Design Pattern : Chain Of Responsability

1. Introduction

Imaginons que l'on cherche à résoudre un problème pour lequel nous disposons de plusieurs experts de ce type de problème. Supposons encore que chacun de ces spécialistes aie une compétence particulière. C'est-à-dire qu'il existe un cas particulier du problème pour lequel il est le seul à pouvoir déterminer la solution exacte. Le problème est évidemment confié au groupe d'experts. Comment ces derniers doivent-ils s'organiser pour trouver la solution ?
L'idée la plus simple est de les placer en file indienne. Chacun d'entre eux examine alors à tour de rôle le problème. Dès que l'un d'entre eux a reconnu la situation pour laquelle il est compétent, il résout le problème et c'est terminé, sinon il confie le problème à l'expert suivant dans la file. Si la chaîne d'experts ainsi formée est entièrement parcourue sans succès, on essaie de fabriquer une solution par défaut.
Cette chaîne d'experts qui résout collectivement le problème est appelée chaîne de responsabilité (COR pour Chain Of Responsability) car chaque maillon prend tour à tour la responsabilité de résoudre le problème.
Le DP COR est la directe application de cette idée au développement de logiciels.

2. Formalisation

Notons *d* la donnée du problème à résoudre. On peut supposer que *d* est une instance d'une classe *D*.
Notons *s* la solution du problème lorsqu'elle existe. Supposons que *s* soit une instance d'une classe *S*.

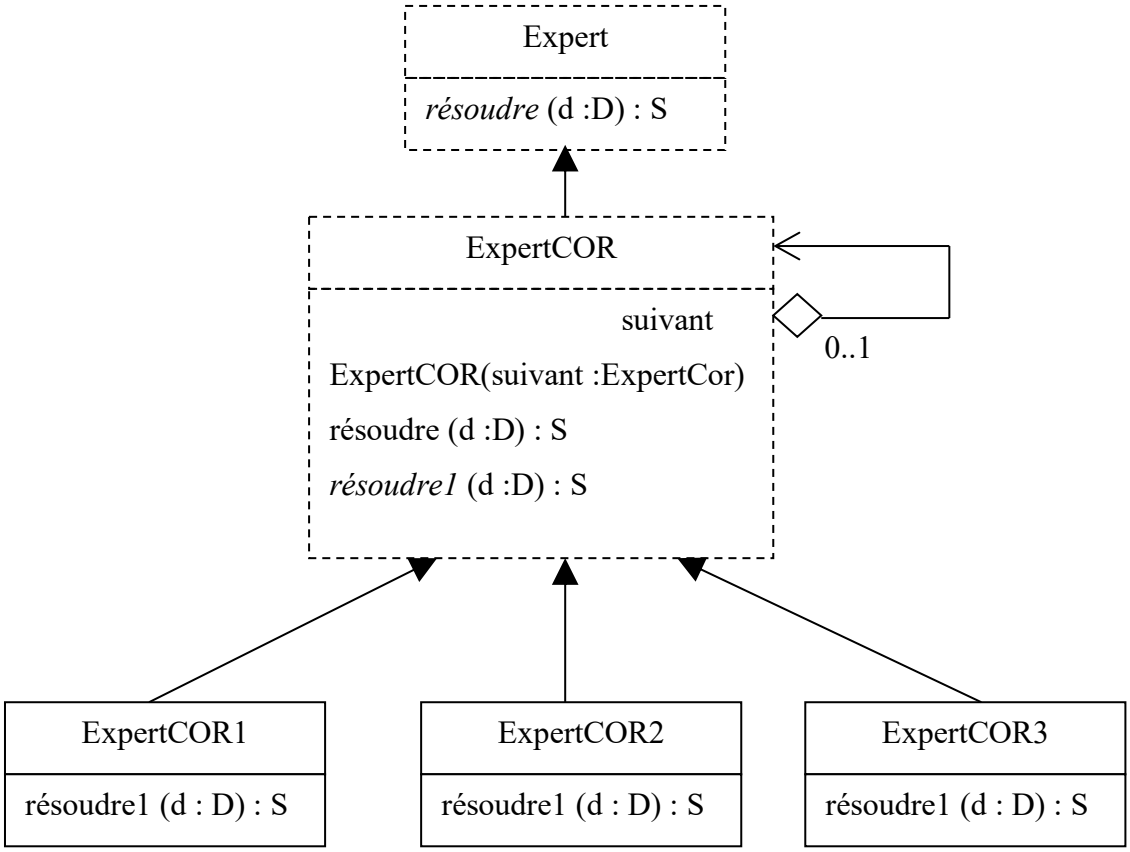
Appelons *Expert* toute classe sachant résoudre le problème. Cette classe est la classe de base de toutes celles qui savent résoudre le problème. Elle est donc très générale et ne met pas en œuvre le DP COR (car sinon on rend impossible toute recherche de solution n'utilisant pas le DP COR).
Notons *résoudre (d : D) : S* la méthode abstraite de Expert qui résout le problème.
En C++, la classe Expert prend la forme suivante :

```
class S;
class D;

class Expert
{
/**
@param d : donnée du problème à résoudre
@return la solution ou NULL en cas déchec
*/
virtual  S * resoudre (const D & d) const = 0;
};
```

