**Design Pattern**Pattern : Solution de construction pour un problème donné. Il s’applique à tous les langages objet. Il a un nom concis et significatif, un problème associé, une solution sous forme de schéma (UML) et des avantages/inconvénients.  
Il y a 4 types de patterns : **de création**, patterns **de structure**, patterns **de comportement** et patterns **de portée**.

**Patterns de création :** Méthode de fabrique - Fabrique abstraite - Singleton :

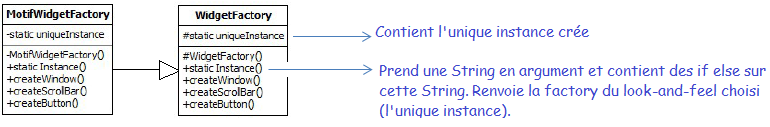
**Problème du look-and-feel (Patterns de création : méthode de fabrique & fabrique abstraite)**

Problème : On veut changer le type de choses affichées en fonction de variables. Par exemple, une appli a un skin d’Halloween à l’approche de Noël.

Solution : On créé une classe Creator pour chaque élément qui peut différer. (Méthode de fabrique).  
On créé également une classe abstraite qui regroupe tous les éléments correspondants à une même skin. (On créé une classe Halloween qui appelle BouttonNoelCreator, ScrollbarNoelCreator, …) (Méthode de fabrique abstraite). Pour garantir la cohérence du look-and-feel choisi (même thème), on ne créé qu’une seule instance de factory/créateur (Singleton).

Fabrique : 

Fabrique abstraite : 

Singleton : 

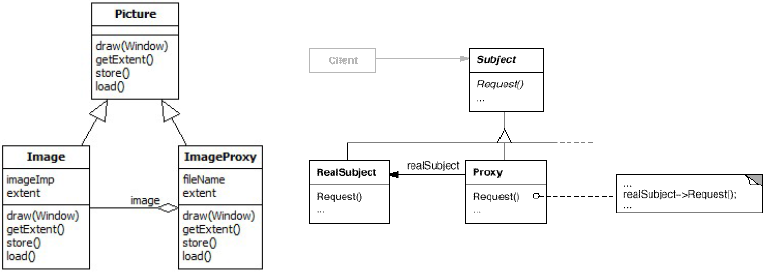
**Patterns de structure :** Façade – Adaptateur – Procuration – Décorateur – Composite :   
Exemple : Il y a une librairie de code pour manipuler les images. Nous n’avons pas besoin de tout. On créé une interface qui n’appelle que quelques fonctions de la librairie. Elle a des noms de fonctions simple et quand on les appelle, elle délègue le travaille à la bonne fonction de la librairie. La classe Façade peut être abstraite et appeler des sous-classes concrètes pour les différentes implémentations. En général il n’y a qu’une seule instance de **Façade** 🡪 Utilisation du pattern Singleton.

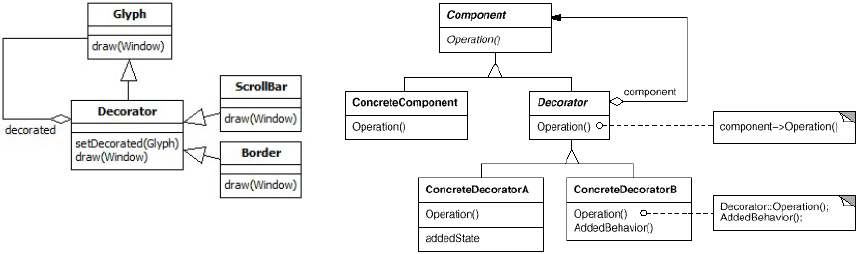
Problème : On ne peut pas utiliser une bibliothèque (pour manipuler des images par exemple) car son interface est incompatible avec l’interface requise par l’éditeur de texte (ex : la classe Glyph)

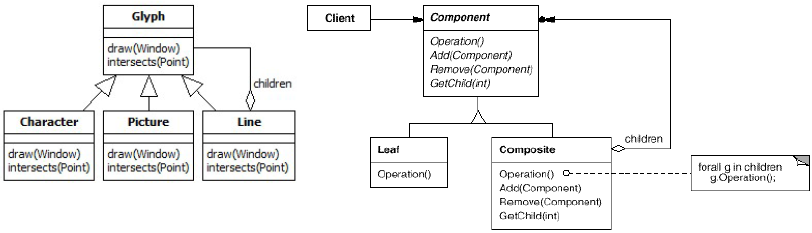
Solution : On créer une classe qui reçoit les requêtes et qui les **adapte** avant d’appeler les méthodes de la seconde interface. Elle hérite des 2 interfaces quand c’est un adaptateur de classe.

Problème : Les images ralentissent l’ouverture du document. On ne veut pas toutes les ouvrir à l’ouverture car elles ne sont pas toutes visibles en même temps sur le document (images en bas par exemple).

Solution: Pattern de **procuration**: les requêtes sont envoyées à un **proxy**, qui servira d’intermédiaire et contrôlera l’accès aux objets.  
  
