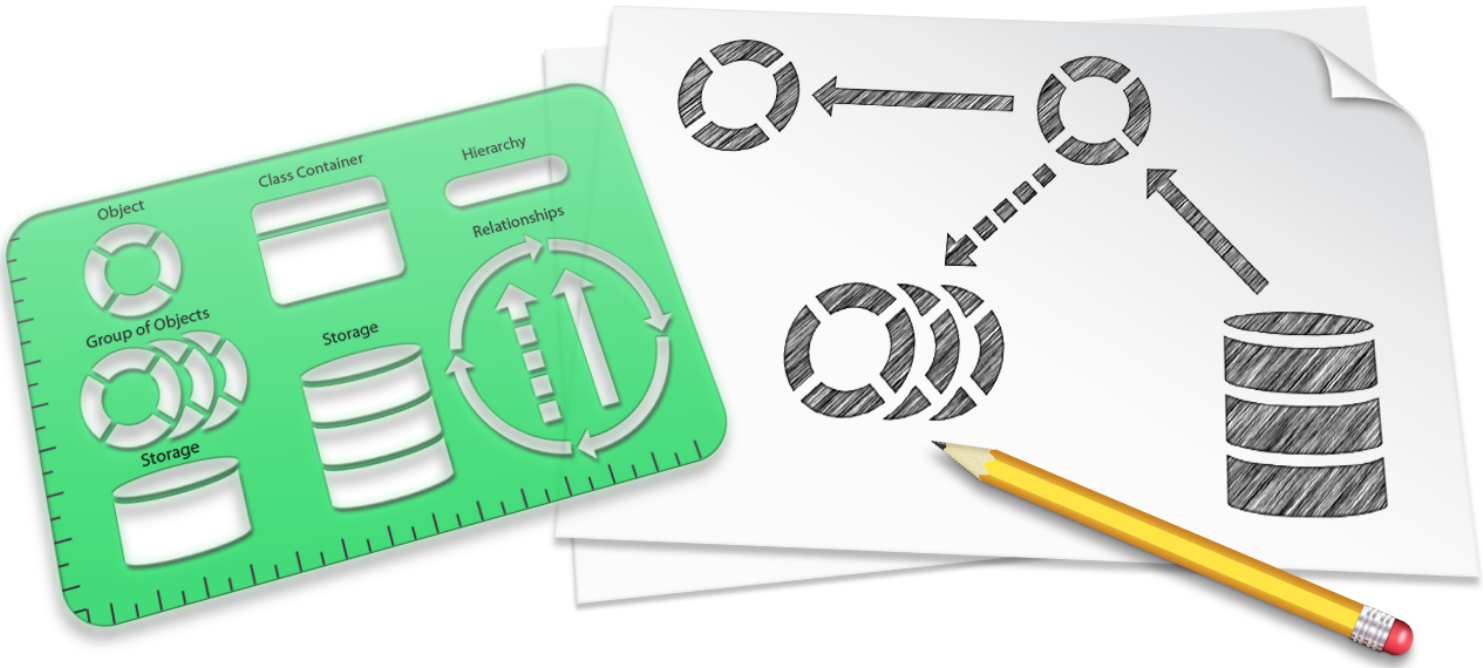
Introduction aux *Design Patterns*



Objectifs du module

- (Re-)connaître plusieurs *design patterns* et savoir les mettre en œuvre dans un langage de programmation par objet
- Comprendre le rôle que peuvent jouer les *design patterns* dans la conception d'architecture de systèmes complexes
- Penser la conception objet différemment
  - Plus flexible, plus modulaire, plus réutilisable, plus compréhensible

Au menu

Au menu

Motivation (1)

- Concevoir du logiciel est difficile
  - Cela nécessite de chercher :
    - ★ Une bonne décomposition du problème
    - ★ De bonnes abstractions logicielles
    - ★ Flexibilité, extensibilité, modularité et élégance
  - Émerge souvent d'un processus itératif (nombreux essais et erreurs)
- Toujours pas de recettes magiques, mais de bonnes pratiques :
  - *KISS : Keep It Simple, Stupid*
  - *DRY : Don't Repeat Yourself*
  - *SOLID*
    - ★ *Single responsibility principle* : une classe n'a qu'une seule responsabilité (ou préoccupation)
    - ★ *Open/closed principle* : une classe doit être ouverte à l'extension mais fermée à la modification
    - ★ *Liskov substitution principle* : les objets doivent pouvoir être remplacés par des instances de leurs sous-types sans "casser" le programme
    - ★ *Interface segregation principle* : il vaut mieux plusieurs interfaces spécifiques qu'une unique interface générique
    - ★ *Dependency inversion principle* : il faut dépendre des abstractions, pas des réalisations concrètes

Motivation (2)

- Concevoir du logiciel *réutilisable* est encore plus difficile
- Bonne nouvelle : des conceptions réussies existent
  - Elles présentent certaines caractéristiques récurrentes
  - Mais ne sont pratiquement jamais identiques
- Peut-on décrire, codifier et standardiser ces bonnes conceptions ?
  - Disposer de briques de conception réutilisables
  - Produire plus rapidement du logiciel meilleur

Un (tout petit) peu d'histoire

- Christopher Alexander
  - Architecte
  - Professeur émérite à Berkeley
  - Père des *design patterns* appliqués à l'architecture (2543 patterns)
  - *A Pattern Language: Towns, Buildings, Construction* (1977)

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

- Kent Beck et Ward Cunningham
  - *Using Pattern Languages for Object-Oriented Programs* à OOPSLA'87
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (*GoF*)
  - Application des *design patterns* à la programmation par objet
  - *Design Patterns: Elements of Reusable Object-Oriented Software* (1994)
    - ★ Catalogue de 23 *design patterns*

Qu'est-ce qu'un *design pattern* ?

Une définition

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

- En français, patron de conception
- Ce N'est PAS :
  - Une classe individuelle ou bibliothèque, telle que les listes ou les tables d'association
  - Une conception complète et concrète, ni une implémentation, mais plutôt une description abstraite sur comment résoudre un problème

Éléments d'un *design pattern*

- Nom
  - Concis et significatif
  - Partie utile du vocabulaire de conception
- Problème résolu et applicabilité
  - Quand appliquer le pattern : problème + contexte
- Solution
  - Représentée sous la forme d'un schéma (*e.g.*, diagrammes UML)
  - Participants (classes et objets) et leurs relations/responsabilités/ collaborations
  - Ils doivent être personnalisés
- Conséquences
  - Avantages et inconvénients d'utiliser le pattern
  - Impacts sur la réutilisation, la flexibilité, l'extensibilité, *etc.*
  - Ils peuvent être différentes en fonction des variations du pattern

Bénéfices des *design patterns*

- Vocabulaire de conception commun
  - Améliore la compréhension et la documentation de la conception
  - Améliore la communication entre les développeurs
- Capture de l'expérience de conception
  - Facilite la réutilisation de solutions éprouvées vis-à-vis du problème
  - Facilite l'implémentation, la maintenance et l'évolution du logiciel
  - Augmente la productivité et la qualité du logiciel
- Enseignement et apprentissage
  - Est plus facile de comprendre une architecture à partir de descriptions de design patterns que de lire du code
  - Permet de comprendre certains schémas et patrons présents dans les composants

Un peu de lecture

- *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson et Vlissides, Addison-Wesley, 1994
- *Pattern Hatching : Design Patterns Applied*, Vlissides, Addison-Wesley, 1998
- *Design Patterns Explained: a New Perspective on Object-Oriented Design*, Shalloway et Trott, Addison-Wesley, 2001
- *Head First Design Patterns*, Freeman et Freeman, O'Reilly, 2004

D'autres types de *patterns*

- Styles architecturaux
  - Ils caractérisent une famille de systèmes qui utilise les même types de composants, d'interactions, de contraintes structurelles et sémantiques, et d'analyses (*e.g.*, *pipes*, *filters*, *brokers*, *blackboard*, *MVC*)
  - Décrits au moyen des termes et des concepts du domaine d'application
  - Niveau d'abstraction/granularité $>$ *design patterns*
- Idiomes de programmation
  - Spécifiques à un langage de programmation particulier
  - Décrits au moyen des constructions du langage de programmation
  - Niveau d'abstraction/granularité $<$ *design patterns*
- *Framework*
  - Modèle d'architecture utilisé pour organiser une application complète
  - Par exemple, J2EE pour les applications Internet en Java
- Anti-patterns
  - Ce qu'il ne faut pas faire (mauvaises solutions)
- Schémas d'organisation
  - Tout ce qui entoure le développement d'un logiciel (humains)
- *Portland Pattern Repository* : `http://c2.com/cgi/wiki`

Comment utiliser les *design patterns* ?

1. Connaître les patterns et les problèmes récurrents qu'ils adressent
2. Durant la phase de conception, identifier les problèmes qui peuvent être résolus par un pattern
3. Consulter le pattern
4. Intégrer correctement le pattern dans le code
   - Déterminer quelles classes devraient remplacer les « stéréotypes » fournis par le pattern
   - Parfois, un pattern ne s'applique pas directement
     - ★ Besoin de l'adapter à la situation
     - ★ Besoin d'utiliser plusieurs patterns pour résoudre le problème

Classification des *design patterns* (*GoF*)

- But : ce que fait le pattern
  - Patterns de création
    - ★ Ils rendent un système indépendant de comment ses objets sont créés, initialisés et configurés
    - ★ Ils sont utiles lorsque le système évolue : les classes qui seront utilisées dans le futur peuvent ne pas être connues maintenant
  - Patterns de structure
    - ★ Ils permettent de composer des classes et des objets afin de former des structures plus importantes
    - ★ Ils réduisent le couplage entre des classes (découplage interface et implémentation)
  - Patterns de comportement
    - ★ Ils s'intéressent aux interactions entre les objets
    - ★ Ils décrivent des flots de contrôle complexes

Classification des *design patterns* (*GoF*)

- Portée : à quoi s'applique le pattern
  - Patterns de classe
    - ★ Ils se focalisent sur les relations entre les classes et leurs sous-classes (héritage)
    - ★ Ces relations sont établies statiquement
  - Patterns d'objet
    - ★ Ils se focalisent sur les relations entre les objets (composition)
    - ★ Ces relations sont établies dynamiquement et modifiées à l'exécution

Exemple : un éditeur de texte

Caractéristiques

- *WYSIWYG*
- Interface graphique
  - Barre d'outils, barres de défilement, menus *etc.*
- Possibilité de mélanger librement texte et images
- Plusieurs systèmes de fenêtrage
- Vérification orthographique, césure, *etc.*

Pédagogie

- Classe sous Microsoft Teams
  - Diapos du cours, feuilles de TD (+ matériel) et corrigés
  - Poser des questions hors créneaux de cours
  - Informations relatives au module et à votre groupe
- Déroulement des séances
  - Avant chaque séance, lire la partie du cours du pattern à étudier
    - ★ Exception pour la première séance
  - Comprendre le problème de conception énoncé dans la feuille de TD
  - Tableau divisé en deux parties :
    - ★ Architecture UML (avec correspondance des classes stéréotypes)
    - ★ Code Java (*via* replit)
  - Questions-discussion (très important !)
- Évaluation
  - Examen semestriel de 2h sur feuille

Au menu

Patterns de création

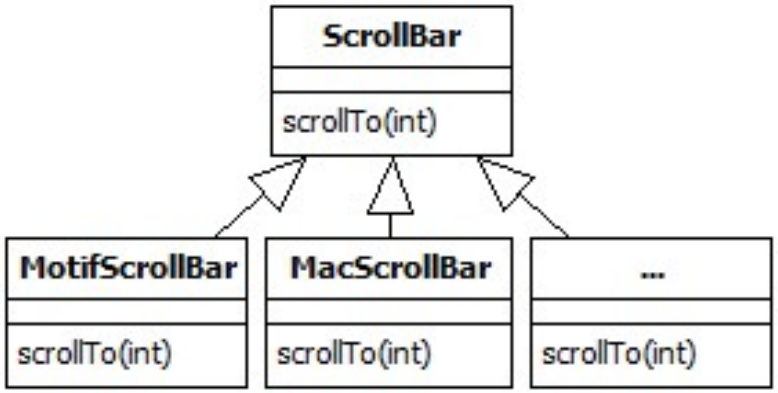
- Ils abstraient le processus d'instanciation (vs l'opérateur `new`)
  - En encapsulant la connaissance des classes concrètes que le système utilise
  - En cachant la manière dont les instances de ces classes sont créées et assemblées
  - Tout ce que le système connaît de ces objets est leurs interfaces telles que définies par des classes abstraites
- Ils offrent une grande flexibilité en *ce qui* doit être créé, *qui* doit le créer, *comment* le créer et *quand*
- Ils permettent de configurer statiquement ou dynamiquement un système avec des objets « produits »

Problème : *look-and-feel*

- Nous souhaitons que l'éditeur de texte supporte différents standards de *look-and-feel*
  - Apparence des barres de défilement, menus, bordures, *etc.*
  - Que faut-il écrire dans le code pour créer ces différents composants d'interface graphique (*widgets*) selon le *look-and-feel* courant :

```
ScrollBar sb = new ?
```

- Autrement dit, disposant d'une hiérarchie de classes, comment créer de manière flexible les instances ?
- Que proposez-vous comme conception ?



Pas très bon

- Très mauvais :

```
ScrollBar sb = new MotifScrollBar(...);
```

- Un peu meilleur :

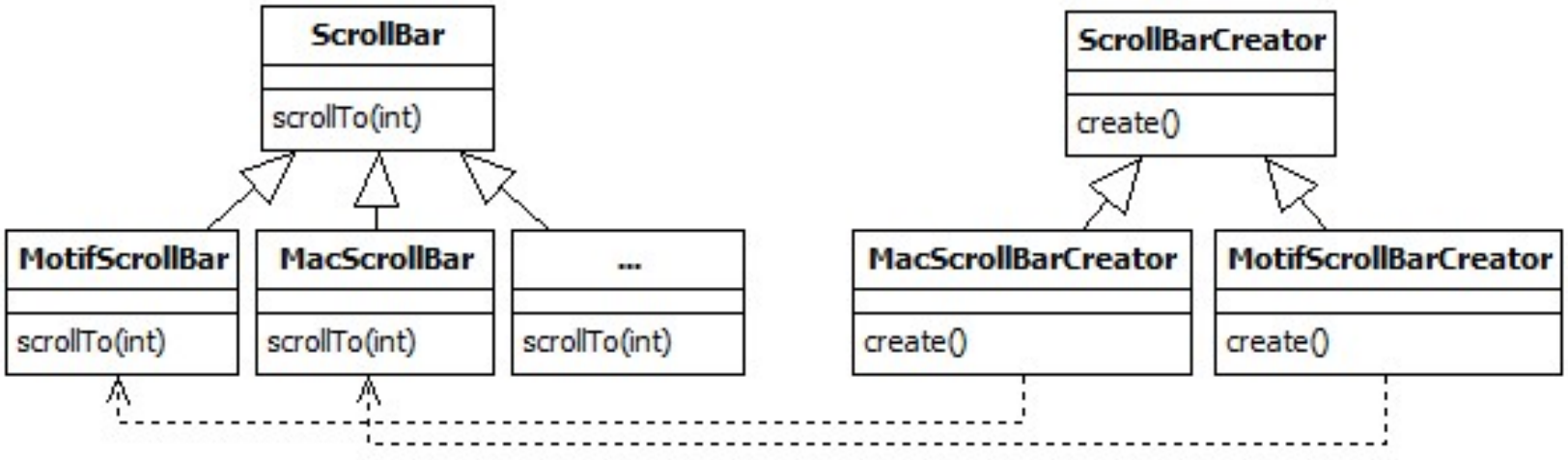
```
ScrollBar sb;
if (style == MOTIF) {
  sb = new MotifScrollBar(...);
} else if (style == ...) {
  sb = new ...
}
```

  - Conditionnels similaires pour les menus, les bordures, *etc.*

- Autre solution :
  - Introspection quand cela est permis

Abstraire la création d'objet

- Encapsuler ce qui varie dans une classe (ici, la création d'objet)
  - Pouvoir créer différents menus, barres de défilement, *etc.* selon le *look-and-feel* courant
- Définir une classe `ScrollBarCreator` avec une méthode abstraite d'instance `create`
- Définir des sous-classes de `ScrollBarCreator` pour chaque *look-and-feel*, redéfinissant la méthode `create`

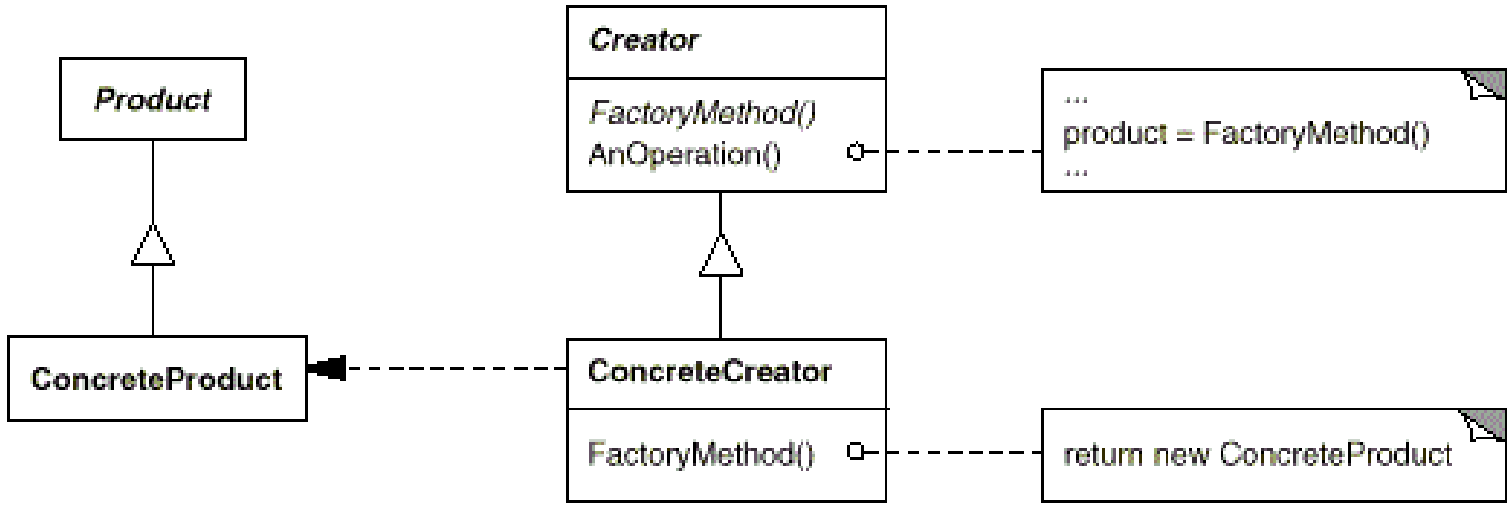


Méthode de fabrique (*Factory Method*)

- Intention
  - Définir une interface pour la création d'un objet, en laissant à ses sous-classes le soin de choisir la classe concrète de l'objet à instancier
- Solution
  - Encapsuler la création d'instances d'objet dans une classe abstraite dédiée
- Métaphore
  - Une chaîne de pizzeria propose une carte de pizzas personnalisée dans chacune de ses franchises
  - Toutefois, c'est toujours la même procédure pour préparer une pizza, seuls les ingrédients changent
  - On factorise donc la méthode de préparation des pizzas et on spécialise leur création effective
- Indications d'utilisation
  - Une classe ne peut pas anticiper la classe des objets qu'elle doit créer
  - Une classe attend de ses sous-classes qu'elles spécifient les objets qu'elle crée

