**Design patterns**

**Introduction**

Un **design pattern** (**patron de conception** en français) est une solution générique à un problème récurrent dans le développement de logiciels. Ils sont souvent catégorisés en trois grandes familles : les *patterns de création*, les *patterns de structure* et les *patterns de comportement*.

1. **Les patterns de création**

Les **design patterns de création** sont des modèles de conception qui abordent la manière dont les objets sont instanciés et créés en général. Ils visent à fournir des mécanismes de création d'objets flexibles et efficaces, tout en favorisant la réutilisation du code. Nous pouvons en dénombrer 6 :

1. **Factory Method Pattern ou Modèle de méthode d’usine**

Ce pattern définit une interface pour la création d'objets, mais laisse les sous-classes décider des classes concrètes à instancier. En d'autres termes, il fournit une méthode pour créer des objets, mais la sous-classe est responsable de l'implémentation de cette méthode pour créer des instances spécifiques de classes. En utilisant ce patron, on peut encapsuler la logique de création d'objets dans une classe dédiée, ce qui permet de déléguer cette responsabilité aux sous-classes. Cela facilite l'extension du code, car il suffit de créer de nouvelles sous-classes pour créer de nouveaux types d'objets, sans avoir à modifier la classe créatrice. On l’utilise lorsque l’on veut assurer que la création d'instances de classes est gérée de manière centralisée, cohérente et flexible, permettant ainsi une meilleure maintenance et évolutivité de votre code.

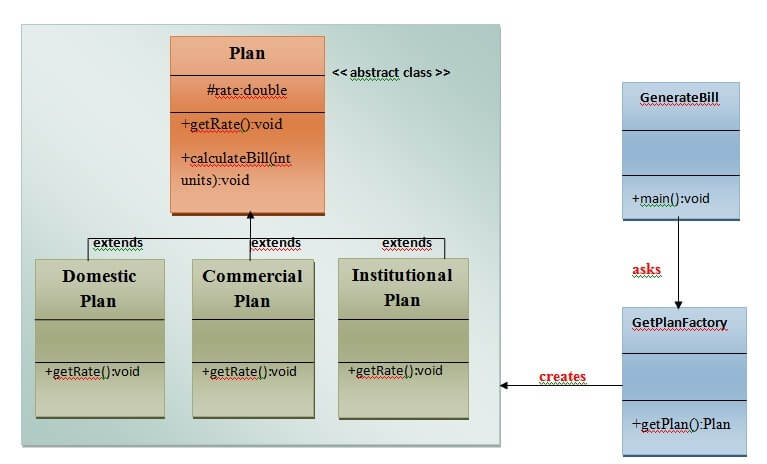
Les classes qui le constituent sont :

* **Product (Produit)** : Cette classe définit l'interface des objets que la Factory Method doit créer.
* **ConcreteProduct (Produit concret)** : Ce sont les classes concrètes qui implémentent l'interface définie par le Product. Chaque ConcreteProduct représente un type spécifique d'objet qui peut être créé par la Factory Method.
* **Creator (Créateur)** : La classe Creator déclare la Factory Method qui retourne un objet de type Product. C'est une classe abstraite qui peut également contenir du code partagé permettant de manipuler les produits générés.
* **ConcreteCreator (Créateur concret)** : Ce sont les sous-classes de Creator qui implémentent la Factory Method pour créer des instances spécifiques de ConcreteProduct.

Pour implémenter le Factory Method Pattern dans votre code, vous pouvez suivre les étapes suivantes :

* **Définir l'interface du produit (Product)** : Créez une interface ou une classe abstraite qui définit le contrat que tous les produits doivent respecter. Cette interface spécifie les méthodes que les produits doivent implémenter.
* **Implémenter le produit concret (ConcreteProduct)** : Créez une ou plusieurs classes qui implémentent l'interface du produit. Chaque classe représente une implémentation spécifique du produit.
* **Créer la classe créatrice abstraite (Creator)** : Définissez une classe abstraite qui servira de classe créatrice. Cette classe doit déclarer la méthode factoryMethod(), qui est responsable de la création d'un objet de type Product. Cette classe peut également inclure d'autres méthodes et fonctionnalités liées à l'utilisation des produits.
* **Implémenter la classe créatrice concrète (ConcreteCreator)** : Créez une ou plusieurs sous-classes de la classe créatrice abstraite. Chaque sous-classe doit implémenter la méthode factoryMethod() en fournissant une logique de création spécifique pour créer un objet de type concret qui implémente l'interface du produit.
* **Utiliser le Factory Method** : Dans votre code client, créez une instance de la classe créatrice concrète (ConcreteCreator) et utilisez sa méthode factoryMethod() pour obtenir un objet du produit. Vous pouvez ensuite utiliser cet objet pour effectuer des opérations spécifiques.

Son diagramme de classe est :



1. **Abstract Factory Pattern ou Modèle d’usine abstrait**

Il fournit une interface pour créer des familles d'objets liés ou dépendants sans spécifier leurs classes concrètes. Le principe de base de l’Abstract Factory Pattern est de définir une interface abstraite (appelée l'usine abstraite) qui déclare des méthodes pour créer différents types d'objets. Chaque sous-classe de l'usine abstraite implémente ces méthodes pour créer des objets spécifiques. Ainsi, le choix de l'usine abstraite détermine les types d'objets concrets créés. Ce patron est utile lorsque vous voulez créer des objets qui sont liés entre eux ou qui dépendent les uns des autres.

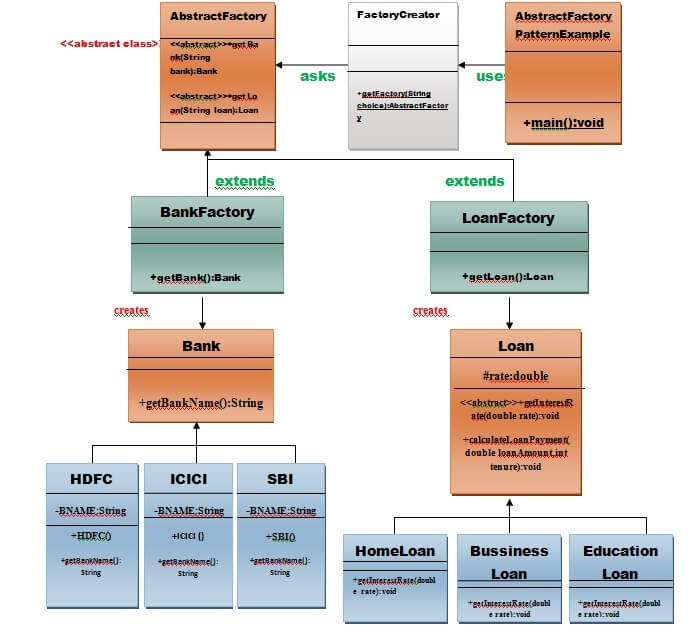
Les classes qui la constituent sont :

* **Product (Produit)** : C'est l'interface commune pour les objets que la Factory Method doit créer. Les différentes implémentations concrètes de Product seront créées par les classes concrètes de la Factory.
* **ConcreteProduct (Produit concret)** : Ce sont les classes concrètes qui implémentent l'interface Product. Chaque ConcreteProduct représente un type spécifique d'objet que la Factory Method peut créer.
* **Creator (Créateur)** : Cette classe abstraite définit une méthode de fabrication (Factory Method) qui retourne un objet de type Product. La classe Creator peut également contenir du code partagé qui manipule les produits créés.
* **ConcreteCreator (Créateur concret)** : Ce sont les sous-classes de Creator qui implémentent la Factory Method pour créer des instances spécifiques de ConcreteProduct. Chaque ConcreteCreator est associé à un type spécifique de ConcreteProduct.

Pour implémenter le design pattern Factory Method dans un code, vous pouvez suivre les étapes suivantes :

* **Définir l'interface du produit (Product)** : Créez une interface ou une classe abstraite qui définit les méthodes que tous les produits concrets devront implémenter. Cette interface représente le type abstrait des objets que la Factory Method pourra créer.
* **Implémenter les produits concrets (Concrete Products)** : Créez des classes concrètes qui implémentent l'interface du produit. Chaque classe concrète représente un type spécifique d'objet qui sera produit par la Factory Method.
* **Créer la classe créatrice abstraite (Creator)** : Définissez une classe abstraite qui contient la méthode de fabrication abstraite (Factory Method) pour créer des objets de type produit. Cette classe peut également contenir d'autres méthodes utilitaires ou du code partagé.
* **Implémenter les classes créatrices concrètes (Concrete Creators)** : Créez des classes concrètes qui étendent la classe créatrice abstraite. Chaque classe créatrice concrète implémente la Factory Method pour créer des instances spécifiques de produits concrets.

Son diagramme de classe est :



1. **Singleton design pattern ou Modèle de conception Singleton**

Le Singleton design pattern (ou Modèle de conception Singleton) est un modèle de conception creational qui garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance. Il est utilisé dans les cas où vous avez besoin de garantir qu'une classe n'a qu'une seule instance dans tout votre programme et que cette instance est accessible de manière globale.

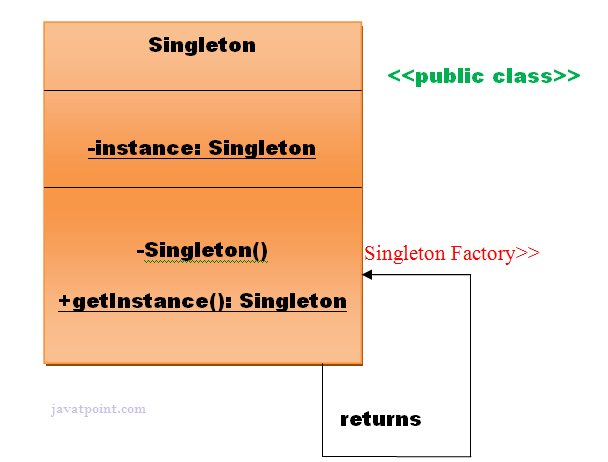
Elle est constituée de :

* **Classe Singleton** : Il s'agit de la classe pour laquelle vous voulez garantir l'existence d'une seule instance. La classe Singleton a un constructeur privé pour empêcher la création d'instances supplémentaires en dehors de la classe elle-même.

