Modèles de Conception Réutilisables

Pier Donini (Pier.Donini@heig-vd.ch)

Répertoire d'enseignement: \\eistore1\cours\tic\Pi\MCR

	1	2	3	4	5	6	7	8
1h	cours	cours	cours	cours	cours	cours	cours	cours
1h15	labo 1	labo 1	labo 1	labo 2	labo 2	labo 2	projet / T	projet / T

	9	10	11	12	13	14	15	16
1h	TE	projet / T	prés. théorique	projet / P	projet / P	projet / P	projet / P	prés. pratique
1h15	projet / T							

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Hes·so
Haute Ecole Spécialisée de Suisse Occidentale

Introduction

- L'utilisation de modèles de conception réutilisables (*design patterns*) est issue des études de l'architecte Christopher Alexander sur l'amélioration de la conception d'immeubles et de zones urbaines.
- « Chaque modèle est une règle […] exprimant une association entre un certain contexte, un problème et une solution » ou, autrement dit,
  - Un modèle est une solution à un problème dans un contexte.
- Les modèles de conception sont applicables dans de nombreux domaines, dont celui de la conception orientée-objet.
  - « Concevoir du logiciel orienté-objet est difficile, et concevoir du logiciel orienté-objet *réutilisable* l'est plus encore » - Erich Gamma.

Introduction (2)

- Conception orientée-objet:
  - Répétition de certains profils de classes, d'associations entre elles et d'hiérarchies d'héritage dans de nombreux systèmes.
  - Les concepteurs expérimentés réutilisent des solutions éprouvées pour des problèmes courants de conception logicielle.
- Intérêt des modèles de conception:
  - Facilitent la réutilisation de modélisations et d'architectures efficaces.
  - Permettent aux concepteurs de bénéficier d'une expérience collective.
  - Facilitent la communication entre concepteurs en fournissant un vocabulaire commun.
  - Par leur formalisation et leur niveau d'abstraction élevé, rendent plus accessibles des techniques usuelles.
  - Améliorent la documentation et la maintenance par la spécification des relations entre classes ainsi que leurs mobiles sous-jacents.

Introduction (3)

- Inconvénients des modèles de conception:
  - Apprentissage non trivial (maîtrise de leurs motivations et des mécanismes orientés-objet sur lesquels ils s'appuient).
  - Assimilation de nombreux modèles.
  - Nécessitent un effort d'abstraction pour savoir reconnaître les situations où il est possible de les appliquer.
  - Peuvent occasionner la multiplication des classes dans un diagramme.
- Un modèle de conception n'est pas forcément la meilleure solution a un problème donné, mais est certainement l'une des moins mauvaises…

Principes OO: Encapsulation

- Les classes devraient être *opaques*:
  - L'encapsulation permet de masquer la structure interne et les détails d'implémentation d'un objet.
  - Les interactions avec un objet s'effectuent au moyen des opérations définies dans son interface publique.
- En général, définir des attributs privés et leurs accesseurs (`getXXX`) et mutateurs (`setXXX`) publics.
  - Permet de contraindre les valeurs des attributs (cohérence de l'objet).
  - La représentation interne de la classe peut être modifiée sans impacter sur son interface ➔ pas de modification des classes y accédant.
  - Assure que des effets de bord désirés seront toujours effectués.

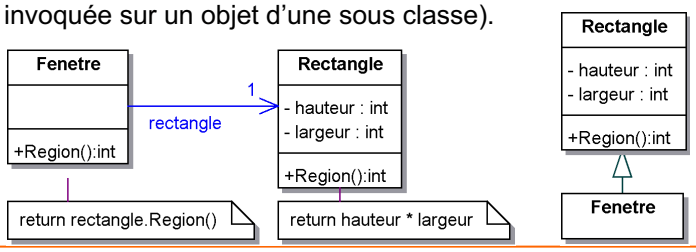
```
public void setSpeed(int speed) {
  if (speed < 0) throw new RuntimeException("…");
  this.speed = speed;
  notifyObservers();
}
```


Principes OO: Héritage

- Avantages
  - Classification hiérarchique: abstraction ➔ spécialisation.
  - Implémentation aisée d'une sous-classe (méthodes et attributs hérités tels quels).
  - Modification ou extension de l'implémentation réutilisée.
  - Mise en œuvre du mécanisme de liaison dynamique.
- Inconvénients
  - Défini statiquement (à la compilation). Pas de modification possible.
  - L'implémentation des super classes est généralement visible depuis les sous classes (*boite blanche*).
  - Rompt l'encapsulation en assujettissant une sous classe à des détails d'implémentation des super classes.
  - La modification de l'implémentation d'une super classe impose souvent une modification de la sous-classe.

Principes OO: Composition

- Méthode de réutilisation où une nouvelle fonctionnalité est obtenue en créant un objet résultant de la composition (ou agrégation) d'autres objets.
  - Peut être effectuée par valeur (C++) ou par référence (C++, Java, C#).
- Délégation:
  - Composition où deux objets sont impliqués.
  - L'objet récepteur d'une requête délègue son traitement à son délégué.
  - Analogue à l'héritage (traitement dans une super classe d'une méthode invoquée sur un objet d'une sous classe).

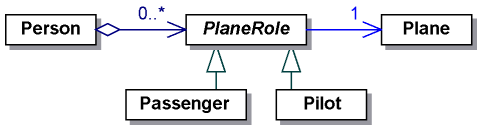


Principes OO: Composition (2)

- Avantages
  - Pas de rupture de l'encapsulation: les objets impliqués doivent posséder des interfaces bien définies (*boite noire*).
  - Minimise les dépendances d'implémentation.
  - Chaque classe se focalise sur une tâche ➔ hiérarchies plus simples.
  - La composition entre différents objets est définie dynamiquement, en spécifiant sur des objets des références à d'autres objets.
  - Les interfaces (Java/C#) permettent d'utiliser des objets instanciés dans des classes non liées par l'héritage dans une même composition.
- Désavantages
  - Les interfaces des objets référencés depuis différentes compositions doivent être particulièrement soignées.
  - Une conception fondée sur la composition contient plus d'objets.
- Préférer la composition à l'héritage (mais ne pas le négliger!).

Principes OO: Composition/héritage

- Règles de Coad:
  - Une sous classe est « un type spécial de » et pas « un rôle joué par ».
  - Une instance d'une sous classe n'aura jamais besoin de devenir une instance d'une autre classe.
  - Une sous classe doit étendre et non pas supprimer les fonctionnalités de la super classe.
- Exemple
  - Un avion contient des passagers et des pilotes.
  - Les pilotes doivent pouvoir être passagers dans d'autres vols.
  - La définition de `Pilot` et `Passenger` comme sous classes de `Person` n'est pas appropriée ➔ Utilisation de la composition. Par exemple:



Principe d'ouverture/fermeture

- *Open/Closed principle* (*OCP*).
- *Tout module (paquetage, classe, méthode) doit être ouvert aux extensions mais fermé aux modifications.* [Meyer]
  - Ouvert: il doit être possible d'étendre le module pour proposer des fonctionnalités non prévues lors de sa conception.
  - Fermé: les extensions sont introduites sans modifier le code existant.
- Mise en œuvre en reposant le code existant sur une abstraction du code amené à évoluer:
  - Polymorphisme.
  - Classes abstraites et d'interfaces.
  - Types génériques (*templates*).
- L'OCP est mis en pratique dans de nombreux modèles de conception.
- Impose une certaine complexité qui n'est utile que si la flexibilité recherchée est effectivement utilisée.

Principe d'ouverture/fermeture (2)

- Soit `Part` une super classe abstraite représentant différents types de pièces.
- Soit une méthode calculant le prix total d'un ensemble de pièces:

```
public static double totalPrice(Part[] parts)
{
  double total = 0;
  for (Part p : parts)
    total += p.getPrice();
  return total;
}
```

- Quand le département des ventes décide de faire des offres sur certaines pièces il n'est pas souhaitable de violer l'OCP par:

```
for (Part p : parts)
  total += (p instanceof Memory ? 0.5 : 1) * p.getPrice();
```

- Exercice: proposer une meilleure modélisation.

Principe de substitution

- *Une méthode utilisant des objets d'une classe doit pouvoir utiliser des objets dérivés de cette classe sans même le savoir.* [Liskov]
- Hériter une classe `Square` d'une classe `Rectangle`?
  - Pour un carré *hauteur* = *largeur* ➔ redéfiniton:

```
public void setHeight(int height) { // idem setWidth
  super.setHeight(height);
  super.setWidth(height);
}
```

  - Après

```
  rectangle.setHeight(10);
  rectangle.setWidth(20);
```

  un client peut raisonnablement s'attendre à ce que `rectangle.area()` rende 200. Mais si c'est un carré qui a été manipulé le résultat vaut 400!
- En conception OO, l'utilisation de l'héritage est validée par la cohérence des comportements de la sous classe avec ceux définis dans la super classe.

