EISTI
Ecole Internationale des Sciences du Traitement de l'Information
CERGY - PAU

Design Patterns – EISTI 2009
Romain N. (promo 2K11)

INTRODUCTION :

A l'origine les **Design Patterns** (**Patron de conception** en Français) sont issus des travaux de l'architecte Christopher Alexander. Ces travaux sont une capitalisation d'expérience qui ont mis en évidence des patrons en architecture des bâtiments.

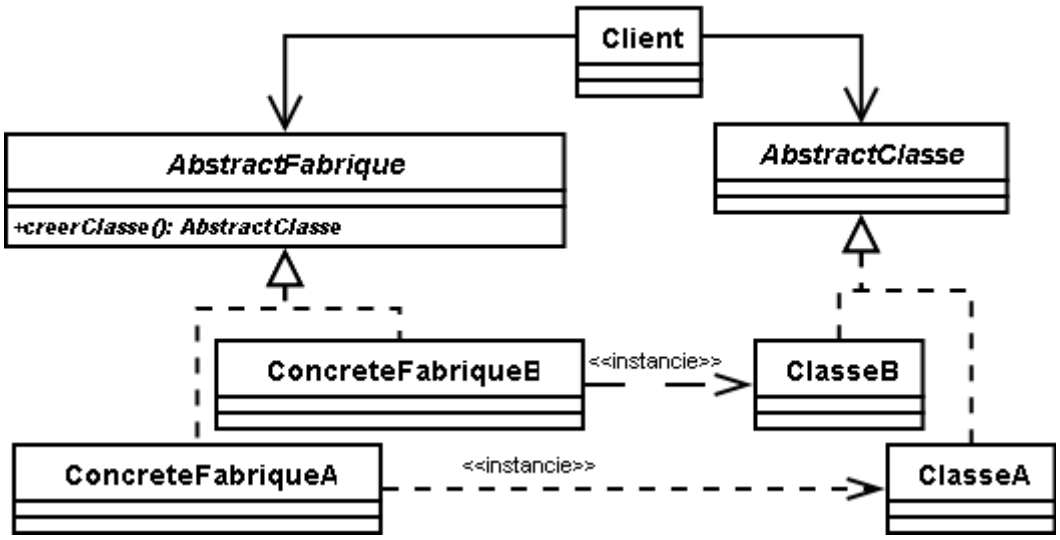
Sur le même principe, en 1995, le livre "Design Patterns -- Elements of Reusable Object-Oriented Software" du GoF, Gang Of Four (**Erich Gamma**, **Richard Helm**, **Ralph Johnson** et **John Vlissides**), présente 23 Design Patterns.

Dans ce livre, chacun des Design Patterns est accompagné d'exemple en C++ et Smalltalk.
En **architecture** des logiciels, un Design Pattern est la description d'une solution à un problème de conception.

Pour faire l'objet d'un Design Pattern, une solution doit être réutilisable. On dit que le Design Pattern est "prouvé" s'il a pu être utilisé dans au moins 3 cas.
Les Design Patterns permettent d'améliorer la qualité de développement et d'en diminuer la durée. En effet, leur application réduit les couplages (points de dépendance) au sein d'une application, apporte de la souplesse, favorise la maintenance et d'une manière générale aide à respecter de " **bonnes pratiques**" de développement.
Les Design Patterns sont classés en trois catégories :
• Créationnels : qui définissent des mécanismes pour l'instanciation et/ou l'initialisation d'objets
• Structuraux : qui organisent les interfaces/classes entre elles
• Comportementaux : qui définissent la communication entre les classes et leurs responsabilités
Le but de ce document est de présenter succinctement chaque Design Pattern du GoF et d'y associer un exemple d'implémentation Java.

CREATIONNELS (CREATIONAL PATTERNS)

A - Fabrique abstraite (Abstract Factory ou Kit)



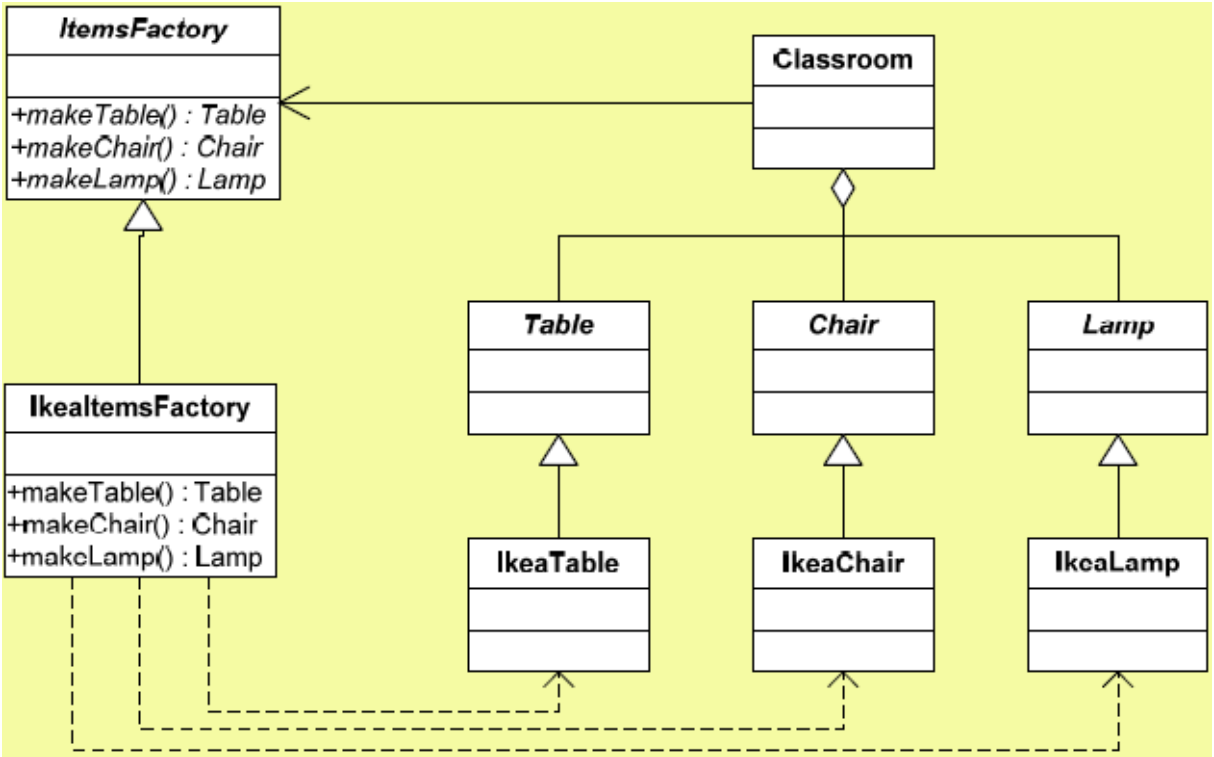
OBJECTIFS :
Fournir une interface pour créer des objets d'une même famille sans préciser leurs **classes concrètes**.

RAISONS DE L'UTILISER :
Le système utilise des objets qui sont regroupés en famille. Selon certains critères, le système utilise les objets d'une famille ou d'une autre. Le système doit utiliser ensemble les objets d'une famille. Cela peut être le cas des éléments graphiques d'un look and feel : pour un look and feel donné, tous les graphiques créés doivent être de la même famille.
La partie cliente manipulera les **interfaces** des **objets** ; ainsi il y aura une indépendance par rapport aux classes concrètes. Chaque fabrique concrète permet d'instancier une famille d'objets (éléments graphiques du même look and feel) ; ainsi la notion de famille d'objets est renforcée.
RESULTAT :
Le Design Pattern permet d'isoler l'appartenance à une famille de classes.

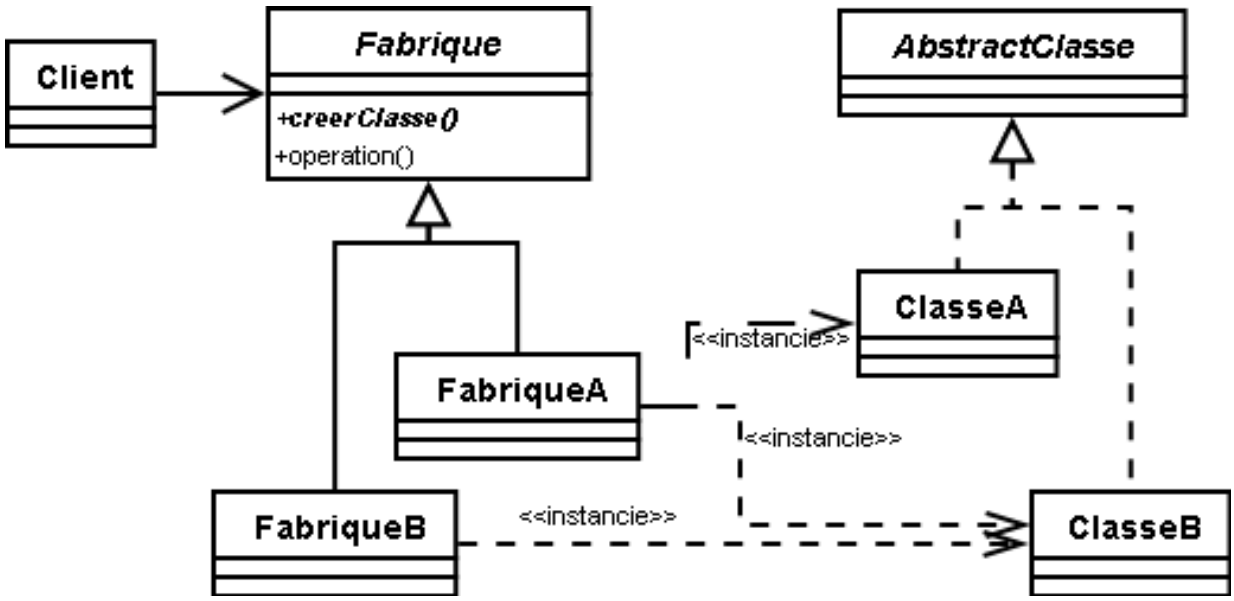
RESPONSABILITES :
• **AbstractFabrique** : définit l'interface des méthodes de création. Dans le diagramme, il n'y a qu'une méthode de création pour un objet d'une classe. Mais, le diagramme sous-entend un nombre indéfini de méthodes pour un nombre indéfini de classes.
• **ConcreteFabriqueA** et **ConcreteFabriqueB** : implèmentent l'interface et instancient la classe concrète appropriée.
• **AbstractClasse** : définit l'interface d'un type d'objet instancié.
• **ClasseA** et **ClasseB** : sont des sous-classes concrètes d'**AbstractClasse**. Elles sont instanciées par les ConcreteFabrique.
• La partie cliente fait appel à une Fabrique pour obtenir une nouvelle instance d'**AbstractClasse**. L'instanciation est transparente pour la partie cliente. Elle manipule une **AbstractClasse**.

En cours nous avons vu que les classes A et B ne dérivaient pas forcément de la même **AbstractClasse** mais pouvaient dériver de deux interfaces différentes.

