COSMIC DESIGN PATTERNS AVEC PHP

Un tutoriel décrivant, en langage PHP, 13 design patterns parmi les plus utilisés.

1 Préambule

Ce tutoriel s'adresse aux développeurs qui souhaitent découvrir ou approfondir les design patterns, ces modèles de programmation répondant chacun à une problématique particulière. (http://fr.wikipedia.org/wiki/Patron_de_conception)

Chaque design pattern est autonome. Il n'est donc pas nécessaire de suivre l'ordre dans lequel ils apparaissent dans ce tutoriel.

Cependant, Pseudo Factory, FactoryMethod et Abstract Factory sont 3 patterns formant une famille.

Par ailleurs Composite s'appuie sur Iterator.

Le langage utilisé est PHP. Il est donc nécessaire de disposer d'un petit environnement web, ainsi que d'un IDE (environnement de développement). Voir à ce sujet le chapitre « **Outils utilisés** » en annexe.

L'ensemble du code PHP est fourni dans une archive à part, disponible sur smaltek.fr.

Table des matières

2 PATTERN STRATEGY

L'USS Enterprise utilise différents types de robots pour effectuer certaines missions: exploration, renseignement, prises de mesures, et même nettoyage des couloirs de l'Enterprise.

Certains marchent, d'autres roulent ou volent.

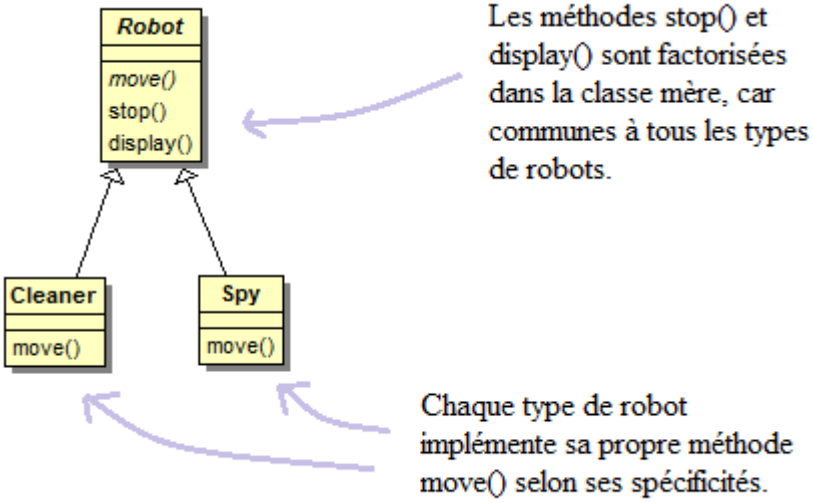
Ils ont connaissance des paramètres de leur mission notamment le parcours à effectuer. Un programme permet de les suivre à l'écran.

Certains types de robots peuvent maintenant devenir invisibles, comme les robots de renseignements type SPY.

Le Capitaine Kirk vous confie le soin de modifier le programme pour pouvoir exploiter ces nouveaux robots et leur capacité à devenir invisible.

Le précédent développeur a utilisé la programmation orientée objet avec le modèle de classes suivant, et deux implémentations:

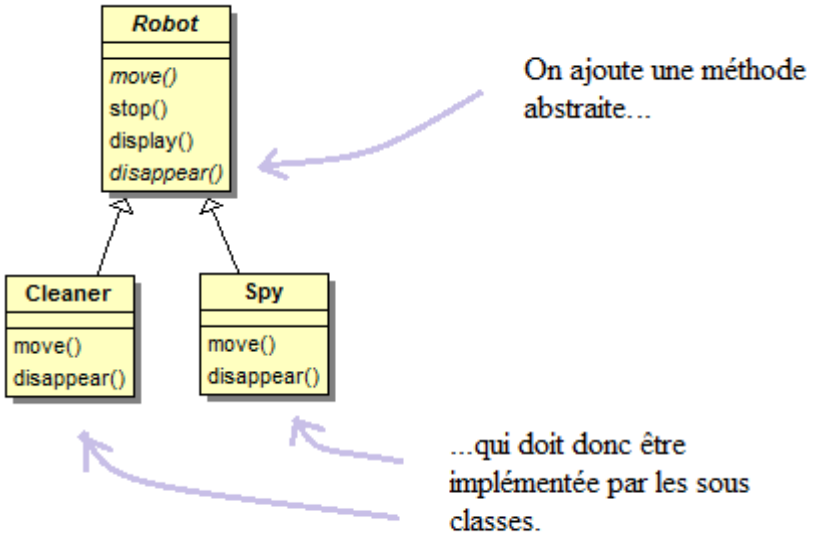
- Cleaner: robot nettoyeur. Il se déplace sur des roulettes.
- Spy: robot de renseignement. Il marche comme un humain.



Il ne reste plus qu'à ajouter une méthode Disappear() à notre modèle et le tour est joué. Nous pourrons contrôler l'invisibilité des robots concernés.

2.1 Première approche

Essayons d'ajouter la méthode disappear() à la classe mère.



Le job n'était pas très difficile, mais on constate rapidement quelques inconvénients:

1. Le robot de type Cleaner n'est pas concerné par l'invisibilité. Nous serons pourtant contraint

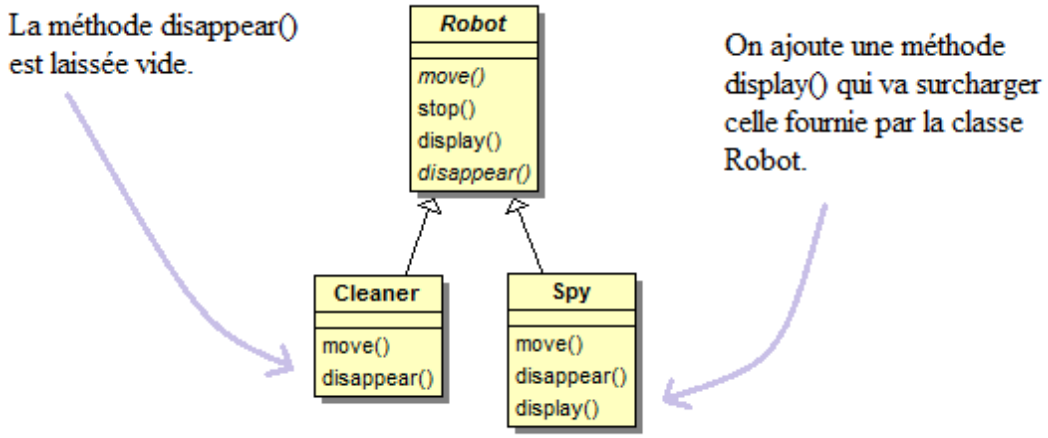
de lui ajouter une méthode disappear().

2. La méthode display() qui est commune à tous les types de robots, puisque implémentée dans la classe mère, devra certainement être adaptée pour les robots dotés de l'invisibilité. En effet il faudra, et pour eux uniquement, ajouter un indicateur visuel d'invisibilité à l'écran.

Continuons sur notre approche pour voir où elle nous mène.

- Concernant l'inconvénient N° 1, laissons la méthode disappear() vide dans la classe Cleaner. Ce n'est pas très joli, mais cela ne crée pas de problème particulier.
- Concernant l'inconvénient N° 2, nous pouvons surcharger (donner une implémentation différente de celle fournie par la classe mère) la méthode display() dans la classe Spy, et toutes les éventuelles sous classes qui nécessitent un affichage particulier du robot. Hummm... de moins en moins joli.

On obtiendrait ceci:



Cela n'a pas l'air trop mal, mais... il semble qu'il manque quelque chose. Les robots capteurs de mesures (type Sensor) ont été oubliés. Ils sont utilisés pour récolter toutes sortes de mesures sur un site donné, mais ne peuvent pas se déplacer. Il faut les déposer manuellement sur un site, ou les faire déposer par un robot Spy par exemple.

En résumé le robot de type Sensor:

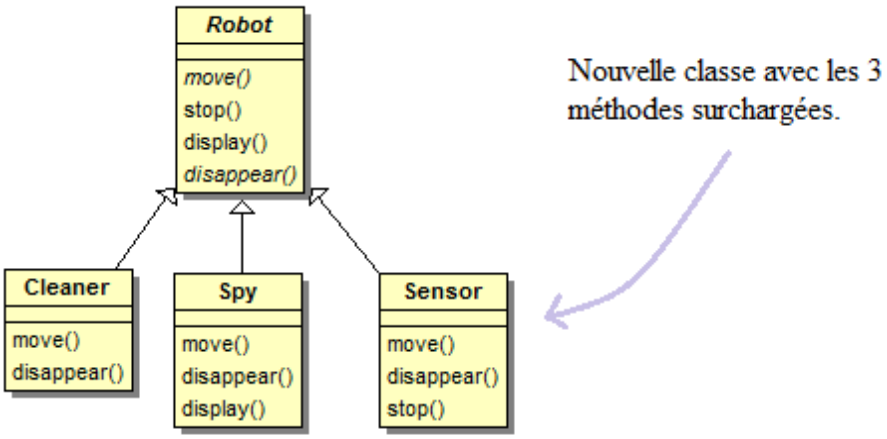
- n'est pas concerné par les méthodes move() et stop()

- n'est pas actuellement concerné par la méthode disappear().

Pas de problème. On va ajouter une classe Sensor dans laquelle:

- On implémente la méthode move() en la laissant vide.
- On implémente la méthode disappear() en la laissant vide.
- On surcharge la méthode stop() en la laissant vide.
- On garde la méthode display() de la classe mère.

On obtient:



Il est temps de faire un premier bilan de notre approche.

- En voulant utiliser uniquement l'héritage, on est obligé de surcharger certaines méthodes dans les sous classes qui le nécessitent, soit pour les adapter à un comportement spécifique, soit pour les inhiber (implémentation à vide).
- Les méthodes abstraites comme move() et disappear(), doivent être implémentées dans les sous classes. Ceci mène tôt ou tard à de la duplication de code. Imaginez deux types de robots qui ont la même manière de se déplacer. Ils auront la même implémentation de la méthode move(). Etre obligé de dupliquer du code est souvent le signe d'une mauvaise approche.
- Les comportements des différents types de robots sont définis par l'héritage, donc de manière statique. Il est impossible de modifier cela dynamiquement, au moment de l'exécution du programme.
- Ici les choses ne sont pas trop compliquées car nous n'avons que 3 types de robots. Mais que se passerait-il avec 15 ou 20 types différents ? Souhaitons bon courage à la personne

chargée de la maintenance d'un tel montage !

2.2 Deuxième approche

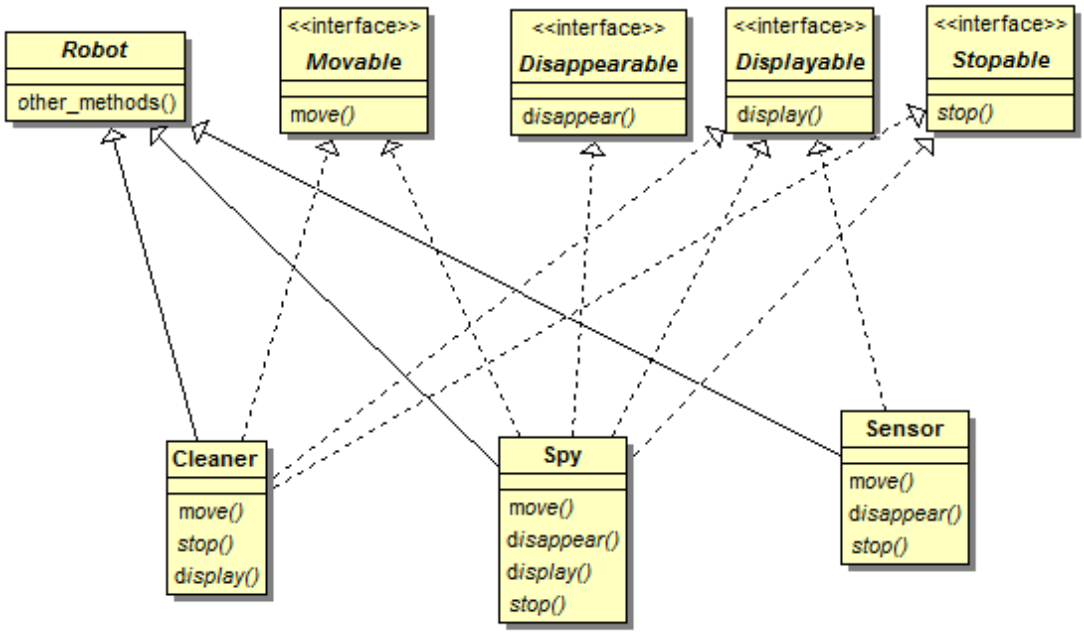
OK. Je sais ce que vous pensez. Il suffit de créer une interface pour chaque comportement qui n'est pas commun à tous les robots. De cette manière, les sous classes pourront implémenter les interfaces dont elles ont besoin.

- Cleaner a besoin d'être Movable, Displayable, Stopable.
- Spy a besoin d'être Movable, Disappearable, Displayable, Stopable.
- Sensor a besoin d'être Displayable.

Les comportements variables sont maintenant matérialisés par des interfaces. Les méthodes correspondantes ont donc été retirées de la classe Robot.

Les sous classes implémentent les interfaces dont elles ont besoin, et doivent définir l'implémentation concrète de ces méthodes.

On obtient:



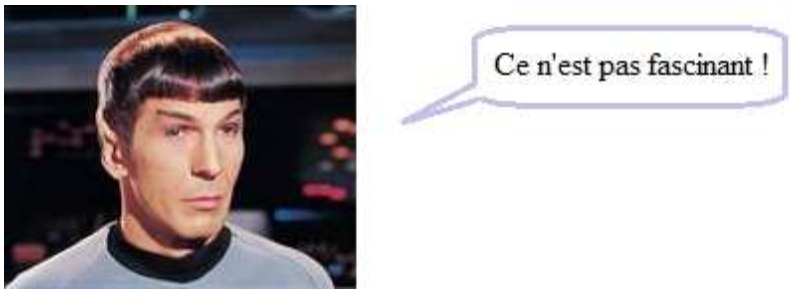
Nous avons ici 4 interfaces et 3 sous classes, donc un maximum théorique de 12 implémentations d'interfaces (flèches en pointillés). Par chance, seulement 8 sur 12 nous sont nécessaires.

