Rapport Sur



Réalisé par : Encadré par :

ACHAHBAR Farah

Mr. Yasser El Madani El Alami

ALLAY Leyla

El FKIH BEN ahmed Khaoula

MAYA Loubna

ROUBYOU Mounia

Design Patterns - Modèle d'état

Dans le modèle d'état, un comportement de classe change en fonction de son état. Ce type de modèle de conception est soumis à un modèle de comportement.

Dans le modèle d'état, nous créons des objets qui représentent différents états et un objet de contexte dont le comportement varie à mesure que son objet d'état change.

**Problème**

– ce motif est à utiliser lorsque l’on veut qu’un objet change de comportement lorsque son état interne change

**Conséquences :**

+ possibilité d’ajouter ou de retirer des états et des transitions de manière simple

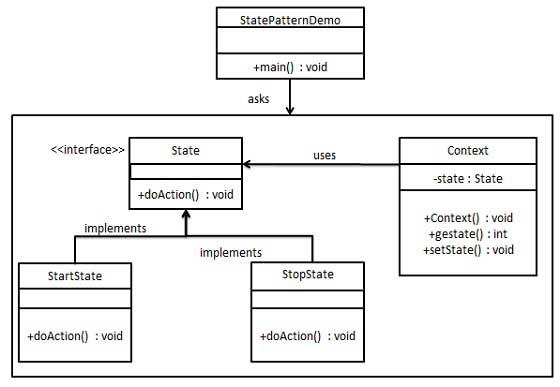
+ suppression de traitements conditionnels

+ les transitions entre états sont rendues explicites

## la mise en œuvre :

Nous allons créer une interface d'*état* définissant une action et des classes d'états concrètes implémentant l'interface d'*état*. *Le contexte* est une classe qui porte un État.

*StatePatternDemo* , notre classe de démonstration, utilisera les objets *Context* et state pour démontrer les changements de comportement du contexte en fonction du type d'état dans lequel il se trouve.



# Design Patterns – Modèle template

Dans le modèle Template, une classe abstraite expose des méthodes ou des modèles définis pour exécuter ses méthodes. Ses sous-classes peuvent remplacer l'implémentation de la méthode selon le besoin mais l'invocation doit être de la même manière que celle définie par une classe abstraite. Ce modèle appartient à la catégorie de modèle de comportement.

## la mise en oeuvre

Nous allons créer une classe abstraite de *jeu* définissant les opérations avec une méthode de modèle définie comme étant finale afin de ne pas pouvoir être surchargée. *Le cricket* et le *football* sont des classes concrètes qui prolongent le *jeu* et remplacent ses méthodes.

*TemplatePatternDemo* , notre classe de démonstration, utilisera *Game* pour démontrer l'utilisation du modèle de modèle.



Design Patterns – Modèle Chaine de Responsabilité

Comme son nom l'indique, le modèle de chaîne de responsabilité crée une chaîne d'objets récepteurs pour une requête. Ce motif découble l'expéditeur et le destinataire d'une requête en fonction du type de demande. Ce modèle relève de modèles comportementaux.

Dans ce modèle, normalement chaque récepteur contient une référence à un autre récepteur. Si un objet ne peut pas gérer la requête, il passe la même chose au destinataire suivant et ainsi de suite.

## la mise en oeuvre

Nous avons créé une classe abstraite *AbstractLogger* avec un niveau de journalisation. Ensuite, nous avons créé trois types d'enregistreurs qui étendent *AbstractLogger* . Chaque enregistreur vérifie le niveau du message à son niveau et imprime en conséquence autrement ne pas imprimer et passer le message à son prochain enregistreur.

# 3.jpg

# Design Patterns - Filter Pattern

Le modèle de filtre ou le modèle de critère est un modèle de conception qui permet aux développeurs de filtrer un ensemble d'objets en utilisant différents critères et en les enchaînant de manière découplée à travers des opérations logiques. Ce type de modèle de conception est structuré car ce modèle combine plusieurs critères pour obtenir des critères uniques.