Voici les étapes pour implémenter le Singleton design pattern :

* **Constructeur privé** : Pour empêcher l'instanciation directe de la classe, le constructeur de la classe Singleton doit être déclaré privé.
* **Variable statique privée** : Déclarez une variable statique privée qui stockera l'unique instance de la classe Singleton.
* **Méthode statique publique getInstance()** : Fournissez une méthode statique publique nommée getInstance() qui renvoie l'instance unique de la classe Singleton. Cette méthode est généralement responsable de la création de l'instance si elle n'existe pas encore.

Son diagramme de classe est :



1. **Prototype Design Pattern ou Modèle de conception de prototype**

Le Prototype Design Pattern (ou Modèle de conception de prototype) est un modèle de conception creational qui permet de créer de nouveaux objets en clonant un objet existant plutôt qu'en les instanciant à partir de zéro. Cela permet de réduire la complexité liée à la création d'objets en évitant la duplication de code et en améliorant les performances. On l’utilise pour cloner un objet existant et ainsi éviter les coûts liés à l'initialisation.

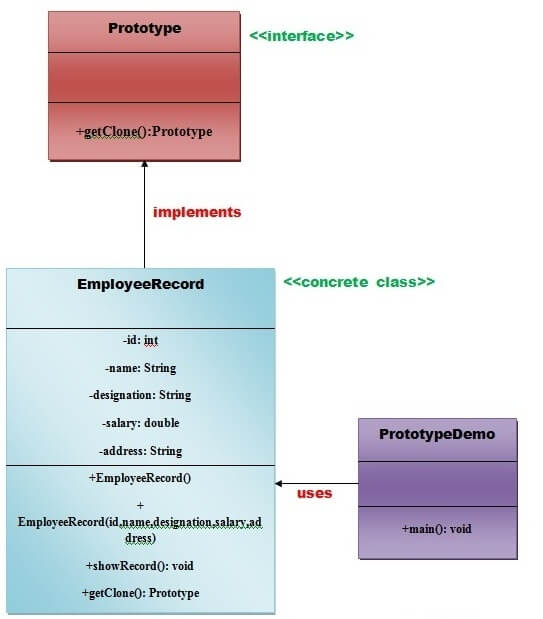
Voici les composants principaux du Prototype Design Pattern :

* **Prototype** : C'est l'interface ou la classe abstraite qui déclare une méthode pour cloner l'objet. Toutes les classes concrètes qui implémentent cette interface peuvent être clonées.
* **Classe concrète** : Cette classe implémente l'interface Prototype et définit la logique de clonage pour créer une copie de l'objet.
* **Client** : C'est la classe qui utilise le prototype pour créer de nouveaux objets. Le client demande au prototype de créer une copie, au lieu de créer une nouvelle instance directement.

Voici comment fonctionne le Prototype Design Pattern :

* **Interface ou classe abstraite** : Définissez une interface ou une classe abstraite qui déclare une méthode de clonage. Cette méthode sera implémentée par les classes concrètes qui souhaitent être clonables.
* **Classe concrète** : Implémentez la méthode de clonage dans les classes concrètes en effectuant une copie profonde (deep copy) de l'objet. Une copie profonde crée une nouvelle instance de l'objet et copie les valeurs de tous ses attributs.
* **Gestionnaire de prototypes** : Créez une classe qui agit comme un gestionnaire de prototypes et qui maintient une collection de prototypes. Cette classe fournit une méthode pour enregistrer et récupérer les prototypes.
* **Utilisation du prototype** : Utilisez le gestionnaire de prototypes pour obtenir des clones d'objets existants au lieu de les instancier directement.

Son diagramme de classe est :



1. **Builder Design Pattern ou Modèle de conception du constructeur**

Il sépare le processus de construction d'un objet de sa représentation finale, ce qui permet de créer différentes représentations en utilisant le même processus de construction. On l’utilise l’on doit créer des objets complexes qui nécessitent plusieurs étapes de configuration.

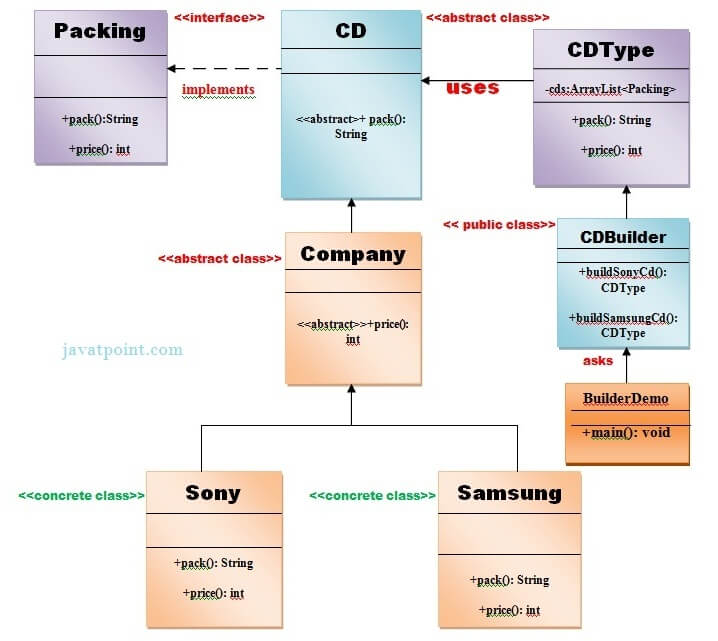
Le Builder Design Pattern est composé des classes suivantes :

* **Interface du Builder (ou classe abstraite)** : Définit les méthodes nécessaires pour construire les parties individuelles de l'objet et pour obtenir le produit final.
* **Implémentation concrète du Builder** : Fournit une implémentation concrète des méthodes définies dans l'interface du Builder. Chaque méthode correspond à une étape de construction spécifique.
* **Classe du produit** : Représente l'objet complexe que vous souhaitez construire. Elle a généralement un constructeur privé et des méthodes pour configurer les différentes parties de l'objet.
* **Directeur (optionnel)** : Facilite la construction de l'objet en utilisant le Builder. Il définit l'ordre des étapes de construction et appelle les méthodes du Builder dans cet ordre spécifié.

Voici comment fonctionne le Builder Design Pattern :

* **Interface ou classe abstraite du Builder** : Définissez une interface ou une classe abstraite qui déclare les étapes de construction de l'objet. Chaque étape est représentée par une méthode dans le Builder.
* **Implémentation concrète du Builder** : Implémentez le Builder en fournissant une classe concrète qui implémente les méthodes de construction définies dans l'interface du Builder. Chaque méthode peut modifier l'état interne du Builder pour représenter une étape spécifique de la construction.
* **Classe du produit** : Définissez la classe de produit que vous souhaitez construire. Cette classe doit avoir un constructeur privé et des méthodes pour configurer ses différentes parties.
* **Directeur** : En option, vous pouvez utiliser un directeur (Director) pour définir l'ordre des étapes de construction et pour appeler les méthodes du Builder dans cet ordre spécifié.
* **Utilisation du Builder** : Utilisez le Builder pour créer l'objet étape par étape en appelant les méthodes de construction appropriées. Une fois que toutes les étapes sont effectuées, le Builder renvoie l'objet final construit.

Son diagramme de classe est :



1. **Object Pool Pattern ou Modèle de pool d’objets**

Le Modèle de pool d'objets (Object Pool Pattern) est un modèle de conception qui vise à améliorer les performances et l'efficacité en réutilisant les objets au lieu de les recréer à chaque fois. Il est particulièrement utile dans les systèmes où la création d'objets coûteuse en ressources, telle que l'allocation et la désallocation de mémoire, peut entraîner une dégradation des performances.

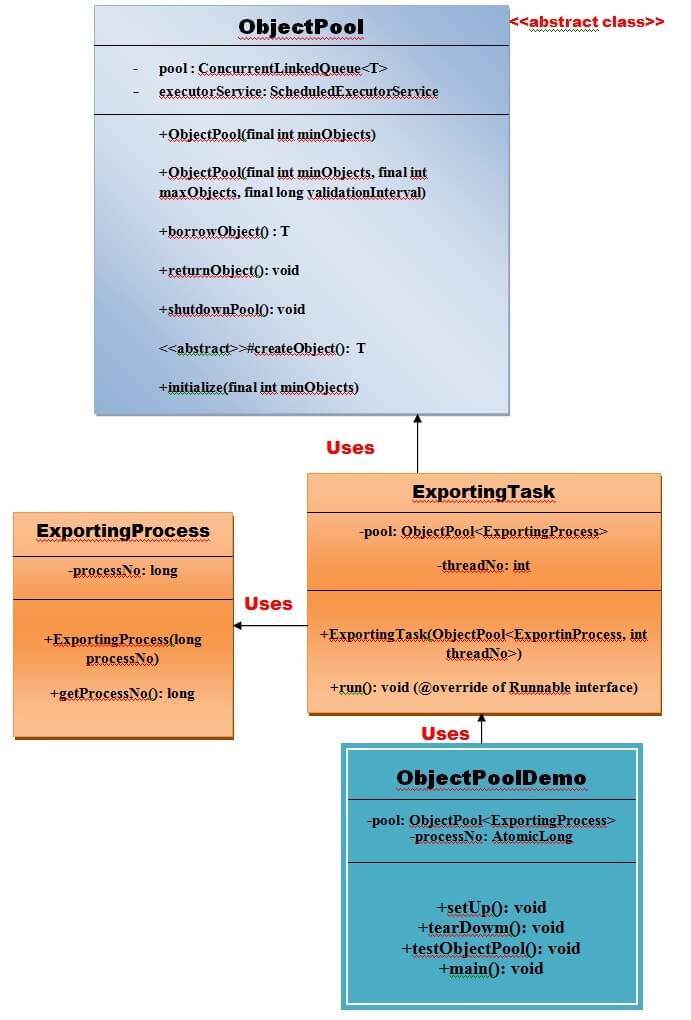
Le Modèle de pool d'objets comprend les éléments suivants

* **Pool d'objets** : C'est une collection d'objets préalablement créés et prêts à être utilisés. Ces objets sont stockés dans une structure de données telle qu'une liste ou une file d'attente.
* **Méthode d'emprunt** : Cette méthode est utilisée par les clients pour obtenir un objet du pool. Si le pool a des objets disponibles, elle en retourne un. Sinon, elle peut choisir de créer un nouvel objet ou d'attendre qu'un objet soit retourné au pool.
* **Méthode de retour** : Lorsqu'un client a terminé d'utiliser un objet, il le retourne au pool en appelant cette méthode. L'objet est alors marqué comme disponible et peut être réutilisé par d'autres clients.
* **Taille maximale du pool** : Il est possible de spécifier une limite maximale de la taille du pool. Si cette limite est atteinte et qu'un client souhaite emprunter un objet, il peut être mis en attente jusqu'à ce qu'un objet soit retourné au pool.