Exemple :



B - Fabrique (Factory Method ou Virtual Constructor)



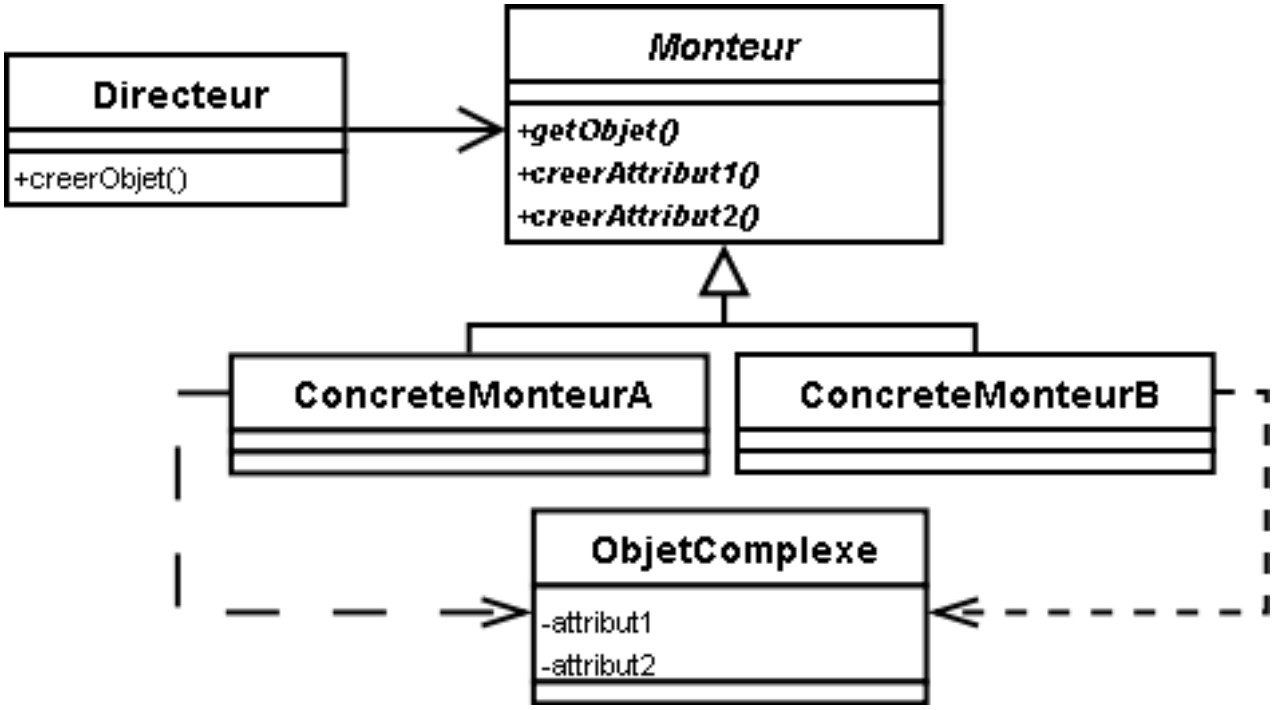
OBJECTIFS :
• Définir une interface pour la création d'un objet, mais laisser les **sous-classes** décider quelle classe instancier.
• Déléguer l'instanciation aux sous-classes.

RAISONS DE L'UTILISER :
Dans le fonctionnement d'une **classe**, il est nécessaire de créer une **instance**. Mais, au niveau de cette classe, on ne connaît pas la classe exacte à instancier.
Cela peut être le cas d'une classe réalisant une sauvegarde dans un flux sortant, mais ne sachant pas s'il s'agit d'un fichier ou d'une sortie sur le réseau.
La classe possède une méthode qui retourne une instance (interface commune au fichier ou à la sortie sur le réseau).
Les autres méthodes de la classe peuvent effectuer les opérations souhaitées sur l'instance (écriture, fermeture).
Les sous-classes déterminent la classe de l'instance créée (fichier, sortie sur le réseau). Une variante du Pattern existe : la **méthode** de création choisit la classe de l'instance à créer en fonction de **paramètres** en entrée de la méthode ou selon des variables de contexte.

RESULTAT :
Le Design Pattern permet d'isoler l'instanciation d'une **classe concrète**.

RESPONSABILITES :
• **AbstractClasse** : définit l'interface de l'objet instancié.
• **ClasseA** et **ClasseB** : sont des sous-classes concrètes d'**AbstractClasse**. Elles sont instanciées par les classes **Fabrique**.
• **Fabrique** : déclare une méthode de création (**creerClasse**). C'est cette méthode qui a la responsabilité de l'instanciation d'un objet **AbstractClasse**. Si d'autres méthodes de la classe ont besoin d'une instance de **AbstractClasse**, elles font appel à la méthode de création. Dans l'exemple, la méthode **operation()** utilise une instance de **AbstractClasse** et fait donc appel à la méthode **creerClasse**. La méthode de création peut être paramétrée ou non. Dans l'exemple, elle est paramétrée, mais le paramètre n'est significatif que pour la **FabriqueA**.
• **FabriqueA** et **FabriqueB** : substituent la méthode de création. Elles implémentent une version différente de la méthode de création.
• La partie cliente utilise une sous-classe de **Fabrique**.
Exemple en fin de chapitre.

C - Monteur (Builder)



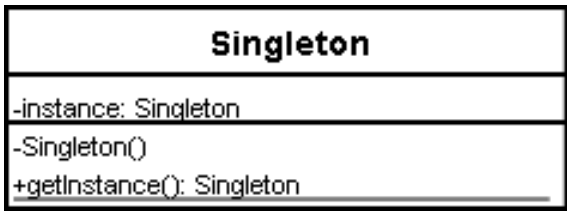
OBJECTIFS :
• Séparer la construction d'un **objet complexe** de sa représentation.
• Permettre d'obtenir des représentations différentes avec le même procédé de construction.

RAISONS DE L'UTILISER :
Le système doit instancier des objets complexes. Ces objets complexes peuvent avoir des représentations différentes.
Cela peut être le cas des différentes fenêtres d'une IHM. Elles comportent des éléments similaires (titre, boutons), mais chacune avec des particularités (libellés, comportements).
Afin d'obtenir des représentations différentes (fenêtres), la partie cliente passe des monteurs différents au directeur.
Le directeur appelera des **méthodes** du monteur retournant les éléments (titre, bouton). Chaque implémentation des méthodes des monteurs retourne des éléments avec des différences (libellés, comportements).

RESULTAT :
Le Design Pattern permet d'isoler des variations de représentations d'un objet.

RESPONSABILITES :
• **ObjetComplexe** : est la classe d'objet complexe à instancier.
• **Monteur** : définit l'interface des méthodes permettant de créer les différentes parties de l'objet complexe.
• **ConcreteMonteurA** et **ConcreteMonteurB** : implémentent les méthodes permettant de créer les différentes parties. Les classes conservent l'objet durant sa construction et fournissent un moyen de récupérer le résultat de la construction.
• **Directeur** : construit un objet en appelant les méthodes d'un **Monteur**.
• La partie cliente instancie un **Monteur**. Elle le fournit au **Directeur**. Elle appelle la méthode de construction du **Directeur**.

D - Singleton (Singleton)



OBJECTIFS :
• Restreindre le nombre d'**instances** d'une classe à une et une seule.
• Fournir une **méthode** pour accéder à cette instance unique.

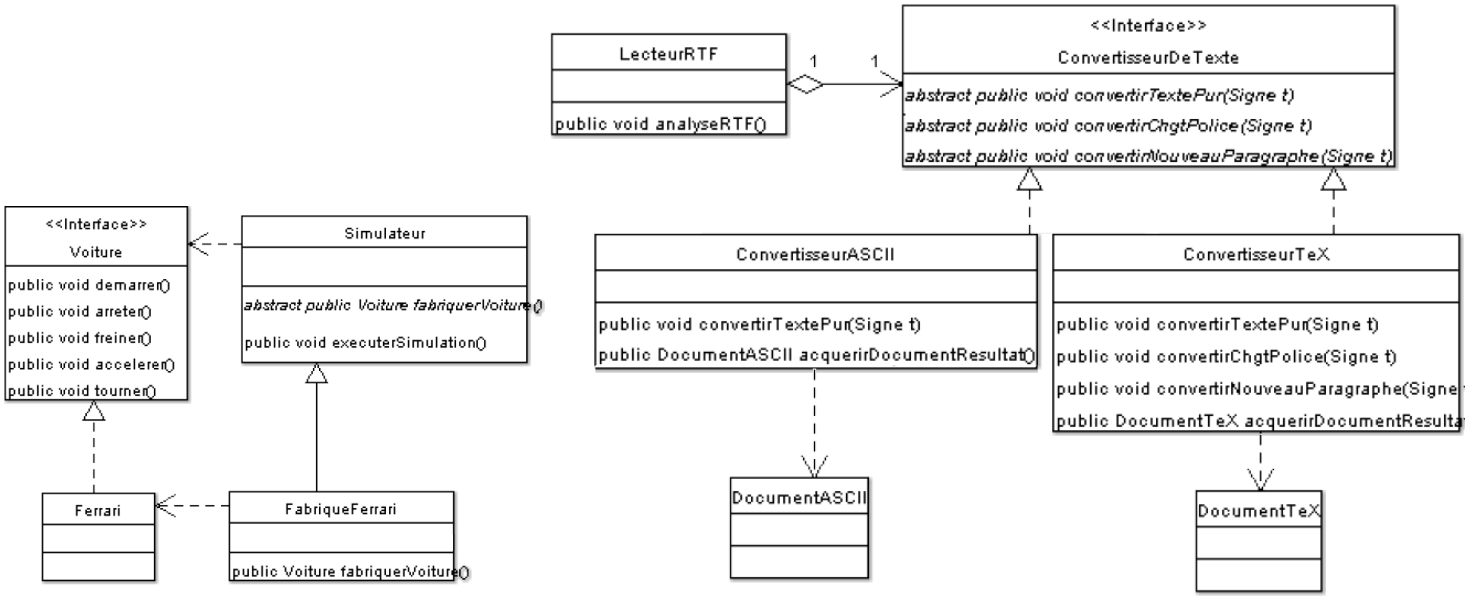
RAISONS DE L'UTILISER :
La classe ne doit avoir qu'une seule instance.
Cela peut être le cas d'une ressource système par exemple.
La classe empêche d'autres classes de l'instancier. Elle possède la seule instance d'elle-même et fournit la seule méthode permettant d'accéder à cette instance.

RESULTAT :
Le Design Pattern permet d'isoler l'unicité d'une instance.

RESPONSABILITES :
Singleton doit restreindre le nombre de ses propres instances à une et une seule. Son constructeur est privé : cela empêche les autres classes de l'instancier. La classe fournit la méthode statique **getInstance()** qui permet d'obtenir l'instance unique.

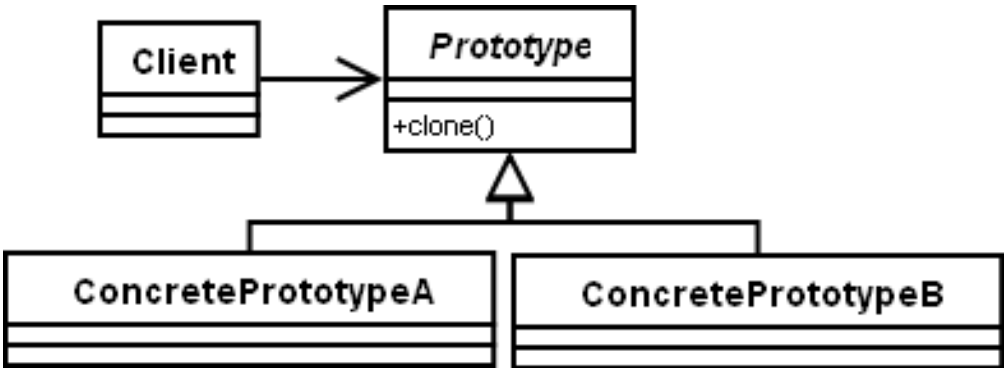
Exemples d'application:

Fabrique concrète : *Monteur :*



Bonus :

E- Prototype (Prototype)



OBJECTIFS :
• Spécifier les genres d'**objet** à créer en utilisant une **instance** comme prototype.
• Créer un nouvel objet en copiant ce prototype.

RAISONS DE L'UTILISER :
Le système doit créer de nouvelles **instances**, mais il ignore de quelle classe. Il dispose cependant d'instances de la classe désirée.
Cela peut être le cas d'un logiciel de DAO comportant un copier-coller. L'utilisateur sélectionne un élément graphique (cercle, rectangle, ...), mais la classe traitant la demande de copier-coller ne connaît pas la classe exacte de l'élément à copier.
La solution est de disposer d'une duplication des instances (élément à copier : cercle, rectangle). La duplication peut être également intéressante pour les performances (la duplication est plus rapide que l'instanciation).