## la mise en oeuvre

Nous allons créer un objet *Person* , une interface *Criteria* et des classes concrètes implémentant cette interface pour filtrer la liste des objets *Person* . *CriteriaPatternDemo* , notre classe de démonstration utilise des objets *Criteria* pour filtrer les objets List of *Person en* fonction de différents critères et de leurs combinaisons.

# 4.jpg

# Design Patterns - BuilderPattern

Le modèle Builder construit un objet complexe en utilisant des objets simples et en utilisant une approche pas à pas. Ce type de modèle de conception est sous un modèle de création car ce modèle fournit l'une des meilleures façons de créer un objet.

## la mise en oeuvre

Une classe Builder génère l'objet final étape par étape. Ce générateur est indépendant des autres objets.

Nous allons créer une interface *Item* représentant des produits alimentaires tels que des hamburgers et des boissons froides et des classes de béton mettant en œuvre l' interface *Item* et une interface *Packing* représentant des emballages de produits alimentaires et des classes de béton implémentant l' interface *Packing.* serait emballé comme une bouteille.

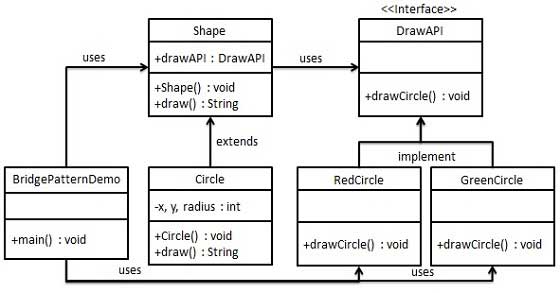
Nous créons ensuite une classe *Meal* ayant *ArrayList* of *Item* et un *MealBuilder*pour construire différents types d' objets *Meal* en combinant *Item* . *BuilderPatternDemo* , notre classe de démonstration utilisera *MealBuilder* pour construire un *repas* .

# Design Patterns - Bridge Pattern

Bridge est utilisé lorsque nous avons besoin de découpler une abstraction de son implémentation afin que les deux puissent varier indépendamment. Ce type de modèle de conception est structuré car ce modèle dissocie la classe d'implémentation et la classe abstraite en fournissant une structure de pont entre eux.

## la mise en oeuvre

Nous avons une interface *DrawAPI* qui agit comme un implémenteur de bridge et des classes concrètes *RedCircle* , *GreenCircle* implémentant l' interface *DrawAPI* . *Shape* est une classe abstraite et utilisera l'objet de *DrawAPI* . *BridgePatternDemo* , notre classe de démonstration utilisera la classe *Shape*pour dessiner un cercle de couleur différente.

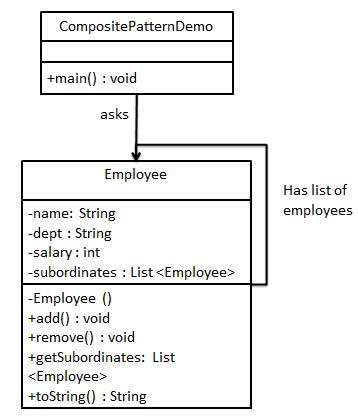


# Design Patterns - Composite Pattern

Le modèle composite est utilisé lorsque nous devons traiter un groupe d'objets de la même manière qu'un seul objet. Le modèle composite compose des objets en terme de structure arborescente pour représenter une partie ainsi qu'une hiérarchie entière. Ce type de modèle de conception est structuré car ce modèle crée une arborescence de groupe d'objets.

## *la mise en oeuvre*

Nous avons un *employé de* classe qui agit en tant que classe d'acteur de modèle composite. *CompositePatternDemo* , notre classe de démonstration utilisera la classe *Employee* pour ajouter une hiérarchie au niveau du département et imprimer tous les employés.



