FEAUX DE LACROIX Martin
22015860
FIDALGO Alex
71600135

PROJET

Projet de language à objets avancés

Framework pour jeux de cartes

(C++)

SOMMAIRE

Nos choix d'implémentations

Pour ce projet nous avons choisi d'implémenter les jeux suivant à partir de notre Framework :

→ **la Bataille *(imposé)*.**

→ **le Uno.**

→ **le 8 Américain.**

→ **la Scopa.**

→ **la Briscola.**

Le pattern Model View Controller

Pour se faire nous avons opté pour un pattern de type Model View Controller étant adapté à la situation.

Le Model

Dans notre Framework, le Model est représenté par la classe **Game**, c'est une classe abstraite ayant pour but d'implémenter le plus de fonctionnalités regroupées dans les différents jeux que nous avons choisi d'implémenter. Pour se faire, elle défini plusieurs méthodes virtuelles pures qui devront être implémentés dans les jeux choisi. Par ailleurs, cette classe possède également des méthodes virtuelles qui selon les jeux devront être ré-implémentés où non.

Les attributs de la classe **Game** :

→ le deck , représenté par une instance de la classe **CardBundle**.

→ la table, représenté par une instance de la classe **CardBundle**.

→ un *vector* de **Player**, représentant les différents joueurs du jeu.

→ ...

La View

La View, représenté par la classe **View** est une classe *purement abstraite* (interface) que notre classe **TerminalView** implémente. Cette dernière est une vue terminale fournie par notre Framework et sera à responsable de tout l'affichage lors des jeux.

Le Controller

Le Controller, représenté par une classe *template* **Controller<class M, class V>** , **M** représentant le **Model** et **V** représentant la **View** associée.
Cette classe est le lien entre notre **Model** et notre **View**, et contient la boucle principale de jeu de notre **Framework**, implémentée dans la méthode **run**.

Les objets

Les cartes

Pour les différentes cartes de nos jeux nous avons choisi d'implémenter une unique classe **Card** assez générale permettant de représenter une carte pour n'importe lequel de ces jeux. Effectivement, une carte est un objet variant sur **4** attributs peut-importe le jeu :

- → **une couleur.**
- → **un signe.**
- → **une valeur.**
- → **des points.**

Pour se faire, nous avons implémenter différents constructeurs permettant de construire une carte avec les spécificités dont elle à besoin, selon le jeu.

Les joueurs

Dans notre **Framework** un joueur est représenté par une instance de notre classe **Player** peut-importe qu'il soit *humain* où *robot*. Un joueur possèdera un **CardBundle** représentant sa main ainsi qu'un second représentant une **pile** qui lui permettra d'accumuler des cartes. Afin de faire la différence entre les joueurs *humains* et les joueurs *robots* nous avons opté pour une manière simple consistant en un attribut *booléen*, étant binaire il suffit pour faire la différence entre ces deux entités.

Les paquets de cartes

Des *paquets de cartes* sont nécessaires dans chaque jeu de cartes. Il permettent de représenter par exemple la **main** d'un joueur, sa **pile** de cartes, la **table** où encore le **deck**. Afin de representer ces différents objets, nous avons choisi d'implémenter une seule classe **CardBundle** représentant donc un paquet de cartes, car chacun des ces objets peuvent selon les contextes interagir de la même manière. Par exemple, nous pourrions penser que seul un **deck** sera amener à être mélangé, mais nous pourrions vouloir implémenter des jeux où un mélange de la **table** et de la **main** du joueur serait nécessaire.

Comment implémenter un jeu à l'aide de notre Framework ?

Afin d'implémenter un jeu, la seule chose nécéssaire sera de définir la classe pour le jeu en question, qui héritera de notre **Model** : la classe **Game** dans le cas de notre **Framework**. Il y a quelques spécificités selon le jeu à implémenter, des méthodes qu'il faudra re-définir dans certains cas et dans d'autres non.

Les méthodes virtuelles pures à définir

Tout d'abord, pour chacun des jeux, il faudra évidemment définir les méthodes virtuelles pures, qui sont donc spécifiques à chacun des jeux.

Voici les méthodes virtuelles pures à définir :

- **virtual int getMaxPlayers() const = 0 ;**
  - cette méthode permet seulement de spécifier le nombre de joueurs **maximum** pouvant participer au jeu, elle est virtuelle pure car chacun des jeux possède un nombre de joueurs maximum différent. La même méthode est définie dans le modèle de base **Game** pour le nombre de minimum de joueurs étant donné que dans les jeux visés le nombre de participants est toujours **2**. Bien évidemment elle peut également être redéfinie pour adapter ce paramètre.

- **virtual void createDeck() = 0 ;**
  - cette méthode permet de créer le deck pour le jeu à implémenter. Effectivement, certains jeux utilises les mêmes cartes, mais les points associées à celle-ci sont différents et rende cette méthode non généralisable facilement.

- **virtual int getCurrentPlayerValidMove() = 0 ;**
  - cette méthode renvoie l'index du premier coup valide de la main d'un joueur si il existe. Elle sera utile afin que le modèle puisse renvoyer automatiquement un coup valide lorsqu'un robot doit jouer.

