Design Pattern

Un design pattern décrit une solution à un problème général et récurrent de conception dans un contexte particulier.

Comment utiliser un pattern ? Il faut les connaître et les problèmes récurrents qu'ils adressent.

Un pattern est la description d’une solution globale : il faut parfois l’adapter à la situation ou en utiliser plusieurs pour résoudre le problème. On doit aussi déterminer quelles classes devraient remplacer les classes stéréotypes fournies par le pattern.

Les types de pattern :

**Création** : ils rendent un système indépendant de comment ses objets sont créés, initialisés et configurés (utile lorsque le système évolue : de futures classes peuvent ne pas être connues).

**Structure** : ils permettent de composer des classes et des objets afin de former des structures plus importantes. Réduit le couplage entre les classes.

**Comportement**: s’intéressent aux interactions entre les objets, décrivent des flots de contrôle complexes.

Les patterns ont une  **portée** (à quoi est-ce que ça s’applique) : les classes ou les objets.

Les patterns de classes (respectivement d’objets) se focalisent sur les relations entre les classes (respectivement objets) et leurs sous-classes (utilisent l'héritage, respectivement la composition).

Les patterns de création

Abstraction du processus d’instanciation (bypass du « new ») : on encapsule la connaissance des classes concrètes que le système utilise, on cache la manière dont les instances de ces classes (les classes concrètes) sont créées et assemblées. Au final le système ne doit connaitre que les classes abstraites.

**Avantages**: grande flexibilité au niveau de la création -> ce qui doit être créé, par qui, comment et quand. Configuration du système dynamique ou statique avec des objets « produits ».

Problème : *look-and-feel.* Disposant d’une hiérarchie de classes (plusieurs implémentations d’une classe pour différents design par exemple), comment flexiblement créer les instances ?

**Méthode de fabrique (Factory method) (portée : classes)** : **Abstraire la création d’objets**. Encapsuler ce qui varie dans une classe (ici, la création d’objet). Définir une classe « créatrice » avec une méthode abstraite d’instance create.

**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.

**Utilisation** : une classe ne peut pas anticiper la classe des objets qu’elle doit créer (cf. « ce que l’on veut faire »), et attend de ses sous-classes qu’elles spécifient les objets qu’elle crée.

**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**, et **ConcreteCreator** redéfinit la méthode de fabrique pour retourner une instance d'un **ConcreteProduct**.

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

La classe **Creator** est écrite sans savoir quelle véritable classe **ConcreteProduct** sera instanciée. Elle sera déterminée seulement à l'exécution par la sous-classe **ConcreteCreator** qui est instanciée et utilisée par l'application (client), mais ne signifie pas que cette sous-classe décide à l'exécution de la classe **ConreteProduct** 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 **ConcreteProcduct** qui implémente cette interface.

**Inconvénients**: Le client peut avoir à créer une sous-classe de la classe **Creator** juste pour instancier un **ConcreteProduct** particulier.

**Implémentation**: la classe **Creator** est soit **abstraite** (pas d’implémentation de la méthode de fabrique), soit **concrète**. La méthode de fabrique peut éventuellement prendre un paramètre (et utiliser des if/else) pour décider de l’objet à créer.

**Fabrique abstraite (portée : objets) : Abstraire la création d’objets** (suite). La méthode de fabrique permet de changer une méthode de classe en une méthode d’instance, mais ce n’est pas suffisant : on peut ajouter une hiérarchie de classes auxiliaires qui permettent de gérer et séparer cette construction de classes utilitaires.

On a une classe abstraite qui va servir à créer une famille d’objets : pour chaque « famille » (par exemple si on veut plusieurs designs pour une application) on a une sous-classe abstraite (implémentations alternatives à cette famille).

**Intention** : fournir une interface pour créer des familles d’objets liés sans spécifier leurs classes concrètes.

**Utilisation**: un système doit être indépendant de la façon dont ses produits sont créés, composés et représentés, et doit être configuré avec l’une des multiples familles de produits. Une famille de produits est conçue pour être utilisée ensemble, il est nécessaire de respecter cette contrainte.

**Structure**: 

**Participants** : **AbstractFactory** déclare une interface pour des opérations qui créent des objets de produit abstrait. **ConcreteFactory** implémente les opérations pour créer des objets de produit concret. **AbstractProduct** déclare une interface pour un type 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.