Notons *ExpertCOR* toute classe sachant résoudre le problème et destinée à être placée dans une chaîne d'experts. Elle est munie de la méthode *résoudre* () qui permet de parcourir la chaîne et de la méthode *résoudre1* () qui est spécifique à chaque expert. Elle est également munie d'un
attribut *suivant* qui est une référence sur l'expert suivant dans la chaîne.
Notons *ExpertCOR1*, *ExpertCOR2* et *ExpertCOR3* trois classes expertes particulières et cas particuliers de *ExpertCOR*.

L'organisation de toutes ces classes est résumée par le diagramme suivant :



Une chaîne de responsabilité *expert* est alors construite puis utilisée de la façon suivante, dans la fonction main, par exemple :

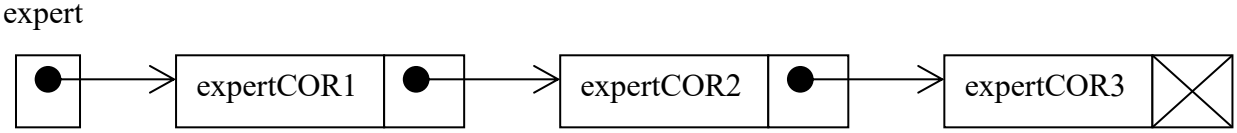
```
int main()
{
ExpertCor * expert ;

expert = new ExpertCOR3(NULL);
expert = new ExpertCOR2(expert);
expert = new ExpertCOR1(expert);

D d(…) ;

S * s = expert->résoudre(d);      // ignore volontairement comment expert est mis en oeuvre
return 0;
}
```

Dans la RAM, *expert* a bien évidemment la forme suivante bien classique des listes chaînées :



Il est aisé de rajouter des compétences à cette chaîne, il suffit d'insérer un nouvel expert.

3. Mise en œuvre de la chaîne

Supposons que la mise en œuvre soit effectuée en C++. Ecrivons la méthode *résoudre(d : D) : S* de la classe *ExpertCOR*. Cette méthode parcourt la chaîne ; chaque maillon tente tour à tour de résoudre le problème. Si il parvient, le parcours s'arrête et la solution est renvoyée et s'il n'y parvient pas il transmet le problème au maillon suivant. Si la chaîne est entièrement parcourue en vain, la méthode renvoie la valeur `NULL`. De même, la méthode *résoudre1(d : D) : S* spécifique à chaque expert, abstraite à ce niveau, renvoie aussi la valeur `NULL` en cas d'échec.

Déclaration de la classe ExpertCOR :

```
class ExpertCOR : public Expert
{
ExpertCOR * suivant;  // expert suivant dans la chaîne

protected:
ExpertCOR(ExpertCOR * expertSuivant);

public:
S * resoudre (const D & d) const;

protected:
virtual S * resoudre1(const D & d) const = 0;
};
```

Mise en oeuvre de la classe ExpertCOR :

```
ExpertCOR::ExpertCOR(ExpertCOR * expertSuivant):suivant(expertSuivant) { }


S * ExpertCOR::resoudre (const D & d) const
{
S * s = this->resoudre1(d);        // cet expert tente de résoudre le
                                                          //  problème

if  (s != NULL)                   // cet expert a trouvé une solution
       return s;

else                               // échec de cet expert

       if  (this->suivant != NULL)             // le problème est transmis à
                                                          // l'expert suivant
              return this->suivant->resoudre(d);

       else    // cet expert est le dernier de la chaîne
              return NULL; // donc échec de la chaîne
}
```

4. Exemple concret

Montrons à présent à l'aide d'une situation concrète comment appliquer le DP COR.