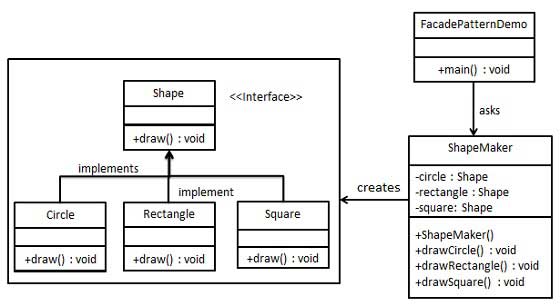
# Design Patterns - Facade Pattern

Le modèle de façade masque les complexités du système et fournit une interface au client grâce à laquelle le client peut accéder au système. Ce type de modèle de conception est structuré car ce modèle ajoute une interface au système existant pour en cacher les complexités.

## *la mise en oeuvre*

Nous allons créer une interface *Shape* et des classes concrètes implémentant l' interface *Shape* . Une classe de façade *ShapeMaker* est définie comme une étape suivante.

*La* classe *ShapeMaker* utilise les classes concrètes pour déléguer des appels d'utilisateur à ces classes. *FacadePatternDemo* , notre classe de démonstration, utilisera la classe *ShapeMaker* pour montrer les résultats.



# Design Patterns – Decorator Pattern

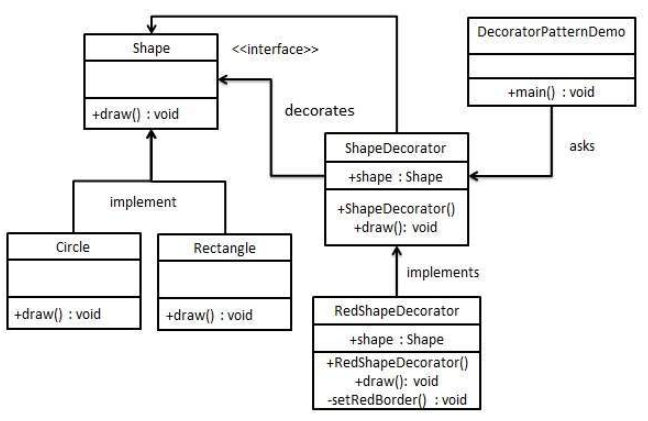
Le motif de conception décorateur (decorator en anglais) permet d'ajouter des fonctionnalités à un objet en mettant en oeuvre une solution plus souple que l'héritage : il permet d'ajouter des fonctionnalités à une ou plusieurs méthodes existantes d'une classe dynamiquement.

Nous démontrons l'utilisation du modèle décorateur par l'exemple suivant dans lequel nous allons décorer une forme avec une certaine couleur sans altérer la classe de forme.

## *la mise en oeuvre*

Nous allons créer une interface Shape et des classes concrètes implémentant l'interface Shape. Nous allons ensuite créer une classe décorative abstraite ShapeDecorator implémentant l'interface Shape et ayant l'objet Shape comme variable d'instance.

RedShapeDecorator est une classe concrète implémentant ShapeDecorator.



# Design Patterns – Adapter Pattern

Le modèle d'adaptateur fonctionne comme un pont entre deux interfaces incompatibles. Ce type de motif de conception se joint à un motif structurel car ce motif combine la capacité de deux interfaces indépendantes.

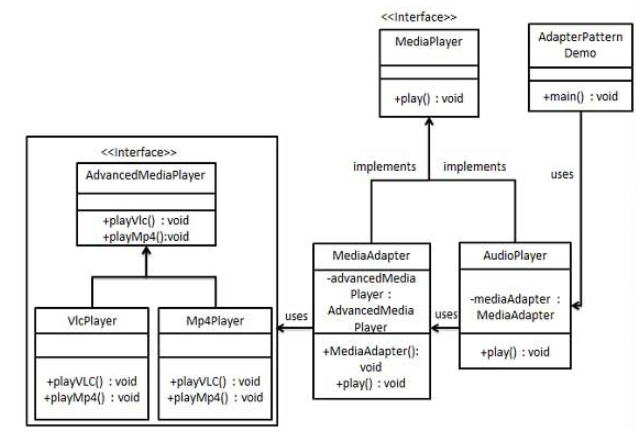
## *la mise en oeuvre*

Nous avons une interface MediaPlayer et une classe AudioPlayer concrète implémentant l'interface MediaPlayer. AudioPlayer peut lire les fichiers audio au format mp3 par défaut.

