Fiche méthodologique de projet

I.2 Implémenter les relations entre classes

Objectif

Coder les associations, agrégations et composition.

Préambule

La plupart des langages de programmation se ressemblent mais même ces langages diffèrent selon le degré de prise en compte des concepts orientés objets. C'est le cas pour les langages majeurs comme C++, Java et C#.

Les exemples sont donnés pour le C++.

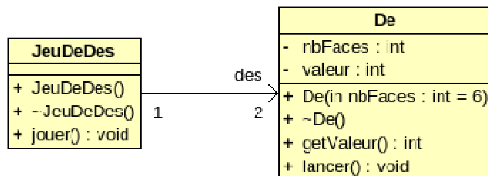
Méthode

La plupart des langages orientés objets implémentent les associations avec des pointeurs d'objets ou un type référence sur un objet.

Les relations d'association et d'agrégation peuvent être implémentées de la même manière en utilisant un pointeur. Par contre en cas de *reverse engineering*, le logiciel d'AGL ne saura pas les distinguer.

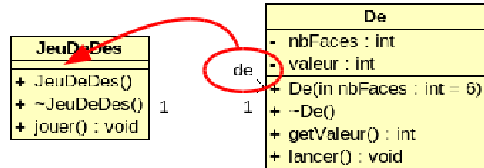
Pour les relations de composition, il faut prendre en compte que le composite est responsable du cycle de vie des parties. On implémente généralement une agrégation par valeur. Une utilisation des pointeurs est tout à fait possible (en assurant la construction et la destruction des parties à l'intérieur du composite) mais en cas de *reverse engineering*, l'AGL ne saura reconnaître une composition. La notion de classe imbriquée peut s'avérer une solution intéressante pour traduire une composition.

Association



Dans la classe JeuDeDes :

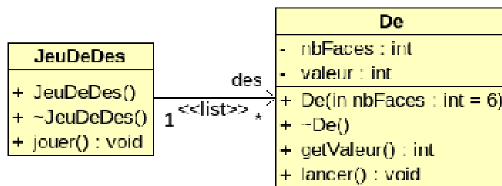
```
De * des[2];
```



Dans la classe JeuDeDes :

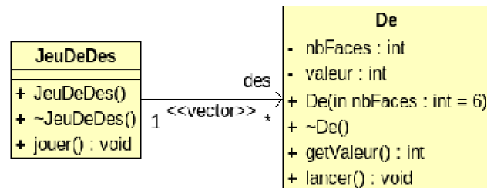
```
De * de;
```

Remarque : le sens de navigation unidirectionnel implique que JeuDedes connaît De mais la réciproque n'est pas vrai.



Dans la classe JeuDeDes :

```
list<De *> des;
```

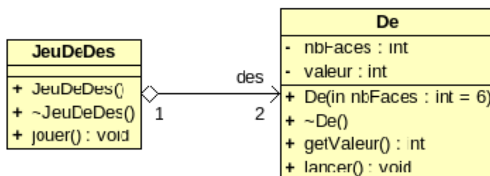


Dans la classe JeuDeDes :

```
vector<De *> des;
```

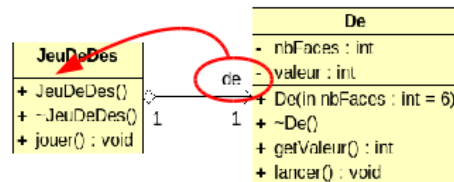
Remarque : pour des multiplicités *, il faut utiliser un conteneur. Bouml propose les conteneurs de la STL : list, map, vector et set.

Agrégation



Dans la classe JeuDeDes :

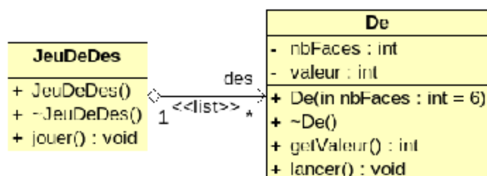
```
De * des[2];
```



Dans la classe JeuDeDes :

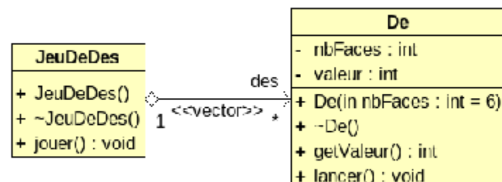
```
De * de;
```

Remarque : bouml « code » de la même manière une association et une agrégation.



