Design Patterns - 3

Les patterns de comportement

Bertrand LIAUDET

SOMMAIRE

SOMMAIRE 1

LES DESIGN PATTERNS 3

Les 11 patterns de comportement 3

Objectif 3

Répartition des traitements : héritage et délégation 3

Héritage *vs*. Délégation 4

Héritage 4

Délégation 5

Exercices 5

Le pattern Strategy (délégation) 6

Description du pattern (objectif, définition) 6

Principes de résolution 6

UML 6

Exercices 7

Le pattern Observer (délégation) 8

Description du pattern (objectif, définition) 8

UML 8

Description de la structure 9

Séquence de la mise à jour 9

Exercices 10

Le pattern Command (délégation) 11

Objectif du pattern 11

UML 11

Exercices : reprise de DP TLP 14

Tester l'exemple de Commande.zip 15

Le pattern State (délégation) 16

Objectif du pattern 16

Principes de résolution 16

UML 16

Exercices 19
Le pattern Template method = Patron de méthode (héritage) 20
Objectif du pattern 20
Raisons de l'utiliser 20
UML 20
Exemple d'utilisation 20
Exercice 21
Le pattern Iterator (délégation) 22
Objectif du pattern 22
Raisons de l'utiliser 22
Diagramme de classes 22
Principes techniques 22
Exercices 23
Tester l'exemple de Iterator.zip 23
Le pattern Interpreter (héritage) 24
Objectifs 24
Diagramme de classes 24
Exercices 24
Le pattern Chain of responsability (délégation) 25
Objectifs 25
Principe de réalisation 25
Diagramme de classes 25
Exercices 26
Le pattern Mediator (délégation) 27
Objectif du pattern 27
UML 27
Exercices 27
Le pattern Memento (délégation) 28
Objectif du pattern 28
UML 28
Exercices 28
Le pattern Visitor (délégation) 29
Objectif du pattern 29
Principes 29
UML 29
Exercices 30

Première édition : mai 2011 – Mise à jour mai 2012, juin 2022

LES DESIGN PATTERNS

Les 11 patterns de comportement

https://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/?page=comportementaux-behavioral-patterns

Objectif

Les patterns de construction encapsulent l'instanciation.

Les patterns de structuration encapsulent l'interface d'un objet ou d'un groupe d'objets.

Les patterns de comportement organisent la répartition des traitements entre les objets

Répartition des traitements : héritage et délégation

La répartition des traitements en programmation objet peut prendre deux formes : l'héritage ou la délégation.

L'héritage permet de récupérer les traitements des classes dont on hérite.

La délégation permet de déléguer le calcul à un autre objet.

Il y a 2 patterns d'héritage et 9 patterns de délégation.

	2 patterns d'héritage	2 patterns de délégation
Principal	• Template methode	• Strategy • Observer • Command • Iterator • State
Secondaire	• Interpreter	• Chain of responsibility • Mediator • Memento • Visitor

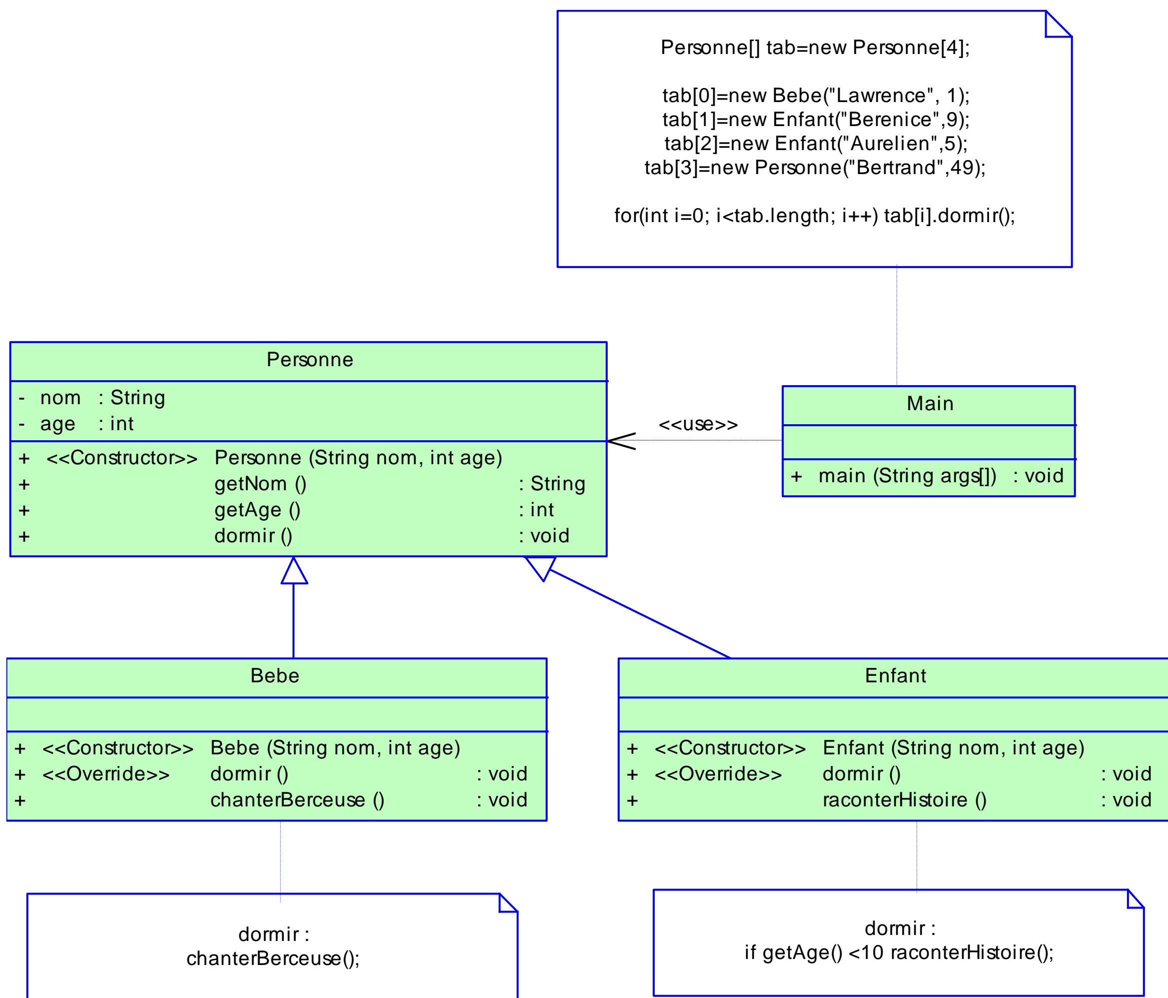
Héritage *vs.* Délégation

La délégation offre un mécanisme de réutilisation aussi puissant que celui offert par la généralisation.

L'héritage est une construction rigide mais la propagation des attributs et des méthodes est automatique. L'héritage correspond à une relation « est-a » stricte, c'est-à-dire pas à une relation « a-un-comportement-de ».

