R3.04 Conception cours n°5

I.Borne

- Etude de patrons de conception
  - création : FactoryMethod
  - structurel : Adapter
  - comportement : State, Strategy, Null Object

Les patrons de conception du GoF

purpose / scope	Creational	Structural	Behavioral
class	Factory Method	Adapter (class)	Interpreter Template Method
object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State, Strategy, Visitor

1. Les patrons de création

Un des buts de la conception par objets :

- déléguer des responsabilités entre différents objets
- Encourager l'encapsulation et la délégation.

- On ne peut pas anticiper la classe d'objet à instancier.
- On ne connaît que l'interface ou une classe abstraite.
- On ne connaît pas la sousclasse à instancier

- Une classe peut déléguer la responsabilité à une ou plusieurs sous-classes (aides) afin que la connaissance soit localisée dans ces classes spécifiques.

Patron de création Factory Method

Nom : Factory Method

Classification : Creation – Classe

Alias : constructeur virtuel

Intention :

Fournir une méthode standard pour créer des objets, en dehors d'un constructeur, mais laisser les sous-classes décider de la classe à instancier

Motivation :

Il s'agit de permettre l'adaptation de classes abstraites de framework qui, bien qu'abstraites, ont à créer des objets dans d'autres classes abstraites

Quand utiliser Factory Method ?

- Pour créer un framework extensible : certaines décisions sont laissées pour plus tard, comme le type spécifique d'objet à créer.
- C'est à une sous-classe et non une superclasse de décider quelle sorte d'objet créer.
- Nous savons quand créer un objet mais pas de quel type il s'agit.
- Besoin de plusieurs versions de constructeurs avec la même liste de paramètres (pas possible en Java), possible avec plusieurs Factory Method de noms différents.

Précaution avec Factory Method

- Le fait qu'une méthode crée un nouvel objet ne signifie pas que c'est un exemple du patron Factory Method.
- Le patron requiert qu'une opération qui crée un objet ne permette pas à son client de savoir quelle classe instancier.
- On peut trouver plusieurs classes qui implémentent la même opération et retournent le même type abstrait mais qui, de façon interne, instancie différentes classes qui implémentent ce type

Exemple : application de logistique

Objectif : créer une application de gestion logistique

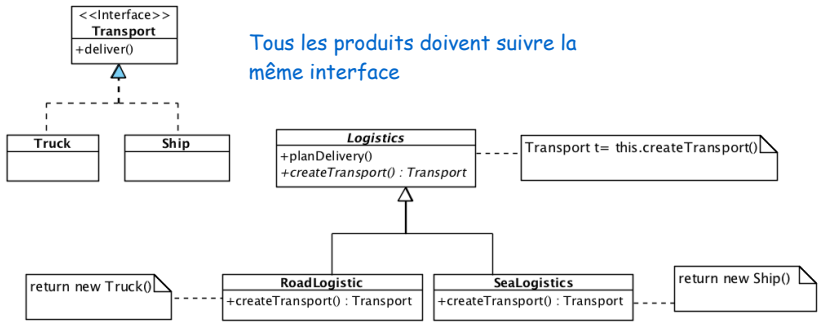
La première version de notre application gère seulement le transport par camions, ainsi tout le code réside dans la classe Truck.

Plus tard notre application a du succès et on nous demande des transports par mer. Il faut donc intégrer cela dans notre application.

Tout le code est couplé à la classe Truck, ajouter Ship requiert d'ajouter une nouveau type de transport. Le code se transforme en des conditionnelles qui selon le type de transport doit changer le comportement de l'application.

Exemple : Logistics avec Factory method

Le patron *factory method* suggère de remplacer les appels de construction des objets avec des appels à une méthode spéciale *factory*



Tous les produits doivent suivre la même interface

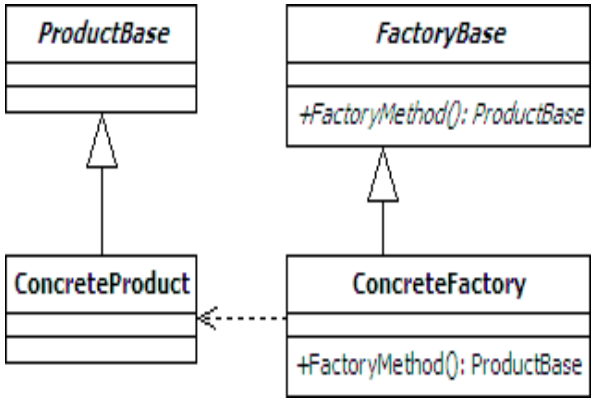
Les sousclasses peuvent modifier la classe des objets retournés par la *factory method*