Pour implémenter le Modèle de pool d'objets dans votre code, vous pouvez suivre les étapes suivantes :

* Créez la classe d'objet que vous souhaitez mettre dans le pool.
* Créez une classe pour le pool d'objets.
* Implémentez les méthodes d'emprunt et de retour dans la classe du pool d'objets.
* Gérez la synchronisation si plusieurs threads peuvent accéder simultanément au pool d'objets.
* Utilisez le pool d'objets dans votre code.
* Assurez-vous de gérer correctement les cas où tous les objets du pool sont en cours d'utilisation.

Son diagramme de classe est :



1. **Les patterns de structure**

Les **patterns de structure** sont des modèles de conception qui se concentrent sur la composition, la relation et la structuration des classes et des objets pour former des structures plus complexes. Ils permettent d'organiser et de gérer les relations entre les entités du système. Nous pouvons en dénombrer 7 :

1. **Adapter Pattern ou Modèle d’adaptateur**

Il permet d'adapter l'interface d'une classe existante pour qu'elle puisse être utilisée par une autre classe incompatible. Le modèle d'adaptateur résout le problème de l'incompatibilité des interfaces en fournissant une classe intermédiaire appelée "adaptateur" entre deux classes incompatibles. Il permet de réutiliser des classes existantes sans les modifier, ce qui est particulièrement utile lorsque vous travaillez avec du code tiers ou du code hérité. Il facilite également la communication entre des classes qui n'ont pas été conçues pour interagir les unes avec les autres.

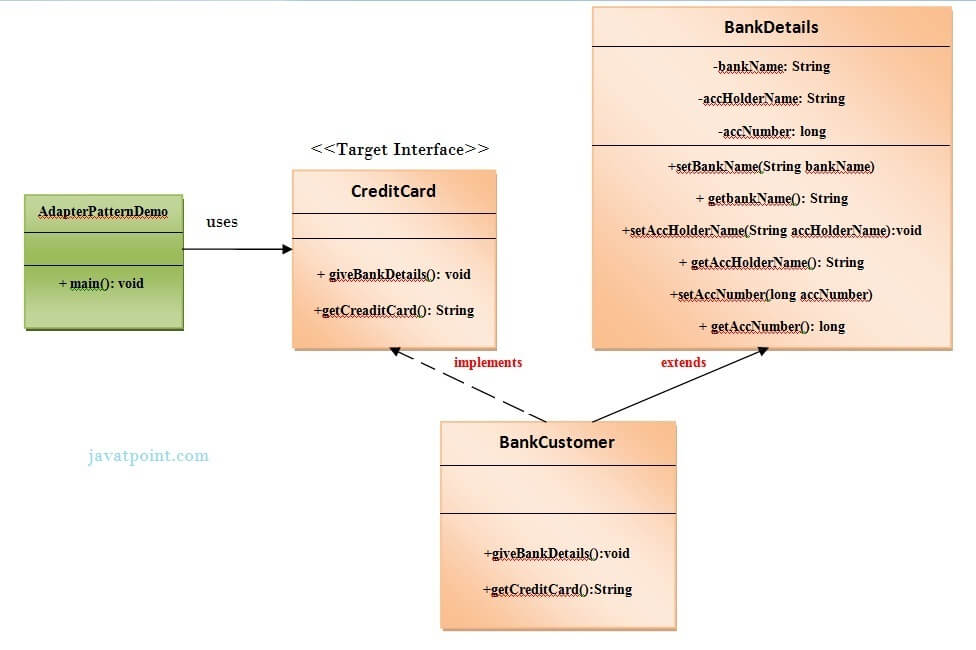
Le modèle d'adaptateur est composé de plusieurs classes :

* **L'interface cible (Target)** : Il s'agit de l'interface que la classe cliente utilise pour interagir avec l'adaptateur. C'est l'interface que l'adaptateur doit implémenter pour être compatible avec la classe cliente.
* **La classe adapteur (Adapter)** : C'est la classe qui sert d'intermédiaire entre la classe cliente et la classe à adapter (adaptee). Elle implémente l'interface cible et utilise l'objet de la classe adaptee pour effectuer les opérations nécessaires.
* **La classe adaptee (Adaptee)** : C'est la classe qui doit être adaptée pour être utilisée par la classe cliente. Elle peut avoir une interface incompatible avec celle attendue par la classe cliente.

Pour implémenter le modèle d'adaptateur dans votre code, vous pouvez suivre ces étapes :

* **Définir l'interface cible** : Créez une interface qui représente l'interface que la classe cliente utilise pour interagir avec l'adaptateur. Cette interface devrait décrire les méthodes que la classe cliente s'attend à utiliser.
* **Implémenter la classe adapteur** : Créez une classe qui sert d'intermédiaire entre la classe cliente et la classe adaptee. Cette classe doit implémenter l'interface cible définie à l'étape précédente. Elle encapsulera un objet de la classe adaptee et utilisera cet objet pour effectuer les opérations nécessaires dans les méthodes de l'interface cible.
* **Implémenter la classe adaptee** : Créez la classe qui doit être adaptée pour être utilisée par la classe cliente. Cette classe peut avoir une interface incompatible avec celle attendue par la classe cliente.
* **Dans la classe adapteur**, définissez une association avec la classe adaptee : Cela peut être réalisé en utilisant la composition (par exemple, en déclarant un objet de la classe adaptee comme membre de la classe adapteur) ou en héritant de la classe adaptee si vous implémentez un adaptateur de classe.
* **Implémenter les méthodes de l'interface cible dans la classe adapteur** : Dans ces méthodes, appelez les méthodes appropriées de la classe adaptee pour effectuer les opérations nécessaires. L'adaptateur traduit les appels de l'interface cible en appels appropriés sur la classe adaptee.
* **Utiliser l'adaptateur** : Dans votre code client, utilisez l'interface cible pour interagir avec l'adaptateur. La classe cliente n'a pas besoin de connaître les détails de la classe adaptee, car elle communique uniquement avec l'adaptateur via l'interface cible.

Son diagramme de classe est :



1. **Bridge Pattern ou Modèle de pont**

Il est utilisé lorsque vous voulez éviter une liaison permanente entre l'abstraction et l'implémentation, permettant ainsi de les modifier indépendamment l'une de l'autre. Le modèle de pont utilise une structure de pont entre les classes abstraites et les classes concrètes. Il se compose de deux hiérarchies de classes : la hiérarchie de l'abstraction et la hiérarchie de l'implémentation.

*La hiérarchie de l'abstraction* définit l'interface abstraite que la classe cliente utilise pour interagir avec l'objet. Elle peut contenir des méthodes abstraites et des méthodes concrètes.

La *hiérarchie de l'implémentation* définit les classes concrètes qui implémentent les détails spécifiques de l'abstraction. Cette hiérarchie peut également contenir des méthodes abstraites et des méthodes concrètes.

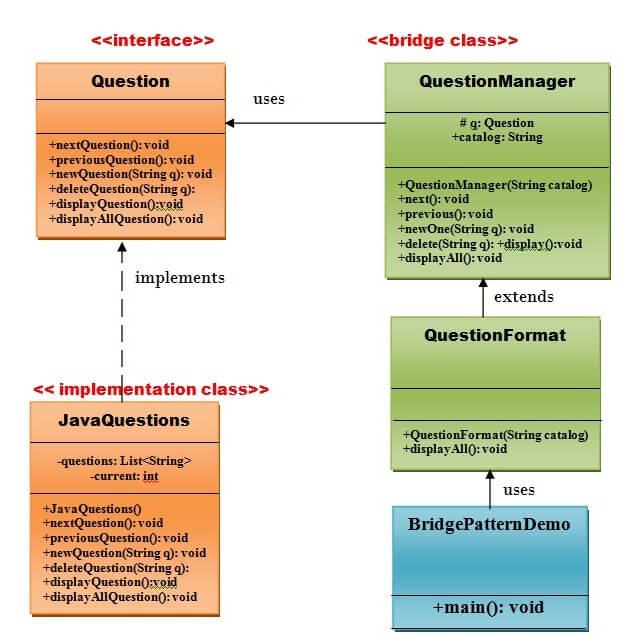
Dans le modèle de pont, les classes qui le constituent sont :

* **Abstraction** : C'est la classe abstraite qui définit l'interface abstraite que la classe cliente utilise pour interagir avec l'objet. Elle contient généralement des méthodes abstraites et des méthodes concrètes qui utilisent l'objet d'implémentation. Dans l'exemple précédent, la classe Shape est l'abstraction.
* **Implémentation** : C'est l'interface ou la classe abstraite qui définit l'interface de l'implémentation. Elle est utilisée par l'abstraction pour appeler les méthodes spécifiques de l'implémentation. Dans l'exemple précédent, l'interface DrawAPI est l'implémentation.
* **Implémentations concrètes** : Ce sont les classes concrètes qui implémentent l'interface de l'implémentation. Elles fournissent les détails spécifiques de l'implémentation. Dans l'exemple précédent, les classes DrawAPI1 et DrawAPI2 sont des implémentations concrètes de l'interface DrawAPI.
* **Raffinements de l'abstraction** : Ce sont les classes qui étendent l'abstraction et ajoutent des fonctionnalités spécifiques. Elles peuvent ajouter des méthodes supplémentaires ou modifier les méthodes héritées. Dans l'exemple précédent, la classe Circle est un raffinement de l'abstraction Shape.