Autres principes OO

- Concevoir pour une interface, pas pour une implémentation:
  - Un objet peut implémenter plusieurs interfaces.
  - Les clients ne connaissent pas la classe spécifique de l'objet utilisé.
  - Un objet peut être facilement remplacé par un autre (types différents, mais mêmes interfaces).
  - Séparation des interfaces: les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas.
- Inversion des dépendances:
  - Les modules de haut niveau ne devraient pas dépendre de modules de bas niveau. Tous deux devraient dépendre d'abstractions.
  - Les abstractions ne doivent pas dépendre de détails, les détails doivent dépendre d'abstractions.

Modèles du *GoF*

- Le *GoF*, *Gang of Four* (1991-1994),
  - Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides.
- Livre: « Design Patterns, Elements of Reusable Object-Oriented Software » (« Design Patterns - Catalogue de modèles de conception réutilisables »), Addison-Wesley, 1995.
  - Application de l'idée des modèles de conception à la modélisation OO.
  - Définition d'une structure permettant de classifier et décrire les modèles.
  - Identification et description de 23 modèles de conception réutilisables.
  - Définissent de stratégies et approches OO basées sur ces modèles.
- Il existe de nombreux autres types de modèles de conception que ceux du GoF (J2EE, EJB, propres à des domaines spécifiques…)

Classification des modèles du *GoF*

- Rôle du modèle (ce qu'il effectue):
  - Créateur: concerne le processus de création des objets.
  - Structurel: traite de l'organisation des classes et des objets.
  - Comportemental: spécifie les manières d'intéragir de classes et d'objets et de se répartir les responsabilités (collaborations).
- Domaine du modèle (où il s'applique):
  - Classe:
    - Traite des relations entre les classes et leurs sous classes.
    - Relations établies statiquement par héritage.
  - Objet:
    - Traite des relations entre les objets.
    - Relations généralement établies dynamiquement par composition.

Classification des modèles du *GoF* (2)

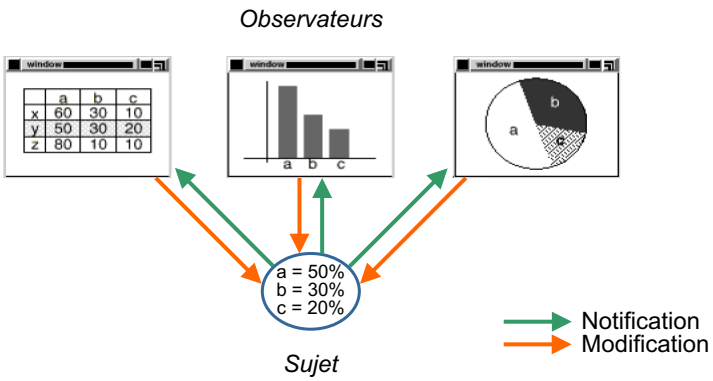
		Rôle		
		Créateur	Structurel	Comportemental
Domaine	Classe	Fabrication	Adaptateur (classe)	Interprète Patron de méthode
	Objet	Fabrique abstraite Monteur Prototype Singleton	Adaptateur (objet) Pont Composite Décorateur Façade Poids mouche Procuration (Proxy)	Chaîne de responsabilités Commande Itérateur Médiateur Mémento Observateur Etat Stratégie Visiteur

Observateur

- [Objet – Comportemental]
- Intention
  - Définit une interdépendance de type un à plusieurs, de façon telle que, quand un objet change d'état, tous ceux qui en dépendent en soient notifiés et mis à jour.
- Motivation
  - Le besoin de maintenir une cohérence d'objets en relation sans introduire un couplage étroit entre les classes (ce qui réduirait leur réutilisabilité).
  - Objets de base: sujet et observateur.
    - Un sujet possède un nombre quelconque d'observateurs sous sa dépendance.
    - Tous les observateurs reçoivent une notification chaque fois que l'état d'un sujet est modifié. En réponse, chaque observateur consulte l'état du sujet afin d'adapter le sien.

Observateur (2)

- Motivation (suite)



Observateur (3)

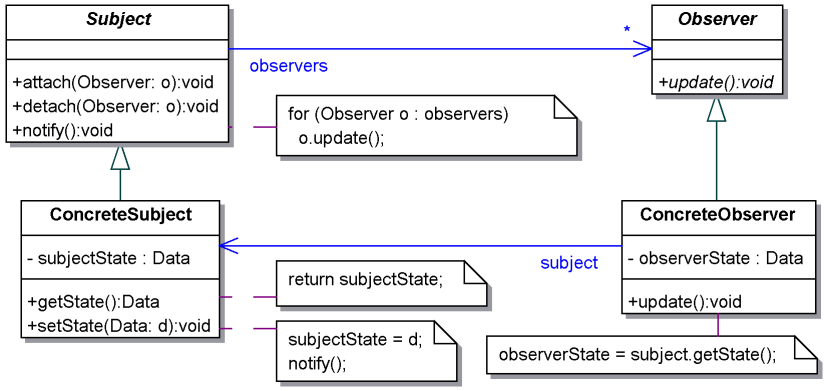
- L'utilisation du modèle observateur est recommandée lorsque:
  - Un concept possède deux représentations dépendantes l'une de l'autre. Encapsuler ces représentations dans des objets distincts permet de les réutiliser et de les modifier indépendamment.
  - Quand la modification d'un objet requiert d'en modifier d'autres et que leur nombre est inconnu.
  - Quand un objet doit être capable de de notifier d'autres objets sans émettre d'hypothèses sur leur nature (i.e., ils ne doivent pas être fortement couplés).

Observateur (4)

- Constituants
  - Sujet:
    - Connaît ses observateurs,
    - Fournit une interface pour attacher et détacher les observateurs.
  - Observateur:
    - Définit une interface pour permettre sa mise à jour.
  - Sujet concret:
    - Mémorise les états intéressant les observateurs concrets,
    - Notifie ses observateurs lorsque son état change.
  - Observateur concret:
    - Possède une référence sur un sujet concret,
    - Possède un état qui doit rester cohérent avec celui du sujet.
    - Implémente l'interface de mise à jour de l'observateur pour conserver la cohérence de son état avec celui du sujet.

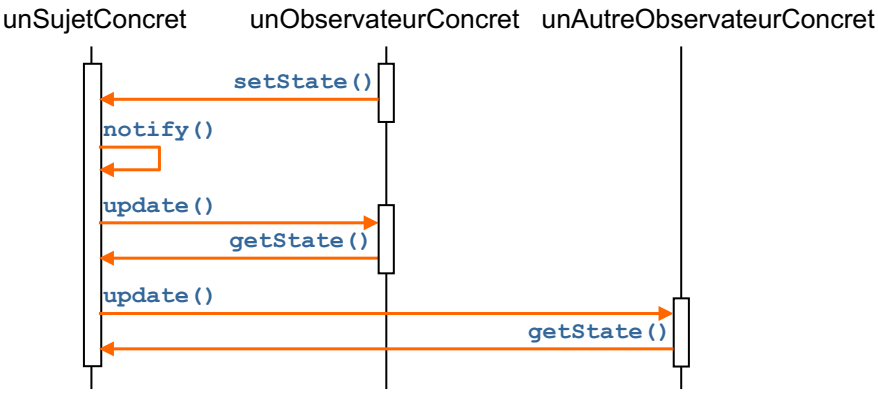
Observateur (5)

- Structure



Observateur (6)

- Collaborations



Observateur (7)

- Conséquences
  - Avantages
    - Couplage minimal entre le sujet et l'observateur

      Le sujet connaît tous ses observateurs depuis le point de vue `Observer`, il ne connaît pas la classe concrète de chacun d'eux.

      Sujet et observateur peuvent ainsi appartenir à des différents niveaux d'abstraction du système.
    - Le sujet notifie automatiquement tous les objets qui se sont abonnés.
    - Des observateurs peuvent être ajoutés/enlevés à tout moment.
  - Désavantages
    - Les observateurs ne se connaissant pas, ils ne peuvent connaître le coût final d'une modification du sujet (cascade de mises à jour).
    - Le simple protocole de mise à jour requiert des observateurs de déduire ce qui a changé.