Contexte (factory method)

- Un type (*créateur*) crée des objets d'une autre type (*produit*)
- Les sous-classes du type créateur doivent créer différentes sortes d'objets de type « *produit* ».
- Les clients ne connaissent pas le type exact des objets « *produit* »

Ce pattern nous enseigne comment fournir une méthode qui peut être surchargée pour créer des objets de types différents.

Solution de Factory Method

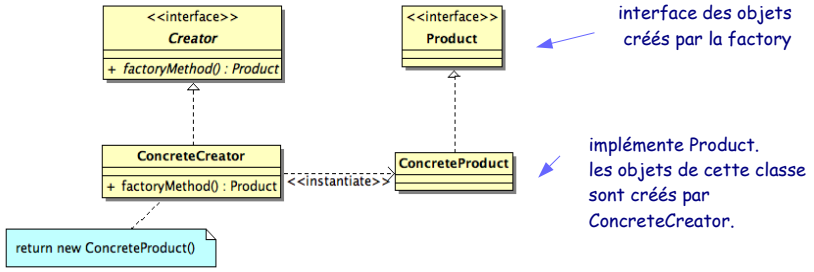


Les classes de Factory Method

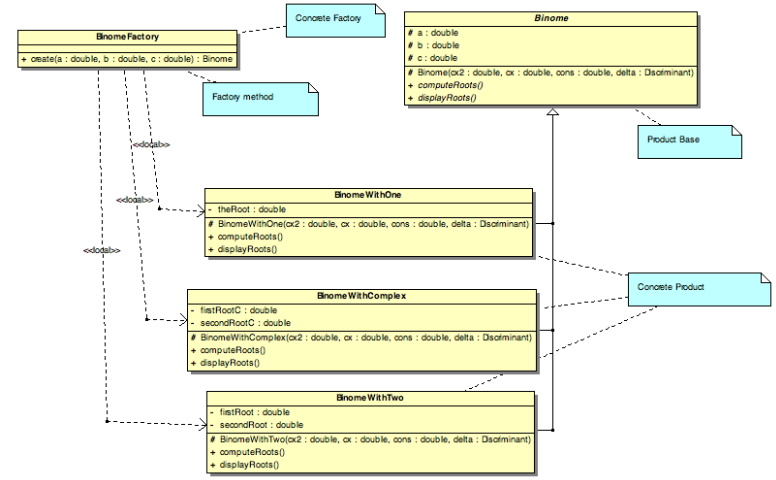
- **FactoryBase** : classe abstraite pour les classes de factory concrètes qui génèreront les nouveaux objets. Cette classe peut aussi n'être qu'une interface contenant la signature de la Factory méthode et dans certaines situations ne pas être abstraite.
- **ConcreteFactory** : hérite de FactoryBase, redéfinit le code de génération de l'objet sauf s'il est déjà implémenté dans la superclasse.
- **ProductBase** : cette classe abstraite est la classe de base pour les types d'objet que la factory peut créer. C'est aussi le type de retour de la *factory method*. Elle peut n'être aussi qu'une interface.
- **ConcreteProduct** : les sousclasses de la classe ProductBase sont définies, chacune contient une fonctionnalité spécifique. Les objets de ces classes sont générés par la *factory method*.

Version avec des interfaces

1. Définir un type créateur qui exprime ce qui est commun à tous les créateurs
2. Définir un type produit qui exprime ce qui est commun à tous les produits
3. Définir une méthode, la *factory method*, dans le type créateur . Elle produit un objet produit.
4. Chaque classe *ConcreteCreator* implémente la *factory method* afin qu'elle retourne un objet de la classe concrète produit.