Les étapes pour implémenter le modèle de pont dans votre code sont les suivantes :

* **Définir l'abstraction** : Créez une classe abstraite qui définit l'interface abstraite que la classe cliente utilise pour interagir avec l'objet. Cette classe peut contenir des méthodes abstraites et des méthodes concrètes qui utilisent l'objet d'implémentation.
* **Définir l'implémentation** : Créez une interface ou une classe abstraite qui définit l'interface de l'implémentation. Cette interface ou classe abstraite est utilisée par l'abstraction pour appeler les méthodes spécifiques de l'implémentation.
* **Implémenter les classes d'implémentation concrètes** : Créez des classes qui implémentent l'interface d'implémentation. Ces classes fournissent les détails spécifiques de l'implémentation.
* **Utiliser le pont** : Dans votre code client, créez une instance de l'abstraction et de l'implémentation souhaitées, puis utilisez-les pour interagir avec l'objet.

Son diagramme de classe est :



1. **Composite Pattern ou Motif composite**

Il permet de créer une structure arborescente d'objets où les objets individuels et les groupes d'objets sont traités de manière uniforme. Le motif Composite utilise une structure hiérarchique composée d'objets composants. Le composant de base est une interface ou une classe abstraite qui définit les opérations communes pour tous les objets, qu'ils soient individuels ou des groupes. Les objets individuels sont appelés "feuilles" (leaf), tandis que les groupes d'objets sont appelés "composites".

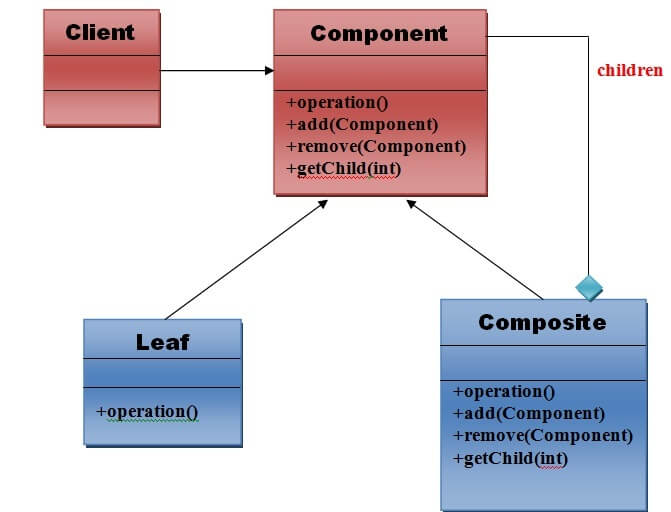
Dans le motif Composite, les classes qui le constituent sont les suivantes :

* **Composant (Component)** : C'est l'interface ou la classe abstraite qui définit les opérations communes pour tous les objets, qu'ils soient individuels (feuilles) ou des groupes (composites). Elle peut contenir des méthodes pour effectuer des opérations sur les objets et pour gérer les objets enfants tels que l'ajout, la suppression et la récupération.
* **Feuille (Leaf)** : C'est la classe qui représente les objets individuels. Elle implémente le composant et fournit une implémentation spécifique des méthodes communes. Les feuilles ne peuvent pas avoir d'enfants.
* **Composite** : C'est la classe qui représente les groupes d'objets. Elle implémente le composant et fournit une implémentation spécifique des méthodes communes, en plus de gérer les objets enfants. Les composites peuvent contenir à la fois des feuilles et d'autres composites.

Les étapes pour implémenter le motif Composite dans votre code sont les suivantes :

* **Définir le composant de base** : Créez une interface ou une classe abstraite qui définit les opérations communes pour tous les objets, qu'ils soient individuels ou des groupes. Cette interface ou classe abstraite peut contenir des méthodes pour effectuer des opérations sur les objets et pour gérer les objets enfants (ajout, suppression, récupération).
* **Implémenter les feuilles** : Créez des classes qui représentent les objets individuels. Ces classes implémentent le composant de base et fournissent une implémentation spécifique des méthodes communes.
* **Implémenter les composites** : Créez des classes qui représentent les groupes d'objets. Ces classes implémentent le composant de base et fournissent une implémentation spécifique des méthodes communes, en plus de gérer les objets enfants. Les composites peuvent contenir à la fois des feuilles et d'autres composites.
* **Utiliser le motif Composite** : Dans votre code client, créez une structure arborescente d'objets en utilisant les feuilles et les composites. Traitez les objets individuels et les groupes d'objets de manière uniforme en utilisant les méthodes définies dans le composant de base.

Son diagramme de classe est :



1. **Decorator Pattern ou Motif de décorateur**

Le motif de décorateur, également connu sous le nom de "Decorator Pattern" en anglais, est un motif de conception structurel qui permet d'ajouter de nouvelles fonctionnalités à un objet existant de manière dynamique, sans modifier sa structure de base. Il permet d'étendre les fonctionnalités d'un objet en l'enveloppant dans des décorateurs qui ajoutent des comportements supplémentaires. Le motif de décorateur utilise une structure en couches où chaque couche représente un décorateur. Les décorateurs implémentent une interface commune avec l'objet à décorer, ce qui leur permet d'être substituables. L'objet à décorer peut-être une classe concrète ou une interface.

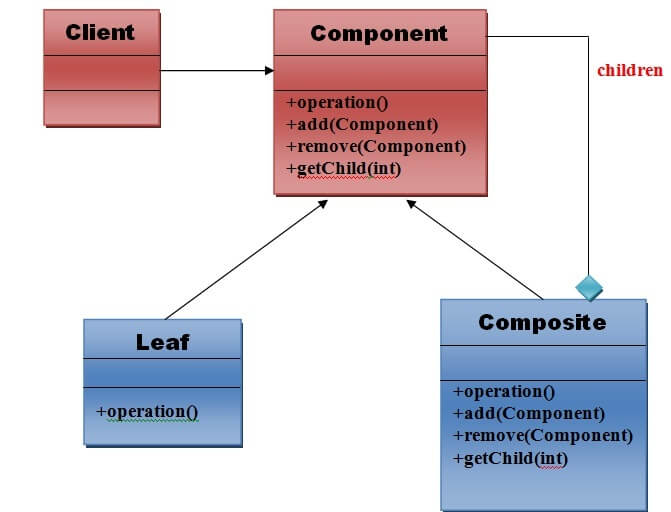
Dans le motif de décorateur, les classes qui le constituent sont les suivantes :

* **Composant (Component)** : C'est l'interface ou la classe abstraite qui définit l'interface commune pour les objets concrets et les décorateurs. Elle expose la méthode principale à décorer.
* **Composant concret (ConcreteComponent)** : C'est la classe de base qui fournit l'implémentation de base de l'objet à décorer. Elle implémente l'interface ou la classe de composant de base.
* **Décorateur (Decorator)** : C'est la classe abstraite qui étend l'interface ou la classe de composant de base. Elle contient une référence à un objet de composant de base et implémente l'interface commune. Elle peut ajouter des fonctionnalités supplémentaires avant ou après l'appel de la méthode principale.
* **Décorateur concret (ConcreteDecorator)** : Ce sont les classes qui étendent la classe de décorateur et ajoutent des fonctionnalités spécifiques. Elles implémentent les méthodes supplémentaires et peuvent appeler les méthodes du décorateur de base avant ou après l'appel de la méthode principale.

Voici les étapes nécessaires pour implémenter le motif de décorateur dans un code :

* **Définir une interface ou une classe de composant de base** : Cette étape consiste à définir une interface ou une classe abstraite qui définit l'interface commune pour les objets concrets et les décorateurs. Cette interface ou cette classe contient la méthode principale que tous les composants et décorateurs doivent implémenter.
* **Implémenter une classe de composant concret** : Dans cette étape, vous devez créer une classe qui implémente l'interface ou la classe de composant de base. Cette classe fournit l'implémentation de base de l'objet à décorer.
* **Implémenter une classe de décorateur abstraite** : Cette étape consiste à créer une classe abstraite qui étend l'interface ou la classe de composant de base. Cette classe abstraite contient une référence à un objet de composant de base et implémente l'interface commune. Elle peut également ajouter des méthodes supplémentaires pour définir les fonctionnalités supplémentaires que les décorateurs peuvent fournir.
* **Implémenter des classes de décorateur concret** : Dans cette étape, vous devez créer des classes qui étendent la classe de décorateur abstraite. Chaque classe de décorateur concret ajoute une fonctionnalité spécifique à l'objet de composant de base. Ces classes doivent implémenter les méthodes supplémentaires définies dans la classe de décorateur abstraite et peuvent appeler les méthodes du décorateur de base avant ou après l'appel de la méthode principale.
* **Utiliser le motif de décorateur** : Enfin, vous pouvez utiliser le motif de décorateur en créant une instance de l'objet de composant de base, puis en l'enveloppant avec des instances de décorateurs concrets pour ajouter des fonctionnalités supplémentaires. Vous pouvez chaîner plusieurs décorateurs pour ajouter plusieurs couches de fonctionnalités.

Son diagramme de classe est :



1. **Facade Pattern ou Motif de façade**

Il permet de regrouper un ensemble de classes et de fournir une interface de haut niveau plus conviviale pour les clients. L'objectif principal du motif de façade est de simplifier l'utilisation d'un sous-système en fournissant une interface unifiée. Il permet aux clients d'interagir avec le sous-système sans avoir à connaître les détails de mise en œuvre de chaque classe qui le compose.

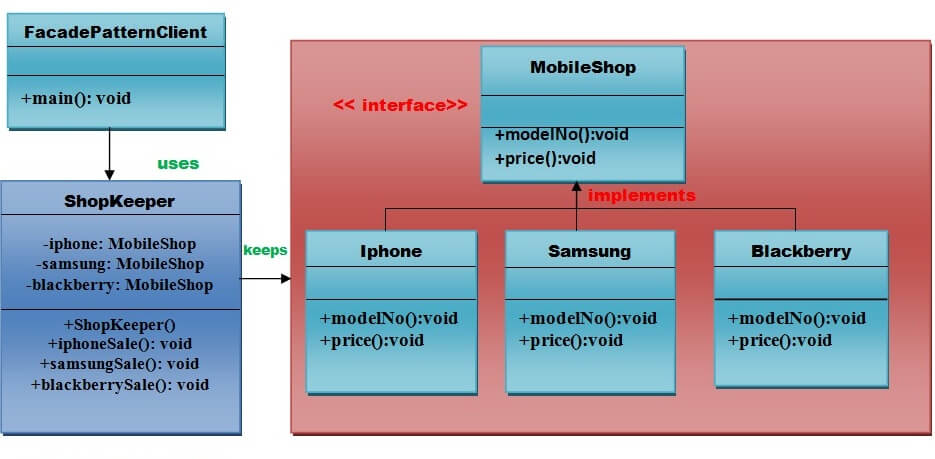
Dans le motif de façade (Facade Pattern), les classes impliquées sont les suivantes :

* **Classe de façade (Facade class)** : Cette classe est responsable de fournir une interface simplifiée pour les clients afin d'interagir avec le sous-système. Elle agit comme une interface de haut niveau pour le sous-système.
* **Classes du sous-système (Subsystem classes)** : Ce sont les classes qui composent le sous-système complexe. Chaque classe du sous-système a une responsabilité spécifique et peut effectuer des opérations complexes. Les classes du sous-système interagissent entre elles pour accomplir des tâches plus complexes.