Observateur (8)

- Considérations d'implémentation
  - Stockage des observateurs dans le sujet par une liste ou un tableau. Coûteux en mémoire s'il existe beaucoup de sujets et peu d'observateurs. Il peut être utile d'utiliser un tableau associatif sujet/observateur.
  - Si un observateur veut observer plusieurs sujets, `update()` doit être étendue pour permettre à l'observateur de savoir quel sujet a été modifié.
  - Qui déclenche la mise à jour (`notify()`)?
    - Le sujet, dès que sont état est modifié. Inefficace en cas de modifications consécutives d'un même sujet.
    - Les clients utilisant le sujet une fois toutes les modifications effectuées. Donne aux clients une responsabilité supplémentaire.
  - Assurer que le sujet met à jour son état *avant* de notifier les observateurs (p.ex. si l'opération de mise à jour d'une sous classe appelle l'opération héritée d'une super classe).

Observateur (9)

- Considérations d'implémentation (suite)
  - Informations transmises lors d'une notification: modèles *push* et *pull.*
    - *Push:* informations détaillées (le sujet présuppose des besoins).
    - *Pull*: peu d'informations, les observateurs demandent ensuite les détails (peut être inefficace).
  - Extension de l'interface d'un sujet pour permettre aux observateurs de s'abonner uniquement aux événements qui les intéressent.
  - Un observateur peut être le sujet d'un autre observateur.
  - Notification d'un observateur après que plusieurs sujets interdépendants aient été modifiés en utilisant un objet intermédiaire comme *médiateur*.
  - Modèle *Modèle/Vue/Contrôleur* (Smalltalk).
    Modèle: sujet, Vue: observateur, Contrôleur: modification du sujet.

Observateur - Java

- Pour ce modèle Java fournit une classe `Observable` (sujet) à dériver par tout sujet concret et une interface `Observer` (observateur) à implémenter.
- Classe `Observable`
  - `public void addObserver(Observer o)`
    `public void deleteObserver(Observer o)`
    `public void deleteObservers()`
    Gestion des observateurs.
  - `protected void setChanged()`
    `protected void clearChanged()`
    `public boolean hasChanged()`
    Gestion du statut de modification de cet objet.
  - `public void notifyObservers(Object arg)`
    Notifie les observateurs si `hasChanged()` est vrai en invoquant leur méthode `update()` et invoque ensuite `clearChanged()`.
- Interface Observer
  - `void update(Observable o, Object arg)`

Exemple

```
class Product extends Observable
{
  private String name;
  private float price;
  public Product(String name, float price) {
    this.name = name;
    this.price = price;
    System.out.println("Product created: " + name + " at " + price);
  }
  private void notify(Object arg) {
    setChanged();
    notifyObservers(arg);
  }
  public void setName(String name) {
    this.name = name;
    notify(name);
  }
  public void setPrice(float price) {
    this.price = price;
    notify(price);
  }
}
```


Exemple (2)

```
class NameObserver implements Observer
{
  public void update(Observable obj, Object arg) {
    if (arg instanceof String) {
      String name = (String)arg;
      System.out.println(
        "NameObserver: Name changed to " + name);
    }
  }
}
```

```
class PriceObserver implements Observer
{
  public void update(Observable obj, Object arg) {
    if (arg instanceof Float) {
      Float price = (Float) arg;
      System.out.println(
        "PriceObserver: Price changed to " + price);
    }
  }
}
```

Exemple (3)

- Programme de test

```
// Create the Subject and Observers.
Product s = new Product("Coca", 1.29f);
s.addObserver(new NameObserver());
s.addObserver(new PriceObserver());
// Make changes to the Subject.
s.setName("Pepsi");
s.setPrice(1.57f);
```

- Résultat

```
Product created: Coca at 1.29
NameObserver: Name changed to Pepsi
PriceObserver: Price changed to 1.57
```


Observateur – Java (2)

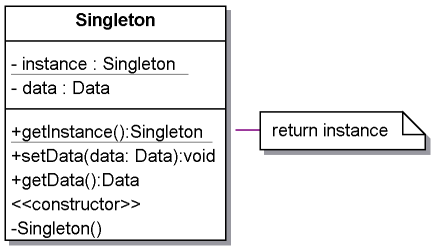
- En Java 1.1 la gestion des événements est basée sur le modèle observateur.
  - Objet générant des événements (souris, clavier, boutons…): *event source*
  - Objet désirant être notifié: *event listener (écouteur)*
  - Pour qu'un objet *écouteur d'événements* soit notifié d'un événement, il doit au préalable s'abonner auprès d'une *source d'évenements*.
- Comparaison avec le modèle Observateur:
  - Source: sujet concret
  - Listener: observateur concret
- Un écouteur d'événements doit implémenter une interface qui définit la méthode à invoquer par la source d'événements quand l'événement survient.
- Contrairement au modèle Observateur qui ne définit qu'une seule interface `Observer`, il existe différentes interfaces spécialisées (`MouseListener`, `ActionListener`, …).

Singleton

- [Objet – Créateur]
- Intention
  - Garantit qu'une classe n'a qu'une seule instance et fournit un accès global à cette instance.
- Motivation
  - Il est parfois important de n'avoir qu'une seule instance d'une classe (serveur d'impression, gestionnaire de fenêtres…).
  - Une variable globale permet d'accéder à un objet mais n'empêche pas les instanciations multiples de la classe.
  - Confier la responsabilité de la création de l'instance et son unicité à la classe elle-même.
- Collaborations
  - Les clients ne peuvent accéder à l'instance d'un Singleton qu'au moyen d'une méthode `getInstance()`.

Singleton (2)

- Structure



- Conséquences
  - Accès contrôlé à une unique instance (par la méthode `getInstance()`).
  - Réduction de l'espace des noms (pas de variable globale).
  - La classe Singleton peut être étendue.
  - Peut être modifiée pour autoriser un nombre contraint d'instances (p. ex. pool de connexions, metaclasses).

Singleton (3)

- Implémentation

```
class Singleton
{
  private static Singleton instance;
  private int data;                          // état du singleton
  private Singleton() { }                    // constructeur privé
  public int getData() {
    return data;
  }
  public void setData(int data) {
    this.data = data;
 }
 public static Singleton getInstance() {
  if (instance == null)
    instance = new Singleton();              // instanciation retardée
  return instance;
 }
}
```


Singleton (4)

- Pourquoi instancier un singleton et ne pas simplement utiliser des attributs statiques sur la classe et des méthodes statiques pour les manipuler?
  - Des informations disponibles lors de l'exécution peuvent être requises avant de pouvoir utiliser les données du singleton .
  - Les méthodes statiques ne peuvent pas être utilisées pour implémenter une interface et ne peuvent être redéfinies (pas de liaison dynamique).
  - Les opérations sur le singleton doivent être préfixées par le nom de la classe.
- Le modèle du singleton permet d'encapsuler et de contrôler le processus de création en assurant que certains prérequis sont satisfaits ou en utilisant l'instanciation retardée.

Singleton et héritage

- La classe Singleton possède des sous classes.
- Si chaque sous classe requiert un singleton propre, simplement les définir en tant que singletons.
- L'application ne peut posséder qu'un seul singleton d'une des sous classes.
- Instanciation dynamique de la sous classe choisie:
  - Définir une méthode `getInstance()` dans chaque sous classe pour effectuer l'instanciation du singleton?
    ➔ Code dupliqué entre les différentes sous classes.
  - Par la classe `Singleton` en fonction du nom de la classe choisie.
    - Pas d'invocation « *en dur* » des constructeurs des sous classes.
      La classe `Singleton` connaîtrait ainsi ses sous classes et devrait être modifiée lorsqu'une nouvelle est créée ➔ violation de l'OCP.
    - Les constructeurs des sous classes ne sont plus privés.
      ➔ Définition d'un package pour éviter la création d'autres instances.

