**Méthode de fabrique :**

* **Intention :**

Définir une interface pour la création d’un objet, mais le choix de la classe concrète de l’objet est délégué à des sous-classes.

* **Structure :**



* **Participants :**

*Product* définit l’interface des objets que la méthode de fabrique crée.

*ConcreteProduct* implémente l’interface *Product.*

*Creator*  déclare la méthode de fabrique qui retourne un objet de type *Product.*

*ConcreteCreator* redéfinit la méthode de pour retourner une instance de *ConcreteProduct*.

* **Collaboration :**

La classe *Creator* compte sur ses sous-classes pour implémenter la méthode de fabrique afin qu’elle retourne une instance du *ConcreteProduct.*

La classe *Creator* est écrite sans savoir quelle véritable classe *ConcreteProduct* sera instanciée :

* Déterminé seulement à l’exécution par la sous-classe *ConcreteCreator* qui est instanciée et utilisé par l’application (client).
* Mais ne signifie pas que cette sous-classe décide à l’exécution de la classe *ConcreteProduct* à instancier.
* **Avantages :**
* En évitant de spécifier le nom de la classe concrète et les détails de son instanciation, le code client devient plus flexible et réutilisable.
* Le client est uniquement dépendant de l’interface *Product* et peut fonctionner avec n’importe quelle classe *ConcreteProduct* qui implémente cette interface.
* **Inconvénients :**
* Le client peut avoir à créer une sous-classe de la classe *Creator*, juste pour instancier un *ConcreteProduct* particulier.

**Fabrique abstraite :**

* **Intention :**

Fournir une interface pour créer des familles d’objets liés sans spécifier leurs classes concrètes.

* **Structure :**



* **Participants :**

*AbstractProduct* déclare une interface pour un style de produit.

*ConcreteProduct* implémente l’interface *AbstractProduct* et définit un produit devant être créé par la fabrique concrète correspondante.

*Client* utilise uniquement les interfaces déclarées par les classes *AbstractFactory* et *AbstractProduct.*

* **Collaboration :**

Cette fabrique concrète crée des produits ayant une implémentation particulière.

Pour créer des produits différents, les clients doivent utiliser une fabrique concrète différente.

La classe *AbstractFactory* défère la création des produits à ses sous-classes concrètes.

* **Avantages :**

Isole les clients des classes concrètes, car ils manipulent seulement des interfaces abstraites.

Rend facile le changement des familles de produits, car une fabrique concrète particulière supporte une famille complète de produits.

Favorise la cohérence entre produits 🡪 Impose l’utilisation de produits d’une seule et même famille.

* **Inconvénients :**

Difficile d’ajouter de nouveaux produits, car requiert de changer l’interface *AbstractFactory* et donc d’étendre l’interface de toutes les classes de fabrique concrète.

**Singleton :**

* **Intention :**

Garantir qu’une classe n’a qu’une seule instance et fournir un point d’accès global à cette instance.

* **Utilisation :**

Il ne doit y avoir exactement qu’une seule instance d’une classe.

Cette instance unique doit être facilement accessible aux clients.

Cette instance unique doit être extensible par héritage et les clients doivent pouvoir utiliser cette instance étendue sans modifier leur code.

* **Structure :**



* **Participants :**

Une seule classe concrète (aucune classe abstraite/interface).

*Singleton* définit une méthode de classe *Instance* permettant d’accéder à son unique instance :

* Cette méthode peut être chargée de créer cette instance unique.
* *uniqueInstance* est une variable de classe de type *Singleton.*
* *singletonData* est une variable de classe de type *Singleton.*
* *singletonOperation* est une méthode d’instance (comportement).
* **Collaboration :**

Les clients accèdent à l’instance d’un *Singleton* via la méthode de classe *Instance.*

* **Avantages :**

Permet un accès contrôlé à une unique instance.

Permet la réduction de l’espace de noms (vs variables globales)

Peut être étendu par héritage.

Permet un nombre variable d’instances (toutes visibles aux clients).

Est plus flexible que les méthodes de classe.

