Les Design Pattern GoF

Ces patterns trés célèbres ont été conçus par 4 informaticiens : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, surnommés le ”**Gang of Four”** (d'où le terme GoF pour ces pattern).
Ils proposent eux aussi des **solutions élégantes,** et toujours différentes pour **résoudre** différents problèmes récurrents rencontrés par les **architectes logiciels**.

Les collections

Très pratique pour stocker un ensemble d'éléments de même type..
Comment les gérer ?
Quelle est l'opération la plus **courante** sur une collection ?
Quelques **types** de collection ?
Quelles **différences** selon le type de collection ?

L'Itérateur

Idée générale :

ne pas **dépendre** de la collection, et cependant être capable de la parcourir !

Être **générique**..

Ne pas avoir à **connaître l'implémentation** interne, n'avoir qu'à s'en servir...

Les collections : **ensembles**, **listes chaînes, tableaux associatifs... Des OBJETS chargés de stocker...**

Des caractéristiques différentes (taille fixe ou évolutive, indexation, parcours, ajout).. Mais une utilisation fréquement séquentielle...

Itérateur

Utilisation

On utilise **Iterator** :

- pour **accéder** à un objet composé dont on ne veut pas exposer la structure interne
- pour **offrir** plusieurs manières de parcourir une structure composée
- pour offrir une **interface uniforme** pour parcourir différentes structures

Exercice Itérateur

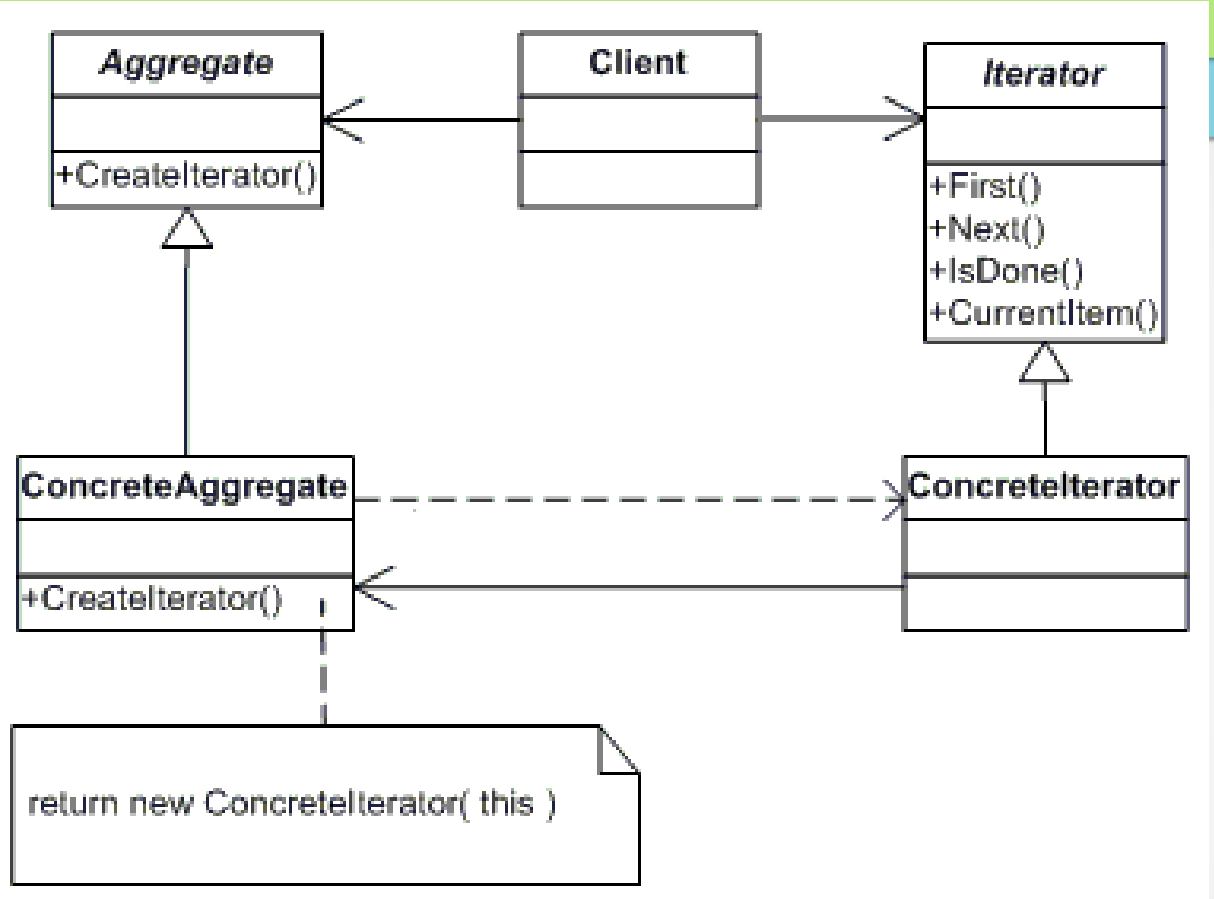
Lister :

les besoins : Qu'est ce *qu'itérer* ?
On a fait besoin de deux méthodes...
lesquelles ? ??

Ensuite :

Si on doit itérer, *qui* doit nous donner l'itérateur ?
Qui sait quel itérateur sait exactement comme itérer ? Avons *nous* à le savoir ?

Iterator



Il faut **connaître** et **comprendre** ce pattern, qui offre un outil pour se déplacer dans une collection, **récupérer** l'élément courant, **aller** au suivant ...

Exemple en Java

On veut gérer un ensemble de boissons servies au Gala de l'Imac 2008. Comment implémenter une solution générique ?
Une classe **BoissonImac** (qui stocke les boissons)
Une classe **DrinkIterator**, qui gère le parcours sur BoissonImac (*c'est d'ailleurs BoissonImac qui nous le fournit*)

Un peu plus proche du Code

Le programme principal...

```
public static void main(String[] args) {
        BoissonImac bi=new BoissonImac(); //on créé notre distributeur de boissons
        Iterator packViking=bi.iterator(); //l'itérator qu'il nous fournit
        while (packViking.hasNext()) { //et je parcours..
            System.out.println(packViking.next()); //et je bois.. :-(((
        }
    }
```

Listons les besoins...