Singleton et héritage (2)

```
package singleton;
public class Singleton                    // eventuellement abstraite
{
  private static Singleton instance;
  // constructeur par défaut, visibilité package
  public static Singleton getInstance()
  {
    if (instance == null)
      throw new RuntimeException("No registered singleton");
    return instance;
  }
  public static void register(String className) throws Exception
  {
    if (Singleton.instance != null)
      throw new RuntimeException("Singleton alredy registered");
    Class c = Class.forName(className);
    if (!Singleton.class.isAssignableFrom(c))
      throw new RuntimeException("Invalid Singleton subclass");

    Singleton.instance = (Singleton) c.newInstance();
  }
}
```

Singleton et héritage (3)

- Sous-classe

```
package singleton;

public class ChildSingleton extends Singleton
{
  // attributs

  ChildSingleton() { /* … */ }
}
```

- Test

```
Singleton.register("singleton.ChildSingleton");
System.out.println(
  Singleton.getInstance().getClass().getName());
```

Résultat: `singleton.ChildSingleton`

Singleton et héritage (3)

-
```
package Singleton;

public abstract class Singleton
{
  private static Singleton instance;

  // constructeur par défaut, visibilité package

  public static Singleton getInstance()
  {
    if (instance == null)
      throw new RuntimeException("No registered singleton");
    return instance;
  }

  public static <T extends Singleton> void register(Class<T> c)
    throws Exception
  {
    if (Singleton.instance != null)
      throw new RuntimeException("Singleton alredy registered");

    Singleton.instance = c.newInstance();
  }
}
```

-
```
Singleton.register(ChildSingleton.class);
System.out.println(Singleton.getInstance().getClass().getName());
```

Singleton et concurrence

- Si un singleton est utilisé par plus d'un processus, il est potentiellement possible qu'il soit instancié plus d'une fois.

```
public static Singleton getInstance()
{
  if (instance == null)
    instance = new Singleton();
  return instance;
}
```

Deux processus peuvent exécuter en même temps dans ce bloc

- Solutions:
  - Ne pas utiliser l'instanciation retardée et instancier directement le singleton. Pas toujours possible, ni optimal.

    ```
    private static instance = new Singleton();
    ```
  - Synchroniser la méthode `getInstance()`. Ce verrou a un coût.
  - Profiter du fait que la machine virtuelle de Java ne charge une classe que lorsqu'elle est référencée pour la première fois.

    ➔ Instancier le singleton depuis une classe interne statique.

Singleton et concurrence (2)

```
class Singleton
{
  private static class Instance
  {
    static final Singleton instance = new Singleton();
  }
  private Singleton()
  {
    System.out.println("-- Singleton()");
  }
  public static Singleton getInstance()
  {
    return Instance.instance;
  }
}
```

```
System.out.println("-- main");
Singleton s = Singleton.getInstance();
```

Résultat:

```
-- main
-- Singleton()
```

Fabriques

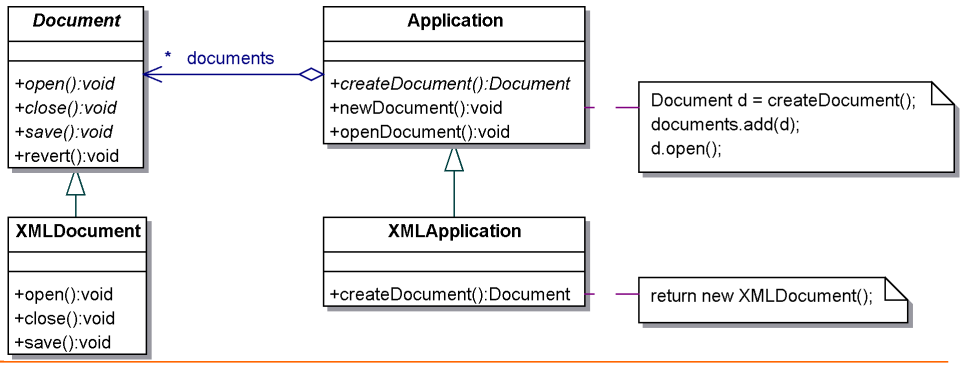
- La « fabrication » et la « fabrique abstraite » sont des modèles créateurs.
- Les modèles créateurs permettent d'abstraire le processus d'instanciation.
  - Les modèles de classe (fabrication) se basent sur l'utilisation de l'héritage pour déterminer quel objet instancier.
  - Les modèles d'objet (fabrique abstraite) se basent sur la délégation de l'instanciation à un autre objet.
- Ces modèles permettent de créer des objets sans devoir utiliser explicitement l'opérateur `new`.
- Ils permettent de définir des méthodes instanciant différents objets et qui peuvent être plus tard redéfinies pour instancier de nouveaux types d'objets, sans que le code des méthodes existantes soit modifié.

Fabrication (*Factory method*)

- [Classe – Créateur]
- Intention
  - Définit une interface pour la création d'un objet, mais en déléguant à ses sous classes le choix des classes à instancier.
- Alias
  - Constructeur virtuel.
- Motivation
  - Soit un framework conçu pour des applications susceptibles de gérer plusieurs types de documents.
  - Deux classes abstraites: `Application` et `Document`.
  - Implémentation d'une application spécifique en dérivant ces classes.
  - La classe `Application` doit pouvoir créer et ouvrir des documents en réponse au événements « New » et « Open » d'un menu.
  - La classe `Application` ne connaît pas le type de document à instancier.

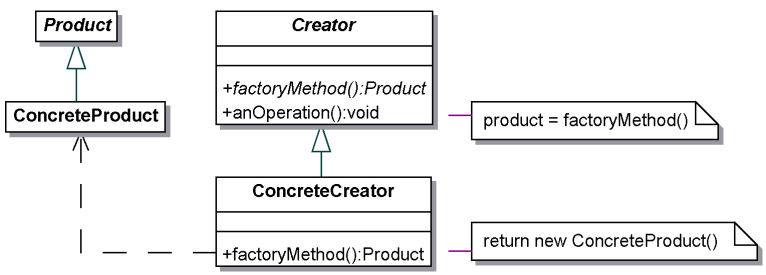
Fabrication (2)

- Motivation (suite)
  - ➔ Définition d'une méthode abstraite `createDocument()` dans la classe `Application` et redéfinie dans les sous classes *fabriquant,* par liaison dynamique, les documents du bon type.



Fabrication (3)

- La fabrication doit être utilisée lorsque:
  - Une classe ne peut prévoir la classe des objets qu'elle doit instancier.
  - Une classe attend que ses sous classes spécifient les objets à créer.
- Structure



  - `Product` définit l'interface des objets que la fabrication `factoryMethod`, déclarée dans `Creator` et redéfinie dans `ConcreteCreator` crée.

Fabrication (4)

- Collaborations
  - La classe `Creator` confie à ses sous classes l'implémentation de la fabrication de sorte à ce qu'elle puisse rendre l'instance appropriée d'une des classes `ConcreteProduct`.
- Conséquences
  - Avantages
    - La fabrication dispense d'avoir à incorporer à son code des classes spécifiques de l'application.
    - Le code ne concerne que l'interface `Product` et peut donc fonctionner avec toute classe `ConcreteProduct` définie par l'utilisateur.
  - Inconvénient
    - Les clients peuvent avoir à hériter de la classe `Creator` juste pour pouvoir créer un produit particulier.

Fabrication (5)

- Remarques
  - La classe `Creator` est écrite sans connaître quelle classe `ConcreteProduct` sera instanciée.
  - La classe `ConcreteProduct` instanciée dépend de la classe `ConcreteCreator` instanciée et utilisée dans l'application.
- Variantes d'implémentation
  - La classe `Creator` peut être abstraite ou concrète: sa méthode `factoryMethod()` peut pourvoir une implémentation par défaut.
  - Lorsqu'une `factoryMethod()` veut créer des produits de type différent elle peut accepter un paramètre pour déterminer (éviter le `if`/`else` si possible; violation potentielle de l'OCP…) la classe concrète à instancier.

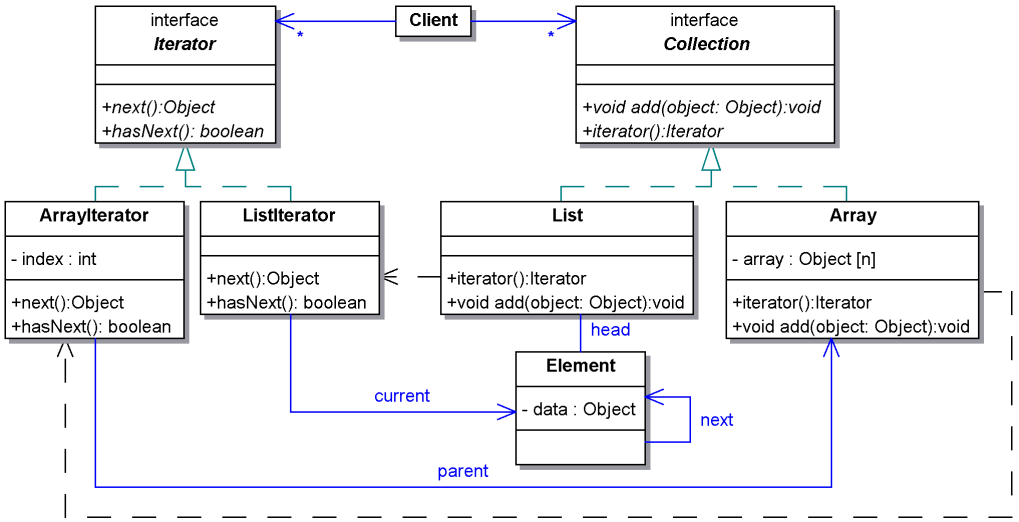
Fabrication (6)

- Les fabrications peuvent être invoquées par des clients et pas seulement par les classes `Creator`.
- Utile dans le cas d'hiérarchies parallèles où une fonctionnalité d'une classe d'une hiérarchie est déléguée à sa contrepartie dans l'autre hiérarchie.
- Exemple:
  - L'obtention d'un itérateur sur un objet d'une classe (ou interface) `Collection` est réalisable par une fabrication `Iterator iterator()`.
  - Plutôt que de tester la collection pour créer le bon itérateur,

```
public static Iterator iterator(Collection c)
{
  if (c instanceof List) return new ListIterator(c);
  if (c instanceof Array) return new ArrayIterator(c);
  return null;
}
```

  déléguer à chaque sous classe de `Collection` la création de l'itérateur.

