lire l'intégralité du sujet avant de commencer. Si vous avez un doute sur une fonction du C#, consultez d'abord le site MSDN. Enfin, si vous avez des questions à propos du sujet ou sur le C# en général, n'hésitez pas à demander aux assistants.

2 Description du jeu

Yu-Gi-Oh est un jeu de carte en un contre un (essayez de faire en plus les duels en équipes si ça vous tente, mais ce n'est pas l'objectif de ce TP).

Chaque joueur a 4000 points de vie au début de la partie, et le but est de faire descendre ceux de son adversaire à 0. Pour cela, vous pouvez invoquer des monstres qui vont attaquer votre adversaire, ou les monstres qu'il a invoqué. Vous pouvez aussi utilisez des cartes magiques et pièges qui influencent certains éléments du jeu.

3 Créons nos cartes

3.1 Classe abstraite

Pour commencer, créez la classe abstraite `Card` qui sera utilisée pour toutes les cartes que nous allons créer. Elle est abstraite car une carte ne sera jamais juste une carte, ce sera forcément une carte monstre, ou une carte magique, ou une carte piège. Elle contient au minimum les attributs `name` et `type`, qui correspondent respectivement au nom de la carte et à son type (monstre, magique ou piège). Ils auront `protected` en visibilité.

Elle doit contenir un constructeur qui initialise ces deux champs à partir des arguments donnés.

De plus, elle doit contenir au moins une méthode `Print()` qui sera définie dans les classes filles. Elle servira à afficher la carte dans la console.

3.2 Cartes monstres

Créez une classe `Monster` qui hérite de `Card`. Elle contient en plus les attributs `level` qui représente le niveau du monstre (entre 1 et 12), `attack` l'attaque (entre 0 et 10 000), `defense` la défense (entre 0 et 10 000).

Les cartes monstres peuvent être en position d'attaque ou de défense. Notre classe aura donc un autre attributs `mode` qui la déterminera.

Le constructeur initialisera toutes ces valeurs.

Tous les champs sont en visibilité `private`. Les attributs `attack` et `defense` doivent cependant pouvoir être lu depuis une méthode extérieure à la classe.

La méthode `Print()` devra être définie pour afficher les cartes comme sur l'exemple suivant :

```
---------------
|Name           |
|Lvl n          |
|---------------|
|               |
|               |
|               |
|               |
|---------------|
|atk n          |
|def n          |
---------------
```

Pour différencier le mode attaque du mode défense, on utilisera deux couleurs (rouge pour attaque et bleu pour défense, par exemple).

```
1 void Print()
```

Vous pouvez rajouter tous les attributs et méthodes que vous jugez nécessaires.

Exemples de cartes monstres

Vous pouvez maintenant créer vos propres cartes monstres ! Libre à vous de copier les cartes déjà existantes du jeu, ou d'inventer les vôtres.

Nous allons maintenant rajouter les cartes magiques.

3.3 Cartes magiques

Créez une classe `Magic` qui hérite de `Card`. Elle contient en plus l'attribut `description` qui, comme son nom l'indique, est une chaîne de caractères décrivant ce que fait la carte.

Tous les champs sont en visibilité `private`.

La méthode `Print()` devra être définie pour afficher les cartes comme sur l'exemple suivant :

```
---------------
|Name           |
|               |
|--------------|
|               |
|               |
|               |
|               |
|--------------|
|(description)|
|               |
---------------
```

Pour ne pas les confondre avec les cartes monstres, on les affichera avec une couleur verte.

```
1 void Print()
```

Vous pouvez rajouter tous les attributs et méthodes que vous jugez nécessaires.

Les cartes magiques ont une méthode `Effect()` qui sera appelée lorsque la carte sera jouée (pensez aux fonctions anonymes et aux delegates). Elle renverra un booléen pour savoir si le sort a bien pu être lancé.

```
1 bool Effect(Board player, Board opponent)
```

La classe `Board` est décrite plus loin dans le sujet.

Toutes les cartes magiques ont un effet différent, a vous de voir comment les implémenter. Nous vous demandons d'en faire au moins 3 :

-équipement : renforce les points d'attaques et/ou de défense d'un monstre

-soin : soigne vos points de vie d'une valeur fixe

-Pot of Greed : La carte Pot of Greed permet de piochez deux cartes de son deck

D'autres sont disponibles dans les boni si vous le souhaitez.

Nous avons donc maintenant besoin de créer le plateau afin d'avoir accès au deck.