Méthode de fabrique (*Factory Method*)

- Structure



- Participants
  - `Product` définit l'interface des objets créés par la méthode de fabrique
  - `ConcreteProduct` implémente l'interface `Product` avec un produit concret
  - `Creator` déclare la méthode de fabrique qui retourne un objet de type `Product`
  - `ConcreteCreator` redéfinit la méthode de fabrique pour retourner une instance d'un `ConcreteProduct`

Méthode de fabrique (*Factory Method*)

- Collaboration
  - 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
    - ★ Déterminé 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
- Exemple dans les *frameworks*
  - Un *framework* est construit avec la classe abstraite `Creator` des objets, et l'implémentation exige des instances concrètes créées par les classes dérivées de la fabrique

Méthode de fabrique (*Factory Method*)

- 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

Méthode de fabrique (*Factory Method*)

- Implémentation
  - La classe `Creator` peut être :
    - ★ Abstraite (*doit* être spécialisée) et n'implémente pas la méthode de fabrique
    - ★ Concrète (*peut* être spécialisée) et fournit une implémentation par défaut de la méthode de fabrique
  - Une méthode de fabrique devrait-elle pouvoir créer différents types de produits ?
    - ★ Si oui, la méthode de fabrique prend un paramètre (possiblement utilisé dans un bloc `if-else`) pour décider quel objet créer

Retour à notre exemple

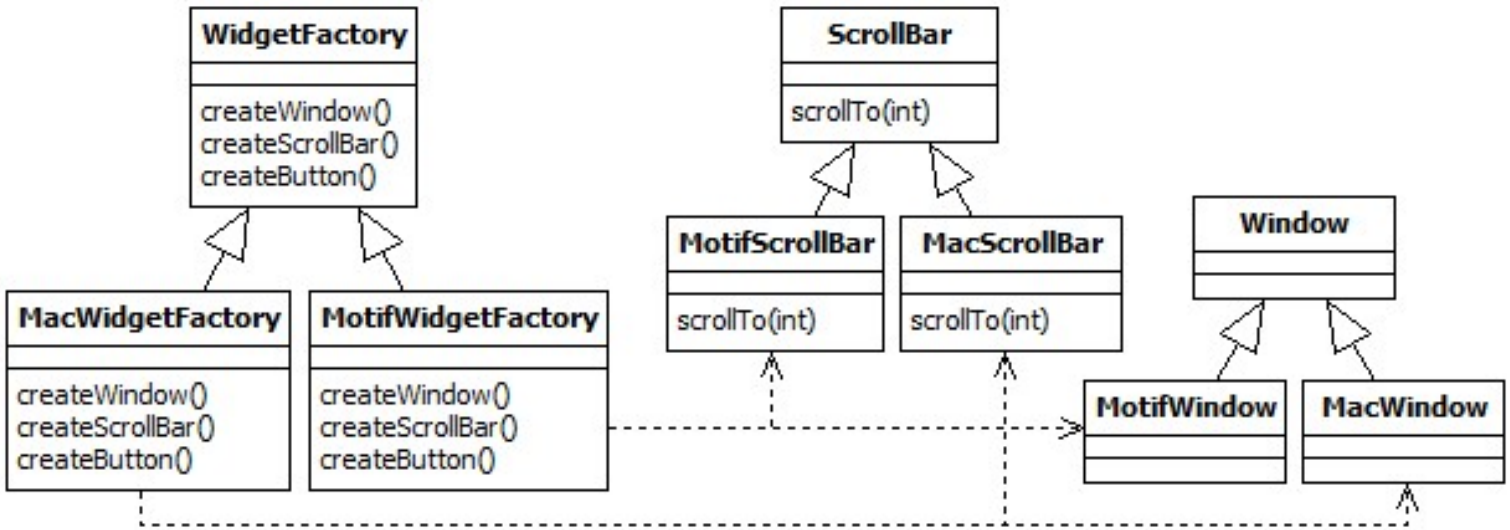
```
public abstract class ScrollBarCreator {
  public abstract ScrollBar create();
}

public class MotifScrollBarCreator extends ScrollBarCreator {
  public ScrollBar create() {
    return new MotifScrollBar();
  }
}

public class MacScrollBarCreator extends ScrollBarCreator {
  public ScrollBar create() {
    return new MacScrollBar();
  }
}
...
ScrollBarCreator sbc = new MotifScrollBarCreator();
ScrollBar sb = sbc.create();
```

Abstraire la création d'objet - suite

- La méthode de fabrique permet de changer une méthode de classe en une méthode d'instance
- Mais ce n'est pas suffisant pour notre problème
  - Ajout d'une hiérarchie de classes auxiliaires
  - Permet de gérer et séparer cette construction des classes utilitaires

Abstraire la création d'objet - suite

- Définir une classe `WidgetFactory`
  - Abstrait la création d'une famille d'objets
    - ★ Une méthode `create` pour créer chaque *widget*
  - Différentes instances fournissent des implémentations alternatives à cette famille
    - ★ Des sous-classes de `WidgetFactory` pour chaque *look-and-feel*
  - Un objet `WidgetFactory` pour créer les différents *widgets* du *look-and-feel* courant
    - ★ Variable globale
    - ★ Peut être changé à l'exécution



Fabrique abstraite (*Abstract Factory*)

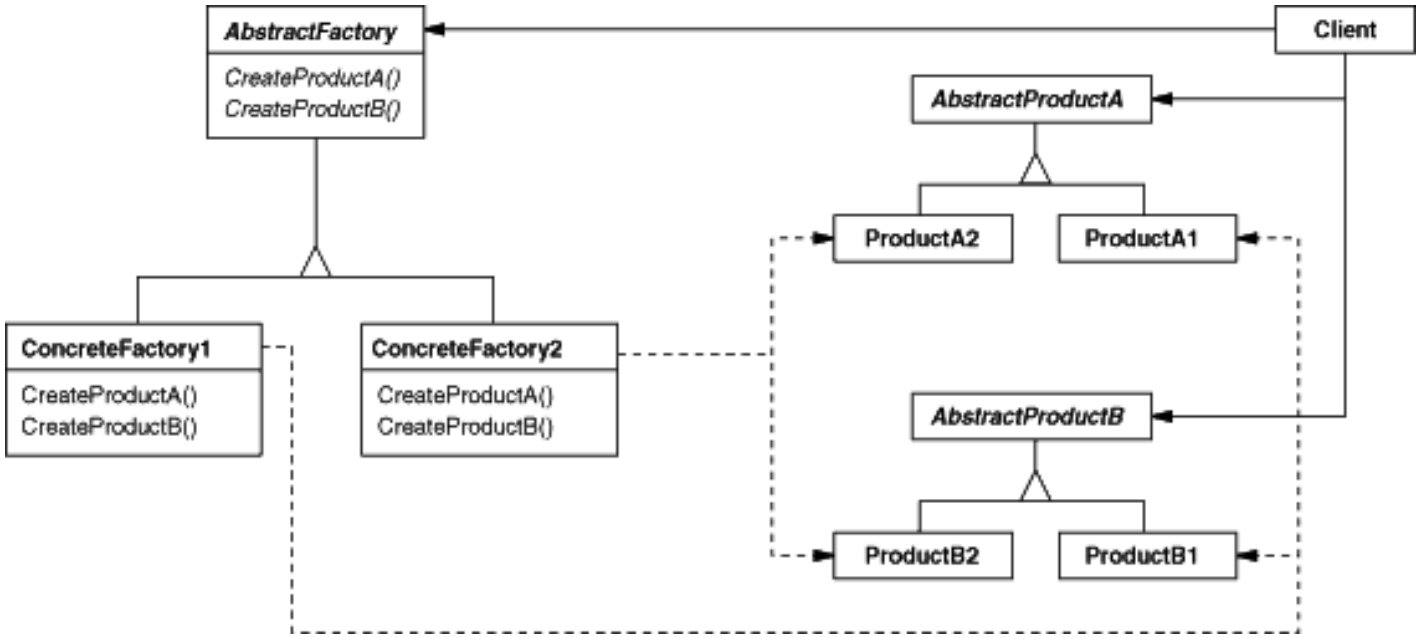
- Intention
  - Fournir une interface pour la création de familles d'objets apparentés ou interdépendants, sans qu'il soit nécessaire de spécifier leur classes concrète
- Solution
  - Coordonner la création de fabrique d'objets : une méthode consiste à extraire les règles d'instanciation de l'objet client qui utilise les objets créés
- Métaphore
  - Le montage d'une voiture correspond toujours aux mêmes étapes, seuls les éléments de la voiture changent (*e.g.*, les portes, le capot)
  - Pour le montage d'une voiture d'un modèle particulier il est nécessaire de récupérer tous les éléments appartenant à ce même modèle
  - Il est donc important de localiser la récupération des éléments d'un même modèle de voiture à un seul endroit : une fabrique abstraite d'élément
  - Chaque modèle possède sa propre fabrique concrète qui est dérivée de la fabrique abstraite

Fabrique abstraite (*Abstract Factory*)

- Indications d'utilisation
  - Un système doit être indépendant de comment ses produits sont crées, composés et représentés
  - Un système doit être configuré avec l'une des multiples familles de produits
  - Une famille de produits est conçue pour être utilisée ensemble et il est nécessaire de respecter cette contrainte

Fabrique abstraite (*Abstract Factory*)

- 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 qui doit être créé par la fabrique concrète correspondante
  - `Client` utilise uniquement les interfaces déclarées par les classes abstraites `AbstractFactory` et `AbstractProduct`

Fabrique abstraite (*Abstract Factory*)

- Collaboration
  - Une instance unique de la classe `ConcreteFactory` est créée à l'exécution
  - 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` délègue la création des produits à ses sous-classes concrètes
- Exemple dans l'API Java
  - La classe `java.awt.Toolkit` est la classe abstraite de toutes les implémentations concrètes du *toolkit* du système de fenêtrage abstrait AWT
  - Les sous-classes de la classe `Toolkit` sont utilisées pour lier les différents composants graphiques à des implémentations particulières du *toolkit* natif de la machine hôte

Fabrique abstraite (*Abstract Factory*)

- Avantages
  - Isole les clients des classes concrètes, car ils manipulent seulement des interfaces abstraites
    - ★ Encapsulation de la prise en charge et de la création des objets produits
  - Facilite 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 types de produit parce qu'il faut changer l'interface de la fabrique abstraite, et donc étendre l'interface de toutes les classes de fabrique concrète

Fabrique abstraite (*Abstract Factory*)

- Implémentation
  - En général, une seule instance d'une fabrique concrète particulière est nécessaire à l'exécution
    - ★ Utilisation du pattern Singleton
  - `AbstractFactory` ne fait que déclarer une interface pour la création de produits, c'est aux sous-classes `ConcreteFactory` de les créer effectivement
  - Une méthode de fabrique par type de produit (la surcharger pour spécifier les objets réels à créer)
  - S'il y a un grand nombre de familles de produits, la fabrique concrète peut être implémentée en utilisant le pattern Prototype
    - ★ Elle est initialisée avec un prototype de chaque produit de la famille et crée un nouveau produit par clonage de son prototype
    - ★ Cela élimine la nécessité d'une nouvelle classe de fabrique concrète pour chaque nouvelle famille de produits

Retour à notre exemple (1)

```
// Method 1: use factory methods
public abstract class WidgetFactory {
  public abstract Window createWindow();
  public abstract ScrollBar createScrollBar();
  public abstract Button createButton();
}

public class MotifWidgetFactory extends WidgetFactory {
  public Window createWindow() {return new MotifWindow();}
  public ScrollBar createScrollBar() {return new MotifScrollBar();}
  public Button createButton() {return new MotifButton();}
}
...
// The client code is the same no matter how the factory creates
// the product
WidgetFactory wf = new MotifWidgetFactory();
Window w = wf.createWindow();
ScrollBar sb = wf.createScrollBar();
Button b = wf.createButton();
```

Retour à notre exemple (2)

```
// Method 2: use factories
public abstract class WidgetFactory {
  protected WindowFactory windowFactory;
  protected ScrollBarFactory scrollBarFactory;
  protected ButtonFactory buttonFactory;
  public Window createWindow() {return windowFactory.createWindow();}
  public ScrollBar createScrollBar() {
    return scrollBarFactory.createScrollBar();
  }
  public Button createButton() {return buttonFactory.createButton();}
}

public class MotifWidgetFactory extends WidgetFactory {
  public MotifWidgetFactory() {
    windowFactory = new MotifWindowFactory();
    scrollBarFactory = new MotifScrollBarFactory();
    buttonFactory = new MotifButtonFactory();
  }
}
```

Retour à notre exemple (3)

```
// Method 3: use factories with no required subclasses
// (pure composition)
public class WidgetFactory {
  private WindowFactory windowFactory;
  private ScrollBarFactory scrollBarFactory;
  private ButtonFactory buttonFactory;
  public WidgetFactory(WindowFactory wf, ScrollBarFactory sbf,
                       ButtonFactory bf) {
    windowFactory = wf;
    scrollBarFactory = sbf;
    buttonFactory = bf;
  }
  ... // setters
  public Window createWindow() {return windowFactory.createWindow();}
  public ScrollBar createScrollBar() {
    return scrollBarFactory.createScrollBar();
  }
  public Button createButton() {return buttonFactory.createButton();}
}
```

Méthode de fabrique vs Fabrique abstraite

- Méthode de fabrique
  - Pattern de création au niveau classe
  - Utilisation de l'héritage (sous-classe) pour décider de l'objet à instancier
- Fabrique abstraite
  - Pattern de création au niveau objet
  - Utilisation de la composition pour déléguer la responsabilité de l'instanciation d'objets à un autre objet
    - ★ L'objet délégué utilise souvent des méthodes de fabrique pour effectuer l'instanciation

Problème : unicité d'une fabrique

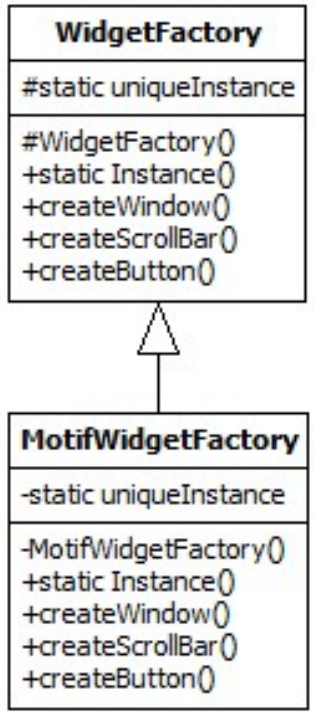
- Nous avons seulement besoin d'une seule instance de fabrique concrète pour créer la famille de *widgets* correspondant au *look-and-feel* choisi
  - Maintient de la cohérence graphique
- Autrement dit, comment garantir qu'il ne peut exister qu'une seule instance d'une classe, facilement accessible ?
- Que proposez-vous comme conception ?

Pas très bon

- Les variables globales permettent d'accéder à un objet n'importe où dans le programme mais n'empêche pas des instanciations multiples de cet objet
- Une constante dans une interface ?
  - C'est très souvent un détail d'implémentation (vs service)
  - Elle se retrouve dans l'interface publique de la classe implémentant l'interface
  - Anti-pattern
- Qualificatif `static`
  - Solution un peu rigide

Diagramme de classes

- La solution passe par la création d'une instance, plus flexible et plus évolutive grâce à l'héritage
  - La classe elle-même a la responsabilité d'assurer l'unicité de son instance
    - ★ En interceptant les requêtes demandant à créer de nouveaux objets
  - Elle peut également fournir un moyen d'accéder à cette instance



Singleton (*Singleton*)

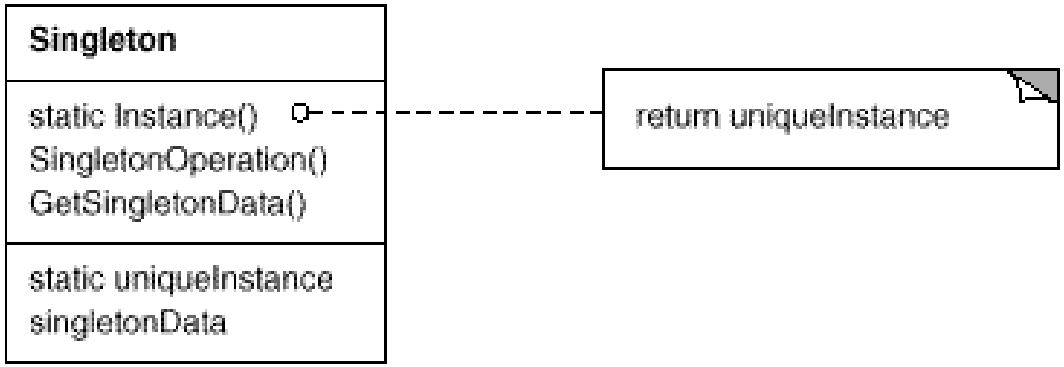
- Intention
  - Garantir qu'une classe n'a qu'une seule instance et fournir un point d'accès global à cette instance
- Problème
  - Divers clients doivent se référer à une même chose et on veut être sûr qu'il n'existe qu'une seule instance
  - Si on ne garantit pas l'unicité, il peut y avoir de graves problèmes de fonctionnement (incohérences, écrasement de données)
- Solution
  - S'assurer qu'il n'existe qu'une seule instance en contrôlant le constructeur
- Métaphore
  - Dans le monde, il n'existe à chaque instant qu'un seul champion du monde de football

Singleton (*Singleton*)

- Indications d'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

Singleton (*Singleton*)

- Structure



- Participants
  - Une seule classe concrète (aucune classe abstraite/interface)
  - `Singleton` construit sa propre instance unique et définit une méthode de classe `Instance()` qui donne l'accès à son unique instance
    - ★ `Instance()` peut être chargée de créer cette instance unique
    - ★ `uniqueInstance` est une variable de classe de type `Singleton`
    - ★ `singletonData` est une variable d'instance (état)
    - ★ `SingletonOperation()` est une méthode d'instance (comportement)
- Collaboration
  - Les clients accèdent à l'instance d'un Singleton uniquement *via* la méthode de classe `Instance()`

Singleton (*Singleton*)

- 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 invisibles aux clients)
  - Est plus flexible que les méthodes de classe
- Inconvénient
  - La construction d'un singleton n'est pas naturelle (pas d'appel à `new Classe()`)

Singleton (*Singleton*)

- Implémentation
  - Le constructeur est privé/protégé
    - ★ Cela évite les créations intempestives
  - L'instance unique de la classe est stockée dans une variable statique privée/protégée
  - Une méthode publique statique de classe (`Instance()`)
    - ★ Elle crée l'instance au premier appel (*lazy instantiation*)
    - ★ Elle retourne cette instance
  - 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()`
  - La concurrence peut compliquer les choses...
    - ★ *Thread safe* (`synchronised`/*eager instantiation*)

