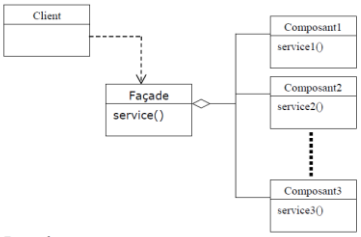
Design pattern

Facade

C'est l'un des Design Patterns les Plus utilisés et les plus simples à mettre en œuvre (basé uniquement sur l'agrégation). Il s'agit d'encapsuler un ensemble d'objets de classes variées dans une même classe appelée façade permettant de simplifier l'accès (généralement complexe) aux différentes classes.

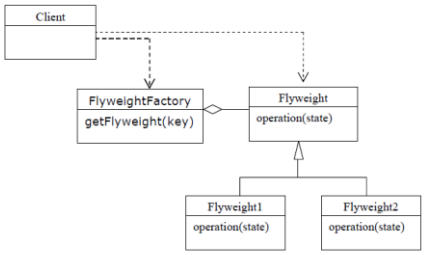


Exemples :

- Une classe Database encapsulant : Class, Connexion, DatabaseMetaData, Resultset, etc…
- Une classe Form encapsulant des JLabel, JTextField, etc..

Poids mouche : Flyweight

Permet de repérer des objets utilisés plusieurs fois dans la même application. La solution est d'en créer une seule instance à enregistrer dans une fabrique (Flyweight Factory) qui se charge de délivrer ces objets aux clients demandeurs. Ces objets peuvent être identifiés à l'aide d'une clé utilisée pour leur extraction depuis la fabrique (la fabrique peut par exemple être une Hashtable). Ils peuvent aussi être configurables par l'intermédiaire de paramètres de configuration externes appelés : propriétés extrinsèques.



Exemple :

- Un bouton « Exit » avec son comportement (L'ActionListener permettant de fermer la fenêtre).
- Un bouton « OK », mais le comportement (ou l'étiquette) constitue la propriété extrinsèque sous forme d'un ActionListener (ou setText(…)).

Proxy

Il ressemble dans sa structure à un Adapter mais avec des objectifs différents :

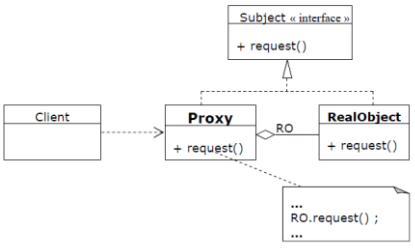
- Contrôler ou Sécuriser les accès aux services d'un RealObject
- Limiter les services accessibles en fonction de la nature du client ➔ un Proxy différent par Client différent (mais, avec la même interface d'accès)
- Corriger les services du RealObject pour éviter des accès pouvant se terminer en erreur.

Exemples :

1- une méthode getName(…) qui retourne dans le RealObject un String pouvant être nul ➔ getName().equals(…) peut se terminer en erreur. A corriger par un Proxy retournant le cas échéant une chaine vide.

2- getImage() dans le Browser permettant de télécharger les images ➔ tant que l'image n'est pas prête le document est bloqué. Correction : un Proxy qui délivre le cas échéant une icône de remplacement ➔ on peut continuer l'interprétation du document en attendant l'arrivée des vraies images.

- On peut envisager de fournir de nouveaux services faisant appel à plusieurs services du RealObject



- On pourra aussi envisager un service de journalisation permettant d'améliorer les services fournis par le RealObject.

Exemple :

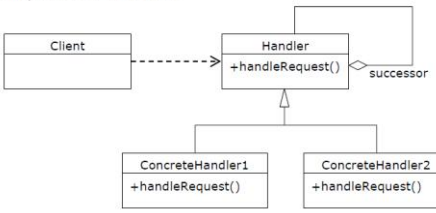
```
public class RealObject {
      public void insertData(…) {
            …
      }
}
```

➔

```
public Proxy {
      private RealObject RO ;
      private Logger logger = new Logger() ;
      Proxy(RealObject RO) {
            This.RO = RO ;
      }
      public void insertData(…) {
            RO.insertData() ;
            logger.setLog("insert",
                        "data inserted", date, client …);
      }
}
```

Chain of Responsibility

C'est une structure de classes implémentée sous forme de liste chaînée polymorphe dont chaque élément a une responsabilité vis-à-vis d'un objet en entrée. Ainsi, ce dernier peut être passé au premier nœud de la chaîne. Chaque nœud se charge d'analyser l'objet en entrée ; s'il peut lui appliquer le traitement dont il est responsble il le fait, sinon il le fait passer au nœud suivant.