Imaginez ce montage avec, par exemple:

- 6 interfaces.
- 6 sous classes.

Nous serions confrontés à un maximum théorique de 36 flèches pointillées. Ingérable n'est-ce pas ?

Et Spock ne trouverait pas ça fascinant !



Dans notre première approche, nous nous sommes appuyé uniquement sur l'héritage. Notre seconde approche basée sur les interfaces, semblait intéressante, mais il s'avère qu'elle mènera rapidement à un modèle peu évolutif et difficile à maintenir.

De plus les deux approches présentent les inconvénients que l'on cherche justement à éviter en

programmation objet:

- Duplication de code.
- Obligation d'implémenter des méthodes à vide.
- Modèle dont la complexité évolue exponentiellement avec le nombre de classes ou d'interfaces.
- Modèle plutôt rigide qui aura du mal à évoluer, et qui est donc difficile à maintenir.
- Les comportements des robots sont figés dans la conception des classes, il est donc impossible de les choisir de manière dynamique, à l'exécution du programme.

2.3 Cap sur le pattern Strategy

Pourquoi avons-nous autant de mal à concevoir un modèle de classes efficace pour nos 3 types de robots ?

Le problème vient du fait que nous essayons de mélanger deux concepts difficilement compatibles:

- D'un côté, les comportements des robots qui varient beaucoup d'un type de robot à un autre. Considérons par exemple la méthode move(): nous savons que les robots ne se déplacent pas tous de la même manière. De plus certains robots ne se déplacent pas du tout.
- De l'autre, les types de robots qui sont des concepts qui ne changent pas ou peu.

En essayant d'associer directement des comportements qui changent à des types de robots qui ne changent pas, nous obtenons un modèle rigide et peu évolutif. Dans le meilleur des cas celui-ci sera utilisable à condition qu'il n'y ait que 2 ou 3 types de robots et 2 ou 3 comportements maximum. Et en priant pour qu'on nous demande jamais d'en ajouter un quatrième.

Le pattern Strategy est basé sur une séparation forte entre ces deux ensembles:

- Les concepts qui changent ou pourraient changer à l'avenir (on parle aussi de variabilité).
- Les concepts qui ne changent pas et ne changeront probablement pas à l'avenir (on parle aussi de communalité).

La première étape consiste à former ces deux ensembles:

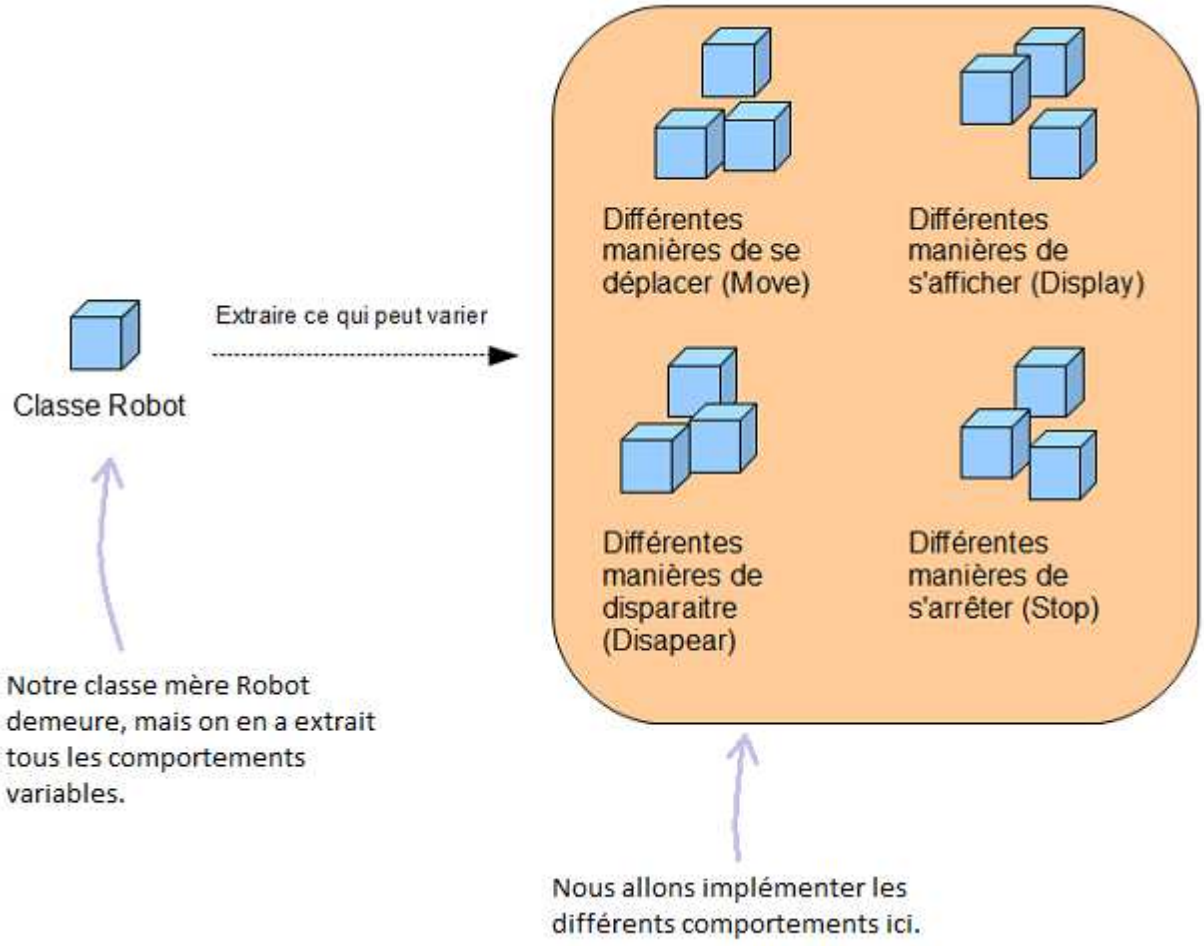
Dans notre communalité, mettons les concepts qui ne présentent pas de caractère changeant:

- Les robots. En effet le concept de robot est matérialisé par une classe mère Robot, et 3 sous

classes Spy, Sensor et Cleaner. Ce montage risque peu d'être modifié. L'ajout de sous classes supplémentaires est possible mais cela ne remet pas en jeu la stabilité du concept.

Dans notre variabilité, mettons les concepts qui présentent un caractère changeant, c'est à dire qui doivent être implémenté de manière différente, d'un type de robot à un autre:

- Move: tous les robots ne se déplacent pas de la même manière.
- Disappear: certains robots ne sont pas concernés.
- Display: Les robots invisibles seront affichés d'une manière différente.
- Stop: certains robots ne sont pas concernés.

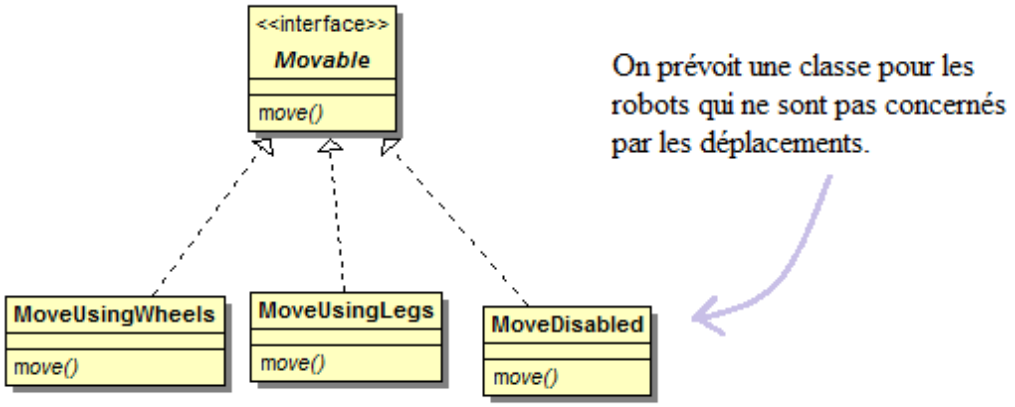


2.3.1 Implémenter les comportements variables

Les différents comportements de type Move, sont situés concrètement dans des classes qui

implémentent une nouvelle interface nommée Movable.

Cette interface contient une méthode Move() qui doit donc être implémentée par chaque classe.



C'est de cette façon que les comportements variables seront dissimulés à la classe Robot, qui ne verra que des objets de type Movable.

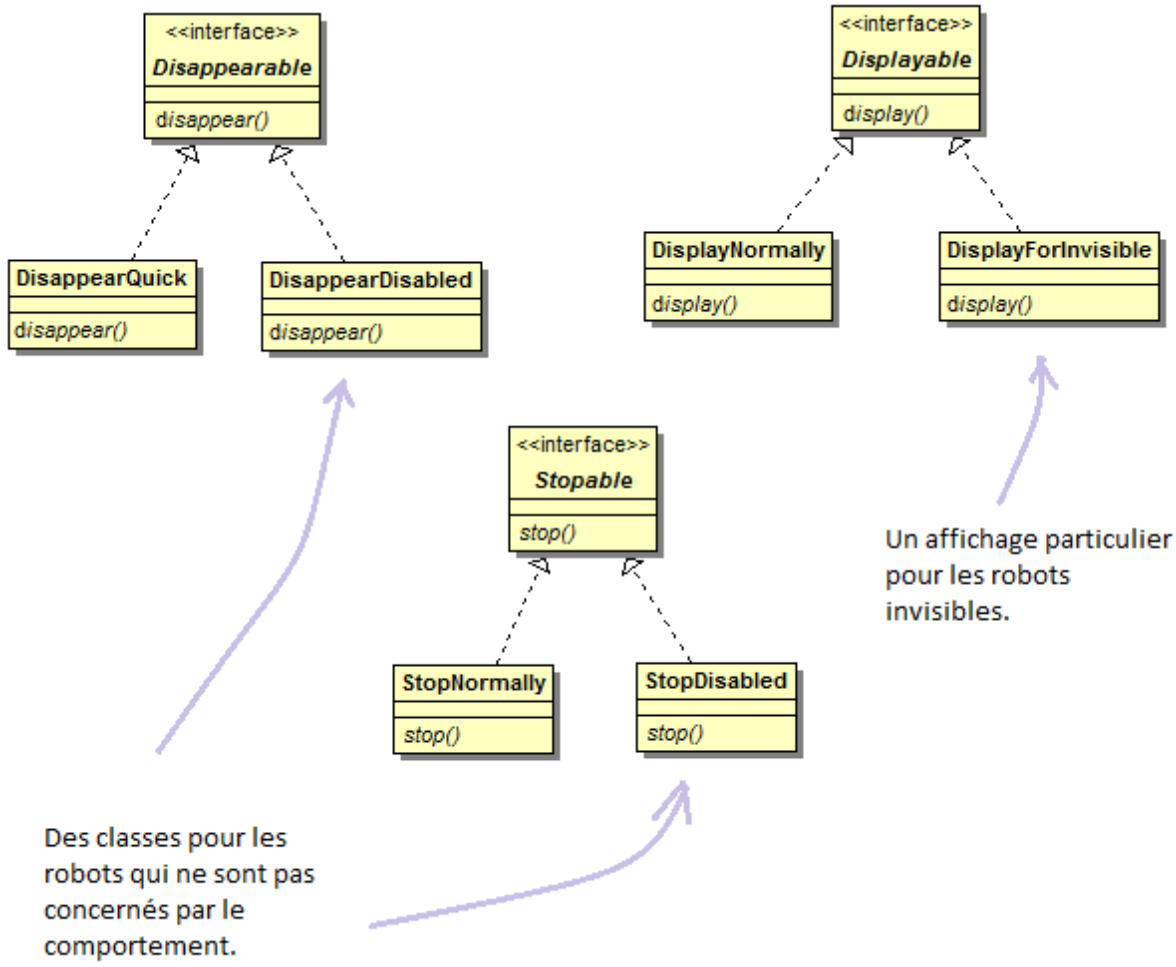
En d'autres termes, les robots n'auront pas connaissance des détails d'implémentation de leur méthode move(). Ils sauront simplement qu'ils disposent d'une méthode move().

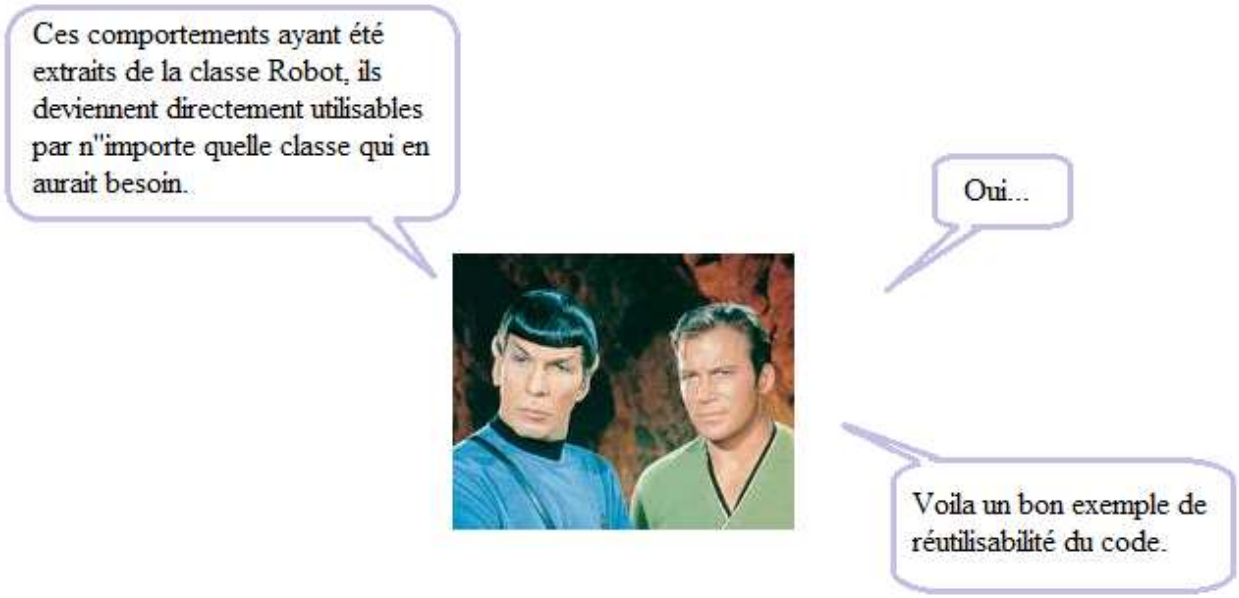
2.3.2 Comment définir les comportements de chaque robots

Dans nos premières tentatives, l'association entre un comportement et un robot, était figé dans l'héritage des classes. Ce manque de souplesse était la cause de nos problèmes.

