KOOC - 5ième Partie

Sommaire

1) Héritage multiple, Interface abstraite, Module mixing.

a) Limitation de l'héritage simple

L'implémentation actuelle de **@class** offre un mécanisme appelé héritage simple. Nous composons nos applications à partir de briques logicielles dont les propriétés et fonctionnalités peuvent être étendues au besoin.

Cependant, avons-nous traité l'ensemble des exigences relevées dans les précédentes parties ?

Dans la 3ième partie, nous avions identifié 2 grands types de librairies :

- Les librairies qui permettent de définir de nouvelles structures de données.
- Les librairies qui sont un patchwork d'algorithmes s'appliquant à un domaine particulier plus qu'à un type particulier.

Or nous n'avions traité qu'un seul type de librairie. En effet, l'héritage simple nous permet de créer simplement le premier type de librairie. Mais qu'en est-il du second ?

b) Un nouveau besoin

De la même manière que nous identifions les objets parents d'un arbre d'héritage, on peut identifier entre plusieurs objets de nature différente (sans ancêtre commun) un ensemble de comportements similaires, qui pourrait faire l'objet d'une bibliothèque d'algorithmes commune (la différence de type mise à part). Nous procédons ainsi à la fois verticalement (héritage) et horizontalement (nos bibliothèques d'algorithmes).

Par exemple : si on prend le cas de la vache, du poney et de la voiture. La vache et le poney sont deux mammifères. Qu'ont-ils de commun avec la voiture ? Ce sont tous trois des moyens de locomotions. Grâce à l'héritage nous pouvons dire que vache et poney héritent de mammifère. Mais comment gérer le cas de la locomotion ?

c) Notre point de vue

Bref, comment traiter efficacement les collections d'algorithmes avec les mêmes soucis de ré-utilisabilité et de généralité que précédemment. Pour répondre à cette question un certain nombre de techniques ont été mises au point pour offrir un éventail plus large aux langages objets.

De manière générale, la conception logicielle a ceci de semblable avec la philosophie : elle pose les questions sans pour autant fournir de réponse objective (non susceptible d'être débattue). Il n'y a pas de dogme, juste de bonnes pratiques.

Donc nous débattrons ici des avantages et inconvénients de ces techniques uniquement sous le prisme technique, laissant les querelles de puriste aux gens qui aiment perdre leur temps.

2) Héritage multiple

La première de ces techniques est l'héritage multiple. En effet, rien n'empêche de réunir un ensemble de comportements au sein d'un même objet. Le seul problème est de mélanger au sein d'un objet fils, à la fois les membres d'une famille objet axée plus type de donnée et les membres d'une famille objet axée plus algorithme.

a) Généralisation de l'agrégation

Concrètement un objet va donc avoir deux ancêtres. Ce principe étant généralisé, un objet a donc maintenant N ancêtres. Comment techniquement gérer ces N ancêtres ? Pour cela nous devons généraliser le principe d'agrégation (mettre du scotch comme le montre la vidéo).

Soit :

```
@class C
{
	@member {
		field c1;
		field c2;
		@virtual method m1();
		@virtual method m2();
	}
}
@class D
{
	@member {
		field d1;
		@virtual method m3();
		@virtual method m4();
	}
}
@class E : C, D // E derive a la fois de C et de D!!
{
	@member {
		field e1;
		@virtual method m2();
		@virtual method m4();
		@virtual method m5();
	}
}
```

Comment représenter E en mémoire ? Quelle est la vtable de E ?

b) Vtable

Pour la vtable, c'est simple on concatène tout avec le même souci d'unicité de signature qu'en héritage simple. On a une branche principale E dérivée de C, puis une branche secondaire E dérivée de D qui s'ajoute ensuite.

```
// debut de decl de C dans E (vtable_branche_C_E)
C~~~m1
E~~~m2 // E redefinit m2
// fin de decl de C dans E
// debut de decl de E branche principale
E~~~m5
// fin de decl de E branche principale
//-------- fin de decl branche principale de E, relation C-E
// debut de decl de D dans E (vtable_branche_D_E)
D~~~m3
E~~~m4 // E redefinit m4
// fin de decl de D dans E
//-------- fin de decl branche secondaire de E
```

c) Représentation mémoire

Pour les données, on concatène de la même manière. cela donne:

```
typedef struct {
      vtable_branche_C_E      *vtable;
      field       c1;
      field       c2;
      field       e1;
      vtable_branche_D_E      *vtable;
      field       d1;
}  E;
```