**Implémentation**: En général, une seule instance d’une fabrique concrète particulière est nécessaire à l’exécution (cf. plus loin « Pattern Singleton »). On a une méthode de fabrique par type de produits (que l’on surcharge pour spécifier les objets réels à créer).

On peut utiliser le pattern **Prototype** pour implémenter la fabrique concrète lorsqu’on a beaucoup de familles de produits. (la fabrique est initialisé avec un prototype de chaque produit de la famille -> plus besoin d’une nouvelle classe de fabrique concrète pour chaque nouvelle famille de produits -> nouveaux produits créer par clonage du prototype)

Méthode de fabrique vs. Fabrique abstraite :

La méthode de fabrique utilise l’héritage pour décider de l’objet à instancier. Tandis que la fabrique abstraite utilise de la composition pour déléguer la responsabilité de l’instanciation d’objets à d’autres objets (qui eux, utilisent souvent des méthodes de fabriques).

Problème : *Unicité d’une fabrique.* Comment garantir qu’il ne peut exister qu’une seule instance d’une classe, facilement accessible ?

**Singleton :** La solution passe par la création d’une instance, plus flexible et plus évolutive grâce à l’héritage (la classe assure l’unicité de son instance en interceptant les requêtes demandant à créer de nouveaux objets, et peut fournir un moyen d’y accéder).

**Intention** : garantir qu’une classe n’a qu’une seule instance et fournir un point d’accès global à cette instance.

**Utilisation** UNE seule instance d’une classe, facilement accessible aux clients, qui doit être extensible par héritage (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 (qui peut aussi avoir le rôle de créer cette instance unique). **uniqueInstance** est une variable de classe de type **Singleton**. **singletonData** est une variable d’instance (d’état) et **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, la réduction de l’espace de noms (vs. Variables globales), un nombre variable d’instances (toutes invisibles aux clients). Peut être étendu par héritage et plus flexible que les méthodes de classe.

**Implémentation**: Pour éviter les créations intempestives le constructeur est privé/protégé. L’instance unique de la classe est stockée dans une variable statique privée/protégée. Une méthode publique statique de classe **Instance()** crée l’instance au premier appel (*lazy instantiation*) et la retourne.

Héritage de la classe **Singleton** : soit la méthode **Instance** détermine la sous-classe à instancier (via un argument ou une variable d’environnement) soit chaque sous-classe fournit une méthode de classe **Instance.**

**Voir les exemples pour la différence lazy/eager instantiation** (l’une se fait dans la méthode Instance tandis que l’autre se fait lors du premier appel de la classe Singleton).

Patterns de structure

Réduit le couplage entre les classes (interface/implémentation), rend un système flexible et extensible pour un type particulier de changement structurel. (On s’intéresse à la manière dont les classes et objets sont composés)

Problème : *Complexité d’une bibliothèque* : On a beaucoup de classes avec (trop) de(s) fonctionnalités puissantes et complexes, on veut éventuellement pouvoir changer de bibliothèque. Comment réduire la complexité d’utilisation d’un sous-système ainsi que les dépendances classes-clients ?



**Façade** : On introduit une interface de plus haut niveau qui offre aux clients une vue simple du sous-système.

**Intention** : fournir une interface unifiée de haut niveau à un ensemble d’interfaces d’un sous-système (un groupe de classes), afin de le rendre plus facile à utiliser.

**Utilisation** : interface simple et unique (ce qui n’empêche pas les clients de « regarder derrière la façade »). On découple les sous-classes du client, on augmente l’indépendance et la portabilité du sous-système. Enfin, les sous-systèmes étant organisés en couches : on définit une façade par point d’entrée dans chaque couche.

**Structure**: ci-dessus.

**Participants** : **Facade** comprend le sous-système (quelle classe fait quoi) et délègue les requêtes du client aux objets appropriés. **Subsystem classes** implémentent les fonctionnalités du sous-système, traitent les requêtes de la **facade,** mais n’ont aucune référence vers elle.

**Collaboration**: Les clients communiquent avec le sous-système via la **facade,** ils n’ont pas à accéder directement au sous-système.

**Avantages** : « Cache » l’implémentation du sous-système au client, couplage faible entre les deux (modification des classes du sous-système sans affecter les clients), et n’empêche pas les clients d’accéder au sous-système directement.

**Inconvénients**: N’empêche pas les clients d’accéder au sous-système directement, l’interface peut être trop restrictive (perte de fonctionnalités).