- **virtual std : :pair<int,int> howToDeal() const = 0 ;**
  - cette méthode est couplée avec une méthode de la classe de base **Game** et va permettre de lui indiquer de quelle façon distribuer les cartes aux joueurs et sur la table au début d'une partie. Elle renvoie une paire dont le premier paramètre indique le nombre de cartes à distribuer à chaque joueur et le second, le nombre de carte à poser sur la table en début de partie.

Les jeux s'évaluant à chaque fin de tour

Il y à une différence dans la façon dont les jeux se déroule, dans le sens où certains jeux comme le **Uno** ne nécessite par d'évaluation particulière à la fin d'un tour de jeu mais d'autre jeux comme la **Bataille** où la **Scopa** sont basé sur ce principe.
Pour se faire, les jeux nécessitant une évaluation de fin de tour devront re-définir les méthodes suivantes :

- **bool evaluationTurnCondition() const override ;**
  - cette méthode renvoie un booléen, qui vaudra **true** quand cette évaluation sera nécessaire (par exemple à la **Bataille** lorsque des deux joueurs ont jouer, à la **Scopa** lorsque chacun des joueurs n'a plus de carte) et permettra ensuite à la méthode suivante **evaluateTurn()** d'être appelée afin d'effectuer cette évaluation.

- **Player * evaluateTurn() override ;**
  - cette méthode permet d'effectuer l'évaluation d'un tour. Par exemple : à la **Bataille** vérifier qui à poser la plus grande carte et déterminer un gagnant. À la **Scopa**, re-distribuer **3** cartes à chacun des joueurs et ajouter les cartes de la table à la pile du gagnant.

- **Player* evaluateWinnerOfRound() const override ;**
  - cette méthode permet cette fois-ci de renvoyer qui à gagner un **round**. Par default, dans la classe **Game** de base, elle est défini telle que le gagnant est le joueur qui n'a plus de cartes (au **Uno** où au **8 Americain** par exemple). Les jeux où le gagnant est basé sur une autre condition, devront re-définir cette méthode.

Les jeux avec des cartes à effets

Les jeux qui possèdent des cartes ayant des effets (**Uno** & **8 Américain**) sont différents et nécessitent donc une adaptation en conséquence. Pour alléger le jeu qui héritera de notre **Model** (classe **Game**), nous avons implémenter les méthodes qui correspondent aux différents effets que nos jeux peuvent rencontrer, en l'occurrence :

- **int skipTurn() ;**
  - saute le tour du joueur suivant.
- **int drawTwo() ;**
  - fait piocher **2** cartes au joueur suivant.
- **int drawFour() ;**
  - fait piocher **4** cartes au joueur suivant.
- **int reverse() ;**
  - inverse le sens du jeu.

Ils doivent donc implémenter les méthodes suivantes :

- **bool needFirstCardEffect() const ;**
  - doit simplement renvoyer **true**, afin de permettre au modèle de base d'appliquer l'effet de la première carte posée sur la table en début de partie.
- **int chooseColor() ;**
  - doit renvoyer **6** pour le **8 Américain** et **7** pour le **Uno**.
- **int getCardEffect(const Card &card)**
  - renvoie l'effet de la carte passée en paramètre. En utilisant les effets implémentés dans la classe **Game** de base, elle devient facile à implémenter. Par ailleurs, si l'on veut rajouter un nouvel effet, il suffira de le définir dans notre classe et ensuite de modifier cette méthode en conséquence.

Autres méthodes

→ Le calcul des points s'effectue via une méthode appelée **void calculatePointsForWinner(Player *winner)** qui ajoutera **1** point au gagnant par défaut mais sera re-définie lorsqu'un comptage des points plus complexe est nécessaire.

→ La méthode **string getGameHeader()** n'est aucunement nécessaire à l'implémentation d'un jeu, son but étant de fournir à la **View** une représentation textuelle du nom du jeu.

→ La méthode **string getRules()** n'est pas non plus nécessaire au bon fonctionnement d'un jeu, elle permet de renvoyer une représentation textuelle des règles du jeux afin de rappeler au joueur comment pouvoir y jouer et également permettre de lui présenter lorsqu'il fait un coup qui est invalide.

→ Les méthodes relatives aux coups permettant de couper comme il est possible de le faire dans la **Scopa** sont des cas particulier de ce jeu et donc doivent être ré-implémentés seulement dans celui-ci.

How to : nouveau jeu

Pour résumer, la création d'un jeu à partir de notre **Framework** consiste à effectuer les étapes suivantes :

→ implémenter la classe du jeu qui héritera de la classe **Game** (*voir documentation pour plus de détails*).