L'exemple de l'application Binome



Comment reconnaître une factory method

- Il y a plusieurs classes qui implémentent la même opération, retournant le même type abstrait mais instanciant de façon interne différentes classes qui implémentent le type.
- En résumé, les signes à regarder :
  - crée un nouvel objet
  - retourne un type qui est une classe abstraite ou une interface
  - est implémenté par plusieurs classes
- Les méthodes *iterator()* sont de bons exemples du pattern Factory Method. Toutes les collections implémentent cette opération qui crée un objet et retourne une séquence des éléments d'une collection.

La méthode iterator() de l'interface Collection<E>

<E> iterator()
Returns an iterator over the elements in this collection.
There are no guarantees concerning the order in which the elements
are returned (unless this collection is an instance of some class that provides a guarantee).

Specified by:

iterator in interface Iterable<E>

Returns:

an Iterator over the elements in this collectionIterator

Nom dans le pattern	Nom actuel
Creator	Collection
ConcreteCreator	Une sous-classe de Collection
factoryMethod	iterator()
Product	Iterator
ConcreteProduct	Une sous-classe d'Iterator(anonyme)

2. Les patterns structurels

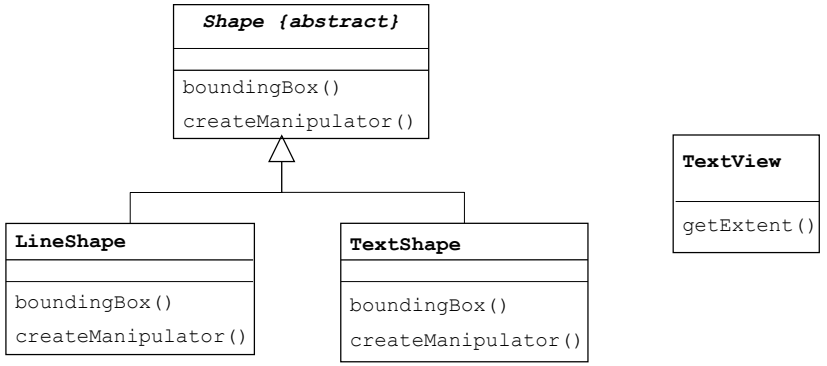
- L'objectif des patterns structurels est de proposer des solutions pour composer des classes et des objets afin d'obtenir des structures plus complexes.
- On distingue :
  - les patterns structurels de classe qui utilisent l'héritage pour composer des interfaces ou des implémentations.
  - les patterns structurels d'objets décrivent comment composer des objets pour réaliser de nouvelles fonctionnalités.

Le patron Adapter

- Intention :
  - Convertir l'interface d'une classe en une autre interface qu'attendent les clients. *Adapter* permet aux classes de travailler ensemble ce qui ne serait pas possible autrement à cause de l'incompatibilité des interfaces.

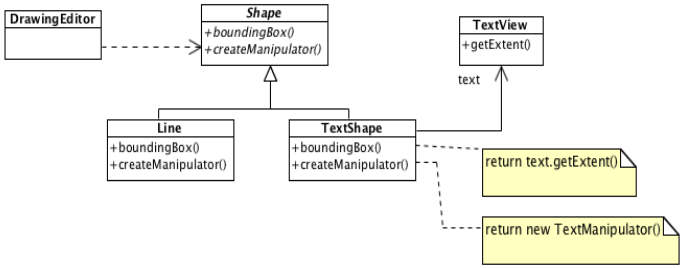
- Description du problème : Cas d'un éditeur de dessin
  - Un objet graphique a une forme éditable et peut se dessiner
  - L 'interface pour les objets graphiques est définie dans une classe abstraite `Shape` et une sous-classe de `Shape` est définie pour chaque type d 'objet.
  - La sous-classe `TextShape` est plus difficile à implémenter. Dans la bibliothèque on a trouvé une classe `TextView` qui sait afficher et éditer du texte et que l'on veut réutiliser.