Maintenant que ces deux concepts sont complètement séparés, c'est au moment de l'instanciation des objets, c'est à dire à l'exécution du programme, qu'on attribuera à chaque robot, la manière de se déplacer qui lui convient.

Faisons de même pour les autres comportements (Disappear, Display, Stop):

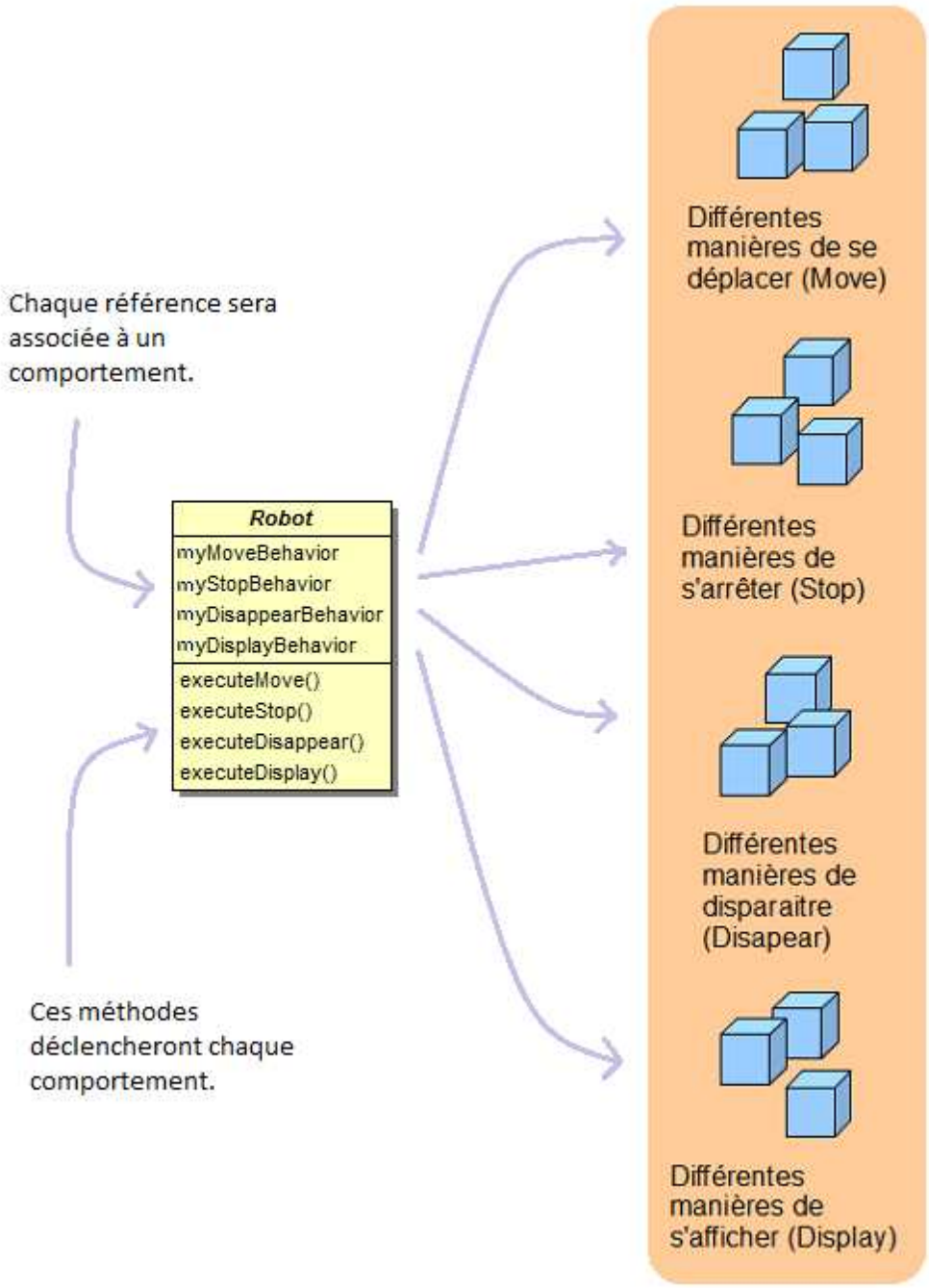

<<interface>>
Disappearable
disappear()
DisappearQuick
disappear()
DisappearDisabled
disappear()
<<interface>>
Displayable
display()
DisplayNormally
display()
DisplayForInvisible
display()
<<interface>>
Stopable
stop()
StopNormally
stop()
StopDisabled
stop()
Un affichage particulier pour les robots invisibles.
Des classes pour les robots qui ne sont pas concernés par le comportement.



2.3.3 Doter les robots de comportements

Il s'agit maintenant de créer le lien entre un robot et les comportements qui vont lui être attribués.

- Tout d'abord, un robot doit posséder une référence pour chacun de ses comportements. Ceci étant une règle s'appliquant à tous les types de robots, nous créons ces références dans la classe mère Robot.
- Enfin tout robot doit pouvoir utiliser les comportements dont il dispose. Pour cela nous créons dans la classe mère Robot, et pour chaque comportement, une méthode publique qui sera responsable de l'exécution de ce comportement.


Chaque référence sera associée à un comportement.
Robot
myMoveBehavior
myStopBehavior
myDisappearBehavior
myDisplayBehavior
executeMove()
executeStop()
executeDisappear()
executeDisplay()
Ces méthodes déclencheront chaque comportement.
Différentes manières de se déplacer (Move)
Différentes manières de s'arrêter (Stop)
Différentes manières de disparaitre (Disapear)
Différentes manières de s'afficher (Display)


J'aimerai voir comment on code tout ça !

2.3.4 Codage

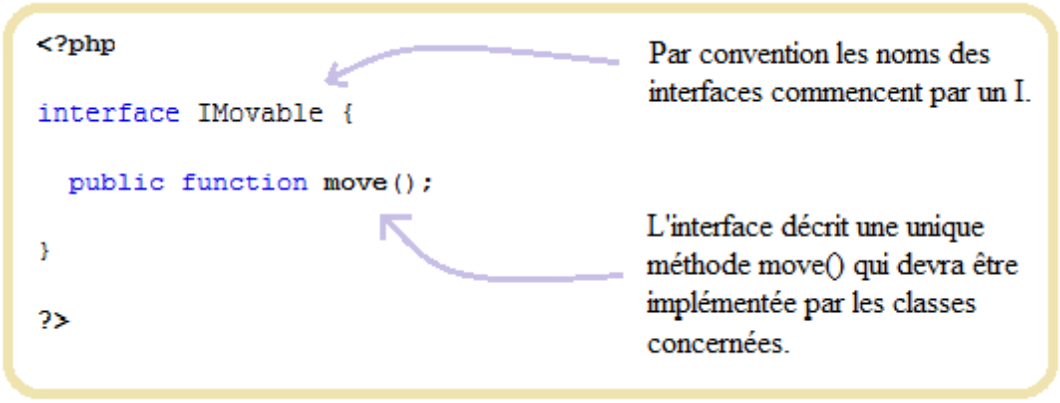
Il est temps de commencer à coder en PHP. Commençons par les comportements variables.

Nous avons 4 interfaces:

- Movable
- Displayable
- Stopable
- Disappearable

...et 2 à 3 classes par interface pour les comportements concrets.

L'interface IMovable:




```php
<?php

interface IMovable {

  public function move();

}

?>
```

Les 3 autres interfaces:

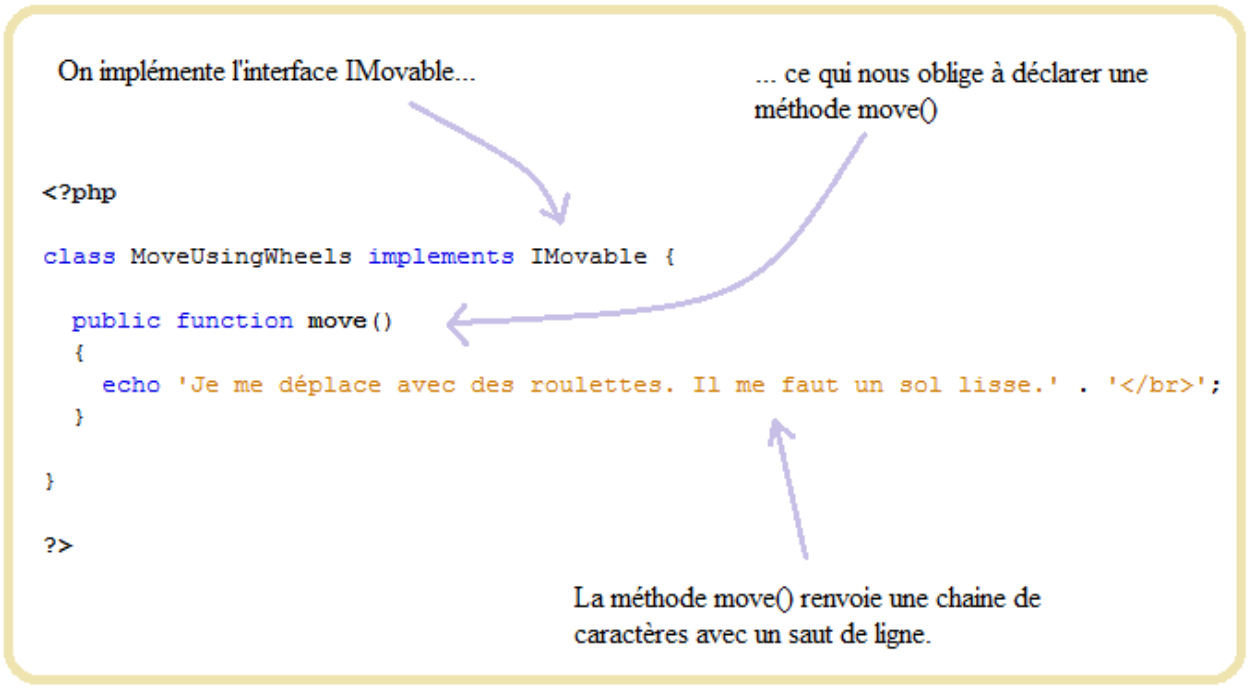
```php
<?php

interface IDisplayable {

  public function Display();

}

?>
```

```php
<?php

interface IStopable {

  public function stop();

}

?>
```

```php
<?php

interface IDisappearable {

  public function disappear();

}

?>
```

Derrière l'interface IMovable, nous avons les classes:

- MoveUsingWheels
- MoveUsingLegs
- MoveDisabled




```php
<?php

class MoveUsingWheels implements IMovable {

  public function move()
  {
    echo 'Je me déplace avec des roulettes. Il me faut un sol lisse.' . '</br>';
  }

}

?>
```

Faisons de même pour les deux autres classes:

```php
<?php

class MoveUsingLegs implements IMovable {

  public function move()
  {
      echo 'Comme un humain, je me déplace avec mes jambes.' . '</br>';
  }

}

?>
```

```php
<?php

class MoveDisabled implements IMovable {

  public function move()
  {
    echo 'Je ne peut pas me déplacer.'  . '</br>';
  }

}

?>
```

Derrière l'interface IDisplayable, nous avons les classes:

- DisplayNormally
- DisplayForInvisible

```php
<?php

class DisplayNormally implements IDisplayable {

  public function display()
  {
    echo 'Affichage normal.' . '</br>';
  }

}

?>
```

```php
<?php

class DisplayForInvisible implements IDisplayable {

  public function display()
  {
    echo 'Affichage spécial pour objets invisibles.' . '</br>';
  }

}

?>
```

Derrière l'interface IStopable, nous avons les classes:

- StopNormally
- StopDisabled

```php
<?php

class StopNormally implements IStopable {

  public function stop()
  {
    echo 'Je m\'arrete.' . '</br>';
  }

}

?>
```

```php
<?php

class StopDisabled implements IStopable {

  public function stop()
  {
    echo 'Je ne dispose pas de la fonction Stop.' . '</br>';
  }

}

?>
```

Derrière l'interface IDisappearable, nous avons les classes:

- DisappearQuick
- DisappearDisabled

```php
<?php

class DisappearQuick implements IDisappearable {

  public function disappear()
  {
    echo 'Passage à l\'état invisible.' . '</br>';
  }

}

?>
```

```php
<?php

class DisappearDisabled implements IDisappearable {

  public function disappear()
  {
    echo 'Invisibilité non disponible.' . '</br>';
  }

}

?>
```

Nous avons codé tous les comportements variables. Il ne reste plus qu'à coder les classes concernant les robots.

Quatre classes sont nécessaires:

- La classe mère Robot.
- Le type de robot Cleaner, dérivant de la classe Robot.
- Le type de robot Spy, dérivant de la classe Robot.
- Le type de robot Sensor, dérivant de la classe Robot.

```php
<?php

abstract class Robot {

  protected $_myMoveBehavior;
  protected $_myStopBehavior;
  protected $_myDisappearBehavior;
  protected $_myDisplayBehavior;



  public function executeMove()
  {
    $this->_myMoveBehavior->move();
  }

  public function executeStop()
  {
    $this->_myStopBehavior->stop();
  }

  public function executeDisappear()
  {
    $this->_myDisappearBehavior->disappear();
  }

  public function executeDisplay()
  {
    $this->_myDisplayBehavior->display();
  }

}  // end class
```

Les 4 références définissant les comportements, seront affectées par les sous classes.

L'exécution d'un comportement est simplement déléguée à l'objet responsable de ce comportement.

Cleaner dérive de la classe Robot.

