Programmation Orientée Objet C++ — Episode #5

Principe de substitution de Liskov

Rectangle-Carré

- `Rectangle` avec `h` et `l`, accesseurs et mutateurs.
- **Post-condition** : hauteur et largeur sont **librement** modifiables.
- `Carré` hérite de `Rectangle`
  mathématiquement cohérent : tout carré est un rectangle.

Le constat
Une instance de type `Carré` devrait être utilisable partout où un `Rectangle` est attendu.

Rectangle-Carré

Le problème

- Dans un `Carré`, hauteur et largeur ne peuvent pas être **librement** modifiables.
- Si un `Carré` est utilisé là où un `Rectangle` est attendu : comportements **incohérents**.

Une mauvaise solution

Modifier les mutateurs de `Carré` : ne respecte pas la post-condition des mutateurs de `Rectangle` !

Principe de substitution de Liskov

L'énoncé © Wikipedia

Si S est un sous-type de T, alors tout objet de type T peut être remplacé par un objet de type S sans altérer les propriétés désirables du programme concerné.

Lié à la **programmation par contrats**.

Principe de substitution de Liskov

Quelques règles (simplifiées © Wikipedia) :

- Les pré-conditions ne peuvent pas être renforcées dans une classe Fille ;
- Les post-conditions ne peuvent pas être affaiblies dans une classe Fille ;
- Pas d'exceptions d'un type nouveau dans une classe Fille, sauf si c'est un sous-type d'une exception lancée par la classe de Base.

Exemple de violation du principe

Le problème **Rectangle-Carré**

Principe de substitution de Liskov

Quelques règles (simplifiées © Wikipedia) :

- Les pré-conditions ne peuvent pas être renforcées dans une classe Fille ;
- Les post-conditions ne peuvent pas être affaiblies dans une classe Fille ;
- Pas d'exceptions d'un type nouveau dans une classe Fille, sauf si c'est un sous-type d'une exception lancée par la classe de Base.

Remarque
Une fonction ne **doit** pas connaître la hiérarchie de classes, et référencer uniquement la classe de Base (sans connaissance des classes Filles).

Le polymorphisme

Essayer de comprendre — #1

```
1  #include <iostream>
2  using namespace std;
3  class Animal {
4    public:
5      Animal() { cout << "ANIMAL" << endl; }
6      ~Animal() { cout << "~ANIMAL" << endl; }
7      void cri() { cout << "CRI-ANIMAL" << endl; }
8  };
9  class Chien : public Animal {
10   public:
11     Chien() : Animal() { cout << "CHIEN" << endl; }
12     ~Chien() { cout << "~CHIEN" << endl; }
13     void cri() { cout << "CRI-CHIEN" << endl; }
14 };
15 class Chat : public Animal {
16   public:
17     Chat() : Animal() { cout << "CHAT" << endl; }
18     ~Chat() { cout << "~CHAT" << endl; }
19     void cri() { cout << "CRI-CHAT" << endl; }
20 };
```

Essayer de comprendre — #2

```cpp
#include <iostream>
using namespace std;
class Animal {
  public:
    Animal() { cout << "ANIMAL" << endl; }
    ~Animal() { cout << "~ANIMAL" << endl; }
    void cri() { cout << "CRI-ANIMAL" << endl; }
};
class Chien : public Animal {
  public:
    Chien() : Animal() { cout << "CHIEN" << endl; }
    ~Chien() { cout << "~CHIEN" << endl; }
    void cri() { cout << "CRI-CHIEN" << endl; }
};
class Chat : public Animal {
  public:
    Chat() : Animal() { cout << "CHAT" << endl; }
    ~Chat() { cout << "~CHAT" << endl; }
    void cri() { cout << "CRI-CHAT" << endl; }
};
```

Essayer de comprendre

```
1  #include <iostream>
2  using namespace std;
3  class Animal {
4    public:
5      Animal() { cout << "ANIMAL" << endl; }
6      ~Animal() { cout << "~ANIMAL" << endl; }
7      virtual void cri() { cout << "CRI-ANIMAL" << endl; }
8  };
9  class Chien : public Animal {
10   public:
11     Chien() : Animal() { cout << "CHIEN" << endl; }
12     ~Chien() { cout << "~CHIEN" << endl; }
13     void cri() { cout << "CRI-CHIEN" << endl; }
14 };
15 class Chat : public Animal {
16   public:
17     Chat() : Animal() { cout << "CHAT" << endl; }
18     ~Chat() { cout << "~CHAT" << endl; }
19     void cri() { cout << "CRI-CHAT" << endl; }
20 };
```

Le mot-clé `virtual`

Sans — `early binding`
La méthode appelée sur un objet (ou la fonction appelée avec ses arguments) est choisie (*nom*) au moment de la compilation.