Comme exemples d'application de ce Design Pattern, on pourra citer :

1. Une chaîne de production, dans laquelle chaque nœud apporte un traitement à l'objet en cours de construction.
2. Une chaîne d'employés (chef de services) qui apportent chacun un traitement à un papier administratif : la secrétaire rédige un papier, le chef corrige le papier, le directeur signe, …
3. Une structure d'héritage dans laquelle chaque constructeur d'une classe appelle le constructeur de la classe mère. Dans ce cas, tous les constructeurs peuvent apporter du nouveau à l'objet à construire. Le mot clé super permettra de faire passer une information à traiter d'un constructeur à l'autre.
4. Une chaîne d'automates et dont un seul sera capable d'extraire la prochaine unité lexicale.

Remarque :

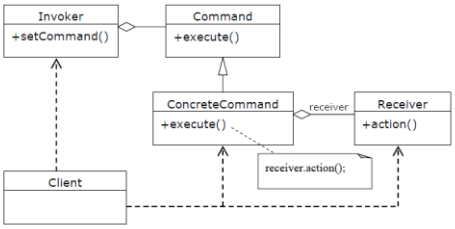
Il est nécessaire dans ce cas de considérer un traitement par défaut, le cas où aucun élément de la chaîne n'arrive à traiter l'objet.

Command

Il s'agit d'associer une commande « Command » différente à chaque objet différent. L'objet est appelé « **Invoker** », ce qui permettra d'avoir l'exécution de la commande une fois l'Invoker connu et sans avoir à procéder par des « **if…else** ».

L'exemple le plus approprié en Java est l'implémentation des écouteurs d'événements : à chaque contrôle de l'interface (« l'Invoker ») on associe, par l'intermédiaire d'une méthode addXListener, un écouteur (« **ConcreteCommand** ») qui implémente l'interface XListener (« **Command** »).

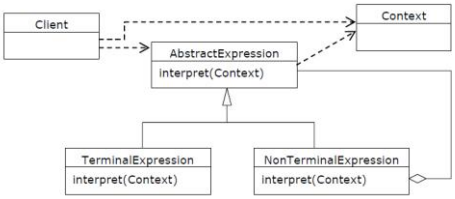
Le « **Receiver** » est dans ce cas le contrôle qui va être utilisé par l'écouteur (ou qui va subir l'action de l'écouteur).



Intrepreter

Il permet d'implémenter sous forme de Composite un schéma de traduction dirigée par la syntaxe : C'est une implémentation de l'analyseur syntaxique d'un compilateur en vue d'interpréter un programme ou une expression en entée.

Le schéma est implémenté d'une manière arborescente, où chaque nœud de l'arbre est soit un terminal (dans tel cas c'est une feuille) ou un non-terminal et dans ce cas il représente une sous expression à interpréter.

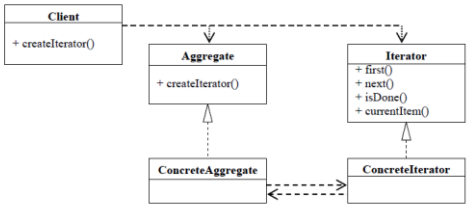


Exemple :

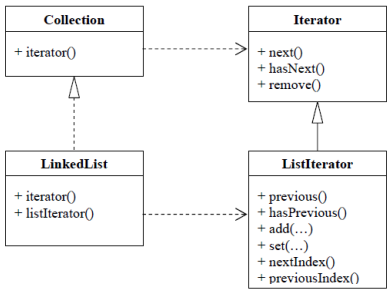
Un évaluateur d'expressions arithmétiques.

Iterator

Ce Design Pattern donne une solution permettant de garder un pointeur sur les nœuds consécutifs d'une liste chaînée (appelée aussi aggrégat d'objets : **aggregate**). Celui-ci (le pointeur) sera accessible par l'utilisateur final sans exposer la structure interne de la liste. La soluton fournit les différents services permettant de parcourir la liste et de récupérer les éléments de celle-ci. Il s'agit des méthodes : **first()**, **next()**, **currentItem()** ainsi que **isDone()** pour détecter la fin de la liste.



Il est à remarquer que l'implémentation de l'itérateur dépendra de la structure de la liste implémentée. C'est le cas des collections en Java :

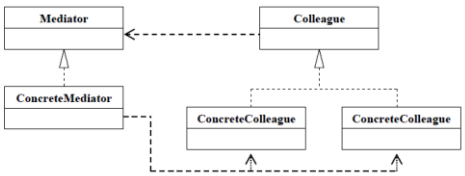


Il est à Remarquer que :

- La méthode next() remplace les 3 méthodes proposées par les GOF, qui sont : first(), next() et currentItem(). En effet, le premier appel à next() donne le même résultat que first() ; d'autre part, la fonction next() retourne la référence de l'objet courant.
- Iterator et ListIterator sont des interfaces implémentées localement dans « LinkedList » (en réalité, elle sont implémentées dans AbstractList qui est la classe abstraite mère de LinkedList).