RESULTAT :
Le Design Pattern permet d'isoler l'appartenance à une classe.

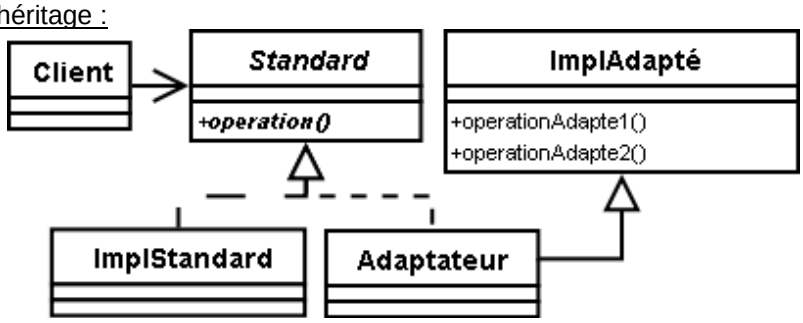
RESPONSABILITES :
• **Prototype** : définit l'interface de duplication de soi-même.
• **ConcretePrototypeA** et **ConcretePrototypeB** : sont des sous-classes concrètes de **Prototype**. Elles implémentent l'inteface de duplication.
• La partie cliente appelle la méthode **clone()** de la classe **Prototype**. Cette méthode retourne un double de l'instance.

STRUCTURAUX (STRUCTURAL PATTERNS)

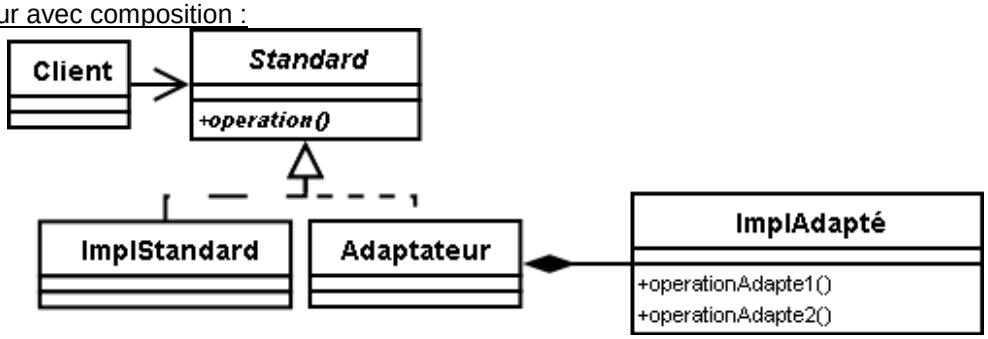
A - Adaptateur (Adapter ou Wrapper)

Le Design Pattern Adaptateur peut avoir deux formes :

Adaptateur avec héritage :



Adaptateur avec composition :



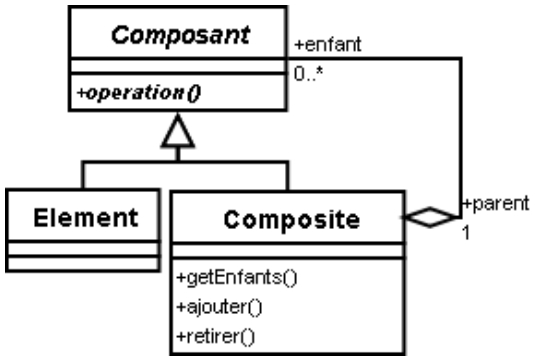
OBJECTIFS :
• Convertir l'interface d'une **classe** dans une autre interface comprise par la partie cliente.
• Permettre à des classes de fonctionner ensemble, ce qui n'aurait pas été possible sinon (à cause de leurs interfaces incompatibles).

RAISONS DE L'UTILISER :
Le système doit intégrer un **sous-système** existant. Ce sous-système a une interface non standard par rapport au système. Cela peut être le cas d'un driver bas niveau pour de l'informatique embarquée. Le driver fournit par le fabricant ne correspond pas à l'interface utilisée par le système pour d'autres drivers.
La solution est de masquer cette interface non stantard au système et de lui présenter une interface standard. La partie cliente utilise les **méthodes** de l'Adaptateur qui utilise les méthodes du sous-système pour réaliser les opérations correspondantes.

RESULTAT :
Le Design Pattern permet d'isoler l'adaptation d'un sous-système.

RESPONSABILITES :
• **Standard** : définit une interface qui est identifiée comme standard dans la partie cliente.
• **ImplStandard** : implémente l'interface **Standard**. Cette classe n'a pas besoin d'être adaptée.
• **ImplAdapte** : permet de réaliser les fonctionnalités définies dans l'interface **Standard**, mais ne la respecte pas. Cette classe a besoin d'être adaptée.
• **Adaptateur** : adapte l'implémentation **ImplAdapte** à l'interface **Standard**. Pour réaliser l'adaptation, l'**Adaptateur** peut utiliser une ou plusieurs méthodes différentes de l'implémentation **ImplAdapte** pour réaliser l'implémentation de chaque méthode de l'interface **Standard**.
• La partie cliente manipule des objets **Standard**. Donc, l'adaptation est transparente pour la partie cliente.

B - Composite (Composite)



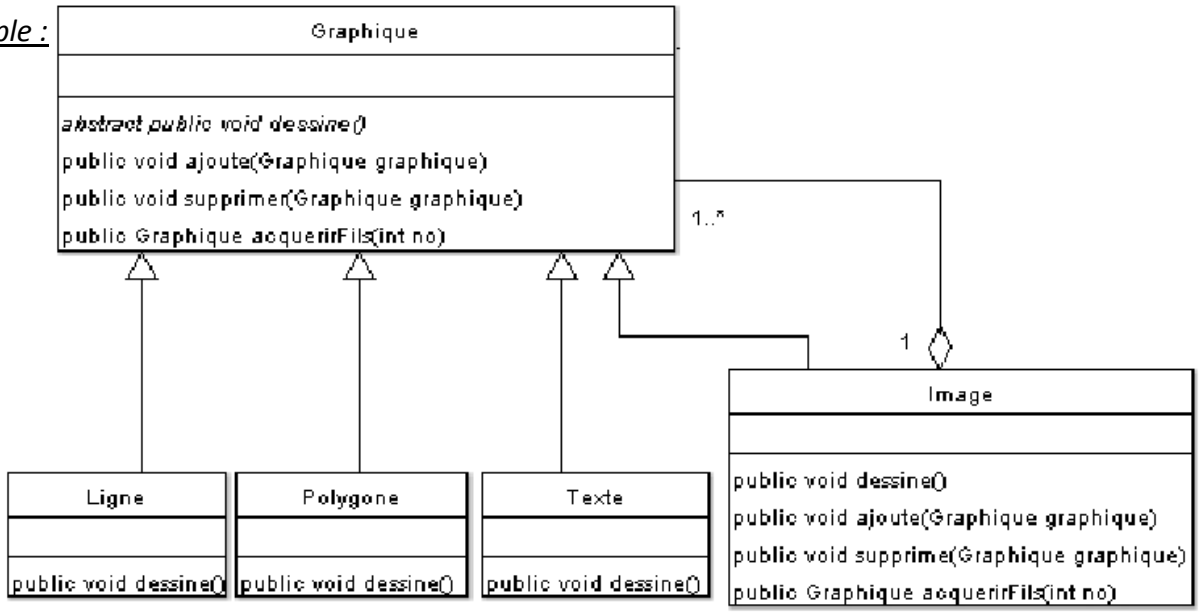
OBJECTIFS :
• Organiser les objets en structure arborescente afin de représenter une hiérarchie.
• Permettre à la partie cliente de manipuler un **objet unique** et un **objet composé** de la même manière.

RAISONS DE L'UTILISER :
Le système comporte une hiérarchie avec un nombre de niveaux non déterminé. Il est nécessaire de pouvoir considérer un groupe d'éléments comme un élément unique.
Cela peut être le cas des éléments graphiques d'un logiciel de DAO. Plusieurs éléments graphiques peuvent être regroupés en un nouvel élément graphique.
Chaque élément est un composant potentiel. En plus des éléments classiques, il y a un élément composite qui peut être composé de plusieurs composants. Comme l'élément composite est un composant potentiel, il peut être composé d'autres éléments composites.

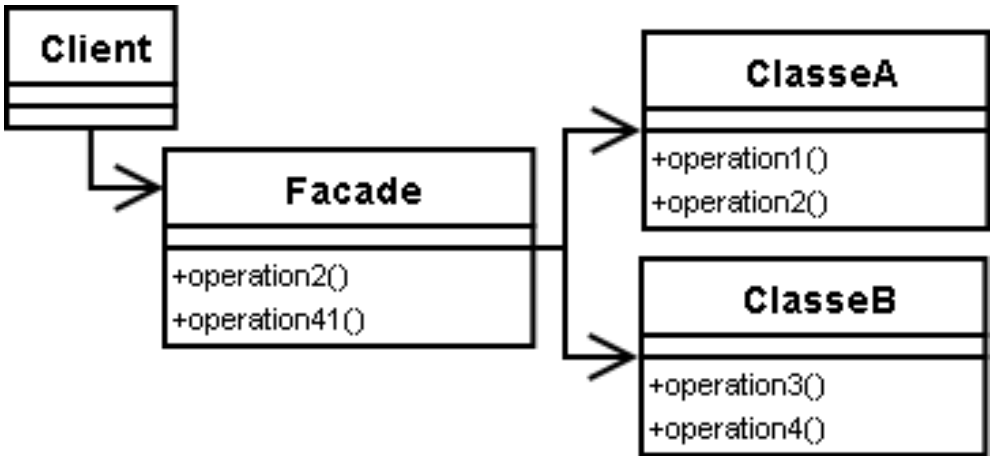
RESULTAT :
Le Design Pattern permet d'isoler l'appartenance à un agrégat.

RESPONSABILITES :
• **Composant** : définit l'interface d'un objet pouvant être un composant d'un autre objet de l'arborescence.
• **Element** : implémente un objet de l'arborescence n'ayant pas d'objet le composant.
• **Composite** : implémente un objet de l'arborescence ayant un ou des objets le composant.
• La partie client manipule les objets par l'interface **Composant**.

Exemple :



C - Façade (Facade)



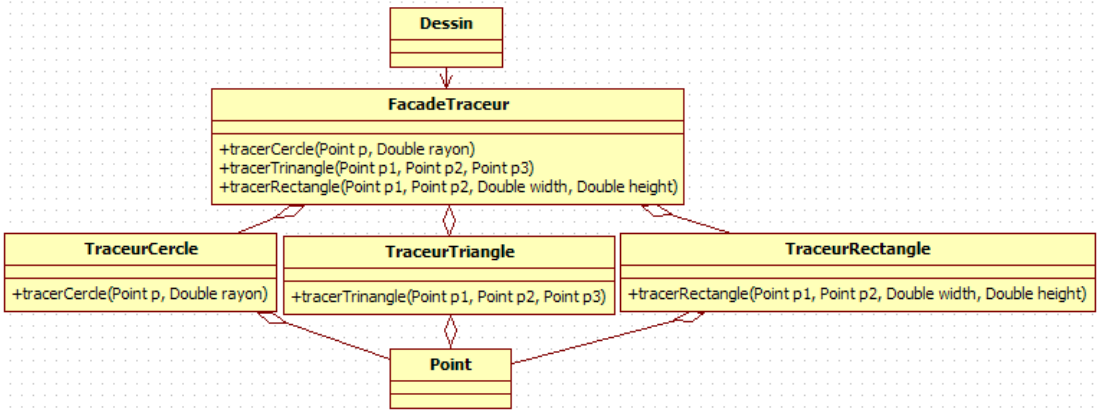
OBJECTIFS :
• Fournir une interface unique en remplacement d'un ensemble d'interfaces d'un **sous-système**.
• Définir une interface de haut niveau pour rendre le sous-système plus simple d'utilisation.