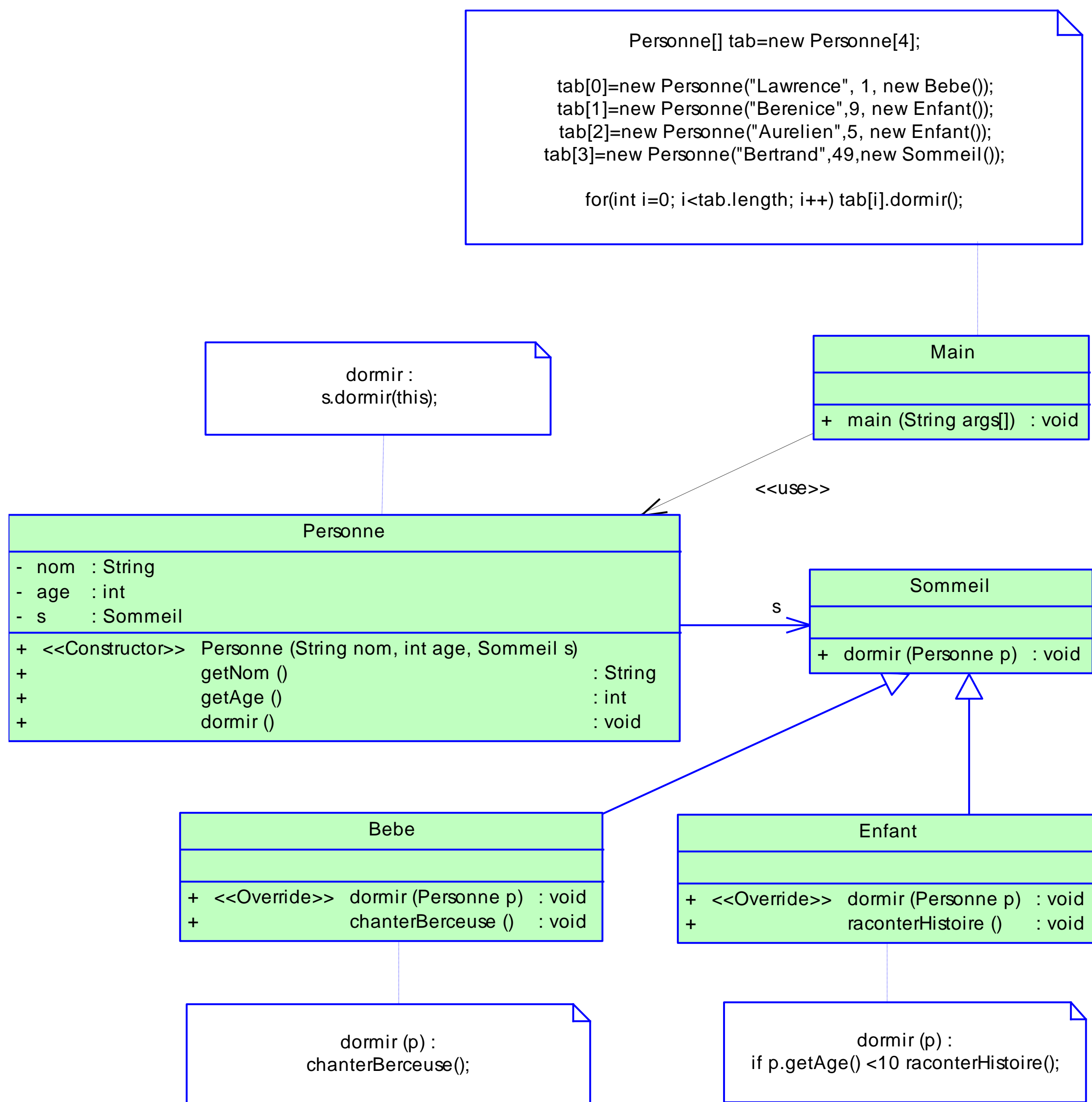
La délégation est une construction plus souple basée sur l'agrégation. La propagation des propriétés doit être réalisée manuellement. Elle permet généralement la mise en œuvre de généralisation multiple.

Héritage



Ici le mécanisme est un mécanisme classique d'héritage et de polymorphisme.

Délégation



Ici on obtient le polymorphisme par la délégation.

On fournit l'objet « p » à la méthode dormir pour accéder aux méthodes de Personne, si nécessaire (ici : if p.getAge <10…).

On peut considérer qu'on « sette » une méthode spécifique à la personne au moment de son instanciation : « new Personne("Lawrence", 1, new Bebe()); » consiste à setter la méthode « dormir » des Bébé à la personne, comme on lui a setté son prénom et son âge.

Cette technique sera particulièrement utile en cas d'héritage multiple car cette fois-ci elle offrira une solution plus performante que la solution de l'héritage classique. On verra ça particulièrement dans le design pattern Stratégie.

Exercices

Coder les deux versions de l'endormissement des enfants.

Le pattern Strategy (délégation)

Description du pattern (objectif, définition)

- **Le pattern** Strategy encapsule des comportements d'une même famille (des méthodes plutôt que des attributs) et utilise la délégation pour savoir lequel utiliser.
- « Sette » des méthodes à travers une interface.
- Adapter le comportement d'un objet en fonction d'un besoin sans changer les interactions, et donc indépendamment du client.
- Définir une famille d'algorithmes interchangeables et permettre de les changer indépendamment de la partie cliente.

Principes de résolution

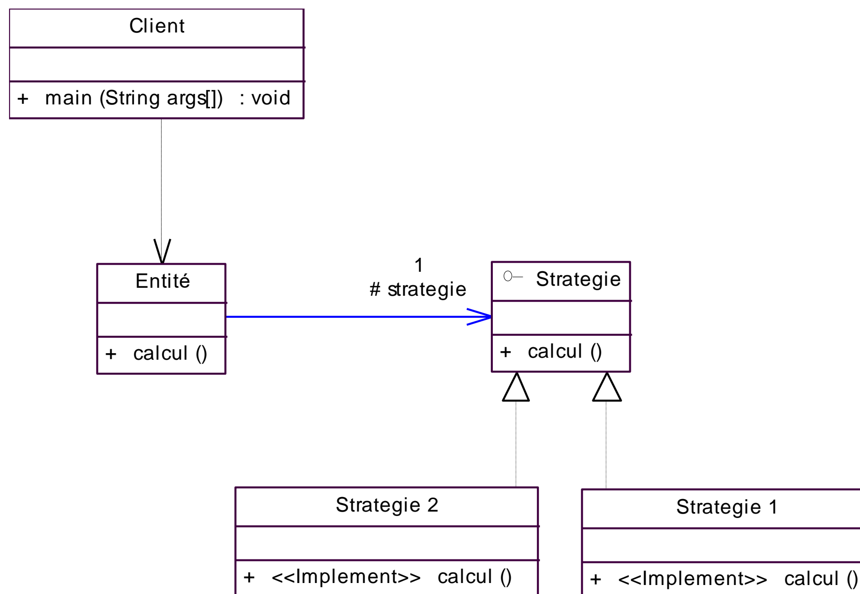
Les objets de la classe qui utilise la classe « Stratégie » contiennent un objet de la classe « stratégie » et sont instanciés en passant en paramètre un objet correspondant à une stratégie concrète.

Ainsi, ces objets accèdent de façon polymorphe aux méthodes de la classe « Stratégie »

La classe Stratégie est une interface qui définit les comportements.

Les stratégies concrètes réalisent cette interface.

UML



Exercices

Coder l'exemple des canards des DP TLP

Le principe est le suivant :

On gère des canards. Tous les canards peuvent voler, cancaner, nager et on peut afficher une phrase qui les décrit.

On a plusieurs sortes de canards : des colverts et des mandarins, qui sont des vrais canards, qui volent et qui font « cancan », des canards en plastique qui ne savent pas voler et qui font « coin-coin », des leurres qui ne savent pas voler et qui sont silencieux.

On a aussi des prototypes de canards qui, par défaut, ne savent pas voler et font « cancan ».

On écrira un programme qui crée une liste avec tous les types de canards puis affiche ce qu'ils sont, les fait cancaner et voler.

Enfin, on modifiera le vol par défaut du prototype de canard pour le faire passer à un vol à réaction !

Tester l'exemple de ENI

Faites le diagramme de classes.

Coder l'exemple de développer.com

https://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/?page=comportementaux-behavioral-patterns#LVI-I

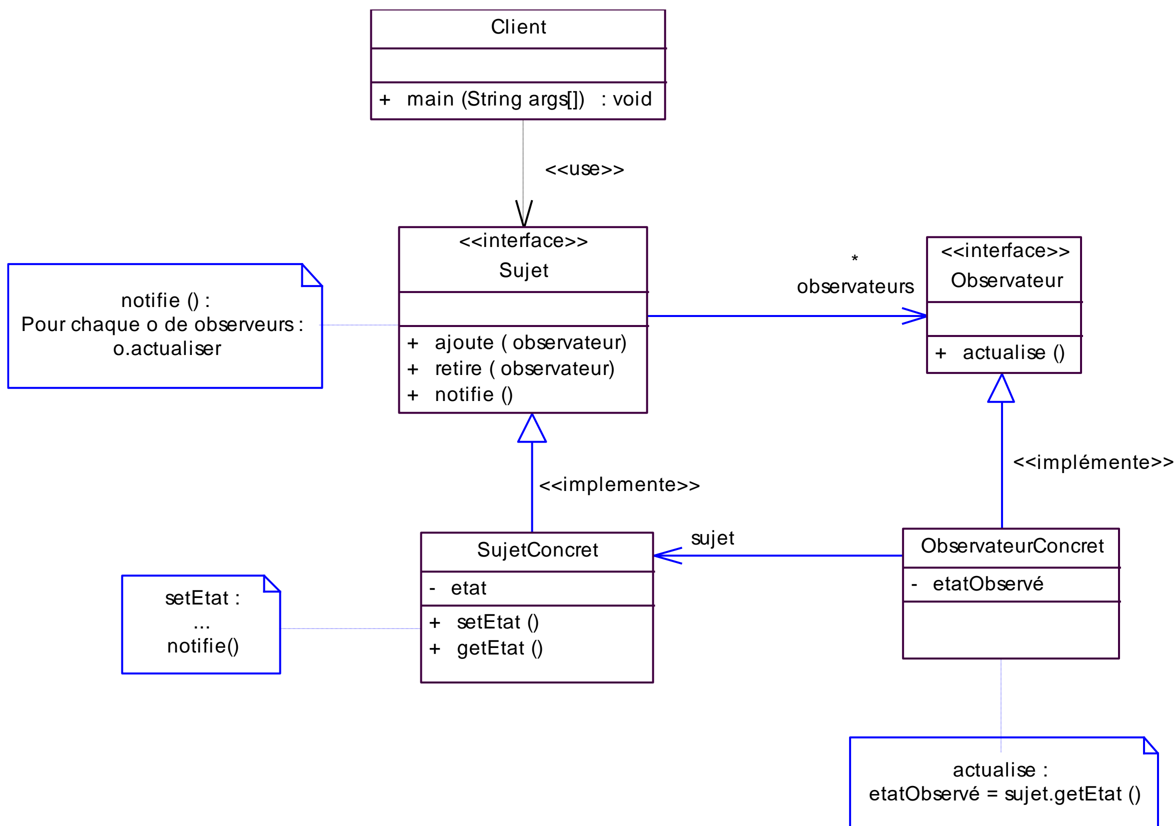
Coder, tester et comprendre l'exemple.