**Façade :**

* **Intention :**

Fournir une interface unifiée et de haut niveau à un ensemble d’interfaces d’un sous-système (i.e., un groupe de classes), afin de le rendre plus facile à utiliser.

* **Structure :**



* **Participants :**

*Facade* sait quelles classes du sous-système sont responsables de telle ou telle requête et délègue les requêtes des clients aux objets appropriés.

*Subsystem classes* implémentent les fonctionnalités du sous-système, traitent les requêtes émises par la façade, mais ne connaissent pas cette dernière (i.e., n’ont aucune référence vers elle).

* **Collaboration :**

Les clients communiquent avec le sous-système en envoyant des requêtes à la façade qui les transmet aux objets appropriés du sous-système, après traduction si nécessaire.

Les clients qui utilisent la façade n’ont pas à accéder directement aux objets du sous-système.

* **Avantages :**

Cache l’implémentation du sous-système aux clients, le rendant plus facile à utiliser.

Favorise un couplage faible entre le sous-système et ses clients 🡪 Modification des classes du sous-système sans affecter les clients.

N’empêche pas les clients d’accéder aux classes sous-jacentes.

* **Inconvénients :**

Possible perte de fonctionnalités des classes interfacées selon la manière dont la façade est réalisée.

N’empêche pas les clients d’accéder aux classes sous-jacentes.

**Adaptateur :**

* **Intention :**

Convertir l’interface d’une classe en une autre interface attendue par les clients et permettre ainsi de faire collaborer des classes même si elles ont des interfaces incompatibles.

* **Structure :**



* **Participants :**

*Target*  définit l’interface spécifique à l’application que le client utilise.

*Client* collabore avec les objets conformes à l’interface *Target.*

*Adaptee* définit une interface existante qui a besoin d’adaptation.

*Adapter* adapte l’interface d’Adaptee à l’interface *Target* et est responsable des fonctionnalités que la classe adaptée ne fournit pas.

* **Collaboration :**

Les clients font appel aux opérations d’une instance de *Adapter.*

L’adaptateur fait alors appel aux opérations d’Adapteepour réaliser le service.

* **Adaptateur de classe :**

Utilise l’héritage multiple pour adapter une interface à une autre.

Hérite à la fois de l’interface et de la classe sur étagère à adapter.

N’introduit qu’une nouvelle classe, aucune indirection n’est nécessaire pour atteindre la classe adaptée.

Permet de redéfinir certain comportement de la classe adaptée (*Adapter* est une sous-classe d’*Adaptee).*

Mais ne permet pas d’adapter une classe et toutes ses sous-classes.

* **Adaptateur d’objet :**

Utilise l’héritage simple et la composition d’objet (délégation).

Hérite de l’interface, mais compose une instance de la classe sur étagère.

Est beaucoup plus fréquent, en particulier lorsque l’héritage multiple de classe n’est pas possible.

Peut travailler avec plusieurs classes adaptées dans une hiérarchie.

Mais rend plus difficile de redéfinir le comportement de la classe adaptée (oblige l’adaptateur à faire référence à la sous-classe).

**Procuration (Proxy) :**

* **Intention :**

Fournir un substitut (le proxy) à un autre objet afin d’en contrôler les accès (i.e. les opérations qui lui sont appliquées).

* **Structure :**



* **Participants :**

*Proxy* a la même interface que l’objet à contrôler et gère une référence au *RealSubject* lui permettant d’y accéder et d’en contrôler les accès.

*Subject* définit une interface commune pour *RealSubject* et  *Proxy,* afin qu’un proxy puisse être utilisé partout où un *RealSubject* est attendu.

*RealSubject* définit l’objet réel que le proxy représente.

* **Collaboration :**

Le proxy retransmet les requêtes au *RealSubject* si nécessaire (délégation), selon le type de proxy.

Un proxy agit comme un intermédiaire entre le client et le sujet réel.