Implémentation

```
public class Singleton {

  private static Singleton uniqueInstance = null;

  public static Singleton Instance() {
    if (uniqueInstance == null) {
      uniqueInstance = new Singleton(); // Lazy instantiation
    }
    return uniqueInstance;
  }

  // No client can instantiate a Singleton object
  private Singleton() {...}
  ...
}
...
Singleton s = Singleton.Instance();
```

Implémentation *thread safe*

```
// The singleton instance is created in a static initializer
// This is guaranteed to be thread safe
// Alternative: make the Instance method synchronized (expensive!)
public class Singleton {

  // Eager instantiation
  private static Singleton uniqueInstance = new Singleton();

  public static Singleton Instance() {
    return uniqueInstance;
  }

  // No client can instantiate a Singleton object
  private Singleton() {...}
  ...
}
...
Singleton s = Singleton.Instance();
```

Retour à notre exemple (1)

```
//Method 1: the Instance method determine the subclass to instantiate
public abstract class WidgetFactory {
  private static WidgetFactory uniqueInstance = null;
  public static WidgetFactory Instance() {
    if (uniqueInstance == null)
      return Instance("motif"); // As default
    return uniqueInstance;
  }
  public static WidgetFactory Instance(String str) {
    if (uniqueInstance == null) {
      if (str.equals("motif")) {
        uniqueInstance = new MotifWidgetFactory();
      } else if (...) {...}
    }
    return uniqueInstance;
  }
  // Constructors cannot be private here
  protected WidgetFactory() {...}
}
// Client code to create factory the first time
WidgetFactory factory = WidgetFactory.Instance("motif");
// Client code to access the factory
WidgetFactory factory = WidgetFactory.Instance();
```

Retour à notre exemple (1)

- Les constructeurs des sous-classes (*e.g.*, `MotifWidgetFactory`) ne peuvent pas être privés, car `WidgetFactory` doit pouvoir les instancier
- Donc, des clients pourraient potentiellement créer d'autres instances de ces sous-classes
- La méthode `Instance(String)` viole le principe *Ouvert-Fermé*, car elle doit être modifiée pour chaque nouvelle sous-classe de `WidgetFactory`
- Nous pourrions utiliser le nom des classes comme argument de la méthode, donnant ainsi un code plus simple :

```
public static WidgetFactory Instance(String str) {
  if (uniqueInstance == null) {
    uniqueInstance = Class.forName(str).newInstance();
  }
  return uniqueInstance;
}
```

Retour à notre exemple (2)

```
// Method 2: each subclass provide a static Instance method
public abstract class WidgetFactory {
  protected static WidgetFactory uniqueInstance = null;
  public static WidgetFactory Instance() {
    return uniqueInstance;
  }
  // Constructor cannot be private here
  protected WidgetFactory() {...}
}
public class MotifWidgetFactory extends WidgetFactory {
  public static WidgetFactory Instance() {
        if (uniqueInstance == null)
            uniqueInstance = new MotifWidgetFactory();
    return uniqueInstance;
  }
  // Private subclass constructor!
  private MotifWidgetFactory() {...}
}
// Client code to create factory the first time
WidgetFactory factory = MotifWidgetFactory.Instance();
// Client code to access the factory
WidgetFactory factory = WidgetFactory.Instance();
```

Retour à notre exemple (2)

- Chaque sous-classe est une implémentation d'une classe Singleton
- Les constructeurs des sous-classes sont maintenant privés
  - Seulement une instance peut être créée
- Un client peut obtenir une référence `null` s'il invoque `WidgetFactory.Instance()` avant que l'unique instance de la sous-classe soit d'abord créée
- `uniqueInstance` est maintenant protégée

Au menu

Patterns de structure

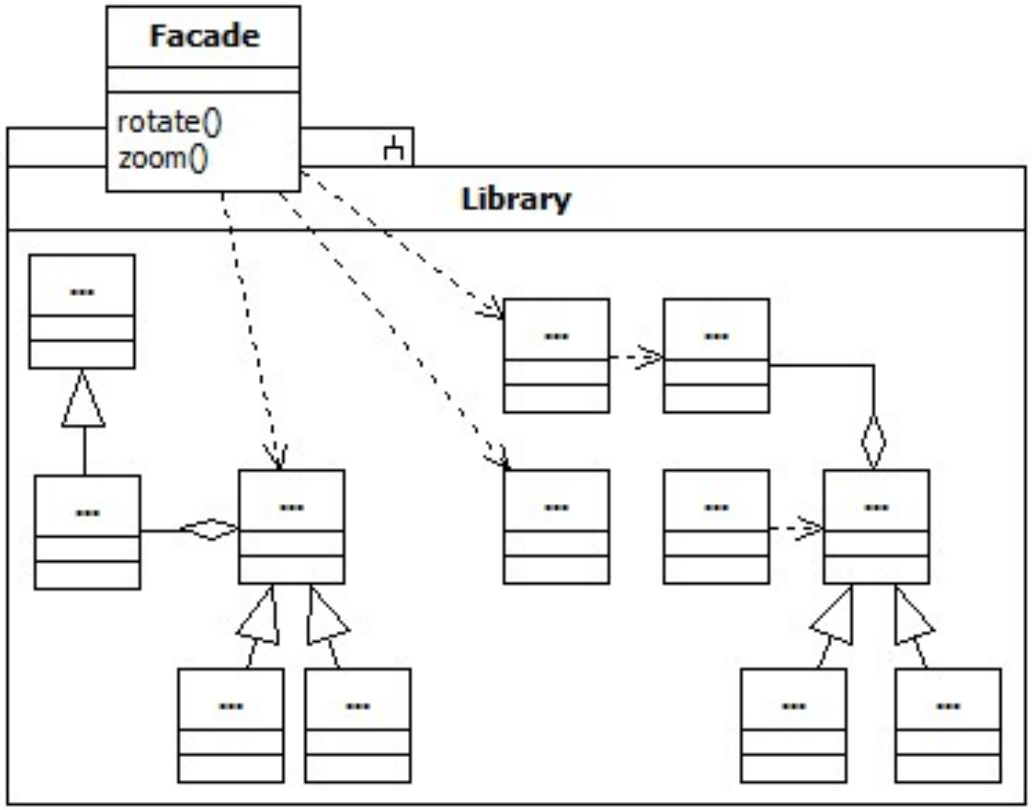
- Ils s'intéressent à la manière dont les classes et les objets sont composés pour former des structures plus importantes
- Ils permettent de modifier certains aspects de la structure d'un système indépendamment d'autres aspects
  - Flexibilité et extensibilité
- Ils réduisent le couplage entre des classes (découplage de l'interface et et de l'implémentation de classes et d'objets)
- Ils aident à rendre un système plus robuste à un type particulier de changement structurel

Problème : complexité d'une bibliothèque

- Nous disposons d'une bibliothèque de code pour manipuler des images
  - Beaucoup de classes
  - Interfaces avec des fonctionnalités puissantes mais de bas niveau
  - Plus que ce dont nous avons besoin
- Nous voulons aussi pouvoir changer de bibliothèque plus tard
- Autrement dit, comment réduire la complexité d'utilisation d'un sous-système (pour la plupart des clients), ainsi que les relations de dépendance entre les classes du sous-système et les clients ?
- Que proposez-vous comme conception ?

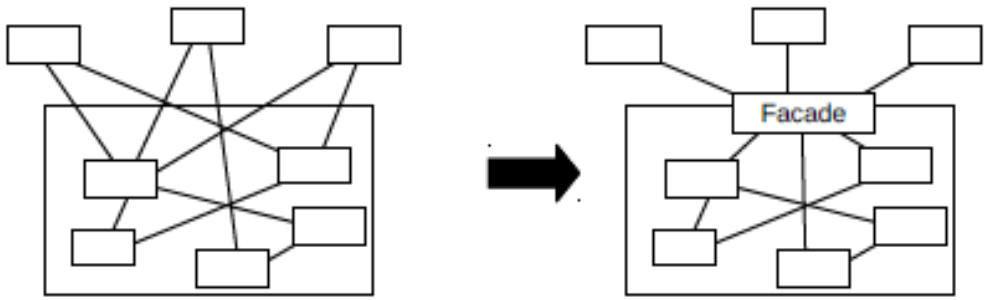
Diagramme de classes

- La solution consiste à introduire une interface de plus haut niveau qui offre aux clients une vue simple du sous-système



Façade (Facade)

- Intention
  - Fournir une interface unifiée et de plus 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
- Problème
  - On veut disposer d'une interface simple pour masquer un sous-système complexe
  - Il n'est pas nécessaire d'utiliser toutes les fonctionnalités du système d'origine
  - Découpler le sous-système du client des autres sous-systèmes
- Solution

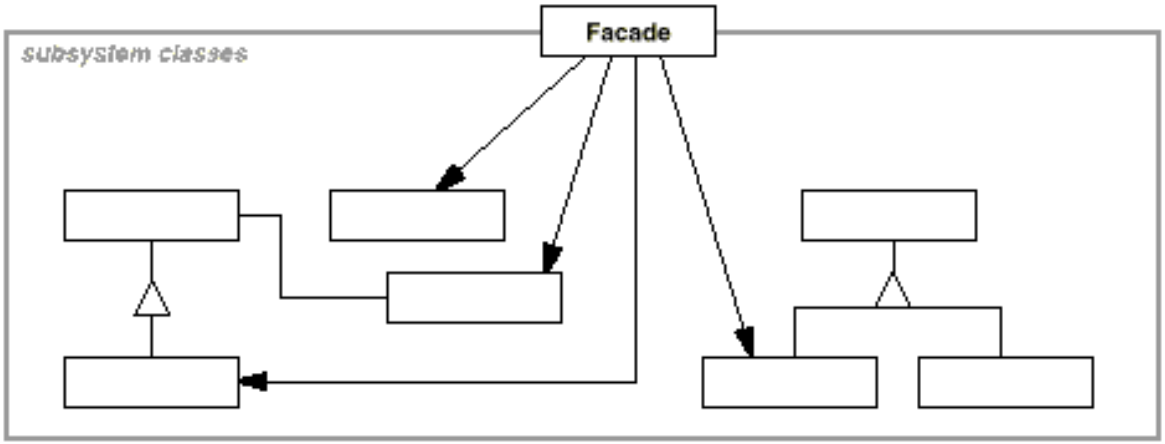


Façade (Facade)

- Indications d'utilisation
  - L'interface exposée par les classes d'un sous-système est complexe
    - ★ Fournir une interface simple et unique à la plupart des clients
    - ★ Les autres peuvent regarder derrière la façade
  - Il y a beaucoup de dépendances entre les classes d'un sous-système et ses clients
    - ★ Découpler l'implémentation d'un sous-système de ses clients
    - ★ Promouvoir l'indépendance et la portabilité d'un sous-système
  - Les sous-systèmes sont organisés en couches
    - ★ Définir une façade par point d'entrée dans chaque couche
- Exemple dans l'API Java
  - La création des fenêtres d'erreur nécessite de faire appel à plusieurs classes : `Jlabel`, `JPanel`, `JButton`...
  - La classe `JOptionPane` permet alors de simplifier la création de la fenêtre d'alerte en une seule fonction statique `JoptionPane.showMessageDialog()`

Façade (Facade)

- Structure



- Participants
  - `Facade` regroupe toutes les méthodes permettant de manipuler simplement le sous-système, sait quelles classes du sous-système sont responsables de telle ou telle requête et délègue le taitement des requêtes du client aux objets appropriés du sous-système
  - `Subsystem classes` implémentent les fonctionnalités du sous-système, traitent les requêtes émises par la façade, mais ne connaissent pas cette dernière (*i.e.*, pas de référence)
- N.B. : la façade n'ajoute pas de fonctionnalité, elle simplifie juste des interfaces

Façade (Facade)

- 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

Façade (Facade)

- 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
    - ★ Un client peut utiliser la façade ou le sous-système directement
- Inconvénients
  - Possible perte de fonctionnalités des classes interfacées selon la manière dont la façade est réalisée (trop restrictive ?)
  - N'empêche pas les clients d'accéder aux classes sous-jacentes !

Façade (Facade)

- Implémentation
  - Le couplage entre un sous-système et ses clients peut être réduit encore davantage
    - La classe `Facade` devient abstraite avec des sous-classes concrètes pour les différentes implémentations du sous-système
  - En général, une seule instance d'une façade est nécessaire à l'exécution
    - Utilisation du pattern Singleton

Retour à notre exemple

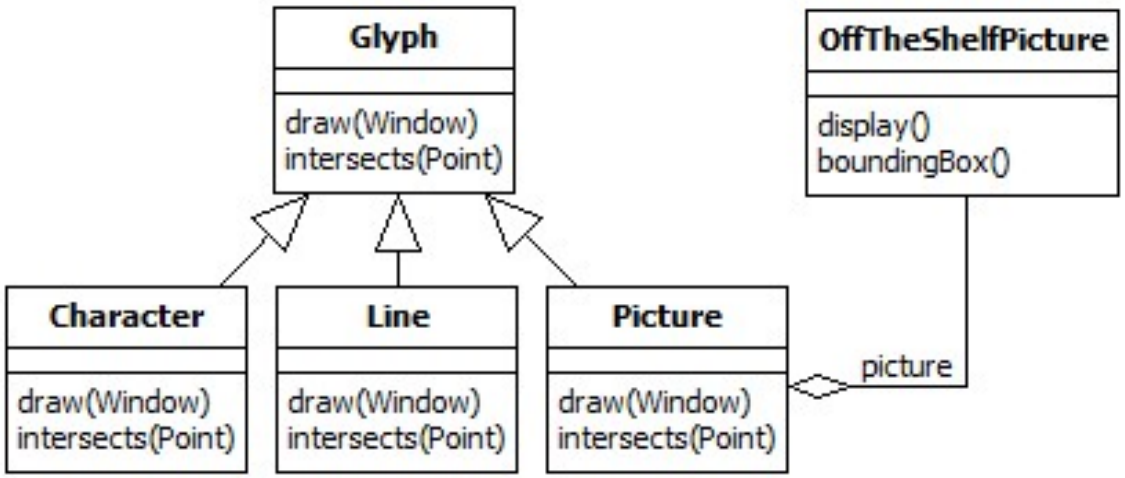
```
public class Facade {
  public Facade() {
    ... // initialize the library
  }
  public void rotate(Picture p) {
    ... // instanciate and use some classes of the subsystem
  }
  public void zoom(Picture p) {
    ... // instanciate and use some classes of the subsystem
  }
  ...
}
...
Facade f = new Facade(...);
f.rotate(picture);
f.zoom(picture);
```

Problème : incompatibilité d'interfaces

- Nous disposons d'une bibliothèque qui fournit un ensemble riche de fonctionnalités pour manipuler des images
  - Mais nous ne pouvons pas l'utiliser, car son interface est incompatible avec l'interface requise par l'éditeur de texte (*i.e.*, la classe `Glyph`)
  - Nous ne pouvons pas changer l'interface de la bibliothèque, car nous n'avons pas les sources (et quand bien même !)
- Autrement dit, comment des classes existantes peuvent-elles être réutilisées et fonctionner dans une application qui attend des classes avec une interface différente et incompatible ?
- Que proposez-vous comme conception ?

Diagramme de classes

- 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



Adaptateur (*Adapter*)

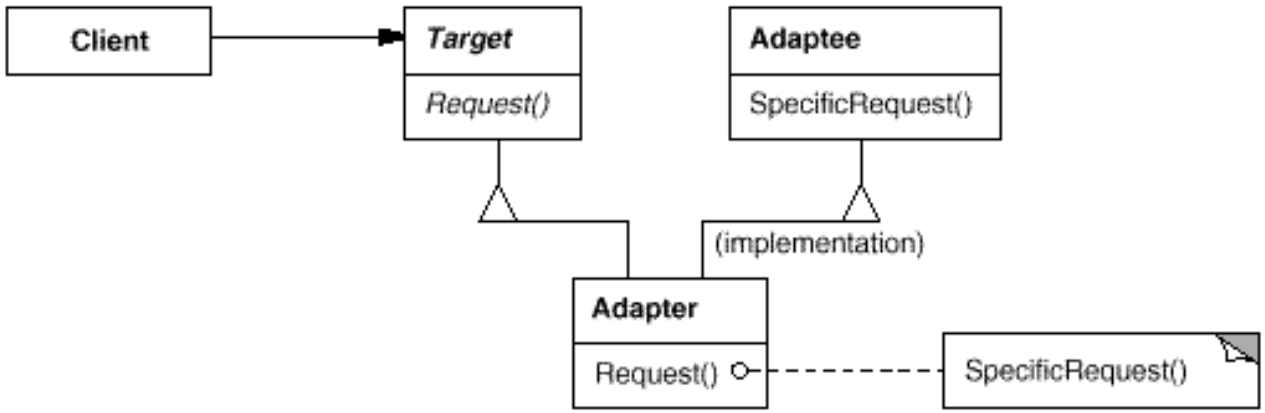
- Intention
  - Faire correspondre à une interface donnée un objet existant qu'on ne contrôle pas
- Problème
  - Un système (une classe ou un ensemble de classes) a les bonnes données et les bons services, mais la mauvaise interface et il n'est pas possible de la modifier
- Solution
  - Créer une nouvelle classe `Adapter` qui fournit l'interface souhaitée et fait le lien entre le système et un programme l'utilisant
- Métaphore
  - Parce que la France et l'Angleterre n'utilisent pas le même standard de prise électrique, il est nécessaire d'utiliser un adaptateur pour brancher un équipement au standard anglais sur une prise de courant au standard français

Adaptateur (*Adapter*)

- Indications d'utilisation
  - On veut utiliser une classe existante dont l'interface ne correspond pas à nos besoins
  - On souhaite créer une classe réutilisable qui collabore avec des classes encore inconnues qui seront développées par une autre équipe, c'est-à-dire avec des classes qui n'auront pas forcément des interfaces compatibles
  - On a besoin d'utiliser plusieurs sous-classes existantes dont l'adaptation de leur interface par héritage est impossible car on ne dispose pas du code pour le modifier

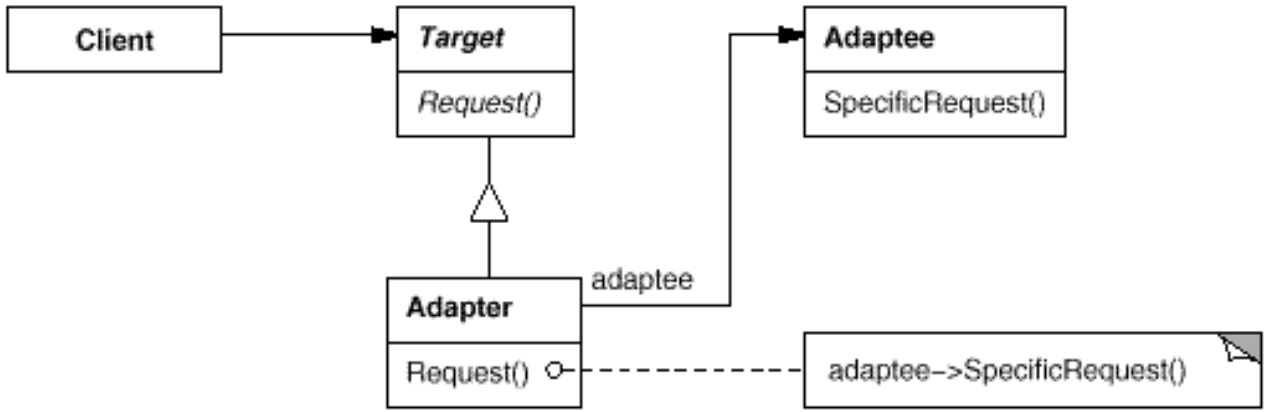
Adaptateur (*Adapter*)