Le pattern Observer (délégation)

Description du pattern (objectif, définition)

- Un objet (le sujet) est observé par plusieurs autres (les observateurs). Quand un événement arrive à l'objet observé, on prévient tous ceux qui l'observent (notification).
- **Le pattern Observer** construit donc une dépendance entre un sujet et des observateurs de sorte que chaque modification du sujet soit notifiée aux observateurs afin qu'ils puissent mettre à jour leur état.

UML



Description de la structure

Coté Sujet

Le sujet observé implémente l'interface « Sujet » : ça l'oblige à se doter de méthodes pour gérer les « Observateurs ». Le sujet gère donc une collection d'Observateurs : on peut ajouter ou retirer un observateur de la collection. Toute classe implémentant l'interface Observateur pourra donc faire partie de la collection.

Le sujet implémente une méthode standard : « notifie » qui consiste à actualiser tous les observateurs. La méthode notifie appelle la méthode actualise des observateurs. Actualise() concret est obtenu par l'observateur concret (polymorphisme, logique de pattern Strategie).

La méthode « notifie » est appelée par le sujet à chaque fois qu'il modifie son état (qu'il sette son état).

Coté Observateur

L'observateur implémente une méthode « actualise ».

La méthode actualise met à jour l'observateur en récupérant l'état du sujet : en effet, l'observateur contient un attribut sujet concret.

Que l'observateur contienne un attribut qui implémente l'interface « Sujet » lui permet aussi de pouvoir s'enregistrer comme observateur auprès du sujet et aussi de se retirer.

Remarque

Une association orientée entre deux interfaces veut dire qu'on a en réalité une association orientée entre une classe implémentant l'interface de départ et l'interface d'arrivée.

Séquence de la mise à jour

1. Le client setEtat() le sujet
2. Le setEtat() du sujet notifie() le sujet
3. Le notifie() du sujet actualise() tous les observateurs
4. L'actualise() de l'observateur récupère l'état du sujet, soit par les paramètres d'appel de l'actualise, soit par un getEtat() du sujet

Exercices

Partez de l'exemple de DP TLP sur la station météo

Faites fonctionner l'exemple.

Pour comprendre son fonctionnement :

- Faites le diagramme de classe avec PowerAMC en utilisant le reverse ingeenering
- Faites le diagramme de séquence du main.

Coder l'exemple de développer.com

https://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/?page=comportementaux-behavioral-patterns#LVI-G

Coder, tester et comprendre l'exemple.

Quelles différences voyez-vous avec le DP précédent ?

Tester l'exemple de Vehicule

Faites fonctionner l'exemple.

Pour comprendre son fonctionnement :

- Faites le diagramme de classe avec PowerAMC en utilisant le reverse ingeenering
- Faites le diagramme de séquence du main.

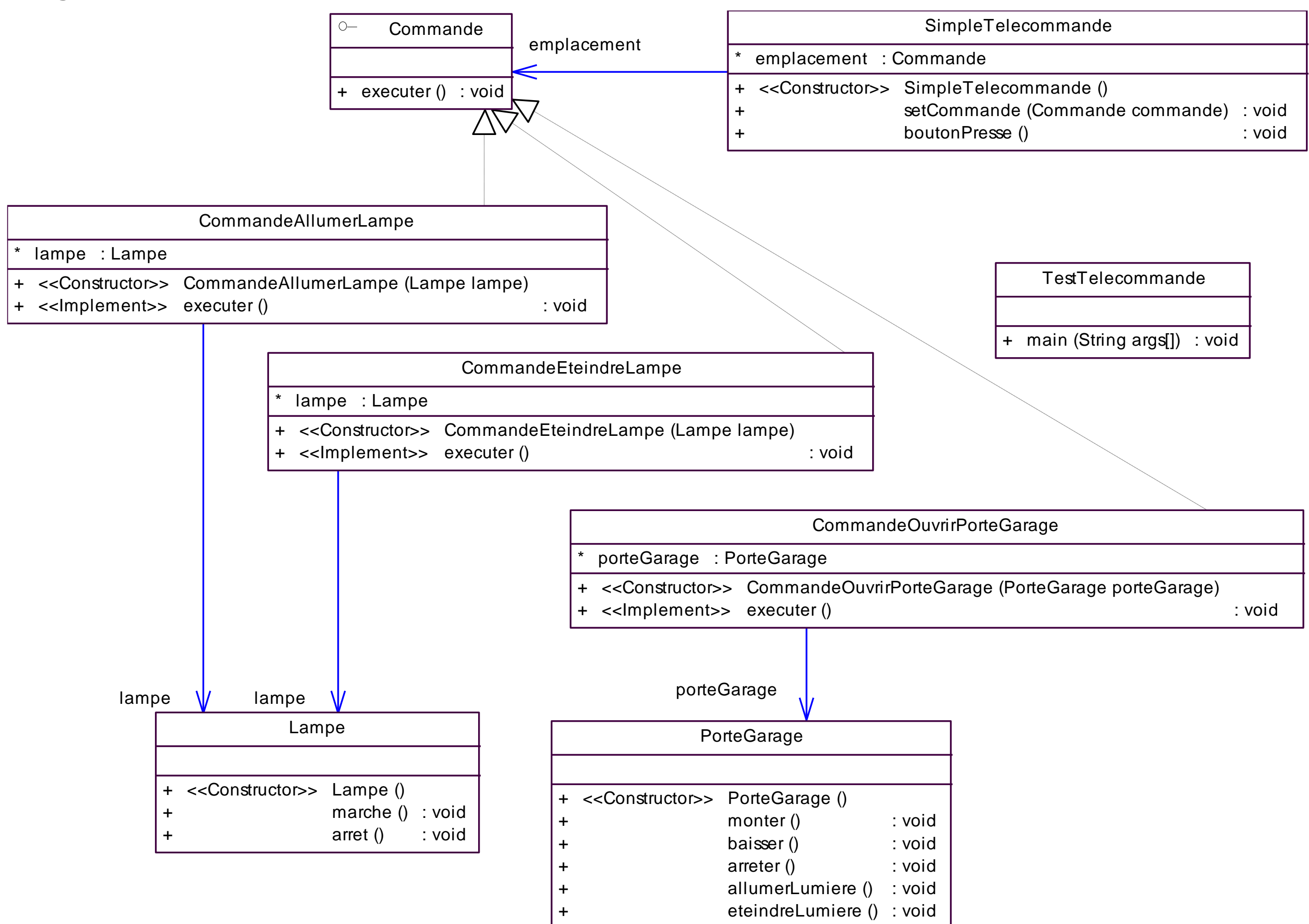
Le pattern Command (délégation)

Objectif du pattern

- **Le pattern Command** encapsule une requête comme un attribut permettant ainsi de faire fonctionner n'importe quel objet tout en en étant découplé.
- Le pattern Command et une fusion du pattern Stratégie (« Sette » des méthodes à travers une interface) et du pattern Adapteur (ajouter une nouvelle classe ou un nouveau composant à un modèle en faisant en sorte que l'interface du modèle ne change pas et donc que le client puisse continuer à fonctionner identiquement).

UML

Diagramme de classe



Le principe est que les objets à commander existent déjà avec leurs méthodes publiques (la lampe et la porte de garage). On veut les manipuler via la télécommande. On crée donc une interface de commande avec la méthode « exécuter » et des classes de commandes spécifiques. Si un nouvel objet arrive (un ventilateur) on ajoutera les classes de commandes spécifiques du ventilateur. La télécommande ne change pas. Et quand on fabrique une télécommande concrète, on lui donne les commandes qu'on veut.