Une classe **BoissonImac**... (elle cachera sa façon de stocker les boissons).. Par contre, elle sera Iterable (donc, obligatoirement, fournit un *Iterator*)
Cet **Iterator** (on ne sait même pas qui il est) offre les méthodes **hasNext(),Next(), et remove())**

La solution de l'implémentation...

```
class BoissonImac implements Iterable {
    String boissons[]={"Biere","Vodka shot Imac 2008","Alcool Frelaté"};
    public Iterator iterator() {
       return new DrinkIterator(this); //ici, le secret.. le choix de l'iterator
    }
    public int length() {
        return boissons.length; //et oui, autre secret.. c'est en fait un tableau !
    }
    public String get(int indice) {
        return boissons[indice]; //ben oui.. le secret du tableau, on le gère jusqu'au bout
    }
}
```

```
class DrinkIterator implements Iterator {
    private BoissonImac collection; //la collection dont je suis l'iterator
    int compteur=0;  // mon petit secret;; c'est un tableau, je compte les indices
   DrinkIterator(BoissonImac collection_A_Parcourir) {
        this.collection=collection_A_Parcourir;
    }
    public boolean hasNext() {  return compteur<collection.length();  }
    public Object next() {  return collection.get(compteur++); }
    public void remove() { //j'ai pas fait);  }
}
```

Singleton

Le pattern **Singleton** permet de créer un système qui contrôle **l'unicité** de la présence d'une instance pour toute la durée de l'application.
Le concepteur veut **instancier** (il veut un objet **dynamique !!!**), mais *une et une seule fois.*
La motivation du programmeur est de s'assurer qu'un objet est **toujours le même**, dans le temps, ou pour **mieux gérer la mémoire.**

Singleton, sa vie son oeuvre

Prenons par exemple un objet utilisé pour se logger, ou pour communiquer, ou pour dialoguer avec la base de données. De nombreuses classes risquent d'en avoir besoin. Faut-il passer cet objet à chaque classe ?
Le Singleton va vous permettre de toujours récupérer l'objet unique, sans vous soucier de vérifier si il existe déjà ou pas, et d'avoir à le passer à d'autres objets avec lesquels vous collaborez...

Singleton : le how to

L'idée, c'est que le singleton CONTIENT une variable statique. C'est elle qui contient l'instance (donc dynamique) du singleton...
Le singleton offre une méthode statique getInstance() qui renvoie la variable statique qu'il contient. On ne peut obtenir le singleton réel qu'en passant par la méthode statique. On récupère toujours le même (le constructeur est privé !).
Le singleton utilise à la fois le coté statique et dynamique du langage. Seul le statique (donc unique) permet d'obtenir le dynamique (l'instance, qui sera sous contrôle des méthodes statiques)

Singleton : le code C++ (*Wikipédia*)

```
template<typename T> class Singleton
{
   public:
    static T& Instance()
    {
        static T theSingleInstance;
              // suppose que T a un constructeur par défaut
        return theSingleInstance;
    }
};

class OnlyOne : public Singleton<OnlyOne>
{
    friend class Singleton<OnlyOne>;
    //...définir ici le reste de l'interface
};
```

Singleton en Java

```
public class Singleton {
    /** Création de l'instance au niveau de la variable statique.  */
    private static final Singleton INSTANCE = new Singleton();

    /**La présence d'un constructeur privé supprime
     * le constructeur public par défaut.
     * personne ne peut faire new Singleton().. sauf moi !  */
    private Singleton() {}

    /**Retourne l'instance du singleton.   */
    public static Singleton getInstance() {
        return INSTANCE;
    }
    /** A partir d'ici, toutes les méthodes de l'instance (non statique)
     */
}
```

Singleton en PHP

- Singleton en PHP : un doute pour ce langage ?

```php
<?php
	class Singleton {
		private static $_instance = null;
		// Constructeur de la classe
		private function __construct() {
			//faire des trucs ici
		}
		// Méthode qui crée l'unique instance de la classe
		//si elle n'existe pas encore puis la retourne.
		public static function getInstance() {
			if(is_null(self::$_instance)) {
				self::$_instance = new Singleton();
			}
			return self::$_instance;
		}
	}
?>
```

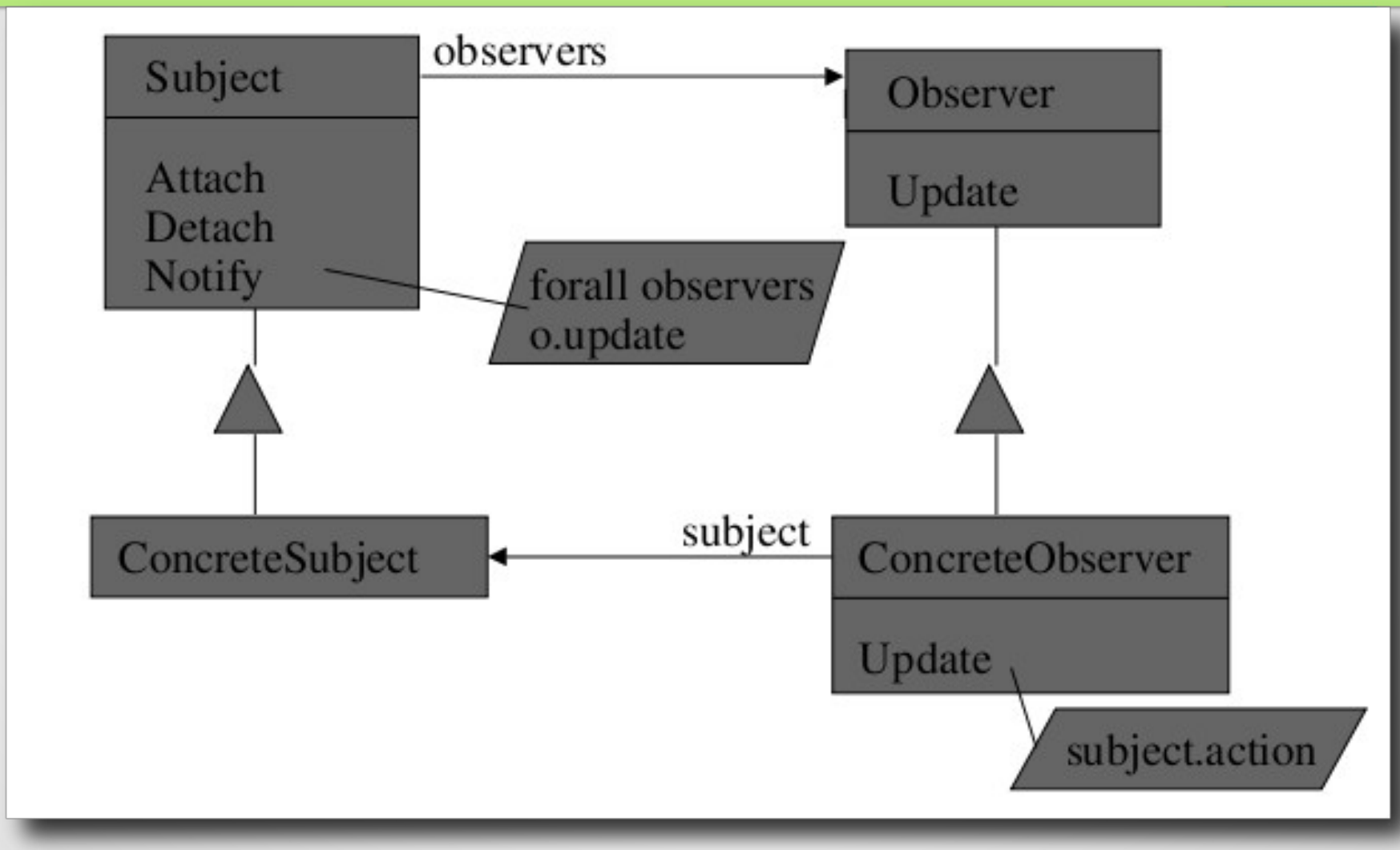
Des Singletons...

