CHAPITRE 11

Identification et spécification des opérations - III

Des opérations, des méthodes et des interfaces

On distingue:

- **Une fonction:** un regroupement d'instructions qui peuvent être exécutées par un appel,
- **Une opération**: en UML, une opération est un service qui peut être demandé à un objet afin de produire un comportement,
- **Une méthode**: est l'implantation logicielle d'une opération, qui spécifie l'algorithme ou la procédure de l'opération,
- **Une interface**: est un ensemble d'opérations visibles de l'extérieur d'un objet, en UML, elle peut être associée à une classe, un type ou un paquetage.

Syntaxe UML

Syntaxe d'une opération en UML

[visibilité] nom [(liste-des-paramètres)] [: type-de-retour][{propriétés}]

Syntaxe des paramètres:

[direction] nom : type [= valeur-de-défaut]

Les directions définies sont:

- *in* (entrée)
- *out* (sortie)
- *inout* (entrée-sortie)

Référence: Larman, fig. 16.1

Définition de l'interface d'une classe

Règle: Viser la définition d'interfaces de classes qui soient complètes et minimales.

Complète: Une interface complète permet à un client de faire tout ce qu'il peut raisonnablement vouloir faire.

Minimale: Une interface minimale contient aussi peu d'opérations que possible et, en particulier, ne possède aucune paire d'opérations dont les fonctionnalités se chevauchent.

Définition de l'interface d'une classe

Complète d'accord, mais pourquoi minimale ?

- Compréhension: un grand nombre d'opérations augmente la difficulté de compréhension de la classe pour les clients potentiels,
- Facilité d'utilisation: une prolifération d'opérations peut amener de la confusion quant au choix de l'opération la plus appropriée,
- Entretien: beaucoup d'opérations augmente les problèmes de maintenance,
- Efficacité: des interfaces comprenant trop d'opérations peuvent augmenter significativement le temps total de compilation d'un système.

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Définition de l'interface d'une classe

Exemple: construction d'une classe permettant de représenter des nombres rationnels.

Rationnel
- m_numerateur : int - m_denominateur : int
+ Rationnel(num : int, den : int)

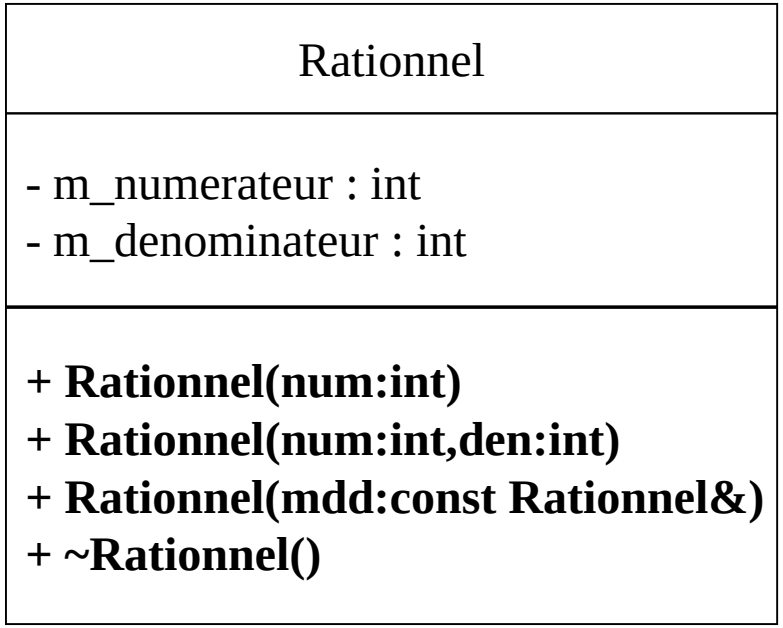
Avec ce seul constructeur, l'interface est minimale, mais le dénominateur n'est pas optionnel.

```
Rationnel une_demie(1, 2);
```

Définition de l'interface d'une classe

Il est permis de déclarer plusieurs constructeurs

- Chaque constructeur permet une configuration spécifique de l'objet,
- ATTENTION: Un constructeur à un seul argument peut être invoqué de façon implicite par le compilateur pour convertir entre 2 types,
  - Le mot-clé **`explicit`** évite les conversions implicites de type
- Le constructeur de copie serait nécessaire pour gérer correctement les données si la classe contenait un/des pointeur(s),
- Le destructeur serait nécessaire pour libérer les données si la classe contenait un/des pointeur(s).



POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Définition de l'interface d'une classe

Une approche équivalente, mais plus compacte, consiste à utiliser des valeurs de défaut pour les paramètres du constructeur.

Rationnel
- m_numerateur : int - m_denominateur : int
+ Rationnel(num : int=0, den : int=1) + Rationnel(const Rationnel&) + ~Rationnel()

Définition de l'interface d'une classe

L'interface actuelle est-elle complète ?

Il est essentiel de pouvoir accéder/modifier les propriétés de l'objet :

```
// récupérer les données
numerateur() : int {isQuery}
denominateur() : int {isQuery}
```

La sémantique de la propriété *isQuery* est essentiellement la même que celle du mot-clé C++ *const* appliqué à une méthode.

```
// modifier les données
setNumerateur( num : int ) : void
setDenominateur( den : int ) : void
```

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Définition de l'interface d'une classe

Rationnel
- m_numerateur : int - m_denominateur : int
+ Rationnel(num : int=0, den : int=1) + Rationnel(const Rationnel&) + ~Rationnel() **+ numerateur() : int {isQuery}** **+ denominateur() : int {isQuery}** **+ setNumerateur(num : int) : void** **+ setDenominateur(den : int) : void**

Définition de l'interface d'une classe

L'interface actuelle est-elle complète ?

Si un client veut utiliser la classe, sera-t-il capable de le faire efficacement ?

- Quelles sont les requêtes typiques auxquelles on peut s'attendre ?
- Sont-elles réalisables à l'aide de l'interface actuelle ?
- Les requêtes les plus probables sont-elles faciles à effectuer ?

Définition de l'interface d'une classe

Il faut pouvoir récupérer la valeur du nombre!

Deux approches:

1. Par une méthode explicite

```
double valeur(void) const;
```

2. Par un opérateur de conversion

```
operator double(void) const;
```

La première approche doit toujours être utilisée explicitement (plus sûr) alors que la deuxième peut être utilisée de façon implicite par le compilateur (plus automatique, moins sûr).

Définition de l'interface d'une classe

L'interface d'une classe désigne l'ensemble des méthodes utilisable par les « clients » de la classe,

- L'interface inclut seulement les méthodes publiques.

Lors de la conception d'une classe, il est souvent nécessaire d'ajouter des méthodes internes, qui ne sont pas faites pour être utilisées à l'extérieur de la classe,

- Les méthodes internes devraient être déclarées privées ou protégées.

Exemple: une méthode interne pour **simplifier** le nombre rationnel après une opération.

Définition de l'interface d'une classe

Rationnel
- m_numerateur : int - m_denominateur : int
+ Rationnel(num : int=0, den : int=1) + Rationnel(const Rationnel&) + ~Rationnel() + numerateur() : int {isQuery} + denominateur() : int {isQuery} **+ valeur() : double {isQuery}** + setNumerateur(num : int) : void + setDenominateur(den : int) : void **- simplifier() : void**

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Définition de l'interface d'une classe

L'interface actuelle est-elle complète ?

D'autres points à considérer:

- Des opérations arithmétiques sur les nombres rationnels,
- Les opérateurs d'entrées/sorties,
- Les opérateurs relationnels (<,>,==,…).

Opérations membres, globales et amies

Quels sont les règles qui permettent de décider si une fonction doit être membres de la classe, globale, ou globale et amie.

Si le polymorphisme est nécessaire, la fonction doit être virtuelle et donc être membre de la classe.

```
class Rationnel {
  private:
     int m_numerateur;
     int m_denominateur;

  public:
     Rationnel( int num = 0, int den = 1 );
     virtual int numerateur() const;
     virtual int denominateur() const;
     …
};
```

Opérations membres, globales et amies

Pour certaines opérations, c'est moins clair :

Comment définir les opérations arithmétiques agissant sur la classe Rationnel ?

- Fonctions membres ?
- Fonctions globales ?
- Fonctions globales amies ?

Opérations membres, globales et amies

Les opérateurs arithmétiques comme des membres

```
class Rationnel {
public:
   ...
   Rationnel operator*( const Rationnel& mdd );
};
Rationnel unHuitieme( 1, 8 );
Rationnel uneDemie( 1, 2 );
```

Essayons l'opérateur:

```
Rationnel resultat = uneDemie * unHuitieme; // Ok
resultat = resultat * unHuitieme; // Ok
```

Opérations membres, globales et amies

Essayons maintenant l'opérateur en arithmétique mixte:

```
resultat = uneDemie * 2; // Ok
resultat = 2 * uneDemie; // erreur !
```

Pourquoi ?