Situation actuelle



Solution pour Adapter

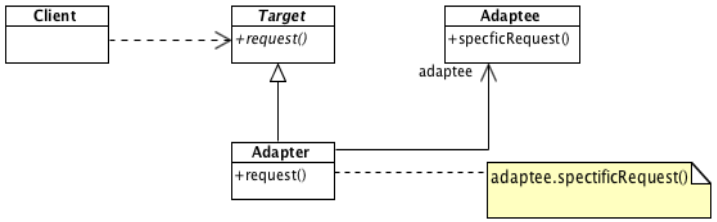
Définir la classe `TextShape` afin qu'elle adapte l'interface de `TextView` à la classe `Shape` :
1) soit en faisant en sorte que `TextShape` hérite de l'interface de `Shape` et de l'implémentation de `TextView`
2) soit en composant une instance de `TextView` avec un `TextShape` et en implémentant `TextShape` en fonction de l'interface de `TextView`



Structure générale de Adapter

Dit comment convertir une interface d'une classe en une autre interface attendue par les clients de cette classe.

Adapter permet à des classes de fonctionner ensemble, ce qu'elles ne pourraient pas faire autrement, parce que leurs interfaces sont incompatibles.



Participants

- *Target (Shape)* : définit l'interface spécifique du domaine que le Client utilise.
- *Client (DrawingEditor)* : collabore avec des objets conformément à l'interface de Target.
- *Adaptee (TextView)* : définit une interface existante qui a besoin d'adaptation.
- *Adapter (TextShape)* : adapte l'interface de Adaptee à l'interface de Target.

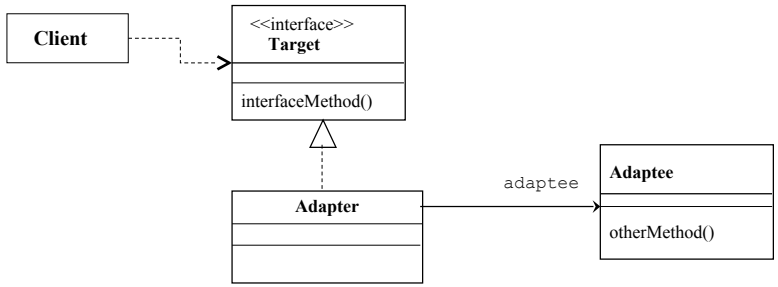
Collaborations

- Les clients envoient des messages à une instance d'Adapter. A son tour, l'adapter envoie des messages à `Adaptee` qui effectue la requête.

Conséquences

- Le client et la classe `Adaptee` restent indépendants l'un de l'autre
- Le patron introduit une indirection supplémentaire qui peut contribuer à une difficulté pour la compréhension du programme

Adapter en Java avec une interface



Héritage et délégation avec Adapter

- Le pattern *Adapter* utilise l'héritage de spécification entre l'interface `Target` et `l'Adapter`.
- L'`Adpater` à son tour délègue à la classe implémentant `Adaptee` la réalisation des opérations déclarées dans l'interface `Target`.
- Cela permet à tout code client de toujours utiliser l'interface Target et d'utiliser des instances de `Adapter` de façon transparente et sans modification du client.
- Par ailleurs, le même Adapter peut être utilisé pour des sous-types de `Adaptee`.
- Le pattern Adapter permet d'encapsuler d'anciens composants qui sont les classes `Adaptee`.

3 - Les patrons comportementaux

- Ces patrons concernent des algorithmes et l'affectation de responsabilités entre objets.
- Ils ne décrivent pas uniquement les patterns d'objet ou de classe mais également les patterns de communication entre eux.
- Ils caractérisent des contrôles de flot compliqués qui sont difficiles à suivre à l'exécution.
- Ils nous permettent de nous concentrer sur la façon dont les objets sont interconnectés.

3.1 Le pattern State
(comportement)

Intention

– Permet à un objet de modifier son comportement quand son état interne change. L'objet semblera changer de classe.

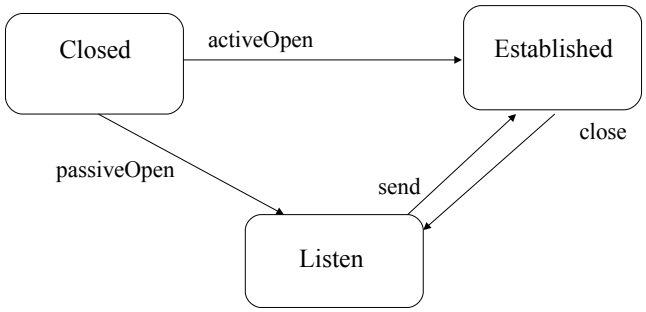
S'utilise quand

– Quand le comportement d'un objet dépend de son état, et qu'il doit changer son comportement à l'exécution en fonction de cet état.