- Trouvez des exemples de singletons...

Observateur

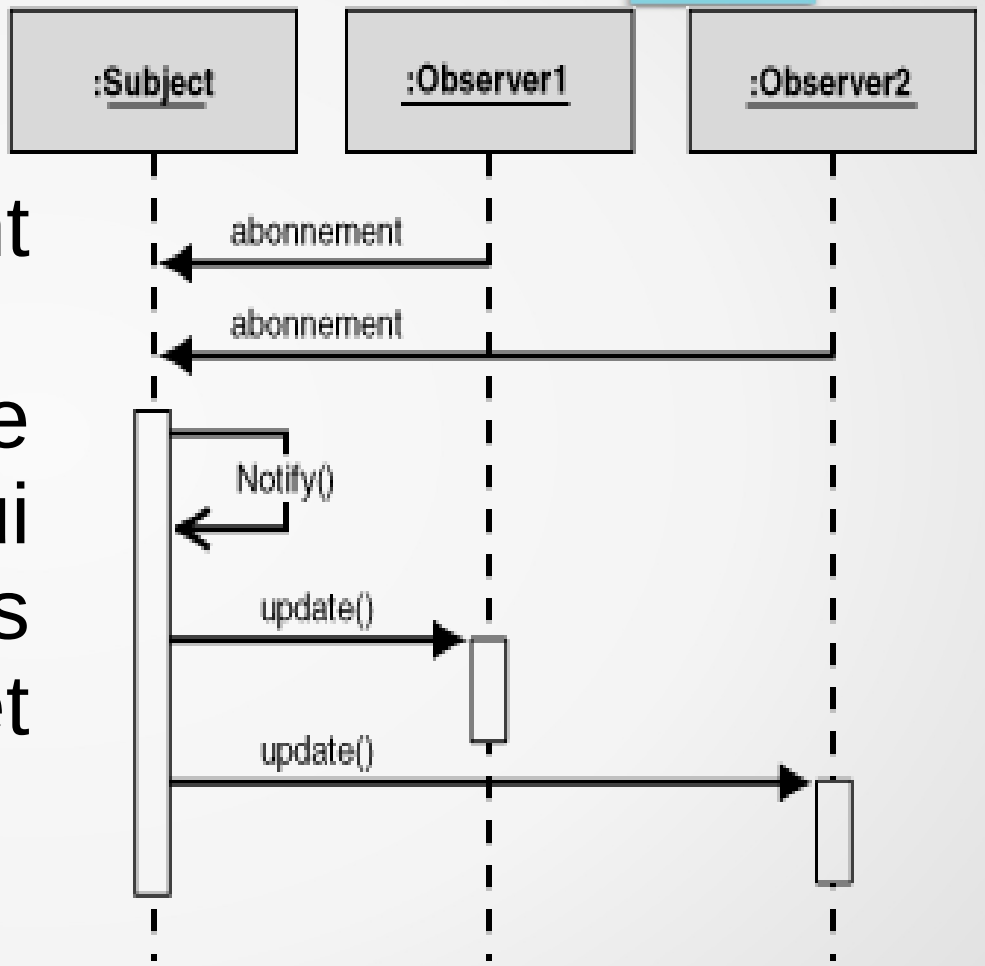
Le modéle Observateur est un mécanisme de diffusion d'évenements, il diffuse des mises à jour d'un sujet, à des écouteurs qui se sont enregistré à celui-ci.
Lorsque le sujet change d'état, il appelle **notify**(), qui lance **update()** sur tous les observateurs de ce sujet, **update**() peut effectuer des taches trés différentes selon les observateurs ...

UML pour Observateur



DSS pour Observateur

Deux objets s'abonnent à un Subject..
Celui-ci implémente une méthode notify(), qui appellera les méthodes update() de chaque objet abonné...



Observateur en PHP

```php
<?php
class Machine implements Subject {
	private $_observers = array();
	//implementation de l'interface Subject
	public function attach(Observer $o){
		//j'ajoute $o à $_observers
	}
	public function detach(Observer $o){
		//je retire $o de mes observers
	}
	public function notify(){
		foreach($this->_observers as $o){
			$o->update($this);
		}
	}
	//ceci marque la fin de l'implementation de l'interface
	public function demarrer(){
		//plein de choses
		$this->notify(); //on avertit nos observateurs
	}
}
```

```php
class Logger implements Observer {
	// -- reste de la classe puis
	//implementation de Observer
public function update(Subject $s){
	$this->log('Petit msg');
	}
}
```

Stratégie

Parfois, il faut pourvoir autoriser la modification du comportement de l'application dynamiquement. Cela peut être un véritable plus pour l'utilisateur : Ainsi, un programme de ventes de produits doit permettre à l'utilisateur l'ajout de nouvelles possibilités, par exemple des promotions (imaginons par exemple ”*1 offert pour l'achat de 2*”, offre ”*Bundle*”, etc). Comment permettre à l'application cette ouverture ?
La réponse s'appelle **STRATEGIE**.

Stratégie...

