SAMUEL GOULET

Analyse de systèmes

420-3D6-LL, Groupe 00001

TP4

Chaîne de responsabilité

Travail présenté à

M. Olivier Lafleur

Département d’informatique

Cégep de Lévis-Lauzon

1 mai 2017

Table des matières

Acrostiche 3

Description générale 4

Fonctionnement 5

Diagramme de classes 5

Avantages et inconvénients 6

Exemples 7

Chaîne simple 7

Chaîne avec des conditions d’exécution 8

Chaîne avec des conditions d’arrêt 8

Chaîne avec retour 9

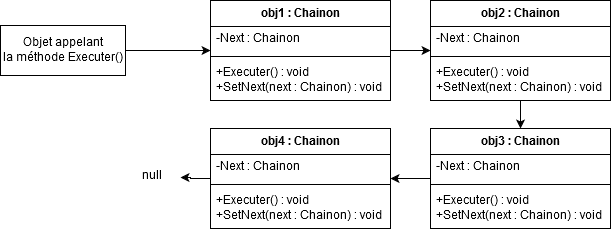
Chaînes multiples 9

Réflexion 10

Chaîne de responsabilité

**C**e patron de conception,  
**H**abituellement utilisé dans les grosses applications,  
**A** pour but de faciliter les appels de méthodes distribuées en masses.  
**I**l permet aux instances de classes implémentant une certaine interface  
**N**onobstant le contexte de l’origine de l’appel de la méthode,  
**E**n fournissant leurs propres conditions d’exécution,  
**D**’exécuter  
**E**t de passer l’appel de la méthode à la prochaine instance,  
**R**éalisant à sa manière une liste chaînée.  
**E**n appelant la méthode homonyme de l’objet suivant  
**S**ous la condition qu’il ne soit pas null, la méthode  
**P**asse dans chaque  
**O**bjet de la chaine, afin d’y subir potentiellement un certain traitement.  
**N**ormalement, l’interface force l’implémentation de méthodes telles que  
**S**etNext(…), qui prend un pointeur vers le prochain objet de la chaine,  
**A**insi qu’une méthode qui vérifiera des conditions d’exécution et fera le traitement.  
**B**ien que le traitement de la méthode soit facultatif, selon les conditions spécifiées,  
**I**l est généralement important que cette méthode continue tout de même la chaine,  
**L**e cas contraire arrêterait nécessairement le traitement.  
**I**l se peut que l’on veuille,  
**T**outefois,  
**E**mpêcher le traitement inutile quand on considère qu’il est terminé.

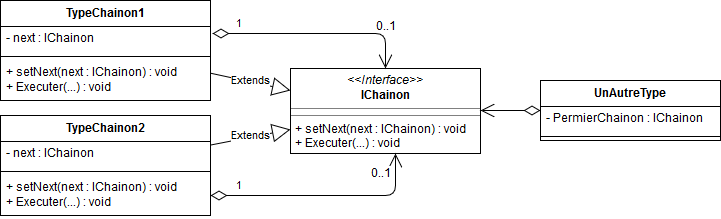
Description générale :

Le patron de conception « Chaîne de responsabilité » est un patron comportemental. Il permet de contrôler facilement la distribution de l’appel d’une méthode dans un groupe d’objets, en faisant en sorte que chaque objet traite la requête, puis l’envoie à son prochain. Ces objets seront liés entre eux à la manière d’une liste chaînée, donc chaque objet a un pointeur vers le prochain objet, et le dernier chaînon pointe vers <null>, où la chaîne se brise.

Dans ce diagramme, un objet quelconque (possiblement le formulaire) appelle la méthode Executer() du premier objet de la chaîne, qui effectue un traitement quelconque puis appelle la méthode Executer() du deuxième objet, qui effectue à son tour le traitement puis appelle la méthode Executer() du troisième objet, et ainsi du suite. Le quatrième objet, par contre, ne peut pas continuer la chaîne, puisque son attribut Next n’est pas défini. La chaîne est donc terminée, les méthodes retournent et le fil d’exécution revient à l’objet appelant.

Ce patron est particulièrement utile lorsque l’on veut appeler une méthode qui sera traitée par plusieurs objets selon des conditions spécifiques à chaque objet. Dans le diagramme précédant, on peut imaginer que les objets obj1 et obj2 ont des attributs différents et que le traitement effectué par chacun sera différent selon ces attributs.

Fonctionnement :



Le fonctionnement d’une chaîne de responsabilités rappelle énormément celui d’une liste chaînée. Le diagramme ci-haut ne représente qu’une idée générale de l’implantation de ce patron, car l’idée en soi peut être utilisée de manière très large. Généralement, un objet faisant partie d’une chaîne de responsabilité devra avoir les caractéristiques suivantes :

* Une méthode publique « SetNext(next : monInterface) » qui prend un pointeur vers le prochain chaînon;
* Une méthode publique « Executer(…) », qui exécute le traitement et appelle la méthode Executer du prochain chaînon;
* Implicitement, un attribut « Next », pour garder en mémoire le prochain chaînon.