Voici les étapes pour implémenter le motif de façade :

* **Identifier les classes du sous-système** : La première étape consiste à identifier les classes du sous-système qui sont complexes et nécessitent une interface simplifiée pour les clients.
* **Concevoir une classe de façade** : La classe de façade est responsable de fournir une interface simplifiée pour les clients. Elle agit comme une interface de haut niveau pour interagir avec le sous-système.
* **Implémenter les méthodes de la classe de façade** : Dans cette étape, vous devez implémenter les méthodes de la classe de façade. Ces méthodes doivent appeler les méthodes appropriées des classes du sous-système pour effectuer les opérations souhaitées.
* **Utiliser la classe de façade** : Les clients doivent utiliser la classe de façade pour interagir avec le sous-système. Ils n'ont pas besoin de connaître les détails de mise en œuvre des classes du sous-système.

Son diagramme de classe est :



1. **Flyweight Pattern ou Patron poids mouche**

Le patron de conception poids mouche (Flyweight Pattern) est un patron de conception structurel qui vise à optimiser les performances et la consommation de mémoire en partageant efficacement les objets qui ont une granularité fine. Le principe du patron poids mouche est de diviser les objets en deux parties : une partie intrinsèque et une partie extrinsèque. La partie intrinsèque contient les attributs qui sont communs à plusieurs objets et peuvent être partagés, tandis que la partie extrinsèque contient les attributs spécifiques à chaque objet.

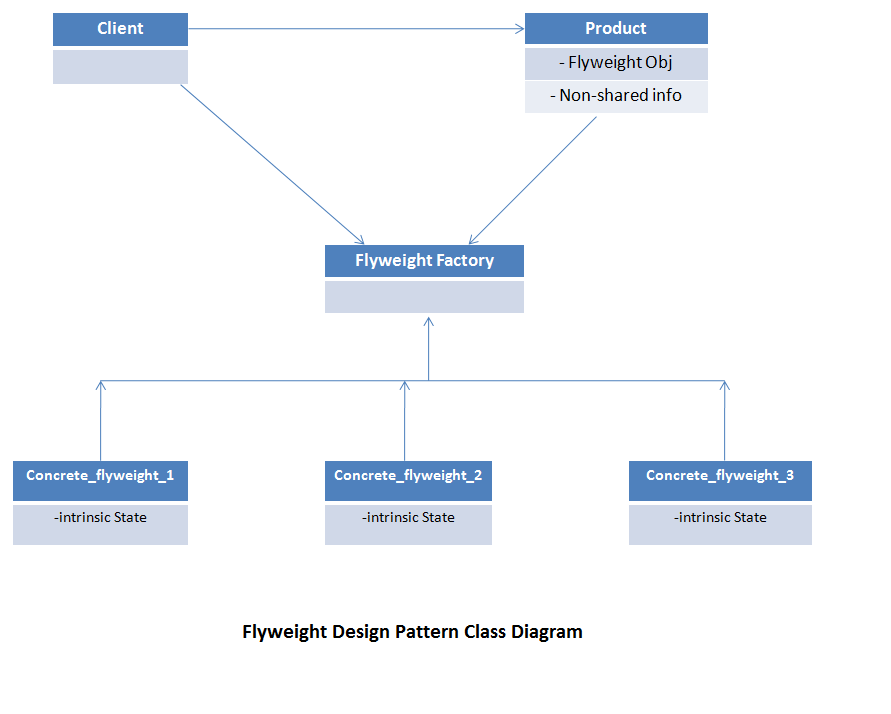
Dans le modèle de conception du poids mouche (Flyweight Pattern), les classes impliquées sont les suivantes :

* **Interface Flyweight** : C'est une interface qui définit les méthodes que les objets poids mouche doivent implémenter. Ces méthodes peuvent inclure des opérations qui prennent un état extrinsèque en tant que paramètre.
* **Classe ConcreteFlyweight** : Cette classe est une implémentation concrète de l'interface Flyweight. Elle définit l'état intrinsèque qui est partagé entre plusieurs objets poids mouche. Elle implémente également les méthodes de l'interface Flyweight.
* **Classe FlyweightFactory** : C'est une classe qui gère la création et le partage des objets poids mouche. Elle maintient une collection (par exemple une map) d'objets déjà créés et fournit des méthodes pour récupérer un objet poids mouche existant ou en créer un nouveau si nécessaire.

L'implémentation du patron de conception poids mouche (Flyweight Pattern) suit généralement les étapes suivantes :

* **Identifier les parties intrinsèques et extrinsèques** : Analysez les objets de votre application et identifiez les parties qui sont communes à plusieurs objets (partie intrinsèque) et celles qui sont spécifiques à chaque objet (partie extrinsèque).
* **Créer l'interface Flyweight** : Définissez une interface qui déclare les méthodes nécessaires pour les objets poids mouche. Ces méthodes peuvent inclure des opérations qui prennent un état extrinsèque en tant que paramètre.
* **Implémenter les classes concrètes Flyweight** : Créez des classes concrètes qui implémentent l'interface Flyweight. Chaque classe concrète représente un type spécifique d'objet poids mouche. Assurez-vous de définir les attributs correspondant à la partie intrinsèque de l'objet.
* **Créer la classe FlyweightFactory** : Cette classe est responsable de la création et du partage des objets poids mouche. Elle peut utiliser une structure de données comme une map pour stocker les objets créés. La FlyweightFactory fournit des méthodes pour récupérer un objet poids mouche existant ou en créer un nouveau si nécessaire.
* **Utiliser les objets Flyweight dans le client** : Dans le code client, utilisez la FlyweightFactory pour obtenir des objets poids mouche. Vous pouvez appeler leurs méthodes en passant un état extrinsèque spécifique. Assurez-vous de comprendre comment l'état extrinsèque est géré par les objets poids mouche et comment il affecte leur comportement.

Son diagramme de classe est :



1. **Proxy Pattern ou Modèle de proxy**

Le patron de conception Proxy, également connu sous le nom de Modèle de proxy, est un patron structurel qui fournit un substitut ou un représentant d'un objet pour contrôler l'accès à celui-ci. Il agit comme une interface intermédiaire entre le client et l'objet réel, en permettant d'ajouter des fonctionnalités supplémentaires ou de contrôler l'accès à l'objet réel.

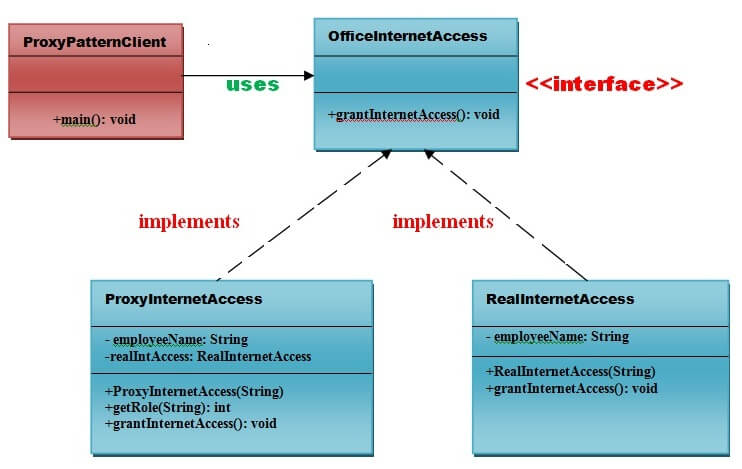
Voici les classes impliquées dans le modèle de proxy :

* **Interface Subject** : C'est l'interface commune entre le proxy et l'objet réel. Elle définit les méthodes que le proxy et l'objet réel doivent implémenter.
* **Classe RealSubject** : C'est la classe qui représente l'objet réel. Elle implémente l'interface Subject et contient la logique métier réelle.
* **Classe Proxy** : C'est la classe qui agit comme un substitut de l'objet réel. Elle implémente l'interface Subject et contient une référence à l'objet réel. Le proxy peut effectuer certaines tâches supplémentaires avant, après ou à la place d'appeler les méthodes de l'objet réel.

L'implémentation du patron de conception Proxy suit généralement les étapes suivantes :

* **Définir l'interface commune** : Créez une interface qui sera implémentée par le proxy et l'objet réel. Cette interface définit les méthodes que le client peut utiliser pour interagir avec l'objet réel.
* **Implémenter la classe de l'objet réel** : Créez une classe qui représente l'objet réel. Cette classe implémente l'interface définie à l'étape précédente et contient la logique métier réelle.
* **Implémenter la classe du proxy** : Créez une classe qui agit comme un substitut de l'objet réel. Cette classe implémente également l'interface définie à l'étape 1. La classe du proxy contient généralement une référence à l'objet réel et peut effectuer des tâches supplémentaires avant ou après l'appel des méthodes de l'objet réel.
* **Utiliser le proxy dans le client** : Dans le code client, utilisez le proxy pour interagir avec l'objet réel. Le client ne doit pas avoir connaissance de la présence du proxy et peut utiliser les mêmes méthodes que celles définies dans l'interface commune.

Son diagramme de classe est :



1. **Les patterns de comportements**

Les **patterns de comportement** sont des modèles de conception qui se concentrent sur les interactions entre les objets et la répartition des responsabilités comportementales. Ils permettent de gérer efficacement les comportements et les algorithmes dans un système.