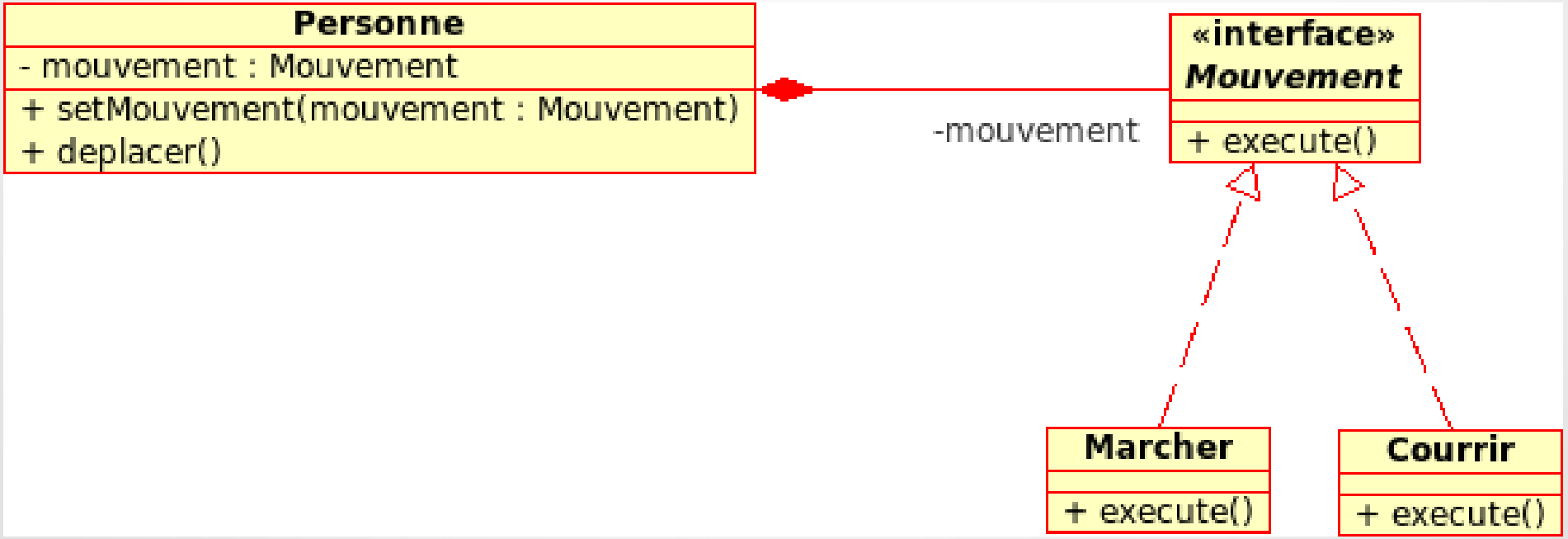
Chaque vente sera passée à la Stratégie de promotion, qui pourra alors appliquer l'algorithme. Si l'entreprise décide d'une nouvelle forme de promotion, il faudra juste coder la Classe correspondante, et l'application appellera cette nouvelle Stratégie...
En général, le système est divisé en deux parties : une **Factory** qui instancie la **Stratégie**. La Factory pourra être pilotée par un fichier de configuration, une variable système, etc...

Exemple de Stratégie

Comment permettre à une Personne d'avoir des mouvements...
Qui peuvent être **marcher**, **courir**.. ou autre chose...
Comment faire pour que le déplacement de chaque instance de Personne puisse être spécifique à cette personne...
Modifiable...
Evolutif...
Et le tout dynamiquement ?
C'est à dire qu'une Personne offre une méthode deplacer() mais que cette méthode soit différente selon les cas, entiérement configurable ?