Sur cet exemple, l'héritage multiple est simple. C et D n'ont pas d'ancêtre commun. Dans le cas d'un ancêtre commun, les choses se compliquent. C'est le fameux problème du diamant. Si une signature collisionne, de quel ancêtre E va-t-elle récupérer l'implémentation ?

3) Héritage multiple et problème du diamant

a) Vtable

Soit :

```
@class A
{
      @member {
            field a1;
            field a2;
            @virtual method m1();
            @virtual method m3();
```

```
	}
}
@class C : A
{
	@member {
		field c1;
		field c2;
		@virtual method m1();
		@virtual method m2();
	}
}
@class D : A
{
	@member {
		field d1;
		@virtual method m3();
		@virtual method m4();
	}
}
@class E : C, D // E derive a la fois de C et de D!!
{
	@member {
		field e1;
		@virtual method m2();
		@virtual method m4();
		@virtual method m5();
	}
}
```

Avec ce genre d'héritage, la vtable est fortement impactée. Nous sommes obligés de garder les branches d'héritages redondantes au niveau du croisement (A redondant de C et A redondant de D).

```
//------ vtable de E -----
//----- branche C
C~~~m1 // C redefinit m1 de A
A~~~m3
//------- fin de decl de A dans C
E~~~m2 // E redefinit m2 de C
//-------- fin de decl de C
//----- branche D
A~~~m1
D~~~m3 // D redefinit m3 de A
//------- fin de decl de A dans D
E~~~m4 // E redefinit m4 de D
//-------- fin de decl de D
E~~~m5
//-------- fin de decl de E
```

On a donc une partie de vtable de l'objet C concaténée à une vtable d'un objet D. Cela va nous poser des problèmes au moment des conversions. En effet, suivant le cast de E (vers C ou D), le début de la vtable n'est plus le même. Par exemple le A

de C a la méthode m1 de C alors que le A de D a la méthode m3 de D. Nous devons donc garder une trace de ces décalages.

b) Représentation mémoire

D'un point de vue purement «structure interne de l'objet» quel est l'aspect en mémoire des différentes variables membres de E ? De plus, comme nous avons plusieurs débuts de vtable à gérer nous devons aussi stocker dans la structure de l'objet ces différentes informations.

```
pointer_sur_vtable_branche_C // debut de l'objet C et E
a1 // membre de A branche C
a2 // membre de A branche C
c1 // membre de C
c2 // membre de C
e1 // membre de E
pointer_sur_vtable_branche_D // debut de l'objet D
a1 // membre de A branche D
a2 // membre de A branche D
d1 // membre de D
```

Comparé à l'héritage simple, on voit bien un objet de type E hérité simplement de C concaténé avec un objet de type D. Où est la vtable de E ? Le premier pointeur conformément à l'héritage simple est confondu avec la vtable de E car, pointant au même endroit. Dorénavant le problème majeur est lors des casts. Un objet E casté en D doit pointer sur le début de l'objet D. Il y a donc manipulation du pointeur. De même, pour caster un objet concret E, précédemment casté en D, que l'on veut upcaster en E (son type réel), nous devons garder trace du décalage à effectuer. Pour cela nous devons garder cette information quelque part. Il est possible de la garder dans la vtable.

c) Thunk function

De plus malgré cette information, un morceau de code spécifique doit être appelé afin de réajuster le pointeur lors d'un appel à une fonction virtuelle.

```
E* e = [E new];

D* d = (D*) e; // code a generer pour decaler le pointeur e vers la sous
partie D.

[d m4]; // appele polymorphique a m4, PB!
/*
En effet dans :
@implementation E
{
      method   m4(E* self)
      {}
```

Ici self est de type E*..., or dans [d m4], d est de type D*. On ne peut donc pas simplement générer le code suivant :

```
d->vt[idx_m4](d, ...)
```
Il faut caster d en E* en entrée de la fonction m4 et ensuite appeler m4 de E.