Nous avons une autre interface AdvancedMediaPlayer et des classes concrètes implémentant l'interface AdvancedMediaPlayer. Ces classes peuvent lire les fichiers au format vlc et mp4.

Nous voulons également faire jouer AudioPlayer à d'autres formats. Pour ce faire, nous avons créé une classe d'adaptateur MediaAdapter qui implémente l'interface MediaPlayer et utilise des objets AdvancedMediaPlayer pour lire le format requis.

AudioPlayer utilise la classe d'adaptateur MediaAdapter en lui transmettant le type audio désiré sans connaître la classe réelle qui peut lire le format désiré. AdapterPatternDemo, notre classe de démonstration utilisera la classe AudioPlayer pour lire différents formats.

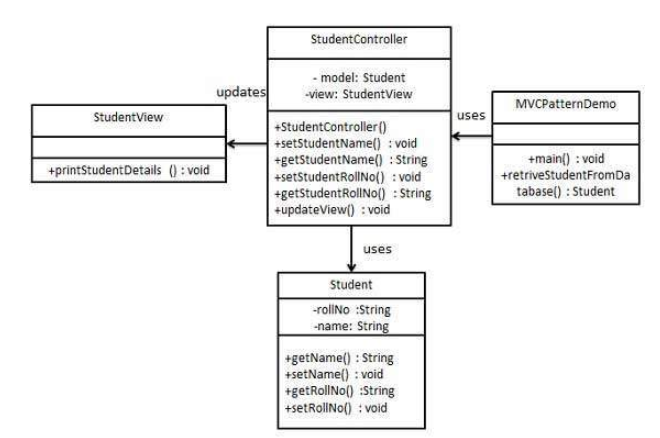
****

# Design Patterns – Decorator Pattern

**Modèle-vue-contrôleur** ou **MVC** est un motif d'[architecture logicielle](https://fr.wikipedia.org/wiki/Architecture_logicielle) destiné aux [interfaces graphiques](https://fr.wikipedia.org/wiki/Interface_graphique) lancé en 1978 et très populaire pour les [applications web](https://fr.wikipedia.org/wiki/Application_web). Le motif est composé de trois types de modules ayant trois responsabilités différentes : les modèles, les vues et les contrôleurs.

* Un modèle (Model) contient les données à afficher.
* Une vue (View) contient la présentation de l'interface graphique.
* Un contrôleur (Controller) contient la logique concernant les actions effectuées par l'utilisateur.

## *la mise en oeuvre*

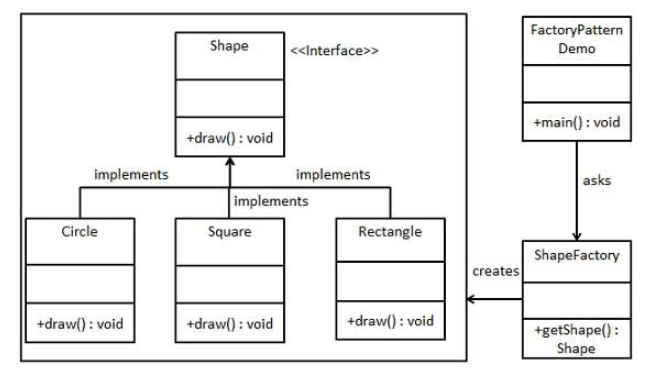
****Nous allons créer un objet Etudiant agissant comme un model.StudentView sera une classe de vue qui pourra imprimer les détails de l'étudiant sur la console et StudentController est la classe de contrôleur responsable de stocker les données dans l'objet Student et de mettre à jour la vue StudentView en conséquence.