Nous pouvons en dénombrer 10 :

1. **Chain Of Responsibility Pattern ou Modèle de chaîne de responsabilité**

Le patron de conception Chain of Responsibility, également connu sous le nom de modèle de chaîne de responsabilité, est un patron comportemental qui permet de traiter une requête parmi une série d'objets récepteurs sans spécifier explicitement le récepteur. Chaque objet récepteur possède une référence à l'objet suivant dans la chaîne, formant ainsi une chaîne de traitement.

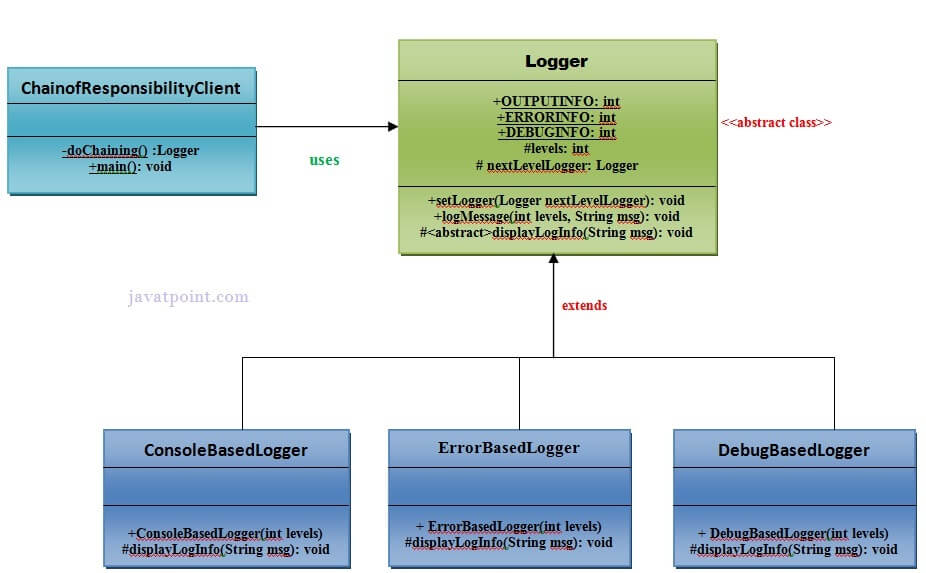
Voici les classes impliquées dans le modèle de chaîne de responsabilité :

* **Interface Handler** : C'est l'interface commune pour tous les objets récepteurs. Elle définit une méthode pour traiter la requête et déclare une méthode pour définir le prochain récepteur dans la chaîne.
* **Classe ConcreteHandler** : Ce sont les classes concrètes qui implémentent l'interface Handler. Chaque classe concrète décide si elle peut traiter la requête ou la passe au prochain récepteur de la chaîne.

L'implémentation du patron de conception Chain of Responsibility suit généralement les étapes suivantes :

* **Définir l'interface commune** : Créez une interface qui sera implémentée par tous les objets récepteurs. Cette interface doit définir les méthodes nécessaires pour traiter la requête et définir le prochain récepteur dans la chaîne.
* **Implémenter les classes réceptrices concrets** : Créez des classes concrètes qui implémentent l'interface commune. Chaque classe réceptrice décide si elle peut traiter la requête ou la passe au prochain récepteur dans la chaîne.
* **Configurer la chaîne de récepteurs** : Dans le code client, créez les instances des récepteurs concrets et configurez la chaîne en définissant le prochain récepteur pour chaque récepteur.
* **Envoyer la requête à la chaîne** : Envoyez la requête au premier récepteur de la chaîne. Chaque récepteur décide s'il peut traiter la requête ou la passe au prochain récepteur.

Son diagramme de classe est :



1. **Command Pattern ou Modèle de commande**

Le patron de conception Command, également connu sous le nom de modèle de commande, est un patron comportemental qui encapsule une requête en tant qu'objet, permettant ainsi de paramétrer des opérations avec différentes requêtes, de les mettre en file d'attente ou de les enregistrer, et de les exécuter à des moments différents.

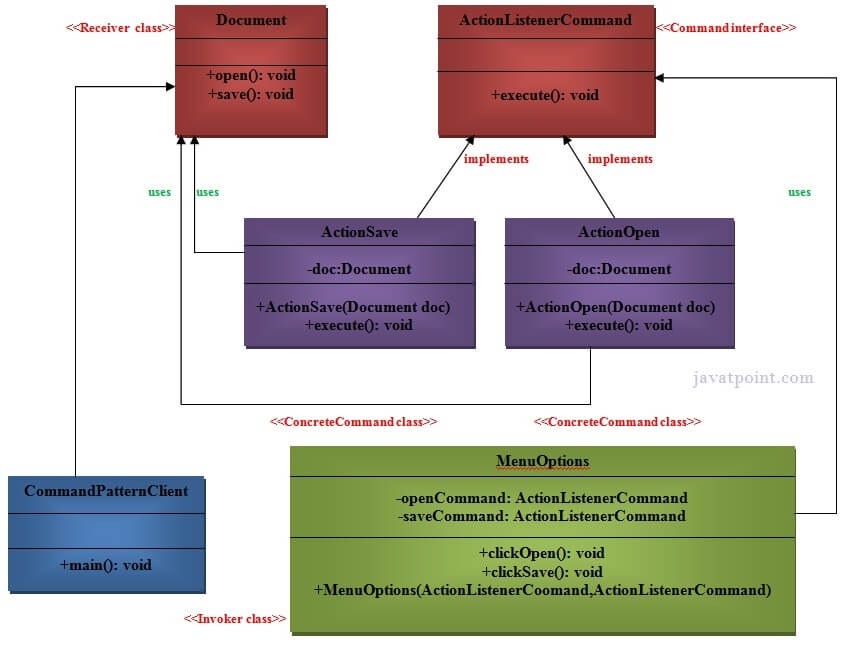
Voici les classes impliquées dans le modèle de commande :

* **Interface Command** : C'est l'interface commune pour toutes les commandes. Elle définit une méthode execute() qui sera implémentée par les classes concrètes pour exécuter l'action associée à la commande.
* **Classe ConcreteCommand** : Ce sont les classes concrètes qui implémentent l'interface Command. Chaque classe concrète encapsule une action spécifique en tant qu'objet commande en implémentant la méthode execute().
* **Classe Receiver** : C'est la classe qui connaît la logique métier spécifique pour exécuter l'action associée à une commande. Elle est invoquée par la méthode execute() de la classe ConcreteCommand.
* **Classe Invoker** : C'est la classe qui invoque la méthode execute() sur l'objet commande pour déclencher l'action associée.

L'implémentation du patron de conception Command suit généralement les étapes suivantes :

* **Définir l'interface commune** : Créez une interface qui représente la commande. Cette interface doit définir une méthode pour exécuter la commande, par exemple execute().
* **Implémenter les classes concrètes** : Créez des classes qui implémentent l'interface de commande et fournissent une implémentation de la méthode execute(). Chaque classe représente une commande spécifique.
* **Créer le récepteur** : Créez une classe qui contient la logique métier spécifique à exécuter lorsqu'une commande est invoquée. Cette classe est souvent appelée le récepteur.
* **Configurer l'invocateur** : Créez une classe invocatrice qui est responsable d'invoquer les commandes. Elle peut avoir une méthode pour définir la commande à exécuter.
* **Utiliser l'invocateur** : Dans le code client, créez une instance de l'invocateur et configurez-le avec une instance de commande appropriée. Ensuite, appelez la méthode d'exécution de l'invocateur pour déclencher l'exécution de la commande.

Son diagramme de classe est :



1. **Interpreter Pattern ou Modèle d’interpréteur**

Le modèle d'interpréteur, également connu sous le nom de patron de conception Interpreter, est un patron comportemental qui permet d'évaluer une grammaire ou une expression donnée. Il définit une représentation abstraite de l'expression à évaluer et fournit des interprètes pour évaluer cette expression.

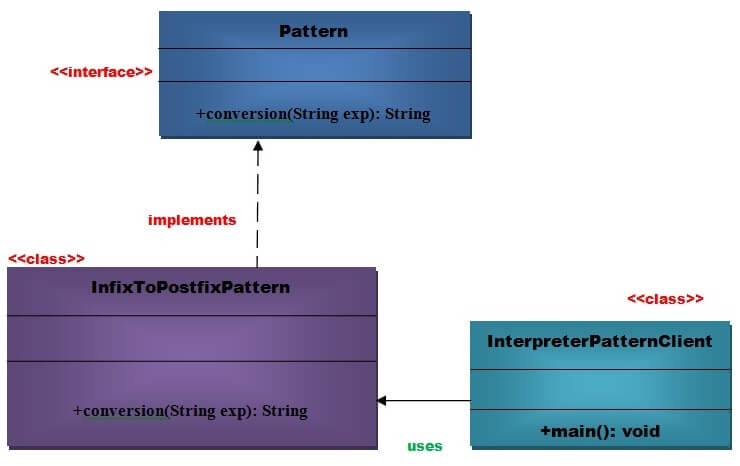
Voici les éléments clés du modèle d'interpréteur :

* **Expression** : C'est l'interface ou la classe abstraite qui représente une expression à évaluer. Elle définit une méthode interpret() qui sera implémentée par les classes concrètes.
* **TerminalExpression** : Ce sont les classes concrètes qui implémentent l'interface Expression et représentent les terminaux de la grammaire. Elles effectuent l'évaluation réelle de l'expression.
* **NonTerminalExpression** : Ce sont les classes concrètes qui implémentent l'interface Expression et représentent les non-terminaux de la grammaire. Elles contiennent des références à d'autres expressions et utilisent ces expressions pour évaluer l'expression globale.
* **Contexte** : C'est une classe qui contient les informations de contexte nécessaires pour évaluer l'expression.