Il faut donc générer la fonction thunk_m4, fonction proxy vers m4 qui correspond au pseudo-code suivant :

```
method thunk_m4(D* d, params)
{
E* e = d - X; // conversion de D* en E*
return m4(e, params);
}
```

En fait cette fonction est constituée de 2 instructions assembleur (Un sub et un jmp) afin d'être transparente lors de l'appel. Il faut par contre modifier la vtable en correspondance...

```
//------ vtable de E -----
//----- branche C
C~~~m1 // C redefinit m1 de A
A~~~m3
//------- fin de decl de A dans C
thunk_E~~~m2 // thunk de E redefinissant m2 de C, et castant un C* en E*
//-------- fin de decl de C
//----- branche D
A~~~m1
D~~~m3 // D redefinit m3 de A
//------- fin de decl de A dans D
thunk_E~~~m4 // thunk de E redefinissant m4 de D, et castant un D* en E*
//-------- fin de decl de D
E~~~m5
//-------- fin de decl de E
}
*/
```

La complexité inhérente à l'héritage multiple a fait que d'autres techniques plus simples ont vu le jour.

4) Interface abstraite + module mixing

L'héritage multiple étant complexe à implémenter, il serait intéressant tout en conservant la simplicité de l'héritage simple d'offrir les mêmes fonctionnalités qu'un héritage multiple.

a) Keep It Simple

La fonctionnalité principale est d'étendre la puissance de modélisation en offrant la possibilité de lier plus fortement les objets en terme de comportements qu'en termes de données.

Pour cela, un autre modèle, celui des interfaces est utilisable. Une interface s'utilise en KOOC grâce au mot-clef **@interface** qui ne délimite que des

déclarations de fonctions et aucun code. Il liste juste un ensemble de fonctions à implémenter quand on respecte l'interface (appelée aussi le contrat, ou le protocole). C'est le choix de java, d'objective C, ruby, de D, ...

Soit :

```
@interface Networkable
{
int   Connect(char* serv, int port);
int   Send(char* buf, size_t size);
int   Recv(char* buf, size_t size);
int   Disconnect();
}

@interface Serializable
{
int   Serialize(char* dump, size_t size);
int   Deserialize(char* dump, size_t size);
}


@class MyObject : Networkable, Serializable
{
      @member field myData;
}

@implementation MyObject
{
// on doit implementer toutes les methodes de l'interface
      @member {
      int   Connect(char* serv, int port) {}
      int   Send(char* buf, size_t size) {}
      int   Recv(char* buf, size_t size) {}
      int   Disconnect() {}
      int   Serialize(char* dump, size_t size) {}
      int   Deserialize(char* dump, size_t size) {}
      }
}
```

Les interfaces n'ont pas de variable membre, non membre ou des fonctions non membres. Seules les fonctions membres sont permises.

b) Vtable

Une interface va imposer à **MyObject** d'implémenter les fonctions décrites dans l'interface. On autorise un objet de dériver de plusieurs interfaces. On autorise aussi aux interfaces d'étendre une ou plusieurs autres interfaces. Contrairement à l'héritage multiple, on ne garde dans la vtable qu'un seul type d'interface.

