Short-Circuit
Design Pattern « Chain Of Responsability »

1) Présentation

En génie logiciel, la "*Chaîne de responsabilité*" est un "*patron de conception*" de la famille "*Comportement*", qui structure un système d'association reliant de façon ordonnée de multiples maillons de processus, respectivement indépendants.

Chaque élément de cette suite représente une instance spécialisée d'un service déterminé, qui tente de consommer/répondre à une requête "*jeton*" passée en paramètre, sans (forcément) connaître les caractéristiques/propriétés opérationnelles détenues par les autres composants, envers ladite demande.

L'objet transmis parcourt progressivement chacun des maillons, qui le procèdent, et peuvent éventuellement (au besoin) l'alimenter/l'enrichir, influant ainsi sur son état.

(cf. patron de conception "*Decorator*" de la famille "*Structuration*", et parallèlement, le principe d' "*Inversion de Contrôle*" (*IoC*), soustrayant à une source une partie de ses capacités fonctionnelles de construction et/ou d'assemblage, en les remettant à des tiers extérieurs alors chargés de la manipuler/gouverner…).

Ce cheminement se poursuit jusqu'à ce qu'une clause de terminaison soit satisfaite:

- D'ordinaire, la demande est résolue/honorée lors de l'inspection pratiquée par l'un des nœuds, (digérant l'élément donné, et clôturant ainsi le cycle de Vie associé à son traitement);

- Ou bien alors, la limite de la séquence est atteinte, et le "jeton" conclu donc sur le dernier cas rencontré (diverses modalités d'exécution sont, dans ces conditions, envisageables: nulle, retour, annulation, rejet, erreur, traçage, action par défaut ou bien triviale, arbitrage simple ou bien complexe sur l'instance potentiellement modifiée, réinjection circulaire, etc…).

L'ordonnancement des membres de la lignée est très souvent « naturel » (linéaire et commutatif - i.e. positions interchangeables), mais peut aussi être parfois « protéiforme » (par ex : assemblé en profondeur et récursif - pour des types plus sophistiqués, comme les chaînes arborescentes couplées au patron de conception "*Composite*", de la famille "*Structuration*").

Il est maintenu, soit par les nœuds entre eux (version la plus courante, chacun pointant sur son prochain/successeur), soit directement par la structure elle même (la collection encadre et coordonne l'ensemble de ses références).

S'il n'est pas commutatif, le séquencement joue alors un rôle prépondérant dans l'accomplissement du processus : Les premiers maillons pouvant préempter et solutionner précocement/promptement la demande, au détriment d'autres composants ayant une position ultérieure au sein de la liste (et donc un rang défavorisé).

Cependant, une fois atteinte(s) et à l'approche de la fin de cycle, la(les) dernière(s) section(s) peu(ven)t parfois (selon la problématique posée...) disposer de plus de pouvoir/responsabilité quant aux traitements effectués.

Sauf exceptions (par exemple, la répétition/reprise d'une position durant la séquence), toute étape antérieure n'ayant pas actée/entérinée la demande, ou plus généralement n'étant pas intervenue, a dès lors cédée son tour et ses prérogatives !

Cette architecture entraîne et privilégie une diminution du couplage/des dépendances entre les classes de Service (affectées à la mécanisation/résolution des appels), et celles de Requête (relatives aux commandes clientes) :

- En économisant/s'épargnant la charge de propager les références des premières à destination des deuxièmes.

- Car dans la gestion des "tickets" entrant, la chaîne n'offrant que son premier maillon comme point d'accès central (les niveaux suivants étant masqués), elle réduit de ce fait la visibilité de sa complexité intrinsèque, et s'avère détentrice exclusive des rouages d'opérations mis en œuvre.

(les unités organisées de cet(te) assemblage/Alliance n'ont, par ailleurs, en commun que le design standard d'une simple interface de réception, d'exécution, et de propagation des "jetons").

Utilisation:

Lorsqu'une source d'information est transmise dans une séquence de traitements.

Remarques:

- Ce modèle (illustré par une très large variété d'usages concrets), propose une approche de séparation/segmentation relativement atomique (disjointe/sans conséquence collatérale) des différentes étapes constituantes de procédures ;

Toutefois en extrapolant, on peut considérer qu'il s'inscrit selon une vision algorithmique "à perspective holistique": pour certaines applications spécifiques, « l'Ensemble » des services effectués par les nœuds, ce "Tout", dépasse la somme de l'apport fonctionnel de chacun d'eux.