L'implémentation du modèle d'interpréteur suit généralement les étapes suivantes :

* **Définir l'interface Expression** : Créez une interface ou une classe abstraite qui représente l'expression à évaluer. Cette interface doit définir une méthode interpret() qui prend un objet de contexte en argument et renvoie le résultat de l'interprétation de l'expression.
* **Implémenter les classes TerminalExpression** : Créez des classes concrètes qui implémentent l'interface Expression et représentent les terminaux de la grammaire. Ces classes effectuent l'évaluation réelle de l'expression en utilisant les données du contexte.
* **Implémenter les classes NonTerminalExpression** : Créez des classes concrètes qui implémentent l'interface Expression et représentent les non-terminaux de la grammaire. Ces classes contiennent des références à d'autres expressions et utilisent ces expressions pour évaluer l'expression globale.
* **Créer le contexte** : Créez une classe qui représente le contexte nécessaire pour l'interprétation des expressions. Cette classe contient les données nécessaires pour évaluer les expressions.
* **Utiliser les expressions pour interpréter l'expression globale** : Dans le code client, créez une instance du contexte avec les données nécessaires. Ensuite, construisez l'expression globale en utilisant les expressions terminales et non terminales appropriées. Enfin, appelez la méthode interpret() de l'expression globale en lui passant le contexte pour obtenir le résultat de l'interprétation.

Son diagramme de classe est :



1. **Iterator Pattern ou Modèle d’itérateur**

Le modèle d'itérateur, également connu sous le nom de patron de conception Iterator, est un patron comportemental qui permet d'accéder séquentiellement aux éléments d'une collection sans exposer sa structure interne. Il fournit une interface commune pour parcourir les éléments d'une collection, indépendamment de la façon dont cette collection est implémentée.

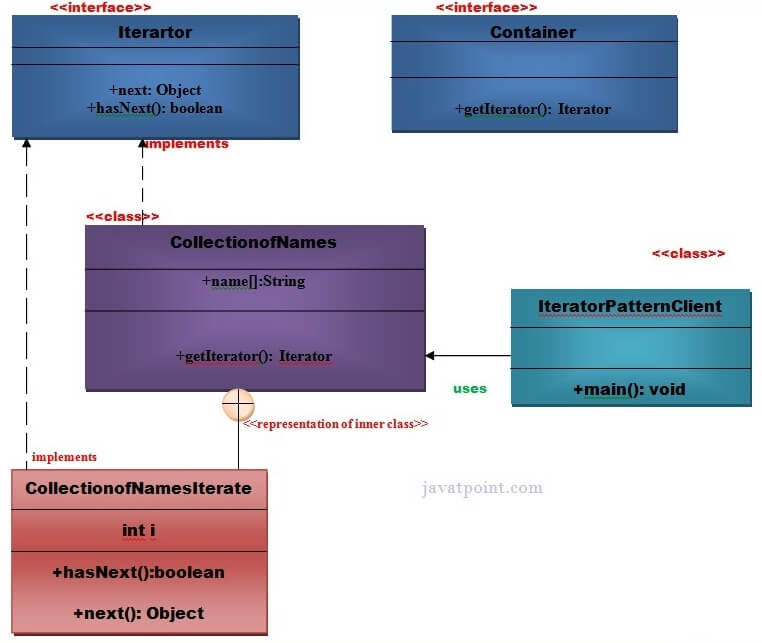
Voici les éléments clés du modèle d'itérateur :

* **Itérateur** : C'est une interface qui définit les méthodes permettant de parcourir les éléments d'une collection. Elle comprend généralement des méthodes telles que hasNext() pour vérifier s'il reste des éléments à parcourir, et next() pour récupérer le prochain élément de la collection.
* **Collection** : C'est une interface ou une classe abstraite qui représente la collection d'éléments à parcourir. Elle définit une méthode createIterator() qui renvoie une instance de l'itérateur approprié pour cette collection.
* **Itérateur concret** : Ce sont des classes concrètes qui implémentent l'interface Itérateur. Chaque classe itérateur est associée à une classe de collection spécifique et est responsable de parcourir les éléments de cette collection.

L'implémentation du modèle d'itérateur suit généralement les étapes suivantes :

* **Définir l'interface Iterator** : Créez une interface qui définit les méthodes nécessaires pour parcourir les éléments d'une collection, telles que hasNext() pour vérifier s'il reste des éléments à parcourir, et next() pour récupérer le prochain élément.
* **Définir l'interface Collection** : Créez une interface qui représente la collection d'éléments à parcourir. Elle doit inclure une méthode createIterator() qui renvoie une instance de l'itérateur approprié pour cette collection.
* **Implémenter l'itérateur concret** : Créez une classe concrète qui implémente l'interface Iterator. Cette classe est associée à une classe de collection spécifique et est responsable de parcourir les éléments de cette collection en utilisant les méthodes définies dans l'interface Iterator.
* **Implémenter la collection concrète** : Créez une classe concrète qui implémente l'interface Collection. Cette classe contient les éléments de la collection et renvoie une instance de l'itérateur concret via la méthode createIterator().
* **Utiliser l'itérateur pour parcourir les éléments** : Dans le code client, créez une instance de la collection concrète et obtenez un itérateur en utilisant la méthode createIterator(). Utilisez ensuite l'itérateur pour parcourir les éléments de la collection en utilisant les méthodes hasNext() et next() de l'itérateur.

Son diagramme de classe est :



1. **Mediator Pattern ou Modèle de médiateur**

Le modèle de médiateur, également connu sous le nom de patron de conception Mediator, est un patron comportemental qui permet de réduire les dépendances directes entre les objets en introduisant un objet médiateur qui facilite la communication entre ces objets. Le médiateur agit comme un intermédiaire qui coordonne les interactions entre les objets, permettant ainsi de réduire le couplage entre eux.

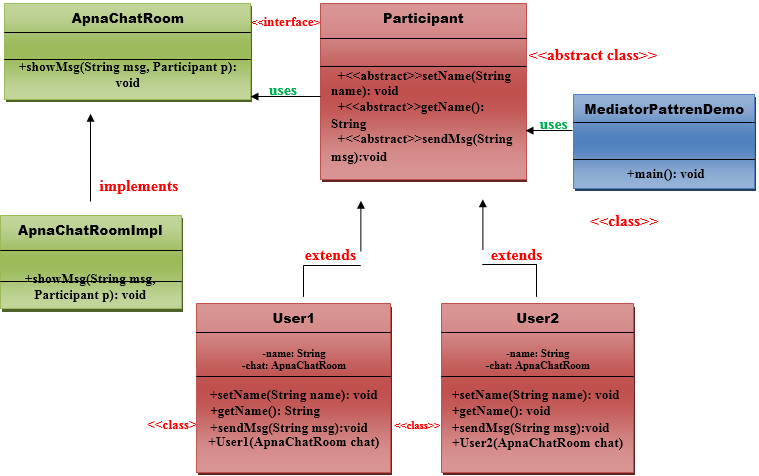
Voici les éléments clés du modèle de médiateur :

* **Médiateur** : C'est une interface ou une classe abstraite qui définit les méthodes de communication entre les objets. Il encapsule la logique de coordination et de communication entre les objets.
* **Objets collègues** : Ce sont des objets qui interagissent entre eux, mais qui évitent les dépendances directes. Ils communiquent uniquement avec le médiateur, qui se charge de relayer les messages aux objets concernés.

L'implémentation du modèle de médiateur comprend généralement les étapes suivantes :

* **Identifier les objets collègues** : Identifiez les objets qui doivent communiquer entre eux de manière complexe. Ces objets deviendront les collègues du médiateur.
* **Concevoir l'interface ou la classe abstraite du médiateur** : Définissez une interface ou une classe abstraite qui déclare les méthodes de communication entre les objets collègues. Cette interface ou cette classe abstraite encapsulera la logique de coordination et de communication.
* **Implémenter le médiateur concret** : Créez une classe concrète qui implémente l'interface ou hérite de la classe abstraite du médiateur. Cette classe sera responsable de la coordination des interactions entre les objets collègues. Elle devra maintenir des références vers les objets collègues et implémenter les méthodes de communication définies par l'interface ou la classe abstraite du médiateur.
* **Implémenter les objets collègues** : Créez des classes pour chaque objet collègue. Ces classes devront avoir une référence vers le médiateur et implémenter les méthodes de communication définies par l'interface ou la classe abstraite du médiateur. Les objets collègues communiqueront uniquement avec le médiateur et n'auront pas de dépendances directes entre eux.
* **Configurer les références du médiateur** : Dans le code client, créez une instance du médiateur concret et des objets collègues. Configurez les références du médiateur vers les objets collègues en utilisant des méthodes ou des constructeurs appropriés.
* **Utiliser le médiateur pour la communication** : Utilisez les méthodes de communication des objets collègues pour envoyer des messages via le médiateur. Le médiateur se chargera de transmettre les messages aux objets collègues appropriés.
* **Tester et ajuster** : Testez l'implémentation pour vous assurer que les objets collègues communiquent correctement via le médiateur. Si nécessaire, ajustez la logique de coordination dans le médiateur pour répondre aux exigences de communication spécifiques.

Son diagramme de classe est :



1. **Memento Pattern ou Motif Memento**

Le modèle de memento, également connu sous le nom de motif memento, est un patron de conception comportemental qui permet de capturer et de restaurer l'état interne d'un objet sans violer son encapsulation. Il permet de sauvegarder et de restaurer l'état d'un objet à un moment donné, offrant ainsi une fonctionnalité de retour en arrière ou d'annulation.

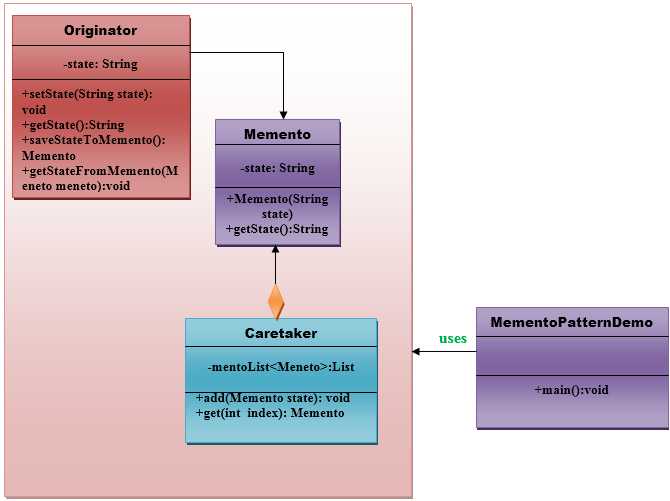
Voici les éléments clés du modèle de memento :

* **Originateur (Originator)** : C'est l'objet dont l'état doit être sauvegardé et restauré. L'originateur crée un objet memento pour représenter son état actuel et peut également restaurer son état à partir d'un objet memento.
* **Memento** : C'est l'objet qui représente l'état sauvegardé de l'originateur. Il peut stocker différentes propriétés et valeurs qui décrivent l'état de l'originateur à un moment donné. Le memento n'est pas accessible aux autres objets, y compris l'originateur, garantissant ainsi l'encapsulation.
* **Gardien (Caretaker)** : C'est l'objet responsable de la gestion des objets memento. Il peut stocker et manipuler plusieurs objets memento pour permettre des opérations de sauvegarde, de restauration ou d'annulation de l'état de l'originateur.