Soit dans notre exemple de diamant.

```
@interace A
{
      method m1();
      method m3();
}
@interface C : A
{
      method m2();
}
@interface D : A
{
      method m4();
}
@class E : C, D // E derive a la fois de C et de D!! normallement A 2
//fois...pas de probleme
{
      @member {
            field e1;
            field e2;
            field e3;
            field e4;
            field e5;
            field e6;
            @virtual method m5();
      }
}
@implementation E
{
      @virtual method m1() {}
      @virtual method m2() {}
      @virtual method m3() {}
      @virtual method m4() {}
      @virtual method m5() {}
}

//------ debut vtable de E -----
//--- class E
E~~~m1
E~~~m2
E~~~m3
E~~~m4
E~~~m5
//--- interface A
E~~~m1
E~~~m3
//--- interface C
E~~~m1
E~~~m2
E~~~m3
//--- interface D
E~~~m1
E~~~m3
```

```
E~~~m4
//------ fin vtable de E -----
```

Nous avons simplement la vtable de l'objet E, suivie d'un exemplaire de vtable de chaque interface. Nous gardons la bonne propriété d'avoir pour une signature donnée au sein d'une interface un index unique.

c) Délégué d'interface

Que se passe-t-il maintenant lors des casts ?

```
int main()
{
	E*	objetReel;
	A*	interfaceAbstraite;

	objetReel = [E new];
	interfaceAbstraite = (A*) objetReel;
}
```

En fait A, C, D bien qu'interfaces sont pourtant bien des types de données. Pour chaque **@interface** vous devez créer un **typedef** sur une **struct** particulière. En effet, à travers eux on ne peut accéder uniquement qu'aux méthodes définies dans l'interface qu'ils représentent. Il s'agit d'une interface déléguée ou classe proxy ou classe d'accès, contenant uniquement un pointeur de vtable et le pointeur de l'instance concernée.

```
//----- contenu de la variable interfaceAbstraite
vtable_sur_interface_A_de_objetReel
valeur_du_pointeur_objetReel
//
```

Ainsi **vtable_sur_interface_A_de_objetReel** permet de toujours trouver la méthode à appeler sur l'interface A. Et **valeur_du_pointeur_objetReel** est la valeur de Self.

Toutefois, il y a un problème. Étant un objet composite (non scalaire), il est difficile de manipuler une variable de type interface abstraite. En effet, dans l'exemple précédent, on caste l'objet en pointeur sur interface abstraite. Or le cast ne permet pas de savoir où on stocke le contenu réel de la variable **interfaceAbstraite**. Nous devons donc interdire le cast direct et introduire une nouvelle syntaxe.

```
int main()
{
	E*	objetReel;
	A	interfaceAbstraiteParente;
	C	interfaceAbstraiteEnfante;

	objetReel = [E new];
	[objetReel UpCastTo :&interfaceAbstraiteParente];
	[&interfaceAbstraiteParente maFonction :mesParamatres];
```

```
      [interfaceAbstraiteParente DownCastTo :&interfaceAbstraiteEnfante];
      [&interfaceAbstraiteEnfante maFonction2 :mesParamatres2];

      A*    delegueSurLeTas = [objetReel createInterfaceDeleguate];

      [delegueSurLeTas maFonction :mesParamatres];

      [delegueSurLeTas delete];
}
// attention a delegueSurLeTas, createInterfaceDeleguate le cree sur le tas
// donc --> delete!
```

Grâce à la signature des fonctions **UpCastTo** et **DownCastTo**, nous pouvons remplacer la fonctionnalité du cast avec des variables sur la pile. Si on veut un délégué créé sur le tas, on utilise **createInterfaceDeleguate**.

Comme le choix de localisation en mémoire des variables d'interfaces est laissé à la charge du développeur, nous gardons une certaine souplesse d'utilisation.

d) Module Mixing

Cependant, comme une interface ne contient aucun code, l'utilisation à long terme de cette mécanique est très fastidieuse. Comment faire pour récupérer une implémentation de base d'un certain nombre de méthodes dans de telles conditions ? Pour cela, nous introduisons le concept de module mixing.

Le module mixing est une technique qui permet d'implémenter un certain nombre de méthodes d'une interface au sein d'une classe par l'inclusion d'un module (contenant le code de ces méthodes).

