TP POO C++ : Mise en œuvre de BOUML

TP POO C++ : Mise en œuvre de BOUML

L'objectif de ce TP est de mettre en œuvre un atelier de génie logiciel (bouml) pour la génération de code C++.

Introduction

Atelier de génie logiciel

Un atelier de génie logiciel (AGL) désigne un **ensemble de programmes informatiques** (outils de gestion de projet, de génération de diagrammes et de code, de tests logiciels, de génération de documentation, de gestion des versions, éditeur, compilateur, édition des liens, etc ...) permettant eux-mêmes de produire des programmes de manière industrielle : Les AGL couvrent donc un champ au-delà des environnements de développement intégrés (EDI).

Quelques AGL : BOUML, Eclipse, Objecteering, Rational Rose d'IBM, ...

BOUML

BOUML est un logiciel, programmé par Bruno Pagès en C++/Qt, permettant la création de diagrammes UML.

Il est multiplateforme, supporte la **génération de code et la rétro-ingénierie pour les langages C++** , Java, PHP et MYSQL. Parmi les gratuiciels UML, il est jugé extrêmement efficace pour la « rétro-modélisation » (créer un modèle UML à partir de codes sources) et pour le développement *roundtrip* (faire des aller-retour entre modèle UML et code source).

Auparavant distribué gratuitement sous licence GPL (version 4.23), puis commercialisé pour les versions 5 et 6, il est de nouveau distribué gratuitement depuis la version 7 (mais pas intégralement sous licence GPL).

Lien : http://www.bouml.fr/

Acteullement, la version installée est la 4.23 (Ubuntu 12.04). Vous pouvez faire la mise à jour vers la version 7.12 : http://www.bouml.fr/download_fr.html

```
$ wget -q http://www.bouml.fr/bouml_key.asc -O- | sudo apt-key add -
$ echo "deb http://www.bouml.fr/apt/precise precise free" | sudo tee -a /etc/apt/sources.
   list
$ echo "deb http://bouml.free.fr/apt/precise precise free" | sudo tee -a /etc/apt/sources.
   list
$ sudo apt-get update

$ sudo apt-get install bouml
```

Ubuntu 12.04

UML

UML (*Unified Modeling Language*) est un **langage de modélisation graphique** à base de pictogrammes. Il est utilisé pour le développement logiciel en orientée objet.

UML 2.3 propose **13 types de diagrammes** (il y avait seulement **9 diagrammes** en UML 1.3).

UML n'étant pas une méthode de développement, leur utilisation est laissée à l'appréciation de chacun dans le cadre d'un développement logiciel.

UML se décompose en plusieurs sous-ensembles :
- Les **vues** : elles décrivent le système d'un point de vue donné, qui peut être organisationnel, dynamique, temporel, architectural, logique, etc. En combinant toutes ces vues, il est possible de définir (ou retrouver) le système complet.
- Les **diagrammes** : ils décrivent le contenu des vues. Le **diagramme de classes** est généralement considéré comme l'élément central d'UML.
- Les modèles d'élément : ils sont les briques des diagrammes UML.

Travail demandé

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

On désire jouer une partie à l'aide de 2 dés. Le joueur lance les dés et 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

On modélisera 2 classes C++ : `Joueur` et `De`.

Itération 1 : rétro-ingénierie de la classe Joueur

La rétro-ingénierie (*reverse engineering*) est l'activité qui consiste à étudier un objet pour en déterminer le fonctionnement interne ou la méthode de fabrication. On parle également de rétroconception. En développement orienté objet avec UML, la rétro-ingénierie consiste à générer des diagrammes UML à partir d'un code source.

Suite à des développements précédents, on suppose que l'on possède déjà le code source d'une classe `Joueur`. On va utiliser BOUML pour obtenir le diagramme de classes du code source existant.

La déclaration d'une classe `Joueur` :

```
#ifndef JOUEUR_H
#define JOUEUR_H

#include <iostream>
using namespace std;

class Joueur
{
    public:
        Joueur(string nom);
        ~Joueur();

        string getNom() const;
        void setNom(string nom);

    private:
        string nom;
};

#endif
```

joueur.h

La définition d'une classe `Joueur` :

```
#include "joueur.h"

Joueur::Joueur(string nom) : nom(nom)
{}

Joueur::~Joueur()
{}

string Joueur::getNom() const
{
    return nom;
}

void Joueur::setNom(string nom)
{
    this->nom = nom;
}
```

joueur.cpp

Étape 1. Créer un nouveau projet avec BOUML.