- Si le parcours de la chaîne est trop long/coûteux en terme de charge (espace, temps, puissance, …), d'autre patron de conception peuvent s'avérer plus adaptés (comme par exemple le "Mediator" qui joue le rôle d'arbitre sur les requêtes passées).

2) Exemples logiques

On retrouve ce modèle comportemental à de très nombreuses échelles d'algorithmie:

- Les instructions de tri "switch-case/break-default" que l'on retrouve dans la plupart des langages de programmation.

- L'agencement des clauses "catch" dans la gestion des Exceptions (d'abord spécialisées, puis celle générale en dernier) avec des technologies comme Java.

- Dans les Langages (Orienté) Objet:
    - La création d'instances chaîne les appels de constructeurs qui allouent, chacun leur tour et selon la hiérarchie d'héritage, les espaces mémoires des propriétés.
    - Le polymorphisme et la résolution des membres, éventuellement abstraits, (re)définis/implémentés sur n niveaux.

- Les séquences d'instanciation et d'injection de dépendances dans un "Context", avec des Frameworks « *d'Inversion de Contrôle* » (*IoC*) de type Spring.

- Le chainage avant/arrière en Intelligence Artificielle, œuvrant dans le cadre des Systèmes Experts et des Moteurs d'Inférence.

- Le principe de "Queue de Tickets" dans les architectures d'informations asynchrones.

- La gestion de la sécurité et des droits d'accès par filtres d'Authentification et d'Habilitation (les maillons inspectent les droits détenus par la requête, par exemple sur le Protocol HTTP, implémentées sous diverses technologies).

- La résolution de valeur dans les variable d'environnement Système (et notamment le PATH).

- Les architectures réseaux de type "Token Ring".

- etc, etc, etc, ...

3) Applications Pratiques

- La logique et les règles de jeux en tour par tour (jeu de rôles: chacun joue lors de sa phase; jeu quizz: chaque joueur répond à la suite, et celui donnant la bonne réponse emporte le point, ou alors personne ne trouve ni ne marque, et on passe à la question suivante…)

- Les jeux de pistes de type "carte au trésor", où l'on suit une chaîne d'indice tour à tour pour terminer sur le dernier révélant le trésor caché.

- De nombreux processus sociaux, politiques et militaires (chaîne de commandement, élection, accréditation, vote, véto, délibéré, …), et d'une manière générale, dans pratiquement tout mécanisme politique associé à une prise de décision dans des groupes/organisations hiérarchisées (Administration, Armée, ONU, monde du travail…).

- Les organisations de type "Fournisseur, Producteur, Distributeur, Consommateur", et par extension, toutes les chaînes de montage industrielles dédiées à la fabrication de produits finis.

- Dans le cadre de passage d'ordre en Bourse, les chaines de procédures attenantes des systèmes Front-Middle-Back Office (par ex : STP, « *Straight Through Processing* »).

- La parcours d'apprentissage dans le cadre scolaire, et par extension la carrière professionnelle qui s'en suit.

- Les transactions financières de type « enchères ».
- Les phénomènes en cascade.
- Pratiquement toute procédure comportementale.
- Le chaînage MIDI en ingénierie du son.
- La planification et le décisionnel (Robotique, Data Mining, ...)
- Les automates de tri pour pièce de monnaie dans les machines de distribution (qui passent dans une série de calibres/diamètres pour déterminer les pièces entrées)
- Les chaînes de Markov comme modèle statistique mathématique.
- Les réactions nucléaires en physique chimie.
- La Grammaire (et par extension une partie de la linguistique... Chaque définition d'un terme est rendue possible par l'utilisation d'une suite de mots qui l'enrichisse)

4) Citations

Confucius: "Le tout est plus grand que la somme des parties"

5) Code Source

- L'exemple suivant présente une Chaîne de Responsabilité (composée de 4 maillons - 3 de traitement et un de conclusion), en charge d'exécuter des opérations de vérification/validation sur une requête entrante.

