Table des matières

Présentation

Il s'agit de réaliser un simple jeu de dés en programmation Orientée Objet **afin de mettre en oeuvre l'utilisation d'un atelier de génie logiciel (bouml)**.

Expression du besoin

Le client désire jouer une partie de jeu de dés. Le joueur demande à lancer les dés. Le système affiche le score obtenu en respectant les règles suivantes :

- un double rapporte 20 points
- une suite rapporte 15 points
- un total supérieur à 7 rapporte 10 points
- sinon la valeur en points du total

Spécifications

Le système est composé de **2 dés à 6 faces**. Un dé sera modélisé par une classe du même nom.

La classe **De** possédera une méthode **`lancer()`** qui aura pour rôle de déterminer de façon pseudo-aléatoire (cf. la fonction `rand()` et `srand()`) la valeur du dé (une valeur comprise entre 1 et 6, cad le nombre de faces que comporte le dé).

La classe **`De`** doit respecter le **principe de séparation Commande-Requête**. c'est un principe de conception Orienté Objet classique pour les méthodes. Ce principe énonce que chaque méthode doit appartenir à l'une des deux catégories suivantes:

- une **commande** est une méthode qui effectue une action. Elle a souvent des effets de bords comme une modification de l'état d'un objet et n'a pas de valeur de retour (sauf pour indiquer si l'action a réussi ou a échoué) ;
- une **requête** est une méthode qui retourne des données à l'appelant et n'a pas d'effets de bord. Elle ne doit pas modifier de façon permanente l'état d'un objet.

La classe **`De`** définira donc une méthode **`lancer()`** qui est une commande : elle a pour effet de modifier la **valeur** du dé. En conséquence, elle ne doit pas également retourner cette nouvelle valeur sinon elle violerait la règle selon laquelle elle ne doit pas appartenir aux deux catégories. Pour obtenir la valeur du dé lancé, on utilisera une méthode **`getValeur()`** qui est une requête.

C'est un ***pattern*** simple : il permet de raisonner plus facilement sur l'état d'un programme et de rendre les conceptions plus faciles à comprendre. Il est donc agréable de pouvoir lui faire confiance.

*Ce qu'il faut retenir : Un **pattern** (ou motif de conception) est un document qui décrit une solution générale à un problème qui revient souvent.*

Dans le monde de l'orienté-objet, les design patterns se présentent comme un catalogue de méthodes de résolution de problèmes récurrents.

La classe De possédera aussi un constructeur par défaut (qui fixera un nombre de faces égal à 6). Pour des soucis de réutilisation, elle disposera aussi d'un constructeur auquel on pourra passer le nombre de

face désiré pour ce dé.

Remarque : il n'est pas demandé de modéliser une classe `Joueur`. On se limitera donc pour l'instant à une seule classe pour l'application.

Contraintes

Contrainte de développement

C'est le programme **`main.cpp`** qui symbolisera le joueur. On y créera les deux dés et on jouera au moins une partie.

Contrainte de l'environnement

Système d'exploitation : **Linux**
Environnement de développement : **GNU C++**
Atelier de génie logiciel : **bouml**

Documentations

Documents :

- Les documentations suivantes :

Ref.	**Description**
exemple-pratique-bouml-1-creation-de-classe.pdf	Tutoriel présentant la création de classe sous bouml
exemple-pratique-bouml-2-generation-de-code.pdf	Tutoriel présentant la génération automatique de code sous bouml
cours-c-c++-programmation-modulaire.pdf	Cours sur la fabrication modulaire en C/C++ présentant notamment l'utilisation de l'outil make et son fichier Makefile

Liens :

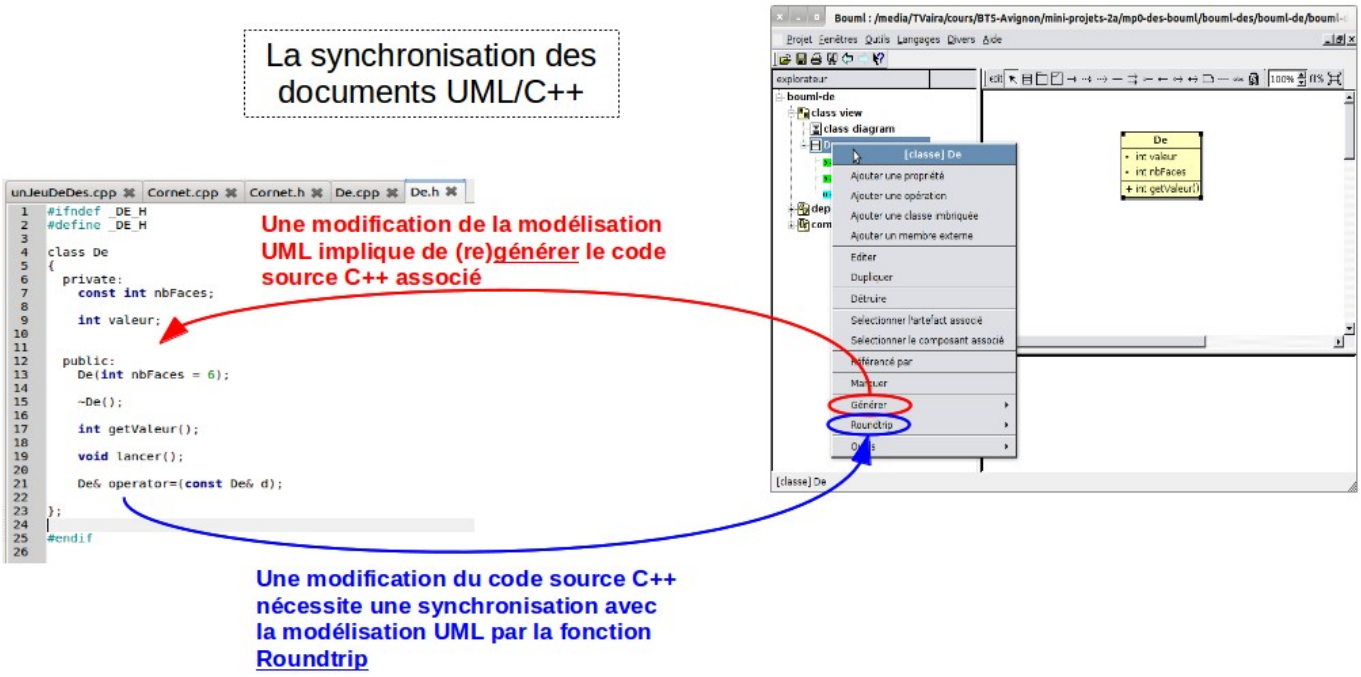
- génération de code avec bouml :
  http://bpages.developpez.com/tutoriels/bouml/classes-generation/