Code du main

```
public class TestTelecommande {
  public static void main(String[] args) {
     //On crée d'abord la télécommande
     SimpleTelecommande telecommande =
        new SimpleTelecommande();

     // On crée une lampe et une commande pour la lampe
     Lampe lampe = new Lampe();
     Commande commande = new CommandeAllumerLampe(lampe);

     // On associe la commande à la telecommande
     telecommande.setCommande(commande);
     // On fait marcher la telecommande
     telecommande.boutonPresse();

     // On crée une porte de garage et une commande
     // pour la porte de garage
     PorteGarage porteGarage = new PorteGarage();
     Commande commande2 =
        new CommandeOuvrirPorteGarage(porteGarage);

     // On associe la commande à la telecommande
     telecommande.setCommande(commande2);
     // On fait marcher la telecommande
     telecommande.boutonPresse();

     // On peut aussi écrire :
     telecommande.setCommande(
        new CommandeAllumerLampe( new Lampe() ) );
     telecommande.boutonPresse();

     // L'objet télécommande est ainsi totalement découplé
     // des objets télécommandés
    }
}
```

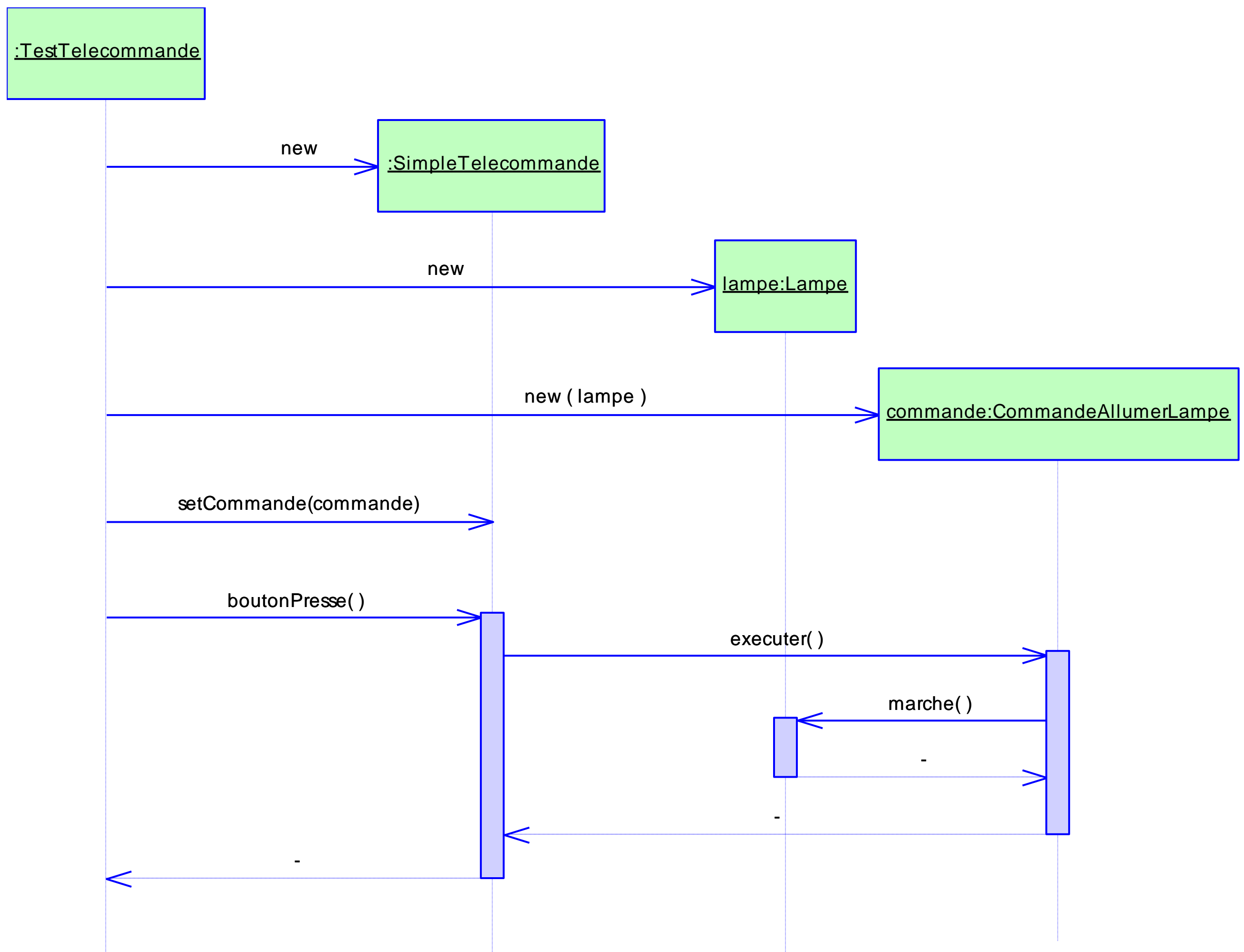
C:\ >java TestTelecommande

La lampe est allumee

La porte du garage est ouverte

La lampe est allumee

Diagramme de séquence



Le diagramme de séquence montre :

- La création de la télécommande : jusqu'au setCommande. On crée la télécommande, puis les objets à manipuler, puis les commandes qui permettent de manipuler les objets. Ces commandes sont settées à la télécommande.
- L'utilisation de la télécommande : boutonPresse() : cette utilisation est générique : selon la méthode associée au bouton, on aura tel ou tel comportement. Comme dans le DP Stratégie, on sette une méthode à la télécommande via un objet. Mais la télécommande accède à l'objet manipulé via une interface (l'interface Commande) qui définit la méthode exécuter, méthode qui est ensuite implémentée spécifiquement selon la commande et l'objet concerné.

Exercices : reprise de DP TLP

1) On doit programmer une télécommande qui prend en charge 10 boutons ON et OFF.

Les classes d'objets télécommandés sont fournies. Dans un premier temps, on n'utilise pas la classe ventilateur. La classe télécommande est fournie.

Vous devez coder le main avec tous les cas ON et OFF possible et le pattern commande. La classe CommandeAllumerLampe est donnée en exemple.

On précise que pour le Jackusi, OFF veut dire : refroidir(); puis eteindre(); et ON veut dire : allumer(); puis chauffer(); puis bouillonner();

Pour la Stereo, ON veut dire : marche(); setCD(); setVolume(11);

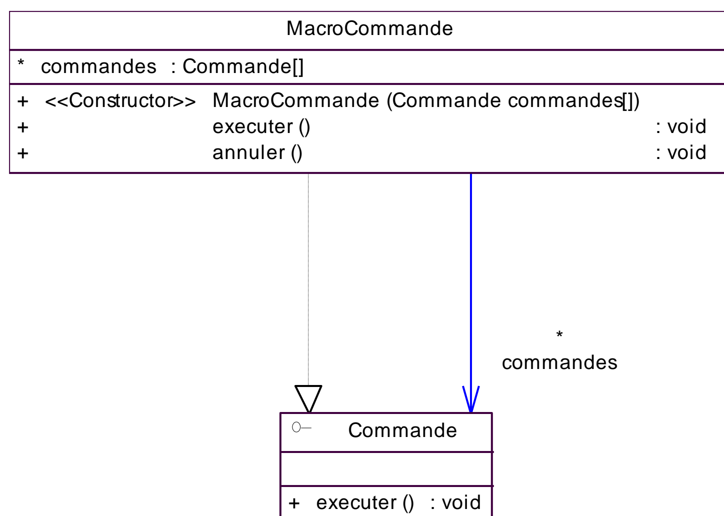
Pour le garage, ON veut dire monter() et OFF baisser.

2) vous devez gérer le ventilateur qui peut être allumé en vitesse lente, moyenne ou rapide. Le principe de fonctionnement est qu'on utilise un bouton ON pour chaque vitesse et que le bouton OFF associé éteindra dans les 3 cas.

La classe ventilateur est fournie.

3) Vous devez gérer l'annulation des commandes. L'annulation devra permettre de revenir à l'état antérieur pour toutes les commandes. La classe CommandeAllumerLampe est donnée en exemple.

4) vous devez gérer une macro commande qui permet de tout allumer et de tout éteindre. Son principe est donné dans le diagramme de classe ci-dessous.



Coder l'exemple de développer.com

http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/?page=page_4#LVI-B

Coder, tester et comprendre l'exemple.

Faites le diagramme de classes.

Faites de diagramme de séquence du main pour une commande.

Tester l'exemple de Commande.zip

Faites le diagramme de classes.

Faites de diagramme de séquence du main pour une commande.

Le pattern State (délégation)

Objectif du pattern

Le pattern [State](#) adapte le comportement d'un objet en fonction de son état interne.

Principes de résolution

Classiquement (en programmation procédurale), on teste l'état de l'objet et on applique les méthodes nécessaires en fonction de l'état.

Le pattern State consiste à transformer chaque état en classe, chacune de ces classes dérivant d'une classe Etat qui contiendra les méthodes des comportements liés à l'état.