- Structure de l'adaptateur de classe (par héritage multiple)



Adaptateur (*Adapter*)

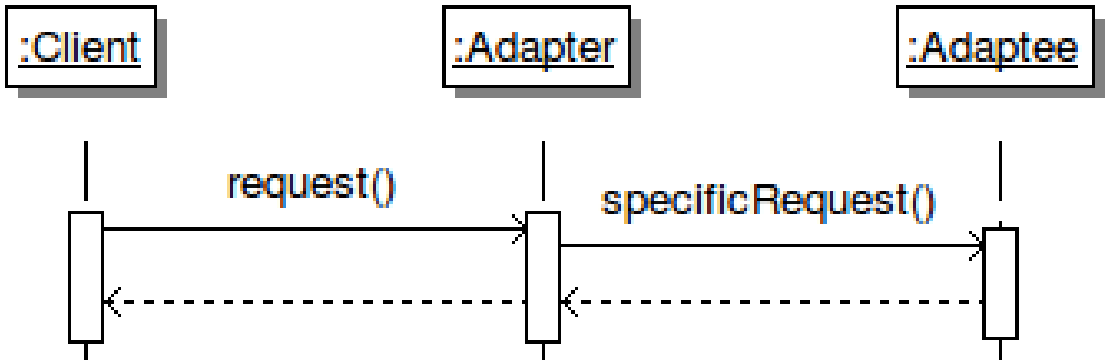
- Structure de l'adaptateur d'objet (par composition d'objets)



- Participants
  - `Target` définit l'interface spécifique à l'application pour un ensemble de services que le client utilise
  - `Client` collabore avec les objets conformes à l'interface `Target` en utilisant le service `Request()`
  - `Adaptee` possède un service recherché mais avec la mauvaise interface
  - `Adapter` adapte l'interface de `Adaptee` à l'interface souhaitée définie par `Target` et est responsable des fonctionnalités que la classe adaptée ne fournit pas

Adaptateur (*Adapter*)

- Collaboration
  - La classe `Adapter` adapte l'interface de la classe `Adaptee` pour qu'elle corresponde à celle de `Target`
  - Cela permet au client d'utiliser la classe `Adaptee` comme si elle était de type de `Target`



1. Les clients appellent les opérations d'une instance de `Adapter`
2. L'adaptateur appellent alors des opérations de `Adaptee` pour réaliser le service

Adaptateur (*Adapter*)

- 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 certains comportements de la classe adaptée (`Adapter` est une sous-classe de `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)

Adaptateur (*Adapter*)

- Implémentation
  - Jusqu'à quel point un adaptateur adapte-t-il ?
    - ★ Conversion simple d'interface : juste changer le nom des opérations et l'ordre des arguments
    - ★ Ensemble totalement différent d'opérations
  - L'adaptateur fournit-il une transparence bidirectionnelle ?
    - ★ Un adaptateur n'est pas transparent pour tous les clients
    - ★ Une fois adapté, un objet n'est plus conforme à l'interface de `Adaptee` (*i.e.*, qu'il ne peut pas être utilisé partout où un objet `Adaptee` peut l'être)
    - ★ Un adaptateur bidirectionnel supporte à la fois les interfaces `Target` et `Adaptee`, permettant à un objet `Adapter` d'apparaître comme un objet `Adaptee` ou comme un objet `Target`

Retour à notre exemple

```
public abstract class Glyph {
  public abstract void draw(Window w);
  public abstract boolean intersects(Point p);
  ...
}
public class OffTheShelfPicture {
  public void display(...) {...}
  public Rectangle boundingBox() {...}
  ...
}
public class Picture extends Glyph {
  private OffTheShelfPicture picture;
  public Picture(OffTheShelfPicture p) {picture = p;}
  public boolean intersects(Point p) {
    Rectangle r = picture.boundingBox();
    ...
  }
  ...
}
...
Glyph picture = new Picture(new OffTheShelfPicture(...));
picture.draw(window);
```

Adaptateur vs Façade

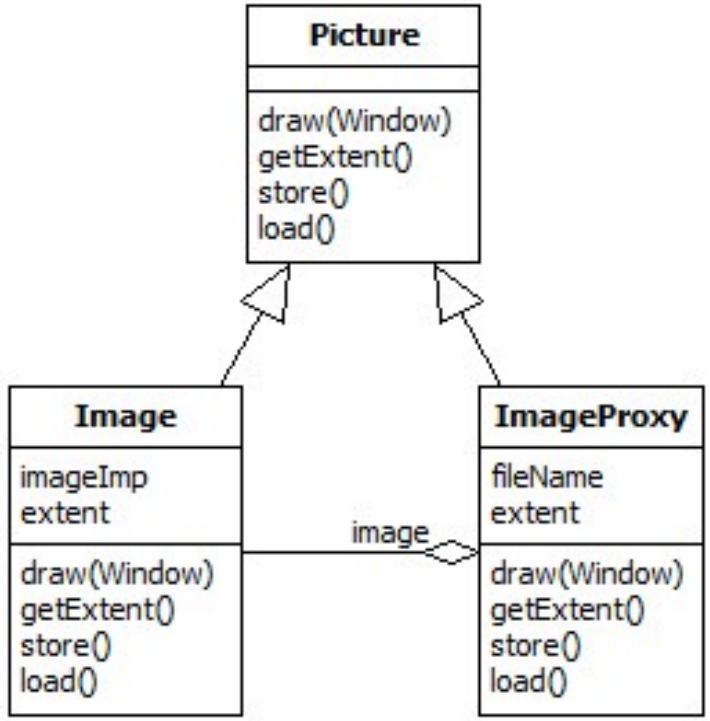
- Une adaptateur réutilise une *ancienne* interface alors qu'une façade définit une *nouvelle* interface
- Un adaptateur fait travailler ensemble deux interfaces *existantes*, au lieu d'en définir une entièrement nouvelle

Problème : chargement des images

- Les images insérées dans un document peuvent être coûteuses à charger, et donc ralentir l'ouverture du document
- Nous voulons éviter de charger toutes les images en même temps à l'ouverture du document, puisqu'elles ne seront pas toutes visibles dans le document au même moment
  - Que mettre dans le document à la place de l'image ?
  - Comment cacher le fait que l'image soit créée à la demande pour ne pas compliquer l'implémentation de l'éditeur ?
  - Cette optimisation ne devrait pas avoir d'effet sur le rendu ni le formatage du document
- Autrement dit, comment contrôler les accès à un objet mais sans le modifier directement, car il est utilisé ou réutilisé ailleurs ?
- Que proposez-vous comme conception ?

Diagramme de classes

- La solution passe par l'utilisation d'un objet intermédiaire qui va contrôler les accès à l'objet



Procuration (*Proxy*, *Surrogate*)

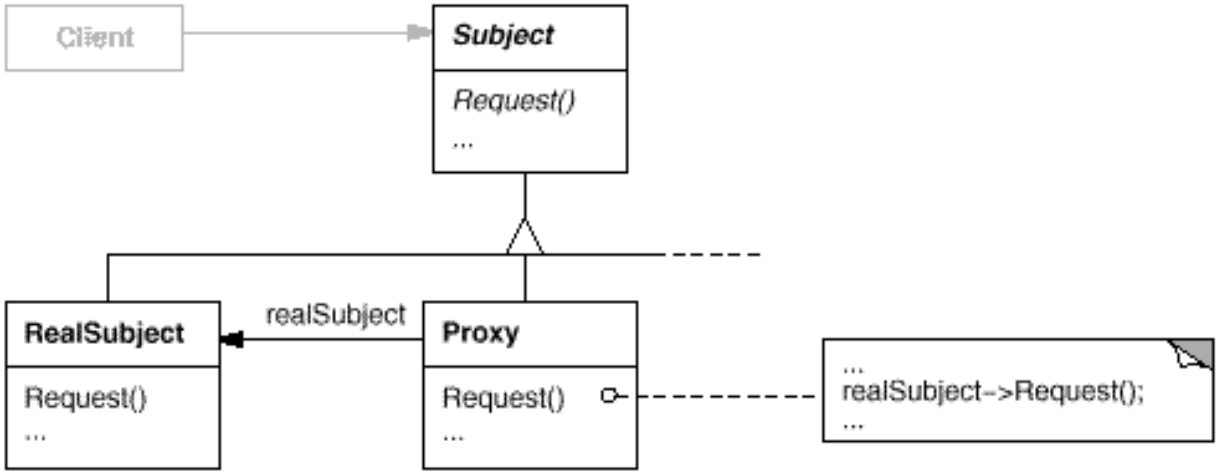
- Intention
  - Fournir un substitut (le proxy) à un autre objet afin d'en contrôler les accès (*i.e.*, les opérations qui lui sont appliquées)
- Problème
  - Ce qui justifie de contrôler l'accès à un objet, c'est le souci de différer sa création ou son initialisation avant son utilisation effective
- Solution
  - Créer un objet subrogé qui instancie l'objet réel la première fois que le client en fait la requête, mémorise l'identité de l'objet réel et redirige les requêtes vers cet objet
- Métaphore
  - Un chèque est un subrogé pour une somme d'argent d'un compte bancaire
  - Il peut être utilisé à la place du numéraire pour faire des achats, et il peut à tout moment donner accès à du numéraire

Procuration (*Proxy*, *Surrogate*)

- Indications d'utilisation
  - Un client ne fait pas ou ne peut pas faire référence à un objet directement, mais veut tout de même interagir avec l'objet
  - Il y a besoin d'un référencement plus sophistiqué qu'un simple pointeur
    - ★ Fournir un représentant local d'un objet distant (*remote proxy*)
    - ★ Créer ou charger des objets coûteux à la demande (*virtual proxy*)
    - ★ Différer la copie d'un objet coûteux jusqu'à ce que ce dernier soit modifié (*copy-on-write proxy*)
    - ★ Contrôler l'accès à un objet (*protection/access proxy*)
    - ★ Effectuer des opérations supplémentaires lors de l'accès à un objet (*smart reference*), telles que compter le nombre de références à un objet réel (*smart pointer*) ou encore charger en mémoire un objet persistant quand il est référencé pour la première fois (*lazy instantiation*)
- Exemple dans l'API Java
  - *Dynamic proxy* : `java.lang.reflect.Proxy`
  - *Remote Method Invocation* (*RMI*) : `java.rmi.*`

Procuration (*Proxy*, *Surrogate*)

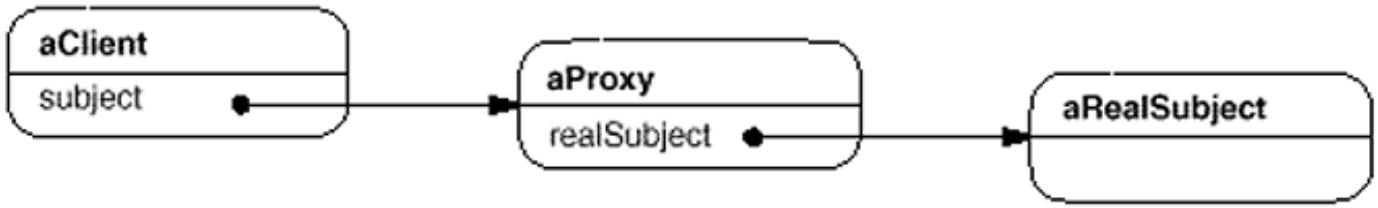
- Structure



- Participants
  - `Proxy` a la même interface que l'objet à contrôler et gère une référence au `RealSubject`, ce qui lui permet d'y accéder, d'en contrôler les accès et d'agir à sa place
  - `Subject` définit une interface commune pour `RealSubject` et `Proxy`, de sorte qu'un `Proxy` puisse être utilisé partout où un `RealSubject` est attendu
  - `RealSubject` définit l'objet réel que le proxy représente

Procuration (*Proxy*, *Surrogate*)

- Collaboration
  - Le proxy retransmet les requêtes au sujet réél si nécessaire (délégation), selon le type de proxy
  - Un proxy agit comme un intermédiaire entre le client et le sujet réel



Procuration (*Proxy*, *Surrogate*)

- 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

Procuration (*Proxy*, *Surrogate*)

- Implémentation
  - Un proxy doit-il forcément connaître le type de son sujet réel ?
    - ★ Dépend de si le proxy instancie ou non le sujet réel
    - ★ Si non, inutile de faire une classe `Proxy` pour chaque classe `RealSubject`
  - Comment faire référence au sujet réel avant son instanciation ?
    - ★ Certains proxies doivent faire référence à leur sujet, qu'il soit sur disque ou en mémoire
    - ★ Utilisation d'une certaine forme d'identificateurs d'objet indépendante de l'espace d'adressage (*e.g.*, un nom de fichier dans l'exemple)

Retour à notre exemple

```
public class ImageProxy extends Picture {
  private Image image;
  private String fileName;
  private Point extent; // width and height
  public ImageProxy(String fileName) {...}
  public void draw(Window) {
    if (image == null) {
      image = new Image(fileName); // load only on demand
    }
    image.draw(w);
  }
  public Point getExtent () {
    if (image == null) {
      return extent;
    } else {
      return image.getExtent();
    }
  }
  ...
}
...
Picture p = new ImageProxy("/home/jml/Pictures/foo.jpg");
editor.insert(p);
```

Problème : amélioration de l'interface utilisateur

- Nous souhaitons décorer les éléments de l'interface utilisateur (*User Interface*)
  - Ajouter des bordures,
  - Des barres de défilements,
  - *etc.*
- Nous avons un certain nombre de décorations qui peuvent être mélangées indépendamment :

```
x = new ScrollBar(new Border(new Picture(...)))
```

  - Pour n décorations, 2^n combinaisons
- Comment les intégrer dans la structure physique du document ?
- Que proposez-vous comme conception ?

Pas très bon (1)

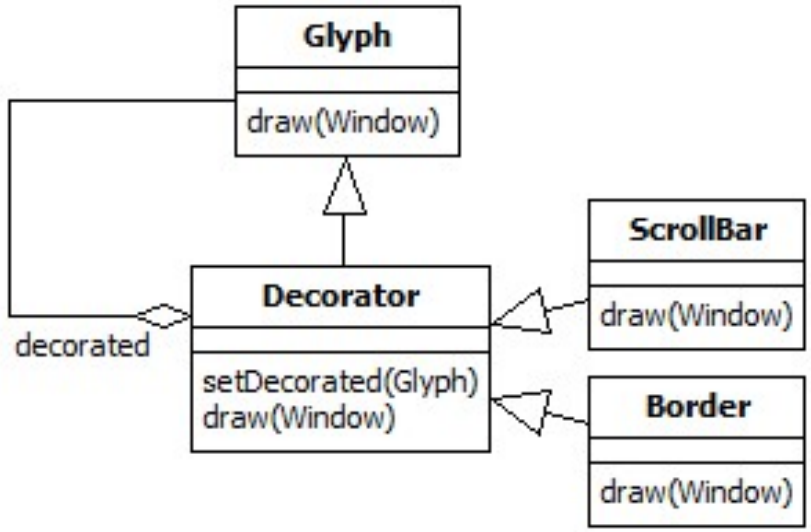
- Le comportement des objets peut être étendu en utilisant l'héritage
  - Inconvénient majeur : l'héritage est statique
  - Un client ne peut pas contrôler à la compilation quand et comment "décorer" un composant
- Sous-classes de `Glyph` :
  - `BorderedComposition`
  - `ScrolledComposition`
  - `BorderedAndScrolledComposition`
  - `ScrolledAndBorderedComposition`
  - *etc.*
- Explosion de classes !

Pas très bon (2)

- Une variable d'instance booléenne pour chaque décoration dans la classe `Glyph` (*i.e.*, la super-classe) pour représenter le fait que le glyphe ait ou non la décoration
- La méthode `draw` de la classe `Glyph` peut dessiner chaque décoration
- Les sous-classes de `Glyph` surcharge la méthode `draw` pour dessiner le glyphe, mais invoque aussi celle de la super-classe pour dessiner les décorations ajoutées
- Quid
  - D'une modification du dessin d'une décoration ?
  - De l'ajout de nouvelles décorations ?
  - De l'ajout de nouveaux glyphes pour lesquels certaines décorations ne sont pas appropriées ?
- Modification du code existant : violation du principe *Ouvert-Fermé* !

