Réutiliser des objets

Introduction aux notions
d’ héritages et de dérivation

Limite des objets « simples »

- Notre compteur
  - Compte de 1 à *max*
  - Revient à 0 une fois *max* atteint

- ***Problème 1*** : On souhaite un compteur qui peut croitre de 0 à max mais aussi décroitre de max à 0
  - Il faut réécrire une classe CounterBis…

- ***Problème 2*** : On souhaite un compteur qui fait quelque chose quand max est atteint ?
  - Il faut réécrire une classe CounterTer…

…

Le sous-classement

Définition : le sous-classement consiste à substituer à une fonction ou un module une autre fonction ou un autre module implantant au moins les mêmes fonctionnalités que la fonction initiale ou le module initial

Concept utilisé de manière générique dans les logiciels.

Exemple :

Handler d'exception en C

Fonction « Call-back » passée aux appels systèmes

Difficulté :

Pas intuitif à mettre en œuvre

Extension de classes ou dérivation de classes

Définition : *une classe est une extension d'une autre classe si elle ajoute à cette classe des méthodes et champs membres additionnels*

La classe « étendue » est aussi appelée classe fille

La classe de base est aussi appelée « classe parent »

Extension de classes et sous-classement

Peut on parler de ***sous-classement*** ?

- La sous-classe récupère toutes les méthodes définies dans la classe
- La sous-classe ajoute des fonctions à la classe (plus spécialisée que le comportement par défaut)
- **Problème** : comment modifier dans la sous-classe le comportement des méthodes définies dans la classe (cf. cours sur le polymorphisme)

Définition : *une sous-classe est une classe qui* ***redéfinit les méthodes*** *d'une classe déjà existante et qui ajoute à cette classe des méthodes et des champs membres additionnels*

Comment définir une extension d'une classe ?

- Une classe « dérivée »
  - Récupère l'ensemble des méthodes et champs d'une classe « parent » (dont elle dérive)
  - Définit un ensemble de méthodes et de champs additionnels
  - Et c'est tout (enfin presque !)

La syntaxe en C++

- **En C++**

```
class MyBiCounter:
    public MyCounter
{
    …
    void decrement() {
        if(counter == 0)
          counter = max;
        else
          counter --;
    }
    …
}
```

TD – Partie 1

- C++
  - Créer une classe Compteur Dérivé `BiDiCounter` qui introduit une méthode additionnelle `decrement`.
  - Tester le nouveau comportement

Les difficultés : l'initialisation d'un objet étendue

- Comment construire une extension d'un objet
  - **Approche 0.0 :** réutiliser les constructeurs de l'objet « parent »
    - Impossible par ce qu'il faut initialiser les éventuels champs ajoutés
  - **Approche 0.1** : initialiser les extensions à l'objet de base puis initialiser l'objet de base ?
    - Incompatible avec la notion d'extension
  - **Approche 0.2** : initialiser l'objet de base puis initialiser les extensions à l'objet de base ?
    - Approche cohérente avec la notion d'extension
    - Pas toujours satisfaisante, l'objet de base est parfois initialisé avec le contenu d'un champ de l'extension

L’initialisation d’un objet étendu (suite)

- **Approche 1.0** : initialiser les champs de l’extension et l’objet de base dans un ordre défini par le programmeur (***solution en C++***)
  - Permet d’implanter l’approche 0.2
  - Ne permet pas facilement d’implanter des actions avant l’initialisation des champs
  - Permet de garantir que tous les champs et objets sont initialisés avant d’utiliser une méthode

L'initialisation d'un objet étendu (fin)

- **En C++**

```
class MyBiCounter:
    public MyCounter
{
  private:
    bool isEnabled;
  public:
    MyBiCounter(): isEnabled(true),
        MyCounter(isEnabled) {}

    explicit MyBiCounter(
        unsigned  theMax): isEnabled(true),
        MyCounter(theMax) {}

    …
}
```

TD – Partie 2

- C++
  - Créer une hiérarchie de classe
  - Ajouter l’ensemble des constructeurs à la classe de base et aux classes dérivées.
  - Tester ces classes et leurs constructeurs.