Les étapes de l'implémentation du modèle de memento sont les suivantes :

* **Définir la classe Originateur (Originator**) : Cette classe représente l'objet dont l'état doit être sauvegardé et restauré.
* **Définir la classe Memento** : Cette classe représente l'objet memento lui-même.
* **Définir la classe Gardien (Caretaker)** : Cette classe est responsable de la gestion des objets memento.

Son diagramme de classe est :



1. **Observer Pattern ou Modèle d’observateur**

Le modèle d'observateur, également connu sous le nom de patron de conception Observer, est un patron de conception comportemental qui permet de créer une relation de dépendance un-à-plusieurs entre les objets, de sorte que lorsque l'état d'un objet change, tous les objets qui en dépendent sont notifiés et mis à jour automatiquement.

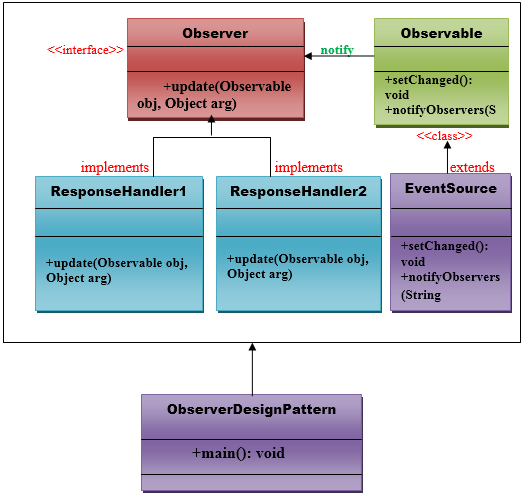
Le modèle d'observateur repose sur deux principaux acteurs :

* **Sujet (Subject)** : C'est l'objet qui est observé et dont l'état change. Le sujet maintient une liste d'observateurs et fournit des méthodes pour ajouter, supprimer et notifier les observateurs. Lorsque l'état du sujet change, il notifie tous les observateurs en invoquant une méthode spécifique.
* **Observateur (Observer)** : C'est l'objet qui observe le sujet et est notifié des changements d'état. L'observateur définit une méthode de mise à jour qui est appelée par le sujet lorsqu'un changement d'état se produit. L'observateur peut alors récupérer les nouvelles données de l'état du sujet et effectuer les actions nécessaires.

Les étapes de l'implémentation du modèle d'observateur sont les suivantes :

* **Définir l'interface Sujet (Subject**) : Cette interface doit définir les méthodes pour enregistrer, supprimer et notifier les observateurs.
* **Définir l'interface Observateur (Observer)** : Cette interface doit définir la méthode mettreAJour() (ou update() en anglais) qui sera appelée par le sujet pour informer les observateurs d'un changement d'état.
* **Implémenter la classe SujetConcret (ConcreteSubject)** : Cette classe doit implémenter l'interface Sujet (Subject).
* **Implémenter la classe ObservateurConcret** **(ConcreteObserver)** : Cette classe doit implémenter l'interface Observateur (Observer).
* **Utiliser le modèle d'observateur** : Dans le code client, créez une instance du sujet concret (ConcreteSubject) et des observateurs concrets (ConcreteObserver).

Son diagramme de classe est :



1. **State Pattern ou Modèle d’état**

Le modèle d'état, également connu sous le nom de patron de conception State, est un patron de conception comportemental qui permet à un objet de modifier son comportement interne à mesure que son état interne change. Il encapsule chaque état possible en tant qu'objet distinct et délègue le comportement spécifique à l'objet d'état correspondant.

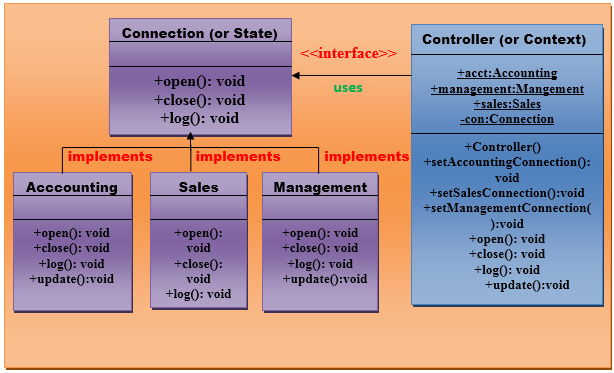
Le modèle d'état repose sur les acteurs clés suivants :

* **Contexte (Context)** : C'est l'objet qui possède un état interne et dont le comportement change en fonction de cet état. Le contexte maintient une référence vers un objet d'état concret qui représente l'état courant. Le contexte fournit des méthodes pour permettre la transition entre les différents états.
* **État (State)** : C'est l'interface ou la classe abstraite qui définit les méthodes communes à tous les états concrets. Cela inclut généralement des méthodes pour effectuer des actions spécifiques à l'état et pour gérer les transitions vers d'autres états.
* **États concrets (Concrete States)** : Ce sont les objets qui représentent les différents états possibles du contexte. Chaque état concret implémente les méthodes définies par l'interface État et gère son propre comportement spécifique à l'état, ainsi que les transitions vers d'autres états.

Les étapes de l'implémentation du modèle d'état sont les suivantes :

* **Définir l'interface ou la classe abstraite pour représenter l'état (State)** : Cette interface ou classe abstraite doit définir les méthodes communes à tous les états concrets.
* **Implémenter les classes d'états concrets (Concrete States)** : Chaque classe d'état concret doit implémenter l'interface ou hériter de la classe abstraite définissant l'état.
* **Définir la classe qui représente le contexte (Context)** : Cette classe doit posséder un attribut pour stocker l'état courant.
* **Utiliser le modèle d'état** : Dans le code client, créez une instance du contexte et attribuez-lui un état initial

Son diagramme de classe est :



1. **Strategy Pattern ou Modèle de stratégie**

Le modèle de stratégie, également connu sous le nom de patron de conception Strategy, est un patron de conception comportemental qui permet de définir une famille d'algorithmes ou de comportements interchangeables, encapsulés dans des classes distinctes, et de les rendre interchangeables à la volée en fonction des besoins du client.

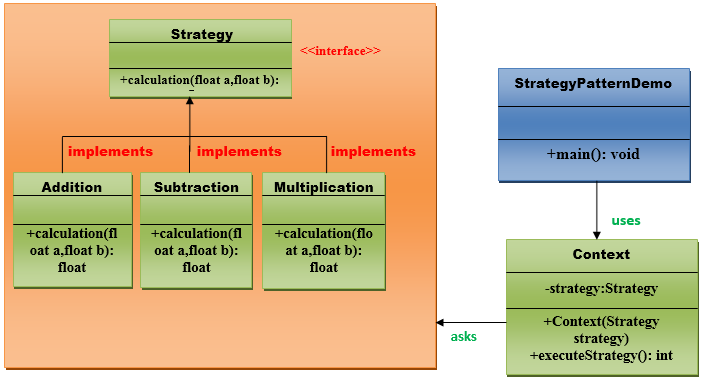
Le modèle de stratégie repose sur les acteurs clés suivants :

* **Stratégie (Strategy)** : C'est l'interface ou la classe abstraite qui définit les méthodes communes à tous les algorithmes ou comportements concrets. Les stratégies peuvent être interchangeables et encapsulent une variante d'algorithme ou de comportement spécifique.
* **Stratégies concrètes (Concrete Strategies)** : Ce sont les classes qui implémentent l'interface Stratégie ou héritent de la classe abstraite Stratégie. Chaque stratégie concrète encapsule un algorithme ou un comportement spécifique.
* **Contexte** : C'est la classe qui utilise la stratégie pour effectuer une opération spécifique. Le contexte dispose d'une référence vers une stratégie concrète et utilise cette stratégie pour exécuter l'opération requise.

Les étapes de l'implémentation du modèle de stratégie sont les suivantes :

* **Définir l'interface ou la classe abstraite pour représenter la stratégie** : Cette interface ou classe abstraite doit définir les méthodes communes à toutes les stratégies.
* **Implémenter les classes de stratégies concrètes** : Chaque classe de stratégie concrète doit implémenter l'interface ou hériter de la classe abstraite définissant la stratégie.
* **Définir la classe qui représente le contexte** : Cette classe doit avoir une référence à l'interface ou à la classe abstraite de la stratégie.
* **Utiliser le modèle de stratégie** : Dans le code client, créez une instance du contexte

Son diagramme de classe est :



1. **Template Pattern ou Modèle de modèle**

Le "Modèle de modèle" (ou "Template Pattern" en anglais) est une conception de modèle qui permet de définir le squelette d'un algorithme dans une classe de base, en laissant les sous-classes remplir les détails spécifiques de l'implémentation. Ce modèle est également connu sous le nom de "Modèle de méthode" (ou "Method Template").

Voici les classes qui constituent le Modèle de modèle :

* **La classe abstraite (ou interface) du modèle**
* **Les sous-classes qui étendent la classe abstraite**
* **Le client qui utilise le Modèle de modèle**

Voici les étapes de mise en œuvre du Modèle de modèle :

* **Définir une classe abstraite (ou une interface) qui représente le modèle de l'algorithme** : Cette classe abstraite (ou interface) définit une méthode (ou un ensemble de méthodes) qui représente le squelette de l'algorithme.
* **Implémenter les sous-classes qui étendent la classe abstraite (ou implémentent l'interface) :** Chaque sous-classe doit fournir une implémentation des parties abstraites (non implémentées) de la méthode de l'algorithme.
* **Utiliser le modèle de modèle** : Dans le code client, créez une instance de la sous-classe appropriée.

Son diagramme de classe est :