Fabrication (7)



Exercice

- Implémenter le diagramme de classes précédent en utilisant la généricité et les classes internes.
- Programme de test:

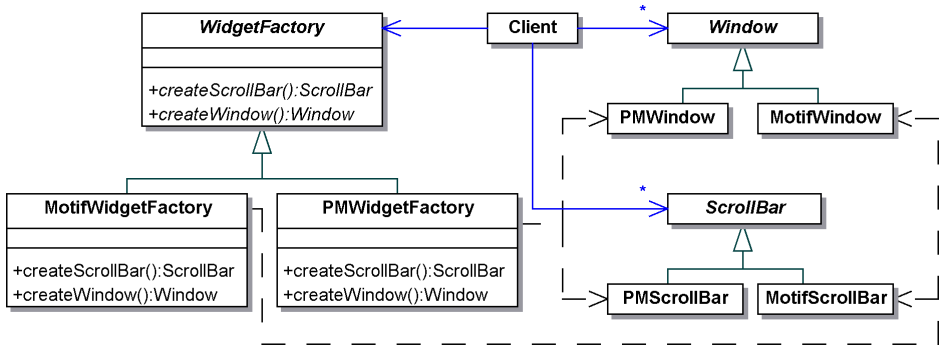
```
List<Integer> l = new List<Integer>();
l.add(1); l.add(2); l.add(3);

Array<String> a = new Array<String>(3);
a.add("one"); a.add("two"); a.add("three");

for (Collection<?> collection : new Collection<?>[] { l, a })
{
  Iterator<?> it = collection.iterator(); // fabrication
  String s = "< ";
  while (it.hasNext())
    s += it.next() + " ";

  System.out.println(s + ">");
}
```


Fabrique abstraite (*Abstract factory*)

- [Objet – Créateur]
- Intention
  - Fournit une interface pour la création de familles d'objets apparentés ou interdépendants sans avoir à spécifier leurs classes concrètes.
  - La fabrique abstraite est similaire à la fabrication.
    - La fabrique abstraite permet de créer des familles d'objets.
    - La fabrique abstraite permet à une classe de déléguer (par composition) au travers d'un objet l'instanciation désirée.
    - La fabrication se base sur l'héritage pour qu'une sous-classe instancie l'objet désiré.
  - Dans la fabrique abstraite l'objet délégué utilise souvent des méthodes de fabrication pour effectuer les instanciations.

Fabrique abstraite (2)

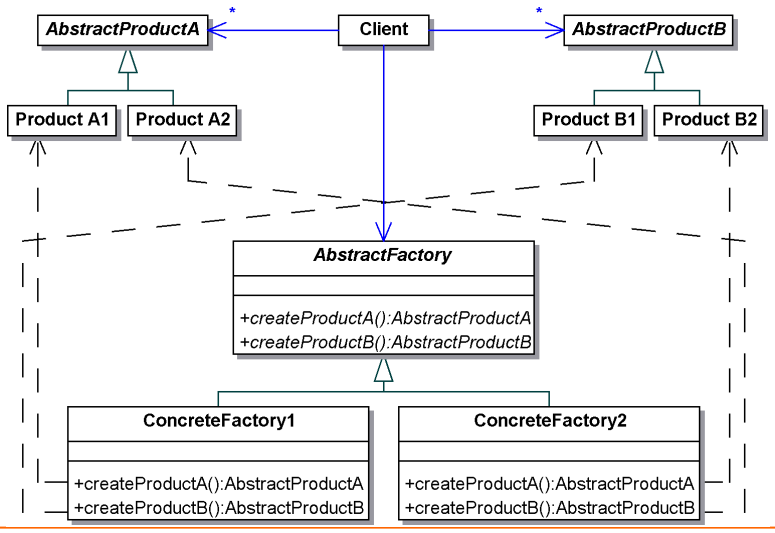
- Motivation
  - Une boite à outils d'interfaces utilisateurs gérant plusieurs « look-and-feel » (décorateurs) standards tels que Motif et PresentationManager.
  - Eviter que les classes *Widget* à instancier soient « codées en dur ».



Fabrique abstraite (3)

- L'utilisation du modèle fabrique abstraite est recommandée lorsque:
  - Un système doit être indépendant de la façon dont ses produits sont créés, combinés et représentés.
  - Une classe ne peut prévoir le type exact des instances qu'elle doit créer.
  - Un système doit utiliser une famille de produits, parmi plusieurs possibles.
  - Les produits d'une famille sont conçus pour être utilisés ensemble.
  - Définir des ensembles de produits sans en révéler les classes exactes.
- Constituants
  - Un client n'utilisera que les interfaces déclarées par la classes `AbstractFactory` et les classes `AbstractProduct`.
  - La classe `AbstractFactory` définit des méthodes abstraites, implémentées dans ses sous classes `ConcreteFactory`, permettant de créer des objets des classes `AbstractProduct`.
  - Les sous classes `ConcreteProduct` des classes `AbstractProduct` représentent les produits effectivement créés.

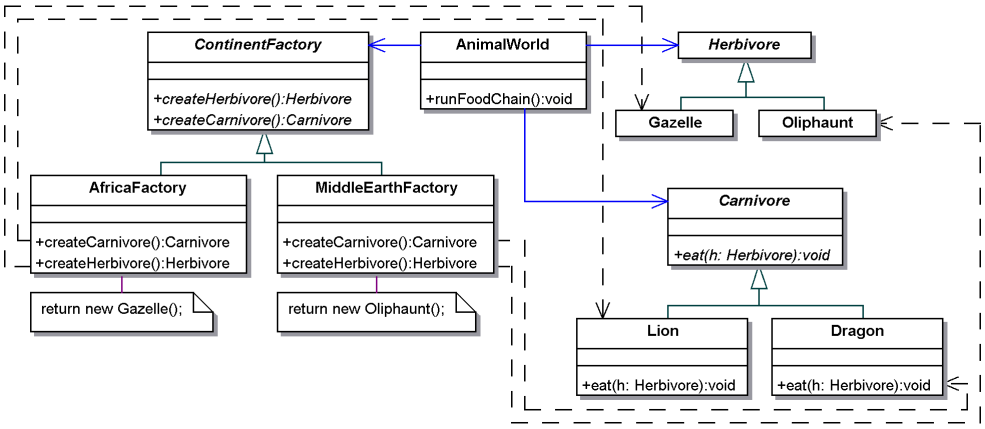
Fabrique abstraite (4)



Fabrique abstraite (5)

- Collaborations
  - La fabrique abstraite délègue la création des objets à une sous classe.
  - Habituellement une seule instance d'une fabrique concrète est créée.
  - Elle créé des objets produits répondant à une implémentation donnée.
  - Pour créer des produits différents, une autre fabrique doit être instanciée.
- Conséquences
  - Avantages
    - Isole les clients des classes d'implémentation concrètes. Les clients manipulent les produits au travers de leurs interfaces abstraites.
    - Facilite la substitution de familles de produits. La fabrique concrète instanciée peut être remplacée par une autre (définie une seule fois).
    - Assure la cohérence des objets produits (même famille).
  - Désavantage
    - Introduire de nouveaux types de produits requiert de définir de nouvelles méthodes dans la fabrique abstraite et ses sous classes.