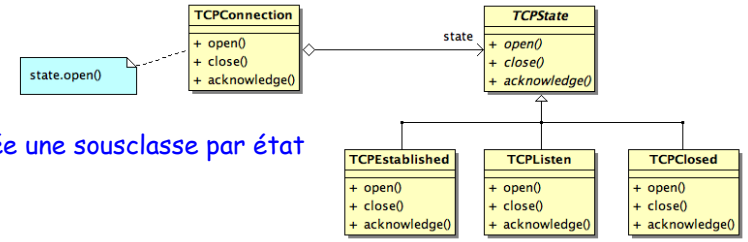
– Quand les opérations ont des instructions conditionnelles multiples qui dépendent de l'état de l'objet. Cet état est habituellement représenté par une ou plusieurs constantes énumérées. Le pattern *State* met chaque branche de la conditionnelle dans une classe séparée.

Exemple

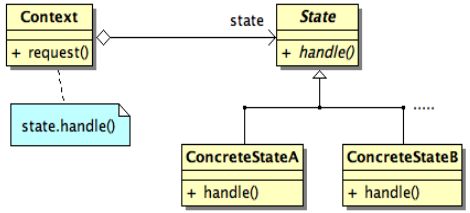
- Une classe nommée *TCPConnection* représente une connexion réseau.
- Une connexion peut être dans un ou plusieurs états : établie, fermée, en écoute, et son comportement dépend de l'état courant.



La solution proposée par le pattern State

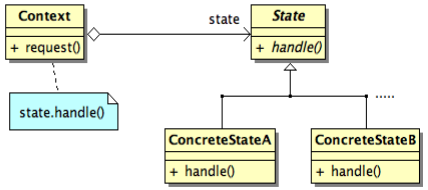


On crée une sousclasse par état



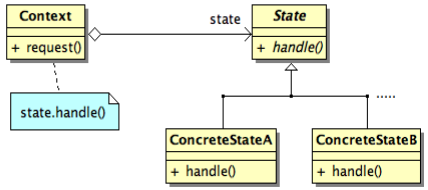
Structure

Participants



- **Context** (TCPConnection)
  - Définit l'interface qui intéresse les clients.
  - Maintient une instance d'une sous-classe `ConcreteState` qui définit l'état courant.
- **State** (TCPState)
  - Définit une interface pour encapsuler le comportement associé à un état particulier de `Context`.
- **ConcreteState subclasses** (TCPEstablished, TCPListen, TCPClosed)
  - Chaque sous-classe implémente un comportement associé à un état de `Context`.

Collaborations



- *Context* délègue les requêtes spécifiques de l'état à l'objet *ConcreteState*.
- Un contexte peut se passer lui-même en argument de l'objet *State* gérant la requête. Ceci permet à l'objet *State* d'accéder au contexte si nécessaire.
- *Context* est l'interface primitif des clients. Les Clients peuvent configurer un contexte avec des objets State. Une fois un contexte configuré, ses clients n'ont pas à faire avec les objets State directement.
- Soit *Context* soit les sousclasses *ConcreteState* peuvent décider quel état succède à un autre et dans quelles circonstances.

Avantage et défauts du pattern State

- State offre une structure qui rend son intention plus claire.
- Les transitions d'états sont explicites
- Ce pattern utilise un grand nombre de classes

Qui gouverne les transitions d'états ?

- Le prochain état est différent pour chaque méthode
  - Context ou
  - State ou
  - une table (unHashMap avec clé ancien état et valeur prochain état pour chaque méthode)

3.2 Le pattern Strategy

Comportement

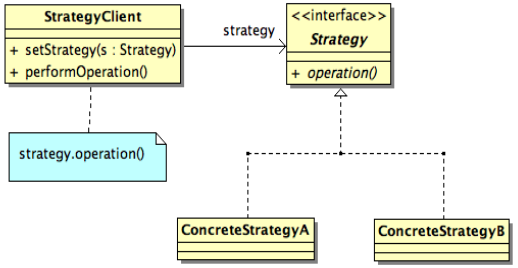
Intention

- Définir un ensemble de classes qui représentent une famille d'algorithmes et les rendre interchangeable.