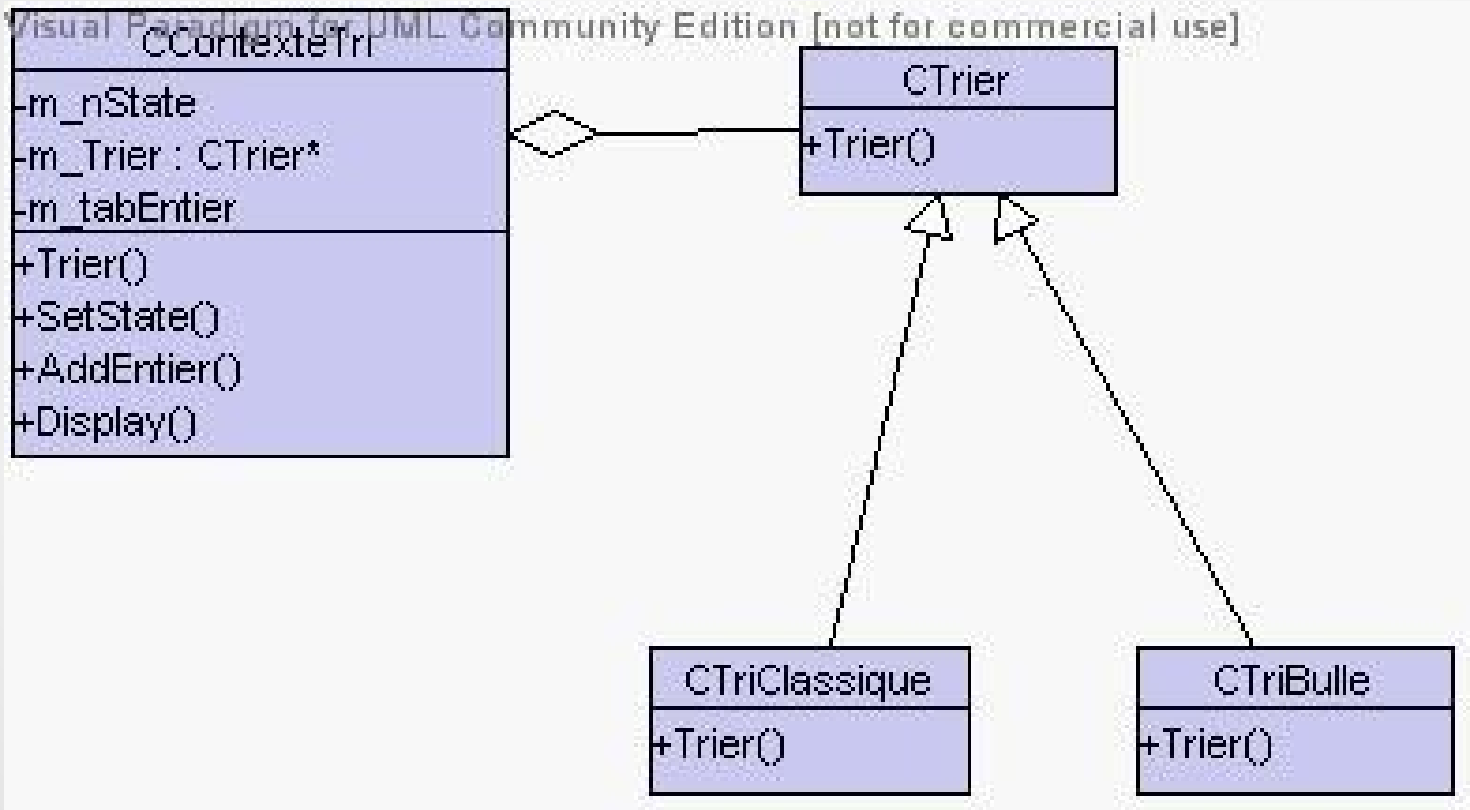
Exemple de Stragégie : Solution

C'est la méthode SetMouvement, appelée en passant une instance de Mouvement concréte qui en décidera...



Autre exemple de Stratégie...

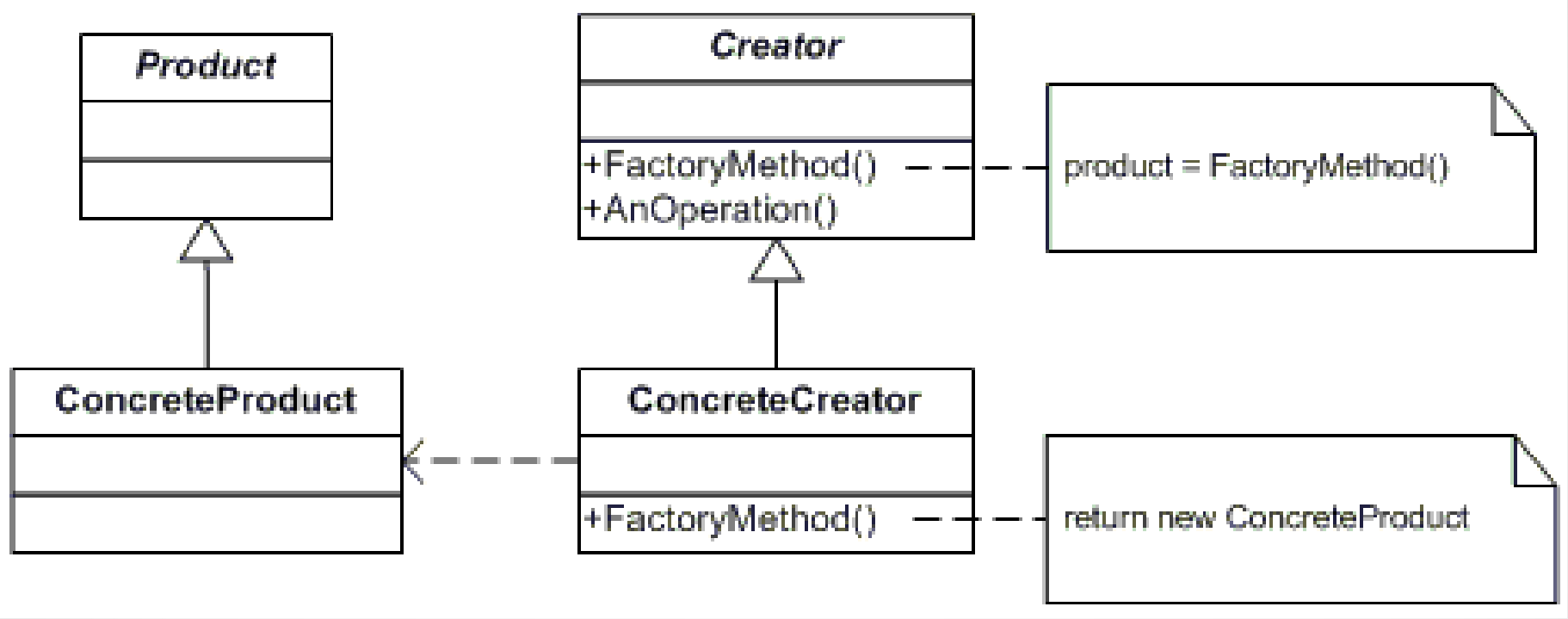
Il s'agit ici de laisser ouvert le choix de la méthode de tri à utiliser...



Factory

Une fabrique de création (ou factory) est une classe qui n'a pour rôle que de construire des objets. Cette classe utilise des interfaces ou des classes abstraites pour masquer l'origine des objets.
Cela permet d'obtenir des objets selon des **mots clés** (on passe le mot clé, on obtient un objet correspondant), soit qui répondent à des **contraintes d'implémentations** (les Interfaces) sans avoir trop à se soucier des contraintes d'instanciation de ces objets... Les objets fournis peuvent évoluer (notamment leurs méthodes d'instanciation) sans géner le programme utilisateur.

UML : Factory



Product est la classe qu'il faut créer – Concrete (en est l'implémentation).