Exemple: Animal World



Exemple: Animal World (2)

```
class AnimalWorld
{
  Herbivore herbivore;
  Carnivore carnivore;
  AnimalWorld(ContinentFactory cf) {
    herbivore = cf.createHerbivore();
    carnivore = cf.createCarnivore();
  }
  void runFoodChain() {
    carnivore.eat(herbivore);
  }
  public static void main(String … args)
  {
    AnimalWorld
      aw1 = new AnimalWorld(new MiddleEarthFactory()),
      aw2 = new AnimalWorld(new AfricaFactory());
    aw1.runFoodChain(); // A dragon eats an oliphaunt
    aw2.runFoodChain(); // A lion eats a gazelle
  }
}
```

Fabrique abstraite (6)

- Considérations d'implémentation
  - Une application ne nécessite qu'une seule fabrique concrète par famille de produit. Celle-ci est généralement implémentée en tant que Singleton.
  - Définition de nouveaux produits dans une fabrique abstraite existante:
    - P. ex., ajout d'une hiérarchie d'omnivores dans l'exemple précédent.
    - Ajout de nouvelles méthodes de création (`createOmnivore`) dans la fabrique abstraite et les fabriques concrètes, ou
    - Définit qu'une seule méthode `createProduct` rendant un `Object` et acceptant en paramètre identifiant le type de produit à créer.
      - Plus souple: pas de modification des interfaces existantes.
      - Moins sûr: les objets créés sont récupérés en tant qu'`Object`.
      - Peu élégant: détermination du produit a créer par `if/else`.
  - Une fabrique abstraite peut être composée d'autres fabriques abstraites (p.ex. `HerbivoreFactory` et `CarnivoreFactory`) et même ne pas requérir de sous classes (fabriques a utiliser fournies au constructeur).

Exemple: Java AWT

- L'AWT (*Abstract Windows Toolkit*) utilise une fabrique abstraite pour générer tous les composants correspondants à la plateforme utilisée.
- Par exemple, dans la classe `List` on trouve:

```
peer = getToolkit().createList(this);
```

- La méthode `getToolkit()`, héritée depuis la classe `Component`, rend une référence sur la fabrique utilisée pour créer les composants graphiques.

```
public Toolkit getToolkit() {
  ComponentPeer peer = this.peer;
  if ((peer != null) && !(peer instanceof LightweightPeer)) {
    return peer.getToolkit();
  }
  Container parent = this.parent; // Container's toolkit
  if (parent != null) {
    return parent.getToolkit();
  }
  return Toolkit.getDefaultToolkit();
}
```

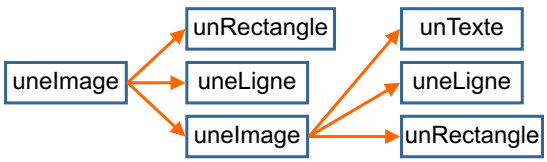
Exemple: Java AWT (2)

- Et dans la classe `Toolkit`,
  - `private static Toolkit toolkit; // singleton`
  - `public static synchronized Toolkit getDefaultToolkit()`

```
{
  if (toolkit == null) {
    /* … */
    String nm = null, defaultToolkit;
    Class cls = null;
    if (System.getProperty("os.name").equals("Linux"))
      defaultToolkit = "sun.awt.X11.XToolkit";
    else
      defaultToolkit = "sun.awt.motif.MToolkit";
    nm = System.getProperty("awt.toolkit", defaultToolkit);
    /* … */
    cls = Class.forName(nm);
    /* … */
    toolkit = (Toolkit) cls.newInstance();
    /* … */
  }
  return toolkit;
}
```

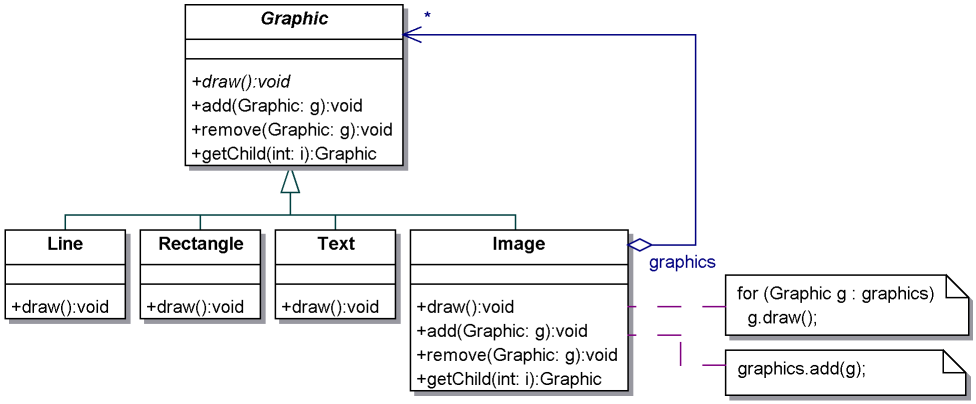

Composite

- [Objet – Structurel]
- Intention
  - Compose les objets en structures arborescentes pour représenter des hiérarchies composant/composé.
  - Permet au client de traiter de la même manière les objets individuels et ceux qui sont des combinaisons d'objets (composition récursive).
- Motivation
  - Dans un éditeur de dessin, manipulation d'objets graphiques qui peuvent être simple ou complexes sans qu'il ne soit nécessaire de les différencier.



Composite (2)

- Motivation (suite)



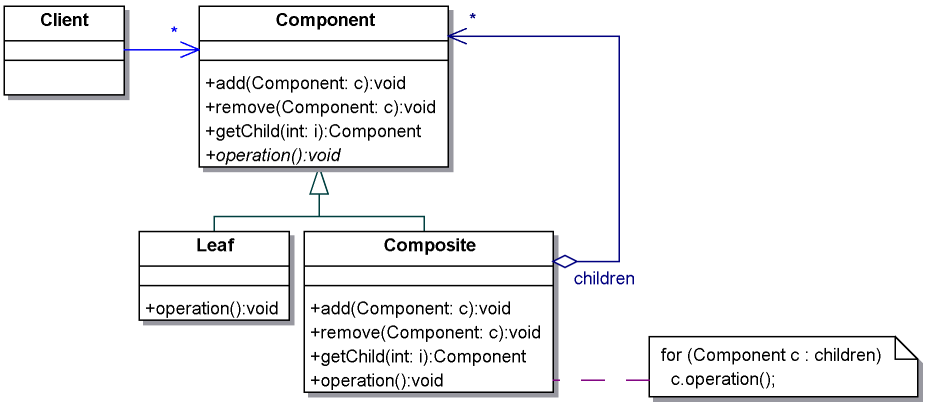
  - La classe abstraite `Graphic` représente à la fois des objets graphiques simples et des agrégats d'objets graphiques (`Image`).

Composite (3)

- L'utilisation du modèle Composite est recommandée:
  - Pour représenter des hiérarchies de l'individu à l'ensemble.
  - Pour que le client n'ait pas considérer les différences entre combinaison d'objets et objets individuels; ils seront traités de manière uniforme.
- Constituants
  - `Component` (`Graphic`) déclare l'interface des objets entrant dans la composition et en définit le comportement par défaut.
    - Déclare une interface pour accéder à ses enfants et les gérer.
    - Optionnellement, implémente un accès au composite parent (`Image`).
  - `Leaf` (`Line`, `Rectangle`, `Text`…) représente des objets sans enfants.
  - `Composite` (`Image`) stocke les enfants et implémente les opérations liées aux enfants déclarées dans `Component`.

Composite (4)

- Structure

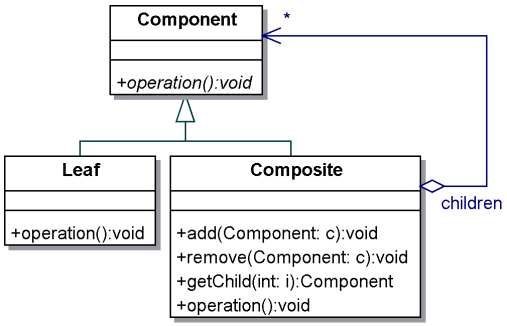


Composite (5)

- Collaborations
  - Les clients utilisent l'interface de `Component` pour manipuler les objets.
  - Si l'objet est une feuille la requête est traitée directement.
  - Si l'objet est un composite, la requête est généralement transférée à ses composants (avec éventuellement post- ou pré- traitement).
- Conséquences
  - Avantages
    - Permet la définition de compositions récursives d'objets.
    - Rend aisée l'adjonction de nouveaux composants (nouvelles feuilles).
    - Simplifie les clients qui n'ont pas besoin de connaître s'ils utilisent un objet simple ou un objet composé.
  - Inconvénient
    - De par sa conception générique, rend difficile l'expression de contraintes sur la composition d'un composite.

