1 HLIN603 - Dernier cours - Synthèse et Invitation à la suite

1.1 Idée

C++ : généraliste, typage statique, complet, complexe

Ocaml : fonctionnel et à objet, typage fort, lien avec **COQ** et la preuve de programmes

Smalltalk-Pharo : pur objet, simple, typage dynamique, réflexif, pensé comme un environnement, l'Avenir pour l'IDM et l'Adaptabilité dynamique.

1.2 Discussion du jour

a) Restreindre **C++** pour faire de la programmation par objets simple

b) **Ocalm**, discussion sur la frontière preuve/réutilisation ?

c) **Smalltalk**, introduction aux design patterns, à l'IDM ... la limite des "`static''`.

2 Restreindre C++ pour faire de la programmation par objet simple

Clé de la discussion : réutilisation et liaison Dynamique (**LD**) (versus liaison statique **(LS)**

- Utiliser des méthodes virtuelles. Toute classe possède au moins une méthode virtuelle.
- Ne pas utiliser les valeurs immédiates.
- Manipuler les adresses via les pointeurs quand c'est possible possible et via les références quand nécessaire.
- Héritage public (sinon pas de sous-typage)
- Héritage virtuel (sinon problème de *dynamic_cast*).
- Maîtriser les redéfinitions (utilisez static_cast ou dynamic_cast)

2.1 Utiliser des méthodes virtuelles

2.1.1 Sans "virtual", liaison statique

```
class A{
 public:
  void m1(){ this->m2(); }
  void m2(){ cout << "m2-A" << endl; }
};

class B : public A{
 public:
  void m2(){ cout << "m2-B" << endl; }
};
```

Listing (1) –

```
int main(){
  A* a = new B();
  a->m1();
  A& aé = *a;
  aé.m1();
  B* b = new B();
  b->m1();
}
```

Listing (2) –

→
m2-A //LS
m2-A
m2-A

2.1.2 Avec “virtual”, liaison dynamique

```
class A{
 public:
  void m1(){ this->m2(); }
  virtual void m2(){ cout << "m2-A" << endl; }
};

class B : public A{
 public:
  void m2(){ cout << "m2-B" << endl; }
};
```

Listing (3) –

```
int main(){
  A* a = new B();
  a->m1();
  A& aé = *a;
  aé.m1();
  B* b = new B();
  b->m1();
}
```

Listing (4) –

→
m2-B // LD
m2-B
m2-B

2.2 Ne pas utiliser les valeurs immédiates sauf pour les types primitifs

Les valeurs immédiates sont créées dans la pile : durée de vie dépendante du compilateur, copies complexes en présence de sous-typage, échec de la liaison dynamique.

Manipuler les objets via leurs adresses

2.2.1 valeurs immédiates $\Longrightarrow$ copies tronquées, liaison statique

```
class A{
  public:
    int x;
    A(int i) { x = i; }
    virtual void affiche(){
      cout << "x : " << x;}
};

class B : public virtual A{
  public:
   int y;
   B(int i, int j):A(i) { y = j; }
   virtual void affiche(){
     this->A::affiche();
     cout << "y : " << y;}
};
```

Listing (5) –

```
int main(){
  A a1(1);
  B b(2,3);
  a = b;
  a.affiche();
}
```

Listing (6) –

→
x : 2 //échec LD

2.2.2 adresses $\Longrightarrow$ copies complètes, liaison dynamique

```
class A{
  public:
    int x;
    A(int i) { x = i; }
    virtual void affiche(){
      cout << "x : " << x;}
};

class B : public virtual A{
  public:
   int y;
   B(int i, int j):A(i) { y = j; }
   virtual void affiche(){
     this->A::affiche();
     cout << "y : " << y;}
};
```

Listing (7) –

```
int main(){
  A* a = new B(2, 3);
  a->affiche();
  A& aé = *a;
  aé.affiche();
}
```

Listing (8) –

→
x : 2 //sucès LD
y : 3
x : 2
y : 3

2.3 Manipuler les adresses via les pointeurs. Limiter l'usage des opérateurs de (dé)référencement

1. Manipuler les adresses via les pointeurs quand c'est possible possible et via les références quand c'est nécessaire (utilisation de bibliothèques).