RAISONS DE L'UTILISER :
Le système comporte un sous-système complexe avec plusieurs interfaces. Certaines de ces interfaces présentent des opérations qui ne sont pas utiles au reste du système.
Cela peut être le cas d'un sous-système communiquant avec des outils de mesure ou d'un sous-système d'accès à la base de données.
Il serait plus judicieux de passer par une seule interface présentant seulement les opérations utiles.
Une classe unique, la façade, présente ces opérations réellement nécessaires. *Remarque : La façade peut également implémenter le Design Pattern **Singleton***.

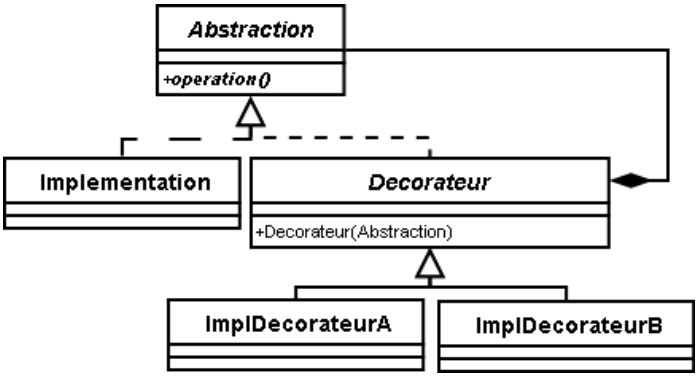
RESULTAT :
Le Design Pattern permet d'isoler les fonctionnalités d'un sous-système utiles à la partie cliente.

RESPONSABILITES :
• **ClasseA** et **ClasseB** : implémentent diverses fonctionnalités.
• **Facade** : présente certaines fonctionnalités. Cette classe utilise les implémentations des objets **ClasseA** et
ClasseB. Elle expose une version simplifiée du sous-système **ClasseA**-**ClasseB**.
• La partie cliente fait appel aux méthodes présentées par l'objet **Facade**. Il n'y a donc pas de dépendances entre la partie cliente et le sous-système **ClasseA**-**ClasseB**.

Exemple :



D - Décorateur (Decorator ou Wrapper)



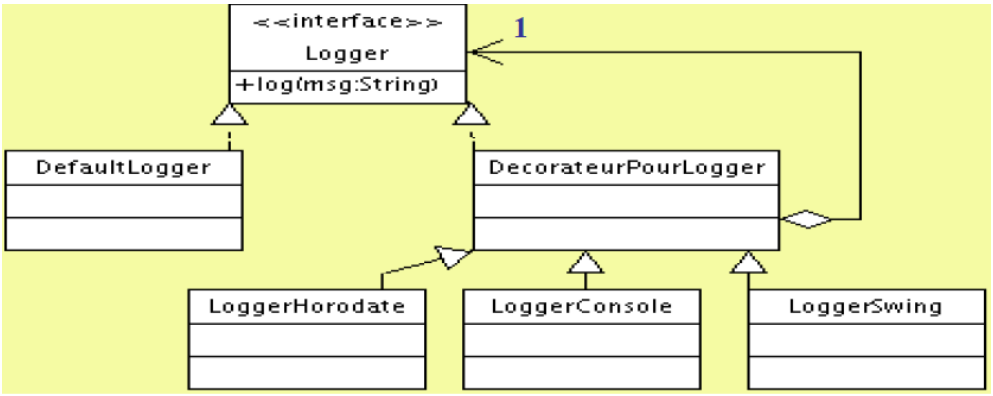
OBJECTIFS :
• Ajouter dynamiquement des **responsabilités** (non obligatoires) à un objet.
• Eviter de sous-classer la classe pour rajouter ces responsabilités.

RAISONS DE L'UTILISER :
Il est nécessaire de pouvoir étendre les responsabilités d'une **classe** sans avoir recours au sous-classage.
Cela peut être le cas d'une classe gérant des d'entrées/sorties à laquelle on souhaite ajouter un buffer et des traces de log.
La classe de départ est l'implémentation. Les fonctionnalités supplémentaires (buffer, log) sont implémentées par des classes supplémentaires : les décorateurs. Les décorateurs ont la même **interface** que la classe de départ.
Dans leur **implémentation** des méthodes, elles implémentent les fonctionnalités supplémentaires et font appel à la méthode correspondante d'une **instance** avec la même interface. Ainsi, il est possible d'enchainer plusieurs responsabilités supplémentaires, puis d'aboutir à l'implémentation finale.

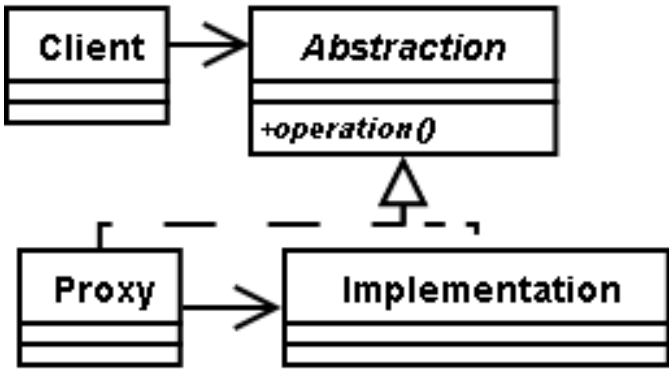
RESULTAT :
Le Design Pattern permet d'isoler les responsabilités d'un objet.

RESPONSABILITES :
• **Abstraction** : définit l'interface générale.
• **Implementation** : implémente l'interface générale. Cette classe contient l'implémentation de l'interface correspondant aux fonctionnalités souhaitées à la base.
• **Decorateur** : définit l'interface du décorateur et contient une référence vers un objet **Abstraction**.
• **ImplDecorateurA** et **ImplDecorateurB** : implémentent des décorateurs. Les décorateurs ont un contructeur acceptant un objet **Abstraction**. Les méthodes des décorateurs appellent la même méthode de l'objet qui a été passée au constructeur. La décoration ajoute des responsabiltés en effectuant des opérations avant et/ou après cet appel.
• La partie cliente manipule un objet **Abstraction**. En réalité, cet objet **Abstraction** peut être un objet **Implementation** ou un objet **Decorateur**. Ainsi, des fonctionnalités supplémentaires peuvent être ajoutées à la méthode d'origine. Ces fonctionnalités peuvent être par exemple des traces de log ou une gestion de buffer pour des entrées/sorties.

Exemple :



E - Proxy (Proxy ou Surrogate)



OBJECTIFS :
Fournir un intermédiaire entre la partie cliente et un **objet** pour contrôler les accès à ce dernier.

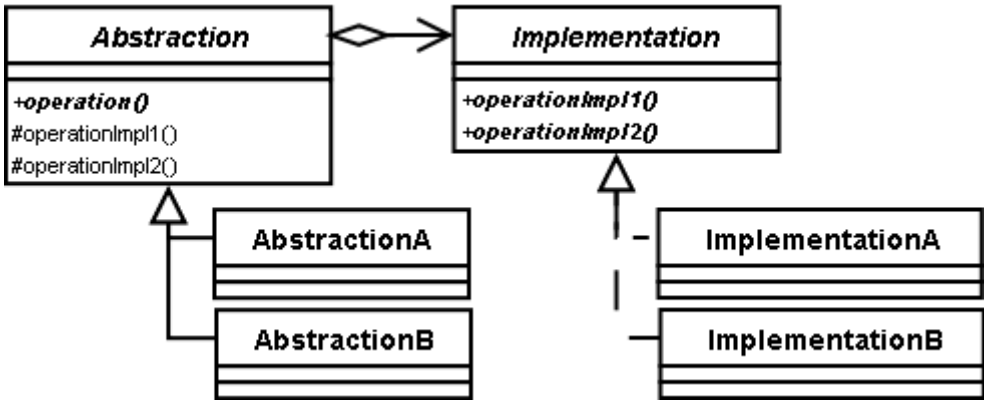
RAISONS DE L'UTILISER :
Les opérations d'un objet sont coûteuses en temps ou sont soumises à une gestion de droits d'accès. Il est nécessaire de contrôler l'accès à un objet.
Cela peut être un système de chargement d'un document. Le document est très lourd à charger en mémoire ou il faut certaines habilitations pour accéder à ce document.
L'objet réel (système de chargement classique) est l'implémentation. L'intermédiaire entre l'implémentation et la partie cliente est le proxy. Le proxy fournit la même interface que l'implémentation. Mais il ne charge le document qu'en cas de réel besoin (pour l'affichage par exemple) ou n'autorise l'accès que si les conditions sont satisfaites.

RESULTAT :
Le Design Pattern permet d'isoler le comportement lors de l'accès à un objet.

RESPONSABILITES :
• **Abstraction** : définit l'interface des classes **Implementation** et **Proxy**.
• **Implementation** : implémente l'interface. Cette classe définit l'objet que l'objet **Proxy** représente.
• **Proxy** : fournit un intermédiaire entre la partie cliente et l'ojet **Implementation**. Cet intermédiaire peut avoir plusieurs buts (synchronisation, contrôle d'accès, cache, accès distant, ...). Dans l'exemple, la classe **Proxy** n'instancie un objet **Implementation** qu'en cas de besoin pour appeler la méthode correspondante de la classe **Implementation**.
• La partie cliente appelle la méthode **operation()** de l'objet **Proxy**.

Bonus

Pont (Bridge ou Handle/Body)



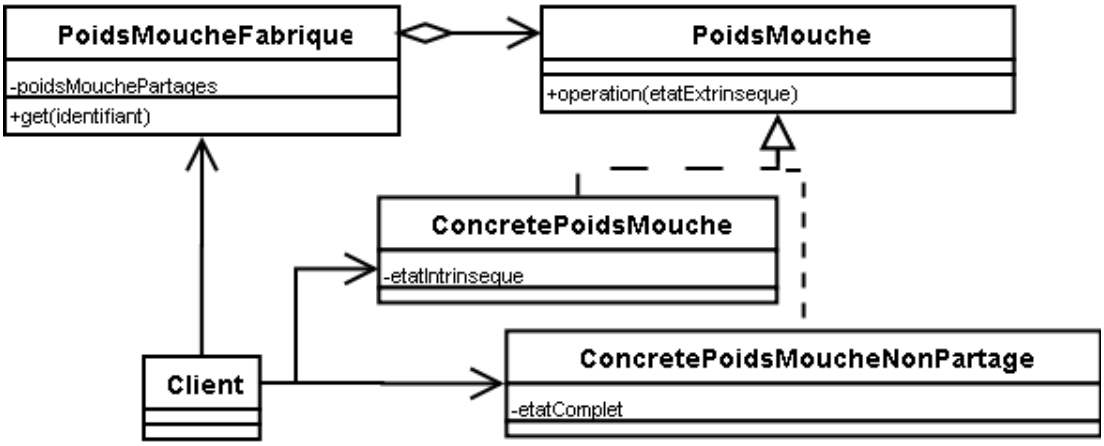
OBJECTIFS :
• Découpler l'abstraction d'un concept de son implémentation.
• Permettre à l'abstraction et l'implémentation de varier indépendamment.

RAISONS DE L'UTILISER :
Le système comporte une couche bas niveau réalisant l'implémentation et une couche haut niveau réalisant l'abstraction. Il est nécessaire que chaque couche soit indépendante.
Cela peut être le cas du système d'édition de documents d'une application. Pour l'implémentation, il est possible que l'édition aboutisse à une sortie imprimante, une image sur disque, un document PDF, etc... Pour l'abstraction, il est possible qu'il s'agisse d'édition de factures, de rapports de stock, de courriers divers, etc...
Chaque implémentation présente une interface pour les opérations de bas niveau standard (sortie imprimante), et chaque abstraction hérite d'une classe effectuant le lien avec cette interface (tracer une ligne). Ainsi les abstractions peuvent utiliser ce lien pour appeler la couche implémentation pour leurs besoins (imprimer facture).

RESULTAT :
Le Design Pattern permet d'isoler le lien entre une couche de haut niveau et celle de bas niveau.