La classe avec un état contient un objet de la classe Etat. Cet objet est modifié par les méthodes des classes correspondant à chaque état.

UML

Sans le pattern

On a la gestion d'un état. Cf DP TLP

Distributeur			
*	EPUISE	: int	= 0
*	SANS_PIECE	: int	= 1
*	A_PIECE	: int	= 2
*	VENDU	: int	= 3
*	etat	: int	= EPUISE
*	compte	: int	= 0
+	<<Constructor>> Distributeur (int nombre)		
+	insererPiece ()	: void	
+	ejecterPiece ()	: void	
+	tournerPoignee ()	: void	
+	delivrer ()	: void	
+	remplir (int numGumBalls)	: void	
+	toString ()	: String	

Une seule classe permet de gérer les différents états et les différentes transitions. C'est simple mais ce sera difficile à maintenir en cas d'évolution.

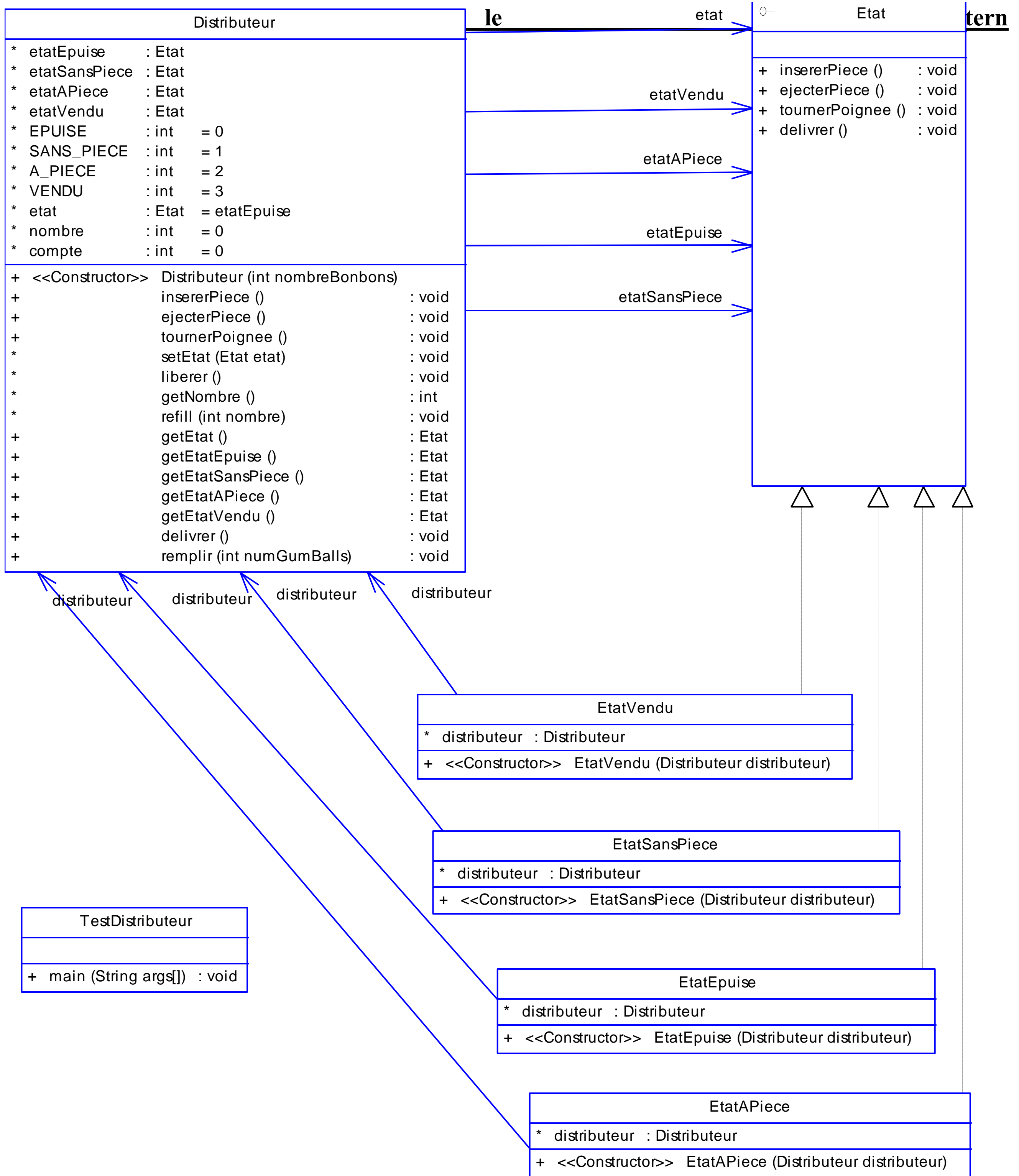
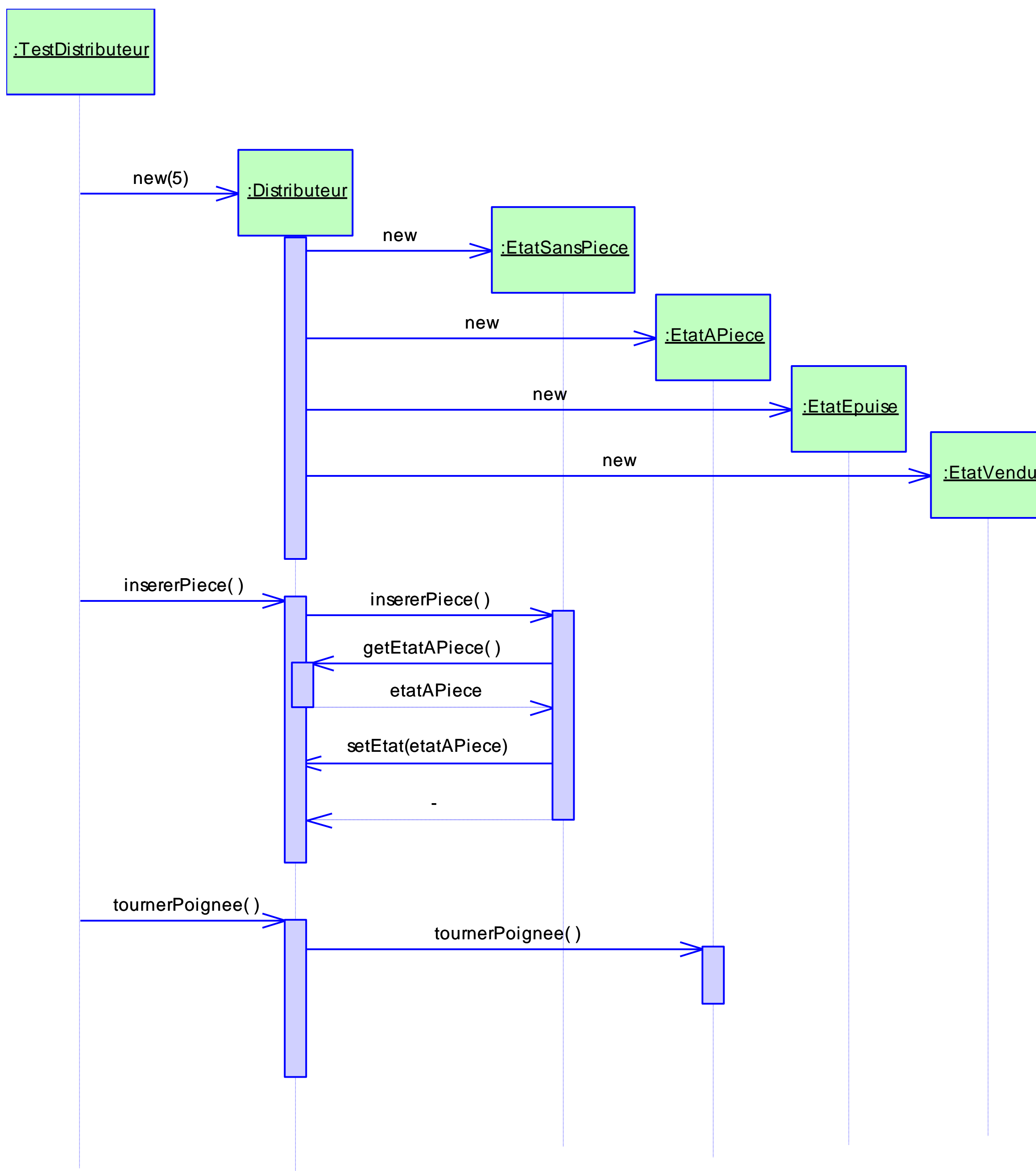