Mediator

Ce Design Pattern permet de centraliser (comme un Hub ou un concentrateur) le traitement dans une seule classe appelée « **médiateur** ». Ainsi au lieu de faire connaître un objet X à tous les autres objets de l'application (les collègues), on le fait connaître uniquement au médiateur. Tout collègue désirant demander un service à l'objet X, il le fait à travers le Médiateur. Il serait alors très simple de mettre à jour le collègue X ou ses services ou le remplacer complètement, aucune mise à jours n'est alors effectuée au niveau d'aucun collègue. Le seul code modifié est celui du médiateur.



Exemple :

Soit une zone de texte T1 déstinée à contenir des dates. On considère un médiateur permettant l'accès à cette zone de texte :

```
public class Mediator {
      private JTextField date ;

      public Mediator() {
      }

      public void setDateColleague(JComponent date) {
            this.date = (JTextField)date;
      }

      public void setDate(String text) {
            date.setText(text);
      }

      public String getDate() {
            return date.getText();
      }
}
```

Dans les collègues, au lieu de faire :

T1.setText(uneDate) ;

On procède comme suit :

m.setDate(uneDate) ;

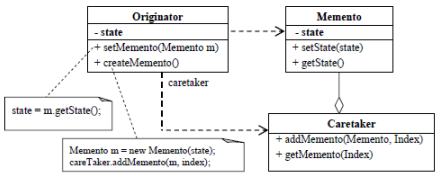
avec m étant un objet de type Mediator.

Si on veut changer la zone de texte par un autre contrôle, il suffira de mettre à jour le code du médiateur, les autres collègues ne seraient aucunement modifiés.

Momento

C'est une solution permettant l'enregistrement des états d'un objet bien déterminé, appelé « **Originator** », à des moments particuliers, en vue de retrouver ces états dans le futur. On considère ainsi un objet dont la valeur de l'une de ses propriétés évolue tout au long de l'existence de l'objet. Par exemple un éditeur de texte tel que le contenu du document en cours de rédaction change d'un instant à l'autre, on pourra donc envisager des opérations « undo » (annuler). Un autre exemple : un salarié tel que la propriété « salaire » peut changer et on voudrais se rappeler des valeurs antérieurs, etc.

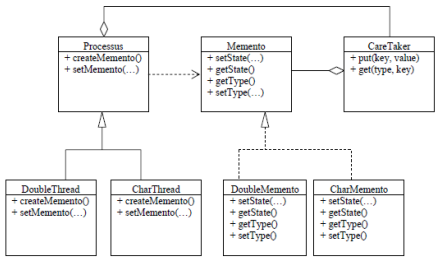
La solution du Mémento permettra de sauvegarder sous forme d'un objet appelé « **Memento** » uniquement la valeur de cet état (classiquement, on pourrait penser à enregistrer tout l'objet : le Salarié par exemple). La sauvegarde de ces Mémentos est réalisée dans une structure de données permettant de prendre soin des Mémentos enregistrés, appelée « **Caretaker** ».



Exercice :

Réalisation d'un Thread disposant d'un état de type réel (double) qui s'incrémente systématiquement dans le run() (par pas de 0,01). Le Thread s'occupe lui-même de sauvegarder ces Memento, avec comme « Index » l'instant : date et heure, dans un Caretaker basé sur une Hashtable. Le client pourra demander de retrouver l'état associé à un instant donné.

On pourra aussi réaliser un autre « originator » ; ce sera une classe Thread qui gère des lettres. L'état est une lettre qui change aléatoirement (ou par pas de 1). On aura donc une autre classe Memento mais un même CareTaker.



Observer

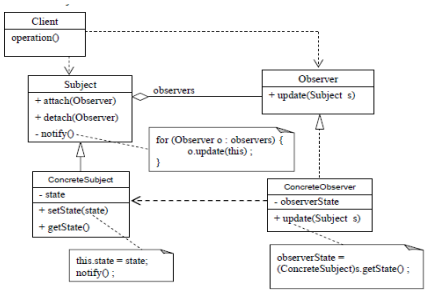
Le design pattern « Observer » permet de résoudre le problème suivant :

Faire en sorte que : à chaque modification d'un objet donné (un sujet), un ou plusieurs autres objects d'autres classes (appelés observateurs) prènent conciance de cette modification ce qui leur permettra de se mettre à jour par rapport au changement du « sujet » d'origine.

En réalité ce n'est pas l'observateur qui reste à l'écoute de tout changement du sujet, mais c'est le sujet lui-même qui à chaque modification il avise les observateurs un par un.