RESPONSABILITES :
• **Implementation** : définit l'interface de l'implémentation. Cette interface n'a pas besoin de correspondre à l'interface de l'**Abstraction**. L'**Implementation** peut, par exemple, définir des opérations primitives de bas niveau et l'**Abstraction** des opérations de haut niveau qui utilisent les opérations de l'**Implementation**.
• **ImplementationA** et **ImplementationB** : sont des sous-classes concrètes de l'implémentation.
• **Abstraction** : définit l'interface de l'abstraction. Elle possède une référence vers un objet **Implementation**.
C'est elle qui définit le lien entre l'abstraction et l'implémentation. Pour définir ce lien, la classe implémente des méthodes qui appellent des méthodes de l'objet **Implementation**.
• **AbstractionA** et **AbstractionB** : sont des sous-classes concrètes de l'abstraction. Elle utilise les méthodes définies par la classe **Abstraction**.
• La partie cliente fournit un objet **Implementation** à l'objet **Abstraction**. Puis, elle fait appel aux méthodes fournit par l'interface de l'abstraction.

Poids-Mouche (Flyweight)



OBJECTIFS :
Utiliser le partage pour gérer efficacement un grand nombre d'**objets** de faible granularité.

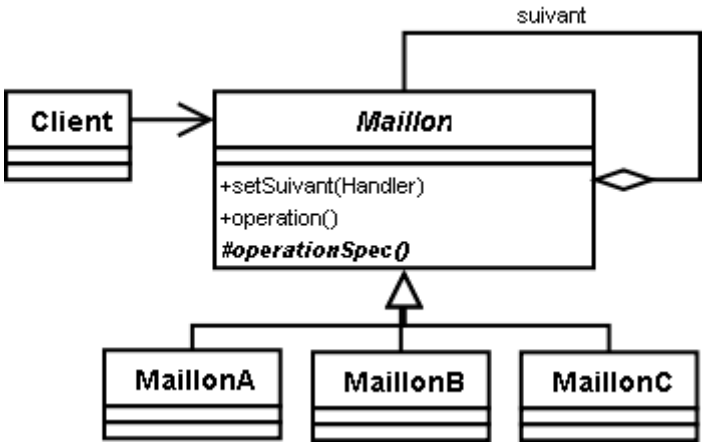
RAISONS DE L'UTILISER :
Un système utilise un grand nombre d'**instances**. Cette quantité occupe une place très importante en mémoire.
Or, chacune de ces instances a des **attributs** extrinsèques (propre au contexte) et intrinsèques (propre à l'objet).
Cela peut être les caractéristiques des traits dans un logiciel de DAO. Le trait a une épaisseur (simple ou double), une continuité (continu, en pointillé), une ombre ou pas, des coordonnées. Les caractéristiques d'épaisseur, de continuité et d'ombre sont des attributs intrinsèques à un trait, tandis que les coordonnées sont des attributs extrinsèques.
Plusieurs traits possèdent des épaisseurs, continuité et ombre similaires. Ces similitudes correspondent à des styles de trait.
En externalisant les attributs intrinsèques des objets (style de trait), on peut avoir en mémoire une seule instance correspondant à un groupe de valeurs (simple-continu-sans ombre, double-pointillé-ombre). Chaque objet avec des attributs extrinsèques (trait avec les coordonnées) possède une référence vers une instance d'attributs intrinsèques (style de trait). On obtient deux types de poids-mouche : les poids-mouche partagés (style de trait) et les poidsmouche non partagés (le trait avec ses coordonnées). La partie cliente demande le poids-mouche qui l'intéresse à la fabrique de poids-mouche. S'il s'agit d'un poids-mouche non partagé, la fabrique le créera et le retournera. S'il s'agit d'un poids-mouche partagé, la fabrique vérifiera si une instance existe. Si une instance existe, la fabrique la retourne, sinon la fabrique la crée et la retourne.

RESULTAT :
Le Design Pattern permet d'isoler des objets partageables.

RESPONSABILITES :
• **PoidsMouche** : déclare l'interface permettant à l'objet de recevoir et d'agir en fonction de données extrinsèques. On externalise les données extrinsèques d'un objet **PoidsMouche** afin qu'il puisse être réutilisé.
• **ConcretePoidsMouche** : implémente l'interface poids-mouche. Les informations contenues dans un **ConcretePoidsMouche** sont intrinsèques (sans lien avec son contexte). Puisque, ce type de poids-mouche est obligatoirement partagé.
• **ConcretePoidsMoucheNonPartage** : implémente l'interface poids-mouche. Ce type de poids-mouche n'est pas partagé. Il possède des données intrinsèques et extrinsèques.
• **PoidsMoucheFabrique** : fournit une méthode retournant une instance de **PoidsMouche**. Si les paramètres de l'instance souhaitée correspondent à un **PoidsMouche** partagé, l'objet **PoidsMoucheFabrique** retourne une instance déjà existante. Sinon l'objet **PoidsMoucheFabrique** crée une nouvelle instance.
• La partie cliente demande à PoidsMoucheFabrique de lui fournir un PoidsMouche

VI - COMPORTEMENTAUX (BEHAVIORAL PATTERNS)

A - Chaîne de responsabilité (Chain of responsibility)



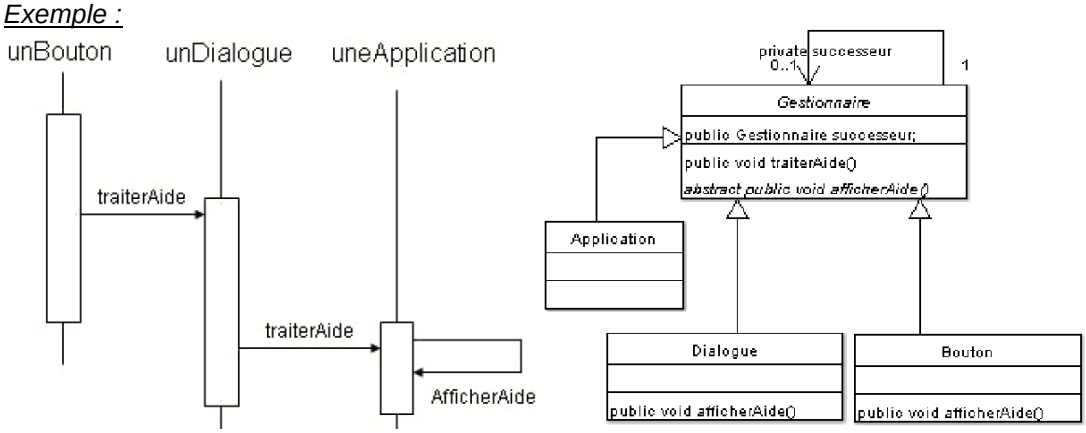
OBJECTIFS :
• Eviter le couplage entre l'**émetteur** d'une **requête** et son **récepteur** en donnant à plus d'un **objet** une chance de traiter la requête.
• Chaîner les objets récepteurs et passer la requête tout le long de la chaîne jusqu'à ce qu'un objet la traite.

RAISONS DE L'UTILISER :
Le système doit gérer un requête. La requête implique plusieurs objets pour la traiter.
Cela peut être le cas d'un système complexe d'habilitations possèdant plusieurs critères afin d'autoriser l'accès. Ces critères peuvent varier en fonction de la configuration.
Le traitement est réparti sur plusieurs objets : les maillons. Les maillons sont chaînés. Si un maillon ne peut réaliser le traitement (vérification des droits), il donne sa chance au maillon suivant. Il est facile de faire varier les maillons impliqués dans le traitement.

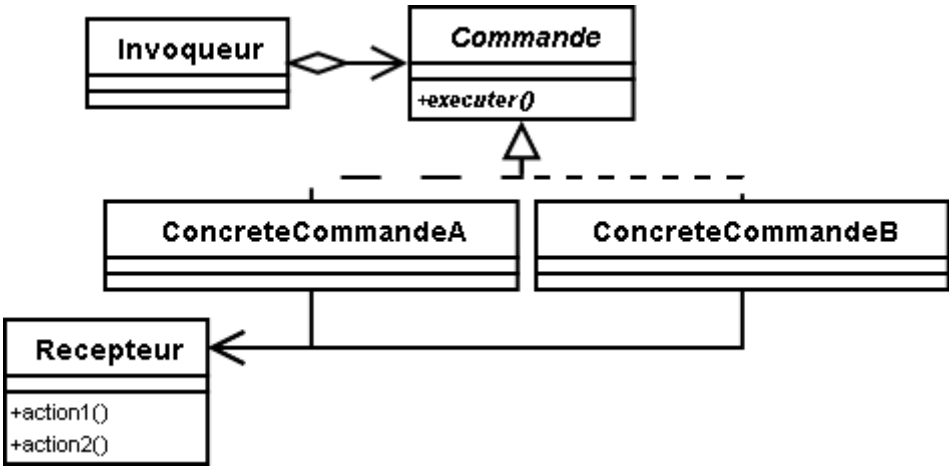
RESULTAT :
Le Design Pattern permet d'isoler les différentes parties d'un traitement.

RESPONSABILITES :
• **Maillon** : définit l'interface d'un maillon de la chaîne. La classe implèmente la gestion de la succession des maillons.
• **MaillonA**, **MaillonB** et **MaillonC** : sont des sous-classes concrètes qui définissent un maillon de la chaîne.
Chaque maillon a la responsabilité d'une partie d'un traitement.
• La partie cliente appelle la méthode **operation()** du premier maillon de la chaîne.

Exemple :



B - Commande (Command, Action ou Transaction)



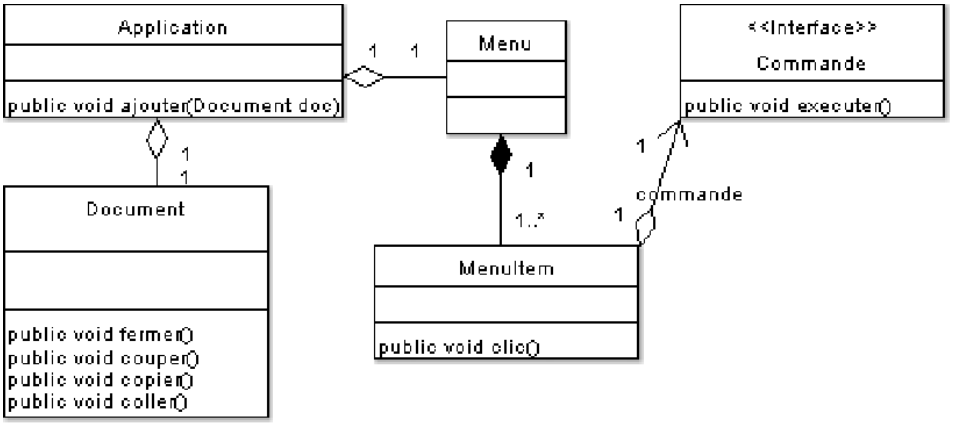
OBJECTIFS :
• Encapsuler une **requête** sous la forme d'**objet**.
• Paramétrer facilement des requêtes diverses.
• Permettre des **opérations** réversibles.

RAISONS DE L'UTILISER :
Le système doit traiter des requêtes. Ces requêtes peuvent provenir de plusieurs **émetteurs**. Plusieurs émetteurs peuvent produire la même requête. Les requêtes doivent pouvoir être annulées. Cela peut être le cas d'une IHM avec des boutons de commande, des raccourcis clavier et des choix de menu aboutissant à la même requête.
La requête est encapsulée dans un objet : la commande. Chaque commande possède un objet qui traitera la requête : le récepteur. La commande ne réalise pas le traitement, elle est juste porteuse de la requête. Les émetteurs potentiels de la requête (éléments de l'IHM) sont des invoqueurs. Plusieurs invoqueurs peuvent se partager la même commande.