**Implémentation**: Le couplage sous-système/clients peut encore être réduit : la classe **Facade** devient abstraite avec des sous-classes concrètes pour les différentes implémentations du sous-système. On utilise souvent le pattern **Singleton**: une seule instance d’une **Facade** est nécessaire à l’exécution.

Problème : *Incompatibilité d’interfaces*. Nous disposons d’une bibliothèque (dont nous n’avons pas les sources). L’interface est incompatible mais nous ne pouvons pas la modifier. Comment pourrait-on réutiliser ces classes ?

**Adapteur** : La solution introduit un niveau d’indirection qui réalise l’adaptation en définissant une classe **wrapper** autour de l’objet dont l’interface est incompatible.

**Intention** : convertir l’interface incompatible d’une classe en une interface attendue par le client.

**Utilisation** : On veut créer une classe réutilisable qui collabore avec des classes non liées et encore inconnues. Il faut utiliser plusieurs sous-classes existantes dont l’adaptation de leur interface par héritage est impossible (adaptateur d’objet).

**Structure**:

(adaptateur de classe)



(adaptateur d’objet)



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

2. L’adaptateur fait alors appel aux opérations d’**Adaptee** pour réaliser le service.

**Adaptateur de classe** : Une seule nouvelle classe **Adapter** est nécessairepour atteindre la classe **Adaptee** (aucune « indirection » contrairement au pattern façade)**,** hérite de l’interface client et d’**Adaptee**, mais ne permet pas d’adapter une classe et toutes ses sous-classes (le pattern façade s’occupe de ça).

**Adaptateur d’objet** : **Adapter** hérite de l’interface client, mais compose une instance de la classe sur étagère (composition d’objet : durée de vie du composant inclus dans le composé). Plus fréquent car ne nécessite pas l’héritage multiple et peut travailler avec plusieurs classes adaptées dans une hiérarchie. Oblige **Adapter** à faire référence à la sous-classe **Adaptee**.

**Implémentation** : un adaptateur sert à faire une conversion simple d’interface : il change simplement le nom des opérations et l’ordre des arguments avec un ensemble totalement différent d’opérations.

Un adaptateur bidirectionnel supporte les deux interfaces (**Target** et **Adaptee**), les objets sont donc des deux types. S’il n’est pas bidirectionnel, l’objet adapté n’est plus conforme à l’interface **Adaptee**.

Adaptateur vs. Façade : Un adaptateur réutilise une interface déjà existante (et la met en relation avec une autre) alors qu’une façade définit une *nouvelle* interface.

Problème : *Chargement des* images (coûteux). Que mettre à la place de l’image ? Comment cacher le chargement à la volée ? Comment contrôler les accès à un objet déjà utilisé ailleurs ?

**Procuration (Proxy)** : La solution passe par l’utilisation d’un objet intermédiaire qui va contrôler les accès à l’objet.

****

**Intention** : Fournir un substitut (le proxy) à un autre objet afin d’en contrôler les accès (les opérations qui lui sont appliqués).

**Utilisation** : Le client veut interagir avec un objet, mais ne peut y faire référence directement. On peut donc utiliser un :

* **Remote proxy** : fournir un représentant local d’un objet distant
* **Virtual proxy** : créer ou charger des objets coûteux à la demande
* **Copy-on-write proxy** : différer la copie d’un objet coûteux jusqu’à ce que ce dernier soit modifié
* **Protection/access proxy** : contrôler l’accès à un objet
* **Smart reference** : effectuer des opérations supplémentaires lors de l’accès à un objet
	+ telles que compter le nombre de références à l’objet (**smart pointer**)
	+ ou encore charger en mémoire un objet persistant quand il est référencé pour la première fois (**lazy instantiation**).

**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 :** Un proxy agit comme un intermédiaire entre le client et le sujet réel. Il retransmet les requêtes au **RealSubject** si nécessaire (délégation), selon le type de proxy.

**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.

**Implémentation** : Si un proxy instancie le sujet réel, il doit connaître son type, si non inutile de faire une classe **Proxy** pour chaque classe **RealSubject**. Pour faire référence au sujet réel avant son instanciation on utilise une certaine forme d’identificateurs d’objet indépendante de l’espace d’adressage (ex : nom de fichier).