Distributeur
* etatEpuise : Etat
* etatSansPiece : Etat
* etatAPiece : Etat
* etatVendu : Etat
* EPUISE : int = 0
* SANS_PIECE : int = 1
* A_PIECE : int = 2
* VENDU : int = 3
* etat : Etat = etatEpuise
* nombre : int = 0
* compte : int = 0
+ <<Constructor>> Distributeur (int nombreBonbons)
+ insererPiece () : void
+ ejecterPiece () : void
+ tournerPoignee () : void
* setEtat (Etat etat) : void
* liberer () : void
* getNombre () : int
* refill (int nombre) : void
+ getEtat () : Etat
+ getEtatEpuise () : Etat
+ getEtatSansPiece () : Etat
+ getEtatAPiece () : Etat
+ getEtatVendu () : Etat
+ delivrer () : void
+ remplir (int numGumBalls) : void
etat
etatVendu
etatAPiece
etatEpuise
etatSansPiece
Etat
+ insererPiece () : void
+ ejecterPiece () : void
+ tournerPoignee () : void
+ delivrer () : void
distributeur
distributeur
distributeur
distributeur
EtatVendu
* distributeur : Distributeur
+ <<Constructor>> EtatVendu (Distributeur distributeur)
EtatSansPiece
* distributeur : Distributeur
+ <<Constructor>> EtatSansPiece (Distributeur distributeur)
TestDistributeur
+ main (String args[]) : void
EtatEpuise
* distributeur : Distributeur
+ <<Constructor>> EtatEpuise (Distributeur distributeur)
EtatAPiece
* distributeur : Distributeur
+ <<Constructor>> EtatAPiece (Distributeur distributeur)

Diagramme de séquence



Exercices

Tester l'exemple de DP TLP

Coder l'exemple de développer.com

https://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/?page=comportementaux-behavioral-patterns#LVI-H

Coder, tester et comprendre l'exemple.

Faire un autre main avec une variante.

Tester l'exemple de State.zip

Faites le diagramme de classes.

Le pattern Template method = Patron de méthode (héritage)

Objectif du pattern

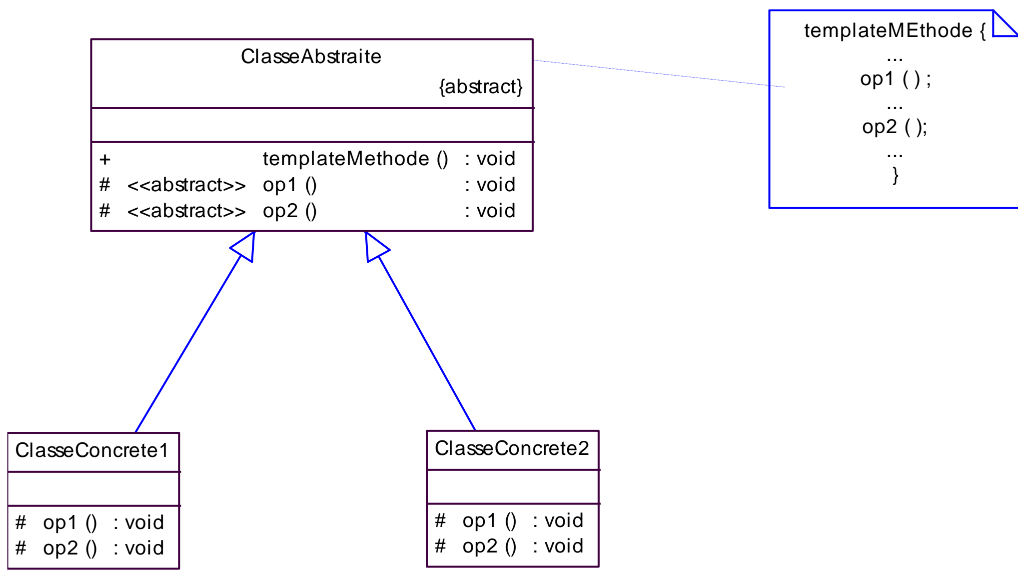
Le patterd Method définit une méthode en déléguant une partie de l'algorithme dans des classes dérivées.

La méthode est donc un « patron » concrétisé grâce aux classes dérivées.

Raisons de l'utiliser

Si une méthode à un fonctionnement spécifique et fonction des classes dérivées de sa propre classe.

UML



La méthode « templateMethode » met utilise les méthodes « op1 » et « op2 » qui abstraites au niveau de la ClasseAbstraite et concrétisées dans les ClasseConcrete 1 et 2.

Exemple d'utilisation

L'interface Comparable du JSE permet d'utiliser la méthode « sort » de la classe Array et des classes Collection. La méthode « sort », si elle reçoit en paramètre un tableau d'Oject implémentant l'interface comparable, utilise la méthode « compareTo ». La méthode « compareTo » est définie dans une classe réalisant la classe Comparable. C'est le principe du design pattern « Patron de méthode » (« Template method »).

Exercice

Thé ou café – DP TLP

Vous avez un code vous permettant de choisir entre du thé ou du café.

Adaptez ce code pour mettre en œuvre le pattern patron de méthode.

Analysez l'exemple ENI

Testez-le

Faites le diagramme des classes

Faites le diagramme de séquence du main

Le pattern Iterator (délégation)

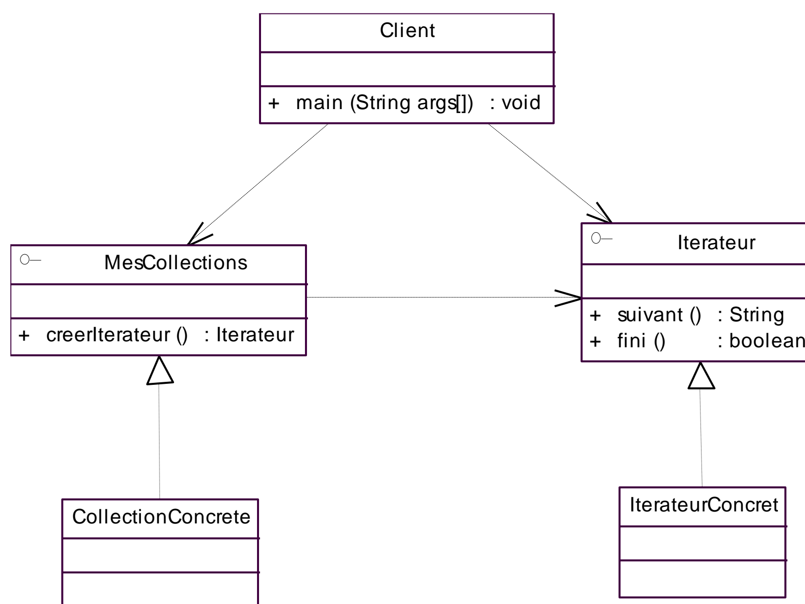
Objectif du pattern

Le pattern **Iterator** fournir un accès séquentiel aux éléments d'une collection sans se soucier de l'implantation concrète de cette collection.

Raisons de l'utiliser

Si le client doit parcourir les éléments d'une collection et que la classe de la collection peut changer.

Diagramme de classes



Principes techniques

Les classes d'une collection, quelle qu'elle soit, vont réaliser l'interface MesCollections.

Cette interface définit la méthode « créerItérateur » qui permet de récupérer un itérateur de collection, quelle qu'elle soit.

Un « itérateur Concret » réalise l'interface « Iterateur » qui définit la méthode « suivant » qui permet de récupérer un élément de la CollectionConcrete.

L'interface « Iterateur » définit aussi la méthode « fini » qui permet de savoir si l'itérateur concret est arrivé au bout de la collection.

Exercices

Coder l’exemple de développer.com

https://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/?page=comportementaux-behavioral-patterns#LVI-D

Coder, tester et comprendre l’exemple.

Cet exemple travaille avec une collection de chaînes de caractères rangées dans un tableau.

Faire un autre main avec un autre exemple de collection de chaînes initiale et de collection de chaînes concrètes.

Tester l’exemple de Iterator.zip

Faites le diagramme de classes. Attention, la génération automatique ne fonctionne que pour les classes Véhicule et Elément. Du fait des génériques, ça ne fonctionne pas pour les autres classes.

Dans le logiciel PowerAMC, on représentera les classe génériques avec un stéréotype : Generic : nomDuType.