S'utilise quand

- Plusieurs classes reliées diffèrent par leur comportement.
- Besoin de différentes variantes d'un algorithme. Les variantes peuvent être implémentées en une hiérarchie d'algorithmes.
- Un algorithme utilise des données que ne connaît pas l'utilisateur.
- Une classe définit de nombreux comportements qui apparaissent dans des conditionnelles.

Structure de Strategy



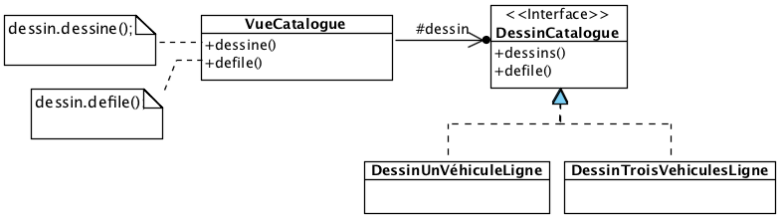
`StrategyClient` : la classe qui utilise les différentes stratégies

`Strategy` : déclare une interface commune à tous les algorithmes

`ConcreteStrategy` : implémente l'algorithme en utilisant l'interface Strategy.

Exemple du patron Strategy

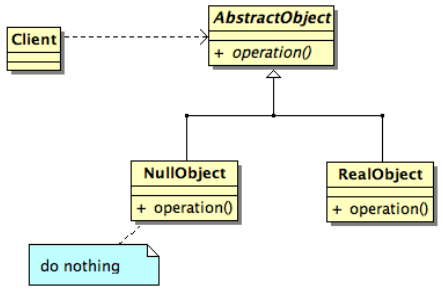
Un système de vente en ligne de véhicule, la classe `VueCatalogue` dessine la liste des véhicules destinés à la vente. Un algorithme de dessin est utilisé pour calculer la mise en page en fonction du navigateur.
Deux versions de l'algorithme sont possibles.



Le pattern « Null Object »

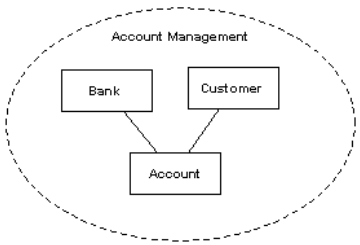
Un autre pattern similaire à State et à Strategy.

Au lieu d'utiliser une référence null pour communiquer l'absence d'un objet on utilise un objet qui implémente l'interface attendue mais dont le corps de la méthode est vide.



Collaborations en UML 2.0

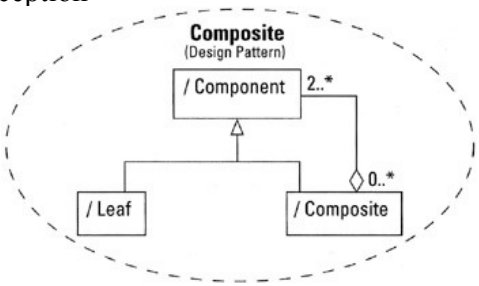
- Représente un ensemble de rôles qui peut être utilisés conjointement pour réaliser une fonctionnalité particulière.
- Les collaborations peuvent être utilisées dans les diagrammes de classes UML pour montrer la collaboration entre les classes impliquées dans un patron de conception.



Collaborations (classes)

Composite

- Une collaboration représente comment des éléments du modèle coopèrent pour réaliser un comportement essentiel.
- les éléments participant peuvent inclure des classes et des objets, des associations, attributs, opérations ...
- Un usage habituel des collaborations est la modélisation des patrons de conception



Références que vous pouvez consulter

Design Patterns: Elements of Reusable Object-Oriented Software, E.Gamma, R.Helm ; R.Johnson, J.Vlissides, Addison-Wesley

Le même en français

- Design Patterns : Catalogue de modèles de conceptions réutilisables, éditions Vuibert.

Quelques sites :

- http://fr.wikipedia.org/wiki/Patron_de_conception
- http://fr.wikibooks.org/wiki/Patrons_de_conception