Par exemple:

```
@interface Networkable
{
int   Connect(char* serv, int port);
int   Send(char* buf, size_t size);
int   Recv(char* buf, size_t size);
int   Disconnect();
}

@interface Serializable
{
int   Serialize(char* dump, size_t size);
int   Deserialize(char* dump, size_t size);
}


@module RawSerialize
{
```

```
      @with Serializable // permet de preciser que les methodes suivantes
doivent exister dans Serializable
      {
            int   Serialize(Serializable* self, char* dump, size_t size);
            int   Deserialize(Serializable* self, char* dump, size_t size);

      // Le self est implicitement Serializable* par definition
      }

}

@implementation RawSerialize
{
// implemente les fonctions decrites
 ....
}

@module RawNetworkize
{
      @with Networkable
      {
            int   Connect(Networkable* self, char* serv, int port);
            int   Send(Networkable* self, char* buf, size_t size);
            int   Recv(Networkable* self, char* buf, size_t size);
            int   Disconnect(Networkable* self);
      }
}

@implementation RawNetworkize
{
// implemente les fonctions decrites
....
}


@class MyObject : Networkable, Serializable
{
@include RawSerialize; // mix le code du module concernant Serializable ici
@include RawNetworkize; // mix le code du module concernant Networkable ici

}

@implementation MyObject
{}
```

Dans cet exemple, l'ajout du nouveau mot-clef **@include** permet d'ajouter facilement l'implémentation offerte par **RawSerialize** et **RawNetworkize** à l'objet **MyObject**. La convention veut qu'une fonction membre corresponde à une fonction de module sauf dans l'expression du premier paramètre (pointeur self).

Rappelez-vous

```
MonInterface*     monInstance = [MonInterface new];

[monInstance MaFonction :monParametre]
```

équivaut à

```
[MonInterface MaFonction :monInstance :monParametre]
```

Ici, nous utilisons ce principe sauf que self est un pointeur d'interface abstraite.

De plus la pratique du module mixing doit rester souple. C’est-à-dire qu'un module qu'on mixe avec une interface n'implémente pas forcément toute l'interface abstraite qu'il complète.

De cette façon, il est facile de programmer des interactions entre la classe réelle et le module. Notamment grâce à des accesseurs.

En effet, il se peut que pour coder la logique propre à un module mixé, le module doive accéder à des données de l'objet. N'ayant qu'un pointeur d'interface en paramètre, il ne connait pas le type réel de l'objet.

Par contre, dans l'interface, on définit un certain nombre d'accesseur pour que le module mixé puisse uniquement à travers l'interface abstraite obtenir les informations dont il a besoin.

```
@interface Serializable
{
// methode publique
int   Serialize(char* dump, size_t size);
int   Deserialize(char* dump, size_t size);

// methode pour les modules mixe

void  *GetInternPointer();
size_t      GetInternSize();

}

@module RawSerialize
{
      @with Serializable
      {
            int   Serialize(Serializable* self, char* dump, size_t size);
            int   Deserialize(Serializable* self, char* dump, size_t size);
      }
}

@implementation RawSerialize
{
```

```
int   Serialize(Serializable* self, char* dump, size_t size)
{
      if (size < [self GetInternSize])
            return -1;
      memcpy(dump, [self GetInternPointer], [self GetInternSize]);
      return [self GetInternSize];
}

int   Deserialize(Serializable* self, char* dump, size_t size)
{
      if (size < [self GetInternSize])
            return -1;
      memcpy([self GetInternPointer], dump, [self GetInternSize]);
      return [self GetInternSize];
}

}

@class MyObject : Serializable
{

      @member fields data;


@include RawSerialize;

// methode de l'interface abstraite non implemente dans le module

      void  *GetInternPointer(MyObject* self); // attention equivalent a
@member
      @member size_t    GetInternSize();
}

@implementation MyObject
{
// bon on n'utilise pas la syntaxe @member mais le format complet. Ca permet
// d'acceder a self directement! :)
      void  *GetInternPointer(MyObject* self)
      {
            return self;
      }

      @member     size_t      GetInternSize()
      {
            return sizeof(MyObject);
      }
}
```

Voilà le concept du module mixing emprunté à Ruby, transposé dans un langage compilé comme KOOC.