Policies

1. Objectif
2. Principe
3. Exemples

Objectif

Nous somme dans le contexte de l'écriture de librairies.
Comment fournir une librairie qui soit à la fois flexible, hautement réutilisable, qui conserve les qualités usuelles d'efficacité algorithmique et qui permette toujours de choisir simplement dans les compromis espace temps ceux qui sont les plus adaptés au logiciel que nous développons.
Une policy (politique) permet de définir une interface qui est spécifique à une problématique donnée, tout choix de politique peut être implémenté tant qu'il se conforte à l'interface définie par la politique.

Pourquoi introduite une nouvelle technique ? Alors que nous disposons déjà des classes abstraites pures pour faire des interface, des template, de l'héritage multiple virtuel ou direct ?
Les politiques offre une méthodologie qui permet facilement de retarder certains choix de conception (design) jusqu'au moment ou nous avons enfin les arguments pour les faire (en général c'est souvent quand de vrais test de fonctionnement sont réalisés que nous pouvons mesurer la valeur de certains choix d'implémentation) .

Le développement logiciel est une discipline qui offre une des plus riche multiplicité de solution. Le nombre de solutions correctes a un problème donné est en général très grand. Il y a une infinité de nuances en bien fait et mal fait. Chaque choix ouvre un nouveau monde. Une fois que l'on a choisi une solution, des quantités de variations apparaisse à tous les niveaux, de l'architecture aux détails les plus fins du code.
La conception d'un logiciel consiste à choisir une solution dans un espace combinatoire de solutions.

Au vu d'un problème logiciel quelle est la bonne solution ? Évènements ? Objets ? Observeur ? CallBack ? Polymorphisme ? Délégation ? Templates ? Jusqu'à un certain niveau de détail et à certains niveau d'analyse de nombreuses solutions semble fonctionner aussi bien.
Ce qui différencie l'expert architecte et le débutant c'est que le premier sait quelles solutions fonctionnes et les quelles ne fonctionne pas. L'expert sait quelles sont les avantages et les inconvénients des différentes solutions et donc celles qui sont adaptés ou non au présent problème .

Objectif (2)

Choisir : comme dans la vie c'est le moment difficile !

Les concepteurs aguerris connaissent les choix qui vont mener à une bonne conception.

Le concepteur expérimenté est comme un bon joueur d'échec il peut voir plusieurs coups à l'avance.

Cela prend du temps à apprendre, c'est pourquoi l'expertise de programmation peut s'acquérir rapidement alors qu'il faut du temps pour maîtriser la conception logiciel.

Cette multiplicité des solutions est un défi pour les concepteurs de librairies, pour implémenter une bonne librairie il faut anticiper un large éventail de situations typiques et laisser la librairie ouverte, pour permettre au développeur d'applications de l'adapter aux situations particulières.

Comment en effet peut on organiser une librairie de bon composants logiciels. Comment se défendre contre la multiplicité des approche en écrivant une quantité de code raisonnable ?

Ces questions sont les thèmes de ce chapitre.

L'interface à tout faire

L'interface à tout faire (Débutant Compagnie trademark) n'est pas une bonne solution.

1° un coût intellectuel non négligeable pour le créateur comme l'utilisateur, (petites interface cohérentes)
2° des classes Mammouths, lourdes inefficaces .
3° Absence de contrôle de type statique, idéalement le design doit permettre de définir des contraintes qui sont validées par le compilateur. Hors les interface à tout faire ne peuvent assurer ce type de contrainte (parfois globalement contradictoires).

Un dangereux écart se creuse entre le syntaxiquement valide et le sémantiquement valide .

Regardons par exemple la construction d'un objet Singleton qui doit être robuste du point de vue d'exécution concurrente dans un programme multi-threads (activités). Si l'aspect multi-activité est entièrement encapsulé dans la librairie alors un utilisateur sur un système mono-activité ne peut utiliser la librairie. A l'inverse si la librairie autorise l'accès aux primitives non protégées (celles qui sont protégé par la couche concurrente) alors il y a un risque que le programmeur viole le concept concurrent en écrivant du code syntaxiquement correct mais pas sémantiquement valide.

Par contre si la librairie implémente différents choix de conception grâce à un ensemble de petites classes, chaque classe représenteras un choix de design. Prenons pour exemple le pointeur intelligent `SmartPtr` nous pouvons facilement imaginer une batterie de classes : `SingleThreadSmartPtr, MultiThreadSmartPrt, RefCountSmartPtr, RefLinkSmartPtr,` etc.

Le problème de cette solution est notre problème initiale de multiplicité de solution. Nous voudrions pouvoir choisir indépendamment plusieurs aspects de notre conception et il nous faut donc la classe `SingleThreadRefCountNotNullSmartPtr` et bien d'autre du même genre.

L'écriture de toutes ces classes n'est pas une solution : Ne jamais utiliser la force contre une exponentielle !

Héritage Multiple ?

L'idée est de construire des petites classes élémentaires que nous allons combinées pour construire nos classes grâce à l'héritage multiple.

On veut une `MultiThreadRefCountSmartPtr` il suffit d'avoir une classe `BasicSmartPtr` et deux classes `MultiThreaded` et `RefCounted` dont nous héritons pour fabriquer notre classe.