On ne peut donc pas avoir d'amibiguïté.

Avec — `late binding`
La méthode appelée sur un objet (ou la fonction appelée avec ses arguments) est vérifiée (*nom*) au moment de l'exécution.

Le mot-clé `virtual`

- Certains langages ne donnent pas le choix (**`late binding`**).
- **Perte de performance !**

Objets et C++

- Extraction de sous-objets
- Possibilité d'envoyer sur d'autres types
  même sans cohérence avec une relation d'héritage
- Méthodes/fonctions choisies pour le type apparent

En JAVA

- Notion d'interface
- Transtypage **avec** respect de la hiérarchie
  exception si impossible
- Late binding uniquement.

Downcasting — retrouver l'`Etudiant`

```cpp
#include "etudiant.hpp"

int main() {
  using namespace std;
  Etudiant e("Anthony", 12);

  // Conversion implicite
  Personne &p = e;
  //Etudiant &ep = p;
  // error: initialisation invalide pour une référence du type
      « Etudiant& »
  // à partir d'une expression de type « Personne »

    Personne p1("Anthony");
  Etudiant &ep = static_cast< Etudiant & >(p1);
  cout << ep << endl;

  return 0;
}
```

Downcasting — retrouver l'Etudiant

```
1  #include "etudiant.hpp"
2
3  int main() {
4    using namespace std;
5    Etudiant e("Anthony", 12);
6
7    // Conversion implicite
8    Personne &p = e;
9    Etudiant &ep = static_cast< Etudiant & >(p);
10   cout << ep << endl;
11
12   Personne *pp = &e;
13   Etudiant *ppp = dynamic_cast< Etudiant * > (pp);
14   // error: ne peut effectuer un dynamic_cast « pp » (du type
         « class Personne* »)
15   // vers le type « class Etudiant* » (le type source n'est pas
         polymorphique --- TBC)
16
17   return 0;
18 }
```

Downcasting — retrouver l'`Etudiant`

```cpp
#include "etudiant.hpp"

int main() {
  using namespace std;
  Etudiant e("Anthony", 12);

  // Conversion implicite
  Personne &p = e;
  Etudiant &ep = static_cast< Etudiant & >(p);
  cout << ep << endl;
    // Etudiant &ep = dynamic_cast< Etudiant & >(p);
    // error: cannot dynamic_cast 'p' (of type 'class Personne')
     to type
    // 'class Etudiant&' (source type is not polymorphic)

  return 0;
}
```

Destructeur et héritage

```
1  #include <iostream>
2  class A {
3    protected:
4      int n;
5    public:
6      A( int _n ) : n(_n)
7        { std::cout << "Constructeur A " << n << std::endl; }
8      A( A const &a ) : n(a.n + 10)
9        { std::cout << "Constructeur A " << n <<
10       " copie A " << a.n << std::endl; }
11     ~A() { std::cout << "Destructeur A " << n << std::endl; }
12 };
13 class B : public A {
14   public:
15     B (int _n) : A(_n)
16       { std::cout << "Constructeur B " << n << std::endl; }
17     ~B() { std::cout << "Destructeur B " << n << std::endl; }
18 };
```

Destructeur et héritage

```
1  #include <iostream>
2  class A {
3    protected:
4      int n;
5    public:
6      A( int _n ) : n(_n)
7        { std::cout << "Constructeur A " << n << std::endl; }
8      A( A const &a ) : n(a.n + 10)
9        { std::cout << "Constructeur A " << n <<
10       " copie A " << a.n << std::endl; }
11     ~A() { std::cout << "Destructeur A " << n << std::endl; }
12 };
13 class B : public A {
14   public:
15     B (int _n) : A(_n)
16       { std::cout << "Constructeur B " << n << std::endl; }
17     ~B() { std::cout << "Destructeur B " << n << std::endl; }
18 };
```

Destructeur et héritage

```
1  #include <iostream>
2  class A {
3    protected:
4      int n;
5    public:
6      A( int _n ) : n(_n)
7        { std::cout << "Constructeur A " << n << std::endl; }
8      A( A const &a ) : n(a.n + 10)
9        { std::cout << "Constructeur A " << n <<
10         " copie A " << a.n << std::endl; }
11     virtual ~A() { std::cout << "Destructeur A " << n << std::
       endl; }
12 };
13 class B : public A {
14   public:
15     B (int _n) : A(_n)
16       { std::cout << "Constructeur B " << n << std::endl; }
17     ~B() { std::cout << "Destructeur B " << n << std::endl; }
18 };
```

Destructeur et héritage

Dès qu'il pourrait y avoir héritage : destructeur `virtual`

Les classes abstraites

Méthode virtuelle pure

Le contexte

Méthodes générales devant être **déclarées** mais ne **pouvant pas** être instanciées.