# Design Patterns – Factory Pattern

La fabrique (factory method) est un patron de conception créationnel utilisé en programmation orientée objet. Elle permet d'instancier des objets dont le type est dérivé d'un type abstrait. La classe exacte de l'objet n'est donc pas connue par l'appelant.

## *la mise en oeuvre*

Nous allons créer une interface Shape et des classes concrètes implémentant l'interface Shape. Une classe d'usine ShapeFactory est définie comme une étape suivante. FactoryPatternDemo, notre classe de démonstration utilisera ShapeFactory pour obtenir un objet Shape. Il transmettra les informations (CIRCLE / RECTANGLE / SQUARE) à ShapeFactory pour obtenir le type d'objet dont il a besoin.



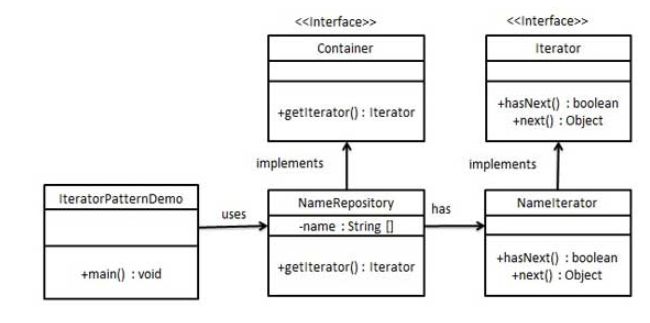
# Design Patterns – Iterator Pattern

Ce modèle est utilisé pour obtenir un moyen d'accéder aux éléments d'un objet de collection de manière séquentielle sans avoir besoin de connaître sa représentation sous-jacente.

## *la mise en oeuvre*

Nous allons créer une interface Iterator qui décrit la méthode de navigation et une interface Container qui retrace l'itérateur. Les classes concrètes implémentant l'interface Container seront responsables de l'implémentation de l'interface Iterator et de l'utiliser

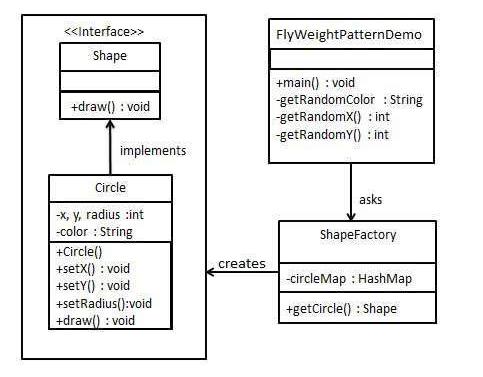
IteratorPatternDemo, notre classe de démonstration utilisera NamesRepository, une implémentation de classe concrète pour imprimer un Names stocké en tant que collection dans NamesRepository.

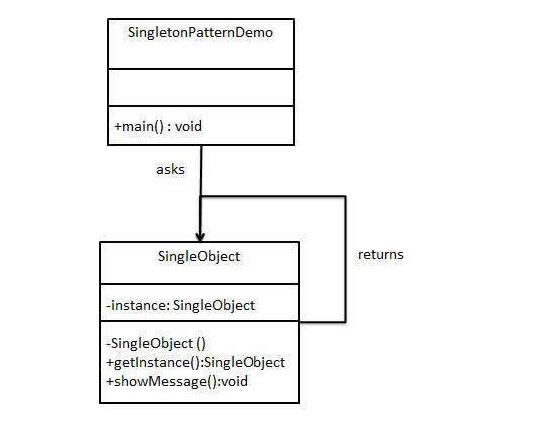


**FlyWeight Pattern**

Le modèle Flyweight est principalement utilisé pour réduire le nombre d'objets créés et pour réduire l'empreinte mémoire et augmenter les performances. Ce type de modèle de conception est structuré car ce modèle fournit des moyens de réduire le nombre d'objets, améliorant ainsi la structure d'objet de l'application.