Cette solution n'est pas viable comme tout les programmeurs expérimentés le savent :

1) le mécanisme du C++ d'héritage multiple est très peu puissant il ne fait que « coller » les classes ensemble ce qui est notoirement insuffisant.
2) Les classes de base ne connaisse pas les types sur lesquelles elle doivent agir. Il est ainsi impossible de créer une classe de base qui réalise `DeepCopy` (copie en profondeur) comme cette classe ne connaît pas la forme du type sur le quel elle doit travailler et les types quelle doit créer pour réaliser la copie.
3) De nombreux aspect comportementaux doivent manipuler et partager l'état ce qui implique un héritage virtuel. Ceci rend utilisant encore plus délicate et de plus retourne la direction d'héritage il faut que les classes de la libraire hérite virtuellement des classes utilisateur (de pire en pire).

Utilisé brutalement l'héritage multiple ne fournis pas une solution à la multiplicité des choix de conception.

Spécialisation

```
template <class T> class Widget {
    void fun() {   ... implémentation générique ... }
};
```
// Spécialisation pour d'une fonction membre

```
template <> Widget<char>::fun() { ... implémentation spécialisé ... }

template <class T, class U> class Gadget
{
    void fun() {   ... implémentation générique ... }
};
```
// Impossible de faire une spécialisation partielle d'une fonction membre

```
// ERROR
template <class U> Gadget<char,U>::fun() { ... implémentation
   spécialisé ... }
```

Ouf

Les deux techniques précédente on miraculeusement des défauts complémentaire.
C'est de la que Andrei Alexandrescu tire les Politiques.

Il faut voire les politiques comme la fusion du concept d'interface avec les techniques combinés des templates et de l'héritage multiple.
Une politique défine une interface de classe ou une interface de class template. L'interface consistant en un ou plusieurs éléments suivants : définition de type internes, fonctions membres et données membres.
Les politiques sont très proches des traits mais sont plus préoccupé par le comportement que par le typage.
Les politiques sont basées sur le Design pattern Stratégie (Strategy) avec l'avantage d'être résolu à la compilation !

Ce sont des interfaces définies au niveau syntaxique et pas au niveau signature comme les interfaces java.

Les templates Alors ?

Les modèles (templates) sont une bonne approche pour résoudre des problèmes de combinaisons de comportements grâce à leur faculté de produire à la compilation du code basé sur les types utilisateur.

Les templates de class sont plus adaptables que les classes standard. En effet vous pouvez spécialisez n'importe quelle fonction membre d'une instanciation donnée d'un template de classe.
Ex: Soit `SmartPtr<T>` votre template vous pouvez spécialisez n'importe quelle méthode de `SmartPtr<Widget>.`
Dans le cas d'un template à paramètres multiple vous pouvez réaliser des spécialisation partielles qui vous permette de préciser le comportement que pour une partie des paramètres :

```
template <class T, class U> class SmartPtr { ... } ;
```
on peut spécialiser avec la syntaxe suivante
```
template <class U> class SmartPtr<Widget,U> {... } ;
```

Jusque la tout vas bien mais ....
quelques problèmes apparaissent

1) Impossible de spécialisé la structure (genre si le type T a telle propriété j'ai besoin d'un entier en plus ou en moins). Vous ne pouvez spécialiser que les fonctions.
2) La spécialisation ne fonction pas sur les template à nombre de paramètres multiples (cf. transparent suivant)

3) Impossible de fournir plus d'une implémentation par défaut, ce qui est fortement limitatif dans le cadre d'une librairie ...

Un exemple de politique

```
class Widget {} ; // on en a besoin pour les exemples d'instanciation
class Gadget {} ;

// politique : fournir une fonction template Create d'allocation

template <class T>
struct OpNewCreator
{
static T* Create ()
{ return new T ; }
};

template <class T>
struct OpMallocCreator
{
static T* Create ()
{ void *buf = std::malloc(sizeof(T));
    if (! buf) return 0;
    return new(buf) T ; }
};
// une version avec une méthode template
struct MyAllocCreator {
template <class t>
static T* Create(){return new T; }
};
```

Utilisation d'une des instance de la politique

```
// patern Factory sur des Widgets
template <class CreationPolicy>
class WidgetManager : public CreationPolicy
{

};
// Instentiation
typedef WidgetManager<OpNewCreator < Widget> > GestionnaireDeWidget;

// mieux utilisons un template template parameter
template <template <class Created> Class CreationPolicy = OpNewCreator>
class ttWidgetManager : public CreationPolicy<Widget> {
void doit(){ Gadget *g = CreationPolicy<Gadget>().Create();
} ;

 // instentiation
ttWidgetManager<OpMallocCreator> uu;
```

Classes Hôtes

Les classes qui utilise une ou plusieurs politiques sont des classes Hôtesses / structures Hôtes.
Les classes qui implémente des politiques sont des classes politiques.

Un premier aspect important des politiques: elles sont orientées syntaxe et pas signature comme les interfaces classiques.
Ceci évite de nombreux problème de réécriture de code et de blocage syntaxique.