`Syntaxe` : `virtual` <type> <nom>(<paramètres>) `= 0;`

Conséquences

- Pas d'instance de classe possible (aucun sens)
- Classe **abstraite**

Méthode virtuelle pure

```
1  #include <iostream>
2  using namespace std;
3  class Animal {
4    public:
5      Animal() { cout << "ANIMAL" << endl; }
6      ~Animal() { cout << "~ANIMAL" << endl; }
7      virtual void cri() = 0;
8  };
9  class Chien : public Animal {
10   public:
11     Chien() : Animal() { cout << "CHIEN" << endl; }
12     ~Chien() { cout << "~CHIEN" << endl; }
13     void cri() { cout << "CRI-CHIEN" << endl; }
14 };
15 class Chat : public Animal {
16   public:
17     Chat() : Animal() { cout << "CHAT" << endl; }
18     ~Chat() { cout << "~CHAT" << endl; }
19     void cri() { cout << "CRI-CHAT" << endl; }
20 };
```

Méthode virtuelle pure

```
1  #include <iostream>
2  using namespace std;
3  class Animal {
4    public:
5      Animal() { cout << "ANIMAL" << endl; }
6      ~Animal() { cout << "~ANIMAL" << endl; }
7      virtual void cri() = 0;
8  };
9  class Chien : public Animal {
10   public:
11     Chien() : Animal() { cout << "CHIEN" << endl; }
12     ~Chien() { cout << "~CHIEN" << endl; }
13     void cri() { cout << "CRI-CHIEN" << endl; }
14 };
15 class Chat : public Animal {
16   public:
17     Chat() : Animal() { cout << "CHAT" << endl; }
18     ~Chat() { cout << "~CHAT" << endl; }
19     void cri() { cout << "CRI-CHAT" << endl; }
20 };
```

Méthode virtuelle pure

```
1  #include <iostream>
2  using namespace std;
3  class Animal {
4    public:
5      Animal() { cout << "ANIMAL" << endl; }
6      ~Animal() { cout << "~ANIMAL" << endl; }
7      virtual void cri() = 0;
8  };
9  class Chien : public Animal {
10   public:
11     Chien() : Animal() { cout << "CHIEN" << endl; }
12     ~Chien() { cout << "~CHIEN" << endl; }
13     void cri() { cout << "CRI-CHIEN" << endl; }
14 };
15 class Chat : public Animal {
16   public:
17     Chat() : Animal() { cout << "CHAT" << endl; }
18     ~Chat() { cout << "~CHAT" << endl; }
19 };
```

Méthode virtuelle pure

```
1  #include <iostream>
2  using namespace std;
3  class Figure {
4    public:
5      Figure() { cout << "Constructeur Figure." << endl; }
6      ~Figure() { cout << "Destructeur Figure." << endl; }
7      virtual void dessin() = 0;
8  };
9  class Rectangle : public Figure {
10   public:
11     Rectangle() { cout << "Constructeur Rectangle" << endl; }
12     ~Rectangle() { cout << "Destructeur Rectangle" << endl; }
13     void dessin() { cout << "Dessin Rectangle !" << endl; }
14 };
```

Méthode virtuelle pure

```
1  #include <iostream>
2  using namespace std;
3  class Figure {
4    public:
5      Figure() { cout << "Constructeur Figure." << endl; }
6      ~Figure() { cout << "Destructeur Figure." << endl; }
7      virtual void dessin() = 0;
8  };
9  class Polygone : public Figure {
10   public:
11     Polygone() { cout << "Constructeur Polygone" << endl; }
12     ~Polygone() { cout << "Destructeur Polygone" << endl; }
13         bool est_ferme();
14     void dessin(); // Que faire ? --- Polygone reste une classe
           abstraite
15 };
16 class Rectangle : public Polygone {
17   // Instanciation de dessin
18 };
```