Diagramme de classes

- Définir une classe `Decorator` (enveloppe transparente)
  - Implémente `Glyph`
  - A une variable d'instance `decorated` de type `Glyph`
  - `Border`, `ScrollBar` étendent `Decorator`



Décorateur (*Decorator*)

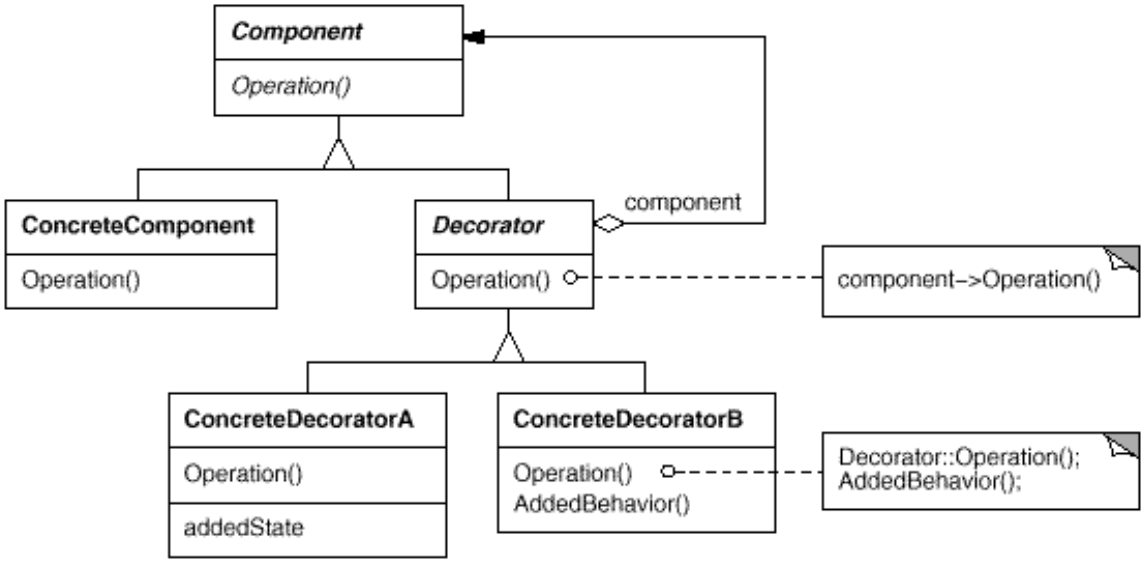
- Intention
  - Attacher dynamiquement des responsabilités supplémentaires à un objet
  - Fournir une alternative flexible à l'héritage pour étendre des fonctionnalités
- Problème
  - Ce qui varie : le comportement d'un service d'un objet en fonction de ses propriétés
  - L'objet que vous souhaitez utiliser possède les fonctions de base dont vous avez besoin, mais vous devrez lui ajouter des fonctionnalités supplémentaires qui s'exécuteront avant ou après la fonctionnalité de base
- Solution
  - Permettre l'extension de la fonctionnalité d'un objet sans avoir recours à des sous-classes, mais en utilisant une chaîne de fonctionnalités à partir de l'objet de base

Décorateur (*Decorator*)

- Indications d'utilisation
  - Un système doit ajouter des responsabilités à des objets individuels de manière dynamique et transparente (*i.e.*, sans affecter d'autres objets)
  - Ces responsabilités peuvent être retirées
  - L'extension par héritage n'est pas pratique (explosion du nombre de sous-classes pour supporter toutes les combinaisons possibles) ou n'est pas possible
- Exemple dans l'API Java
  - La gestion des entrées-sorties en Java : `InputStream`, `FilterInputStream`, `OutputStream`, `FilterOutputStream`

Décorateur (*Decorator*)

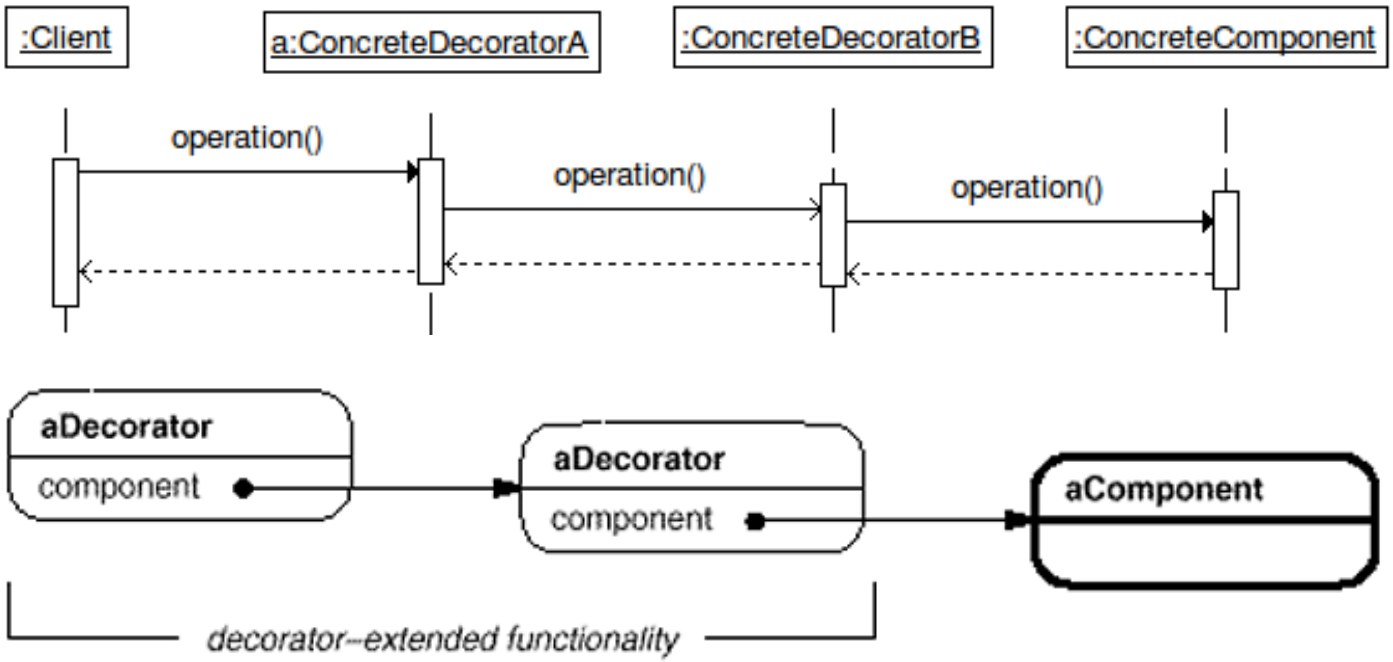
- Structure



- 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

Décorateur (*Decorator*)

- Collaboration
  - `Decorator` transmet les requêtes à son objet décoré
  - Il peut éventuellement effectuer des opérations supplémentaires avant et/ou après



Décorateur (*Decorator*)

- Avantages
  - Offre plus de flexibilité que l'héritage (statique)
  - Évite de surcharger les classes situées en haut de la hiérarchie avec des fonctionnalités (classes monolithiques)
- Inconvénients
  - Casse l'identité de l'objet
    - ★ Un décorateur et son composant ne sont pas identiques
    - ★ Un décorateur se comporte comme une enveloppe transparente
  - Conduit à des systèmes composés de beaucoup de petits objets, tous d'apparence voisine, interconnectés qu'il est souvent difficile à appréhender et à déboguer
  - L'interface d'un objet `Décorateur` doit être conforme à l'interface du composant décoré
    - ★ On ne peut pas ajouter de méthode supplémentaire aux décorateurs
  - Il est facile d'ajouter dynamiquement de nouvelles décorations à un composant, mais il est impossible d'en supprimer puisqu'elles sont enfouies dans les couches

Décorateur (*Decorator*)

- Implémentation
  - Conformité d'interface
    - ★ Décorateurs et composants doivent descendre d'une classe commune
    - ★ En Java, utilisation d'interface
  - Est-il nécessaire de définir une classe abstraite `Decorator` ?
    - ★ Non s'il n'y a qu'une seule responsabilité à ajouter
  - Importance de garder la classe `Component` légère
    - ★ Définition d'une interface et non stockage des données
    - ★ Mais un grand nombre de fonctionnalités pénaliseraient les sous-classes concrètes avec des fonctionnalités dont elles n'ont pas besoin

Retour à notre exemple

```
public abstract Decorator extends Glyph {
  protected Glyph decorated;
  public Decorator(Glyph g) {this.decorated = g;}
  public abstract void draw(Window w);
}
public class Border extends Decorator {
  public void draw(Window w) {
    decorated.draw(w);
    ... //code to draw the border object itself
  }
}
public class ScrollBar extends Decorator {
  public void draw(Window w) {
    decorated.draw(w);
    ... //code to draw the scrollbar object itself
  }
}
...
Glyph picture = new Picture();
Glyph borderPicture = new Border(picture);
Glyph borderAndScrolledPicture = new Scrollbar(borderPicture);
borderAndScrolledPicture.draw(window);
```

Décorateur vs Proxy

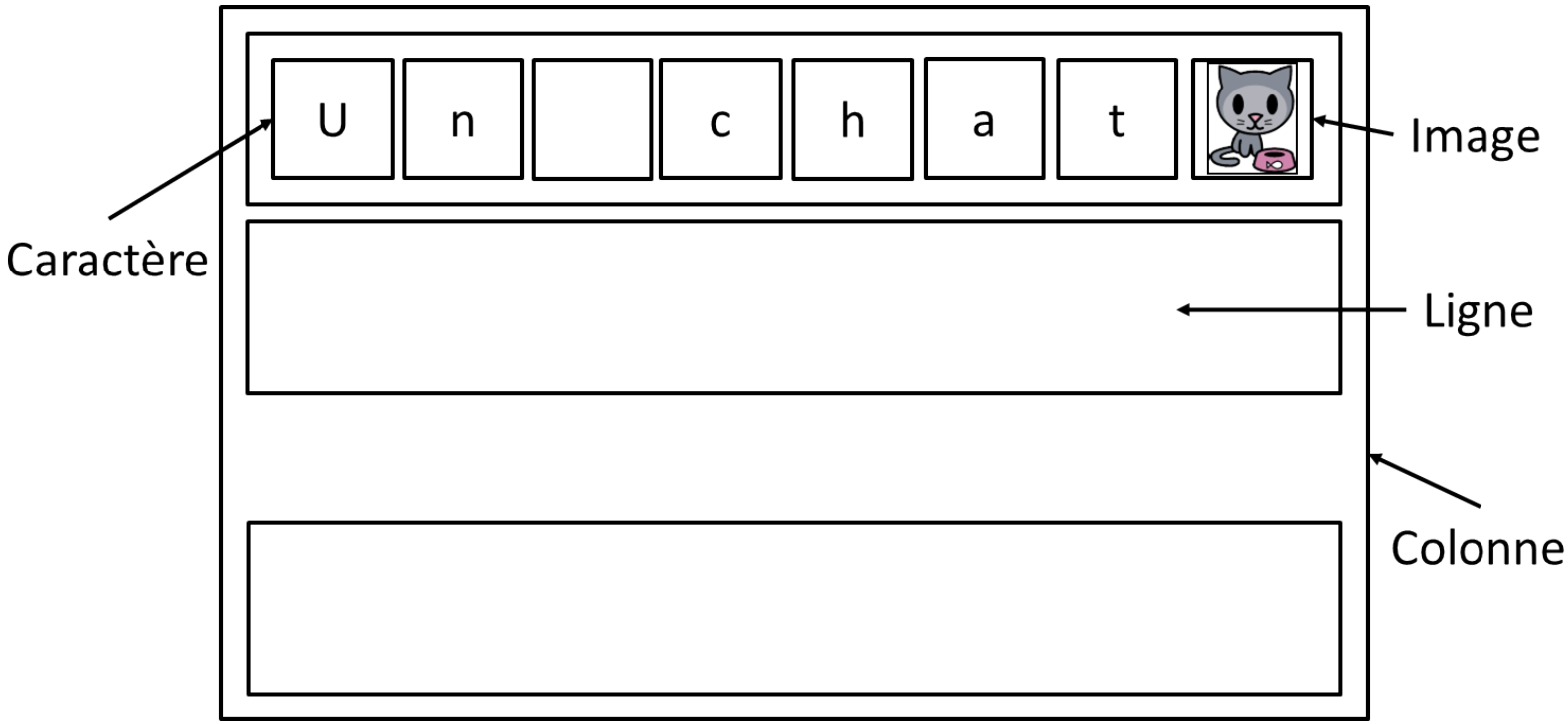
- Ils ont une structure similaire
  - Ils fournissent un niveau d'indirection à un autre objet
  - Ils gèrent une référence à cet objet auquel ils retransmettent les requêtes
- Mais ils ont des intentions bien distinctes !
  - Le décorateur ajoute dynamiquement des fonctionnalités à un objet (composition récursive)
  - 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

- Un document est représenté par sa structure physique :
  - *Glyphes* primitifs
    - ★ Caractères, rectangles, cercles, images...
  - Lignes
    - ★ Une séquence de glyphes
  - Colonnes
    - ★ Une séquence de lignes
  - Pages
    - ★ Une séquence de colonnes
  - Documents
    - ★ Une séquence de pages

Problème : structure de document

- Exemple de composition hiérarchique



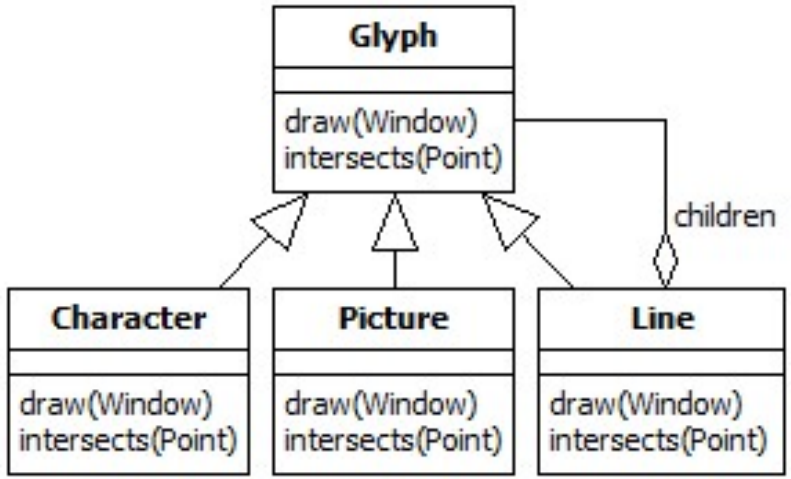
- Que proposez-vous comme conception ?

Pas très bon

- Des classes `Character`, `Circle`, `Line`, `Column`, `Page`, *etc.*
  - Beaucoup de duplication de code

Diagramme de classes

- La solution passe par une organisation subtile liant héritage et composition
- Une classe (abstraite) `Glyphe`
  - Chaque élément est réalisé par une sous-classe de `Glyphe`
  - Tous les éléments présentent la même interface
    - ★ Comment dessiner
    - ★ Calcul du rectangle englobant
    - ★ Détection des clics de souris
    - ★ *etc.*



Composite (*Composite*)

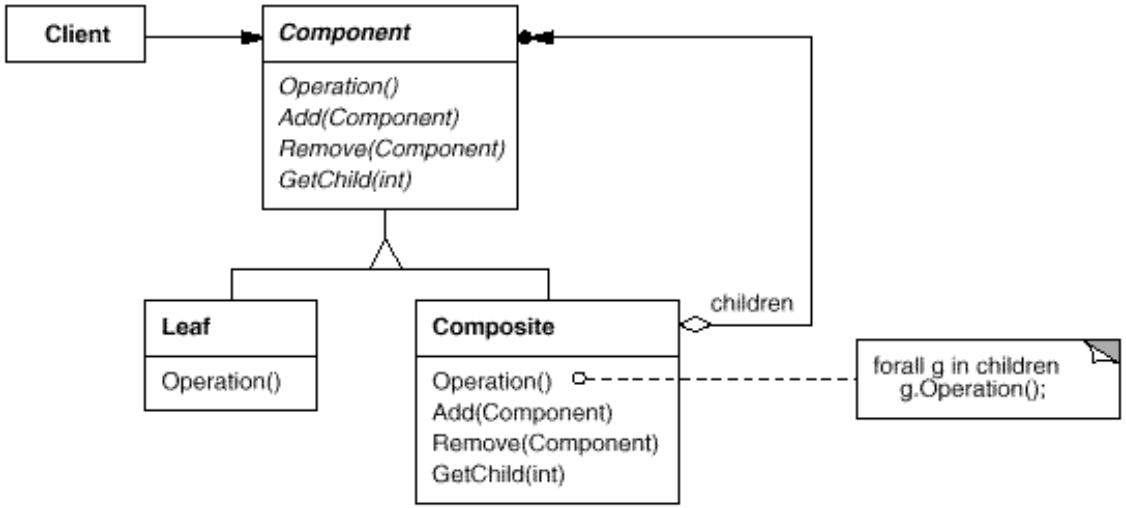
- Intention
  - Composer des objets dans des structures arborescentes pour représenter des hiérarchies composants/composés
  - Permettre aux clients de traiter de la même façon des objets individuels des groupements de ces objets
- Autres noms
  - Composition récursive, induction structurelle...
- Solution
  - Définir une classe abstraite qui spécifie le comportement commun à tous les composés et composants
  - La relation de composition lie un composé à tout objet du type de la classe abstraite

Composite (*Composite*)

- Indications d'utilisation
  - On souhaite représenter des hiérarchies de l'individu à l'ensemble,(composants/composés)
  - On souhaite que les clients n'aient pas à se soucier de la différence entre les objets individuels et leurs compositions (uniformité apparente)

Composite (*Composite*)

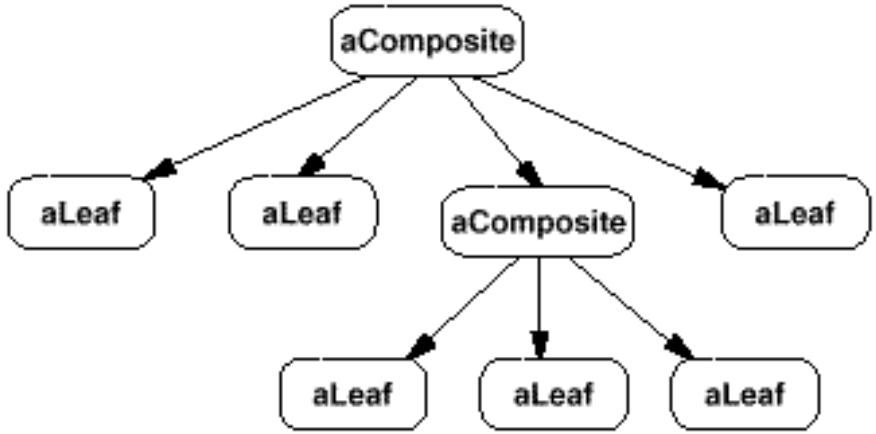
- Structure



- 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) dans la composition et définit le comportement des objets primitifs
  - `Composite` définit le comportement des objets composés, stocke les objets enfants et implémente les opérations nécessaires à leur gestion
  - `Client` manipule les objets de la composition à travers l'interface `Component`

Composite (*Composite*)

- Il permet de créer un arbre abstrait du type



- Collaboration
  - Un client utilise l'interface de la classe `Component` pour manipuler les objets de la structure composite
  - Si l'objet manipulé par le client est une feuille, la requête est traitée directement. Si c'est un composite, la requête est transférée à ses composants fils, en effectuant éventuellement des traitements supplémentaires avant et/ou après

Composite (*Composite*)

- Avantages
  - Facilite l'ajout de nouveaux types de composants
  - Simplifie l'accès à la structure pour le client qui n'a pas à savoir (et ne devrait pas se soucier de) s'il traite avec une feuille ou un composite (unification des requêtes)
  - Permet de définir des hiérarchies de classes qui peuvent être composées de façon récursive
- Inconvénients
  - Est difficile de restreindre et vérifier le type des composants d'un composite (*e.g.*, ne doit contenir que certains types de composant)

Composite (*Composite*)

- Implémentation
  - Un composite connaît ses composants enfants, pour autant ces derniers doivent-ils gérer une référence à leur composant parent ?
    - ★ Dépend de l'application (voir le pattern Chaîne de responsabilité)
  - Où déclarer les opérations de gestion des fils (*i.e.*, `add`, `remove`, `getChild`) ?
    - ★ Dans la classe `Component` pour la transparence : tous les composants peuvent être traités de la même manière, mais ce n'est pas sûr car les clients peuvent essayer de faire des choses qui n'ont pas de sens sur les feuilles à l'exécution
    - ★ Dans la classe `Composite` pour la sûreté : toute tentative pour effectuer une opération de gestion des fils sur une feuille est interdite à la compilation, mais la transparence est perdue car maintenant les feuilles et les composites ont des interfaces différentes

Composite (*Composite*)

- Implémentation (suite)
  - La classe `Component` doit-elle gérer la liste des composants utilisés par un composite ? Autrement dit, cette liste doit-elle être une variable d'instance de `Component` plutôt que de `Composite` ?
    - ★ Mieux vaut garder cette partie dans `Composite` et éviter de gaspiller l'espace de chaque feuille
  - L'ordonnancement des fils est-il important ?
    - ★ Dépend de l'application (voir le pattern Itérateur)
  - Qui supprime les composants ?
    - ★ En Java, ce n'est pas un problème : le ramasse-miettes est là !
  - Quelle est la meilleure structure de données pour stocker les composants ?
    - ★ Dépend de l'application
  - Pourquoi une relation d'agrégation plutôt que de composition ?
    - ★ Partage des composants

Retour à notre exemple

```
public class Line extends Glyph {
  private List<Glyph> glyphs;
  public Line() {...}
  public void add(Glyph g) {...}
  public void remove(Glyph g) {...}
  public Glyph getChild(int i) {...}
  ...
  public void draw(Window w) {
    for (Glyph g : glyphs) {
      g.draw(w);
    }
  }
}
...
Glyph c = new Character();
c.draw(window);
Glyph l = new Line();
l.add(c);
l.draw(window);
```

Retour à notre exemple

- Dessin
  - Chaque élément primitif se dessine lui-même à sa position assignée
  - Chaque élément composé appelle récursivement la méthode `draw` sur ses éléments, sans se soucier de ce que sont ces éléments
- Permet d'étendre la classe facilement avec d'autres éléments
- Traite tous les éléments de manière uniforme

Composite vs Décorateur

- Ils ont des diagrammes de structure similaires qui reposent sur la composition récursive pour organiser un nombre ouvert d'objets
- Mais ils ont des intentions bien distinctes !
  - Le composite se focalise la structure des classes afin que des objets individuels et leurs compositions puissent être traités uniformément (*i.e.*, sur la représentation)
  - Le décorateur se focalise sur l'ajout dynamique de responsabilités à des objets sans utiliser l'héritage (*i.e.*, sur la décoration)
- Généralement un décorateur étend un composite

Au menu

Patterns de comportement

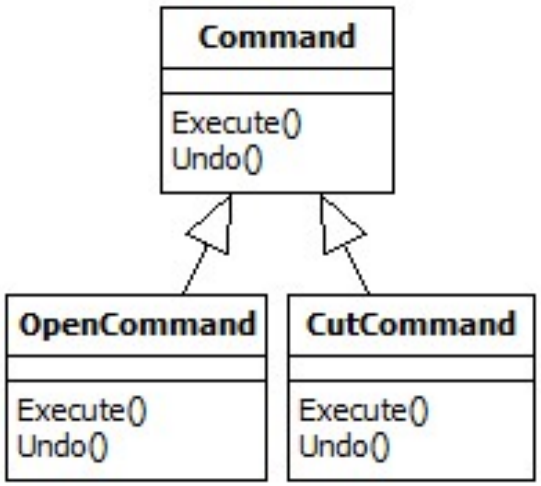
- Ils traitent de l'interaction des classes et des objets, et de la façon de se repartir les tâches
  - Description de comportements d'interaction entre objets
  - Gestion des interactions dynamiques entre des classes et des objets

Problème : les commandes utilisateur

- L'utilisateur a un vocabulaire d'opérations (*e.g.*, sauter à une page particulière, copier/coller un mot)
- Les opérations peuvent être invoquées de plusieurs manières :
  - À partir d'un menu
  - En cliquant sur une icône
  - Par un raccourci clavier
- Nous voulons également pouvoir défaire (*undo*) et refaire (*redo*) ces opérations
- Autrement dit, comment réaliser un traitement sans avoir besoin de savoir de quoi il s'agit et de qui va l'effectuer ?
- Que proposez-vous comme conception ?