* **Avantages :**

Introduit un niveau d’indirection lors de l’accès à un objet :

* Pour cacher au client le fait que l’objet réside dans un autre espace d’adressage (distribution).
* Pour effectuer des optimisations transparentes pour le client.
* Pour vérifier si l’appelant a les permissions requises pour effectuer la requête.
* **Inconvénients :**

Le proxy est simplement une réplique exacte de son sujet réel.

**Décorateur :**

* **Intention :**

Attacher dynamiquement des responsabilités supplémentaires à un objet, fournissant une alternative flexible à l’héritage pour étendre des fonctionnalités.

* **Structure :**



* **Participants :**

*Component* définit l’interface des objets qui peuvent recevoir dynamiquement des responsabilités supplémentaires.

*ConcreteComponent* définit un objet auquel des responsabilités supplémentaires peuvent être attachées.

*Decorator* gère une référence à un objet de type *Component* (i.e. l’objet décoré) et définit une interface conforme à ce dernier.

*ConcreteDecorator* ajoute des responsabilités au composant décoré.

* **Collaboration :**

*Decorator* transmet les requêtes à son objet décoré et peut éventuellement effectuer des traitements supplémentaires avant et/ou après.

* **Avantages :**

Offre plus de flexibilité que l’héritage (statique).

Evite de surcharger de fonctionnalités les classes situées en haut des hiérarchies (classes monolithiques).

* **Inconvénients :**

Casse l’identité de l’objet : un décorateur et son composant ne sont pas identiques.

Conduit à des systèmes composés de beaucoup de petits objets.

**Composite :**

* **Intention :**

Composer des objets dans des structures arborescentes pour représenter des hiérarchies composants/composés, et permettre aux clients de traiter les objets individuels et leurs compositions de manière uniforme.

* **Structure :**



* **Participants :**

*Component* déclare l’interface des objets de la composition et implémente le comportement par défaut commun à toutes les classes.

*Leaf* représente les objets feuilles (i.e. sans enfant) de la composition.

*Composite* définit le comportement des composants ayant des fils, stocke les fils et implémente les opérations nécessaires à leur gestion.

*Client* manipule les objets de la composition à travers l’interface *Component*.

* **Collaboration :**

Si l’objet manipulé par le client est une feuille, la requête est traitée directement ? Sinon, c’est un composite, ce dernier retransmet la requête à ses composants fils, en effectuant éventuellement des traitements supplémentaires avant et/ou après.

* **Avantages :**

Facilite l’ajout de nouveaux types de composants.

Simplifie le code client, car il n’a pas à savoir (et ne devrait pas se soucier de) s’il traite avec une feuille ou un composite.

* **Inconvénients :**

Est difficile de restreindre et vérifier le type des composants d’un composite.

**Commande :**

* **Intention :**

Encapsuler une requête comme un objet, permettant ainsi de la manipuler de différentes manières.

* **Structure :**



* **Participants :**

*Command* déclare une interface pour exécuter une opération.

*ConcreteCommand* définit un lien entre un objet *Receiver* et une action, et implémente *Execute* en appelant l’opération correspondante du récepteur.

*Client* crée un objet *ConcreteCommand* et définit son récepteur.

*Invoker* demande à la commande d’exécuter la requête

*Receiver* sait comment effectuer les opérations associées à l’exécution d’une requête.

* **Collaboration :**

Le client crée une commande et spécifie son récepteur.

Un objet *Invoker* stocke la commande.

L’invocateur appelle la méthode *Execute* de la commande.

La commande invoque les opérations de son récepteur pour exécuter la requête.

* **Avantages :**

Découple l’objet qui invoque l’opération (i.e. l’invocateur) de celui qui sait comment la réaliser (i.e. le récepteur).

Fait les commandes des objets de première classe : elles peuvent être manipulées et étendues comme tout objet.

Permet de grouper des commandes dans une commande composite (i.e. une macro-commande).

Facilite l’ajout de nouvelles commandes sans modifier le code existant.

**Observateur :**

* **Intention :**

Définir une interdépendance de type un à plusieurs, de telle sorte que si un objet change d’état, tous ceux qui en dépendent en soient notifiés et mis à jour automatiquement.