C'est le constructeur de la classe, qui est responsable du choix des 4 comportements.

```php
<?php

class Cleaner extends Robot {

    function __construct() {

        echo 'Je suis le constructeur de Cleaner.'. '</br>';

        // Initialisation des comportements:
        $this->_myMoveBehavior = new MoveUsingWheels();
        $this->_myDisappearBehavior = new DisappearDisabled;
        $this->_myStopBehavior = new StopNormally;
        $this->_myDisplayBehavior = new DisplayNormally;

    }

}

?>
```

On instancie les comportements concrets adaptés à Cleaner.

```php
<?php

class Spy extends Robot {

    function __construct() {

        echo 'Je suis le constructeur de Spy.'. '</br>';

        // Initialisation des comportements:
        $this->_myMoveBehavior = new MoveUsingLegs();
        $this->_myDisappearBehavior = new DisappearQuick;
        $this->_myStopBehavior = new StopNormally;
        $this->_myDisplayBehavior = new DisplayForInvisible;

    }

}

?>
```

```php
<?php

class Sensor extends Robot {

    function __construct() {

        echo 'Je suis le constructeur de Sensor.'. '</br>';

        // Initialisation des comportements:
        $this->_myMoveBehavior = new MoveDisabled();
        $this->_myDisappearBehavior = new DisappearDisabled;
        $this->_myStopBehavior = new StopDisabled;
        $this->_myDisplayBehavior = new DisplayNormally;

    }

}
?>
```

Notre pattern Strategy est prêt à être utilisé. Pour cela nous allons écrire un petit programme de quelques lignes, qui va utiliser le pattern Strategy, c'est à dire instancier des robots, et leur demander d'exécuter des actions.

Ce programme est en quelque sorte l'utilisateur du pattern Strategy. On peut également l'appeler le Contrôleur dans la mesure où c'est lui qui pilote l'ensemble des opérations.

Dans le cas de PHP, le contrôleur peut être placé dans le fichier index.php car ce fichier est automatiquement exécuté, s'il existe, à l'invocation d'une URL.

Fichier index.php:

```php
<?php

//************************************************************
// STRATEGY pattern controler
//************************************************************

require_once 'includePaths.php';
$newline = "</br>";

echo '*** Controleur: Début' . $newline;

// Tests avec une instance de Spy:
$oSpyRobot1 = new Spy();
$oSpyRobot1->executeMove();
$oSpyRobot1->executeDisplay();
$oSpyRobot1->executeStop();
$oSpyRobot1->executeDisappear();
echo $newline;

// Tests avec une instance de Cleaner:
$oCleanerRobot1 = new Cleaner;
$oCleanerRobot1->executeMove();
$oCleanerRobot1->executeDisplay();
$oCleanerRobot1->executeStop();
$oCleanerRobot1->executeDisappear();
echo $newline;

// Tests avec une instance de Sensor:
$oSensorRobot1 = new Sensor();
$oSensorRobot1->executeMove();
$oSensorRobot1->executeDisplay();
$oSensorRobot1->executeStop();
$oSensorRobot1->executeDisappear();
echo $newline;

echo '*** Controleur: Fin.' . $newline;

?>
```

On instancie un robot.

On lui fait executer ses 4 comportements.

Il ne reste plus qu'à exécuter notre contrôleur (index.php) via une URL dans un navigateur web.

Mr Sulu affichez le résultat à l'écran.

Firefox http://localhost/Des...2012/Strategy_2012/

localhost/DesignPatterns_2012/Strategy_2012/

Désactiver Cookies CSS Form. Images Information Divers Entourer

*** Controleur: Début
Je suis le constructeur de Spy.
Comme un humain, je me déplace avec mes jambes.
Affichage spécial pour objets invisibles.
Je m'arrete.
Passage à l'état invisible.

Je suis le constructeur de Cleaner.
Je me déplace avec des roulettes. Il me faut un sol lisse.
Affichage normal.
Je m'arrete.
Invisibilité non disponible.

Je suis le constructeur de Sensor.
Je ne peux pas me déplacer.
Affichage normal.
Je ne dispose pas de la fonction Stop.
Invisibilité non disponible.

*** Controleur: Fin.

On passe bien par le constructeur.

Les comportements sont bien exécutés

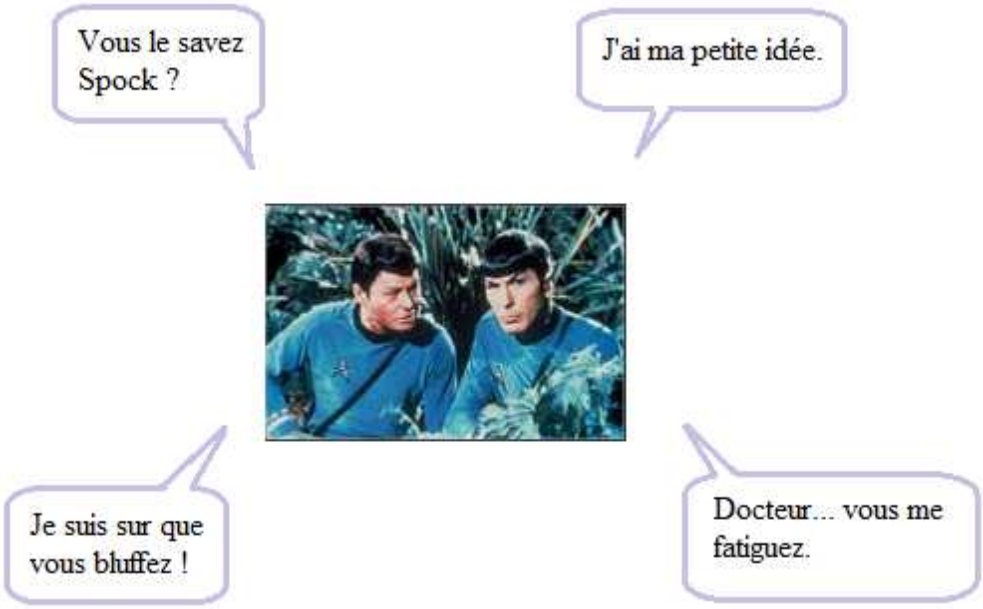
Chaque robot a les comportements qu'on lui a attribué.

Intéressant Mr Spock, mais pas complètement satisfaisant: on s'est donné beaucoup de mal pour séparer la variabilité de la communalité, mais au final le choix des comportements des robots reste défini de manière statique, puisque ce choix est fait par le constructeur de chaque robot.

Si on aborde les choses en termes de responsabilité, on peut dire qu'actuellement chaque robot est responsable du choix de ses comportements.

Mais est-ce vraiment aux robots d'en décider ?

Et si ce n'est eux, qui doit prendre cette responsabilité ?

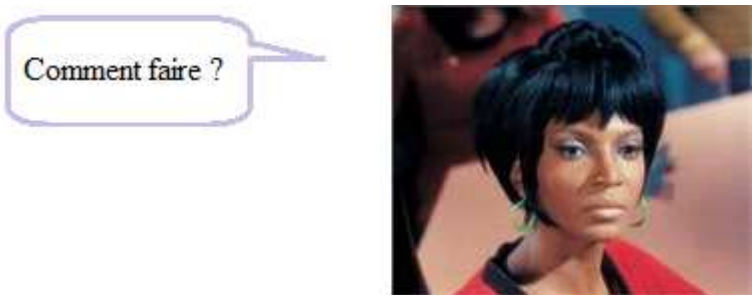


Dans le cadre de notre petit pattern Strategy, le choix des comportements des robots, peut être confié au chef, et le chef c'est le Contrôleur (index.php).

Non seulement le Contrôleur va instancier les robots, mais il pourra modifier leurs comportements à n'importe quel moment.

En transférant cette responsabilité au Contrôleur, on se réserve la possibilité de choisir les comportements des robots, à l'exécution du programme, de manière totalement dynamique, en

fonction d'un contexte.



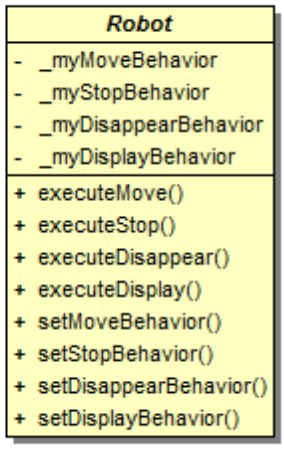
2.3.5 Encore plus de souplesse

Le Contrôleur doit pouvoir:

- Choisir (instancier) un comportement.
- Passer ce comportement à un robot pour qu'il puisse l'adopter.

Ajoutons 4 méthodes à la classe Robot pour pouvoir modifier les comportements:

```
public function setMoveBehavior($oMoveBehavior)
{
  $this->_myMoveBehavior = $oMoveBehavior;
}
```



Le contrôleur pourra utiliser la méthode setMoveBehavior() chaque fois qu'il voudra changer la manière de se déplacer d'un robot. Le comportement en question devra être passé en paramètre.

Voici le code pour les 3 autres méthodes:

```
public function setStopBehavior($oStopBehavior)
{
  $this->_myStopBehavior = $oStopBehavior;
}

public function setDisappearBehavior($oDisappearBehavior)
{
  $this->_myDisappearBehavior = $oDisappearBehavior;
}

public function setDisplayBehavior($oDisplayBehavior)
{
  $this->_myDisplayBehavior = $oDisplayBehavior;
}
```

Modifions légèrement notre contrôleur:

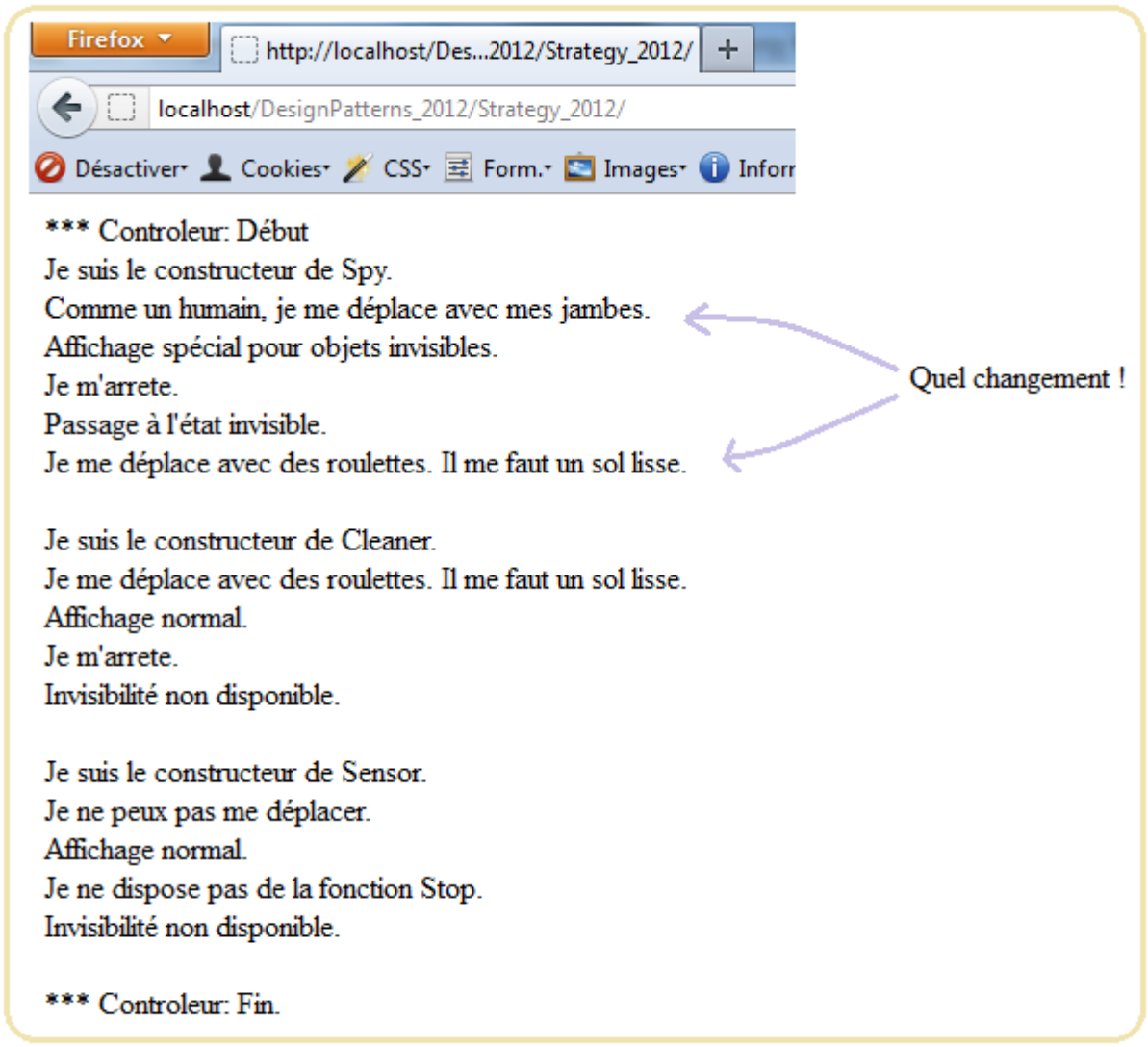
```php
<?php

//*****************************************************
// STRATEGY pattern controler
//*****************************************************

require_once 'includePaths.php';
$newline = "</br>";

echo '*** Controleur: Début' . $newline;

// Tests avec une instance de Spy:
$oSpyRobot1 = new Spy();
$oSpyRobot1->executeMove();
$oSpyRobot1->executeDisplay();
$oSpyRobot1->executeStop();
$oSpyRobot1->executeDisappear();

// Et si $oSpyRobot1 se déplacait maintenant sur des roulettes ?
$oSpyRobot1->setMoveBehavior(new MoveUsingWheels);
$oSpyRobot1->executeMove();


echo $newline;

...
```

Le constructeur de Spy continue de définir les comportements par défaut.

Modifions un comportement de notre robot.

Et voila un robot sur roulettes.

Cela nous donne à l'exécution:



Il a véritablement changé de comportement. C'est fascinant !