- Les nœuds de processing utilisent fonctionnellement des "*Prédicats*", c'est à dire l'expression de "règles" (*notNull*, *instanceof*, *equal, ...*) inspectant un Objet (ici une Requête, passée en paramètre d'entrée), sur une de ses propriété (dans l'exemple le champ "*value*", standardisé selon la norme des JavaBean – getter/setter).

On utilise techniquement 2 différents types de Prédicats (assemblés par le biais du design pattern "*Composite*" - de la famille "*Structuration*"):

- La classe *BeanPredicate*, issue de la bibliothèque d'Apache-Jakarta *Common-BeanUtils*, qui offre des mécanismes de haut niveau en terme de manipulation générique d'instances (lecture/écriture de valeurs sur propriété d'Objet, …).

- Les classes: *NotNullPredicate, InstanceofPredicate, EqualPredicate*, provenant de la librairie d'Apache-Jakarta *Common-Collection*, représentant des opérations spécifiques.

Ces trois éléments mettent en place respectivement les vérifications suivantes :

- La valeur de la propriété ne doit pas être nulle (remarque: construction d'un Prédicat par appel *getInstance*(), Design Pattern "*Singleton*", famille "*Création*").
- Le type associé à la propriété doit être une *String* (chaine de caractère).
- La valeur de la propriété doit être égale à "D3" (valeur arbitraire d'exemple).

Il s'agit bien la d'un ordre cohérent: on vérifie d'abord que la valeur existe, qu'il s'agit bien d'une chaine de caractère, et qu'elle a la valeur "D3".

- Le dernier nœud se charge de terminer le traitement de la chaîne par un simple message sur la sortie standard de la console.

- Les maillons et la requête suivent un design basé sur des interfaces (*IChainNode*, et *IRequest*), ce qui est standard en Java pour définir le comportement d'objets.

```
/** Design d'interface: accesseur/modifieur pour la propriété "nextNode" + méthode de
traitement */

public interface IChainNode
{
      public IChainNode   getNextNode();
      public void         setNextNode(IChainNode nextNode);
      public void         process(IRequest r);
}

---

import org.apache.commons.collections.Predicate;

/** Design d'interface: accesseurs à l'Object "value" + la liste des Predicats vérifiés,
méthode du pattern Décorator */

public interface IRequest
{
      public Object             getValue();
      public void               decorate(Predicate matchingRulePredicate);
      public List<Predicate>    getMatchingPredicates();
}
```

- *IChainNode* est partiellement implémentée par la classe « *abstraite* » *AbstractProcessorNode* (qui, détenant la référence du nœud suivant, fournie les méthodes "*get/set NextNode*") ; puis finalisée par les classes « *concrètes* » (héritant d'*AbstractProcessorNode*) qui complètent cette réalisation en fournissant le corps de traitement réel et spécifique du maillon (méthode "*process*").

```
/** Classe d'abstraction de haut niveau, fournissant:
 * - La déclaration dans sa signature de classe de l'implémentation de IChainNode
 * - Un constructeur simple/trivial pour la commodité des sous-classes
 * - L'implémentation des getters/setters de "nextNode" (associé à l'interface IChainNode),
     à travers une propriété privée dédiée.
 *
 * Remarque: La responsabilité de l'implémentation de la méthode "process", requise et
   restante depuis la déclaration d'IChainNode, est renvoyée aux sous-classes concrètes.
*/

public abstract class AbstractProcessorNode implements IChainNode
{
    private IChainNode nextNode;


    public AbstractProcessorNode() {}

    @Override
    public IChainNode getNextNode()                 {return nextNode;}
    @Override
    public void setNextNode(IChainNode node)        {this.nextNode = node;}
}


---
```

```
import org.apache.commons.collections.Predicate;

/** Une classe de traitement, représentant un noeud de la chaîne,
 * spécialisée dans l'évaluation de Prédicat sur des objets IRequest cibles.
 */

public class PredicateProcessorNode extends AbstractProcessorNode
{
	/** Le Prédicat local, qualifiant ce noeud. */
	private Predicate   predicateRule;


	public PredicateProcessorNode(Predicate predicateRule)
	{
		this.predicateRule = predicateRule;
	}

	/** Traite l'IRequest donnée avec le Predicat,
	 *
	 * si:		La règle évaluée est vérifiée/vraie,
	 * alors:	L'IRequest est décorée par rappel avec le Predicat actuel
			(ajouté à sa liste),
	 *		et le noeud essaye de faire suivre la IRequest à son successeur
			(s'il existe)
	 * sinon:	La chaîne se termine avec ce noeud courant.
	 */

	@Override
	public void process(IRequest request)
	{
		if (predicateRule.evaluate(request))
		{
			IChainNode nextNode = getNextNode();

			request.decorate(predicateRule);

			if (nextNode != null)
				nextNode.process(request);
		}
	}
}

---

/** Un simple noeud de fin/conclusion, affichant juste un message.
 * Remarque: Emplacement alloué pour des instructions de traitement additionels... */

public class TerminalProcessorNode extends AbstractProcessorNode
{
	@Override
	public void process(IRequest request)
	{
		System.out.println("TerminalProcessorNode: process: hey '" + request.toString()
		 + "', you've reached terminal node, congratulations, chain is over...");
	}
}
```