La chaîne de responsabilités est construite en appelant la méthode SetNext(…) du dernier élément de la chaîne pour lui donner le prochain chainon (dont le « Next » sera null et qui deviendra conséquemment le nouveau dernier élément). Elle est utilisée en appelant la méthode « Executer(…) » du premier élément de la chaîne. L’appel de cette méthode devrait, si le programmeur sait bien s’en servir, propager cette méthode dans les objets appropriés afin de bien la traiter. Il peut donc être pertinent d’avoir une classe qui gère la chaîne de responsabilité, donc qui gardera en mémoire le premier et le dernier élément (ou encore un tableau contenant tous les éléments).

Avantages et Inconvénients:

Avantages :

* Réduit le couplage entre les objets.  
  (L'objet qui appelle la méthode n'a qu'à connaître le premier élément de la chaîne pour l'utiliser.)
* Facilite la gestion de code optionnel.  
  (Comme chaque objet peut géré lui-même les conditions d'exécution, il est plus facile de gérer les cas d'exception et les conditions particulières.)
* Peut être modifiée dynamiquement.  
  (En enlevant ou en ajoutant des objets dans la chaîne, de la même façon que dans une liste chaînée, on contrôle directement quels objets reçoivent l'appel de la méthode, et dans quel ordre.)
* Pratique lors de la création d'APIs.  
  (Comme on n'a qu'à savoir que l'objet a un certain interface, on peut utiliser des types créés par d'autres utilisateurs sans même en connaître le contenu (en utilisant une fabrique, par exemple, pour avoir un pointeur vers l'objet en question.))
* Standardise la diffusion de méthodes dans des groupes d'objets.  
  (Au lieu d'avoir à appeler une méthode en boucle dans tous les objets, on ne fait qu'un appel de fonction et tous les objets concernés reçoivent la demande.)  
  (Lorsque la chaîne risque de se complexifier et devenir difficile à gérer, il peut être recommandable d'avoir une classe dédiée à la chaîne (qui agit un peu en tant que façade et/ou d'objet composite) qui s'occupera de modifier la chaîne, d'appeler la méthode appropriée, de gérer les retours, etc.)
* Facilite la récolte d'information dans un groupe d'objets, ou les retours multiples.  
  (En passant un objet (une liste, une structure, etc) par référence en paramètre, chaque objet dans le chaîne peut le modifier, créant ainsi une manière facile et rapide de regrouper les retours de plusieurs objets.)

Inconvénients :

* Peut être dangereux si le traitement est lourd et la chaîne est longue.  
  (Comme les méthodes s'appellent entre elles avant leur fin, les ressources utilisées ne sont que libérées lorsque la chaîne se termine. Manquer de mémoire est donc un risque à considérer.)
* Rend le débogage difficile.  
  (Comme la chaîne peut être modifiée dynamiquement, il peut être difficile de prévoir le comportement d'une longue chaîne de responsabilité.)

Exemples :

*Chaîne simple :* Nous recevons des ingrédients par la ligne de commande et nous voulons les afficher. (Dans ce genre de cas simple, il peut être préférable d'utiliser une boucle.)

static void Main(string[] aliment)

{

if (aliment.Length == 0) return;

Ingredient premier, dernier, curr = null;

// Début de la chaîne. On garde le premier élément en mémoire

dernier = premier = new Ingredient(aliment[0]);

int i = 1;

// Construction de la chaîne à partir de chacun des arguments reçus

while (i < aliment.Length) {

curr = new Ingredient(aliment[i]);

dernier.SetNext(curr);// Spécification du prochain chaînon

dernier = curr; // Garder en mémoire le dernier élément

i++;

}

// À ce stade, notre chaîne ressemble à {Ing1 -> Ing2 -> ... -> null}

premier.Executer(); // Appel de la fonction

Console.Read();

}

}

public class Ingredient

{

private string m\_Nom;

private Ingredient m\_Next; // Prochain élément dans le chaîne

public Ingredient(string Nom) {

m\_Nom = Nom;

// Le prochain est null tant qu'il n'est pas spécifié

m\_Next = null;

}

public void SetNext(Ingredient i) {

m\_Next = i; // Modification du prochain élément

}

public void Executer() {

Console.WriteLine(m\_Nom); // Traitement de la méthode

// Appel de la méthode homonyme du prochain élément

// (s'il y a un prochain élément)

if (m\_Next != null)

m\_Next.Executer();

}

}

**Input ->**

CoR-Exemple1.exe patate celeri carotte navet poire citrouille tomate piment fraise

**Output ->** (traiter les \n comme des changements de ligne)

patate\nceleri\ncarotte\nnavet\npoire\ncitrouille\ntomate\npiment\nfraise

*Chaîne avec des conditions d’exécution :* Maintenant, seuls les ingrédients dont le nom commence par ‘p’ s’afficheront.

static void Main(string[] aliment) {

// ...

premier.Executer('p'); // Appel de la fonction

Console.Read();

}

public class Ingredient {

// ...

public void Executer(char c) {

if (m\_Nom != null && m\_Nom.Length >= 1 && Char.ToLower(m\_Nom[0]) == c)