2.3.6 Une vision globale

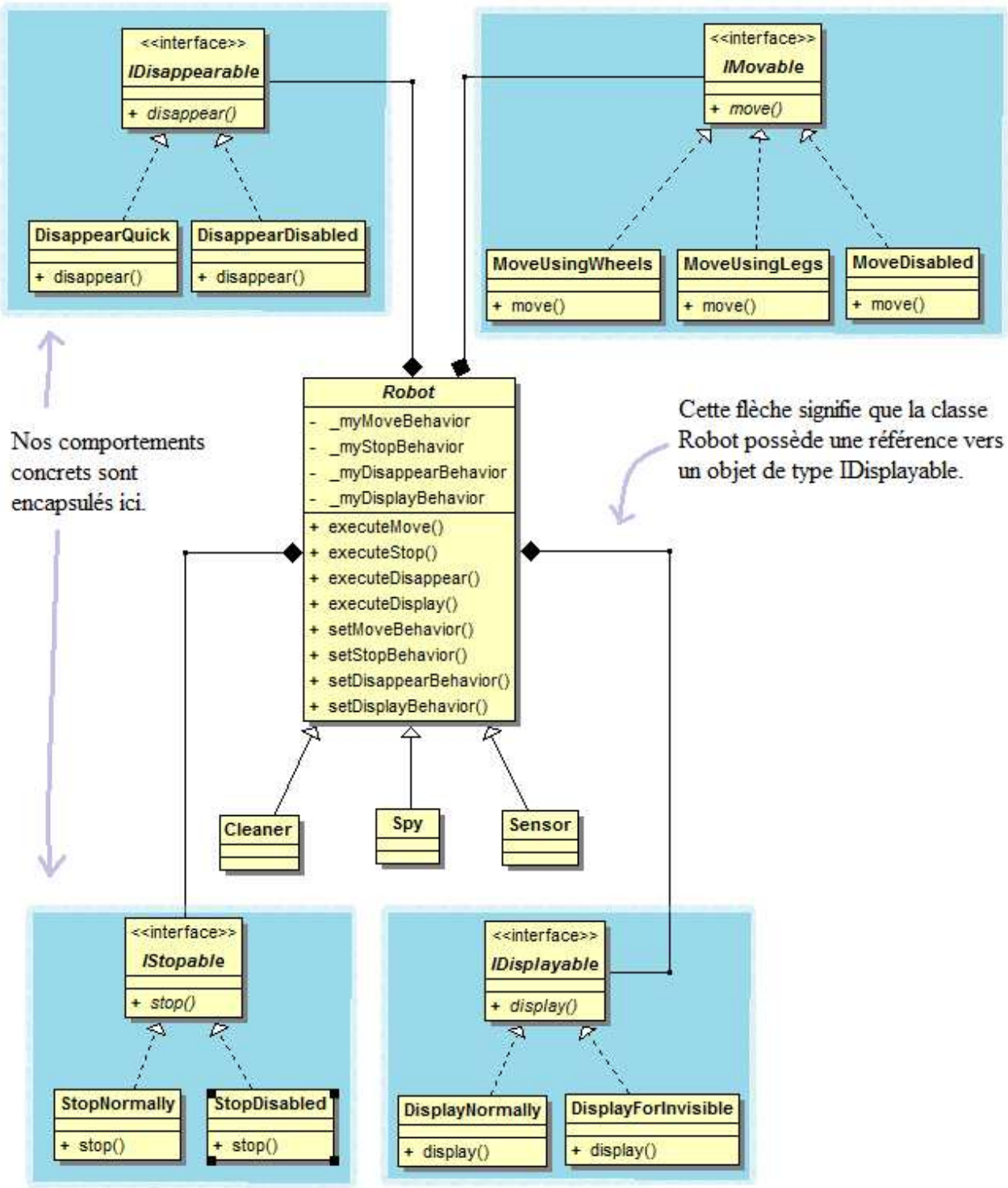
Nous avons bien codé, et notre pattern Strategy fonctionne. Le design d'origine a été complètement transformé, mais qu'avons-nous fait au juste ?

Nous avons extrait les concepts variables et nous les avons encapsulés dans des familles de comportements.

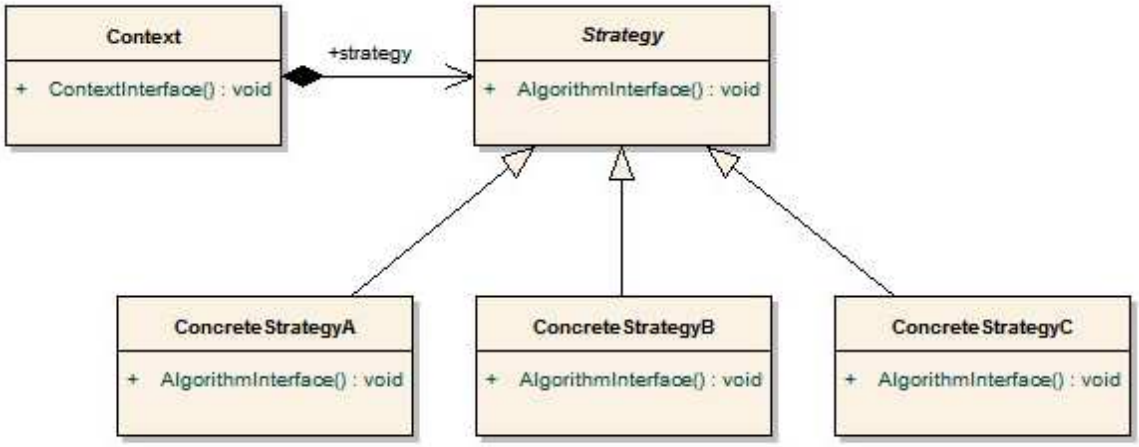
Nous avons gardé la classe mère Robot et ses sous classes, mais les robots ne contrôlent plus directement leurs comportements, car ces derniers ont été délégués à des classes qui encapsulent les comportements concrets.

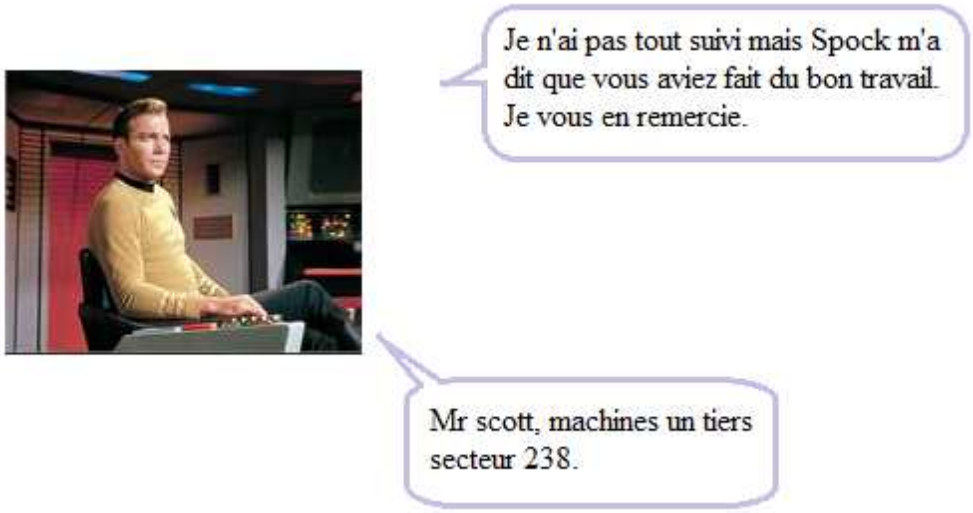
Enfin nous avons ajouté des méthodes permettant de modifier les comportements des robots, et ceci à l'exécution. Ceci était absolument impossible dans l'approche initiale.

Photo de famille:

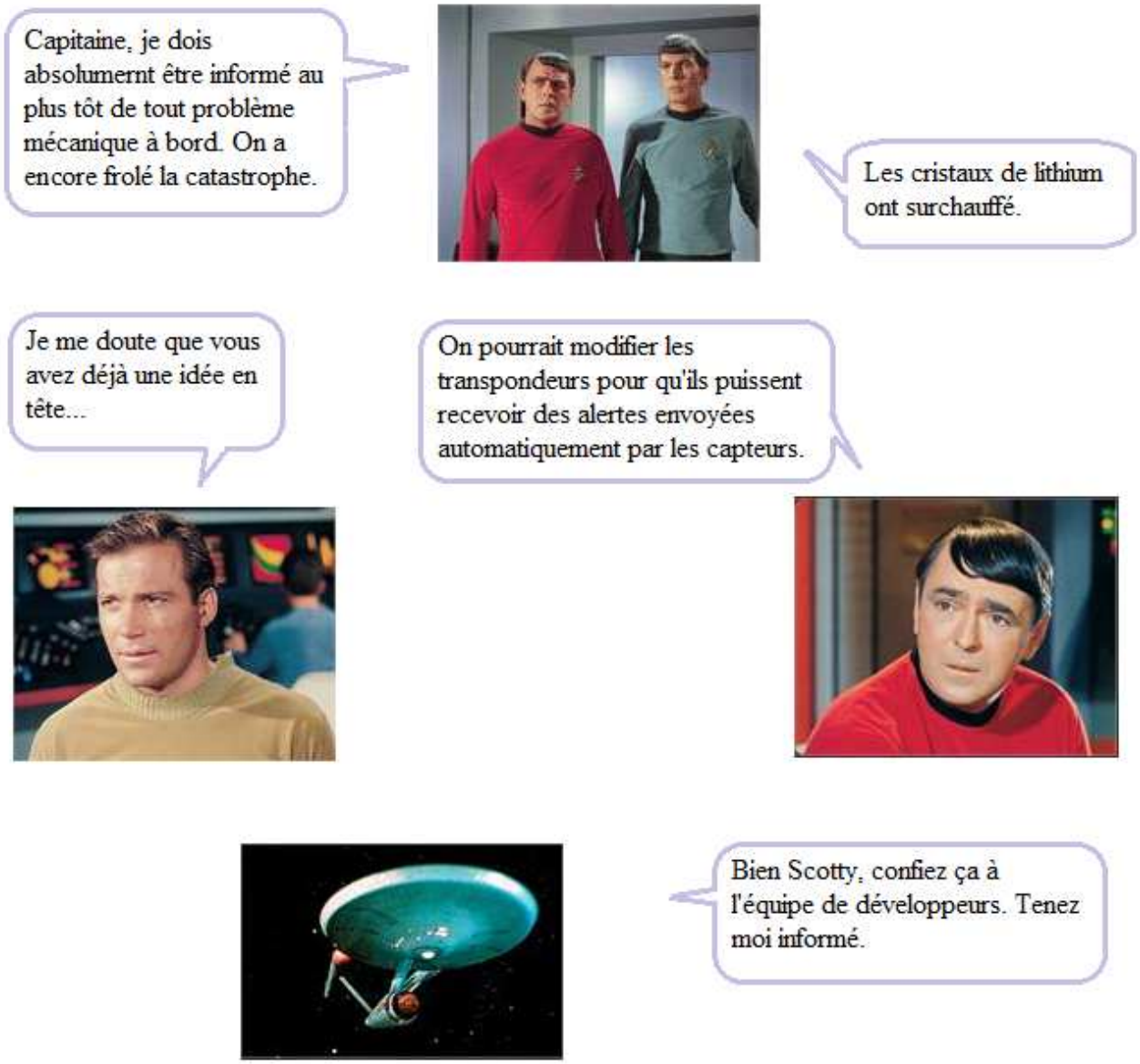


Voici une représentation standard du pattern Strategy que l'on peut trouver sur internet:


Context
+ ContextInterface() : void
+strategy
Strategy
+ AlgorithmInterface() : void
Concrete StrategyA
+ AlgorithmInterface() : void
Concrete StrategyB
+ AlgorithmInterface() : void
Concrete StrategyC
+ AlgorithmInterface() : void


Je n'ai pas tout suivi mais Spock m'a dit que vous aviez fait du bon travail. Je vous en remercie.
Mr scott, machines un tiers secteur 238.

3 PATTERN OBSERVER



Les transpondeurs sont ces petits appareils portatifs que les membres de l'Enterprise portent à la ceinture. Ils servent à communiquer, un peu comme un... téléphone mobile.

On nous demande d'ajouter une fonctionnalité permettant aux transpondeurs de recevoir des alertes lorsque que certains événements se produisent, concernant le système de propulsion du vaisseau. Cela peut concerner:

- La température du cristal de lithium (le carburant du moteur de l'Enterprise). Le seuil d'alerte est fixé à 400 degrés centigrades.
- Le niveau de vibrations du système de propulsion. Sur une échelle de 0 à 7, le seuil d'alerte

est fixé à 5.

- Le régime du système de propulsion. Sur une échelle de 0 à 100, le seuil d'alerte est fixé à 95.

Les transpondeurs pourront prendre l'initiative d'être notifiés, ainsi que de cesser d'être notifié.

Par chance, ce qui nous est demandé correspond exactement à ce qui peut être traité par le pattern Observer. Allons-y.

3.1 Cap sur le pattern Observer

Les personnes qui aiment le jardinage, peuvent s'abonner à un magazine spécialisé. Ils recevront alors périodiquement un numéro du magazine, jusqu'au jour où ils décideront de mettre un terme à leur abonnement.