**La mise en oeuvre**

Nous allons créer une interface Shape et une classe concrète Circle implémentant l'interface Shape. Une classe d'usine ShapeFactory est définie comme une étape suivante. ShapeFactory a une HashMap de Circle ayant la clé en couleur de l'objet Circle. Chaque fois qu'une requête vient créer un cercle de couleur particulière à ShapeFactory, il vérifie l'objet circle dans sa HashMap, si l'objet de Circle trouvé, cet objet est retourné sinon un nouvel objet est créé, stocké dans hashmap pour une utilisation future, et retourné à client. FlyWeightPatternDemo, notre classe de démonstration, utilisera ShapeFactory pour obtenir un objet Shape. Il transmettra les informations (rouge / vert / bleu / noir / blanc) à ShapeFactory pour obtenir le cercle de la couleur désiré



l’implementation java de ces classes se trouve dans le dossier qui accompagne ce document

**Proxy Pattern**

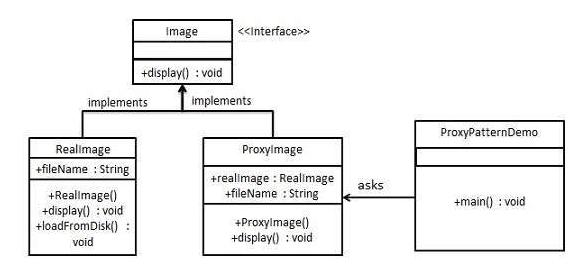
Dans un modèle de proxy, une classe représente la fonctionnalité d'une autre classe. Ce type de modèle de conception est soumis à un modèle structurel.

Dans le modèle de proxy, nous créons un objet ayant un objet original pour interfacer sa fonctionnalité avec le monde extérieur

**La mise en oeuvre**

Nous allons créer une interface Image et des classes concrètes implémentant l'interface Image. ProxyImage est une classe proxy pour réduire l'empreinte mémoire du chargement des objets RealImage.

ProxyPatternDemo, notre classe de démonstration, utilisera ProxyImage pour obtenir un objet Image à charger et afficher comme il le faut.



l’implementation java de ces classes se trouve dans le dssier qui accompagne ce document

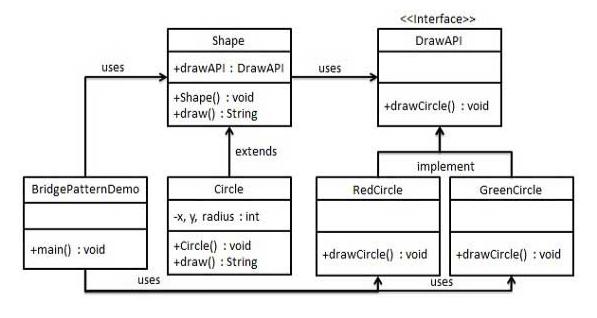
**Bridge Pattern**

Bridge est utilisé lorsque nous avons besoin de découpler une abstraction de son implémentation afin que les deux puissent varier indépendamment. Ce type de modèle de conception est structuré car ce modèle dissocie la classe d'implémentation et la classe abstraite en fournissant une structure de pont entre eux.

Ce modèle implique une interface qui agit comme un pont qui rend la fonctionnalité des classes concrètes indépendante des classes d'implémentation d'interface. Les deux types de classes peuvent être modifiés structurellement sans s'influencer mutuellement.

Nous démontrons l'utilisation du modèle de Bridge via l'exemple suivant dans lequel un cercle peut être dessiné dans différentes couleurs en utilisant la même méthode de classe abstraite mais différentes classes d'implémentation de ponts.