Console.WriteLine(m\_Nom); // Traitement de la méthode avec conditions

if (m\_Next != null)

m\_Next.Executer(c); // Appel du prochain élément

}

}

**Input ->**

CoR-Exemple1.exe patate celeri carotte navet poire citrouille tomate piment fraise

**Output ->** (traiter les \n comme des changements de ligne)

patate\npoire\npiment

*Chaîne avec des conditions d’arrêt :* On ne veut plus que les deux premiers éléments.

static void Main(string[] aliment) {

// ...

int max = 2;

premier.Executer('p', ref max);// Appel de la fonction en spécifiant un max

Console.Read();

}

public class Ingredient {

// ...

public void Executer(char c, ref int max) {

if (m\_Nom != null && m\_Nom.Length >= 1 && Char.ToLower(m\_Nom[0]) == c) {

Console.WriteLine(m\_Nom); // Traitement de la méthode

max--; // Décrémentation du compteur

}

if (m\_Next != null && max > 0) // Si la fin n’est pas atteinte,

m\_Next.Executer(c, ref max); // continuer la chaine

}

}

**Input ->**

CoR-Exemple1.exe patate celeri carotte navet poire citrouille tomate piment fraise

**Output ->** (traiter les \n comme des changements de ligne)

patate\npoire\n

*Chaîne avec retour :* Au lieu d’afficher les éléments dans la console, on veut maintenant que la chaîne retourne le deuxième élément qui commence par ‘p’.

static void Main(string[] aliment) {

// ...

int max = 2;

string s = premier.Executer('p', ref max);

// On assigne à une variable le retour de la chaîne

Console.WriteLine(s);

Console.Read();

}

public class Ingredient {

// ...

public string Executer(char c, ref int max) {

if (m\_Nom != null && m\_Nom.Length >= 1 && Char.ToLower(m\_Nom[0]) == c)

max--; // Nous n’affichons plus les ingrédients

if (m\_Next != null && max > 0)

return m\_Next.Executer(c, ref max);

// On retourne le retour du restant de la chaîne.

// Lorsque l’un des chaînons retournera une valeur, elle sera

// transmise à l’objet appelant.

else

{

if (max == 0)

return m\_Nom;

// Si la condition est atteinte, on retourne notre nom

else

return "";

// Si on a atteint la fin de la chaîne,

// on retourne une chaîne vide

}

}

}

**Input ->**

CoR-Exemple1.exe patate celeri carotte navet poire citrouille tomate piment fraise

**Output ->** (traiter les \n comme des changements de ligne)

poire\n

*Chaînes multiples :* Au lieu d’appeler systématiquement la méthode du prochain chaînon, on a plusieurs branches différentes qui peuvent être appelées, à la manière d’un arbre. Par exemple, un interpréteur de commandes pourrait avoir différentes branches pour une même commande, ou un interpréteur XML pourrait avoir plusieurs nœuds à exécuter pour un même nœud.

Ex :   
<NoeudPrincipal>  
 <NoeudSecondaire1></NoeudSecondaire1>  
 <NoeudTertiaire1></NoeudTertiaire1>  
 <NoeudSecondaire2></NoeudSecondaire2>  
 <NoeudTertiaire1></NoeudTertiaire1>  
</NoeudPrincipal>

Réflexion :

De manière générale, le patron de conception de la chaîne de responsabilité est bénéfique à la qualité, à la modularité, à la lisibilité, à l’ouverture, etc. du code, en autant qu’il soit utilisé convenablement.

Il est basé sur le fait que chaque objet gère semi-indépendamment ses conditions d’exécution et son traitement, le code est par le fait même très modulaire et facilement maintenable. Si un problème advient, il suffit de modifier la classe responsable du problème et le régler. Par le fait même, puisque le code est spécifique aux cas d’utilisation de la classe, le principe « responsabilité unique » de SOLID est encouragé.

La structure de la chaîne ne requiert qu’une interface externe en commun. Donc, il est possible, dans le cadre d’un API par exemple, de créer une chaîne de responsabilité et de laisser l’utilisateur de l’API créer ses propres classes et les ajouter à la chaîne, en autant que celles-ci implémentent l’interface (ou héritent de la classe abstraite, etc.) Donc, le principe « ouvert-fermé » de SOLID est encouragé par ce patron.

Puisque les différents chaînons de la chaîne doivent en général avoir le même comportement que les autres chaînons, nonobstant leur type en autant que des conditions générales soient respectées, le principe de « substitution de Liskov » est encouragé par ce patron. Chaque chaînon de la chaîne devraient avoir un comportement semblable à celui des autres.

Comme il est encouragé d’utiliser des interfaces pour décrire le comportement des différentes classes utilisées, la dépendance entre les objets de la chaîne et le reste du code est très réduite. Et puisqu’en théorie, cette interface ne fait que décrire le comportement de la chaîne et rien d’autre, les principes de ségrégation d’interface et d’inversion de dépendance sont encouragés.  
 En général, c’est un bon patron à utiliser dans beaucoup de cas. Par contre, par souci d’optimisation et de facilité de maintenance (et de débogage), ce patron peut être vu comme étant négatif jusqu’à un certain point.