Problème : *Amélioration de l’interface utilisateur* (bordure, barres de défilements…). On a n décorations qui peuvent être mélangées indépendamment (x = new ScrollBar(new Border (new Picture( . . . ) ) ) )(2n combinaisons). Comment les intégrer dans la structure physique du document ?

**Décorateur** : On définit une classe **Decorator** (enveloppe transparente) qui implémente **Glyph** (l’élément de décoration que l’on veut mettre en place), à une variable d’instance **decorated** de type **Glyph**, et a pour fils les éléments à décorer (barre de défilement etc.)



**Intention** : attacher dynamiquement des responsabilités supplémentaires à un objet, fournissant une alternative flexible à l’héritage pour étendre des fonctionnalités.

**Utilisation** : un système doit ajouter des responsabilités à des objets individuels de manière dynamique et transparente (sans affecter d’autres objets), ces responsabilités peuvent être retirées. On n’utilise pas l’héritage car il y aurait trop de sous-classes pour supporter toutes les combinaisons possibles (quand c’est possible).

**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** (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). Évite 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) et conduit à des systèmes composés de beaucoup de petits objets.

**Implémentation** : conformité d’interface (décorateurs et composants descendent d’une classe commune ; en java, une interface). Il n’est pas nécessaire de définir une classe abstraite **Decorator** s’il n’y a qu’une seule responsabilité à ajouter. La classe **Component** doit être légère car un grand nombre de fonctionnalités handicaperaient les sous-classes concrètes qui n’en ont pas l’utilité.

Décorateur vs. Proxy : Structure similaire : un niveau d’indirection à un autre objet, gèrent une référence à cet objet auquel ils retransmettent les requêtes. Mais, le décorateur ajoute dynamiquement des fonctionnalités à un objet (composition récursive) alors que le proxy contrôle les accès à un objet. Le décorateur ou le proxy, comme tout **wrapper**, augmente le nombre de classes et d’objets.

Problème : *Structure de document*. Il est représenté par sa structure physique (glyphes primitifs -> caractères, cercles, images ; lignes -> séquence de glyphes ; colonnes -> séquence de lignes ; pages -> séquence de colonnes ; documents -> séquence de pages. Comment modéliser cette structure ?

**Composite** : La solution passe par une organisation liant héritage et composition : chaque élément est réalisé par une sous-classe de la classe abstraite **Glyph**, et présente la même interface.



**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.

**Utilisation** : Un système doit représenter des hiérarchies composants / composés et les clients ne doivent pas être capable de faire la différence entre les objets individuels et leurs compositions (uniformité apparente).



**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 (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, si 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 nouveau types de composants et simplifie le code client (il ne se sait pas s’il traite avec une feuille ou un composite).

**Inconvénients** : Il est difficile de restreindre et vérifier le type des composants d’un composite (s’il ne doit contenir que certains types de composant).

**Implémentation** :

Un composite connaît ses composants enfants, ces derniers doivent-ils gérer une référence à leur parent ? Ça dépend, voir pattern Chaîne de responsabilité.

Où déclarer les opérations de gestion des fils (add, remove, getChild) ? Soit dans la classe **Component** pour la transparence : tous les composants peuvent être traités de la même manière, mais les clients peuvent effectuer des opérations qui n’ont pas de sens sur les feuilles (getChild). Soit dans la classe **Composite** pour la sûreté : toute tentative insensé sur une feuille sera interdite à la compilation, mais la transparence est perdue car maintenant les feuilles et composites ont des interfaces différentes.

L’ordonnancement des fils est-il important ? Ça dépend, voir pattern Itérateur.

Composite  vs. Décorateur : Ils ont des diagrammes de structure similaires qui reposent sur la composition récursive pour organiser un nombre ouvert d’objets mais le Composite se focalise sur la structure des classes afin que des objets individuels et leurs compositions puissent être traités uniformément (sur la représentation) alors que le Décorateur se focalise sur l’ajout dynamique de responsabilités à des objets sans utiliser l’héritage (sur la décoration). Un Décorateur étend un Composite.

Pattern de Comportement

Problème : *Les commandes utilisateur.* L’utilisateur peut effectuer plusieurs opérations (sauter à une page, copier/coller…) en cliquant sur une icône, à partir d’un menu, d’un raccourci clavier et peut défaire et refaire ces opérations. Comment réaliser un traitement sans avoir besoin de savoir de quoi il s’agit et de qui va l’effectuer ?