Composite – Implémentation

- Définition d'opérations dans `Component` non forcément utiles et significatives pour toutes les sous-classes.
  - L'accès aux enfants peut se justifier par le fait qu'un objet feuille n'en aura jamais (rendre `null`).
  - La définition des opérations de gestion des enfants (ajout et suppression) est un compromis entre sécurité et transparence:
    - Définition dans `Component`: transparent mais peu sûr (un client peut tenter d'ajouter un objet à une feuille).
    - Définition seulement dans `Composite`: sûr mais interfaces différentes.



Composite - Implémentation (2)

- Référence explicite au parent, stockée dans `Component`.
  - Facilite la *remontée* d'une structure composite.
  - Attention à respecter le principe que tous les enfants d'un composite possèdent ce composite comme parent: ne modifier le parent d'un composant que lors de son ajout ou retrait d'un composite.
- Il peut être utile de partager les composants (p.ex. pour limiter l'utilisation de la mémoire). Difficile si un composant possède un seul parent…
- Ordonnancement des enfants (p.ex. arrière-/avant- plan dans une application graphique).
- Pour les langages sans ramasse-miette la suppression des composants s'effectuera de préférence depuis l'objet `Composite` quand il est détruit.
- Structure de données pour stocker les enfants: listes, arbres, tableaux, tables de hachage ou attributs références *ad hoc*.

Exemple

```
abstract class Component
{
  private String name;
  public Component(String name) {
    this.name = name;
  }
  protected void display(int depth, char start) {
    String s = "";
    for (int i = 0; i < depth; i++) s += ' ';
    System.out.println(s + start + "-- " +
                       getClass().getName() + " " + name);
  }
  abstract public void display(int depth);
}
```

```
class File extends Component
{
  public File(String name)  {
    super(name);
  }
  public void display(int depth)  {
    super.display(depth, 'o');
  }
}
```


Exemple (2)

```
class Folder extends Component
{
  private LinkedList<Component> children;

  public Folder(String name)  {
    super(name);
    children = new LinkedList<Component>();
  }

  public void add(Component component) {
    children.add(component);
  }

  public void remove(Component component) {
    children.remove(component);
  }

  public void display(int depth) {
    super.display(depth, '+');

    for(Component component : children)
      component.display(depth + 2);
  }
}
```

Exemple (3)

- Programme de test

```
Folder root = new Folder("Root");
root.add(new File("A"));
root.add(new File("B"));
Folder x = new Folder("X");
x.add(new File("XA"));
x.add(new File("XB"));
root.add(x);
root.add(new File("C"));
root.display(0);
```

- Résultat

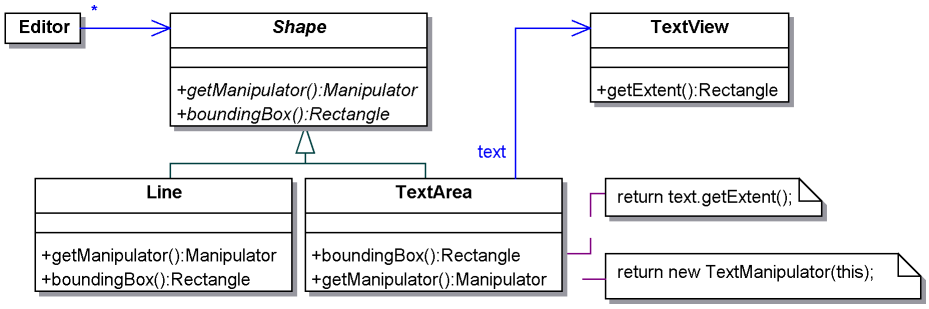
```
+-- Folder Root
  o-- File A
  o-- File B
  +-- Folder X
    o-- File XA
    o-- File XB
  o-- File C
```


Adaptateur

- [Classe, Objet – Structurel]
- Intention
  - Convertit l'interface d'une classe en celle attendue par un client. Permet la collaboration de classes possédant des interfaces incompatibles.
- Motivation
  - Parfois, une librairie de classes n'est pas réutilisable telle quelle car son interface n'est pas compatible avec l'interface requise par une application.
  - La modification de la librairie n'est pas toujours possible (accès au code source) ou souhaitable (la rendrait spécifique à un domaine d'application).
  - ➔ Définir des classes *adaptant* les classes de la librairie.
- Exemple:
  - Réutilisation d'une classe `TextView` dans une hiérarchie de formes.
  - `TextView` définit une méthode `GetExtent` au lieu de `BoundingBox` et ne fournit pas de manipulateur pour un déplacement interactif des objets.

Adaptateur (2)

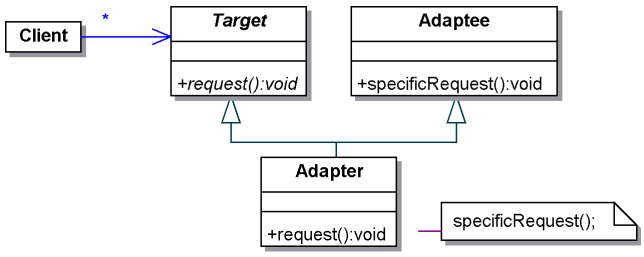
- Exemple (suite)



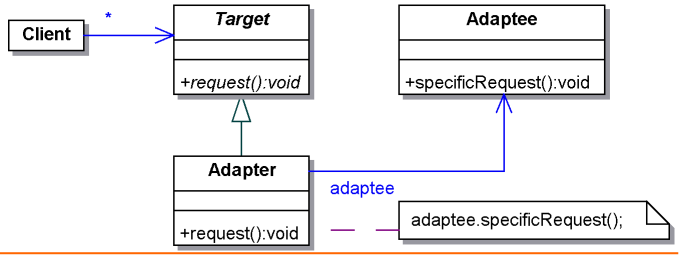
- Indications d'utilisation
  - Utilisation d'une classe existante mais d'interface incompatible.
  - Création d'une classe réutilisable destinée à collaborer avec des classes ne possédant pas nécessairement des interfaces compatibles.

Adaptateur (3)

- Structure d'un adaptateur de classe
  - Par héritage multiple.



- Structure d'un adaptateur d'objet
  - Par composition.



Adaptateur (4)

- Conséquences
  - Contrairement à un adaptateur d'objet, un adaptateur de classe ne permet pas d'adapter une classe *et* toutes ses sous-classes (héritage).
  - L'adaptation peut aller de la simple conversion d'interface jusqu'à la définition d'un ensemble entièrement différent d'opérations.
  - Un *adaptateur bidirectionnel* se conforme non seulement à l'interface de `Target` mais également à celle de `Adaptee`, afin d'être complètement transparent (un objet `Adapter` est utilisable dans les deux contextes).
- Considération d'implémentation
  - En C++ un adaptateur de classe (non bidirectionnel) est défini en héritant en mode `public` de `Target` et en mode `private` de `Adaptee`.
  - En Java un adaptateur de classe peut être défini au moyen d'une interface `Targettable` (déclarant les opérations de `Target`) et une sous-classe d'`Adaptee` implémentant `Targettable`.

Exemple

```
abstract class Peg
{
  private static int count;
  private int index = count++;
  abstract public void insert(String msg);
  public String prompt()  {
    return "Peg #" + index + "> ";
  }
}
```

```
class SquarePeg extends Peg
{
  public void insert(String str) {
    System.out.println(prompt() + "SquarePeg.insert(): " + str);
  }
}
```

```
class RoundPeg
{
  public void insertIntoHole(String msg) {
    System.out.println("RoundPeg.insertIntoHole(): " + msg);
  }
}
```

Exemple (2)

```
class RoundPegAdapter extends Peg
{
  private RoundPeg roundPeg;
  public RoundPegAdapter(RoundPeg peg) {
    this.roundPeg = peg;
  }
  public void insert(String str) {
    System.out.print(prompt());
    roundPeg.insertIntoHole(str);
  }
}
```

- Programme de test

```
Peg squarePeg = new SquarePeg();
squarePeg.insert("Inserting square peg...");
RoundPeg roundPeg = new RoundPeg();
Peg adapter = new RoundPegAdapter(roundPeg);
adapter.insert("Inserting round peg...");
```

- Résultat

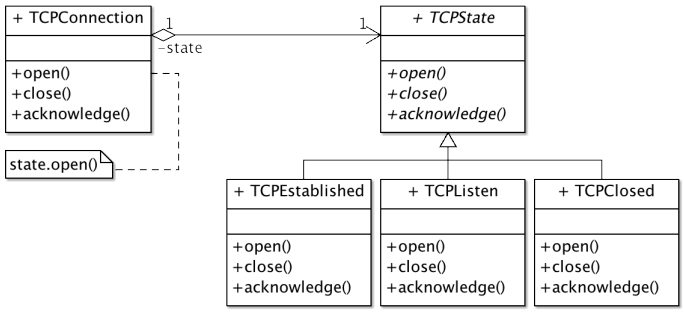
```
Peg #0> SquarePeg.insert(): Inserting square peg...
Peg #1> RoundPeg.insertIntoHole(): Inserting round peg...
```


Etat

- [Objet – Comportemental]
- Intention
  - Permet à un objet de modifier son comportement, quand son état interne change; comme s'il changeait de classe.
- Motivation
  - Soit une classe `TCPConnection` représentant une connexion réseau.
  - Une connexion peut être dans différents états: établie, en écoute, fermée.
  - Quand une requête est reçue, la réponse sera différente selon cet état.
  - ➔Introduire une classe abstraite `TCPState` représentant les états de la connexion du réseau. Elle déclare l'interface commune aux classes représentant les différents états opérationnels.
  - Les sous-classes de `TCPState` implémentent les comportements spécifiques des états.