→ pour lancer le **Jeu** depuis votre **main** procéder comme suit :

Listing 1 – main

```
1 #include "Jeu.h"
2 #include "TerminalView.h"
3 #include "Controller.h"
4
5 int main(){
6     auto c = new Controller<Jeu,TerminalView>();
7     c->run();
8 }
```

→ créer votre makefile qui devra inclure tout les fichiers de notre **Framework**, exemple :

Listing 2 – Makefile

```
1  CC = g++
2  CFLAGS= --std=c++11 -Wall
3  PREFIX = ../..
4
5  # Paths needed
6  MODEL_INCLUDE = -I $(PREFIX)/Framework/include/Model/Game \
7  -I $(PREFIX)/Framework/src/Model/Game \
8  -I $(PREFIX)/Framework/src/Model/Objects \
9  -I $(PREFIX)/Framework/include/Model/Objects \
10 -I $(PREFIX)/Framework/src/Model/Others \
11 -I $(PREFIX)/Framework/include/Model/Others
12
13 VIEW_INCLUDE = -I $(PREFIX)/Framework/include/View \
14 -I $(PREFIX)/Framework/src/View
15
16 CONTROLLER_INCLUDE = -I $(PREFIX)/Framework/include/Controller \
17 -I $(PREFIX)/Framework/src/Controller
18
19 EXT_INCLUDE = -I $(PREFIX)/Framework/include/Exception \
20 -I $(PREFIX)/Framework/src/Exception
21
22 # Includes
23 INC = $(MODEL_INCLUDE) $(VIEW_INCLUDE) $(CONTROLLER_INCLUDE)
24 $(EXT_INCLUDE)
25
26 # Add other needed libraries here
27 LIBS = #-lcardgameslib
28
29 # Target
30 TARGET = Jeu
31
32 # Sources
33 MODEL_SOURCES = $(wildcard $(PREFIX)/Framework/src/Model/*/*.cpp)
34 $(wildcard $(PREFIX)/Framework/include/Model/*/*.cpp)
```

```
35 VIEW_SOURCES = $(wildcard $(PREFIX)/Framework/src/View/*.cpp)
36 MAIN = $(wildcard *.cpp)
37
38 SOURCES := $(MODEL_SOURCES) $(VIEW_SOURCES) $(MAIN)
39 OBJS := $(patsubst %.cpp,%.o,$(SOURCES))
40 DEPENDS := $(patsubst %.cpp,%.d,$(SOURCES))
41
42 all: $(TARGET)
43
44 $(TARGET): $(OBJS)
45     @printf "== everything linked %s ==\n" $@
46     @$(CC) $(CFLAGS) $(INC) $^ -o $@ $(LIBS)
47
48 -include $(DEPENDS)
49
50 %.o: %.cpp Makefile
51     @printf "== linking %s ==\n" $@
52     @$(CC) $(CFLAGS) $(INC) -MMD -MP -c $< -o $@
53
54 .PHONY:clean
55
56 clean:
57     rm -rf $(TARGET) $(OBJS) $(DEPENDS)
```

Amélioration

Les joueurs sont pour l'instant gérer de manière simpliste par le modèle ou la vue (pour les joueurs humains). Dans le cas des bots, ceux-ci joue pour l'instant un coup par défaut génère par l'instance de Game correspondante. Cependant si on souhaite implémenter de nouveau type de joueur avec une intelligence plus développer il faudra modifier la classe Player selon l'UML suivant 1. Chaque sous classe de Player devra implémenter chooseCut et chooseMove, le paramètre Game *game est un pointeur vers l'instance du jeu actuelle, permettant au joueur d'examiner la position du jeu avant de jouer son coup. (On garde la même convention qui si on retourne pour coup -1, on ne joue pas/ ne peut pas jouer, exemple : si chooseCut renvoie (-1, -1, -1) alors le joueur a choisie de ne pas couper.)

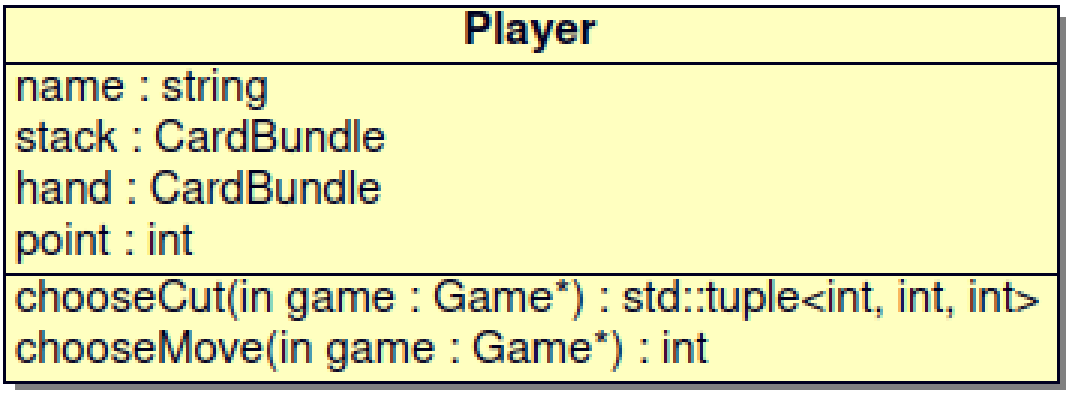

FIGURE 1 – Player