* **Structure :**

****

* **Participants :**

*Subject* connaît ses observateurs (en nombre quelconque) et fournit une interface pour en ajouter et en retirer.

*Observer* définit une interface de mise à jour pour les objets qui doivent être notifiés de changements dans un sujet.

*ConcreteSubject* représente l’objet observé, mémorise l’état qui intéresse ses observateurs et leur envoie une notification lorsque celui-ci change.

*ConcreteObserver* gère une référence à un sujet, mémorise l’état qui doit rester cohérent avec celui du sujet et implémente l’interface de mise à jour *Observer* pour assurer cette cohérence.

* **Collaboration :**

Un sujet notifie ses observateurs de tout changement qui pourrait rendre leur état incohérent avec le sien propre.

La méthode *Update* de chaque observateur est alors appelée.

Un observateur peut demander des informations au sujet pour mettre son état en conformité avec celui du sujet.

* **Avantages :**

Favorise un couplage faible entre le sujet et ses observateurs.

Fournit un support à la diffusion.

* **Inconvénients :**

Peut causer une cascade de notifications/mises à jour inopinées.

Une simple interface de mise à jour requiert que les observateurs déduisent eux-mêmes de ce qui a changé dans le sujet (difficile).

**Itérateur :**

* **Intention :**

Fournir un moyen de parcourir séquentiellement un agrégat (collection, conteneur) d’éléments sans connaître sa structure interne.

* **Structure :**



* **Participants :**

*Iterator* définit une interface pour accéder aux éléments et les parcourir.

*ConcreteIterator* implémente l’interface *Iterator*, garde une trace de l’élément courant lors de la traversée de l’agrégat et détermine l’élément suivant.

*Aggregate* définit une interface pour la création d’un objet *Iterator* (une méthode de fabrique).

*ConcreteAggregate* implémente l’interface de création de *Iterator,* afin de retourner une instance appropriée de *ConcreteIterator*.

* **Avantages :**

Simplifie l’interface de l’agrégat, en ne polluant pas de méthodes relatives à son parcours.

Permet de gérer plusieurs parcours simultanés sur un agrégat.

Permet de modifier l’algorithme de parcours d’un agrégat.

* En remplaçant l’instance de l’itérateur par une autre différente.
* En définissant une nouvelle sous-classe de *Iterator.*

**Visiteur :**

* **Intention :**

Représenter une opération à effectuer sur les éléments d’une structure et permettre de définir une nouvelle opération sans modifier les classes des éléments sur lesquels il opère.

* **Structure :**

****

* **Participants :**

*Visitor* déclare une opération *Visit* pour chaque classe de *ConcreteElement* de la structure.

* La signature de l’opération identifie la classe qui envoie la requête de visite au visiteur.
* Le visiteur détermine alors la classe concrète de l’élément visité et accède à cet élément directement à travers son interface.

*ConcreteVisitor* implémente chaque opération (i.e. un fragment de l’algorithme défini pour la classe d’élément correspondante) déclarée par *Visitor*, fournit le contexte de l’algorithme et stocke son état local (e.g. pour accumuler des résultats lors du parcours de la structure.

*Element* définit une opération *Accept* qui prend un visiteur en paramètre.

*ConcreteElement* implémente l’opération *Accept* et fait appel au visiteur en se passant lui-même en paramètre.

*ObjectStructure* peut énumérer ses éléments et peut fournir une interface de haut niveau permettant au visiteur de visiter ses éléments.

* **Collaboration :**

Un client crée un visiteur concret pour parcourir les éléments d’une structure et faire un traitement variable selon le type de l’élément.

Lorsqu’un élément est visité (méthode *Accept)*, celui-ci appelle l’opération du visiteur qui correspond à sa classe.

L’élément se passe lui-même en paramètre de cette opération, afin de permettre au visiteur d’accéder à son état si nécessaire.

* **Avantages :**

Facilite l’ajout de nouvelles opérations.