Fixons nous comme objectif de réaliser un dictionnaire « langue étrangère vers Français ». L'unique tâche de ce dictionnaire consiste à analyser un texte et de traduire celui-ci en Français. La langue dans laquelle est écrit le texte n'est pas fixée, le dictionnaire doit donc d'abord reconnaître la langue puis effectuer la traduction.
Ce dictionnaire Etranger -> Français peut idéalement être représenté à l'aide d'une chaîne de responsabilité d'experts en traduction où chaque maillon de la chaîne reconnaît exactement une seule langue et sait effectuer la traduction de celle-ci vers le français. Dans l'exemple qui suit, la chaîne est composée de trois traducteurs : Anglais -> Français, Allemand -> Français et Espagnol -> Français.
La hiérarchie de classes et la mise en œuvre sont copiées sur celles du paragraphe 3.
On reprend donc le diagramme de classes du paragraphe 2 dans lequel, on effectue les remplacements de noms suivants :

Expert	devient	DicoEtrangerFrancais
ExpertCOR	devient	DicoEtrangerFrancaisCOR
ExpertCOR1	devient	DicoAnglaisFrancaisCOR
ExpertCOR2	devient	DicoAllemandFrancaisCOR
ExpertCOR3	devient	DicoEspagnolFrancaisCOR
S résoudre(D d)	devient	char * traduit(char * texte)
S résoudre1(D d)	devient	char * traduit1(char * texte)

On obtient le code suivant :

4.1 classe abstraite DicoEtrangerFrancais

```
class DicoEtrangerFrancais
{
/**
 *
 * traduit texte en français. texte peut être un mot, un groupe nominal, une phrase, etc.
écrite dans une langue étrangère
 *
 * en cas d'échec, retourne NULL
 *
 * */
public:

virtual const char * traduit(const char *  texte) const = 0;
};
```

4.2 Classe abstraite DicoEtrangerFrancaisCOR

Déclaration de la classe DicoEtrangerFrancaisCOR :

```
class DicoEtrangerFrancaisCOR : public DicoEtrangerFrancais
{
public:
DicoEtrangerFrancaisCOR * suivant;

protected:
DicoEtrangerFrancaisCOR(DicoEtrangerFrancaisCOR * suivant);

public :

/**
 * traduction utilisant la chaîne de responsabilité. algo récursif
```

```
 *
 * */
const char * traduit(const char * texte) const;

virtual const string toString() const = 0;

/**
 * savoir-faire de l'un des traducteurs de la chaîne
 * */
protected:
virtual const char * traduit1(const char * texte) const = 0;

}; // DictionnaireEtrangerVersFrancaisCOR
```

Mise en oeuvre de la classe DicoEtrangerFrancaisCOR :

```
DicoEtrangerFrancaisCOR::DicoEtrangerFrancaisCOR(DicoEtrangerFrancaisCOR * suivant) :
suivant(suivant){}

/**
 * traduction utilisant la chaîne de responsabilité. algo récursif
 *
 * */
const char * DicoEtrangerFrancaisCOR::traduit(const char * texte) const
{
const char * resultat;
resultat = this->traduit1(texte); // ce traducteur tente de traduire le texte

if (resultat != NULL)  // une traduction a été trouvée
    return resultat;

else    // échec de ce traducteur

    if (this->suivant != NULL)    // puisque il y a un suivant, on lui confie la tâche de
traduction
        return this->suivant->traduit(texte);
    else // c'était le dernier traducteur, c'est donc un échec
        return NULL;
}
```

4.3 Classe DicoAnglaisFrancaisCOR

Déclaration de la classe DicoAnglaisFrancaisCOR :

```
#define TAILLE_ANGLAIS 5

class DicoAnglaisFrancaisCOR : public DicoEtrangerFrancaisCOR
{
static char * t[TAILLE_ANGLAIS][2];

public :

DicoAnglaisFrancaisCOR(DicoEtrangerFrancaisCOR * suivant);

const string toString() const;

protected :
```

```
const char * traduit1(const char * texte) const;
};
```