Travail demandé (itération n°1)

- identifier la fonctionnalité à réaliser pour le client
- spécifier et créer la classe **`De`** à partir d'un atelier de génie logiciel (**bouml**)
- générer automatiquement le squelette de la classe **`De`** à partir de l'atelier de génie logiciel (**bouml**)
- coder la fonctionnalité à réaliser ainsi que la classe **`De`** à partir d'un environnement de développement intégré (EDI)
- assurer la synchronisation entre l'AGL (uml) et l'EDI (c++)
- tester et valider le logiciel

Remarque : une fois votre projet créé dans bouml, n'oubliez pas de configurer le langage utilisé : ici le **C++**.

Le cycle de travail : bouml (UML) ↔ EDI (C++)



Aller plus loin (itération n°2)

En appliquant un **cycle de développement itératif et incrémental**, il est possible d'ajouter de nouvelles fonctionnalités pour évoluer l'application.

On a déjà vu que les dés sont des objets génériques utilisables dans toutes sortes de jeu. L'affectation de la responsabilité au programme principal de les lancer et de les additionner empêche de réutiliser ce service dans d'autres jeux.

> *Ce qu'il faut retenir : Un développement itératif s'organise en une série de développement très courts de durée fixe nommée itérations. Le résultat de chaque itération est un **système partiel exécutable, testé et intégré** (mais incomplet). Comme le système croît avec le temps de façon incrémentale, cette méthode de développement est nommée **développement itératif et incrémental**.*

Il y a un autre problème : il n'est pas possible de demander simplement le total actuel des dés sans devoir les lancer de nouveau.

On va donc introduire un nouveau concept (→ une nouvelle classe) :

- Il faut toujours essayer d'employer un vocabulaire en rapport avec le domaine. De nombreux jeux de plateau proposent d'utiliser un cornet pour secouer les dés et les lancer sur la table. On propose donc de créer une classe **Cornet** qui contiendra les dés, les lancera et connaîtra leur valeur, leur total et leur score.

Pour couvrir toutes les utilisations possibles d'un atelier de génie logiciel, on vous propose de mettre en œuvre la fonction **Reverse** de bouml.

Pour cela, vous devez écrire le squelette de la classe **Cornet** (les fichiers **Cornet.cpp** et **Cornet.h**) afin de pouvoir compiler le programme fourni en Annexe 2.

Comme il faut toujours conserver l'ensemble des documents (uml et code source) parfaitement **synchronisé**, il vous faudra donc importer cette nouvelle classe dans votre projet bouml : c'est ici que la fonction **Reverse** devient intéressante.

Exemple de programme :

```
#include <iostream>
#include "Cornet.h"

using namespace std;

int main( int argc, char* argv[])
{
  Cornet cornet; // par défaut un cornet composé de 2 dés
  int total = 0;

  cornet.lancer();
  total = cornet.getTotal();

  cout << "Vous avez réalisé " << cornet.getValeurDe(1) << ":" <<
cornet.getValeurDe(2) << endl;

 // TODO : calculer le score obtenu

  return 0;
}
```

Remarque : n'oubliez pas de mettre à jour votre fichier Makefile !

Une troisième itération serait de placer le code qui calcule le score dans une classe JeuDeDes ...

Annexe 1 : le fichier Makefile (itération n°1)

```
CC = g++
RM = rm
MAKEDEP = makedepend

INCLUDES = -I.
CFLAGS = -g $(INCLUDES)
LIBS =

OBJS = main.o de.o
TARGET1 = jeuDeDesv1

$(TARGET1): $(OBJS)
        $(CC) $(CFLAGS) $(LIBS) $(OBJS) -o $@

.cpp.o:
        $(CC) -c $(CFLAGS) $<

clean:
        $(RM) -f *.o $(TARGET1)

cleanall:
        $(RM) -f *~ *.o $(TARGET1)

dep:
        $(MAKEDEP) $(CFLAGS) -Y -s "# Dependances by 'make dep'" *.cpp
2> /dev/null
        rm -f Makefile.bak
```