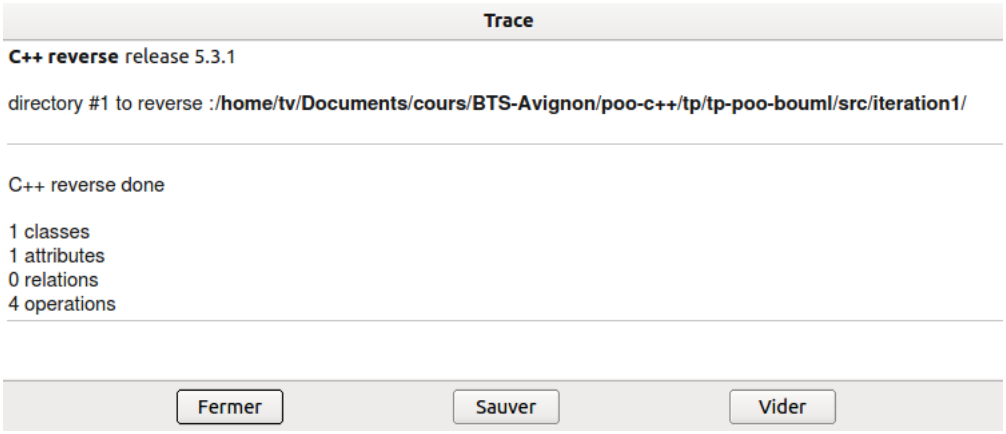
Il vous faut tout d'abord créer un nouveau projet : « Projet → Nouveau ». Puis, choisissez un nom (`jeu-de-des`) et un répertoire pour ce nouveau projet.

Terminer l'opération en choisissant le langage de développement dans : « Langages → gestion de C++ ... ».

Dans le menu « Divers », vous pouvez régler d'autres paramètres comme : la taille des diagrammes, l'environnement, ...

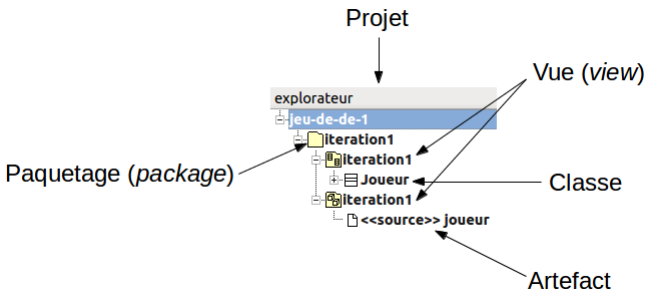
Étape 2. Effectuer le *Reverse Engeneering* C++.

Il vous faut maintenant aller dans : « Outils → Reverse Engeneering C++ ». Puis, choisissez le répetroire où se trouve le code source de la classe `Joueur`.



BOUML a détecté une classe avec 1 attribut et 4 opérations (méthodes). Il a créé automatiquement dans le projet :

- un paquetage (*package*) avec le nom du répertoire (ici `iteration1`)
- deux vues (*view*) : une vue de classe et une vue de déploiement (avec le même nom que le paquetage)
- la classe `Joueur`
- l'artefact `«source» joueur`



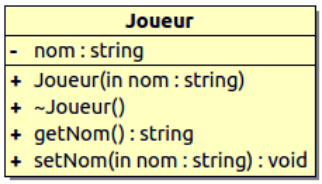
En UML, le **paquetage** (*package*) est un mécanisme général de regroupement d'éléments (cas d'utilisation, acteurs, classes, ...). Une vue de **déploiement** permettra de représenter l'architecture physique d'un système et la manière dont les composants (artefacts en UML2) sont répartis ainsi que leurs relations entre eux. Un **artefact** est une manière de définir un élément concret (un fichier, un programme, une bibliothèque ou une base de données, ...) construit ou modifié dans un projet.

Étape 3. Créer un diagramme de classes.

On va maintenant créer un diagramme de classes en cliquant avec le bouton droit de la souris sur la vue de classe et sélectionner « Nouveau diagramme de classe » en lui donnant un nom. Double cliquer ensuite sur le diagramme de classes pour l'éditer. En utilisant le *drag & drop*, placer la classe `Joueur` dans le diagramme.