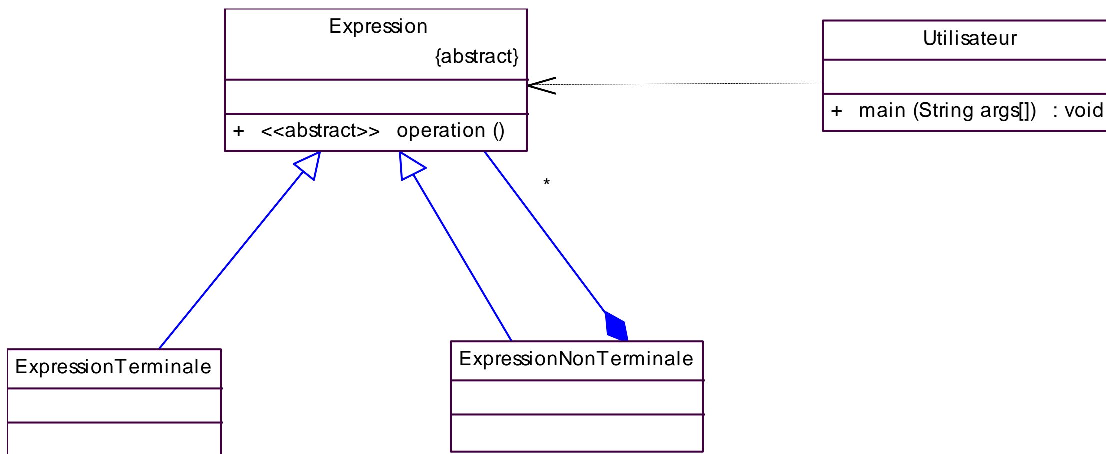
Le pattern Interpreter (héritage)

Objectifs

Le pattern Interpreter permet d'interpréter des expressions écrites dans un langage décrit par une grammaire constituée d'éléments terminaux et d'éléments non terminaux.

Il reprend la structure du pattern composite puisqu'il s'agit d'une structure d'arbre.

Diagramme de classes



Dans un langage, une expression est constituée soit d'une expression terminale (un mot clé, une variable, une constante, etc.), soit d'une expression non terminale.

Une expression non terminale est constituée de plusieurs expressions

On a une structure arborescente qui permet de décrire le langage.

La méthode opération est abstraite et définie dans les classes dérivées.

L'opération d'une expression non terminale va faire appel aux opérations des expressions qui la compose jusqu'à arriver à une expression terminale.

L'opération d'une expression terminale est une action simple.

Exercices

Coder l'exemple de développer.com

http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/?page=page_4#LVI-C

Coder, tester et comprendre l'exemple.

Faire un autre main avec un autre exemple XML ou HTML.

Le pattern Chain of responsability (délégation)

Objectifs

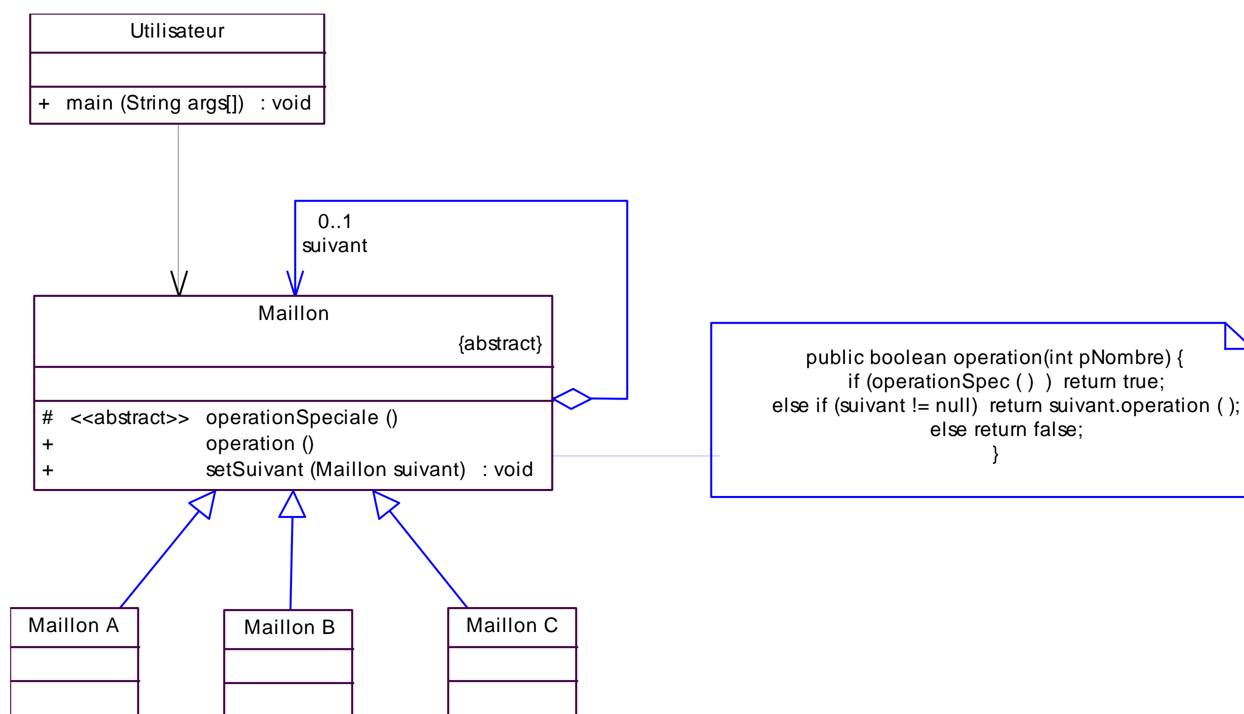
Le patten **Chain of responsability** permet de passer la requête tout le long d'une chaîne d'objets jusqu'à ce qu'un objet la traite.

Principe de réalisation

Le principe consiste à chaîner les objets par une association et à distinguer les opérations des objets en les réalisant dans des classes dérivées.

On retrouve une partie de la structure du design pattern Template method.

Diagramme de classes

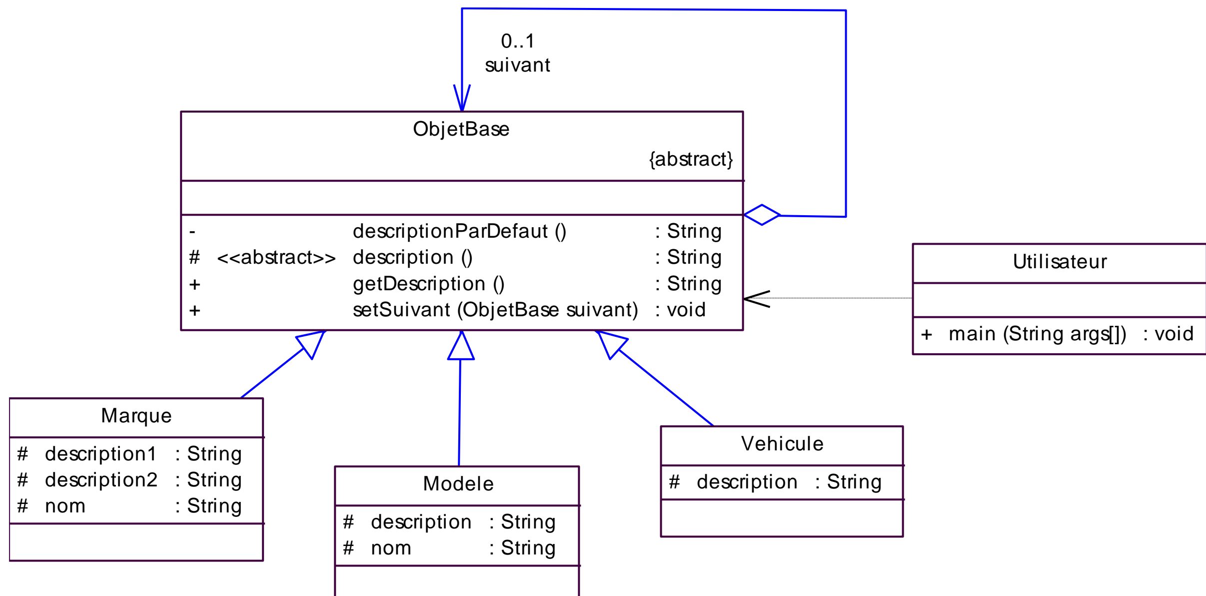


On voit que dans le code de la méthode « operation », on fait appel à l'opération spéciale du maillon concret en jeu. Si cette méthode renvoie faux, alors on passe au maillon suivant jusqu'au dernier. A la fin, si aucun maillon concret ne possède une méthode qui renvoie vrai, alors on renvoie faux.

Exercices

Compléter le code du modèle

Le modèle suivant correspond au code que vous pouvez charger dans ChainOfResponsability.zip.



Toutefois, les méthodes « description() » et « getDescription() » de la classe « objetBase » sont restent à coder.

Vous devez les coder pour obtenir le résultat suivant :

```
Auto++ KT500 Véhicule d'occasion en bon état
Modèle KT400 : Le véhicule spacieux et confortable
Marque Auto++ : La marque des autos de grande qualité
description par défaut
```

Modifier le modèle

L'attribut « description » est considéré comme faisant partie de l'objetBase. Mettre à jour le code en conséquence.