   ```
   ostream& operator<<(ostream&, const T&);
   ```

2. Les références de *Lisp*, *Scheme*, *Smalltalk*, *Java*, *Ocaml* permettent la manipulation des objets par adresse, avec passage sans copie, avec déréférencement automatique.
   Comparaison avec pointeurs et références c++ ...

2.3.1 Utilisation standard (usage Object-oriented Programming en Scheme/Smalltalk/Ocaml/-Java)

```
class A{
  public:
    int x; int y;
    A(int i, int j) { x = i; y = j;}
    void setX(int i){ x = i; }
    virtual void affiche(){
      cout << "x : " << x
           << "y : " << y;}
};

//test de partage du paramètre

void parAdresse(A* s){ s->setX(4); }

void parReferenceCpp(A& s){ s.setX(6); }
```

Listing (9) –

```
int main(){
  A* a = new A(1,1);
  parAdresse(a);
  a->affiche();
  parReferenceCpp(*a);
  a->affiche();
```

Listing (10) –

→
x : 4 y : 1 //modidié
x : 6 y : 1 //modifié

2.3.2 Danger des références C++

```
class A{
  public:
    int x; int y;
    A(int i, int j) { x = i; y = j;}
    void setX(int i){ x = i; }
    virtual void affiche(){
      cout << "x : " << x << " y : " << y << endl;}
};

// Tests de partage d'environnement
void parAdresse2(A* s){
  s = new A(8,8); }
void parReferenceCpp2(A& s){
  s = *(new A(10,10)); }
void parAdresse3(A* s){
  *s = *(new A(12,12)); }
```

Listing (11) –

```
A* t2 = new A(1, 1);

parAdresse2(t2);
t2->affiche();

parReferenceCpp2(*t2);
t2->affiche();

parAdresse3(t2);
t2->affiche();
```

Listing (12) –

→
x : 1 y : 1 //isolation
x : 10 y : 10 // modif. env !!!!!
x : 12 y : 12 // modif. env.

2.4 Utiliser l'héritage public

Héritage non public : pas de sous-typage

```
class A{
};

class B : A{
};
```

Listing (13) –

```
int main(){
  A* a = new B();
  //error: cannot cast 'B' to its private base
       class 'A'
```

Listing (14) –

2.5 Utiliser l'héritage virtuel

Si héritage non "virtual" [a] et héritage en losange :
– duplication d'attributs
– problème avec "dynamic-cast"

a. Rappel : le "virtual" des méthodes et celui de l'héritage n'ont rien à voir.

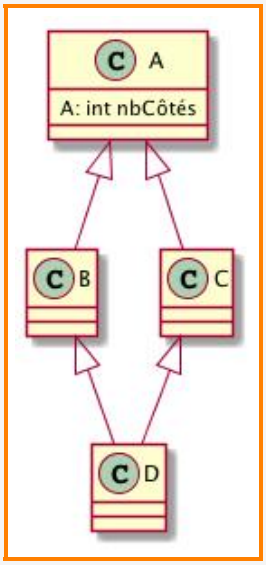

Figure ***(1)*** – *Héritage en losange*

2.6 Maîtrisez les redéfinitions de méthodes

Mêmes règles qu'en *Java* - *Ocaml*, invariance ou contravariance des types des paramètres, invariance ou covariance du type retour.