Un diagramme décrit un « point de vue ». Parfois, on ne souhaite montrer que les relations entre classes ou que les attributs et leur visibilité, etc ... Le niveau de détails affiché dépendra donc de ce « point de vue ». En cliquant avec le bouton droit sur le diagramme, vous pourrez modifier les options de dessin avec BOUML.

Question 1. Produire le diagramme de classes ci-dessous pour la classe `Joueur`.



Itération 2 : génération de code de la classe `De`

La génération de code consiste à produire le code source C++ à partir d'un diagramme UML, par exemple :

- un diagramme de classes pourra produire le « squelette » des classes (déclaration et définition vide des méthodes) à compléter
- un diagramme de séquences pourra produire le corps des méthodes (certains AGL le proposent)

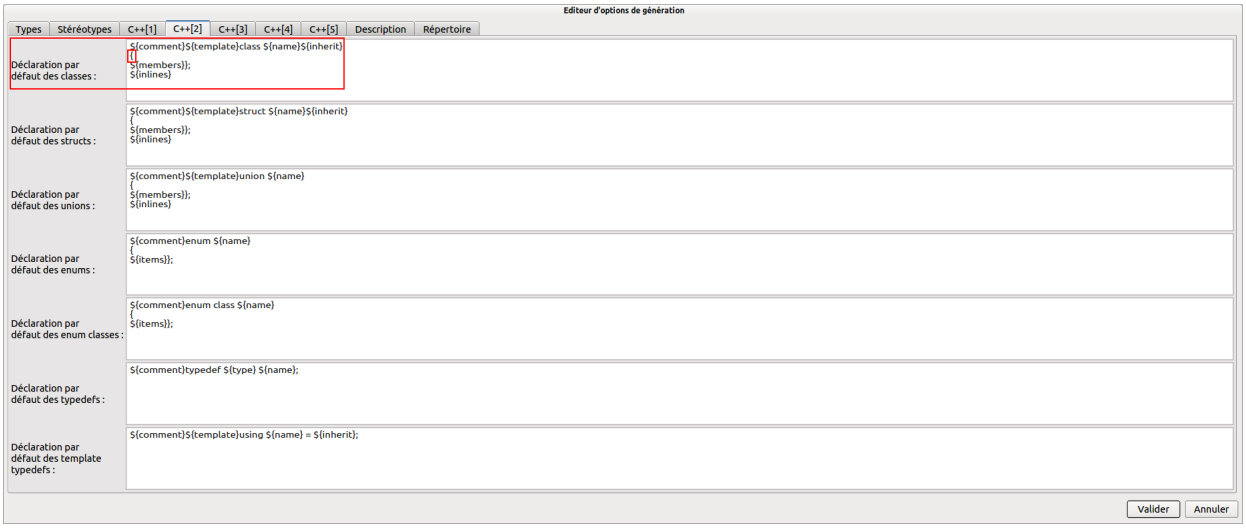
Étape 1. Éditer préalablement les options de génération.

Il faut préalablement configurer le répertoire destination dans « Projet → Éditer → Éditer les options de génération → Onglet : Répertoire → répertoire racine pour C++ ».

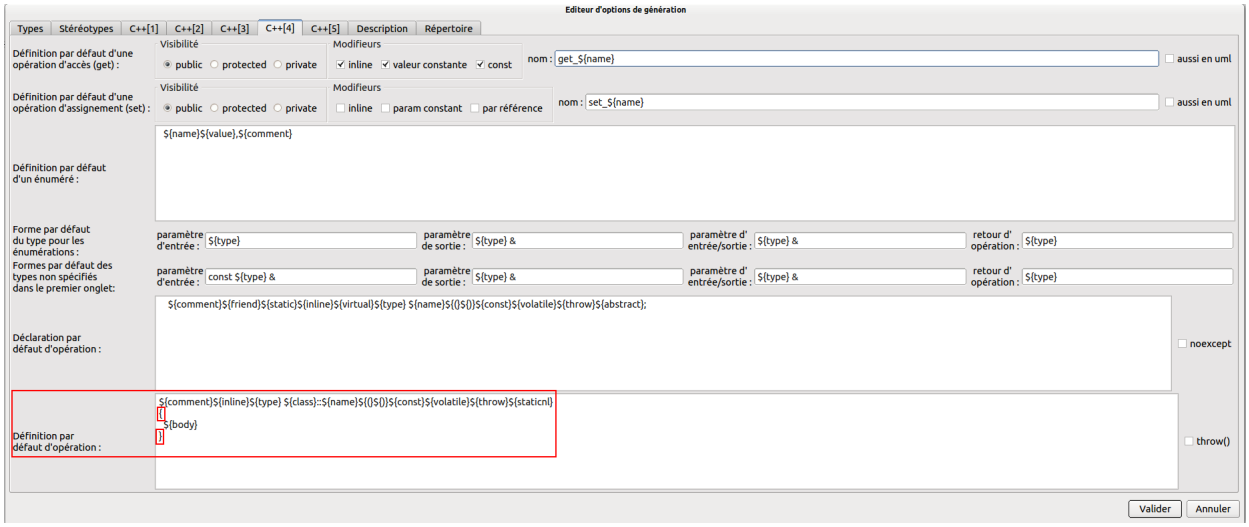
Choisir un chemin relatif est préférable si vous souhaitez déplacer plus tard le dossier de projet.