Diagramme de classes

- Définir une classe abstraite d'opérations utilisateur
  - Interface commune à toutes les opérations
- Chaque opération est alors une sous-classe
  - Sauter à une page page, sauvegarder, couper, coller, *etc.*



Commande (*Command*, *Action*)

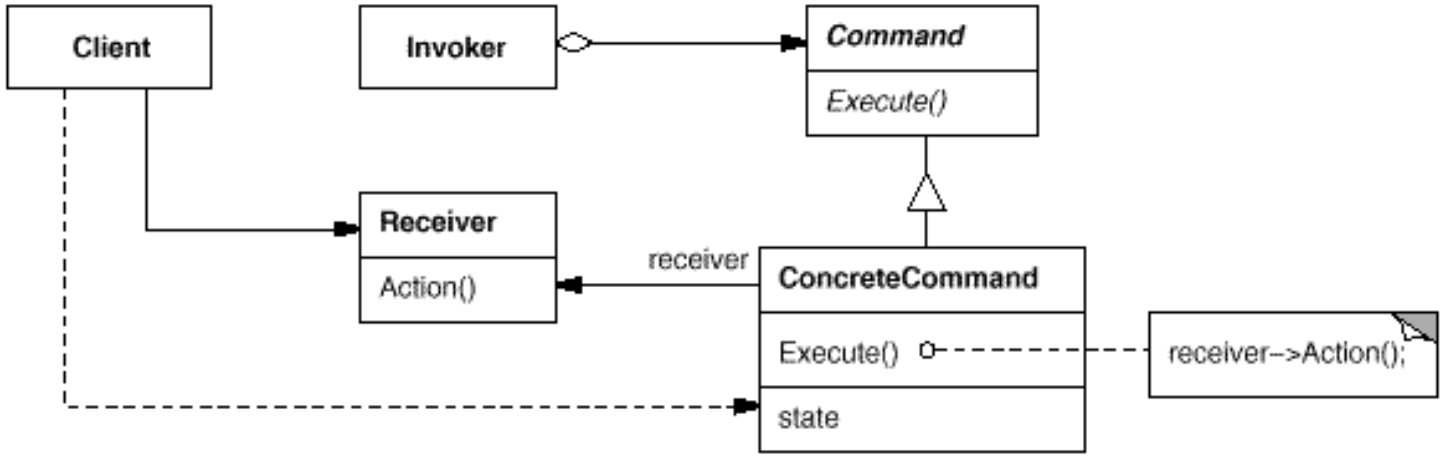
- Intention
  - Encapsuler une requête comme un objet, pour permettre de gérer une file d'attente ou un historique de requêtes et d'assurer le traitement des opérations réversibles
  - Promouvoir l'invocation d'une méthode d'un objet comme un objet à part entière
- Problème
  - Nécessité de traiter des requêtes sur des objets sans rien savoir sur l'opération invoquée ou sur le receveur de la requête
  - Besoin de stocker la suite des requêtes utilisateurs
- Solution
  - Encapsuler l'exécution d'un service dans un objet à part entière
- Métaphore
  - Dans un restaurant, le serveur prend une commande auprès d'un client et encapsule la commande en l'écrivant sur le carnet de commandes
  - La commande est ensuite envoyée en cuisine dans la file courante de commandes
  - Les commandes sont enregistrées pour être traitées ultérieurement par les cuisiniers

Commande (*Command*, *Action*)

- Indications d'utilisation
  - Des objets doivent être paramétrés par une action à effectuer
    - ★ Fonction de rappel (*callback*) : enregistrée à un moment donné pour être appelée ultérieurement
    - ★ Version objet des *callbacks*
  - On veut pouvoir spécifier, mettre en file d'attente et exécuter des requêtes à différents moments
  - On souhaite garder un historique des commandes effectuées afin de pouvoir les défaire/refaire (*undo*/*redo*) (après un éventuel crash)
    - ★ Enregistrement persistant des modifications
  - On veut structurer un système autour d'opérations de haut niveau construites à partir de primitives
    - ★ Systèmes d'information supportant des transactions
- Exemple dans l'API Java
  - Les classes `AbstractAction` et `Undoable`

Commande (*Command*, *Action*)

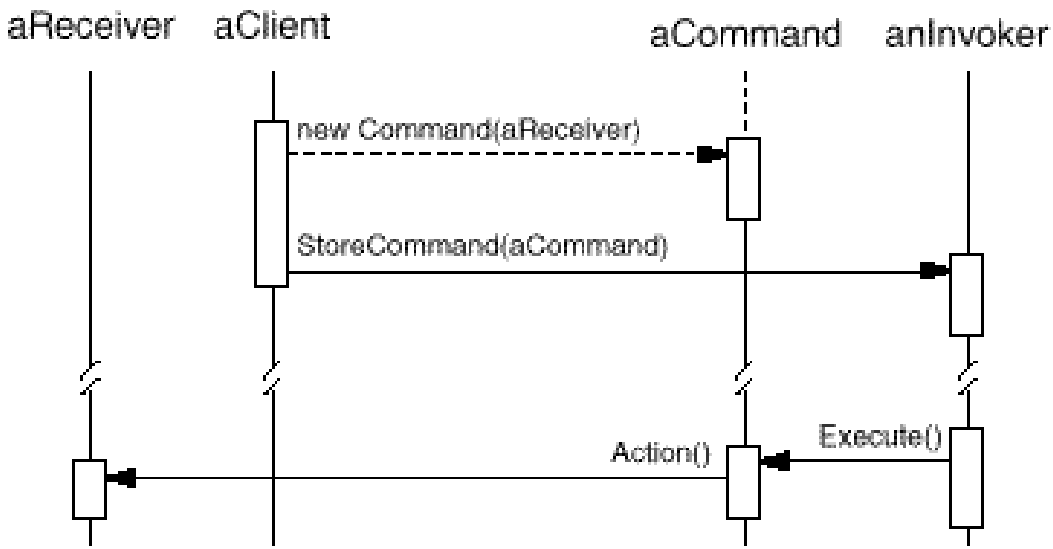
- Structure



- Participants
  - `Command` déclare une interface pour exécuter une opération (en différé)
  - `ConcreteCommand` définit un lien entre un objet `Receiver` et une action, et implémente `Execute()` pour l'invocation des opérations 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

Commande (*Command*, *Action*)

- 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

Commande (*Command*, *Action*)

- Avantages
  - Supprime tout couplage entre l'objet l'objet qui invoque l'opération (*i.e.*, l'invocateur) de celui qui la réalise (*i.e.*, le récepteur)
  - Fait des 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

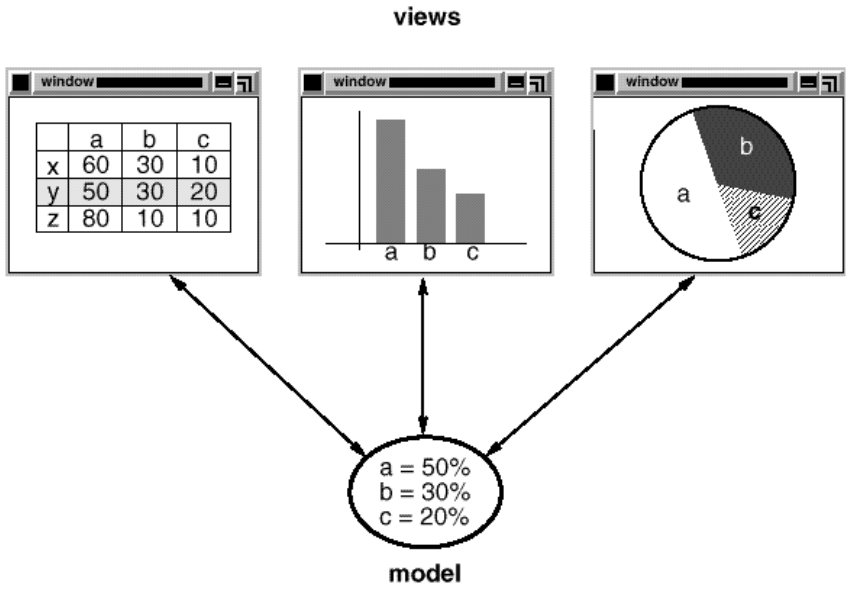
Commande (*Command*, *Action*)

- Implémentation
  - À quel point une commande doit-elle être intelligente ?
    - ★ Simple : délègue simplement les actions requises à un récepteur
    - ★ Intelligent : implémente tout elle-même sans rien déléguer à un récepteur (commandes indépendantes des classes existantes, aucun récepteur approprié n'existe)
  - Comment supporter les opérations *undo* et *redo* ?
    - ★ Nécessite de stocker des informations supplémentaires
    - ★ Nécessite un historique des commandes pour réaliser les *undo* et *redo* à plusieurs niveaux

Retour à notre exemple

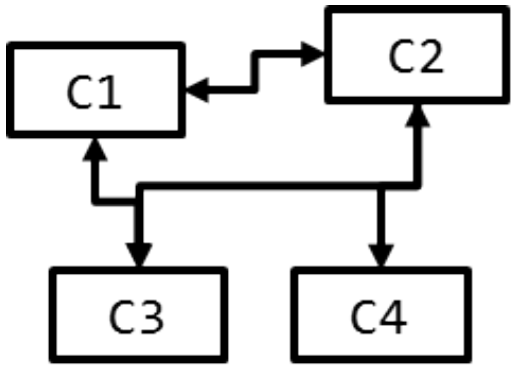
```
public interface Command {
  void execute();
  void undo();
}
public class OpenCommand implements Command {
  private Editor editor;
  public OpenCommand(Editor e) {...}
  public void execute() {
    ...
    editor.add(document);
    ...
  }
  public void undo() {...}
}
public class CutCommand implements Command {
  private Document document;
  public CutCommand(Document d) {...}
  public void execute() {
    document.cut();
  }
  public void undo() {...}
}
```

Problème : MVC

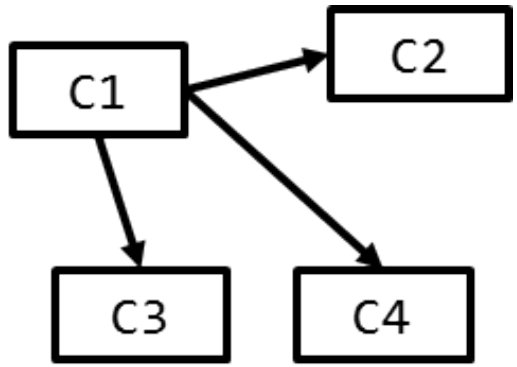


- Comment faire pour que chacun d'eux soit informé de ces changements ?
- Comment maintenir un faible couplage entre diffuseur et souscripteurs ?
- Autrement dit, comment assurer la cohérence entre des classes coopérant entre elles tout en maintenant leur indépendance ?
- Que proposez-vous comme conception ?

Diagramme de classes



✗



✓

- Couplage fort = risque de propagation d'erreurs
- Il faut limiter le couplage entre les classes

Observateur (*Observer*)

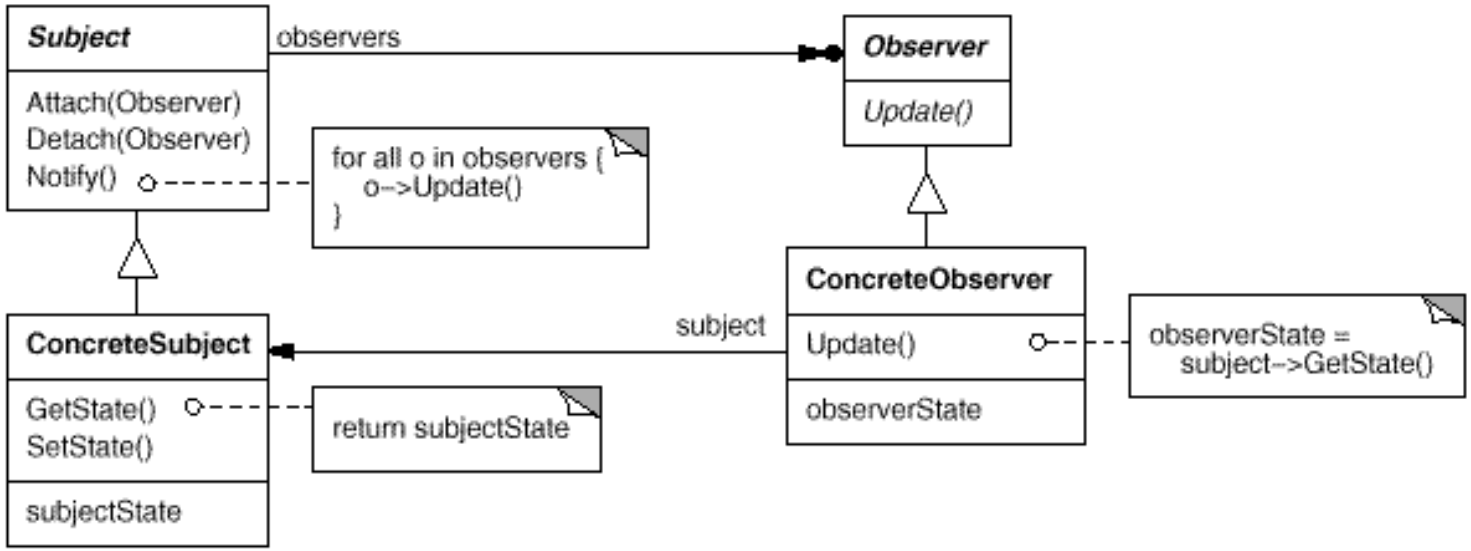
- Intention
  - Définir une interdépendance de type un à plusieurs, de telle sorte que si un objet change d'état, tous ceux qui en dépendent en soient notifiés et mis à jour automatiquement
- Problème
  - Ce qui varie : la liste des objets dépendant des modifications d'un objet de référence (sujet)
  - On veut avertir une liste variable d'objets qu'un événement a eu lieu
  - Nécessité de maintenir la cohérence d'un système en classes coopérantes
- Solution
  - Le modèle de base est constitué d'un sujet (détenteur de données) et d'observateurs
  - Tous les objets reçoivent une notification chaque fois que le sujet subit une modification
  - Les observateurs doivent s'enregistrer dynamiquement auprès du client

Observateur (*Observer*)

- Indications d'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
  - Un objet doit être capable d'en notifier d'autres sans savoir qui ils sont (*i.e.*, que ces objets ne doivent pas être fortement couplés)
- Exemple dans l'API Java
  - La classe `Observable` et l'interface `Observer`
  - Tous les objets graphiques (bouton, boîte de dialogue, champs de texte, *etc.*) de la bibliothèque Swing fonctionnent sur le modèle d'observateur (les *listeners*)

Observateur (*Observer*)

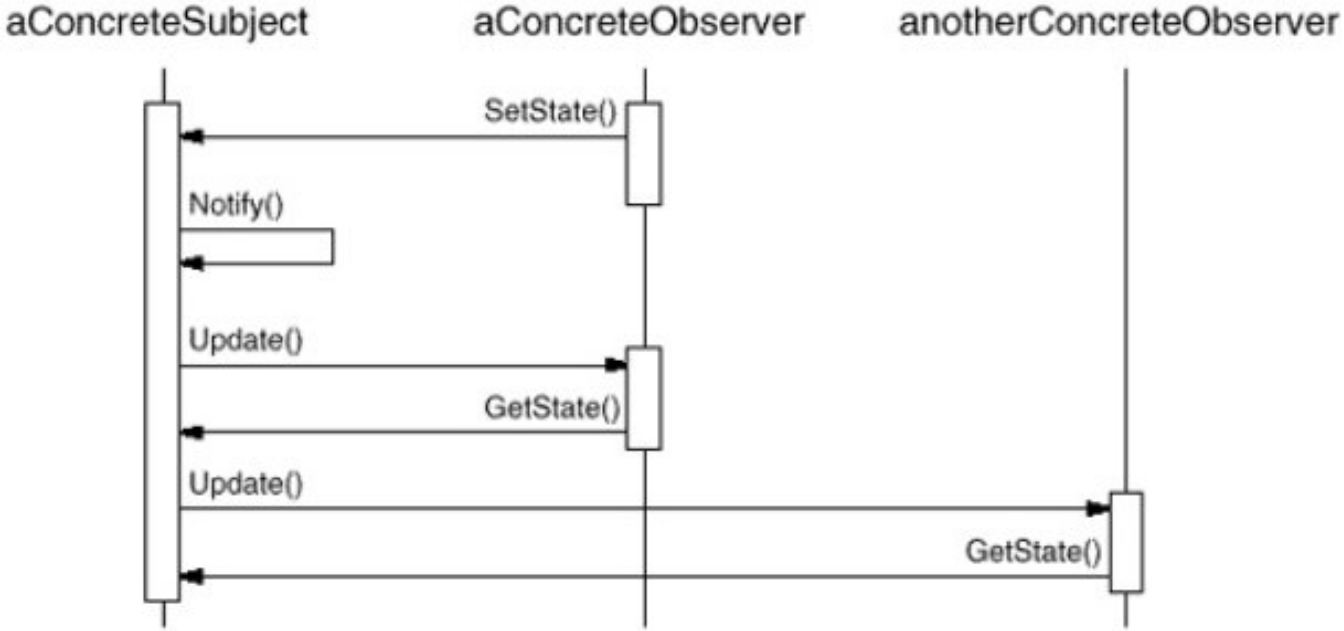
- Structure



- 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 son état 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