4 Le plateau de jeu

Nous allons maintenant faire le plateau du jeu et la main du joueur. Le plateau se présente sous la forme suivante :

4.1 Une case

Créez une classe `Box` qui contiendra une case du plateau. Cette classe contient un attribut qui est un vecteur de cartes (`vector<Card>`). Ainsi, on pourra stocker les cartes du deck et du cimetière sur une seule case.

Elle contient une méthode `Print()` qui :

- affiche la carte s'il n'y en a qu'une
- affiche un rectangle vide s'il n'y a aucune carte
- affiche un carte avec uniquement des points s'il y a plusieurs cartes (les cartes faces cachées de manière générale)

```
1 void Print()
```

Vous pouvez rajouter tous les attributs et méthodes que vous jugez nécessaires.

4.2 Le plateau

Créez une classe `Board` qui correspondra au plateau d'un joueur. Il contient un attribut `monsters` un tableau de 5 `box`, un attribut `magicTrap` un tableau de 5 `box`, un attribut `deck`, un attribut `cemetery` et un attribut `hand`.

Elle doit aussi posséder les informations sur le joueur, tel que son nom dans l'attribut `name`, et ses points de vie dans `life`.

Elle contiendra aussi une méthode `Display()` qui affichera tout le plateau, et une méthode `IsAlive()` pour indiquer si le joueur a encore des points de vie.

```
1 void Display()
2 bool IsAlive()
```

Vous pouvez rajouter tous les membres et méthodes que vous jugez nécessaires.

5 Polymérisation !

Maintenant, nous allons fusionner les différentes parties afin de construire le jeu.

Dans la méthode main, il vous faut deux joueurs, donc deux instances de `Board`. Créez dedans deux decks avec les cartes que vous aurez créées, et mélangez-les avec des fonctions de randomisation (voir la classe `Random` sur MSDN).

Le tour d'un joueur va se dérouler de la manière suivante :

- Le joueur pioche une carte. S'il n'a plus de cartes, le joueur perd. Vous implémenterez cela dans la méthode `Draw()` prenant en paramètre l'instance de `Board` correspondant au joueur et le nombre de cartes à piocher. La méthode renvoie faux si le joueur n'a plus de cartes, et donc ne peux plus piocher.

```
1 bool Draw(Board player, int nbCard)
```

- Le joueur peut ensuite invoquer un monstre et un seul depuis sa main. Pour cela, il faut vérifier que la case est vide, et transférer la carte depuis la main vers la case. Vous ferez cela dans la méthode `Invocation()` prenant en paramètre l'instance de `Board` correspondant au joueur, puis l'indice de la carte à invoquer, l'indice de la case où invoquer le monstre, et enfin un booléen précisant si le monstre est invoqué en mode attaque ou non. Vous devrez gérez les cas d'erreur où le joueur veut invoquer une carte magique, la case est occupée, il n'y a pas de carte à cet indice dans la main. Dans ces cas là, vous afficherez juste un message d'erreur dans la console puis quitterez la fonction. La méthode renvoie faux si le moindre problème pour l'invocation arrive (pas de carte, pas de place, etc).

```
1 bool Invocation(Board player, int card, int case, bool attack)
```

  Le joueur peut aussi jouer une carte magique depuis sa main. Pour cela, implémentez la méthode `Spell()` prenant en paramètre l'instance de `Board` correspondant au joueur, puis l'indice de la carte magique dans la main, puis l'indice de la case sur laquelle le joueur va jouer la carte. Vous devrez gérer les cas d'erreur où le joueur essaye de jouer une carte monstre, ou bien lorsque le case est déjà occupée par une carte ou encore lorsqu'il n'y a pas de carte à cet indice de la main. Dans ces cas là, vous afficherez juste un message d'erreur dans la console puis quitterez la fonction. La méthode renvoie faux si le sort n'a pas pu être lancé.

```
1 bool Spell(Board player, int card)
```

  Vous pouvez demander d'autres informations à l'utilisateur afin de savoir, par exemple, sur quel monstre la carte doit-elle agir, etc.

- Le joueur peut ensuite attaquer le joueur adverse. Il doit attaquer les monstres adverses avant de pouvoir attaquer le joueur directement. Seuls les monstres en mode attaque peuvent attaquer.

  Si le monstre attaqué est en mode attaque, alors le monstre le plus faible est détruit et envoyé au cimetière, et son possesseur perd la différence entre leurs attaques en point de vie; si le monstre attaqué est en mode défense, alors soit il a moins de points de défense que l'attaquant a de points d'attaque et est détruit, soit il en a plus et l'attaquant perd la différence entre les points de défense du défenseur et les points d'attaque de l'attaquant en point de vie.