**Commande** : On définit une classe abstraite d’opérations utilisateur (interface commune à toutes les opérations), chaque opération est alors une sous-classe.

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

**Utilisation** : Les objets doivent être paramétrés par une action à effectuer (fonction de rappel *callback :* enregistrée à un moment donné pour être appelée ultérieurement). Un système doit permettre de spécifier, mettre en file d’attente et exécuter des requêtes à différents moments, mémoriser ses changements afin de pouvoir les refaire après un éventuel crash (persistance des modifications) et doit être structuré autour d’opérations de haut niveau (transactions) construites à partir de primitives.

**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 :**



1. Le client crée une commande et spécifie son récepteur.
2. Un objet **Invoker** stocke la commande.
3. L’invocateur appelle la méthode **Execute()** de la commande.
4. La commande invoque les opérations de son récepteur pour exécuter la requête.

**Avantages** : découple l’objet qui invoque l’opération (l’invocateur) de celui qui sait comment la réaliser (le récepteur), fait les commandes des objets de première classe (qui peuvent être manipulées et étendues comme tout objet), permet de grouper des commandes dans une commande composite (macro-commande), et facilite l’ajout de nouvelles commandes sans modifier le code existant.

**Implémentation** : une commande peut être simple (délègue simplement les actions requises à un récepteur), ou intelligente (implémente tout elle-même sans rien déléguer à un récepteur ; commandes indépendantes des classes existantes, aucun récepteur approprié n’existe). On doit stocker des informations supplémentaires et mettre en place un historique pour faire des *undo* et *redo*.

Problème : *Modèle Vue Contrôleur*. Comment faire pour que chacun d’eux soit informé de ces changements ? Maintenir un faible couplage entre diffuseur et souscripteurs ? Assurer la cohérence des classes tout en les maintenant indépendantes ?

**Observateur** : On limite le couplage entre les classes. (sinon risque de propagation d’erreurs)

**Intention** : Définir une interdépendance de type de sorte que si un objet change d’état, tous ceux qui en dépendent en soient notifiés et maj.

**Utilisation** : Une abstraction a deux aspects, l’un dépendant de l’autre et chacun pouvant être modifié et réutilisé indépendamment. La modification d’un objet nécessite d’en modifier d’autres, mais leur nombre n’est pas connu : il doit être capable d’en notifier d’autres sans savoir qui ils sont (objets découplés).

**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. 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** : Couplage faible entre le sujet et ses observateurs : possible de réutiliser les sujets sans réutiliser leurs observateurs et vice versa, des observateurs peuvent être ajoutés sans modifier le sujet. Le sujet connaît sa liste d’observateurs, n’a pas besoin de connaître la classe concrète d’un observateur (juste l’interface qu’ils implémentent). Les deux peuvent appartenir à différentes couches d’abstraction. Fournit un support à la diffusion d’évènements (broadcast) : le sujet envoie une notification à tous les observateurs abonnés et des observateurs peuvent être ajouté/supprimé à tout moment.

**Inconvénients** : Peut causer une cascade de notifications/maj inopinées : les observateurs ignorent la présence des uns et des autres, ils doivent faire attention au déclenchement des maj. Une simple interface de maj requiert que les observateurs déduisent d’eux-mêmes ce qui a changé dans le sujet (difficile) : on a besoin d’un protocole additionnel.

**Implémentation** : Le sujet garde une trace de ses observateurs via un tableau, une liste chaînée ou une table d’association. Si un observateur veut observer plus d’un sujet il faut étendre l’interface de maj pour savoir qui notifie. Chaque fois qu’un état de sujet change, il déclenche une maj. De même pour les observateurs lorsqu’ils effectuent des changements d’états. D’autres objets (tiers) peuvent maj. Si un sujet est détruit on notifie les observateurs (sans les détruire). Il faut toujours s’assurer que le sujet maj son état avant d’envoyer des notifications.

Quantité d’information concernant le changement que le sujet doit envoyer aux observateurs : *push* (beaucoup) et *pull* (très peu) (modèles). Les observateurs peuvent-ils souscrire à des évènements spécifiques ? Modèle *publish-subscribe.* Un observateur peut aussi être sujet. Si un observateur veut être notifié uniquement après que plusieurs sujets ont changé d’état on peut utiliser un objet intermédiaire (médiateur) : les sujets envoient les notifications à l’objet médiateur qui réalise un traitement nécessaire avant de notifier les observateurs.

