
NO to O

Objectif

- Offrir une interface C++ à une librairie non-objet du type C

Motivation

Les fonctions présentes dans une librairie C sont souvent proches d'une approche objet sans en offrir l'interface. Ce pattern de codage se propose de faire le lien entre la librairie C et les classes C++. L'objectif est de ne pas impacter les performances de la librairie tout en offrant une interface plus sympathique. Ce pattern offre une conversion d'une interface non-objet vers une interface objet (No Object to Object).

Applicabilité

Ce pattern est applicable pour toutes les librairies traditionnelles renvoyant des pointeurs sur des structures opaques. Ces pointeurs sont souvent appelés « handle ». Attention, ce n'est pas applicable avec de véritables « handles » qui ne sont pas des pointeurs. Par exemple, les handles de fenêtre de Windows™ 16 bits ne sont pas des pointeurs. Dans ce cas, ce pattern n'est pas utilisable.

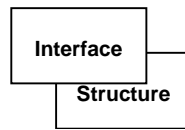
La structure `FILE` du C Ansi est très rarement manipulée en tant que structure. Les fonctions standard utilisent un *pointeur* de ce type. La fonction `fopen()` s'occupe de construire une instance de `FILE`. Pour offrir une interface C++ à la structure `FILE`, il est possible d'utiliser le pattern Adapter (139). Celui-ci oblige à construire une instance associée à chaque structure `FILE`. Si cette instance ne possède pas d'attributs supplémentaires, il est préférable de s'en passer. En effet, la durée de vie de celle-ci est difficile à gérer.

Imaginons qu'une fonction d'une librairie quelconque retourne une structure `FILE`. Il faut alors construire une instance `CFile` ayant une relation avec cette structure. Si par la suite la librairie décide de détruire la structure `FILE` précédemment retournée, l'instance `CFile` associée deviendrait invalide. Quand la détruire ?

La librairie MFCs de Microsoft™ utilise un artifice d'instances « furtives ». Lorsqu'un handle doit être converti en objet C++, une instance est créée mais maintenue sur une liste spéciale. Avant la réception d'un nouveau message Windows™, toutes les instances furtives de cette liste sont détruites. Une nouvelle instance est reconstruite à chaque nouvelle demande. Ces objets d'interfaces ne sont présents qu'entre deux messages.

Il serait beaucoup plus confortable de considérer la structure `FILE` directement comme un objet C++. C'est ce que ce pattern permet. L'objet `CFile` est un alias de la structure correspondante `FILE`. Il ne représente que l'interface avec celle-ci. Il ne possède pas de relation avec une structure `FILE`, il est la structure `FILE`.

Structure



Participants

- **Structure**
Structure opaque manipulée par la librairie non-objet.
- **Interface**
Classe offrant l'interface avec la librairie classique.

Collaborations

- La classe d'interface est une conversion de la structure C correspondante.

Conséquences

1. Le coût mémoire supplémentaire est nul
2. La vitesse d'exécution supplémentaire est nul
3. La structure peut évoluer sans modification de l'interface.
4. Il n'y a pas de création d'instances d'interfaces.

Implementation

Les difficultés à résoudre pour obtenir ce pattern concerne la construction et la destruction de l'objet `Interface`. Celui-ci ne peut être construit qu'à l'aide de la librairie utilisée. Comme la taille de `Structure` n'est pas connue, il n'est pas possible de construire une instance locale. Les constructeurs et destructeurs ne doivent pas pouvoir être appelés. Par ce fait, les opérateurs `new` et `delete` ne seront également pas utilisables. Il faut offrir des méthodes « fabricante » (Factory Method (107)) permettant la construction et la destruction de l'objet. Pour les services attendus de celui-ci, une conversion de `this` va permettre de déléguer les appels vers la librairie. Tous les services de l'interface doivent être en `inline`. Ainsi, le coût de cette interface sera nul.

Exemple de Code

```
class CFile
{ private:
    CFile();          // Non implanté
    ~CFile();        // Non implanté
public:
    // Méthode fabricante
    static CFile* New(const char* filename,const char* mode)
    { return (CFile*)fopen(filename,mode);
    }
    static void Delete(CFile* p)
    { ::fclose((FILE*)this);
    }
    size_t read(void* buffer,size_t size,size_t count)
    { return ::fread(buffer,size,count,(FILE*)this);
    }
    size_t write(const void* buffer,size_t size,size_t count)
```

```
    { return ::fwrite(buffer,size,count,(FILE*)this);  
    }  
    // ...  
};  
  
void main()  
{ CFile* f=CFile::New("file.txt","r");  
  char buf[10];  
  f->read(buf,sizeof(*buf),sizeof(buf));  
  CFile::Delete(f);  
}
```

Utilisations connues

Patterns en relation

- Le pattern Factory Method (107) est utilisé pour construire les instances des objets.
- Le pattern Adapter (139) permet d'offrir un service équivalent et de rajouter des services et/ou des attributs à l'objet [Structure](#). Par contre, il oblige à gérer la durée de vie des objets d'[Interfaces](#) ce qui n'est pas toujours évident.