Mise en oeuvre de la classe DicoAnglaisFrancaisCOR :

```
/*static*/ char * DicoAnglaisFrancaisCOR::t[TAILLE_ANGLAIS][2]  = {{"Hello","Salut"},
                                {"Nice to meet you" ,"Enchanté de faire votre connaissance"},
                                {"Mind your own business","Occupez vous de vos affaires"},
                                {"turnip","navet"},{"mouse","souris"}};

DicoAnglaisFrancaisCOR::DicoAnglaisFrancaisCOR(DicoEtrangerFrancaisCOR * suivant) :
DicoEtrangerFrancaisCOR(suivant){}

const char * DicoAnglaisFrancaisCOR::traduit1(const char * texte) const
{
int i;
for ( i = 0; i < TAILLE_ANGLAIS; ++i)
    if (strcmp(texte,t[i][0]) == 0)
        return t[i][1];

return NULL;
}

const string DicoAnglaisFrancaisCOR::toString() const
{
int i;
string r;

for ( r = "", i = 0; i < TAILLE_ANGLAIS; ++i)
       r += (string)t[i][0] + " <---> " + t[i][1] + "\n";
return r;
}
```

4.4 Classe DicoAllemandFrancaisCOR

Déclaration de la classe DicoAllemandFrancaisCOR :

```
#define TAILLE_ALLEMAND 3
class DicoAllemandFrancaisCOR : public DicoEtrangerFrancaisCOR
{
static char * lAllemand [TAILLE_ALLEMAND];
static char * lFrancais [TAILLE_ALLEMAND];

public :

DicoAllemandFrancaisCOR(DicoEtrangerFrancaisCOR * suivant);

const string toString() const;

protected :
const char * traduit1(const char* texte) const;
};
```

Mise en oeuvre de la classe DicoAllemandFrancaisCOR :

```
/*static*/ char * DicoAllemandFrancaisCOR::lAllemand [TAILLE_ALLEMAND] =
{"maus","blum","hund"};
/*static*/ char * DicoAllemandFrancaisCOR::lFrancais [TAILLE_ALLEMAND] =
{"souris","fleur","chien"};



DicoAllemandFrancaisCOR::DicoAllemandFrancaisCOR(DicoEtrangerFrancaisCOR * suivant) :
DicoEtrangerFrancaisCOR(suivant) {}


const string DicoAllemandFrancaisCOR::toString() const
{
int i;
string r;
for( r = "", i = 0; i < TAILLE_ALLEMAND; ++i)
    r+="("+(string)lAllemand[i]+", "+lFrancais[i]+")";
return r;
}

const char * DicoAllemandFrancaisCOR::traduit1(const char * texte) const
{
int i;

for ( i = 0; i < TAILLE_ALLEMAND; ++i)
    if (strcmp(texte,lAllemand[i]) == 0)
        return lFrancais[i];
return NULL;
}
```

4.5 Classe DicoEspagnolFrancaisCOR

Déclaration de la classe DicoEspagnolFrancaisCOR :

```
class DicoEspagnolFrancaisCOR : public  DicoEtrangerFrancaisCOR
{
vector<char *>  l1, l2;

public :

DicoEspagnolFrancaisCOR(DicoEtrangerFrancaisCOR * suivant);


const string toString() const;

protected:
const char * traduit1(const char * texte) const;
};
```

Mise en oeuvre de la classe DicoEspagnolFrancaisCOR :

```
DicoEspagnolFrancaisCOR::DicoEspagnolFrancaisCOR(DicoEtrangerFrancaisCOR * suivant) :
DicoEtrangerFrancaisCOR(suivant)
{
l1.push_back("Un ratoncito verde");                  l2.push_back("Une souris verte");
l1.push_back("Una cerveza bien fria");               l2.push_back("Une bière bien fraîche");
l1.push_back("Un bizcocho de chocolate");            l2.push_back("Un gâteau au chocolat");
l1.push_back("El perrito negro");                    l2.push_back("Le petit chien noir");
```

```
}

const char * DicoEspagnolFrancaisCOR::traduit1(const char * texte) const
{
unsigned int i;

for ( i = 0; i < l1.size(); ++i)
    if (strcmp(texte,l1[i]) == 0) return l2[i];
return NULL;
}


const string DicoEspagnolFrancaisCOR::toString() const
{
unsigned int i;

string r;
for ( r = "", i = 0; i < l1.size(); ++i)
       r += (string)l1[i] +" <---> " + l2[i] + "\n";

return r;

}
```