Ensuite, vous avez la possibilité d'éditer les modèles (*template*) de génération en fonction de vos règles de codage. Il vous faudra modifier au moins la génération des `{}` qui ne correspondent pas par défaut à nos règles de codage.

Pour les déclarations des classes :



Pour les définitions des méthodes :



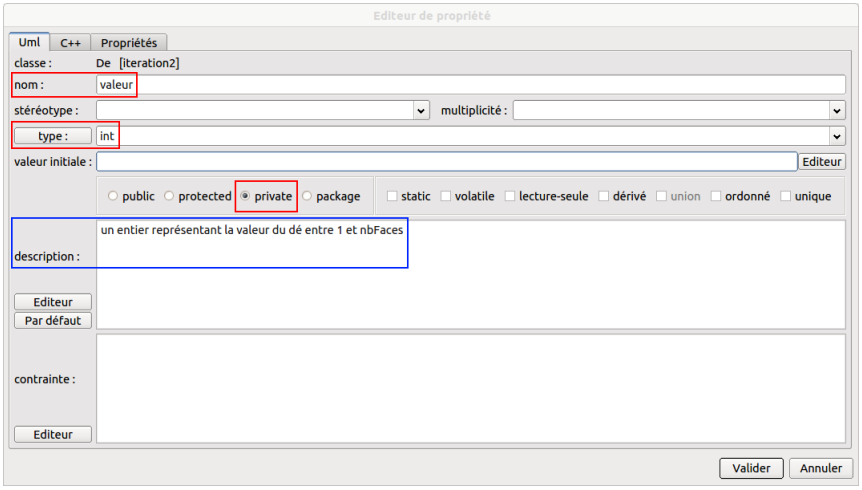
Étape 2. Créer un paquetage `iteration2`.

Étape 3. Créer une vue de classe `iteration2`.

Étape 4. Créer une classe `De`.

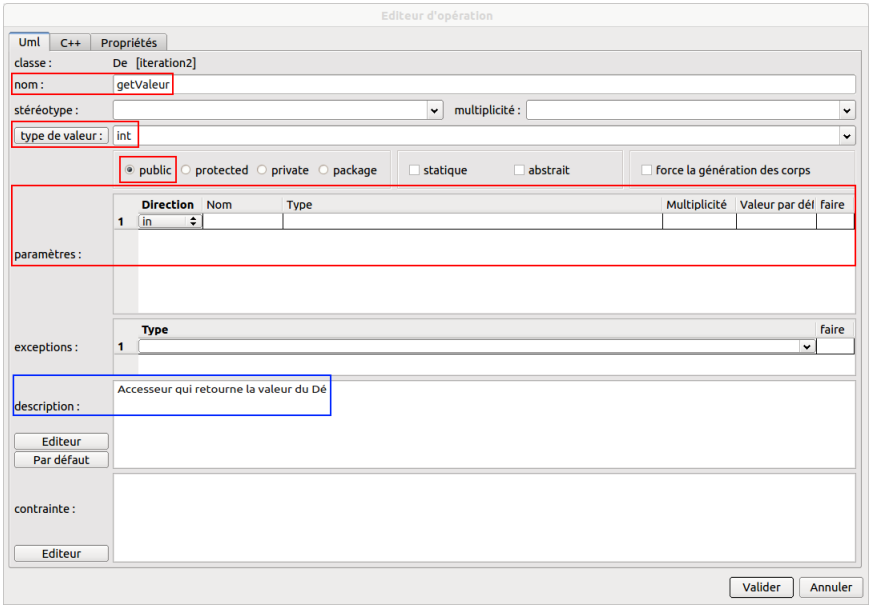
Étape 5. Ajouter les propriétés (attributs) de la classe `De`.