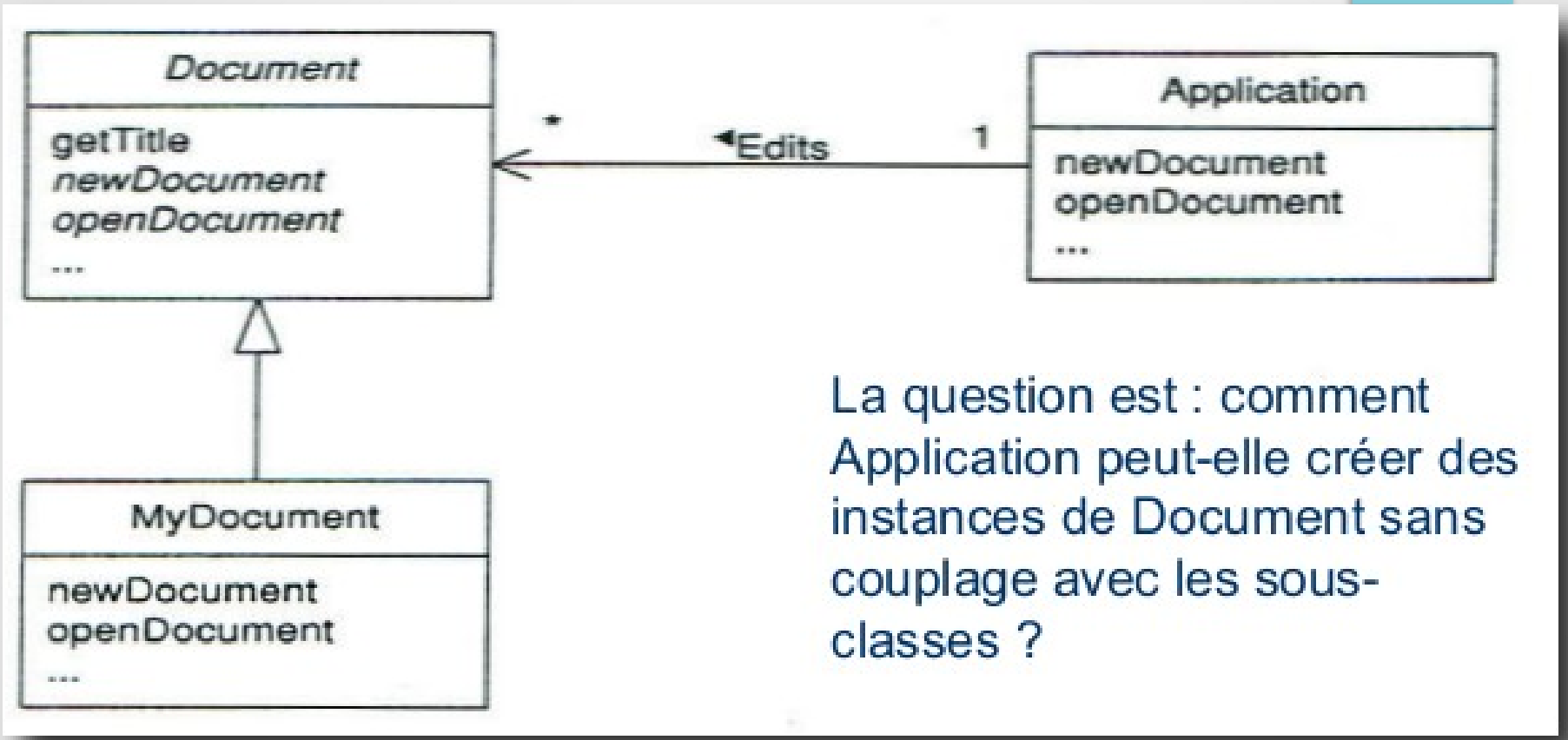
Exemple de Factory (en Java)

Des analyseurs XML sont proposés par défaut au sein de la plateforme Java.Le paquetage javax.xml.parsers présente deux classes très intéressantes : SAXParser et SAXParserFactory.

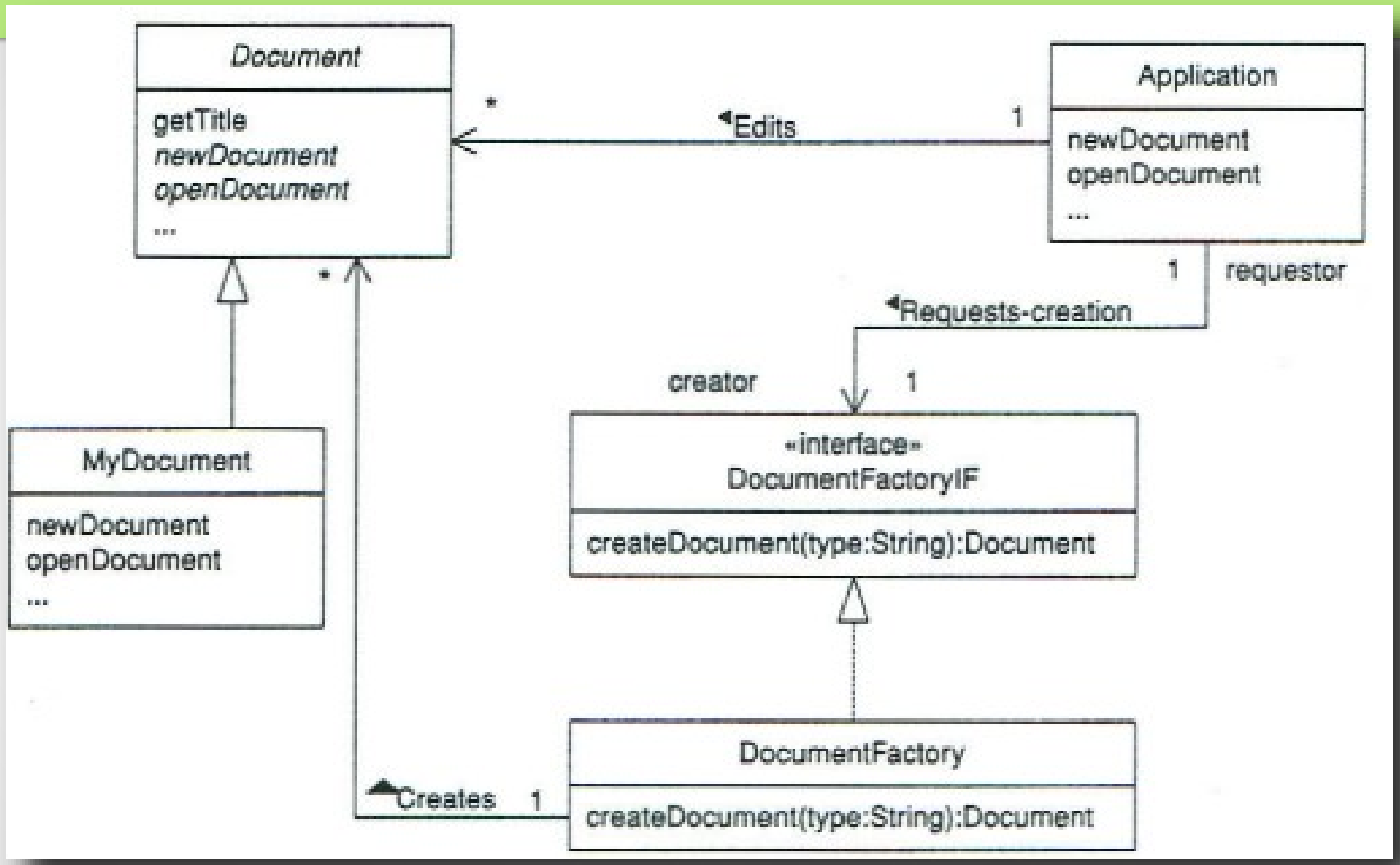
SAXParserFactory fabric = SAXParserFactory.newInstance();
SAXParser analyseur = fabric.newSAXParser();

La procédure de création d'analyseurs SAX passe par l'obtention d'une instance de la classe SAXParserFactory par l'intermédaire de la méthode de classe *newInstance*(). Puis, il suffit d'invoquer la méthode d'instance *newSAXParser()* sur la fabrique pour récupérer un analyseur SAX.