Subject est une interface commune pour RealSubject et Proxy.  
Le proxy peut délèguer les requêtes à RealSubject si nécessaire.  
Problème : Nous voulons décorer les élémT de l’interface utilisateur. Bcp de décos peuvent ê mélangées indépendement (ex : x= new ScrollBar(new Border(nex Picture(…))) pour n décos : 2n combinaisions).

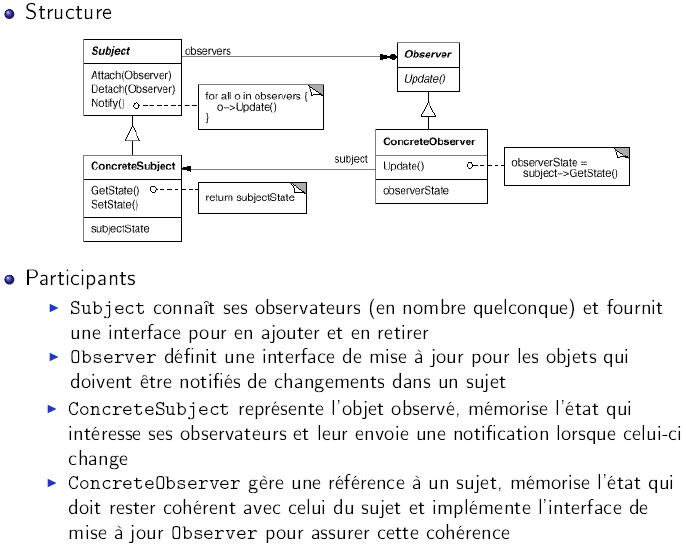
Solution : Définir une classe **Decorator** qui implémente Glyph et a une variable d’instance de type Glyph. Les décos étendent cette classe.  
  
Component est l’interface des objets pouvant ê décorés. Decorator gère une réf à un Component et définit une interface. ConcreteDecorator ajoute des responsabilités au composant décoré.  
Problème : Un document est représenté par sa structure physique : glyphes primitifs, lignes, colonnes, pages, …

Solution : **Pattern composite** : Component déclare l’interface des objets de la composition.  
Leaf rpz les objets feuilles (sans fils) de la composition.  
Composite définit le comportemT des composants pères, stocke les fils et implémente les opérat° nécessaires à leur gestion.

Si l’objet est une feuille, la requête est traitée, sinon c’est un composite, et il la transmet à son fils (possible que ce soit encore un composite). Il peut éventuellement effectuer des traitemenT avant et/ou après. Glyph a une liste de glyphs en attribut (ses fils).

**Patterns de comportement :** Commande – Observateur – Itérateur – Visiteur : Problème : Les opérations peuvent être effectuées de plusieurs manières (raccourci clavier, clic sur icône, menu, …).

Solution : **Pattern commande** : n créé un classe abstraite Commande dont héritera toutes les commandes possibles. Chaque opération est une sous-classe. On encapsule la requête comme un objet permettant ainsi de la manipuler de différentes manières.

Problème : On veut modifier plusieurs objets en même temps car ils dépendent d’une même chose. S’il y en a un qui change les autres devront changer aussi.

Solution : **Pattern observateur** : On créé une classe Observer qui maj les objets quand il y a un changement. Les objets doivent notifier son/ses observer(s) quand ils sont modifiés. Comme ça l’observer modifie les autres. Un objet doit connaître tous ses observer. Si un observer veut être notifié seulement après que plusieurs objets aient changé d’état, on créé un objet intermédiaire qui reçoit les notifications et réalise un traitement avant d’en informer l’observer.

Problème : vérification orthographique  
Solution : Cacher la structure d’un conteneur (ici le document) aux clients.  
Une méthode pour : - pointer sur le 1er élément - Avancer à l’élément suivant - Obtenir l’élément courant - Tester la terminaison  
Il faut accéder au contenu (ie les éléments) d’un agrégat sans révéler sa rpz° interne. Les **itérateurs** sont svt utilisés pour traverser récursivement des structures composites.  
Iterator : interface pour accéder aux éléments Agregate : interface pour créer un Iterator (fabrique).  
ConcreteIterator : implémente l’interface Iterator , garde une trace de l’élément courant et détermine l’élément suivant.  
ConcreteAgregate : implémente l’interface Agregate afin de retourner une instance appropriée de ConcreteIterator.

Problème : avec le pattern Iterator, on a besoin de caster les items car il ne prend pas en compte le type d’item.  
Solution : 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. Chaque méthode accept appelle l’action spécifique au visiteur passé en argument. Il faut un visiteur pour chaque action.  
Intent° : Rpz une pérat° 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  
  
Visitor : déclare une opérat° visit pour chaque classe de ConcreteElement. Le visiteur détermine la classe concrète de l’élément visité et accède à cet élément à travers son interface.  
ConcreteVisitor : implémente chaque opération, et stocke l’état local des algos  
Element : déffinit l’opérat° 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 permet au visiteur de les visiter.  
  
Un client créé un visiteur pour parcourir les éléments d’une structure et faire un traitement variable selon leur type.  
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.  
  
  
  
  


**Patterns de portée :**

Tout est fiché