Héritage (multiple) et `virtual`

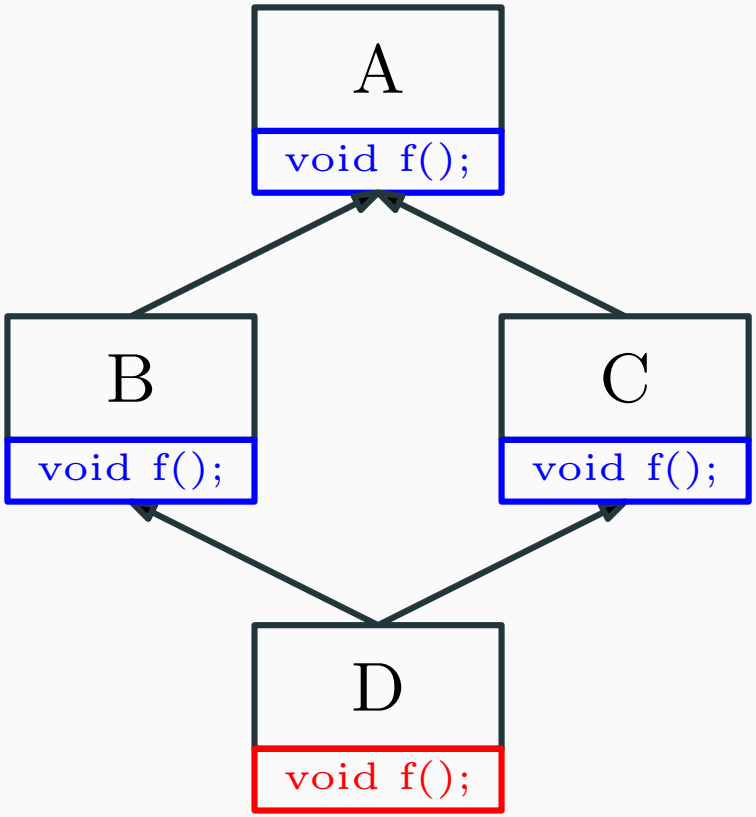
Héritage multiple en C++

```
1  #include <iostream>
2  using namespace std;
3  class A {
4    public : A() { cout << "A" << endl; }
5          virtual ~A() { cout << "~A" << endl; }
6  };
7  class B : public A {
8    public : B() { cout << "B" << endl; }
9          ~B() { cout << "~B" << endl; }
10 };
11 class C : public A {
12   public : C() { cout << "C" << endl; }
13         ~C() { cout << "~C" << endl; }
14 };
15 class D : public B, public C {
16   public: D() { cout << "D" << endl; }
17        ~D() { cout << "~D" << endl; }
18 };
```

Héritage multiple en C++

```
1  #include <iostream>
2  using namespace std;
3  class A {
4    public : A() { cout << "A" << endl; }
5          virtual ~A() { cout << "~A" << endl; }
6  };
7  class B : public A {
8    public : B() { cout << "B" << endl; }
9          ~B() { cout << "~B" << endl; }
10 };
11 class C : public A {
12   public : C() { cout << "C" << endl; }
13         ~C() { cout << "~C" << endl; }
14 };
15 class D : public B, public C {
16   public: D() { cout << "D" << endl; }
17        ~D() { cout << "~D" << endl; }
18 };
```

Retour sur l'héritage (en diamant)



Retour sur l'héritage (en diamant)

```
1  #include <iostream>
2  using namespace std;
3  class A {
4    public : A() { cout << "A" << endl; }
5          virtual void f() { cout << "f.A" << endl; }
6  };
7  class B : public A {
8    public : B() { cout << "B" << endl; }
9          void f() { cout << "f.B" << endl; }
10 };
11 class C : public A {
12   public : C() { cout << "C" << endl; }
13         void f() { cout << "f.C" << endl; }
14 };
15 class D : public B, public C {
16   public: D() { cout << "D" << endl; }
17 };
```

Substitution

```
1  #include <iostream>
2  using namespace std;
3  class A {
4    public : A() { cout << "A" << endl; }
5          virtual void f() = 0;
6  };
7  class B : public A {
8    public : B() { cout << "B" << endl; }
9          void f() { cout << "f.B" << endl; }
10 };
11 class C : public A {
12   public : C() { cout << "C" << endl; }
13         void f() { cout << "f.C" << endl; }
14 };
15 class D : public B, public C {
16   public: D() { cout << "D" << endl; }
17       // fonction de substitution
18       void f() { B::f(); C::f(); cout << "f.D" << endl; }
19 };
```

Héritage multiple et `virtual`

```
1  #include <iostream>
2  using namespace std;
3  class A {
4    public : A() { cout << "A" << endl; }
5          virtual ~A() { cout << "~A" << endl; }
6  };
7  class B : virtual public A {
8    public : B() { cout << "B" << endl; }
9          ~B() { cout << "~B" << endl; }
10 };
11 class C : virtual public A {
12   public : C() { cout << "C" << endl; }
13         ~C() { cout << "~C" << endl; }
14 };
15 class D : public B, public C {
16   public: D() { cout << "D" << endl; }
17        ~D() { cout << "~D" << endl; }
18 };
```