Contrôle semi-statique (différence Ocaml)

2.6.1 non invariance (types des paramètres) $\Longrightarrow$ non redéfinition

```
class T {};

class U : public T {};

class A{
public:
  void m1(T* t){ this->m2(t); }
  virtual void m2(T* t){ cout << "m2-A"; }
};

class B : public A{
public:
  virtual void m2(U* u){ cout << "m2-B"; }
};
```

Listing (15) –

```
B* b = new B();
b->m1(new U());

A* a = new B();
a->m1(new U());
```

Listing (16) –

→
m2-A //échec LD
m2-A //échec LD

2.6.2 Redéfinition $\Longrightarrow$ invariance (types paramètres)

```
class T {};

class U : public T {};

class A{
public:
  void m1(T* t){ this->m2(t); }

  virtual void m2(T* t){ cout << "m2-A"; }
};

class B : public A{
public:
  virtual void m2(T* t){ cout << "m2-B"; }
};
```

Listing (17) –

```
B* b = new B();
b->m1(new U());

A* a = new B();
a->m1(new U());
```

Listing (18) –

→
m2-B //succès LD
m2-B //succès LD

Mais `t` de `m2-B` non utilisable comme un `U`!

2.7 Simuler une redéfinition covariante $\Longrightarrow$ surcharge + "downcast"

Downcast, règles :

- ne pas utiliser le cast de C, (int) ... dont la syntaxe a été reprise en Java.

```
1 int* anInt = (int*)aFloat; // is equivalent to

2 int* anInt = reinterpret_cast<int*>(aFloat);
```

- utiliser possiblement **static-cast**, mais aucune vérification à l'exécution.

- Utiliser **dynamic-cast** mais le type du paramètre doit être polymorphe (avoir au moins une méthode virtuelle, car `dynamic-cast` utilise la table des méthodes virtuelles).

Utilisation de dynamic-cast

```
class T { virtual void f(){} };

class U : public T { public: int i = 3; };

class A{ public:
  void m1(T* t){ this->m2(t); }

  virtual void m2(T* t){ cout << "m2-A"; }
};

class B : public virtual A{ public:
  void m2(T* t){
    U* u = dynamic_cast<U*>(t);
    if (u != nullptr) this->m2(u);
    else cerr << "t n'est pas de type U";}

  void m2(U* u){
     cout << "m2-B-with-an-U " << u->i; }
};
```

Listing (19) –

```
int main(){
  B* b = new B();
  b->m1(new U());

  A* a = new B();
  a->m1(new U());

  //test de l'exception
  a->m1(new T());
```

Listing (20) –

→
m2-B-with-an-U 3
m2-B-with-an-U 3
t n'est pas de type U

2.8 Toute classe possède au moins une méthode virtuelle

```
class T {
  //virtual void f(){}
};

class U : public T { public: int i = 3; };

class A{ public:
  void m1(T* t){ this->m2(t); }

  virtual void m2(T* t){ cout << "m2-A"; }
};

class B : public virtual A{ public:
  void m2(T* t){
    U* u = dynamic_cast<U*>(t);
    if (u != nullptr) this->m2(u);
    else cerr << "t n'est pas de type U";}

  void m2(U* u){
     cout << "m2-B-with-an-U " << u->i; }
};
```

Listing (21) –

```
1 g++ -std=c++11 ...

2 error: 'T' is not polymorphic
3   U* u = dynamic_cast<U*>(t);
```

3 Réutilisation versus Typage statique : Sous-classe versus Sous-type ?

— Réutilisation : Polymorphisme d'inclusion
`A a = new B()`, avec `B` sous-classe de `A`.

La collection de comptes d'une banque (`Bank`) peut recevoir n'importe quelle instance de Compte (`Account`) ou d'une de ses sous-classe.

— Contrôle de type : correct si la sous-classe définit un sous-type.
Contrôle ... jusqu'où ?

  — **C++**, **Java** : statique + dynamique (*dynamic_cast*, voir section précédente)
  — **Ocaml** : statique, limite des *self-types* ?