Maintenant, **ajouter une propriété** (un attribut) `valeur`. Il vous faudra au moins fixer : son type (ici `int`), sa visibilité (par défaut `private` pour les attributs) et éventuellement une description (un entier représentant la valeur du dé entre `1` et `nbFaces`).



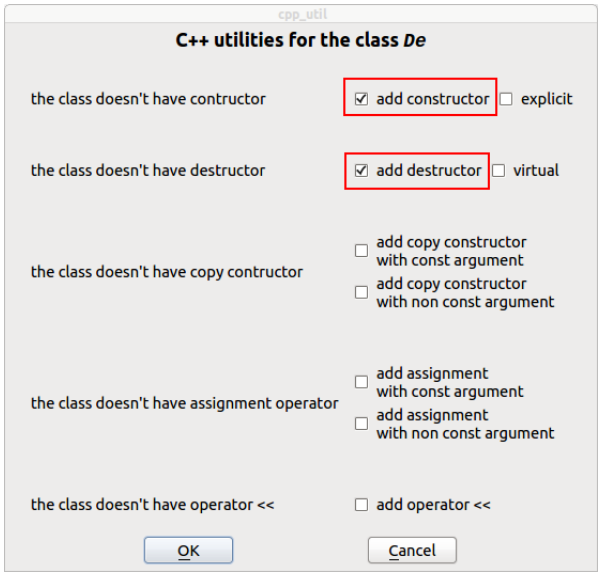
Étape 6. Ajouter les opérations (méthodes) de la classe `De`.

Ensuite, **ajouter une opération** (une méthode) `getValeur`. Il vous faudra au moins fixer : son type de retour (ici `int`), sa liste de paramètres (ici vide), sa visibilité (par défaut `public` pour les méthodes) et éventuellement une description (Accesseur qui retourne la valeur du Dé).



Étape 7. Ajouter un constructeur et destructeur à la classe `De`.

BOUML propose un outil pour créer automatiquement des constructeurs, destructeur, ... Avec le bouton droit sur la classe `De`, sélectionner « Outils → C++ utilities » :



Pour **générer du code** avec BOUML, il faut créer des artefacts. Dans BOUML, les artefacts sont associés à une vue de déploiement, qui elle-même doit être associée à une vue de classe.

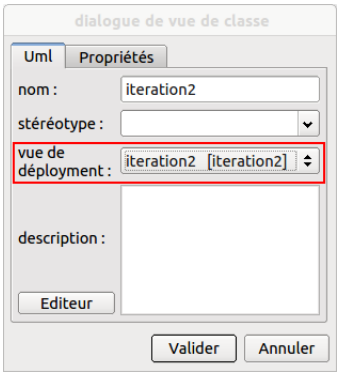
Il faudra donc réaliser les actions dans l'ordre suivant :

- créer une vue de déploiement (*Deployment View*)
- associer cette nouvelle vue de déploiement à la vue de classes de conception
- créer les artefacts associés aux classes
- puis lancer la génération de code automatique

Étape 8. Créer une vue de déploiement `iteration2`.

Étape 9. Associer la vue de classe `iteration2` à la vue de déploiement `iteration2`.

Avec le bouton droit sur la vue de classe `iteration2`, sélectionner la vue de déploiement `iteration2` :



Étape 10. Créer les artefacts «source» associés à la vue de classe `iteration2`.

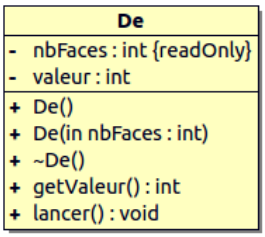
Avec le bouton droit sur la classe `De`, sélectionner « Créer un artefact source ».

Étape 11. Générer le code source C++.

Avec le bouton droit sur l'artefact «source» `De` ou sur la vue de déploiement `iteration2` (pour la totalité des artefacts «source»), sélectionner « Générer → C++ ».