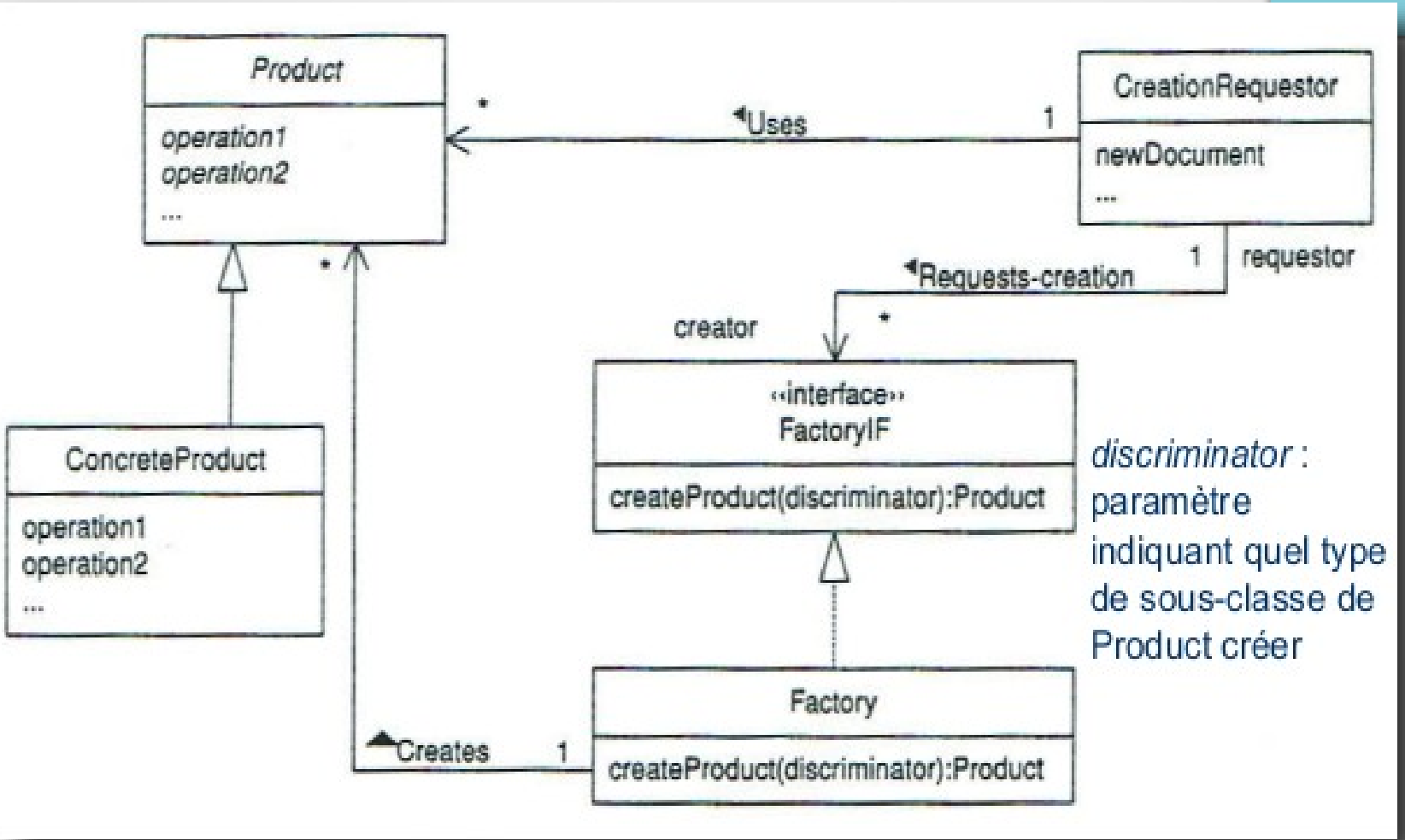
Problème de gestion générique



Solution : la Factory *(new, c'est le mal !)*



Solution générique Factory



Exemple Factory

Une application doit pouvoir exporter les données de plusieurs types d'objets (disons les objets Person, Company, Vehicle) dans un format spécifique (choisissons XML).
Ces objets ne doivent même pas avoir connaissance du fait qu'il peuvent être exportés. Aucune méthode export(). On aura une classe ExportFactory. Elle ne comporte qu'une seule méthode:
public static ExportObject createExportable(Object source)
ExportObject est une interface que tous les objets exportés concrets doivent implémenter. Elle ne contient qu'une méthode (du moins dans cet exemple):
public String export();

Suite exemple Factory

Dans la méthode, createExportable, on instancie dynamiquement des objets en fonction du paramètre source : (ici en Java)

```
public static ExportObject createExportable(Object source){
      if(source instanceof Person){
            return new PersonExport(source); // on renvoie...
      }else if(source instanceof Company){
            return new CompanyExport(source); // .. le bon objet..
      }else if(source instanceof Vehicle){
            return new VehicleExport(source);
      }
      return null;
}
```

Factory...

La classe PersonExport ressemblera à ceci:

```
public class ExportPerson implements ExportObject {
  private Person person;

  public ExportPerson(Person person){
      super();
      this.person=person;
  }

  public String export(){
      StringBuffer xml = new StringBuffer();
      xml.append("<?xml version='1.0'?><person>");
      xml.append("<firstname>")+person.getFirstName()+"</firstname>");
      xml.append("</person>");
      return xml.toString();
  }
}
```

Utilisation de la Factory

Maintenant, assemblons le tout.
Un utilisateur sélectionne une personne et demande à l'exporter (en XML, donc).
La méthode ExportFactory.createExportable(Object source) est appelée. La méthode renvoie un ExportObject sur lequel on peut appeler la méthode export(); même sans savoir de quel type d'objet (concret) il s'agit. (Il n'y a donc pas de new !)

```
Person p1 = new Person(); p1.setFirstName("Hack");
Vehicle veh = new Vehicle(); veh.setMark("Seat");

ExportObject expObj=ExportFactory.createExportable(p1);
System.out.println(expObj.export());
expObj=ExportFactory.createExportable(veh);
System.out.println(expObj.export());
```

Différence Stratégie / Factory ?