Observateur (*Observer*)

- Collaboration



- Les observateurs sont responsables de son inscription auprès d'un sujet en utilisant la méthode `Attach()` du sujet
- 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

Observateur (*Observer*)

- Avantages
  - Favorise un couplage faible entre le sujet et ses observateurs
    - ★ Il est possible de réutiliser les sujets sans réutiliser leurs observateurs et *vice versa*
    - ★ Des observateurs peuvent être ajoutés à chaud sans modifier le sujet
    - ★ Le sujet connaît sa liste d'observateurs
    - ★ Le sujet n'a pas besoin de connaître la classe concrète d'un observateur, juste que chaque observateur implémente l'interface de mise à jour
    - ★ Les sujets et les observateurs peuvent appartenir à différentes couches d'abstraction
  - Fournit un support à la diffusion d'événements (*broadcast*)
    - ★ Le sujet envoie une notification à tous les observateurs abonnés
    - ★ Des observateurs peuvent être ajouté/supprimé à tout moment
- Inconvénients
  - Peut causer une cascade de notifications/mises à jour inopinées
    - ★ Parce que les observateurs ignorent la présence des uns et des autres, ils doivent faire attention au déclenchement des mises à jour (coûteuses)
  - Une simple interface de mise à jour requiert que les observateurs déduisent eux-même de ce qui a changé dans le sujet (difficile)
    - ★ Besoin d'un protocole additionnel pour les aider

Observateur (*Observer*)

- Implémentation
  - Comment le sujet garde-t-il une trace de ses observateurs ?
    - ★ Tableau, liste chaînée vs table d'association : coût différent
  - Que faire si un observateur veut observer plus d'un sujet ?
    - ★ Étendre l'interface de mise à jour pour savoir qui notifie
  - Qui déclenche la mise à jour ?
    - ★ Le sujet chaque fois que son état change
    - ★ Les observateurs après qu'ils causent un ou plusieurs changements d'état
    - ★ Un ou plusieurs objets tiers
  - Que faire lors de la destruction d'un sujet ?
    - ★ Détruire les observateurs : non !
    - ★ Notifier les observateurs : oui
  - Toujours s'assurer que le sujet met à jour son état avant d'envoyer des notifications

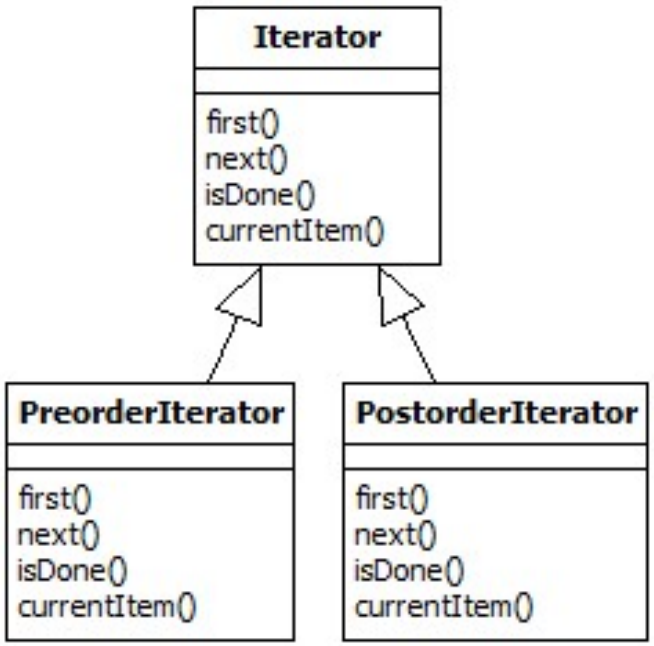
Observateur (*Observer*)

- Implémentation (suite)
  - Combien d'informations concernant le changement le sujet doit-il envoyer aux observateurs ?
    - ★ Modèle *push* : beaucoup, le sujet envoie aux observateurs des informations détaillées sur les modifications, et donc potentiellement des informations superflues pour certains observateurs intéressés que par un sous-ensemble seulement (`Update(Data data)`)
    - ★ Modèle *pull* : très peu, le sujet n'envoie aucune information (donc aucune superflue), ce sont les observateurs qui demandent les informations complémentaires au sujet (`Update(Subject s)`)
  - Les observateurs peuvent-ils souscrire à des événements spécifiques ?
    - ★ Modèle *publish-subscribe*
  - Un observateur peut-il être aussi un sujet ?
    - ★ Oui !
  - Que faire si un observateur veut être notifié uniquement après que plusieurs sujets ont changé d'état ?
    - ★ Utiliser un objet intermédiaire qui agit comme un médiateur
    - ★ Les sujets envoient les notifications à l'objet médiateur qui réalise un traitement nécessaire avant de notifier les observateurs

Problème : vérification orthographique

- La vérification orthographique nécessite de parcourir le document
  - Il est nécessaire de voir tous les glyphes, dans l'ordre
  - Les informations dont nous avons besoin sont dispersées partout dans le document
- Nous voulons également pouvoir effectuer d'autres analyses (*e.g.*, une analyse grammaticale)
- Autrement dit, comment accéder aux objets d'une structure (*e.g.*, un composite) sans se soucier de l'organisation de la structure ou encore d'éventuels changements ?
- Que proposez-vous comme conception ?

Diagramme de classes



- Une solution est de cacher la structure d'un conteneur (ici, le document) aux clients
- Une méthode pour :
  - Pointer sur le premier élément
  - Avancer à l'élément suivant
  - Obtenir l'élément courant
  - Tester la terminaison

Itérateur (*Iterator*, *Cursor*)

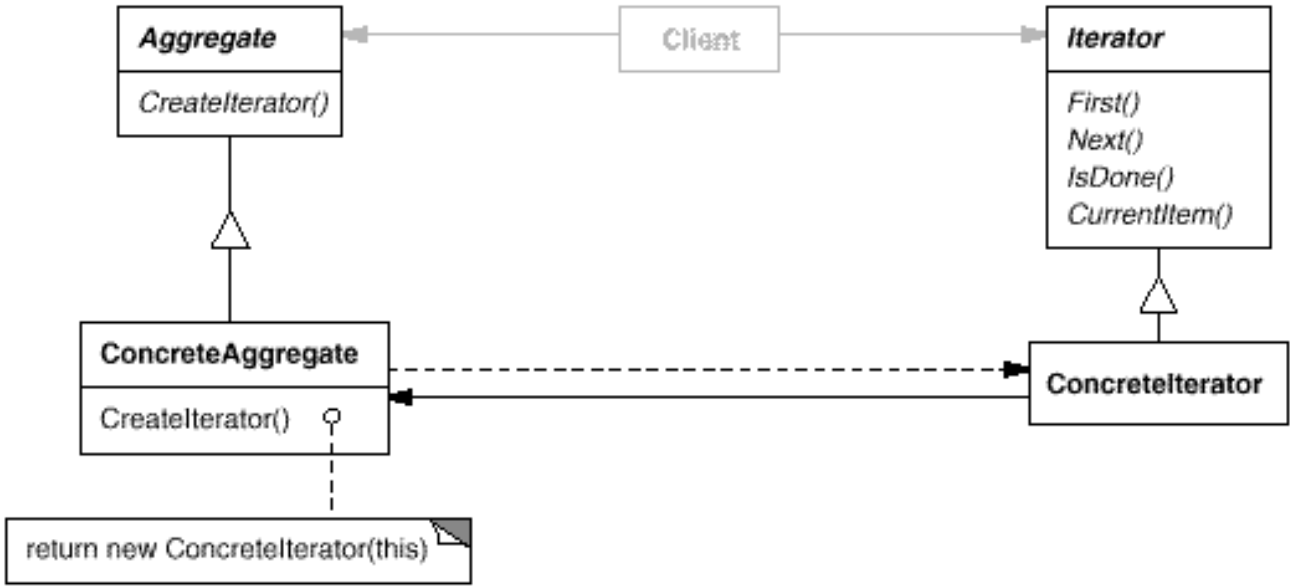
- Intention
  - Fournir un moyen de parcourir séquentiellement un agrégat (collection, conteneur) d'éléments sans connaître sa structure interne
- Problème
  - Séparer le mécanisme de parcours d'un agrégat :
    - ★ Sans risque pour la structure interne de l'agrégat
    - ★ Accès de façon variée, selon les besoins
    - ★ Plusieurs parcours possibles
- Solution
  - Déléguer le parcours d'un agrégat dans un objet annexe

Itérateur (*Iterator*, *Cursor*)

- Indications d'utilisation
  - On veut accéder au contenu (*i.e.*, les éléments) d'un agrégat sans révéler sa représentation interne
    - ★ Les itérateurs sont souvent utilisés pour traverser récursivement des structures composites
  - On souhaite gérer simultanément plusieurs parcours d'un agrégat
  - On a besoin d'offrir une interface uniforme pour parcourir différentes structures d'agrégats (*i.e.*, itération polymorphe)
- Exemple dans l'API Java
  - L'interface des itérables : `Iterable` (qui définit la méthode `iterator()`)
  - L'interface des itérateurs d'agrégat : `Iterator`

Itérateur (*Iterator*, *Cursor*)

- Structure



- Participants
  - `Iterator` définit une interface pour accéder aux éléments et les parcourir
  - `ConcreteIterator` implémente l'interface `Iterator`, assure le suivi 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 l'itérateur, afin de retourner l'instance adéquate d'un itérateur concret

Itérateur (*Iterator*, *Cursor*)

- Avantages
  - Simplifie l'interface de l'agrégat, en ne la 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'itéareur par une autre différente
    - ★ En définissant une nouvelle sous-classe de `Iterator`

Itérateur (*Iterator*, *Cursor*)

- Implémentation
  - Qui contrôle l'itération ?
    - ★ Le client : itérateur externe (plus flexible)
    - ★ L'itérateur : itérateur interne (plus facile d'utilisation)
  - Qui définit l'algorithme de parcours ?
    - ★ L'itérateur : plus courant et plus facile d'utiliser différents algorithmes d'itération sur le même agrégat ou le même algorithme sur différents agrégats (attention à ne pas casser l'encapsulation de l'agrégat pour accéder à ses variables privées ; en Java, l'itérateur sera une classe interne de son agrégat)
    - ★ L'agrégat : l'itérateur ne conserve que l'état de l'itération (curseur)
  - L'agrégat peut-il être modifié pendant qu'il est parcouru ?
    - ★ Dangereux
    - ★ Une solution simple est de copier l'agrégat et de parcourir la copie, mais coûte trop cher en général
    - ★ Un itérateur robuste permet de faire des insertions et des suppressions, sans affecter le parcours, ni faire de copie de l'agrégat
  - Doit-on augmenter l'interface de `Iterator` avec des opérations supplémentaires, telles que `previous()` ou `goto()` ?
    - ★ Dépend de l'application

Retour à notre exemple

```
...
Iterator i = document.createIterator();
i.first();
while(!i.isDone()) {
   Glyph glyph = i.currentItem();
   ... // do something with the glyph
   i.next();
}
...
```

Problème : vérification orthographique (suite)

- Le pattern Itérateur marche bien si nous n'avons pas besoin de connaître le type des éléments en train d'être parcourus
- Donc il ne convient pas au problème de vérification orthographique, car il faut caster `i.currentItem()` :

```
if(i.currentItem() instanceof Character) {...} else {...}
```

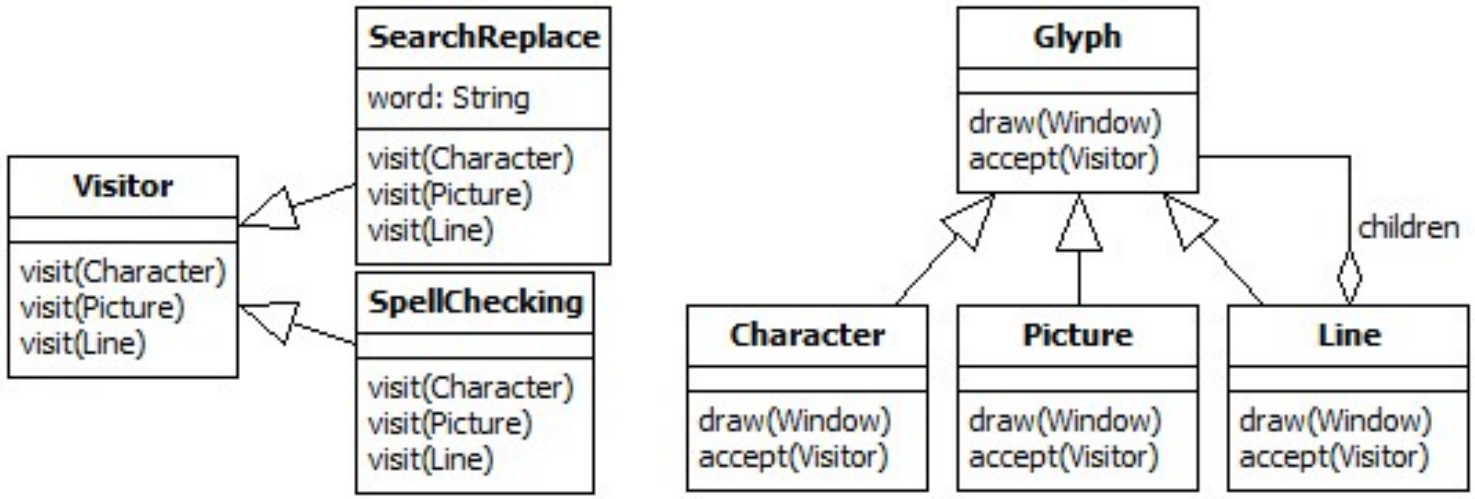
- Autrement dit, comment effectuer des opérations qui peuvent avoir besoin de traiter chaque type d'élément d'une structure différemment ?
- Que proposez-vous comme conception ?

Pas très bon

- Définir chaque opération dans la classe de l'élément
- Problèmes :
  - Ajouter de nouvelles opérations nécessite de modifier toutes les classes d'élément
  - Avoir un tel ensemble varié d'opérations dans chaque classe d'élément peut être dur à comprendre et à maintenir

Diagramme de classes

- La 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
  - Aiguillage dynamique vers les méthodes `accept` des classes `Character`, `Picture`, *etc.*
- Chaque méthode `accept` :
  - Appelle l'action spécifique au visiteur passé en argument (*e.g.*, `visit(Character)`)
  - Implémente la traversée (*e.g.*, la méthode `accept` de la classe `Line`)
- Il faut un visiteur pour chaque action (*e.g.*, rechercher-remplacer)



Visiteur (*Visitor*)

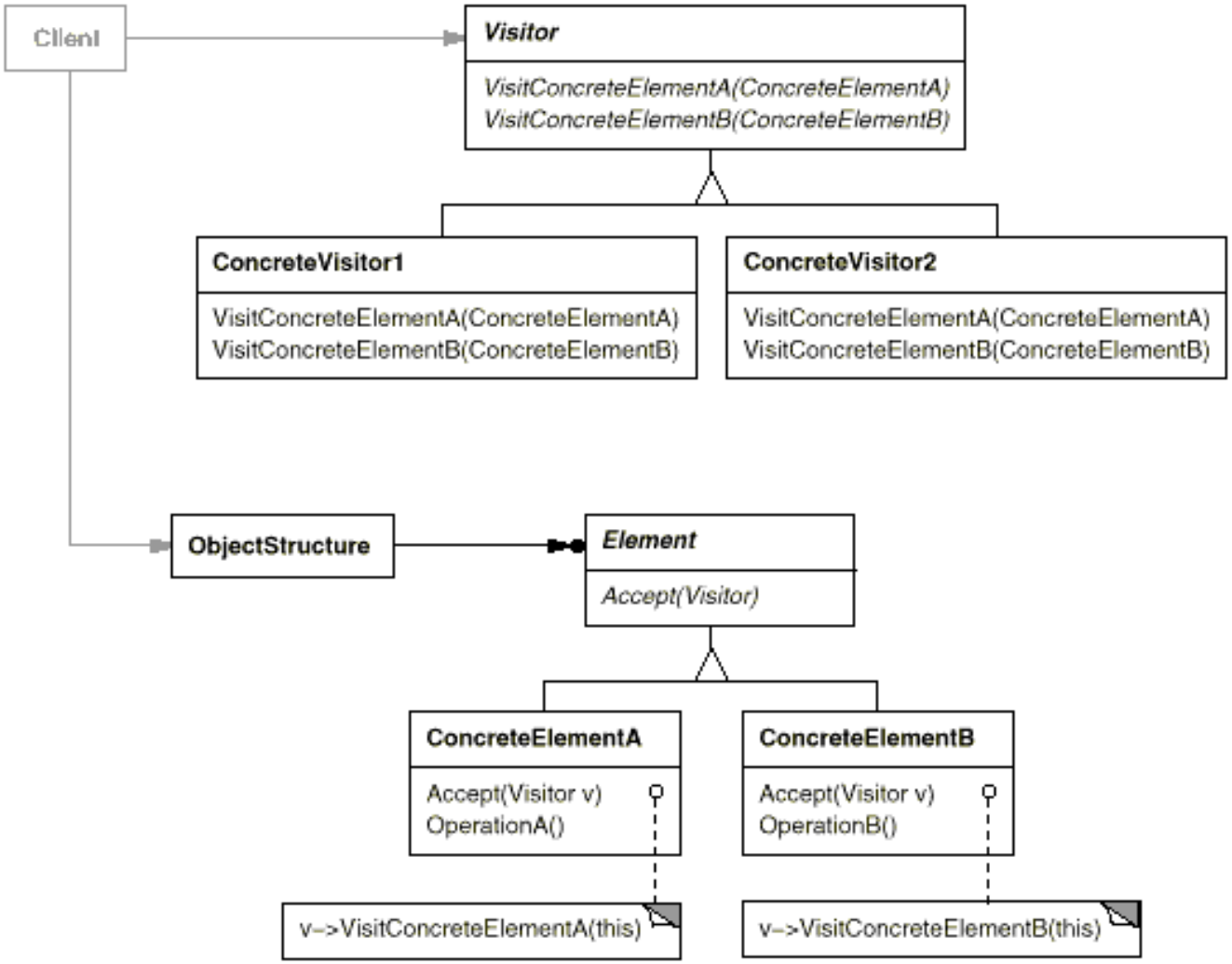
- Intention
  - Représenter une opération à effectuer sur les éléments d'une structure et permettre de définir une nouvelle opération sans modifier les classes des éléments sur lesquels il opère
- Problème
  - Ce qui varie : la liste des services rendus par une structure d'objet
  - Différentes opérations distinctes doivent être réalisées sur les nœuds d'une structure composite, mais on ne souhaite pas polluer les classes des nœuds avec ces opérations, ni tester le type de chaque nœuds pour transtyper la référence avec le bon type avant de réaliser l'opération
- Solution
  - Externaliser les opérations d'une structure d'objets dans une hiérarchie séparée et ajouter une méthode dans la structure d'objet pour accueillir des instances des opérations
- Métaphore
  - Quand une personne appelle une compagnie de taxi, la compagnie envoie un véhicule au client
  - Quand il entre dans le taxi, le client ne gère plus son transport, c'est le taxi qui le fait