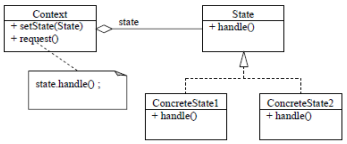
Solution :

+ La classe « Subject » doit contenir une structrure permettant d'y enregistrer tous les observateurs possibles. Donc deux méthodes pour attacher et détacher des observateurs.
+ Une méthode « notify » permettant d'informer les observateurs du changement. Celle-ci doit être appelée à chaque changement du « subject »



State

Ce design pattern permet à un processus, ou généralement à un objet, de changer de comportement lorsque son état change. L'opération de changement de comportement sera réalisée sans avoir à faire des tests sur l'état (if … else if…). La solution consiste à transformer l'état en un objet encapsulant au sein de lui-même le comportement qui lui est associé. Ainsi, pour fait changer d'état à un processus, on lui communique le nouvel état sous forme d'un objet « State ».

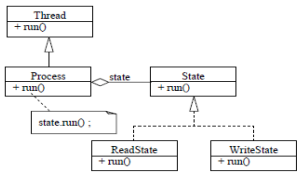


Remarque :

Le processus « Context » peut évoluer (chager d'état) d'une manière autonome, comme il peut être piloté depuis l'extérieur par un autre processus qui lui fait chager d'état selon le besoin.

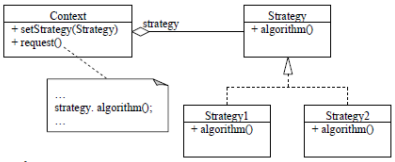
Exemple :

On voudrait réaliser un processus de Lecture/Ecriture. Le processus change d'état (lecture/écriture) aléatoirement (ou commandé par le client).



Strategy

Il permet à un processus (context) de changer de strategie (d'algorithme), ou encore de « manière de faire », sans avoir à le recompiler. La stratégie se trouve externalisée et peut être injectée dynamiquement dans le processus.



Exemple :

+ Algorithmes de tri.
+ Un Parseur XML

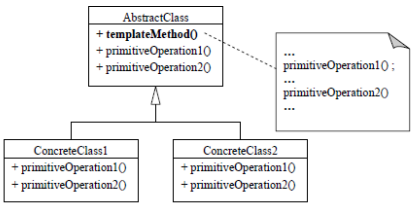
Remarque :

Ce design pattern ressemble au design pattern State dans sa structure mais avec la différence suivante :

Le design pattern resoud le problème de changement d'état d'un (pouvant être fréquent ou le long de son cycle de vie) qui entraire un changement de comportement. Par contre, le design pattern strategy permet juste de paramétrer le processus pour le choix d'un seul algorithme à appliquer dans un contexte donné.

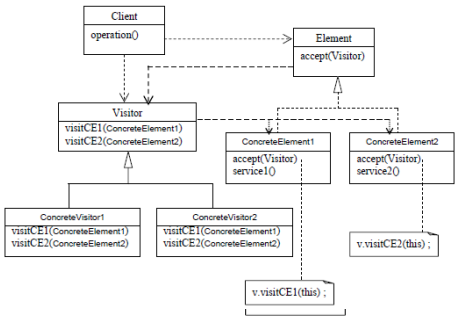
Template Method

Ce design pattern permet de réaliser un algorithme dont cetraines étapes (primitives) ne sont pas figées et peuvent changer celon le besoin. Le squelette ou la structure générale de l'algorithme reste constante, mais le développeur aura la posibilité, à l'aide de l'héritage, de redéfinir (ou définir) certains blocs de l'algorithme qui sont détachés sous forme de méthodes abstraites.



Visitor

Le visiteur est une structure de classes permettant à une classe (« Element ») d'être extensible en terme de services sans avoir à la modifier ou à l'étendre (par héritage). C'est par l'intermédiaire d'une autre classe, le visiteur, autorisée à visiter celle-ci (« Element ») que des services supplémentaires seront rajoutés.



Remarques :

1. Ce design pattern permet d'éviter de polluer des classes avec une multitude de services variés et qui change d'une classe à une autre.
2. Il permet d'éviter 2 problèmes :
   - Retoucher à chaque fois une classe « Element » avec des services supplémentaires.
   - Utiliser des switch pour faire des traitements adapté à la nature de l'élément à traiter.
3. Il y a donc autant de visiteurs (exp. Afficheur, Lecteur) à ajouter que de services à programmer (afficher, lire).
4. Un visiteur s'occupe d'un et un seul type de service (exp. Affichage)
5. Il y a autant de méthodes dans le visiteur que d'éléments à visiter (exp. A, B, C ➔ vistA, visitB, visitC). Toutes les méthodes font la même chose (exp. Affichage), mais elles portent des noms différents, des paramètres différents et un contenu ou une manière de faire différente. En fait, le paramètre de chaque méthode est l'élément sur lequel le service sera appliqué (visitA(A), visitB(B), visitC(C)).

Architecture N-tiers