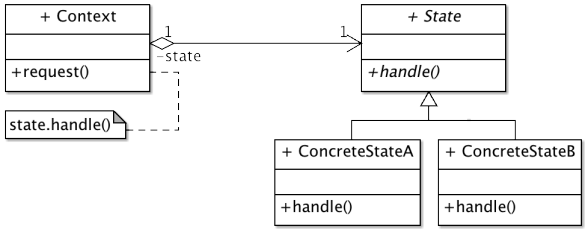
Etat (2)

- Motivation (suite)
  - La classe `TCPConnection` gère un objet etat (une instance d'une sous-classe `TCPState`) et lui délègue ses requêtes (liaison dynamique).
  - Quand la connexion change d'état, l'objet `TCPConnection` change l'objet état qu'il utilise.



Etat (3)

- Le modèle Etat sera utilisé dans l'un des cas suivants:
  - Le comportement d'un objet dépend de son état, et ce changement de comportement doit intervenir dynamiquement, en fonction de cet état.
  - Des opérations différentes d'un objet comportent la même structure conditionelle fonction de l'état de l'objet (souvent un type énuméré). Le modèle Etat permet de placer dans des classes séparées les différentes branches de la condition et de traiter l'état comme un objet.
- Structure



Etat (4)

- Constituants
  - `Context`
    - Définit l'interface intéressant les clients.
    - Gère une instance d'une sous-classe `State` qui définit l'état en cours.
  - `State`
    - Définit une interface qui encapsule le comportement associé avec un état particulier du contexte.
  - Sous-classes `ConcreteState`
    - Chaque classe implémente un état associé avec l'état du contexte.

Etat (5)

- Collaborations
  - Le `Context` délègue les requêtes spécifiques d'état à l'objet `ConcreteState` courant.
  - Un contexte peut passer en paramètre sa propre référence à l'objet `State` courant pour lui permettre d'accéder au contexte si nécessaire.
  - `Context` est l'interface d'intérêt pour les utilisateurs. Ils peuvent définir un contexte au moyen des objets `State` mais, une fois configuré, ils n'ont plus à intéragir directement avec des objets `State`.
  - C'est soit le `Context` soit les sous-classes de `State` qui déterminent l'état succédant à un autre état, et selon quelles modalités.

Etat (6)

- Conséquences
  - Ce modèle isole les comportements spécifiques d'états et fait un partitionnement des différents comportements, état par état.
    - Il place tous les comportements associés à un état dans une classe.
      ➔ De nouveaux états et leurs transitions pourront être aisément être ajoutés par la définition de nouvelles sous-classes.
    - Complexité accrue par le nombre de classes, mais il évite une classe `Context` monolithique dont les méthodes doivent tester l'état courant et dont la maintenance et l'extension seraient compliquées.
    - La logique de transition entre d'états est répartie entre les sous-classes et non pas dans des ifs et autres switchs.
  - Il rend les transitions d'état plus explicites: pas par l'assignation de valeurs d'attributs qui pourraient aboutir à des états incohérents.
  - Les objets `State` peuvent être partagés (s'ils ne possèdent pas d'attributs propres) entre différents contextes .

Etat (7)

- Implémentation
  - Qui définit les transitions d'état ?
    - Si les critères sont prédéterminés, le `Context` peut les effectuer.
    - Plus souple: confier aux sous-classes de `State` la responsabilité de définir quel(s) état(s) leurs succède(nt) et dans quelles conditions.
      ➔ rajouter une méthode à `Context` qui permette à l'objet `State` de définir l'état courant du contexte.
      Inconvénient de la décentralisation des états: chaque sous-classe de `State` doit en connaître au moins une autre, ce qui induit une dépendance de code entre les sous-classes.
  - Utiliser des tables dont les entrées correspondent à des transitions d'état.
    - Remplace le code conditionnel par des explorations de tables et il est plus aisé de changer les critères de transition (données des tables).
    - Moins efficace qu'utiliser la liaison dynamique, plus complexe à comprendre, difficile d'ajouter des actions pour les transitions d'état.

Etat (8)

- Implémentation (suite)
  - Création et destruction d'états
    - Préférer la création d'objets état à la volée lorsque les états requis ne sont pas connus et que le contexte change peu fréquemment d'état.
    - Préférer la pré-création d'états (jamais détruits) lorsque les changements d'état sont fréquents et qu'un ancien état risque d'être réutilisé. Coût: le contexte doit conserver des références sur tous les états pouvant être utilisés.
  - Utilisation de l'*héritage dynamique*
    - Le changement d'état pourrait être effectué en modifiant dynamiquement la classe d'un objet `State` sans devoir le recréer.
      - Impossible dans la plupart des langages OO actuels.
      - Le langage Self (basé sur la délégation des opérations à d'autres objets) permet d'avoir un comportement comparable…

Exemple

```
abstract class State
{
  protected Account account;
  protected double balance;
  protected State(Account account) {
    this.account = account;
  }
  protected State(State oldState) {
    this.account = oldState.account;
    this.balance = oldState.balance;
  }
  double getBalance() { return balance; }
  void deposit(double amount) {
    balance += amount;
    stateChangeCheck();
  }
  public void payInterest() {
    balance += balance * getInterest();
  }
  abstract void withdraw(double amount);
  abstract double getInterest();
  abstract void stateChangeCheck();
}
```

Exemple (2)

```
class RedState extends State
{
  RedState(State oldState) {
    super(oldState);
  }

  void withdraw(double amount) {
    System.out.println("No funds available for withdrawal!");
  }

  double getInterest() { return 0.2; }

  void stateChangeCheck() {
    if (balance >= 0)
      account.setState(new SilverState(this));
  }
}
```


Exemple (3)

```
class SilverState extends State
{
  SilverState(Account account) {
   super(account);
  }
  SilverState(State oldState)  {
    super(oldState);
  }
  void withdraw(double amount) {
    balance -= amount;
    stateChangeCheck();
  }
  double getInterest() { return 0.05; }
  void stateChangeCheck() {
    if (balance < 0)
      account.setState(new RedState(this));
  }
}
```

Exemple (4)

```
class Account
{
  private State state;
  public Account() {
    this.state = new SilverState(this);
  }
  void setState(State newState) {
   this.state = newState;
  }
  private void info(String format, Object ... args) {
    System.out.printf(String.format(format, args) +
                      "Balance: %.2f. Status: %s.\n",
                       state.getBalance(), state.getClass().getName());
  }
  public void deposit(double amount){
    state.deposit(amount);
    info("Deposited: %.2f. ", amount);
  }
```


Exemple (5)

```
  public void withdraw(double amount)  {
    state.withdraw(amount);
    info("Withdrew: %.2f. ", amount);
  }
  public void payInterest()  {
    state.payInterest();
    info("Interest paid. ");
  }
  public static void main(String ... args) {
    Account account = new Account();
    account.deposit(500.0);
    account.withdraw(100.0);
    account.payInterest();
    account.withdraw(500.00);
    account.withdraw(100.0);
    account.payInterest();
    account.deposit(500.0);
  }
}
```

Exemple (6)

- Output

```
Deposited: 500.00. Balance: 500.00. Status: SilverState.
Withdrew: 100.00. Balance: 400.00. Status: SilverState.
Interest paid. Balance: 420.00. Status: SilverState.
Withdrew: 500.00. Balance: -80.00. Status: RedState.
No funds available for withdrawal!
Withdrew: 100.00. Balance: -80.00. Status: RedState.
Interest paid. Balance: -96.00. Status: RedState.
Deposited: 500.00. Balance: 404.00. Status: SilverState.
```