Problème : *Vérification orthographique.* Nécessité de parcourir tout le document. Comment accéder aux objets d’une structure (un composite par exemple) sans se soucier de l’organisation de la structure ou encore d’éventuels changements ?

**Itérateur** : Une solution est de cacher la structure d’un conteneur (ici, le document) aux clients. L’itérateur doit avoir une méthode pour pointer sur le premier élément, pour avancer à l’élément suivant, obtenir l’élément courant, et tester la terminaison (isDone).

**Intention** : fournir un moyen de parcourir séquentiellement un agrégat d’éléments sans connaître sa structure interne.

**Utilisation** : Il faut accéder au contenu (les éléments) d’un agrégat sans révéler sa représentation interne, gérer plusieurs parcours simultanés d’un agrégat, offrir une interface uniforme pour parcourir différents agrégats (itération polymorphe).

**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. **Agregate** définit une interface pour la création d’un objet **Iterator** (méthode de fabrique). **ConcreteAggregate** implémente l’interface de création d’**Iterator**, afin de retourner une instance appropriée de **ConcreteIterator**.

**Avantages** : Simplifie l’interface de l’agrégat (pas de méthodes de parcours), permet de gérer plusieurs parcours simultanés, de modifier l’algorithme de parcours (en remplaçant l’instance de l’itérateur par une autre différente ou en définissant une nouvelle sous-classe de **Iterator**).

**Implémentation** : Pour la flexibilité, le client contrôle l’itération (externe) ; mais c’est plus facile d’utilisation si l’itérateur contrôle l’itération (interne). L’algorithme de parcours peut être définit par l’agrégat, l’itérateur ne conserve que l’état de l’itération (curseur), ou l’itérateur, plus courant et facilite l’utilisation de différents algorithmes d’itération sur le même agrégat ou le même algorithme sur différents agrégats (attention : ne pas casser l’encapsulation de l’agrégat pour accéder à ses variables privées !). Un itérateur robuste permet de faire des insertions et des suppressions sans affecter le parcours ni faire de copie de l’agrégat. On peut éventuellement enrichir son interface (previous()… ).

Problème : *Vérification orthographique (suite).* Le pattern itérateur fonctionne bien si on n’a pas besoin de connaître le type des éléments en train d’être parcourus (on aurait besoin d’un cast). Comment traiter des éléments hétérogènes ?

**Visiteur :** 

 Une solution est d’encapsuler l’opération désirée dans un objet séparé. L’aiguillage dynamique sur la méthode **accept** de la classe **Glyph** permet un cast sans erreur de type (méthodes **accept** des classes **Character, Picture,**...). Chaque méthode **accept** appelle l’action spécifique au visiteur passé en argument (**visit(Character)**) et implémente la traversée (méthode **accept** de la classe **Line**). Il faut un visiteur pour chaque action (rechercher-remplacer par exemple).

**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.

**Utilisation** : Une structure d’éléments contient beaucoup de classes avec des interfaces différentes et les opérations à effectuer sur ces éléments dépendent de leur classe concrète. Beaucoup d’opérations indépendantes doivent être effectuées sur les éléments d’une structure et il faut éviter de polluer leur classe avec ces opérations. Les classes définissant la structure d’éléments changent rarement, mais il faut souvent définir de nouvelles opérations sur cette structure.

**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, et ce dernier détermine alors la classe concrète de l’élément visité et accède à cet élément directement à travers son interface). **ConcreteVisitor** implémente chaque opération (pour la classe d’élément correspondante) déclarée par **Visitor**, fournit le contexte de l’algorithme et stocke son état local (pour accumuler des résultats lors du parcours de la structure par exemple). **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, il 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 le 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** plus 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 con encapsulation).

**Implémentation** : *Single-Dispatch* (rappel) : nom de l’opération + type du receveur (utilisation du polymorphisme). *Double-Dispatch* : c’est la clé du pattern Visiteur : permet d’ajouter des opérations à des classes sans les changer, nom de l’opération plus types des deux receveurs (le visiteur et l’élément), méthode **Accept** pour effectuer la liaison à l’exécution.

