Génie Logiciel TD n° 5

Méta-Propriétés UML et correspondance avec C++

Objectif du TD

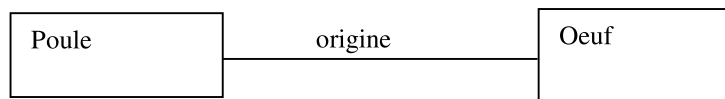
Maîtriser les méta-propriétés UML (propriétés des propriétés UML) :
- propriété de classe et de propriété d'instance,
- visibilité des propriétés,
- attribut intrinsèque et attribut associatif.

Correspondance entre UML et POO :
- associations UML et pointeurs C++,

Exercice MOINS-MOINS

Cet exercice pose le problème concret du lien entre deux classes C++, de la correspondance entre association UML et attributs pointeurs C++.

On part de deux classes, `Poule` et `Œuf` associée par l'association " `origine` ". La poule a pour origine l'œuf qui a pour origine la poule… Pour simplifier, on part du diagramme de classe ci-dessous et on suppose que l'association binaire " origine " est un-un.



a) Quelles sont les deux manières de traduire le rôle Poule-Œuf de l'association " origine " en C++ ?

b) Comment s'appellent les attributs `monOeuf` et `maPoule` ?

c) g++ ne compile pas le programme C++ suivant :

```
class Poule  { public :
      Oeuf * monOeuf ;
      void maMethode() { ;  }
} ;
class Oeuf  { public :
      Poule * maPoule ;
      void maMethode() { ; }
} ;
void main() {
      Poule * p = new Poule(); Oeuf * o = new Oeuf();
      p->monOeuf = o; o->maPoule = p;
      p->maMethode(); o->maMethode();
}
```

Pourquoi ? Rajouter l'instruction convenable pour qu'il soit compilable par g++.

d) Le programme étant modifié convenablement, on remplace le corps de la méthode `void maMethode()` de la classe Poule par `monOeuf->maMethode() ;` A nouveau, g++ ne compile pas le programme.
Pourquoi ? Modifier le programme pour qu'il soit compilable par g++.

e) g++ ne compile pas le programme C++ suivant :

```
class Poule  { public :
      Oeuf monOeuf ;
      void maMethode() { ;  }
} ;
class Oeuf  { public :
      Poule maPoule ;
      void maMethode() { ; }
} ;
void main() {// idem
}
```

Pourquoi ? Y a-t-il une solution ?

Exercice " PROPRIETE DE CLASSE ET PROPRIETE D'INSTANCE "

1) Que signifie " PROPRIETE DE CLASSE " ? " PROPRIETE D'INSTANCE " ?

Pour chaque propriété des classes UML suivantes, nous avons volontairement omis de la souligner quand cela était nécessaire.

2) En vous aidant du nom de la propriété et de sa signification, indiquer si la propriété est "d'instance" ou bien "de classe".

Objet
nombreObjets : int ; mesObjets : Liste ; numéro : int ;

Château
muraille : Muraille ; listeTours : Liste ; listeChateaux : Liste ;
void imprimerMuraille() ; void imprimerTours() ; void imprimerChateaux() ; void imprimer() ;

On peut souligner void imprimerChateaux() ; ou pas ; quelles sont les deux significations ?

On peut souligner void imprimer () ; ou pas ; quelles sont les deux significations ?

Exercice MOT CLE STATIC

1) Que signifie " PROPRIETE DE CLASSE " ? " PROPRIETE D'INSTANCE " ?

2) Quel mot-clé C++ utilise-t-on pour spécifier une propriété de classe ? et d'instance ?

3) Le programme C++ suivant n'est pas compilable par g++. Pourquoi ? Rajouter les deux lignes nécessaires pour qu'il le soit.

```
#include <iostream.h>
class R { public:
  static R * notreR;
  static int j;
  int i;
  R * monR;
  R(int a) { i = a; monR = NULL; j++; }
  static void p(R * r) {
    j = 0;
    notreR = new R(0);
    m(notreR);
  }
  static void q(R * r) {
    cout << "R::notreR = "; n(notreR);
    cout << "R::j = " << j << endl;
  }
  static void m(R * r) {
      cout << "R, i = " << r->i << endl;
  }
  static void n(R * r) {
    m(r);
    cout << "monR = ";
    if (r->monR!=NULL) r->monR->m(r->monR);
    else cout << "NULL" << endl;
  }
};

class S { public:
  static void main() {
    R::p(NULL);
    R * r1 = new R(1);
    R::notreR->monR = r1;
    R * r2 = new R(2);
    cout << "r2 = "; R::n(r2);
    r1->monR = r2;
    cout << "r1 = "; R::n(r1);
    R::q(r1);
  }
};

void main() {
  S::main();
}
```

4) Maintenant que g++ compile ce programme, quelle est sa sortie ?

5) Le mot-clé 'static' est correctement placé en ce qui concerne les attributs mais pas pour les méthodes. Reprogrammer ce programme pour que sa sortie soit identique, que les mot-clé 'static' soient partout correctement placés, et enfin que les paramètres inutiles soient enlevés.

6) Donner le pour et le contre de la phrase : " le constructeur est une méthode d'instance ".