**La mise en oeuvre**

Nous avons une interface DrawAPI qui agit comme un implémenteur de bridge et des classes concrètes RedCircle, GreenCircle implémentant l'interface DrawAPI. Shape est une classe abstraite et utilisera l'objet de DrawAPI. BridgePatternDemo, notre classe de démonstration utilisera la classe Shape pour dessiner un cercle de couleur différente

l’implementation java de ces classes se trouve dans le dossier qui accompagne ce document

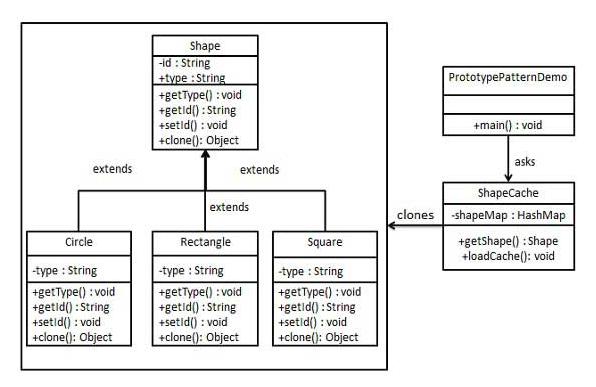
**Prototype Pattern**

Le modèle de prototype fait référence à la création d'objets dupliqués tout en gardant à l'esprit les performances. Ce type de modèle de conception est sous un modèle de création car ce modèle fournit l'une des meilleures façons de créer un objet.

Ce modèle implique l'implémentation d'une interface prototype qui dit de créer un clone de l'objet courant. Ce modèle est utilisé lorsque la création d'un objet est directement coûteuse. Par exemple, un objet doit être créé après une opération de base de données coûteuse. Nous pouvons mettre en cache l'objet, renvoyer son clone à la prochaine requête et mettre à jour la base de données au fur et à mesure des besoins, réduisant ainsi les appels à la base de données.

**La mise en oeuvre**

Nous allons créer une classe abstraite Shape et des classes concrètes qui étendent la classe Shape. Une classe ShapeCache est définie comme une étape suivante qui stocke des objets de forme dans une table de hachage et renvoie leur clone à la demande.

PrototypPatternDemo, notre classe de démonstration utilisera la classe ShapeCache pour obtenir un objet Shape.

l’implementation java de ces classes se trouve dans le dossier qui accompagne ce document

**Singleton Pattern**

Le modèle Singleton est l'un des modèles de conception les plus simples de Java. Ce type de modèle de conception est sous un modèle de création car ce modèle fournit l'une des meilleures façons de créer un objet.

Ce modèle implique une seule classe qui est responsable de créer un objet tout en s'assurant que seul objet unique est créé. Cette classe fournit un moyen d'accéder à son seul objet accessible directement sans avoir besoin d'instancier l'objet de la classe.

**La mise en oeuvre**

Nous allons créer une classe SingleObject. La classe SingleObject a son constructeur comme private et possède une instance statique de lui-même.

La classe SingleObject fournit une méthode statique pour placer son instance statique dans le monde extérieur. SingletonPatternDemo, notre classe de démonstration utilisera la classe SingleObject pour obtenir un objet SingleObject.

Strategy Pattern

Dans le modèle Stratégie, un comportement de classe ou son algorithme peut être modifié au moment de l'exécution. Ce type de modèle de conception est soumis à un modèle de comportement.

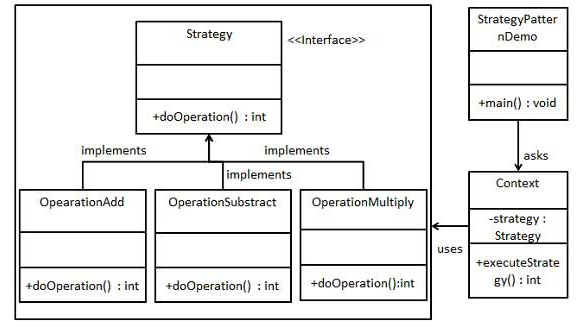
Dans le modèle Stratégie, nous créons des objets qui représentent différentes stratégies et un objet contexte dont le comportement varie selon son objet stratégique. L'objet de stratégie modifie l'algorithme d'exécution de l'objet de contexte.

## *Implementation*

Nous allons créer une interface Stratégie définissant une action et des classes de stratégies concrètes implémentant l'interface Stratégie. Le contexte est une classe qui utilise une stratégie.