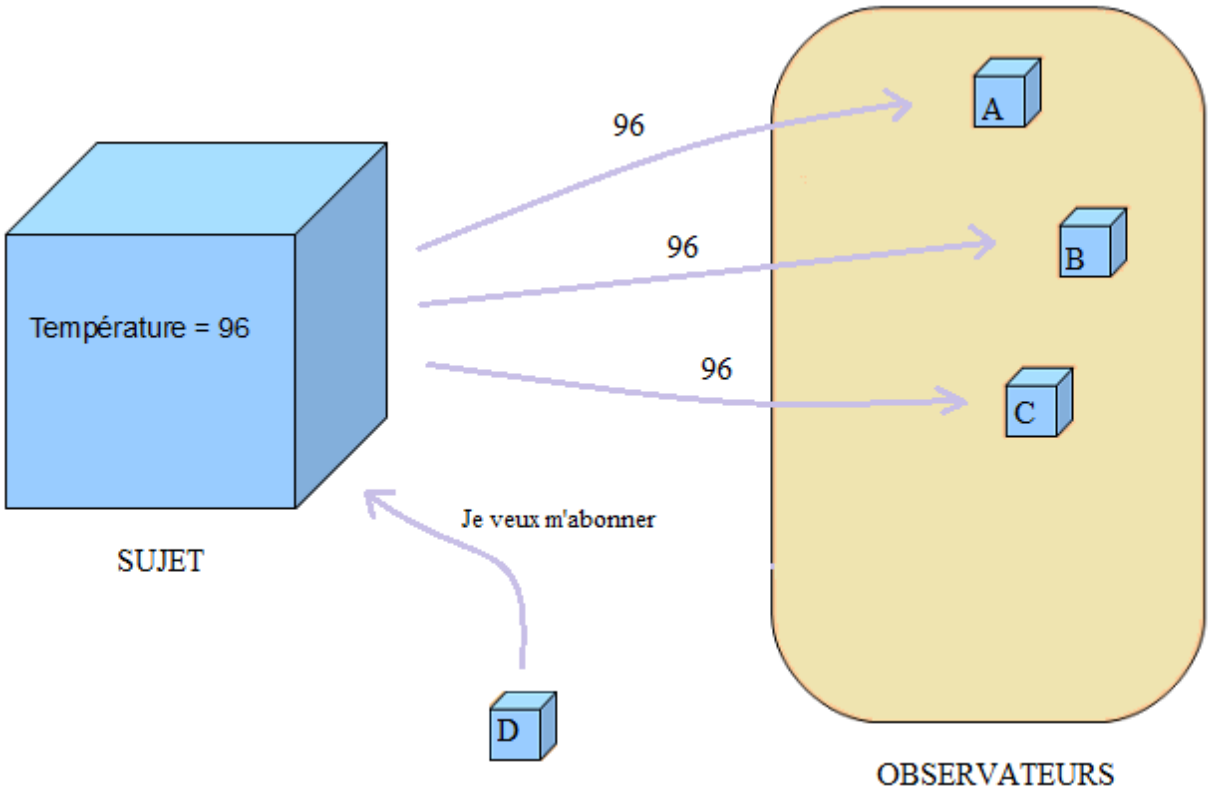
Le pattern Observer fonctionne sur le même principe:

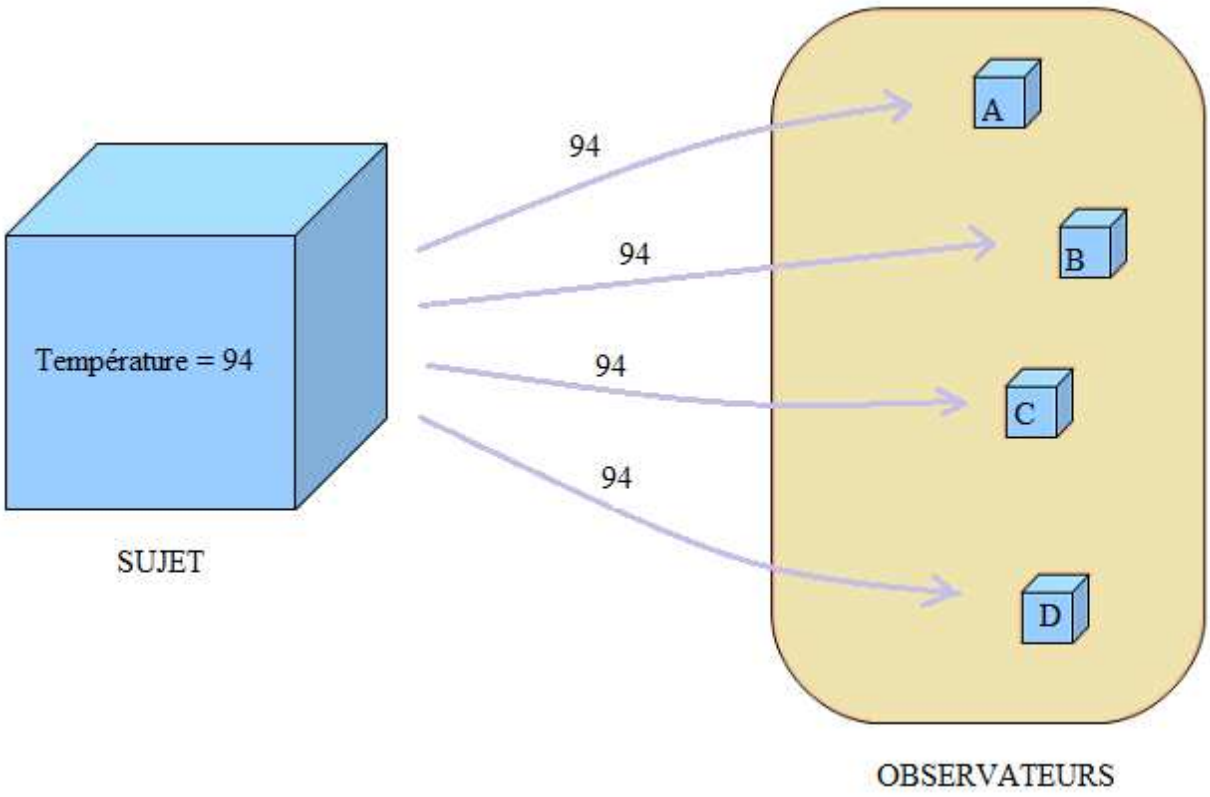
Une ressource est disponible via un abonnement. Cette ressource est appelée le Sujet.

Les personnes intéressées par cette ressource s'abonnent. Ces personnes sont appelées les Observateurs (Observers en anglais).

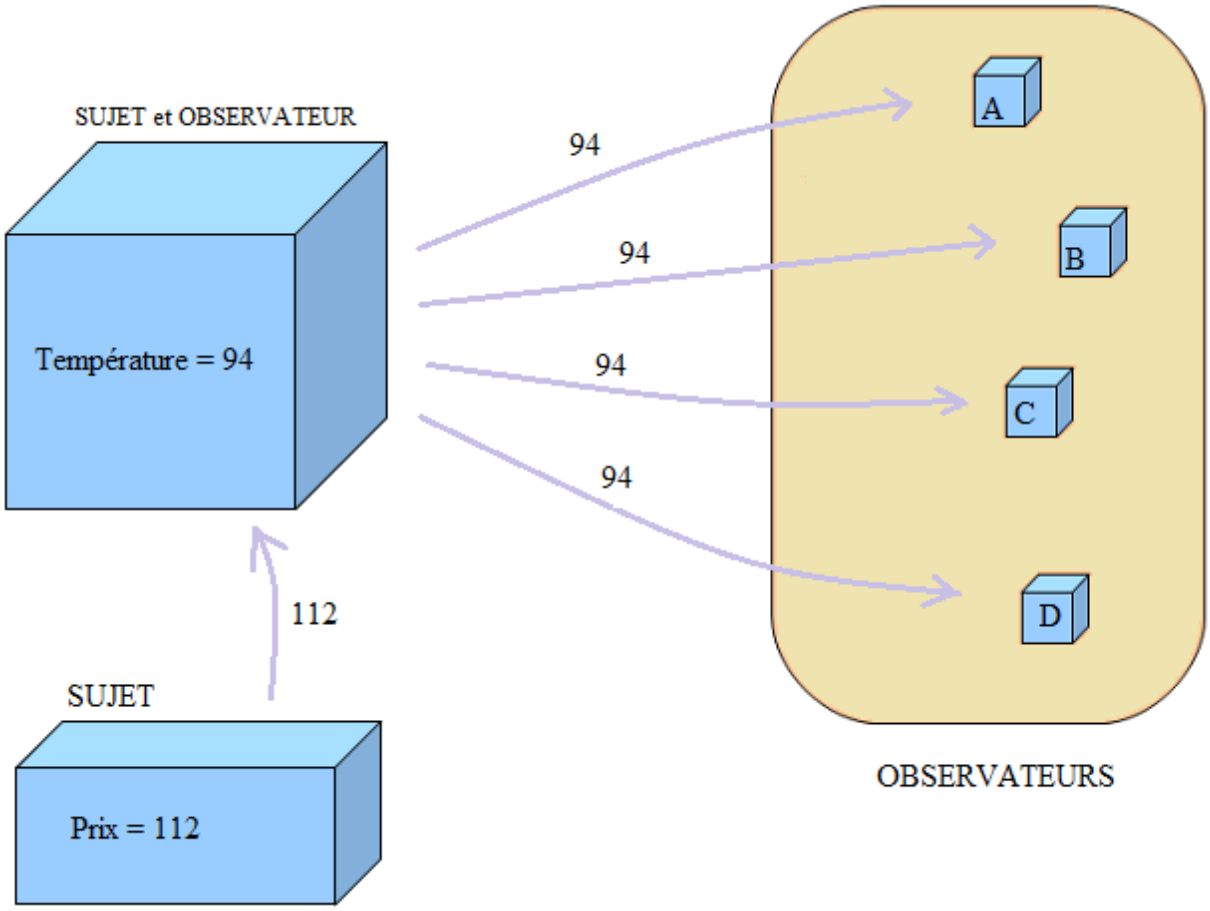
Tant que l'Observateur est abonné, il reçoit des notifications émises par le Sujet.

Tout Observateur peut mettre fin à son abonnement. Il cesse alors de recevoir les notifications émises par le Sujet.


Température = 96
SUJET
96
96
96
A
B
C
Je veux m'abonner
D
OBSERVATEURS

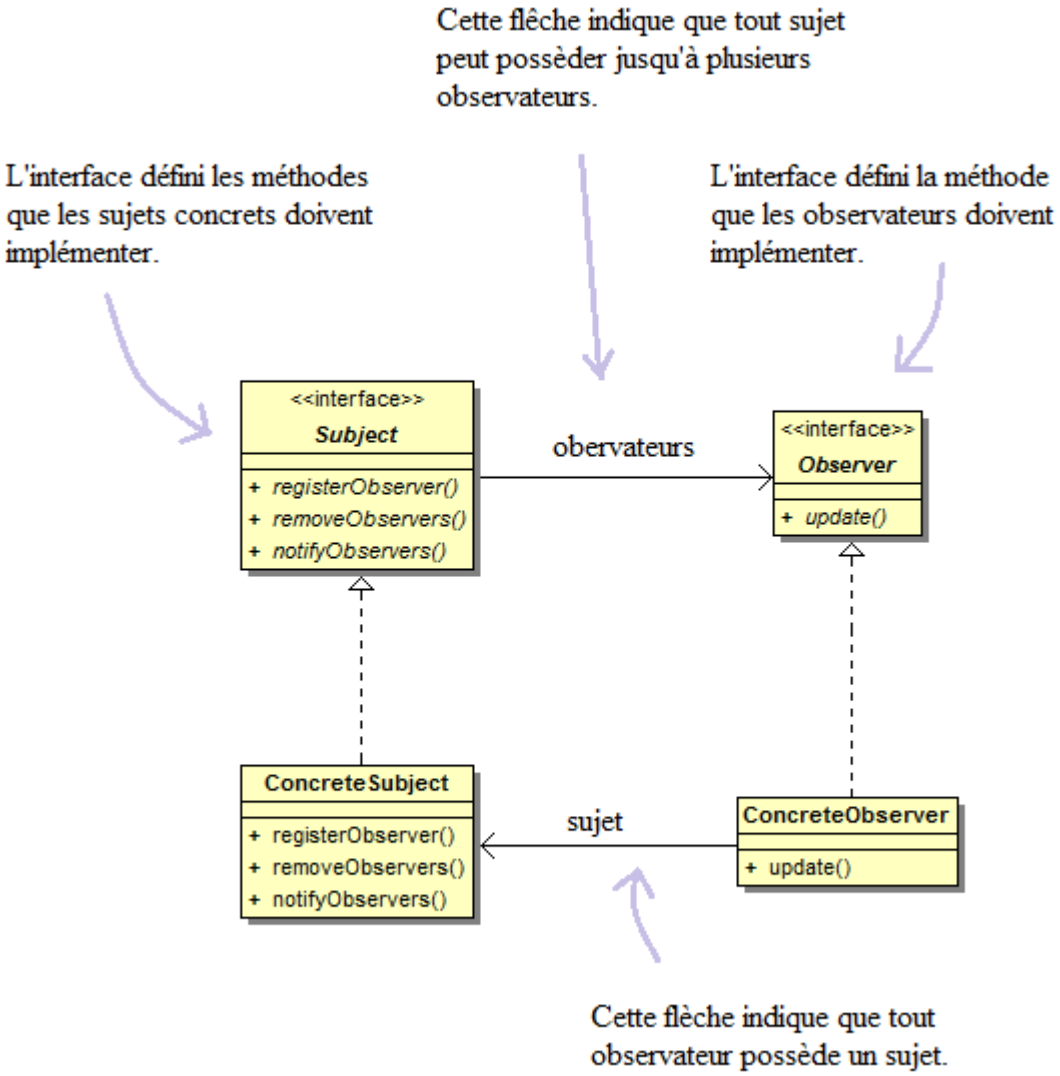

Température = 94
SUJET
94
94
94
94
A
B
C
D
OBSERVATEURS

Un Sujet peut également être lui-même Observateur d'un autre Sujet :



3.1.1 Diagramme de classe

Voici le diagramme de classe standard du pattern Observer:



3.2 Implémentation

On y arrive Mr Scott.