Exercice “ ABCDE ”

On a les déclarations et définitions suivantes:

```
#include <iostream.h>
class A { public :
      A * monA ;
      virtual void maMethode();
      virtual void saMethode();
} ;
class C;
class B : public A { public :
      C * monC ;
      void saMethode();
} ;
class C : public A { public :
      B * monB ;
      void maMethode();
      void saMethode();
} ;
class E;
class D : public B { public :
      E * monE;
      void maMethode();
      void saMethode();
} ;
class E : public B { public :
      D * monD ;
      void maMethode();
} ;

void A::maMethode() { cout << " A ";  }
void A::saMethode() { monA->maMethode();  }
void B::saMethode() { monC->maMethode() ;  }
void C::maMethode() { cout << " C ";  }
void C::saMethode() { monB->maMethode() ;  }
void D::maMethode() { cout << " D "; }
void D::saMethode() { monE->maMethode() ;  }
void E::maMethode() { cout << " E " ;  }

void main(int argc, char * argv[]) {
      A * a = new A(); B * b = new B(); C * c = new C(); D * d = new D(); E
* e = new E();
      a->monA = b; b->monA = c ; c->monA = d; d->monA = e; e->monA = a;
      b->monC = c; c->monB = b; d->monC = c; e->monC = c; d->monE = e; e-
>monD = d;
      a->maMethode();  b->maMethode();  c->maMethode();  d->maMethode();
e->maMethode();
      a->saMethode();  b->saMethode();  c->saMethode();  d->saMethode();
e->saMethode();
}
```

1) Dessiner les diagrammes de généralisation, de classe-association et d’instances correspondant.

2) Donner la sortie du programme.

Exercice " Reverse engineering de C++ à UML"

Le " Reverse engineering " est une technique de développement informatique qui consiste à concevoir un logiciel à partir du code source d'un logiciel existant. " engineering " signifie concevoir et " reverse " signifie que l'on travaille en sens inverse au sens habituel : en général dans le cycle en V, la conception précède la production du code source.

Cet exercice est un exemple de reverse engineering. A partir d'un code C++, vous devez retrouver la conception UML. On a les déclarations et définitions C++ suivantes:

```
#include <iostream.h>
class Objet { public:
      static int nombre_objets = 0;
      int numero ;
      Objet();
      ~Objet();
};
class Porte : public Objet { public:
      Porte();
      ~Porte();
};
class Batiment : public Objet { public:
      Porte * porte;
      int surface;
      Batiment(int);
      ~Batiment();
};
class Tour : public Objet { public:
      int hauteur;
      Muraille * muraille;
      Tour(int, Muraille *);
      ~Tour();
};
class Donjon : public Tour { public:
      Chateau * chateau;
      Donjon(Chateau *, int);
      ~Donjon();
};
class Muraille : public Objet { public:
      int longueur;
      Tour * tour1;
      Tour * tour2;
      Chateau * chateau;
      Muraille(int, int, int, Chateau *);
      ~Muraille();
};
class Chateau : public Batiment { public:
      Donjon * donjon;
      Muraille * muraille;
      Chateau(int, int, int, int, int);
      ~Chateau();
};

int Objet::nombre_objets = 0 ;
Objet::Objet() {
      cout << "++ Objet debut" << endl;
      nombre_objets++;
      numero = nombre_objets ;
      cout << "++ Objet fin" << nombre_objets << endl;
}
Objet::~Objet() {
      cout << "-- Objet debut " << endl;
      nombre_objets--;
      cout << "-- Objet fin " << endl;
}
Porte::Porte() {
      cout << "++ Porte " << endl;
}
```

```
Porte::~Porte() {
      cout << "-- Porte " << endl;
}
Batiment::Batiment(int s) {
      cout << "++ Batiment debut" << endl;
      porte = new Porte();
      surface = s;
      cout << "++ Batiment fin" << surface << endl;
}
Batiment::~Batiment() {
      cout << "-- Batiment debut" << endl;
      delete porte;
      cout << "-- Batiment fin" << endl;
}
Tour::Tour(int h, Muraille * m) {
      cout << "++ Tour debut" << endl;
      hauteur = h;
      muraille = m;
      cout << "++ Tour fin " << hauteur << endl;
}
Tour::~Tour() {
      cout << "-- Tour " << endl;
}
Donjon::Donjon(Chateau * c, int h) : Tour(h, NULL) {
      cout << "++ Donjon debut" << endl;
      chateau = c;
      cout << "++ Donjon fin" << endl;
}
Donjon::~Donjon() {
      cout << "-- Donjon " << endl;
}
Muraille::Muraille(int l, int h1, int h2, Chateau * c) {
      cout << "++ Muraille debut" << endl;
      longueur = l;
      tour1 = new Tour(h1, this);
      tour2 = new Tour(h2, this);
      chateau = c;
      cout << "++ Muraille fin " << longueur << endl;
}
Muraille::~Muraille() {
      cout << "-- Muraille debut" << endl;
      delete tour1;
      delete tour2;
      cout << "-- Muraille fin" << endl;
}
Chateau::Chateau(int l, int h1, int h2, int hd, int s) : Batiment(s)
{
      cout << "++ Chateau debut" << endl;
      muraille = new Muraille(l, h1, h2, this);
      donjon = new Donjon(this, hd);
      cout << "++ Chateau fin" << endl;
}
Chateau::~Chateau() {
      cout << "-- Chateau debut" << endl;
      delete muraille;
      delete donjon;
      cout << "-- Chateau fin" << endl;
}
main() {
      Chateau chateau(400, 40, 60, 80, 10000);
}
```

1) Dessiner le diagramme de généralisation UML.

2) Dessiner un diagramme de classes UML. Placer les ordres de multiplicité.

3) Dessiner un diagramme d'objets UML correspondant à l'état du programme après la déclaration du château dans le programme principal main() et avant la fin de celui-ci.