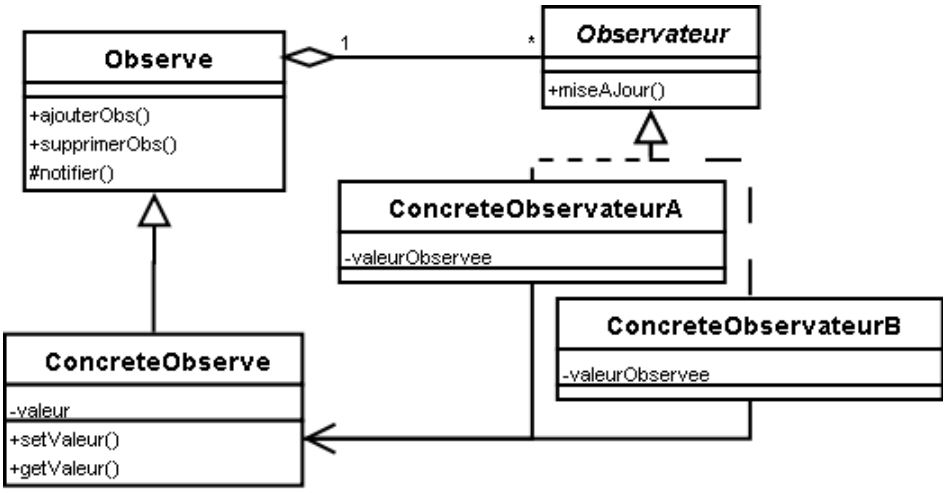
RESULTAT :
Le Design Pattern permet d'isoler une requête.

RESPONSABILITES :
• **Commande** : définit l'interface d'une commande.
• **ConcreteCommandeA** et **ConcreteCommandeB** : implémentent une commande. Chaque classe implémente la méthode **executer()**, en appelant des méthodes de l'objet **Recepteur**.
• **Invoqueur** : déclenche la commande. Il appelle la méthode **executer()** d'un objet **Commande**.
• **Recepteur** : reçoit la commande et réalise les opérations associées. Chaque objet **Commande** concret possède un lien avec un objet **Recepteur**.
• La partie cliente configure le lien entre les objets **Commande** et le **Recepteur**.

Exemple :



C - Observateur (Observer, Dependents ou Publish-Subscribe)



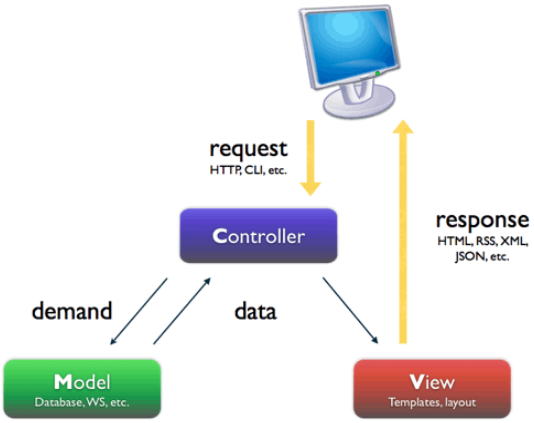
OBJECTIFS :
Prévenir des **objets** observateurs, enregistrés auprès d'un objet observé, d'un **événement**.

RAISONS DE L'UTILISER :
Un objet doit connaitre les changements d'**état** d'un autre objet. L'objet doit être informé immédiatement.
Cela peut être le cas d'un tableau affichant des statistiques. Si une nouvelle donnée est entrée, les statistiques sont recalculées. Le tableau doit être informé du changement, afin qu'il soit rafraîchi.
L'objet devant connaître le changement (le tableau) est un observateur. Il s'enregistre en tant que tel auprès de l'objet dont l'état change. L'objet dont l'état change (les statistiques) est un "observe". Il informe ses observateurs en cas d'événement.

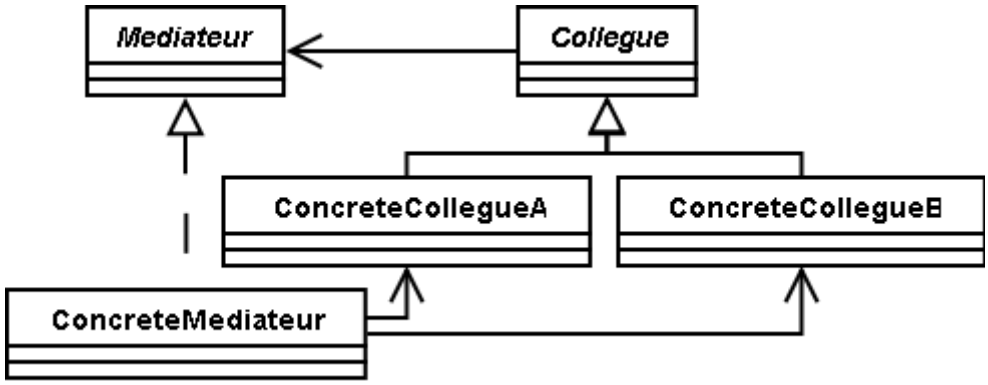
RESULTAT :
Le Design Pattern permet d'isoler un **algorithme** traitant un événement.

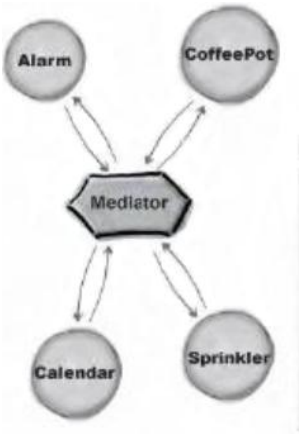
RESPONSABILITES :
• **Observe** : est l'interface de l'objet à observer. Il possède une liste d'objets **Observateur**. Il fournit des méthodes pour ajouter ou supprimer des objets **Observateur** à la liste. Il fournit également un méthode pour avertir les objets **Observateur**.
• **ConcreteObserve** : est l'implémentation de l'objet à observer. Lorsqu'une valeur est modifiée, la méthode **notifier()** de la classe **Observe** est appelée.
• **Observateur** : définit l'interface de l'observateur. Il déclare la/les méthode(s) que l'objet **Observe** appelle en cas d'événements.
• **ConcreteObservateurA** et **ConcreteObservateurB** : sont des sous-classes concrètes de **bservateur**. Ils implémentent des comportements de mise à jour en cas d'événements.
• La partie cliente indique à l'objet **Observe** les objets **Observateur** qu'il avertira.

MVC ?



D - Médiateur (Mediator)





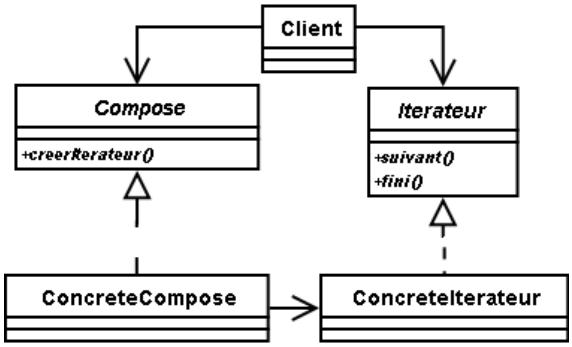
OBJECTIFS :
• Gérer la transmission d'informations entre des **objets** interagissant entre eux.
• Avoir un couplage faible entre les objets puisqu'ils n'ont pas de lien direct entre eux.
• Pouvoir varier leur interaction indépendamment.

RAISONS DE L'UTILISER :
Différents objets ont des interactions. Un événement sur l'un provoque une action ou des actions sur un autre ou d'autres objets.
Cela peut être les éléments d'IHM. Si une case est cochée, certains éléments deviennent accessibles. Si une autre case est cochée, des couleurs de l'IHM changent.
Si les **classes** communiquent directement, il y a un couplage très fort entre elles. Une classe dédiée à la communication permet d'éviter cela. Chaque élément interagissant (élément de l'IHM) est un collègue. La classe dédiée à la communication est un médiateur.

RESULTAT :
Le Design Pattern permet d'isoler la communication entre des objets.

RESPONSABILITES :
• **Collegue** : définit l'interface d'un collègue. Il s'agit d'une famille d'objets qui s'ignorent entre eux mais qui doivent se transmettre des informations.
• **ConcreteCollegueA** et **ConcreteCollegueB** : sont des sous-classes concrètes de l'interface **Collegue**. Elles ont une référence sur un objet **Mediateur** auquel elles transmettront les informations.
• **Mediateur** : définit l'interface de communication entre les objets **Collegue**.
• **ConcreteMediateur** : implémente la communication et maintient une référence sur les objets **Collegue**.

E - Itérateur (Iterator ou Cursor)

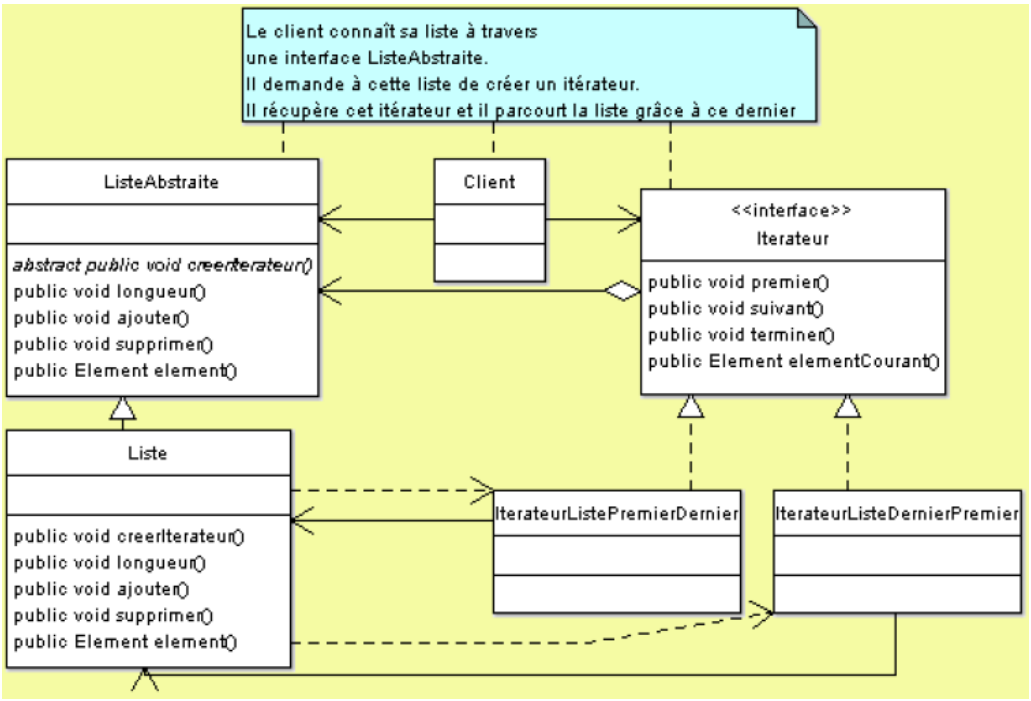


OBJECTIFS :
Fournir un moyen de parcourir séquentiellement les éléments d'un **objet composé**.

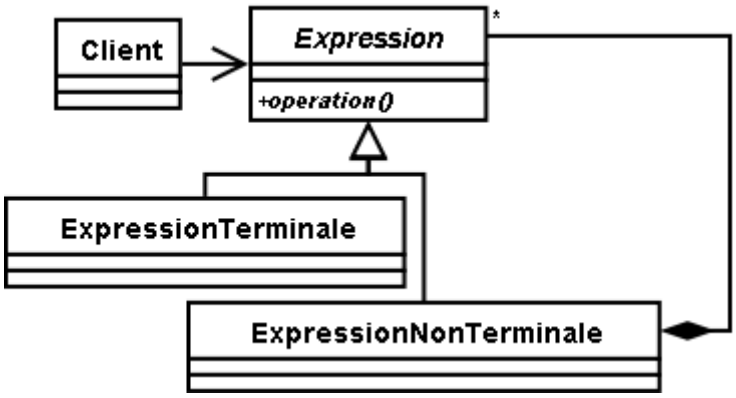
RAISONS DE L'UTILISER :
Le système doit parcourir les éléments d'un objet complexe. La classe de l'objet complexe peut varier. Cela est le cas des classes représentant des listes et des ensembles en Java.
Les classes d'un objet complexe (listes) sont des "composes". Elles ont une méthode retournant un itérateur, qui permet de parcourir les éléments. Tous les iterateurs ont la même **interface**. Ainsi, le système dispose d'un moyen homogène de parcourir les composes.