- Une conversion implicite de type est effectuée sur 2 lorsqu'il est placé à droite (argument de l'opérateur),
- À peu près équivalent à:

```
const Rationnel temp(2); // convertir 2 en rationnel
resultat = uneDemie * temp;
```

- Ou

```
resultat = uneDemie.operator*(temp);
```

Opérations membres, globales et amies

Aucune conversion de type n'est effectuée sur l'objet qui invoque une fonction membre – l'objet doit être du bon type.

- Les conversions de type ne sont effectuées que sur les paramètres inscrits dans une liste de paramètres.

```
resultat = 2 * uneDemie;


// à peu près équivalent à:
2.operator*(uneDemie); // Aucune conversion de type n'est
                       // possible ici
```

Opérations membres, globales et amies

Les fonctions globales traitent tous leurs arguments de la même façon:

```
class Rationnel {
... // ne contient pas d'opérateur arithmétique
};

Rationnel operator*( const Rationnel& mdg,
                     const Rationnel& mdd )
{
  return Rationnel{ mdg.numerateur()*mdd.numerateur(),
                    mdg.denominateur()*mdd.denominateur() };
}
```

Opérations membres, globales et amies

Essayons la fonction globale:

```
Rationnel unQuart(1,4);
Rationnel resultat;
resultat = unQuart * 2; // Ok
resultat = 2 * unQuart; // aussi Ok
```

Dans ce cas, la fonction globale n'a même pas besoin d'être amie de la classe puisque toutes les données nécessaires sont accessibles par l'interface publique de la classe.

Opérations membres, globales et amies

Que faire pour les opérateurs d'entrées/sorties ?

```
class Rationnel {
public:
    Rationnel( int num = 0, int den = 1, string nom = "" );
    virtual int numerateur() const;
    virtual int denominateur() const;
    …
    istream& operator>>(istream& input);
   ostream& operator<<(ostream& output);

private:
    int m_numerateur;
    int m_denominateur;
    string m_nom;
};
```

Définition de l'interface d'une classe

Essayons:

```
int x;
Rationnel une_demie(1,2);

cin >> x;   // tel qu'espéré
une_demie >> cin;   // correct, mais inhabituel

cout << x;  // tel qu'espéré
une_demie << cout;  // aussi inhabituel
```

Les opérateurs d'entrées/sorties **doivent toujours** être déclarés comme des **fonctions globales** afin de les utiliser de façon standard (comme avec les types primitifs).

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Définition de l'interface d'une classe

Mais en général, les opérateurs d'entrées/sorties doivent avoir accès au données privées de la classe:

```
istream& operator>>(istream& input, Rationnel& r )
{
   // lire les données et initialiser les attributs
   return input;
}
ostream& operator<<(ostream& output, const Rationnel& r)
{
   return output << r.m_numerateur << “,” << r.m_denominateur  << “,” << r.m_nom;
}
```

Définition de l'interface d'une classe

Cette combinaison fait que les opérateurs d'entrées/sorties doivent généralement être déclarés comme des fonctions globales amies:

- Globales pour respecter l'ordre habituel des paramètres
- Amies pour avoir accès aux attributs privés de la classe

Définition de l'interface d'une classe

Algorithme pour choisir entre des opérations membres, globales ou globales et amies:

```
Si( f doit être virtuelle )
   f doit être une fonction membre
Sinon, si( f est un opérateur d'E/S ) {
   f doit être une fonction globale
   si( f doit accéder aux attributs privés )
      f doit être amie
}
Sinon, si( f doit permettre la conversion de type
           sur tous ses arguments ) {
   f doit être une fonction globale
   si( f doit accéder aux attributs privés )
      f doit être amie
}
Sinon,
   f est une fonction membre.
```

Définition de l'interface d'une classe

Comment modéliser une fonction globale avec UML ?

UML ne permet pas de définir une opération à l'extérieur d'une classe; deux alternatives, selon les règles d'accès dont on a besoin pour la fonction globale:

- Une fonction globale qui doit être amie peut être déclarée comme une opération de classe (statique),
- Une fonction globale qui n'a pas besoin d'accéder aux attributs privés de la classe peut être déclarée comme une opération de classe dans une classe auxiliaire,
- Placer les fonctions globales dans un *namespace*.
- Comment modéliser en UML une opération dont la sémantique fait partie du langage, comme un opérateur?