3.1 Un exemple avec les comptes bancaires

...

```
1  class account initv =
2  object (self : 'a)
3    val mutable balance = initv
4    method get = balance
5    method deposit a = balance <- balance +. a
6    method withdraw a = balance <- balance -. a
7    method equals (c : 'a) = balance = c get
8    method equalsSuite b = balance = b
```

Listing (22) – Account *: self-type (méthode binaire* `equals`*)*

```
1  class interestAccount initv =
2  object (self)
3    inherit account initv
4    val mutable interestRate= 5.
5    method deposit a =
6      balance <- balance +. a;
7      self#depositInterest a
8    method depositInterest a =
9      balance <- balance +. interestRate *. a /. 100.
```

Listing (23) – InterestAccount*, sous classe et ajoute une méthode :* depositInterest

3.2 *InterestAccount* n'est pas un sous-type de *account*

Les méthodes de `account` et `interestAccount` ont-elles les mêmes types ?

— C'est le cas de `deposit` ...

— Mais, pour `equals`, on a :

  — *equalsaccount* : 'a `->` bool, où 'a = account
  — *equalsinterestAccount* : 'a `->` bool, où 'a = interestAccount

  donc, le type de *equalsinterestAccount* est un sous-type de celui de *equalsaccount* si et seulement si :
  account est un sous-type de *interestAccount*,
  donc si *account* et *interestAccount* sont égaux.
  Mais `interestAccount` possède une méthode en plus (`depositInterest`) ...

3.2.1 Conséquence : pas d'instance de `interestAccount` dans une banque

```
class bank =
object
  val mutable acc_list = []
  method add (a : account) = acc_list <- a::acc_list
  method balance = List.fold_left (fun a b -> a +. b#get) 0. acc_list
  method depositAll = List.iter (fun a -> a#deposit 5.) acc_list
```

Listing (24) –

```
let b = new bank;;
let a = new account 50.;;
let i = new interestAccount 100.;;
b#add a;; (* OK *)
b#add (i:>account);; (* ERREUR, même avec la restriction :> *)
```

Listing (25) –

3.2.2 Solution ? ... partielle

```
class interestAccount initv =
object (self)
  inherit account initv
  val mutable interestRate= 5.
  method deposit a =
    balance <- balance +. a;
    self#depositInterest a
  method private depositInterest a =
    balance <- balance +. interestRate *. a /. 100.
end;;
```

Listing (26) – *private (sémantique ocaml) est applicable à ce cas*

4 Design Pattern Singleton et IDM, la limite des "static"

Singleton : Faire en sorte qu'une classe ne puisse avoir qu'une seule instance.

4.1 Singleton en C++

```
class Singleton{

  protected:
    //stocker l'unique instance
    static Singleton* UniqueInstance = 0;
    //visibilité interne
    Singleton() {}

  public:
    static Singleton* getInstance(){
      if (UniqueInstance == 0)
        UniqueInstance = new Singleton();
      return UniqueInstance;}
};
```

Listing (27) –

```
int main(){
  //new Singleton(); // échec
  Singleton* s =
    Singleton::getInstance();
  Singleton* r =
    Singleton::getInstance();

  std::cout << r==s;}
```

Listing (28) –

→
1

4.2 Singleton Smalltalk - version 1

```
Object subclass: #Singleton
    instanceVariableNames: ''
    classVariableNames: 'soleInstance'
    package: 'ExosPharo-Metaprog'

new
    soleInstance isNil
        ifTrue: [ soleInstance := super new ].
    ^ soleInstance
```

```
Singleton new == Singleton new
```

Listing (29) –

→
true

4.3 Challenge : Partager cette fonctionalité sur une classe abstraite

C++ ... ? ? ?

limite des statics

limite du new non objet

4.4 Singleton Smalltalk - version 2

```
Singleton2 class
    instanceVariableNames: 'soleInstance'

new
    soleInstance isNil
        ifTrue: [
            soleInstance := super new ].
    ^ soleInstance
```

```
Singleton2 subclass: #SA.

Singleton2 subclass: #SB.

SA new == SA new.

SB new == SB new.

SA new == SB new.
```

Listing (30) –

true
true
false →

4.5 Challenge : programmer une transformation d'une Classe en Singleton

Refactoring Browsers ...