- *IRequest* est implémentée par la classe « *concrète* » *Request*, qui fournie un accès à la propriété "*value*" (méthode "*getValue*", permettant l'inspection par les nœuds de Prédicats), et une méthode publique « *decorate* » (illustrant le patron de conception « *Decorator* » - Famille « *Structuration* »), offrant aux objets extérieur la possibilité d'ajouter à sa liste interne les règles ayant été vérifiées, lors du parcours de la chaîne (liste ensuite accessible par l'appel « *getMatchingPredicates* »).

```
/** Une simple classe "jeton" de requête, fournissant un minimum de propriétés
fonctionnelles */

public class Request implements IRequest
{
	/** Une liste de règle/prédicat, que l'objet courant vérifie,
	 * et alimentée selon le Design Pattern "Decorator" */
	private List<Predicate>   matchingPredicates = new ArrayList<Predicate>();

	/** Titre de l'objet */
	private String                  title;

	/** Une simple valeur, typée "Object" */
	private Object                  value;


	public Request(String title, Object value)
	{
		this.title = title;
		this.value = value;
	}

	/** Injection de valeurs par rappel/callback */
	public void decorate(Predicate matchingRulepredicate)
	{
		matchingPredicates.add(matchingRulepredicate);
	}

	/** Une redefinition de java.lang.Object.toString composée de propriétés locales */
	@Override
	public String toString()
	{return title + " (value=" +
			((value == null)? "null" : value.toString()) + ")";}

	/** propriétés en lecture seule (pas de setter/d'accès en écriture) */
	public Object getValue()               {return value;}
	public String getTitle()               {return title;}

	public List<Predicate> getMatchingPredicates()
	{return matchingPredicates;}
}
```