À chaque nouvelle génération de code, les fichiers générés précédemment seront écrasés.

Question 2. Générer le code source C++ pour le diagramme de classe ci-dessous pour la classe De.



Pensez à organiser les membres de vos classes : constructeur(s), destructeur, accesseur(s), mutateur(s), service(s) ...

Vous devez obtenir la déclaration suivante pour la classe De :

```
#ifndef _DE_H
#define _DE_H


class De
{
  private:
    //Le nombre maximal de faces pour un dé
    const int nbFaces;

    //un entier représentant la valeur du dé entre 1 et nbFaces
    int valeur;


  public:
    De();

    De(int nbFaces);

    ~De();

    //Accesseur qui retourne la valeur du Dé
    int getValeur();

    //Lance le dé !
    void lancer();

};
#endif
```

De.h

Vous devez obtenir la définition suivante pour la classe De :

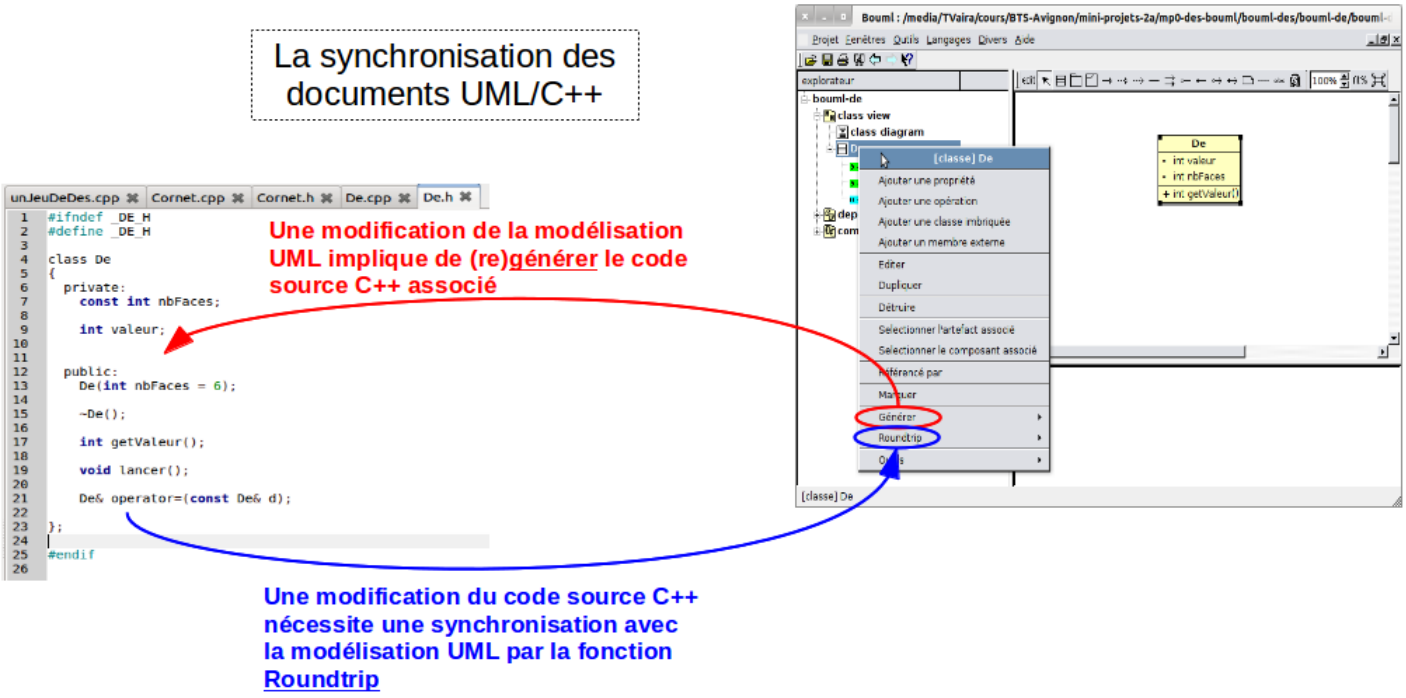
```
#include "De.h"

De::De()
```

```
{

}

De::De(int nbFaces)
{

}

De::~De()
{

}

//Accesseur qui retourne la valeur du Dé
int De::getValeur()
{

}

//Lance le dé !
void De::lancer()
{

}
```

De.cpp

Itération 3 : développement *roundtrip* du jeu de dés

Le développement *roundtrip* consiste à faire des aller-retour entre le modèle UML et le code source. L'objectif est de conserver une cohérence entre la modélisation UML et le code source et donc de propager les modifications de l'un vers l'autre.