RESULTAT :
Le Design Pattern permet d'isoler le parcours d'un agrégat.

RESPONSABILITES :
• **Compose** : définit l'interface d'un objet composé permettant de créer un **Iterateur**.
• **ConcreteCompose** : est une sous-classe de l'interface **Compose**. Elle est composée d'éléments et implémente la méthode de création d'un **Iterateur**.
• **Iterateur** : définit l'interface de l'itérateur, qui permet d'accéder aux éléments de l'objet **Compose**.
• **ConcreteIterateur** : est une sous-classe de l'inteface **Iterateur**. Elle fournit une implémentation permettant de parcourir les éléments de **ConcreteCompose**. Elle conserve la trace de la position courante.
• La partie cliente demande à l'objet **Compose** de fournir un objet **Iterateur**. Puis, elle utilise l'objet **Iterateur** afin de parcourir les éléments de l'objet **Compose**.



F - Interpréteur (Interpreter)



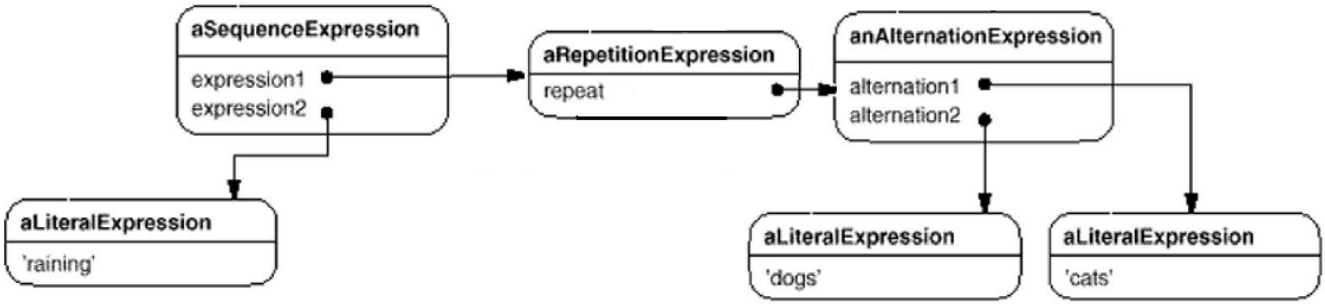
OBJECTIFS :
• Définir une représentation de la grammaire d'un **langage**.
• Utiliser cette représentation pour interpréter les éléments de ce langage.

RAISONS DE L'UTILISER :
Le système doit interpréter un langage. Ce langage possède une grammaire prédéfinie qui constitue un ensemble d'**opérations** qui peuvent être effectuées par le système.
Cela peut être le cas d'un logiciel embarqué dont la configuration des écrans serait stockée dans des fichiers XML.
Le logiciel lit ces fichiers afin de réaliser son affichage et l'enchaînement des écrans.
Une structure arborescente peut représenter la grammaire du langage. Elle permet d'interpréter les différents éléments du langage.

RESULTAT :
Le Design Pattern permet d'isoler les éléments d'un langage.

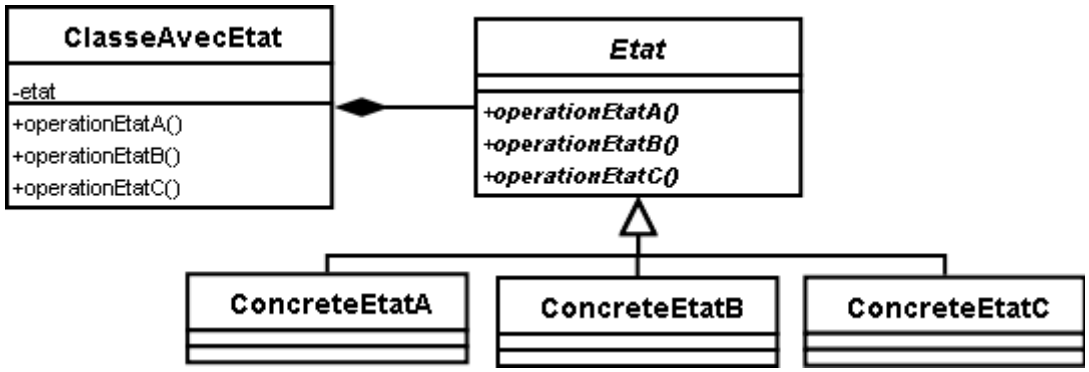
RESPONSABILITES :
• **Expression** : définit l'interface d'une expression.
• **ExpressionNonTerminale** : implémente une expression non terminale. Un expression non terminale peut contenir d'autres expressions.
• **ExpressionTerminale** : implémente une expression terminale. Un expression terminale ne peut pas contenir d'autres expressions.
• La partie cliente effectue des opérations sur la structure pour interpréter le langage représenté.

Exemple : « Raining cats dogs »



Ceci est la schématisation de la décomposition de la phrase, ce n'est pas un UML !

G - Etat (State ou Objects for States)



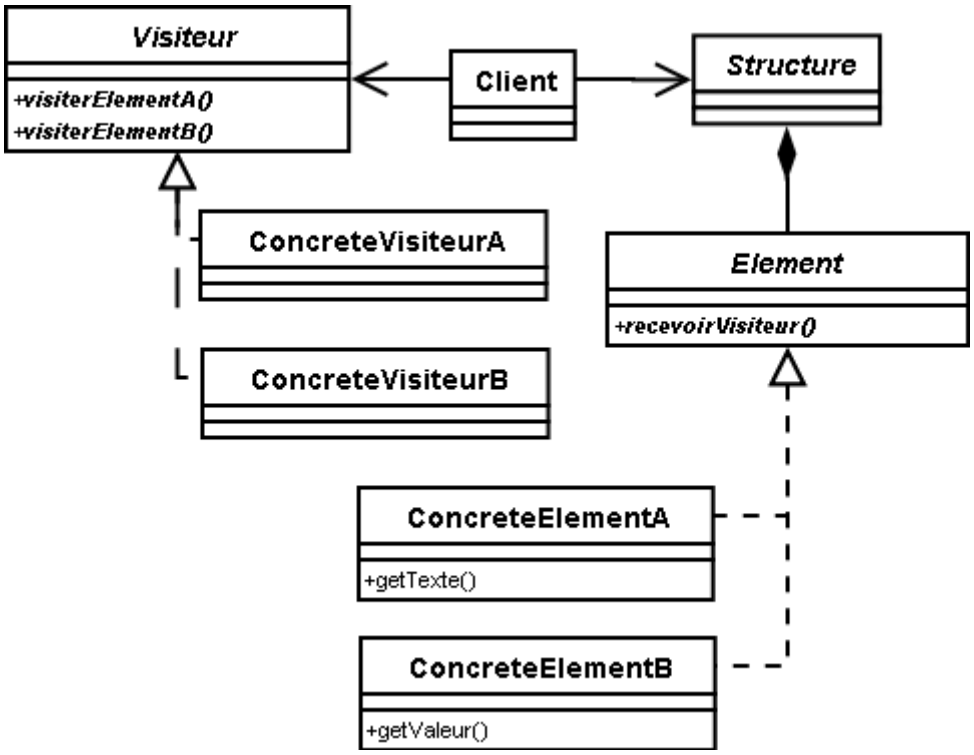
OBJECTIFS :
Changer le comportement d'un **objet** selon son **état interne**.

RAISONS DE L'UTILISER :
Un objet a un fonctionnement différent selon son état interne. Son état change selon les **méthodes** appelées.
Cela peut être un document informatique. Il a comme fonctions ouvrir, modifier, sauvegarder ou fermer. Le comportement de ces méthodes change selon l'état du document.
Les différents états internes sont chacun représenté par une **classe** état (ouvert, modifié, sauvegardé et fermé).
Les états possèdent des méthodes permettant de réaliser les opérations et de changer d'état (ouvrir, modifier, sauvegarder et fermer). Certains états bloquent certaines opérations (modifier dans l'état fermé). L'objet avec état (document informatique) maintient une référence vers l'état actuel. Il présente les opérations à la partie cliente.

RESULTAT :
Le Design Pattern permet d'isoler les **algorithmes** propres à chaque état d'un objet.

RESPONSABILITES :
• **ClasseAvecEtat** : est une classe avec état. Son comportement change en fonction de son état. La partie changeante de son comportement est déléguée à un objet **Etat**.
• **Etat** : définit l'interface d'un comportement d'un état.
• **ConcreteEtatA**, **ConcreteEtatB** et **ConcreteEtatC** : sont des sous-classes concrètes de l'interface **Etat**. Elles implémentent des méthodes qui sont associées à un **Etat**.

H - Visiteur (Visitor)

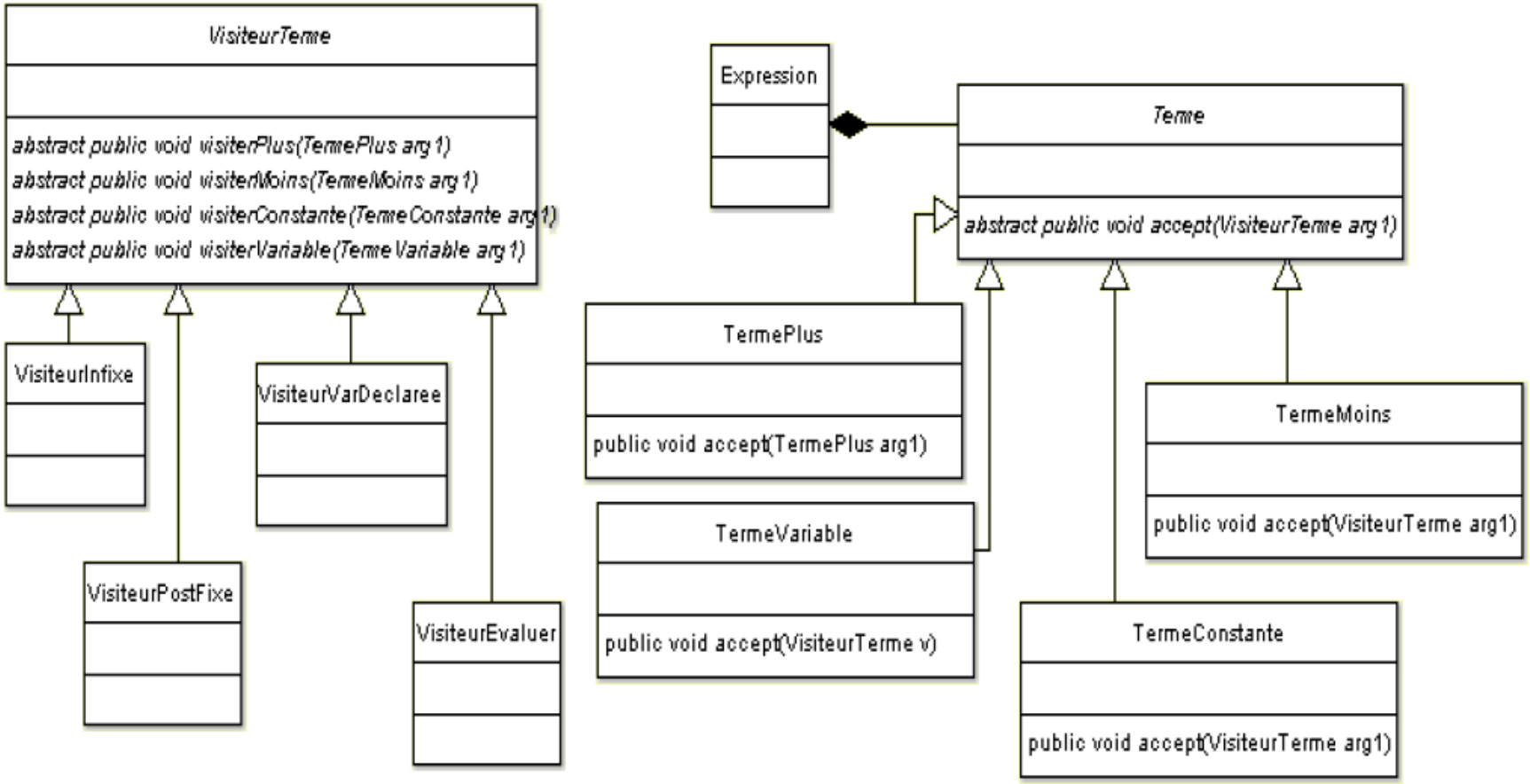


OBJECTIFS :
Séparer un **algorithme** d'une **structure de données**.