Visiteur (*Visitor*)

- Indications d'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

Visiteur (*Visitor*)

- Structure

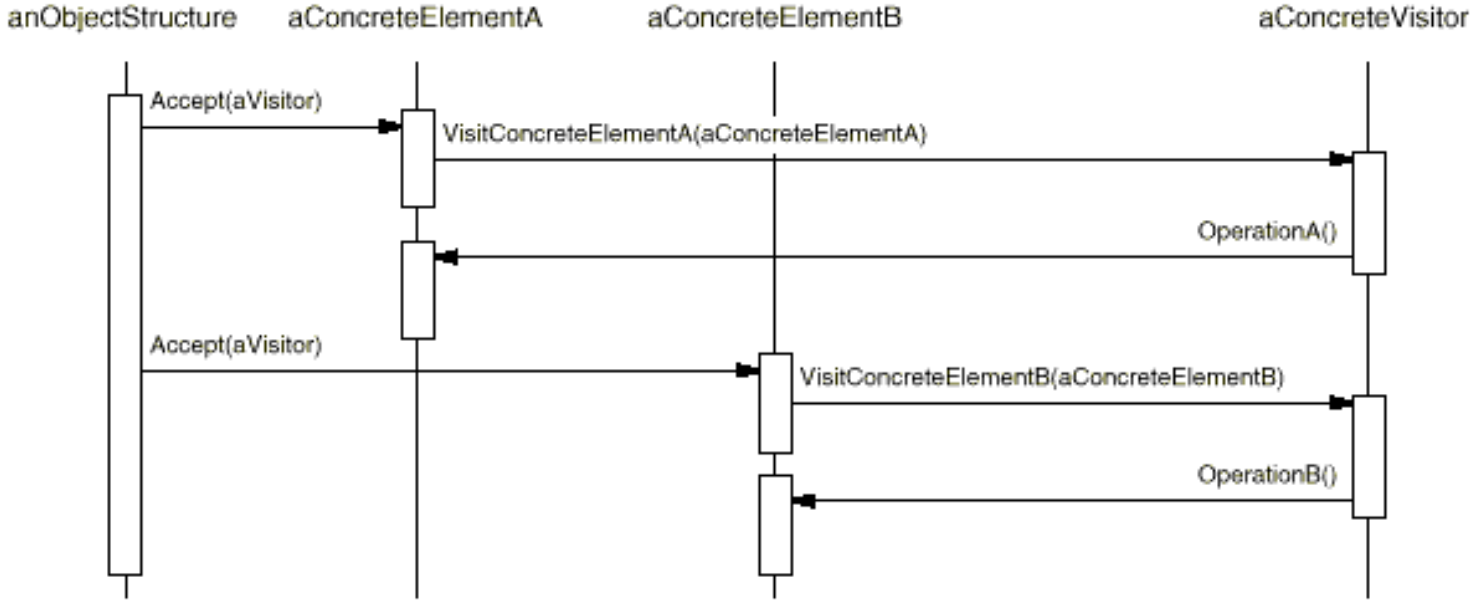


Visiteur (*Visitor*)

- 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 fournit une interface de haut niveau permettant au visiteur de visiter ses éléments

Visiteur (*Visitor*)

- 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

Visiteur (*Visitor*)

- Avantages
  - Facilite l'ajout de nouvelles opérations, en ajoutant un nouveau visiteur
  - 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` + 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

Visiteur (*Visitor*)

- 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 + types des 2 receveurs (le visiteur et l'élément)
    - ★ Méthode `Accept` pour effectuer la liaison à l'exécution
  - Qui est responsable du parcours de la structure ?
    - ★ La structure d'éléments, le plus souvent mais figée
    - ★ Le visiteur, 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
    - ★ Un itérateur, retour aux deux cas précédents

Retour à notre exemple

```
public abstract class Glyph {
  public abstract void draw(Window w);
  public abstract boolean intersects(Point p);
  ...
  public abstract void accept(Visitor v);
}

public class Character extends Glyph {
  ...
  public void accept(Visitor v) {
    v.visit(this);
  }
}

public class Line extends Glyph {
  ...
  public void accept(Visitor v) {
    v.visit(this);
    for (Glyph g : children)
      g.accept(v);
  }
}
```

Retour à notre exemple (suite)

```
public interface Visitor {
  void visit(Character c);
  void visit(Picture p);
  void visit(Line l);
  ...
}

public class SpellChecking implements Visitor {
  public void visit(Character c) {...}
  public void visit(Picture p) {} //do nothing
  public void visit(Line l) {...}
  ...
}

...
Visitor v = new SpellChecking();
line.accept(v);
```

Itérateur vs Visiteur

- Le pattern Itérateur fournit des parcours de conteneurs
- Le pattern Visiteur est plus général et permet :
  - Le parcours
  - 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 redéfinie pour un parcours particulier

Problème : sauvegarde d'un document

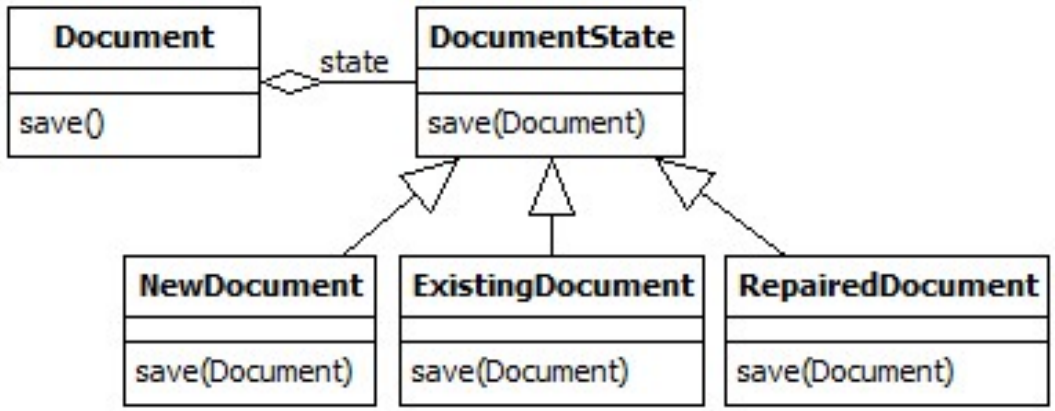
- La sauvegarde d'un document est différente selon si c'est :
  - Un nouveau document
  - Un document déjà existant sur le disque
  - Un document récupéré après un arrêt anormal de l'éditeur de texte
  - *etc.*
- Autrement dit, comment structurer et représenter les différents états d'un système (et les transitions entre ces états) dont dépendent certains de ses comportements ?
- Que proposez-vous comme conception ?

Pas très bon

- Utiliser des constantes (représentant les états) dans des conditionnelles
  - Souvent, plusieurs opérations contiennent les mêmes structures conditionnelles
  - Que se passe-t-il lors de l'ajout/suppression d'un état/transition ?

Diagramme de classes

- La solution met chaque branche des conditionnelles dans une classe à part
- Créer des classes d'états qui implémentent une interface commune



État (*State*)

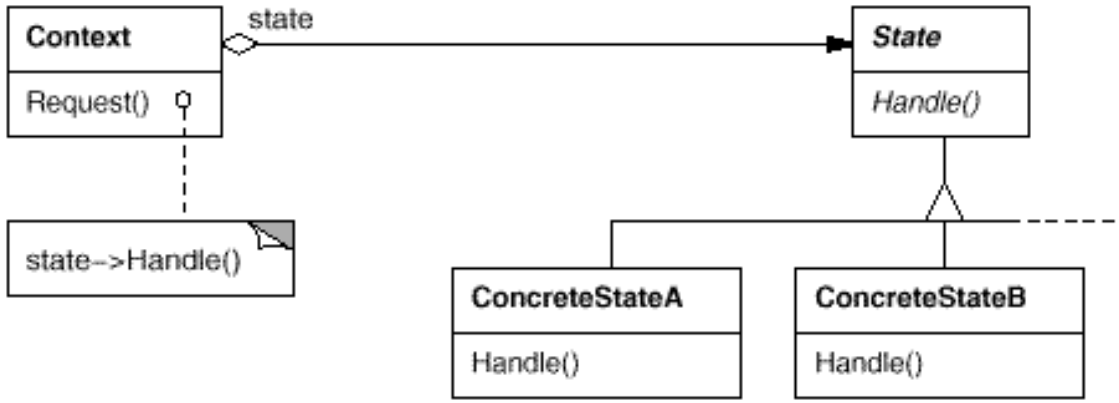
- Intention
  - Permettre de modifier le comportement d'un objet lorsque son état interne change (tout se passe comme si l'objet changeait de classe)
- Problème
  - Ce qui varie : la nature des services rendus par un objet en fonction de son état
- Solution
  - Externaliser dans un objet annexe les méthodes qui sont dépendantes d'un état particulier d'un objet
  - Faire autant d'objets annexes que d'états différents

État (*State*)

- Indications d'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

État (*State*)

- Structure



- Participants
  - `Context` définit l'interface qu'utilise les clients et gère une instance d'une sous-classe `ConcreteState` qui définit 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`

État (*State*)

- Collaboration
  - `Context` est la principale interface pour les clients qui peuvent la configurer avec des objets `State`, puis n'ont plus à traiter directement avec les états
  - Il délègue 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

État (*State*)

- Avantages
  - Sépare les comportements relatifs à chaque état et fait un partitionnement des différents comportements, état par 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
  - Élimine les conditionnelles
  - Rend les transitions entre états plus explicites
- Inconvénients
  - Augmente le nombre d'objets

État (*State*)

- Implémentation
  - Les classes états doivent avoir un accès privilégié aux données de la classe `Contexte`
    - ★ Classes internes en Java
  - Qui définit les transitions entre états ?
    - ★ La classe `Context` : pour des situations simples
    - ★ Les classes `ConcreteState` : généralement plus flexible, mais entraîne des dépendances d'implémentation entre les classes `ConcreteState`
  - Quand sont créés les objets `ConcreteState` ?
    - ★ Lorsque c'est nécessaire : les états ne sont pas connus à la compilation et le contexte change d'état rarement
    - ★ Une seule fois en avance, le contexte gère des références à ces objets : les changements d'état arrivent fréquemment
  - N'est-il pas possible d'utiliser juste une table de transitions d'état ?
    - ★ Le pattern `État` se fonde sur le comportement propre d'un état, alors que le modèle des tables se focalise sur les transitions
    - ★ Difficile d'ajouter d'autres actions et comportements

Retour à notre exemple

```
public interface DocumentState {
  boolean save(Document d);
}
public class NewDocument implements DocumentState {
  public boolean save(Document d) {
    ...
    d.setState(new ExistingDocument()); //transition
    ...
  }
}
public class ExistingDocument implements DocumentState {
  public boolean save(Document d) {...}
}
public class Document {
  private DocumentState state;
  public void setState(DocumentState s) {
    state = s;
  }
  public boolean save() {
    return state.save(this);
  }
  ...
}
```

Problème : mise en forme

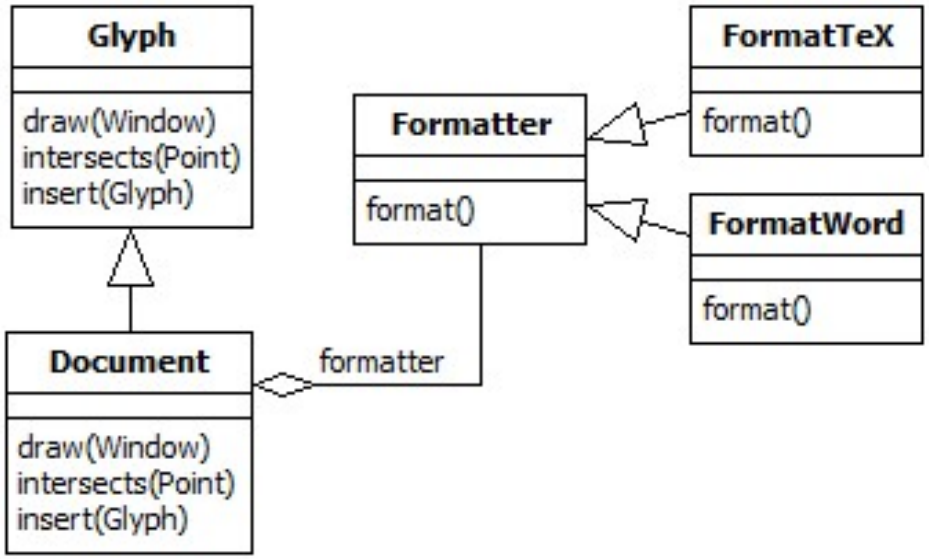
- Une structure physique particulière pour un document
  - Décisions concernant la mise en page (*e.g.*, gestion des retours à la ligne)
- La mise en forme est un problème complexe
  - Il n'y a pas de meilleur algorithme
    - ★ Beaucoup d'alternatives, des simples aux plus complexes
  - Nous ne voulons pas que cette complexité pollue la classe `Glyph`
  - Nous voulons pouvoir changer d'algorithme de mise en forme
- Autrement dit, comment représenter et utiliser différentes variantes d'un même algorithme pour un comportement particulier ?
- Que proposez-vous comme conception ?

Pas très bon

- Ajouter une méthode `format` à chaque sous-classe de `Glyph`
  - Problèmes :
    - ★ Ne permet pas de modifier l'algorithme sans modifier `Glyph`
    - ★ Ne permet pas de facilement ajouter de nouveaux algorithmes de mise en forme
- Définir plusieurs sous-classes de `Glyph`, chacune avec une variante de l'algorithme de mise en forme
  - Problèmes :
    - ★ Couple `Glyph` à l'algorithme, les deux deviennent plus difficiles à modifier
    - ★ Ne permet pas de changer l'algorithme dynamiquement

Diagramme de classes

- Encapsuler la mise en forme (*i.e.*, l'algorithme) derrière une interface
  - Chaque algorithme de mise en forme est mis dans une classe
  - `Document` traite uniquement avec l'interface



Stratégie (*Strategy*)

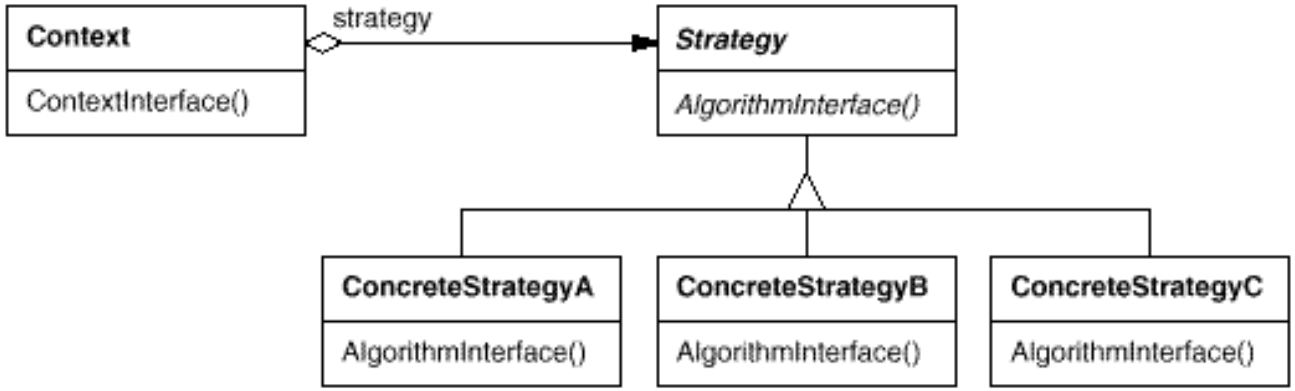
- Intention
  - Définir une famille d'algorithmes, encapsuler chacun d'eux, et les rendre interchangeables
  - Utiliser différents algorithmes identiques sur le plan conceptuel en fonction du contexte
  - Permettre aux algorithmes d'évoluer indépendamment des clients qui les utilisent
- Problème
  - Ce qui varie : l'algorithme implémentant un service
- Solution
  - Encapsuler les algorithmes dans une hiérarchie et lier la hiérarchie à l'objet appelant par composition
  - Séparer la sélection de l'algorithme de son implémentation
  - Le pattern permet une sélection dynamique fondée sur le contexte
- Métaphore
  - Un voyageur dispose de plusieurs modes de transport pour se rendre à un aéroport
  - Il choisit la stratégie qui lui semble la plus adaptée en fonction de critères, tels que le prix, le confort, le temps de parcours....

Stratégie (*Strategy*)

- Indications d'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 (*e.g.*, 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
- Exemple dans l'API Java
  - L'interface `Comparable` qui donne la possibilité de personnaliser des fonctions de comparaison entre deux objets

Stratégie (*Strategy*)

- Structure



- Participants
  - `Strategy` déclare une interface commune à tous les algorithmes et est utilisée par `Context` pour appeler l'algorithme défini par une stratégie concrète
  - `ConcreteStrategy` implémente l'algorithme en utilisant l'interface `Strategy`
  - `Context` choisit une référence à un objet `ConcreteStrategy` et peut définir une interface qui permet à cet objet d'accéder à ses données

Stratégie (*Strategy*)

- 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)

Stratégie (*Strategy*)

- 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
    - ★ Chacune peut être modifiée indépendamment l'une de l'autre
    - ★ L'algorithme peut être changé dynamiquement (*i.e.*, à l'exécution)
    - ★ La compréhension et l'extension des algorithmes sont facilitées
  - Élimine les conditionnelles pour sélectionner le bon comportement
  - Fournit différentes implémentations du même comportement
    - ★ Les clients peuvent choisir parmi 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
  - 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`

Stratégie (*Strategy*)

- Implémentation
  - Les interfaces `Strategy` et `Context` doivent assurer à une stratégie concrète l'accès aux données d'un contexte dont elle a besoin
  - Il y a deux possibilités :
    1. Les informations peuvent être passées sous forme de paramètres (`AlgorithmInterface(Data data)`)
    2. Le contexte peut fournir sa référence en paramètre, et la stratégie lui requête les données (`AlgorithmInterface(Contexte c)`)
  - Dans le dernier cas, il faut un lien privilégié entre les deux classes : en Java, on utilisera des classes internes

Retour à notre exemple

```
public class Document {
  private Formatter formatter;
  public void setFormatter(Formatter f) {
    formatter = f;
  }
  ...
}
public interface Formatter {
  void format(...);
}
public class FormatTeX implements Formatter {
  public void format(...) {...}
}
public class FormatWord implements Formatter {
  public void format(...) {...}
}
...
Document d = new Document(...);
d.setFormatter(new FormatTeX(...));
d.insert(line);
```

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 ils n'ont pas le même objectif !
  - Un objet `State` encapsule un comportement dépendant d'états (et éventuellement les transitions entre états)
  - Un objet `Strategy` encapsule un algorithme