Ce mécanisme nommé « *Roundtrip* » dans BOUML vous permet d'éditer les fichiers sources générés en dehors de BOUML puis de synchroniser les diagrammes.

Étape 1. Créer un paquetage `iteration3` et les vues associées (classe et déploiement).

La dernière version de BOUML permet de créer les deux vues en une seule action !

Étape 2. Déplacer les classes `Joueur` et `De` dans la vue de classe `iteration3`.

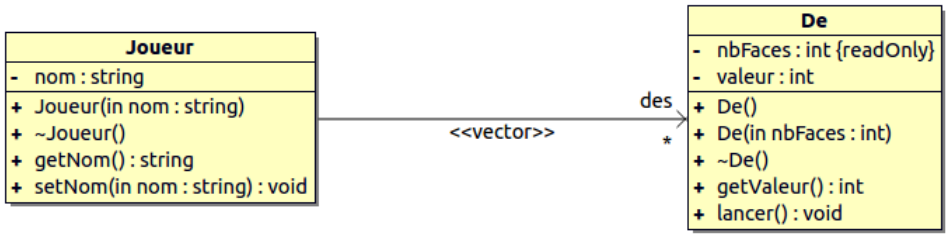
Étape 3. Déplacer les artefacts `«source» joueur` et `De` dans la vue de déploiement `iteration3`.

En fait, les iterations 1 et 2 dans BOUML peuvent être supprimées. Dans un développement itératif et incrémental, le résultat des itérations « écrasent » les précédentes. C'est le rôle d'une gestionnaire de versions d'archiver les révisions successives d'un développement.

Étape 4. Créer un diagramme de classes `jeu_de_des` avec les classes `Joueur` et `De`.

Étape 5. Ajouter une relation d'association (unidirectionnelle) entre les classes `Joueur` et `De`.

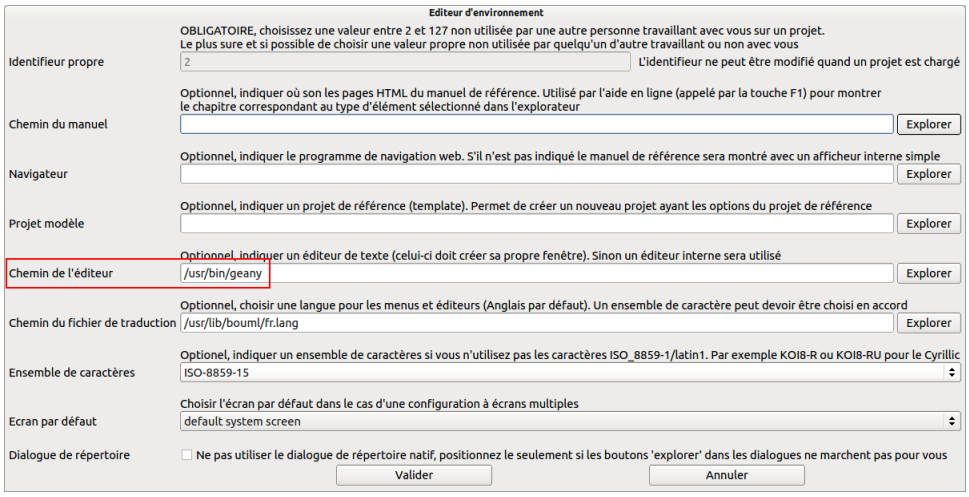
Même pour 2 objets `De`, on utilisera un `vector` :