StrategyPatternDemo, notre classe de démonstration, utilisera les objets Contexte et Stratégie pour démontrer les changements de comportement du Contexte en fonction de la stratégie déployée ou utilisée.

Nous allons créer une interface Stratégie définissant une action et des classes de stratégies concrètes implémentant l'interface Stratégie. Le contexte est une classe qui utilise une stratégie.



**Mediator Pattern**

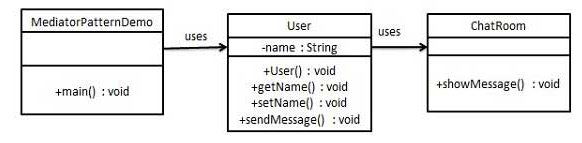
Le modèle Médiateur est utilisé pour réduire la complexité de la communication entre plusieurs objets ou classes. Ce modèle fournit une classe de médiateur qui gère normalement toutes les communications entre différentes classes et prend en charge la maintenance facile du code par un couplage lâche. Le modèle de médiateur relève de la catégorie de comportement.

## *Implementation*

Nous montrons un modèle médiateur par l'exemple d'une salle de discussion où plusieurs utilisateurs peuvent envoyer un message à la salle de discussion et il est de la responsabilité du salon de discussion de montrer les messages à tous les utilisateurs.

Nous avons créé deux classes ChatRoom et User. Les objets utilisateur utiliseront la méthode ChatRoom pour partager leurs messages.

MediatorPatternDemo, notre classe de démonstration, utilisera des objets Utilisateur pour montrer la communication entre eux.



**Memento Pattern**

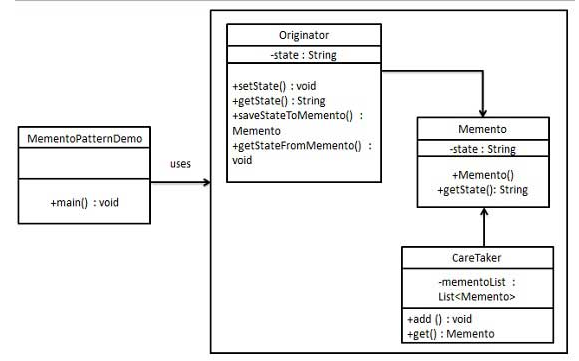
Le motif Memento est utilisé pour restaurer l'état d'un objet à un état antérieur. Le motif de souvenir tombe sous la catégorie de comportement.

## *Implementation*

Le motif Memento utilise trois classes d'acteurs. Memento contient l'état d'un objet à restaurer. L'auteur crée et stocke les états dans les objets Memento et l'objet Caretaker est chargé de restaurer l'état de l'objet à partir de Memento.

Nous avons créé des classes Memento, Originator et CareTaker.

MementoPatternDemo, notre classe de démonstration, utilisera les objets CareTaker et Originator pour montrer la restauration des états de l'objet.



**Visitor Pattern**

Dans le modèle Visitor, nous utilisons une classe de visiteurs qui modifie l'algorithme d'exécution d'une classe d'éléments. De cette façon, l'algorithme d'exécution de l'élément peut varier au fur et à mesure que le visiteur varie.

Ce modèle relève de la catégorie de modèle de comportement. Conformément au modèle, l'objet élément doit accepter l'objet visiteur afin que l'objet visiteur gère l'opération sur l'objet élément.

## *Implementation*

Nous allons créer une interface ComputerPart définissant l'acceptation de l'operation . Le clavier, la souris, le moniteur et l'ordinateur sont des classes concrètes implémentant l'interface ComputerPart.

Nous allons définir une autre interface ComputerPartVisitor qui va définir une opération de classe visiteur. L'ordinateur utilise le visiteur concret pour faire l'action correspondante.

VisitorPatternDemo, notre classe de démonstration, utilisera les classes Computer et ComputerPartVisitor pour démontrer l'utilisation du modèle de visiteur.