Le responsable du parcours de la structure peut être le plus souvent la structure d’éléments (mais figée), ou le visiteur, solution flexible mais duplication du code de parcours dans chaque **ConcreteVisitor** pour chaque **ConcreteElement.** La principale raison pour mettre la stratégie de traversée dans le visiteur est d’implémenter un parcours particulièrement complexe qui dépend des résultats des opérations sur la structure d’éléments. L’itérateur aussi peut en être responsable (retour aux deux cas précédents).

p u b l i c c l a s s S p e l lCh e c k i n g implements V i s i t o r {

p u b l i c void v i s i t ( Ch a r a c t e r c ) { . . . }

p u b l i c void v i s i t ( Pi c t u r e p ) {} //do n o t h i n g

p u b l i c void v i s i t ( Li n e l ) { . . . }

. . .

}

. . .

V i s i t o r v = new S p e l lCh e c k i n g ( ) ;

l i n e . a c c ep t ( v ) ;

Itérateur vs. Visiteur: Le pattern Itérateur fournit des parcours de conteneurs, le pattern Visiteur est plus général et permet le parcours et des actions spécifiques aux types d’éléments. L’idée est de séparer le parcours des actions, avoir une méthode « faire ça » pour chaque type d’éléments (peut être redéfinir pour un parcours particulier).

Problème : *Sauvegarde d’un document*. La sauvegarde d’un document est différente selon si c’est un nouveau doc, ou déjà existant, ou qu’il y a eu un crash… Comment structurer et représenter les différents états d’un système (et les transitions entre ces états) dont dépendent certains de ses comportements.

On pourrait utiliser des constantes pour représenter les états dans des confitionnelles, mais mauvaise solution.

Etat :



La solution met chaque branche des conditionnelles dans une classe à part et créer des classes d’états qui implémentent une interface commune.

**Intention** : permettre à un objet de modifier son comportement lorsque son état interne change.

**Utilisation** : Le comportement d’un objet dépend de son état et il doit changer son comportement à l’exécution en fonction de cet état. Des opérations ont plusieurs grandes parties avec des conditionnelles qui dépendent de l’état de l’objet.

**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, une sous-classe de **State** par exemple), facilite l’ajout et la suppression des états et des transitions, élimine les conditionnelles et rend les transitions entre états plus explicites.

**Inconvénients** : augmente le nombre d’objets.

**Implémentation** : Les transitions entre états sont définit soit par la classe **Context** (pour des situations simples), soit les classes **ConcreteState** (généralement plus flexible, mais entraîne des dépendances d’implémentation entre les classes **ConcreteState**). Les objets **ConcreteState** sont créés soit lorsque nécessaire (les états ne sont pas connus à la compilation et le contexte change d’état rarement), soit une seule fois en avance, le contexte gère des références à ces objets (les changements d’état arrivent fréquemment).

 Problème : *Mise en forme.* Mise en forme (retours à la ligne), beaucoup d’alternatives algorithmiques (simples et complexes). On veut pourvoir en changer et ne pas polluer la classe **Glyph**. Comment représenter et utiliser différentes variantes d’un même algorithme pour un comportement particulier ?



Stratégie : Encapsuler la forme (l’algorithme) derrière une interface (chaque algorithme de mise en forme est mis dans une classe, **Document** traite uniquement avec l’interface).

**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.

**Utilisation** : de nombreuses classes liées ne diffèrent que par leur comportement, les clients ont besoin de différentes variantes d’un même algorithme à différents moments (différentes complexités en temps/mémoire). Un algorithme utilise des structures de données complexes que les clients n’ont pas à connaître. Une classe définit plusieurs comportements qui figurent sous la forme de conditionnelles dans ses opérations.

**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** : 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) et 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 du contexte (modification indépendantes, changement dynamique - à l’exécution – de l’algorithme, extension et compréhension facilitées), élimine les conditionnelles pour sélectionner le bon comportement et fournit différentes implémentations du même comportement (les clients peuvent choisir des stratégies avec des compromis différents en temps/mémoire).

**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 et tous les algorithmes doivent utiliser la même interface **Strategy** (pas nécessaire pour toutes les implémentations de **Strategy**, surcoût lié à la communication entre **Strategy** et **Context**).

État vs. Stratégie : Les patterns état et stratégie semblent faire une utilisation du polymorphisme quasiment identique sur le plan structurel (tous deux sont des exemples de composition avec délégation) mais n’ont pas le même objectif ! Un objet **State** encapsule un comportement d’états (et éventuellement les transitions entre états) alors qu’un objet **Strategy** encapsule un algorithme.