Dans la classe JeuDeDes :

```
list<De *> des;
```

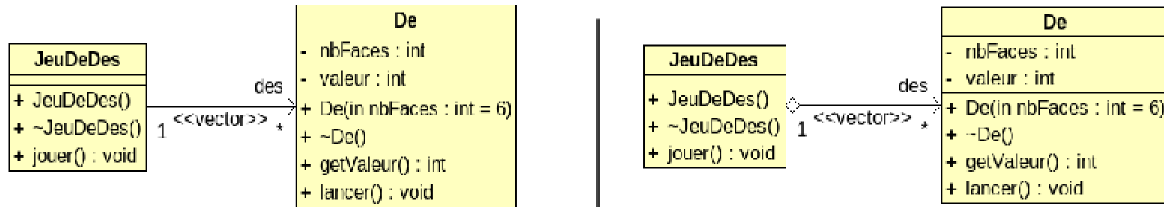


Dans la classe JeuDeDes :

```
vector<De *> des;
```

Remarque : pour des multiplicités *, il faut utiliser un conteneur. Bouml propose les conteneurs de la STL : list, map, vector et set.

Association - Agrégation



Dans le fichier JeuDeDes.h :

```
private:
      vector<De *> des;

public:
  void addDes(De *de);
  void removeDes(De *de);
  const vector<De *> & getDes() const ;
  void setDes(vector<De *> & des);
```

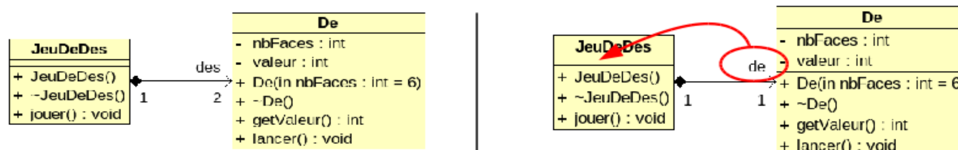
Dans le fichier JeuDeDes.cpp :

```
void JeuDeDes::addDes(De *de) {
  des.push_back(de);
}
```

Dans le fichier JeuDeDes.cpp :

```
void JeuDeDes::removeDes(De *de) {
  int i, size = des.size();
  for ( i = 0; i < size; ++i) {
       De * item = des(i);
       if(item == de) {
         vector<De *>::iterator it = des.begin() + i;
         des.erase(it);
         return;
       }
   }
}

// peut être inline
const vector<De *> & JeuDeDes::getDes() const {
  return des;
}

// pas obligatoire
void JeuDeDes::setDes(vector<De *> & des)
{
  this->des = des;
}
```

Composition



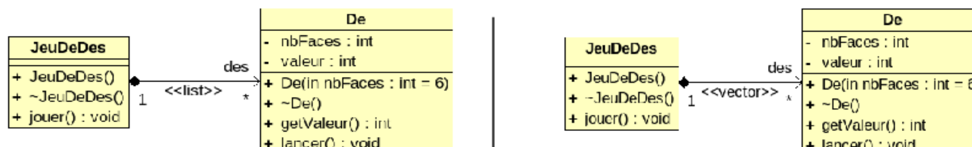
Dans la classe JeuDeDes :

```
De des[2];
```

Dans la classe JeuDeDes :

```
De de;
```

Remarque : Les objets De n'existent que si l'objet JeuDeDes existe : c'est donc une composition. Lorsque l'objet JeuDeDes sera détruit, les objets De le seront aussi. L'objet JeuDeDes est donc responsable de la création et la destruction des objets De. Le code généré n'est pas le même que pour une association ou une agrégation.