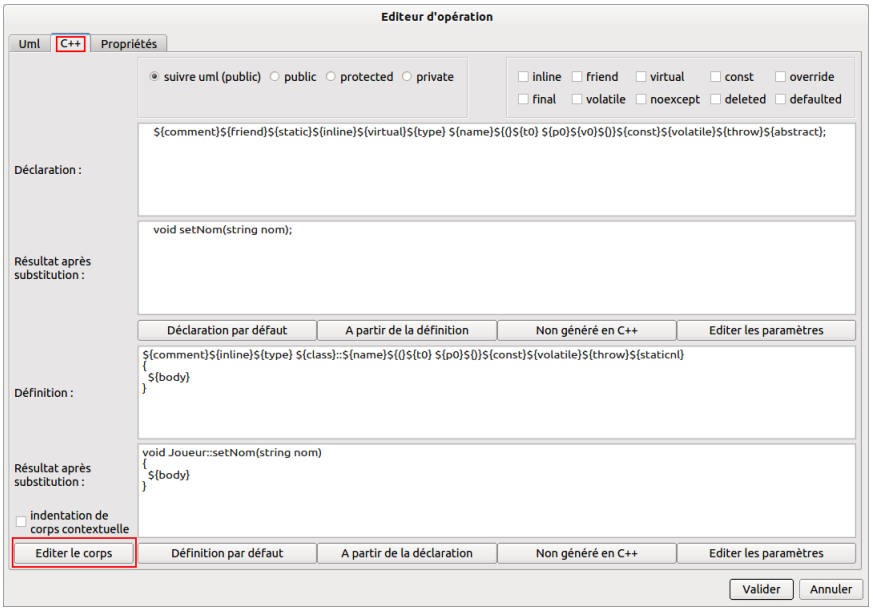
Étape 6. Générer les codes sources C++.

À partir de là, vous avez le choix de continuer à developper dans BOUML en utilisant des liens vers un éditeur ou de travailler dans un environnement externe et d'utiliser le mécanisme « *Roundtrip* ».

BOUML possède un éditeur intégré par défaut. Vous pouvez en choisir un autre dans « Divers → Etablir l'environnement → Chemin de l'éditeur » :



Vous pouvez ensuite **éditer le corps** d'une méthode :



☞ On préfère tout de même développer avec un EDI externe. C'est la méthode que l'on va utiliser pour la suite.

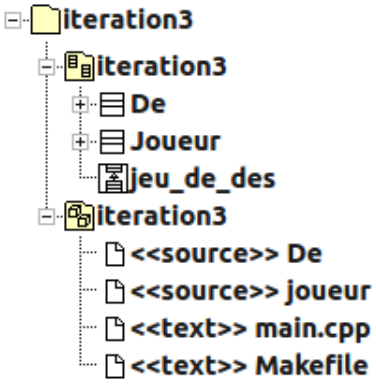
Étape 7. À partir d'un éditeur de texte externe, ajouter une méthode `void jouerAuxDes()` à la classe `Joueur`.

Étape 8. Faire un *Roundtrip* sur la vue de classe `iteration3`.

Vous devez constater que votre classe `Joueur` possède une méthode supplémentaire ! Le diagramme de classes a aussi été mis à jour. À partir de maintenant vous devez procéder par aller - retour entre BOUML et votre EDI.

Quand vous « générer du code » avec BOUML, les fichiers sources sont « écrasés ». Vous devez donc vous assurer d'avoir synchronisé avec un *roundtrip* précédemment.

Étape 9. Créer et générer les artefacts `«text» main.cpp` et `Makefile`.



Vous pouvez ponctuellement utiliser l'éditeur interne pour créer un contenu pour ces fichiers. Les fichiers ne doivent pas être vides pour être générés par BOUML.

Question 3. Finaliser le développement du logiciel pour pouvoir « jouer une partie de dés ».

Les dés sont des objets génériques utilisables dans toutes sortes de jeu. La classe `De` posséde un constructeur par défaut (qui fixe un nombre de faces égal à 6). Pour des soucis de réutilisation, elle dispose aussi d'un constructeur auquel on pourra passer le nombre de faces désirées pour un dé.

La classe `De` posséde une méthode `lancer()` qui aura pour rôle de déterminer de façon pseudo-aléatoire (cf. les fonctions `rand()` et `srand()`) la valeur du dé (une valeur comprise entre 1 et 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` définit 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 la 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.

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.

Bonus : un Cornet pour jouer avec des dés

L'affectation de la responsabilité de lancer les dés et de les additionner empêche de réutiliser ce service dans d'autres jeux. 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 : c'est-à-dire 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.

```
Cornet cornet; // par défaut notre cornet est composé de 2 dés

cornet.lancer();

int total = cornet.getTotal();

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

Exemple d'utilisation d'un objet Cornet