  Si le joueur adverse n'a plus de monstre sur son plateau, vous pouvez alors lancer une attaque directe sur ses points de vie, qui descendront de la valeur des points d'attaque du monstre. Dans ce cas, la valeur envoyée en tant que monstre attaqué sera -1.

  Chaque monstre ne peut attaquer qu'une seule fois par tour.

```
bool Attack(Board player, Board opponent, int attackingMonster,
                int attackedMonster)
```

La fonction prend en paramètre le plateau du joueur courant, puis celui de l'adversaire, la position sur le plateau du monstre attaquant et la position du monstre attaqué, sur leur plateau respectif

- C'est ensuite la fin du tour et le joueur suivant peut jouer.

A chaque joueur qui joue, vous devrez appeler la méthode `Play()` qui prend en paramètre d'abord l'instance de `Board` du joueur jouant, puis celle du joueur adverse. Cette fonction est donnée dans le fichier zip, disponible sur l'intranet des ACDC. De plus, son utilisation pour le joueur est expliquée dans le README donné avec.

```
void Play(Board player, Board opponent)
```

A chaque fois que les points de vie des joueurs sont modifiés, vous devez vérifier que l'un d'eux n'a pas perdu, et déclarer le vainqueur si besoin.

Pour toutes les méthodes décrites dans ce chapitre, vous devez respecter exactement les prototypes, car elles seront appelées par la méthode `Play()`. Si vous voulez changer les prototypes, vous devez modifier cette méthode.

6 Boni

Il y a de nombreux boni possibles pour que le jeu soit plus ressemblant à l'original. Voici une liste non-exhaustive de ceux-ci :

-image : Vous vous demandiez pourquoi il y avait un carré blanc dans l'affichage des cartes ? Vous pouvez y mettre une image de votre carte en ascii art (certains sites internet permettent de convertir une image en ascii art). Vous pouvez modifier la taille des cartes à l'affichage, tant qu'elles ont toutes la même taille et qu'elles sont toutes visibles a l'écran.

-invocation avec sacrifice(s) : Les monstres de niveau entre 1 et 4 s'invoquent normalement. Cependant, les monstres de niveau entre 5 et 8 nécessitent le sacrifice d'un monstre déjà présent sur le plateau qui sera envoyé au cimetière à l'issue de cette invocation. Pour les monstres de niveau entre 9 et 12, il faut faire 2 sacrifices. Il faudra donc demander en plus au joueur de désigner les monstres à sacrifier.

-invocation face cachée : Les monstres peuvent être invoqué en mode défense face cachée. L'adversaire ne peut donc pas voir de qui il s'agit avant de l'attaquer ou avant que le joueur ne décide de la retourner. La carte est affichée avec des points lorsqu'elle est cachée, comme pour le deck.

-cartes magiques : Rajoutez différentes cartes magiques, permettant par exemple de changer de mode un monstre, changez la main, détruire un monstre, ressusciter un monstre mort, etc.

-monstres à effet : Certains monstres peuvent avoir des effets lorsqu'on les invoque, lorsqu'ils attaquent, lorsqu'ils sont attaqués, etc. Par exemple, **Slifer, le dragon du ciel** a une attaque égale au nombre de cartes dans la main du joueur fois 1000.

-cartes pièges : Implémentez la classe `Trap` héritant de `Card` pour faire des cartes pièges. Celles-ci sont similaires aux cartes magiques, mais il faut d'abord les jouer face cachée et on ne peut les activer qu'à partir du tour suivant. Cependant, on peut les jouer à n'importe quel moment, même pendant le tour de l'adversaire. Par exemple, la carte **Annulation d'attaque** empêche le joueur adverse d'attaquer pendant le tour où la carte est jouée.

-Yu-Gi-Gi évolue en Yu-Gi-Oh : Implémentez un tour de jeu comme dans le jeu original, avec toutes les phases (draw phase, battle phase, etc). Vous pouvez évidement modifier la fonction donnée pour lire les commandes du joueur.

-Windows Form : Si vous êtes sur Windows, vous pouvez faire une interface graphique avec Windows Form sur Visual Studio. Vous pouvez ainsi mettre des images pour les cartes, et jouer avec la souris plutôt que par lignes de commande.

-réseau : Faites le jeu en réseau pour jouer en un contre un sur deux ordinateurs différents (allez voir du côté des socket).

The code is the law.