Dans la classe JeuDeDes :

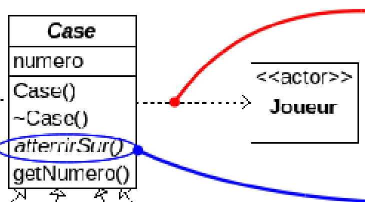
```
list<De> des;
```

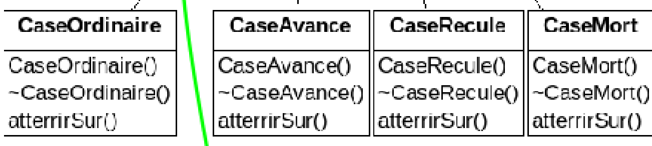
Dans la classe JeuDeDes :

```
vector<De> des;
```

Remarque : pour des multiplicités *, il faut utiliser un conteneur. Bouml propose les conteneurs de la STL : list, map, vector et set.

Case est une classe abstraite qui possède une méthode virtuelle pure : atterrirSur()





Fichier : Case.h — **Déclaration de la classe *Case***

```
#ifndef _CASE_H
#define _CASE_H
#include "Joueur.h"

class Case
{
 public:
  Case(int numero);
  virtual ~Case();
  virtual void atterrirSur(Joueur *j) = 0;

 protected:
  int numero;

 public:
  virtual int getNumero();
};

#endif
```

Fichier : Case.cpp — **Définition de la classe *Case***

```
#include "Case.h"

Case::Case(int numero)
{this->numero = numero;}

Case::~Case() {}

int Case::getNumero()
{return numero;}
```

Fichier : CaseOrdinaire.h — **Déclaration de la classe CaseOrdinaire**

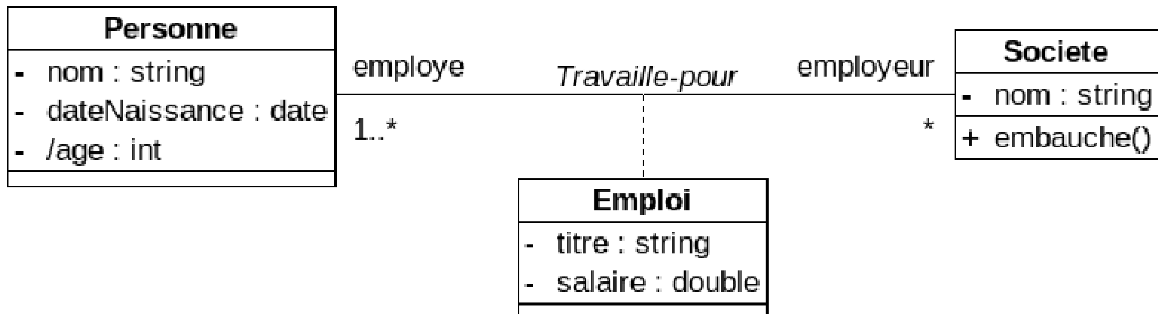
```
#ifndef _CASEORDINAIRE_H
#define _CASEORDINAIRE_H


class CaseOrdinaire : public Case
{
  public:
    CaseOrdinaire(int numero);
    ~CaseOrdinaire();
    virtual void atterrirSur(Joueur *j);
};

#endif
```

Fichier : CaseOrdinaire.cpp — **Définition de la classe CaseOrdinaire**

```
#include "Case.h"
#include "Joueur.h"
#include "CaseOrdinaire.h"

CaseOrdinaire::CaseOrdinaire(int numero
):Case(numero)
{}

CaseOrdinaire::~CaseOrdinaire()
{}

void CaseOrdinaire::atterrirSur(Joueur *j)
{
   joueur->deplacerPion(numero);
}
```

Exemple d'implémentation de l'héritage en C++

Classe d'association



Le concept de classe d'association n'existe pas dans les langages de programmation objet.
Il faut donc le traduire en le transformant en classe normale :

```
class Emploi
{
  private:
    string     titre;
    double     salaire;
    Personne * employe;
    Societe  * employeur;
};
```