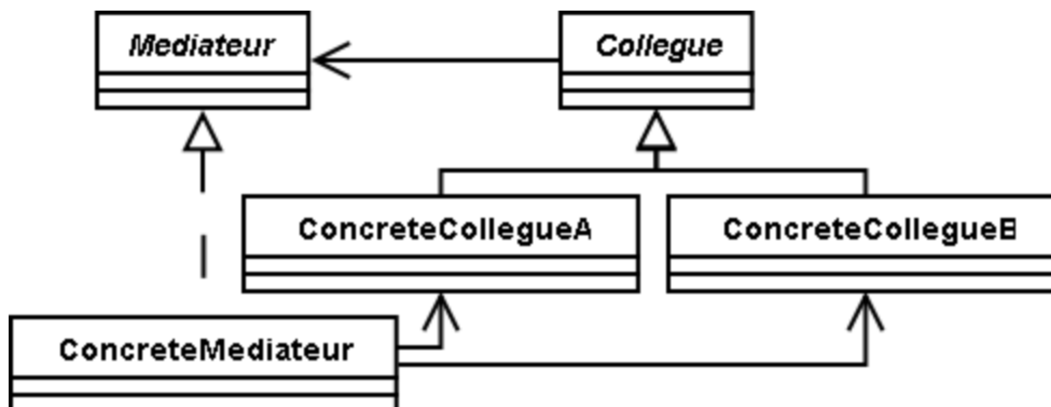
Le pattern Mediator (délégation)

Objectif du pattern

Le pattern mediator permet de créer un objet « médiateur » dont l'objectif sera de gérer les interactions entre des objets de la même classe de base mais qui ne se connaissent pas.

L'intérêt principal est de découpler les objets concernés.

UML



Exercices

Coder l'exemple de développer.com

https://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/?page=comportementaux-behavioral-patterns#LVI-E

Coder, tester et comprendre l'exemple.

Faire un autre main avec une variante.

Tester l'exemple de Mediator.zip

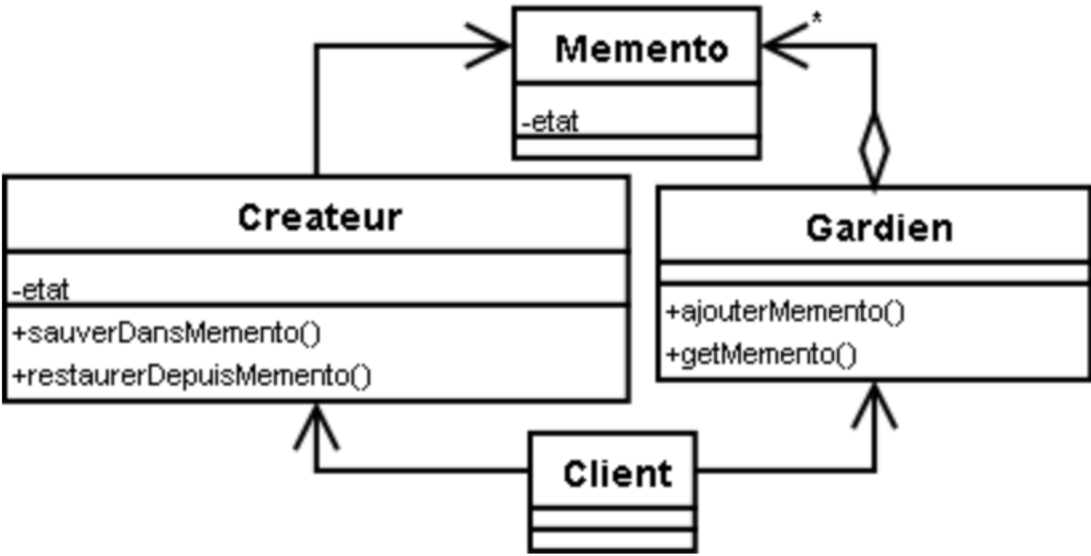
Faites le diagramme de classes.

Le pattern Memento (délégation)

Objectif du pattern

Le pattern [**memento**]() permet de sauvegarder l'état interne d'un objet en respectant l'encapsulation, afin de le restaurer plus tard.

UML



Exercices

Coder l'exemple de développer.com

https://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/?page=comportementaux-behavioral-patterns#LVI-F

Coder, tester et comprendre l'exemple.

Faire un autre main avec une variante.

Tester l'exemple de Memento.zip

Faites le diagramme de classes.

Le pattern Visitor (délégation)

Objectif du pattern

Le pattern Visitor permet, dans une collection d'objet, de séparer les méthodes spécifiques des méthodes de la structure de donnée.

Principes

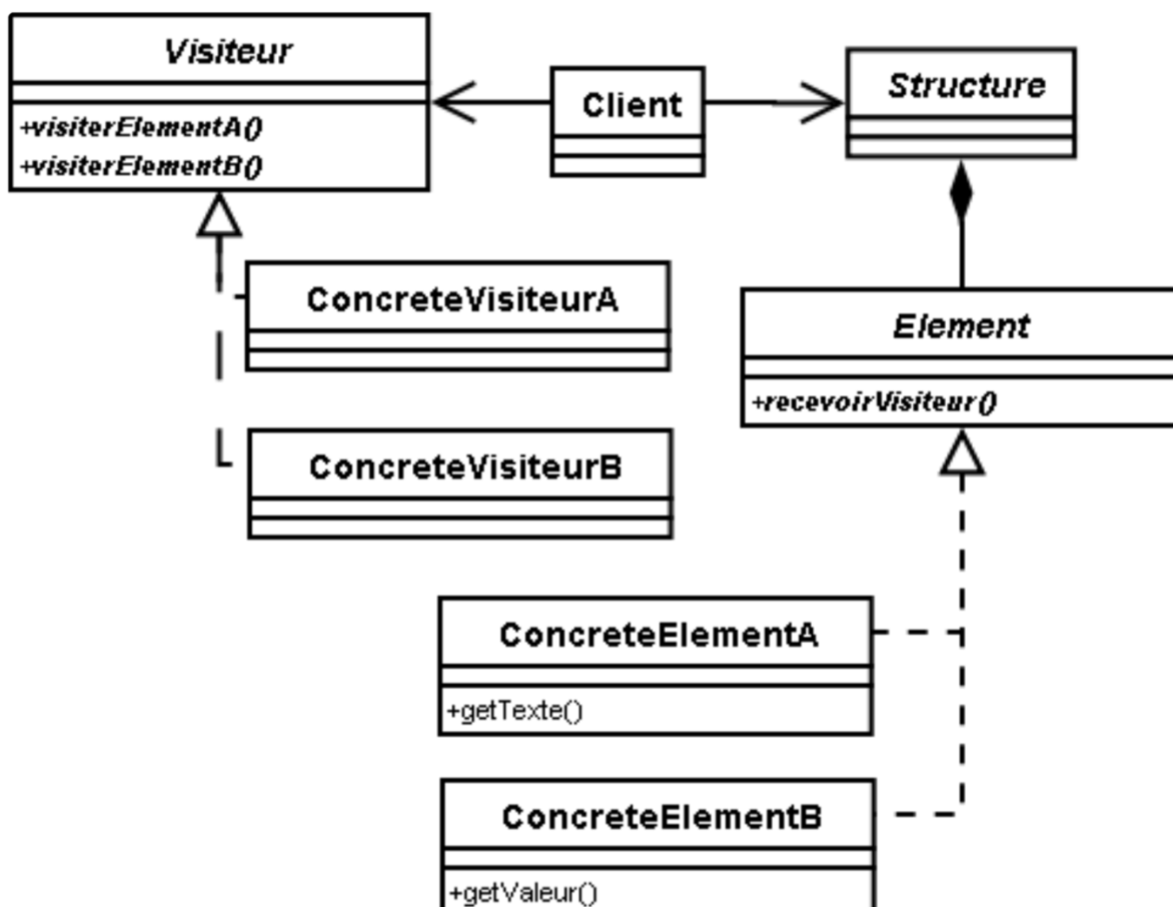
On ajoute une méthode « recevoirVisiteur » dans les objets de la collection. Cette méthode reçoit en paramètre un objet de l'interface Visiteur (comme dans le pattern Strategy).

L'interface Visiteur a autant de méthodes visiterElément qu'il y a de classes distinctes dans la collection.

Chaque méthode visiterElement reçoit en paramètre un objet de la collection ce qui lui permet d'accèder aux méthodes de cet objet et donc de « visiter » cet objet.

Ca permet aussi de rajouter des méthodes spécifiques des objets de la collection dans le visiteur correspondant et d'y accéder par la méthode « visiterElement ».

UML



Exercices

Coder l'exemple de développer.com

https://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/?page=comportementaux-behavioral-patterns#LVI-K

Coder, tester et comprendre l'exemple.

Faire un autre main avec une variante.

Tester l'exemple de Visitor.zip

Faites le diagramme de classes.