RAISONS DE L'UTILISER :
Il est nécessaire de réaliser des **opérations** sur les éléments d'un objet structuré. Ces opérations varient en fonction de la nature de chaque élément et les opérations peuvent être de plusieurs types. Cela peut être le cas d'un logiciel d'images de synthèse. L'image est composée de plusieurs objets : sphère, polygone,
personnages de la scène qui sont constitués de plusieurs objets, etc... Sur chaque élément, il faut effectuer plusieurs opérations pour le rendu : ajout des couleurs, effet d'éclairage, etc...
Chaque type d'opération (ajout des couleurs, effet d'éclairage) est implémenté par un visiteur. Chaque visiteur implémente une **méthode** spécifique (visiter un sphère, visiter un polygone) pour chaque type d'élément (sphère, polygone). Chaque élément (sphère) implémente un méthode d'acceptation de visiteur où il appelle la méthode spécifique de visite.

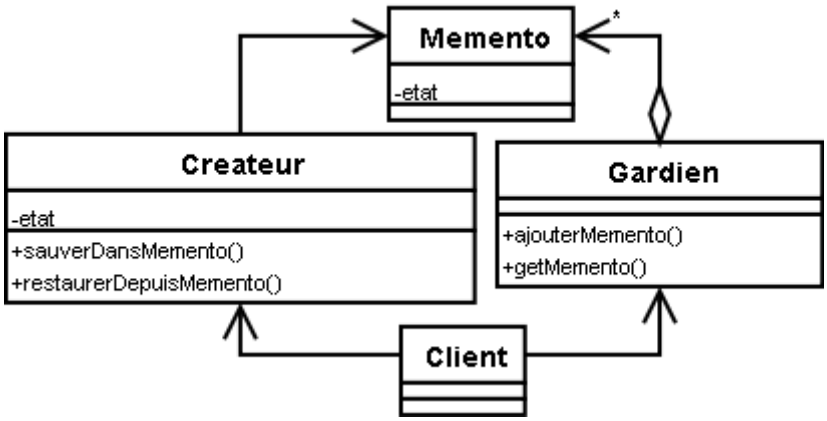
RESULTAT :
Le Design Pattern permet d'isoler les algorithmes appliqués sur des structures de données.

RESPONSABILITES :
• **Element** : définit l'interface d'une élément. Elle déclare la méthode de réception d'un objet **Visiteur**.
• **ConcreteElementA** et **ConcreteElementB** : sont des sous-classes concrètes de l'interface **Element**. Elles implémentent la méthode de réception. Elles possèdent des données/attributs et méthodes différents.
• **Visiteur** : définit l'interface d'une visiteur. Elle déclare les méthodes de visite des sous-classes concrètes de **Element**.
• **ConcreteVisiteurA** et **ConcreteVisiteurB** : sont des sous-classes concrètes de l'interface **Visiteur**. Elles implémentent des comportements de visite des **Element**.
• **Structure** : présente une interface de haut niveau permettant de visiter les objets **Element** la composant.
• La partie cliente appelle les méthodes de réception d'un **Visiteur** des **Element**.


VisiteurTerme
abstract public void visiterPlus(TermePlus arg1)
abstract public void visiterMoins(TermeMoins arg1)
abstract public void visiterConstante(TermeConstante arg1)
abstract public void visiterVariable(TermeVariable arg1)
VisiteurInfixe
VisiteurPostFixe
VisiteurVarDeclaree
VisiteurEvaluer
Expression
Terme
abstract public void accept(VisiteurTerme arg1)
TermePlus
public void accept(TermePlus arg1)
TermeVariable
public void accept(VisiteurTerme v)
TermeMoins
public void accept(VisiteurTerme arg1)
TermeConstante
public void accept(VisiteurTerme arg1)

Bonus

Memento (Memento)



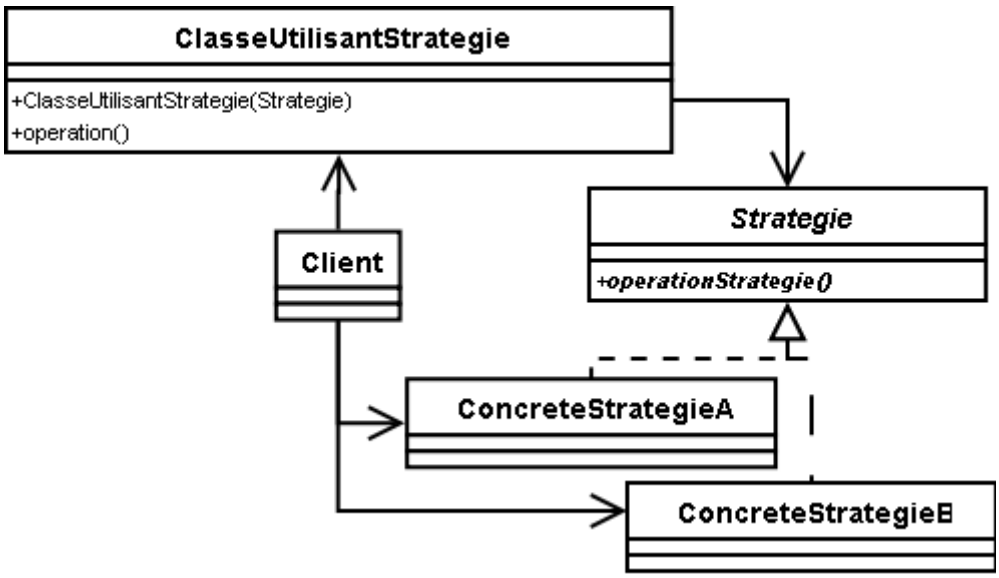
OBJECTIFS :
Sauvegarder l'**état interne** d'un objet en respectant l' **encapsulation**, afin de le restaurer plus tard.

RAISONS DE L'UTILISER :
Un système doit conserver et restaurer l'état d'un objet. L'état interne de l'objet à conserver n'est pas visible par les autres objets.
Cela peut être un éditeur de document disposant d'une fonction d'annulation. La fonction d'annulation est sur plusieurs niveaux.
Les informations de l'état interne (état du document) sont conservées dans un memento. L'objet avec l'état interne (document) est le créateur du memento. Afin de respecter l'encapsulation, les valeurs du memento ne sont visibles que par son créateur. Ainsi, l'encapsulation de l'état interne est préservée.
Un autre objet est chargé de conserver les mementos (gestionnaire d'annulation) : il s'agit du gardien.

RESULTAT :
Le Design Pattern permet d'isoler la conservation de l'état d'un objet.

RESPONSABILITES :
• **Memento** : contient la sauvegarde le l'état d'un objet. La classe doit autoriser l'accès aux informations seulement au **Createur**.
• **Createur** : sauvegarde son état dans un **Memento** ou restitue son état depuis un **Memento**.
• **Gardien** : conserve les **Memento** ou retourne un **Memento** conservé.
• La partie cliente demande au **Createur** de stocker son état dans un **Memento**. Elle demande au **Gardien** de conserver ce **Memento**. Elle peut alors demander au **Gardien** de lui fournir un des **Memento** conservés, ou bien elle demande au **Createur** de restituer son état depuis le **Memento**.

Stratégie (Strategy ou Policy)



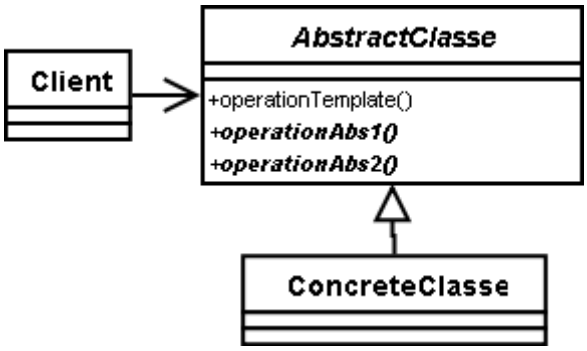
OBJECTIFS :
• Définir une famille d'**algorithmes** interchangeables.
• Permettre de les changer indépendamment de la partie cliente.

RAISONS DE L'UTILISER :
Un objet doit pouvoir faire varier une partie de son algorithme.
Cela peut être une liste triée. A chaque insertion, la liste place le nouvel élément à l'emplacement correspondant au tri. Le tri peut être alphabétique, inverse, les majuscules avant les miniscules, les minuscules avant, etc...
La partie de l'algorithme qui varie (le tri) est la stratégie. Toutes les stratégies présentent la même **interface**. La classe utilisant la stratégie (la liste) délègue la partie de traitement concernée à la stratégie.

RESULTAT :
Le Design Pattern permet d'isoler les algorithmes appartenant à une même famille d'algorithmes.

RESPONSABILITES :
• **Strategie** : définit l'interface commune des algorithmes.
• **ConcreteStrategieA** et **ConcreteStrategieB** : implémentent les méthodes d'algorithme.
• **ClasseUtilisantStrategie** : utilise un objet **Strategie**.
• La partie cliente configure un objet **ClasseUtilisantStrategie** avec un objet **Strategie** et appelle la méthode de **ClasseUtilisantStrategie** qui utilise la stratégie. Dans l'exemple, la configuration s'effectue par le constructeur, mais la configuration peut également s'effectuer par une méthode "setter".

Patron de méthode (Template Method)



OBJECTIFS :
Définir le squelette d'un **algorithme** en **déléguant** certaines étapes à des **sous-classes**.

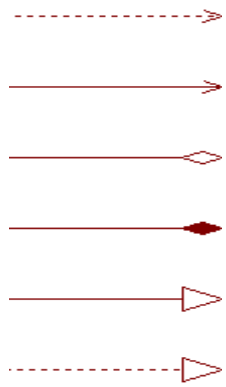
RAISONS DE L'UTILISER :
Une classe possède un fonctionnement global. Mais les détails de son algorithme doivent être spécifiques à ses sous-classes.
Cela peut être le cas d'un document informatique. Le document a un fonctionnement global où il est sauvegardé.
Pour la sauvegarde, il y aura toujours besoin d'ouvrir le fichier, d'écrire dedans, puis de fermer le fichier. Mais, selon le type de document, il ne sera pas sauvegardé de la même manière. S'il s'agit d'un document de traitement de texte,
il sera sauvegardé en suite d'octets. S'il s'agit d'un document HTML, il sera sauvegardé dans un fichier texte.
La partie générale de l'algorithme (sauvegarde) est gérée par la classe abstraite (document). La partie générale réalise l'ouverture, fermeture du fichier et appelle un méthode d'écriture. La partie spécifique de l'algorithme (écriture dans la fichier) est définie au niveau des classes concrètes (document de traitement de texte ou document HTML).

RESULTAT :
Le Design Pattern permet d'isoler les parties variables d'un algorithme.

RESPONSABILITES :
• **AbstractClasse** : définit des **méthodes** abstraites primitives. La classe implémente le squelette d'un algorithme qui appelle les méthodes primitives.
• **ConcreteClasse** : est une sous-classe concrète de AbstractClasse. Elle implémente les méthodes utilisées par l'algorithme de la méthode **operationTemplate()** de **AbstractClasse**.
• La partie cliente appelle la méthode de **AbstractClasse** qui définit l'algorithme.

Rappel utile

Symboles :



Dépendance

Association

Agrégation

Composition

Généralisation

Réalisation