Les difficultés : l'accessibilités aux membres de la classe de base

- *Accéder aux membres de la classe de base à partir dans la classe dérivée*
  - Membre « privé » : non
  - Membre « public » : oui
  - Membre « protégé » : oui
- *Accéder aux membres d'une classe de base de la classe de laquelle la classe dérive*
  - Classe de base « privé » : non
  - Classe de base « public » : oui si le membre est public ou protégé
  - Classe de base « protégé » : oui si le membre est public ou protégé
- *Accéder aux membres de la classe de base à l'extérieur de la classe dérivée*
  - Classe de base « privée » : non
  - Classe de base « protégée » : non
  - Classe de base « publique » : oui si le membre est public
- Intérêt de ces règles compliquées ?
  - Permettre de contrôler ce qui est visible de l'extérieur de la classe

Les difficultés : rédéfinir les méthodes

- ***Rappel (Problème 2***) : On souhaite un compteur qui fait quelque chose quand max est atteint ?

Solution : réécrire une méthode «increment()»

- **<u>En C++</u>**

```
class MyBiCounter:
    public MyCounter
{
  …
  public:
     void increment() {
    … }
}
```

Le masquage simple

La méthode *increment*() de *MyCounter* et *increment*() de *MyBiCounter* ont le même nom

- *increment*() de *MyBiCounter* et *increment*() de *MyCounter* coexiste dans MyBiCounter
- Les méthodes définies dans *MyCounter* appelant *increment*() appeleront la méthode *increment*() de *MyCounter*
- Les méthodes définies dans *MyBiCounter* appelant *increment*() appeleront la nouvelle méthode *increment*() de *MyBiCounter*
- Seulement *increment*() de *MyByCounter* sera visible de l'extérieur

Le masquage par surcharge

- ***Problème 3*** : On souhaite ajouter une méthode *increment*(**unsigned** *theCount*) qui augmente de *theCount* le compteur.

- **En C++**

```
class MyAdvCounter:
    public MyCounter
{
  …
  public:
  void increment(
       unsigned theCount)
   { … }
}
```

Le masquage par surcharge

La méthode *increment*() de *MyCounter* et *increment*(unsigned) de *MyAdvCounter* ont le même nom

- *increment*(unsigned) de *MyAdvCounter* et *increment*() de *MyCounter* coexiste dans *MyAdvCounter*
- Les méthodes définies dans *MyAdvCounter* appelant *increment*() appeleront la méthode *increment*() de *MyCounter*
- Les méthodes définies dans *MyAdvCounter* appelant *increment*(**unsigned**) appeleront la nouvelle méthode *increment*(**unsigned**) de *MyAdvCounter* .
- Seulement *increment*(**unsigned**) sera visible de l'extérieur

Accéder aux membres dans une classe

- Accéder aux membres définis dans la classe étendue
  - `this->increment()` ou `increment()`
- Accéder aux membres définis dans la classe de base **B** même si rédéfinie dans la classe.
  - `B::increment()`
- Rendre une méthode ou un champ masqué défini dans la classe de base visible

```
class myAdvCounter: public MyCounter {
  …
  public:
    MyCounter::increment;
    void increment(unsigned aCount) {…}
  …
};
```

TD – Partie 3

- C++
  - Création d’une nouvelle classe dérivée
  - Surcharge d’une méthode déjà existante.
  - Masquage de méthode
  - Rendre une méthode masquée visible.

Le transtypage statique (sûr)

- Un objet **Obj** de type **A** dérive d'une classe de base de type **B** :
  - Est une extension d'une classe de base type **B**
  - Implante tous les membres d'une classe de type **B**
  - Est donc un objet de type **B**

- **L'objet de type A peut-être utilisé là où un objet de type B est requis**
  - L'objet de type A peut-être « transtypé » dans un objet de type B

- En C++ , le transtypage statique peut être implicite ou explicite et s'écrit dans ce cas :

  (B) Obj

  static_cast<B>(Obj)

Le transtypage dynamique (dangereux)

- Un objet **Obj** de type **A** dérive d'une classe de base de type **B** :
  - Un objet de type **B** est éventuellement un objet dérivé
  - Cet objet est éventuellement un objet de type **A**
  - Il est parfois nécessaire de transformer un objet de type **B** en objet de type **A**
  - **La transformation n'est pas sure et peut échouer**

- Le transtypage dynamique est explicite
  - En C++ : dynamic_cast<A>(**Obj**)

TD – Partie 4

- C++
  - Transtypage sûr d’un objet
  - Impact sur le comportement & sélection de méthode

Les difficultés: la destruction d’un objet étendu

- En C++
  - Exécution du code de destruction
  - Destruction ensuite des champs et des classes « parents »

TD – Partie 5

- C++
  - Créer des destructeurs
  - Vérifier que les destructeurs sont bien appelés dans le bon ordre.

Conclusion

Et ce n’est que le début !

….

ou comment la mise en œuvre d’une idée simple peut devenir complexe