Regroupe les opérations communes dans *Visitor* et sépare celles indépendantes dans leur propre sous-classe *ConcreteVisitor.*

* Simplifie à la fois les classes définissant les éléments et les algorithmes définis dans les visiteurs (toutes les structures de données spécifiques à l’algorithme peuvent être cachées dans le visiteur).

Peut accumuler un état, plutôt que de la passer comme un paramètre supplémentaire à l’opération de visite.

Permet de parcourir une structure composée d’éléments de types différents (contrairement à un itérateur).

* **Inconvénients :**

Rend difficile l’ajout de nouvelles classes *ConcreteElement*: Entraîne l’ajout d’une nouvelle opération abstraite dans *Visitor* + une implémentation correspondante dans chaque classe *ConcreteVisitor.*

L’interface de *ConcreteElement* doit être assez riche pour permettre au visiteur de faire son travail : force à fournir des opérations publiques qui accèdent à l’état interne d’un élément, ce qui peut casser son encapsulation.

**Etat :**

* **Intention :**

Permettre à un objet de modifier son comportement lorsque son état interne change.

* **Structure :**

****

* **Participants :**

*Context* est une classe qui permet d’utiliser un objet à états et qui gère une instance d’une sous-classe *ConcreteState définissant l’état courant.*

*State* définit une interface qui encapsule le comportement associé à un état particulier de *Context.*

*ConcreteState* implémente un comportement associé à un état de *Context.*

* **Collaboration :**

*Context* est la principale interface pour les clients qui peuvent la configurer avec des objets *State,* puis n’ont plus à traiter avec les états directement.

Il transmet les requêtes spécifiques aux états à son état courant :

* Il faut veiller à ce que le contexte pointe sur un objet état reflétant son état courant.
* Il peut se passer lui-même comme argument à l’état traitant la requête, permettant à celui-ci d’accéder au contexte si nécessaire.

Il revient, soit à *Context*, soit aux sous-classes *ConcreteState* de décider de l’état qui succède à un autre état et sous quelles conditions.

* **Avantages :**

Sépare les comportements relatifs à chaque état : ils sont placés dans un objet (i.e. une sous-classe de *State*).

Facilite l’ajout et la suppression des états et des transitions.

Elimine les conditionnelles.

Rend les transitions entre états plus explicites.

* **Inconvénients :**

Augmente le nombre d’objets.

**Stratégie :**

* **Intention :**

Définir une famille d’algorithmes, encapsuler chacun d’eux, et les rendre interchangeables, tout en leur permettant d’évoluer indépendamment des clients qui les utilisent.

* **Structure :**



* **Participants :**

*Strategy* déclare une interface commune à tous les algorithmes.

*ConcreteStrategy* implémente l’algorithme en utilisant l’interface *Strategy*.

*Context* gère une référence à un objet *Strategy* et peut définir une interface qui permet à cet objet d’accéder à ses données.

* **Collaboration :**

En général, les clients créent un objet *ConcreteStrategy* et le passent au contexte, puis interagissent exclusivement avec le contexte.

Un contexte transmet les requêtes de ses clients à sa stratégie :

* Il peut passer à la stratégie toutes les données requises par l’algorithme lorsque celui-ci est appelé (contexte et stratégie découplés).
* Alternativement, le contexte peut se passer lui-même comme argument aux opérations de la stratégie, permettant à celle-ci de rappeler le contexte si nécessaire (contexte et stratégie plus fortement couplés).
* **Avantages :**

Fournit une alternative à l’héritage de la classe *Context* pour obtenir une variété d’algorithmes ou de comportements.

Sépare l’implémentation de l’algorithme de celle de contexte.

Elimine les conditionnelles pour sélectionner le bon comportement.

Fournit différentes implémentations du même comportement.

* **Inconvénients :**

Augmente le nombre d’objets.

Nécessite que les clients connaissent les différentes stratégies disponibles et comprennent en quoi elles diffèrent avant de pouvoir choisir la plus appropriée.

Tous les algorithmes doivent utiliser la même interface *Strategy*.