Notre Sujet doit gérer les informations qui intéressent les Observateurs:

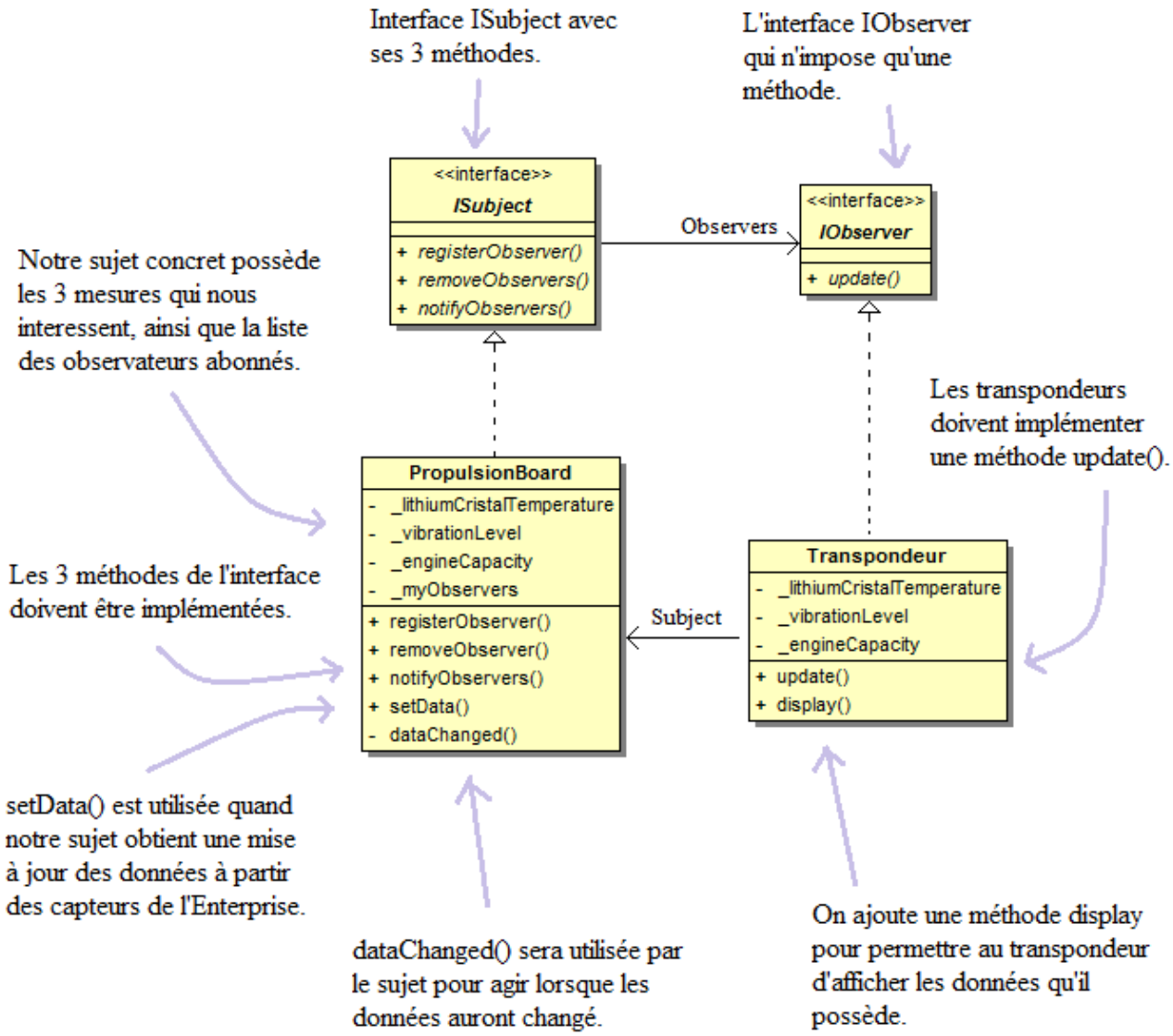
- Température du cristal de lithium.
- Niveau de vibrations du système de propulsion.
- Régime du système de propulsion.

Le Sujet obtient ces informations directement des différents capteurs de l'Enterprise. Peu importe comment il les obtient, l'important est qu'il les possède, et qu'elles sont constamment mises à jour.

Le sujet concret est donc comme un tableau de bord du système de propulsion de l'Enterprise. Par conséquent nommons le "PropulsionBoard".

Les transpondeurs sont les observateurs concrets. Ils doivent pouvoir s'abonner et de désabonner auprès du sujet.

Ceci nous donne le diagramme de classe suivant:


Interface ISubject avec ses 3 méthodes.
L'interface IObserver qui n'impose qu'une méthode.
<<interface>>
ISubject
+ registerObserver()
+ removeObservers()
+ notifyObservers()
Observers
<<interface>>
IObserver
+ update()
Notre sujet concret possède les 3 mesures qui nous interessent, ainsi que la liste des observateurs abonnés.
Les transpondeurs doivent implémenter une méthode update().
PropulsionBoard
- _lithiumCristalTemperature
- _vibrationLevel
- _engineCapacity
- _myObservers
+ registerObserver()
+ removeObserver()
+ notifyObservers()
+ setData()
- dataChanged()
Subject
Transpondeur
- _lithiumCristalTemperature
- _vibrationLevel
- _engineCapacity
+ update()
+ display()
Les 3 méthodes de l'interface doivent être implémentées.
setData() est utilisée quand notre sujet obtient une mise à jour des données à partir des capteurs de l'Enterprise.
dataChanged() sera utilisée par le sujet pour agir lorsque les données auront changé.
On ajoute une méthode display pour permettre au transpondeur d'afficher les données qu'il possède.

3.2.1 Codage

Les interfaces:

```php
<?php

interface ISubject {

  public function registerObserver($oObserver);
  public function removeObserver($oObserver);
  public function notifyObservers();

}

?>
```

```php
<?php

interface IObserver {

  public function update($lithiumCristalTemperature, $vibrationLevel,
                         $engineCapacity);

}

?>
```

La classe Transpondeur:

```php
<?php

class Transpondeur implements IObserver {

  private $_lithiumCristalTemperature;
  private $_vibrationLevel;
  private $_engineCapacity;
  private $_mySubject;

  public function  __construct($theSubject) {
        $this->_mySubject = $theSubject;
        $this->_mySubject->registerObserver($this);
    }

  public function update($lithiumCristalTemperature, $vibrationLevel,
                         $EngineCapacity)
  {
        $this->_lithiumCristalTemperature = $lithiumCristalTemperature;
        $this->_vibrationLevel = $vibrationLevel;
        $this->_engineCapacity = $engineCapacity;
        $this->display();
  }

  public function display()
  {
        echo get_class() . ": display(): </br>";
        echo "   Lithium cristal temperature:
                        $this->_lithiumCristalTemperature </br>";
        echo "   Vibration level: $this->_vibrationLevel </br>";
        echo "   Engine capacity: $this->_engineCapacity </br>";
  }

} // end class

?>
```

On implémente l'interface IObserver.

Le constructeur effectue l'enregistrement auprès du Sujet.

La classe PropulsionBoard:

```php
<?php

class PropulsionBoard implements ISubject {

  private $_lithiumCristalTemperature;
  private $_vibrationLevel;
  private $_engineCapacity;
  private $_myObservers = array();

  public function registerObserver($oObserver)
  {
      $this->_myObservers[] = $oObserver;
      echo get_class() . ": Un observer de plus.</br>";
  }

  public function removeObserver($oObserver)
  {
        foreach (array_keys($this->_myObservers) as $key) {
            if($this->_myObservers[$key] === $oObserver){
                unset($this->_myObservers[$key]); // delete array cell.
            }
        } // end foreach
  }

  public function notifyObservers()
  {
        foreach (array_keys($this->_myObservers) as $key) {
            $observer = $this->_myObservers[$key];
            $observer->update($this->_lithiumCristalTemperature,
                              $this->_vibrationLevel,
                              $this->_engineCapacity);
        } // end foreach
  }

  public function setData($lithiumCristalTemperature, $vibrationLevel,
                          $EngineCapacity)
  {
        $this->_lithiumCristalTemperature = $lithiumCristalTemperature;
        $this->_vibrationLevel = $vibrationLevel;
        $this->_engineCapacity = $engineCapacity;
        $this->dataChanged();
  }

  private function dataChanged()
  {
     $this->notifyObservers();
  }

} // end class

?>
```

Les observateurs sont gérés dans un tableau.

Enregistrer un observateur revient à l'ajouter au tableau.

Ici on supprime un observateur.

On notifie l'ensemble des observateurs.

C'est ici que PropulsionBoard reçoit les mises à jour des données.

Nous utiliserons index.php comme contrôleur de l'ensemble de notre pattern.

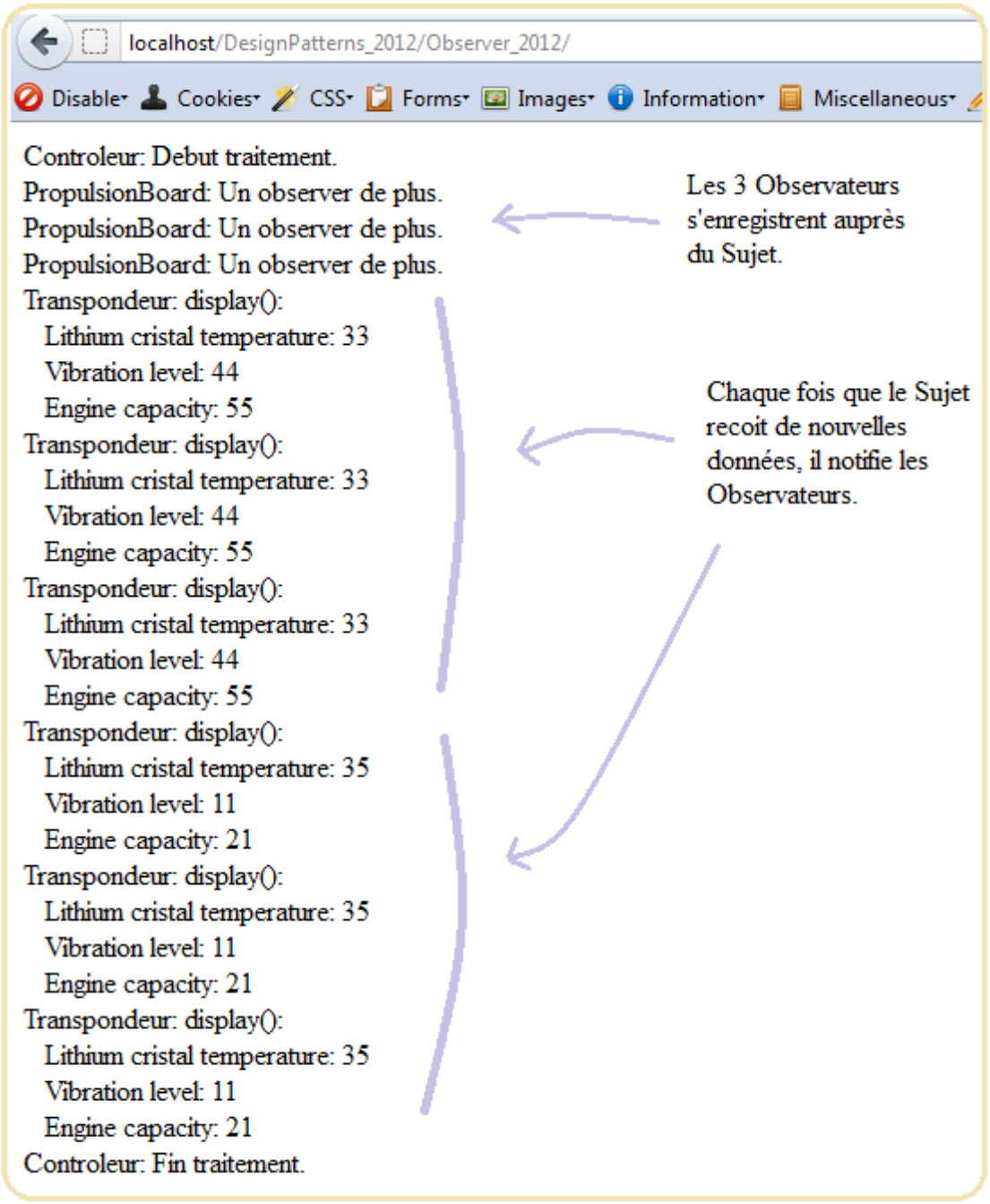
Le contrôleur va instancier des objets et les utiliser.

```php
<?php

//*************************************************************
// OBSERVER pattern controler
//*************************************************************

require_once 'includePaths.php';
$newline = "</br>";

echo 'Controleur: Debut traitement.' . $newline;

// Instanciations:
$myPropulsionBoard = new PropulsionBoard();
$Transpondeur1 = new Transpondeur($myPropulsionBoard);
$Transpondeur2 = new Transpondeur($myPropulsionBoard);
$Transpondeur2 = new Transpondeur($myPropulsionBoard);

// traitemment:
$myPropulsionBoard->setData(33,44,55);
$myPropulsionBoard->setData(35,11,21);

echo 'Controleur: Fin traitement.' . $newline;

?>
```

On instancie 1 Sujet et 3 Observateurs.

L'envoi de nouvelles données au Sujet, déclenchera l'envoi de notifications vers les Observateurs.

Le résultat de l'exécution du contrôleur doit ressembler à ceci:



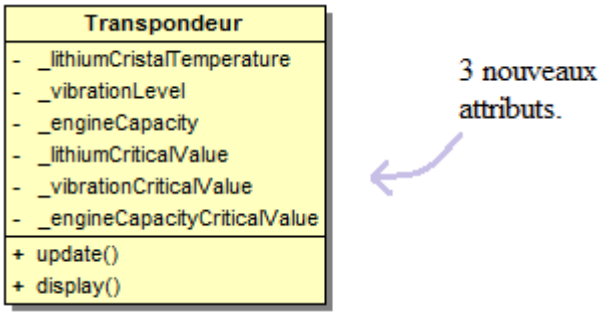
Vous avez absolument raison Mr Scott. Nous allons corriger cela tout de suite.

La détection du dépassement des valeurs critiques est une responsabilité qui revient au Transpondeur. Ce dernier doit donc connaître ces valeurs critiques pour être en mesure de réagir quand elles sont atteintes ou dépassées.

Ajoutons donc dans la classe Transpondeur, les attributs permettant de stocker les valeurs critiques.

Ces valeurs seront transmises aux Transpondeurs lors de leur instanciation.

Nous modifierons également la méthode display() pour adapter l'affichage du Transpondeur lorsque les valeurs sont supérieures ou égales à leurs valeurs critiques.



```php
<?php

class Transpondeur implements IObserver {

  private $_lithiumCristalTemperature;
  private $_vibrationLevel;
  private $_engineCapacity;
  private $_lithiumCriticalValue;
  private $_vibrationCriticalValue;
  private $_engineCapacityCriticalValue;
  private $_mySubject;

  public function __construct($theSubject, $lithiumCriticalValue,
                $vibrationCriticalValue, $engineCapacityCriticalValue) {
        $this->_mySubject = $theSubject;
        $this->_lithiumCriticalValue = $lithiumCriticalValue;
        $this->_vibrationCriticalValue = $vibrationCriticalValue;
        $this->_engineCapacityCriticalValue = $engineCapacityCriticalValue;
        $this->_mySubject->registerObserver($this);
    }

  public function update($lithiumCristalTemperature, $vibrationLevel,
                         $engineCapacity)
  {
        $this->_lithiumCristalTemperature = $lithiumCristalTemperature;
        $this->_vibrationLevel = $vibrationLevel;
        $this->_engineCapacity = $engineCapacity;
        $this->display();
  }

  public function display()
  {
        echo get_class() . ": display(): </br>  ";
        if ($this->_lithiumCristalTemperature >= $this->_lithiumCriticalValue)
            echo "ALERT: ";
        }
        echo "Lithium cristal temperature:
                    $this->_lithiumCristalTemperature </br>  ";

        if ($this->_vibrationLevel >= $this->_vibrationCriticalValue) {
            echo "ALERT: ";
        }
        echo "Vibration level: $this->_vibrationLevel </br>  ";

        if ($this->_engineCapacity >= $this->_engineCapacityCriticalValue) {
            echo "ALERT: ";
        }
        echo "Engine capacity: $this->_engineCapacity </br>";
  }

} // end class

?>
```

Les nouveaux attributs sont initialisés par le constructeur.