Cela se ressemble...
La Factory est une forme particulière de Strategy.
Extrait de Forum.java.sun.com..

Factory is a creation pattern. Strategy is not.

Abstract Factory is a case of Strategy applied to the creation objects.

You can use both patterns to achieve a result: have different things done in different cases, the difference is where you make that choice: when you instantiate the objects or when you call a method to execute some action.

Architecture logicielle : MVC

Bien que plus ancien (1979 au Palo Alto Research Center de XEROX) que les Design Pattern GOF ou GRASP, cette architecture pose clairement les différentes responsabilités des Classes, afin d'offrir des libertés d'évolution de l'application.
Dans cette architecture, on divise l'application en trois parties : les données, les traitements et la présentation -> ce qui nous donne Modèle, Contrôleur, et Vue...

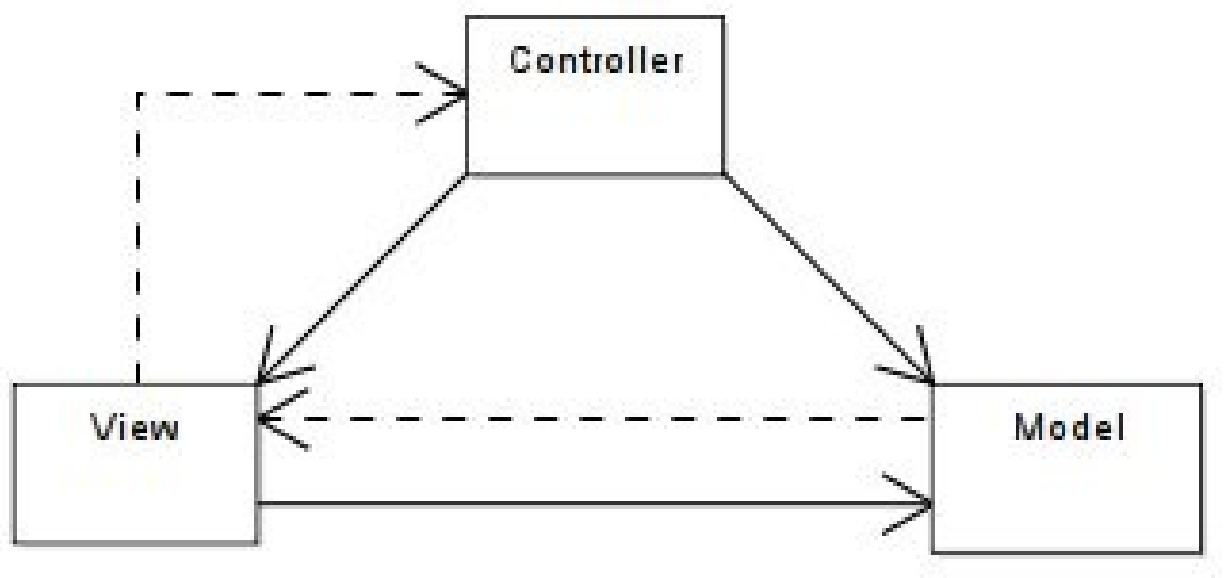
MVC (source Wikipédia)

* Le **Modèle** représente le comportement de l'application : traitements des données, interactions avec la base de données, etc. Il décrit les données manipulées par l'application et définit les méthodes d'accès.

* la **Vue** correspond à l'interface. Elle peut être conçue en html, ou tout autre « langage » de présentation. La vue n'effectue aucun traitement, elle affiche les résultats des traitements effectués par le modèle, et de permet à l'utilisateur d'interagir avec elles.

* le **Contrôleur** prend en charge la gestion des événements de synchronisation pour mettre à jour la vue ou le modèle. Il n'effectue aucun traitement, ne modifie aucune donnée, il analyse la requête du client et se contente d'appeler le modèle adéquat et de renvoyer la vue correspondant à la demande.

MVC...



Les différentes interactions entre le modèle, la vue et le contrôleur sont résumées par le schéma de la figure suivante... Les flèches en pointillées indiquent un Design Pattern Observateur...

MVC et le WEB...

Beaucoup d'applications Web utilisent le MVC, afin d'être plus évolutives. Il existe des frameworks de développement correspondant à ce paradigme (Rails, Symfony, Struts, Spring).
Les vues sont les pages HTMLs, les contrôleurs récupérent les données, les traitent, en interrogeant le Modèle. Les contrôleurs reconstruisent ensuite des vues qui sont envoyées aux clients.

Exemple Stratégie en PHP (kilooctet.net)

Dans cet exemple, on veut pouvoir valider des champs d'un formulaire HTML. Il faut parfois valider des dates, des numériques, des emails, etc...

D'abord une interface qui va définir la méthode indiquant si un champ est valide ou non :

```
/** * @author Benoit MILGRAM */
interface com.nodule.validator.IValidator {
    public function isValid( s : String ) : Boolean;
}
```

Suite exemple Strategy PHP

Stratégie pour valider une adresse email :

```
/**  * @author Benoit MILGRAM  */
class EmailValidator implements IValidator {
    public function EmailValidator() {   }
    public function isValid( s : String ) : Boolean {
        return ((StringUtil.contains(s,"@"))&amp;amp;&amp;amp;(StringUtil.contains(s,".")));
    }
}
```

Un code postal :

```
class ZipCodeValidator implements IValidator {
    private static var ZIPCODE_LENGTH : Number = 5;
    public function ZipCodeValidator() {    }
    public function isValid( s : String ) : Boolean {
       return ((Number(s) != NaN) &amp;amp;&amp;amp; ( s.length == ZIPCODE_LENGTH) );
    }
}
```

La classe qui appliquera la Stragégie

Une classe Validator, qui contient un validator !!
Et qui se sert de la méthode fournie par l'Interface

```
class Validator {
    private var _oValidatorStrategy : IValidator;
    public function Validator() {    };
    public function setValidator(strategy : Ivalidator) {
        _oValidatorStrategy = strategy;
     }
    public function isOkay( s : String ) : Boolean {
        return _oValidatorStrategy.isValid( s );
    }
 }
```

Fin : Utilisation du Validator

```
...
var validate : IValidator = new Validator();
validate.setValid( new EmailValidator());
validate.isOkay( tiEmail.text);
validate.setValid( new DateValidator());
validate.isOkay( DateNaissance.text);
...
```

On voit qu'on peut changer le comportement de la méthode isOkay... Les instances de EmailValidator et de DateValidator pourraient être stockées dans des variables, afin d'être directement réutilisées.