- La classe *Main* de test se charge de construire les 4 maillons, les 4 requêtes (valeurs: nulle, numérique *Integer*, chaine de caractère non correspondante, puis correspondante), d'assembler la chaîne, puis de lancer les traitements, et enfin d'afficher sur la console les informations des Prédicats vérifiés, lorsqu'évalués positivement par les nœuds sur les instances d'*IRequest*, alors décorées.

```
import java.util.ArrayList;
import java.util.List;

import org.apache.commons.beanutils.BeanPredicate;
import org.apache.commons.collections.Predicate;
import org.apache.commons.collections.functors.EqualPredicate;
import org.apache.commons.collections.functors.InstanceofPredicate;
import org.apache.commons.collections.functors.NotNullPredicate;


/**
 * Une classe principale:
 *
 * - Instanciant les 3 noeuds de traitements Predicats
     + le noeud de traitement de conclusion.
 *
 * - Définissant l'initiateur de chaîne (premier noeud),
     qui va déclencher la réaction comportementale de la chaîne,
     lorsqu'appelée par une partie cliente.
 *
 * - Construisant des éléments variés de type IRequest,
     avec différents arguments satisfaisant aux diverses possibilités de la chaîne.
 *
 * - Utilisant IChainNode.setNextNode pour établir l'ordre des noeuds de la chaîne:
     nodeNotNull -> nodeInstanceOfString -> nodeValueEquals -> nodeTerminal
 *
 * - Appelant le processing de la chaîne sur chacune des instances de type IRequest.
 *
 * - Loggant avec System.out la liste IRequest.matchingPredicates décorée,
     résultant du parcours de la chaîne.
 *
 * Résultats:
 *
 * - L'objet requestNull, avec une propriété "value" null non affectée,
     échoue sur le premier noeud: un NotNullPredicate.
 *
 * - L'objet requestInteger, avec une propriété "value" définie en Integer,
     passe au travers du NotNullPredicate
     mais s'arrête sur le InstanceofPredicate(String.class)
 *
 * - L'objet requestStringNOK, avec une propriété "value" à "testNOK",
     passe au travers des NotNullPredicate et InstanceofPredicate(String.class)
     mais échoue sur le EqualPredicate("D3")
 *
 * - L'objet requestStringOK, avec une propriété "value" à "D3",
     vient à bout du NotNullPredicate, du InstanceofPredicate(String.class) et
     du EqualPredicate("D3"), atteignant le nodeTerminal final,
     qui renvoit un message clôturant le traitement de la chaîne.
 *
 */

public class Main
{
       /** Le noeud primaire de la chaîne */
       public static IChainNode chainInitiator;


       public static void main(String args[])
       {
              //Création des noeuds
              IChainNode nodeNotNull           = new PredicateProcessorNode(
                                                        new BeanPredicate("value",
                                                               NotNullPredicate.getInstance()));
```

```
            IChainNode nodeInstanceOfString  = new PredicateProcessorNode(
                                                   new BeanPredicate("value",
                                                       new InstanceofPredicate(String.class)));
            IChainNode nodeValueEquals       = new PredicateProcessorNode(
                                                   new BeanPredicate("value",
                                                       new EqualPredicate("D3")));

            IChainNode nodeTerminal          = new TerminalProcessorNode();

            //Initialisation du nœud primaire
            chainInitiator                   = nodeNotNull;

            //Création des objets de requêtes
            Request requestNull              = new Request("requestNull",           null);
            Request requestInteger           = new Request("requestInteger",         42);
            Request requestStringNOK         = new Request("requestStringNOK",      "testNOK");
            Request requestStringOK          = new Request("requestStringOK",       "D3");

            //initialisation de l'ordre de la chaîne
            nodeNotNull.setNextNode(               nodeInstanceOfString);
            nodeInstanceOfString.setNextNode(      nodeValueEquals);
            nodeValueEquals.setNextNode(           nodeTerminal);

            //Lancement des traitements de la chaîne sur chacune des requêtes
            chainProcess(requestNull);
            chainProcess(requestInteger);
            chainProcess(requestStringNOK);
            chainProcess(requestStringOK);
      }

      /** Démarre la chaîne sur le paramêtre IRequest donné. */
      private static void chainProcess(IRequest request)
      {
            chainInitiator.process(request);

            logMatchedPredicates(request);
      }

      /** Log sur la console les informations de la requête, et
          le résultat du processus de décoration,
          effectué par la chaîne sur la propriété "matchingPredicates", de type List */
      private static void logMatchedPredicates(IRequest request)
      {
            List<String> l      = new ArrayList<String>();

            for (Predicate p :  request.getMatchingPredicates())
                  l.add(((BeanPredicate) p).getPredicate().getClass().getSimpleName().toString());

            System.out.println("Main: logMatchingPredicates: '" + request.toString() + "', "
                               + request.getMatchingPredicates().size()
                               + " prédicats vérifiés: " + l.toString() + "\r\n");
      }
}
```

Traces de la sortie standard :

```
Main: logMatchingPredicates: 'requestNull (value=null)',
0 prédicats vérifiés: []

Main: logMatchingPredicates: 'requestInteger (value=42)',
1 prédicats vérifiés: [NotNullPredicate]

Main: logMatchingPredicates: 'requestStringNOK (value=testNOK)',
2 prédicats vérifiés: [NotNullPredicate, InstanceofPredicate]

TerminalProcessorNode: process: hey 'requestStringOK (value=D3)', you've reached terminal node,
congratulations, chain is over...

Main: logMatchingPredicates: 'requestStringOK (value=D3)',
3 prédicats vérifiés: [NotNullPredicate, InstanceofPredicate, EqualPredicate]
```