4.6 Création et utilisation du dictionnaire

```
#define LONGUEUR_DONNEE 500

int main()
{
DicoEtrangerFrancais * dicoEtrangerFrancais ;
DicoEtrangerFrancaisCOR * dicoAnglaisFrancais, * dicoAllemandFrancais, * dicoEspagnolFrancais;


//on construit la chaîne de responsabilité.
// Dans une version définitive, ceci devrait être réalisé ailleurs à l'aide du design pattern
FACADE

dicoAnglaisFrancais = new DicoAnglaisFrancaisCOR(NULL);

dicoAllemandFrancais = new DicoAllemandFrancaisCOR(dicoAnglaisFrancais);

dicoEspagnolFrancais = new DicoEspagnolFrancaisCOR(dicoAllemandFrancais);


dicoEtrangerFrancais = dicoEspagnolFrancais;


DicoEtrangerFrancaisCOR * p;

for ( p = (DicoEtrangerFrancaisCOR*)dicoEtrangerFrancais; p ; p = p->suivant)
       cout << p->toString() << endl;


// le dictionnaire sera utilisé par l'intermédiaire de dicoEtrangerFrançais qui pointe sur le
1er maillon de la chaîne

// Dans la suite, le client (c-à-d main()) ignore la nature du dictionnaire, il ne doit
connaître que l'interface

cout <<"Tapez la chaîne à traduire en Français : ";
char donnee[LONGUEUR_DONNEE];
const char *  resultat;
```

```
cin.getline(donnee,LONGUEUR_DONNEE-1,'\n');

resultat = dicoEtrangerFrancais->traduit(donnee);

const char * reponse = resultat ? resultat : "non connue dans notre dictionnaire";

cout << "La traduction de : " <<  donnee <<" est : " <<reponse << endl;

char ch; cin >> ch;

return 0;
}
```

5. Conclusion : Intérêt du DP COR

Considérons la situation suivante.
Une méthode *f* reçoit une donnée *d* brute (de bas niveau sémantique) (un paquet d'octets, une chaîne de caractères, un bloc de pixels, un fichier binaire, etc.) qu'elle doit analyser. *f* doit reconnaître à travers *d* un cas particulier parmi une liste de cas possibles puis effectuer un traitement spécifique à ce cas particulier identifié. Comment écrire le code de la fonction *f* ? L'attitude naïve consiste à écrire une cascade d'instructions *if – else* ou bien une instruction *switch* qui prévoit tous les cas puis dans chacun des cas, à écrire le code du traitement spécifique. La méthode *f* ainsi obtenue est généralement illisible et doit, de plus, être revue, corrigée et testée à chaque nouveau cas particulier qui apparaît. Ces deux inconvénients disparaissent si *f* est écrite en utilisant le DP COR, comme le montre l'exemple avec les dictionnaires. En effet, ajouter une nouvelle langue au dictionnaire est facile à faire et ne demande pas d'écrire d'avantage de code que celui consacré à la traduction de la nouvelle langue. L'utilisation du DP COR accroît donc l'extensibilité du logiciel dans toutes les situations décrites par la fonction *f*.
Le DP COR convient bien dans toutes les situations où on dispose de peu d'informations sur les données ; lorsque celles-ci peuvent beaucoup varier ; lorsqu'elles sont très hétérogènes.
En particulier, Le DP COR est une excellente manière d'organiser le code chaque fois qu'il faut analyser du texte ; les parseurs et traducteurs l'appliquent donc très souvent.

Remarques

- Dans l'exemple des dictionnaires, les classes *DicoAnglaisFrancaisCOR*, *DicoAllemandFrancaisCOR* et *DicoEspagnolFrancaisCOR* sont écrites de façons très différentes alors qu'elles pourraient suivre un modèle commun aidant à la refactorisation du code. Cette absence d'unification est volontaire car il n'est pas sûr que, dans une version professionnelle de l'application, des linguistes utiliseraient la même approche pour analyser de l'Allemand ou de l'Espagnol.
- Avec un petit effort de conceptualisation, il est possible d'écrire une classe chaîne de responsabilité universelle et donc d'écrire une fois pour toutes la méthode *résoudre* (). Cette réflexion est laissée au lecteur.

- La complexité de l'assemblage de la chaîne de responsabilité peut être masquée par l'application du DP Façade