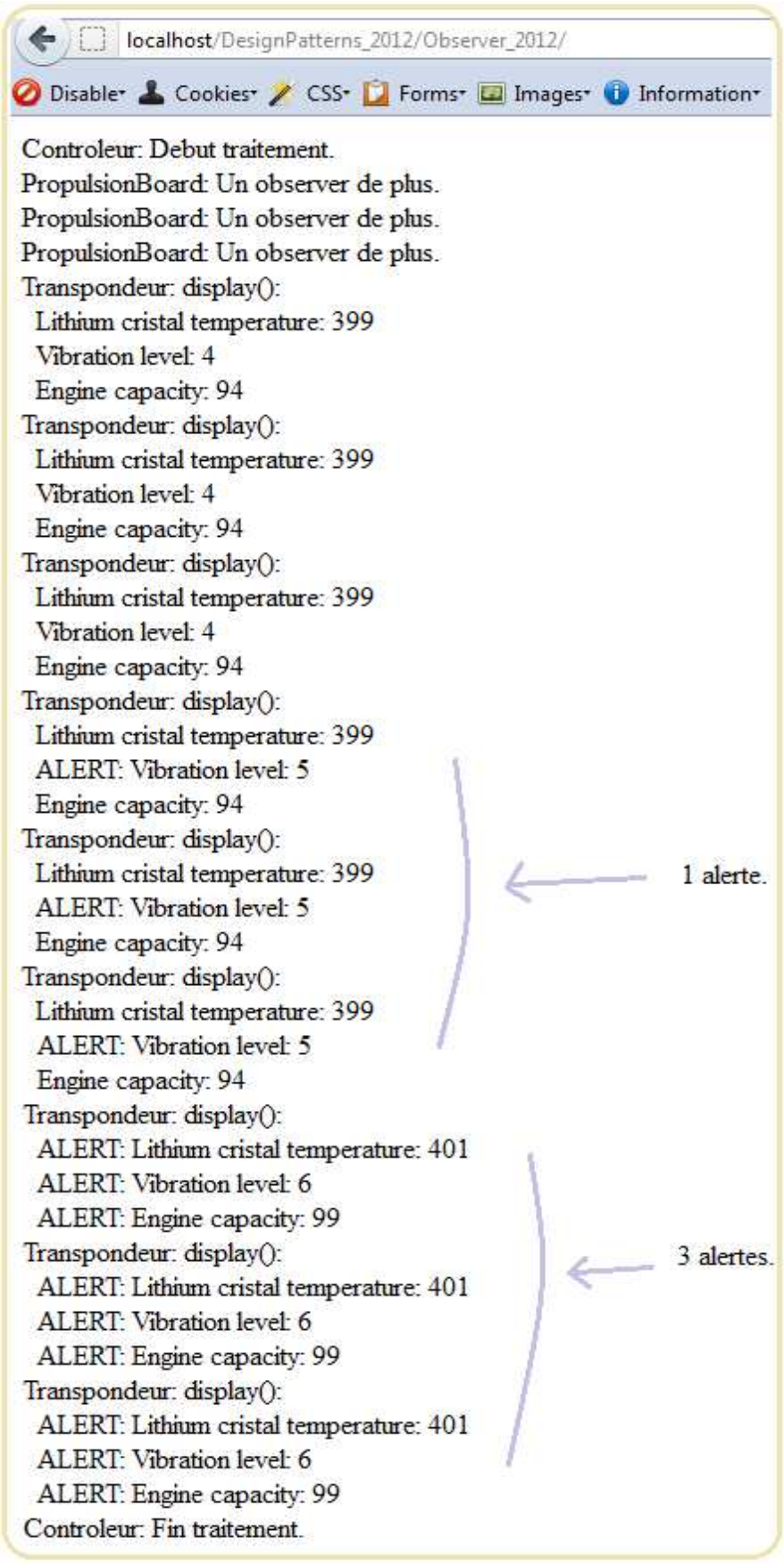
La méthode display() sait quand il faut lancer une alerte.

Enfin nous devons modifier notre contrôleur :

```php
<?php

//************************************************************
// OBSERVER pattern controler
//************************************************************

require_once 'includePaths.php';
$newline = "</br>";

echo 'Controleur: Debut traitement.' . $newline;

// Critical values parameters:
$lithiumCriticalValue = 400;
$vibrationCriticalValue = 5;
$engineCapacityCriticalValue = 95;


// Instanciations:
$myPropulsionBoard = new PropulsionBoard();
$Transpondeur1 = new Transpondeur($myPropulsionBoard,$lithiumCriticalValue,
                        $vibrationCriticalValue,$engineCapacityCriticalValue);
$Transpondeur2 = new Transpondeur($myPropulsionBoard,$lithiumCriticalValue,
                        $vibrationCriticalValue,$engineCapacityCriticalValue);
$Transpondeur2 = new Transpondeur($myPropulsionBoard,$lithiumCriticalValue,
                        $vibrationCriticalValue,$engineCapacityCriticalValue);

// traitemment:
$myPropulsionBoard->setData(399,4,94);  // pas d'alertes.
$myPropulsionBoard->setData(399,5,94);  // 1 alerte.
$myPropulsionBoard->setData(401,6,99);  // 3 alertes

echo 'Controleur: Fin traitement.' . $newline;

?>
```

Le controleur connait les valeurs critiques, et les transmet aux Observateurs lors de leur instanciation.

Voyons l’exécution du contrôleur:


localhost/DesignPatterns_2012/Observer_2012/
Disable Cookies CSS Forms Images Information
Controleur: Debut traitement.
PropulsionBoard: Un observer de plus.
PropulsionBoard: Un observer de plus.
PropulsionBoard: Un observer de plus.
Transpondeur: display():
Lithium cristal temperature: 399
Vibration level: 4
Engine capacity: 94
Transpondeur: display():
Lithium cristal temperature: 399
Vibration level: 4
Engine capacity: 94
Transpondeur: display():
Lithium cristal temperature: 399
Vibration level: 4
Engine capacity: 94
Transpondeur: display():
Lithium cristal temperature: 399
ALERT: Vibration level: 5
Engine capacity: 94
Transpondeur: display():
Lithium cristal temperature: 399
ALERT: Vibration level: 5
Engine capacity: 94
Transpondeur: display():
Lithium cristal temperature: 399
ALERT: Vibration level: 5
Engine capacity: 94
1 alerte.
Transpondeur: display():
ALERT: Lithium cristal temperature: 401
ALERT: Vibration level: 6
ALERT: Engine capacity: 99
Transpondeur: display():
ALERT: Lithium cristal temperature: 401
ALERT: Vibration level: 6
ALERT: Engine capacity: 99
Transpondeur: display():
ALERT: Lithium cristal temperature: 401
ALERT: Vibration level: 6
ALERT: Engine capacity: 99
3 alertes.
Controleur: Fin traitement.

Dans notre montage, le transpondeur est le seul observateur du sujet PropulsionBoard. Mais imaginons qu'on nous demande un serveur vocal qui égrène les valeurs de PropulsionBoard dans un haut-parleur.

Comme les transpondeurs, cette nouvelle classe devrait implémenter l'interface IObserver et s'abonner à PropulsionBoard. Elle aurait simplement une utilisation différente des données obtenues.

4 PATTERN DECORATOR

L'Enterprise effectue régulièrement des missions qui consistent à envoyer des équipes et du matériel à l'extérieur du vaisseau pour exécuter une tache.

Il peut s'agir de missions d'exploration, de contrôle, de ravitaillement, d'évacuation...

Le capitaine Kirk souhaite disposer d'un système permettant de définir les constituants d'une mission (équipes et unités matérielles) et de connaître le poids de ces constituants, ainsi que le poids total et le nombre de personnes.

Les équipes sont les suivantes:

- Groupe médical (4 personnes).
- Groupe de sécurité (6 personnes).
- Groupe scientifique (3 personnes).

Les unités matérielles sont les suivantes:

- Unité matérielle médicale légère.
- Unité matérielle médicale lourde.
- Unité matérielle défensive (armement).
- Unité matérielle logistique (abris, couchage, restauration)

Une mission est constituée d'une combinaison quelconque d'équipes et d'unités matérielles.

Exemple 1:

1 groupe médical.

1 unité matérielle médicale lourde.

2 groupes de sécurité.

Exemple 2:

1 unité matérielle logistique.

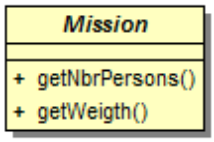
2 groupes scientifiques.

Au travail !

La difficulté ici est qu'on doit pouvoir à la fois:

- "composer" une mission avec un ensemble d'éléments que sont les équipes et les unités matérielles,.
- Interroger chaque élément, notamment pour obtenir son poids ou le nombre de personnes.
- Interroger la mission dans sa globalité, pour obtenir le poids total, et le nombre total de personnes.

On pourrait imaginer une classe abstraite Mission, dont dériveraient un ensemble de classes représentant chacune une combinaison d'éléments:



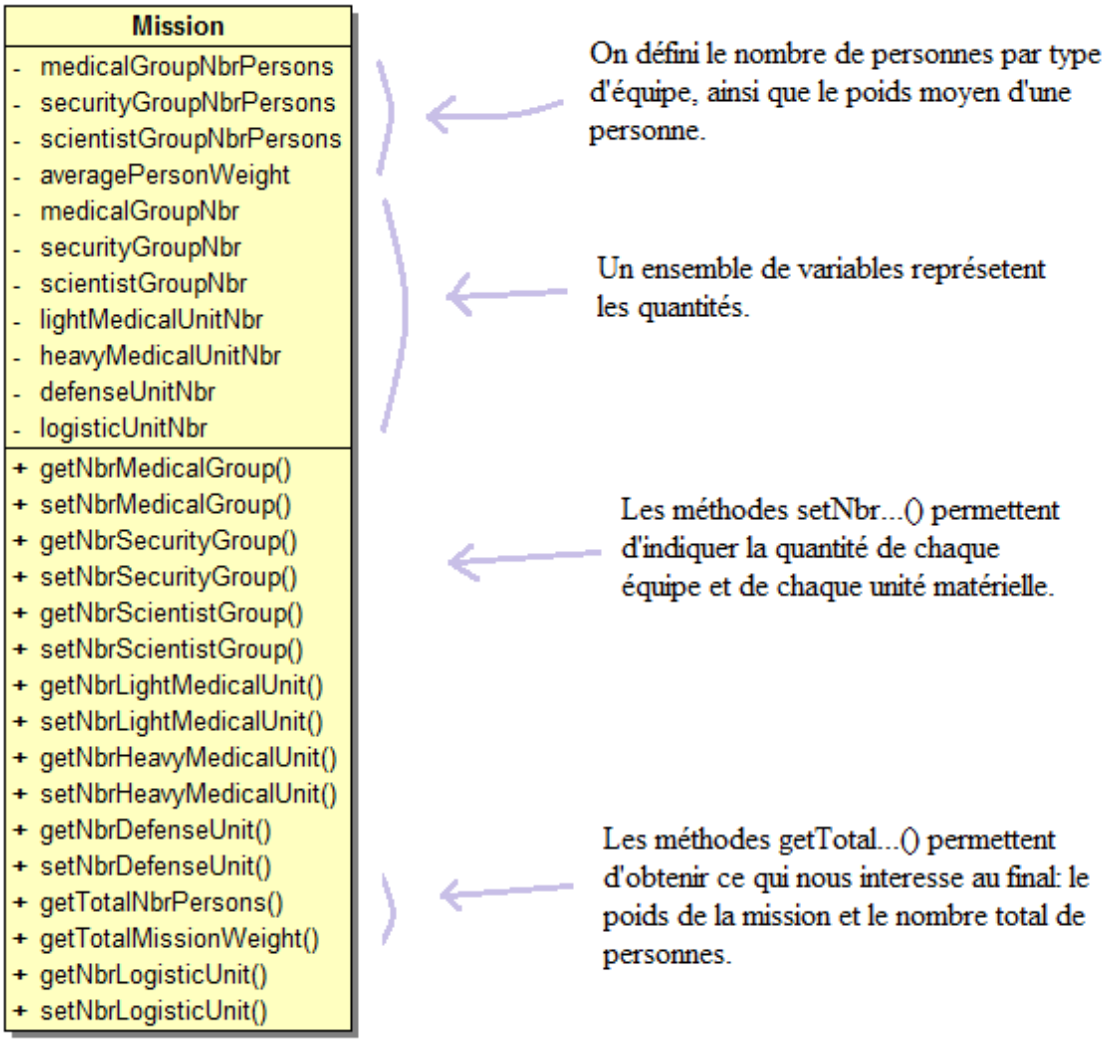
Mais sachant:

- qu'on a 3 types d'équipes et 4 types d'unités matérielles
- que, pour une mission, chacun de ces éléments peut être doublé, triplé (il n'y a en fait pas de limite).

Le nombre de classes à créer serait virtuellement illimité. Cette approche est donc définitivement écartée.

OK ce n'était pas une bonne idée. Essayons quelque chose de plus simple:

On pourrait garder la classe Mission, mais la composition de la mission serait représentée par des attributs qui indiqueraient leurs quantités respectives:



On obtient une classe unique qui contient la totalité des informations nécessaires à une mission.

Cette approche est envisageable mais présente des inconvénients:

- La classe est grosse et regroupe des informations de natures différentes.
- L'ajout d'un nouveau type d'équipe ou d'unité matérielle nécessite de modifier la classe.
- Une mission n'utilisant pas un certain constituant, implémentera tout de même les méthodes concernant ce constituant.

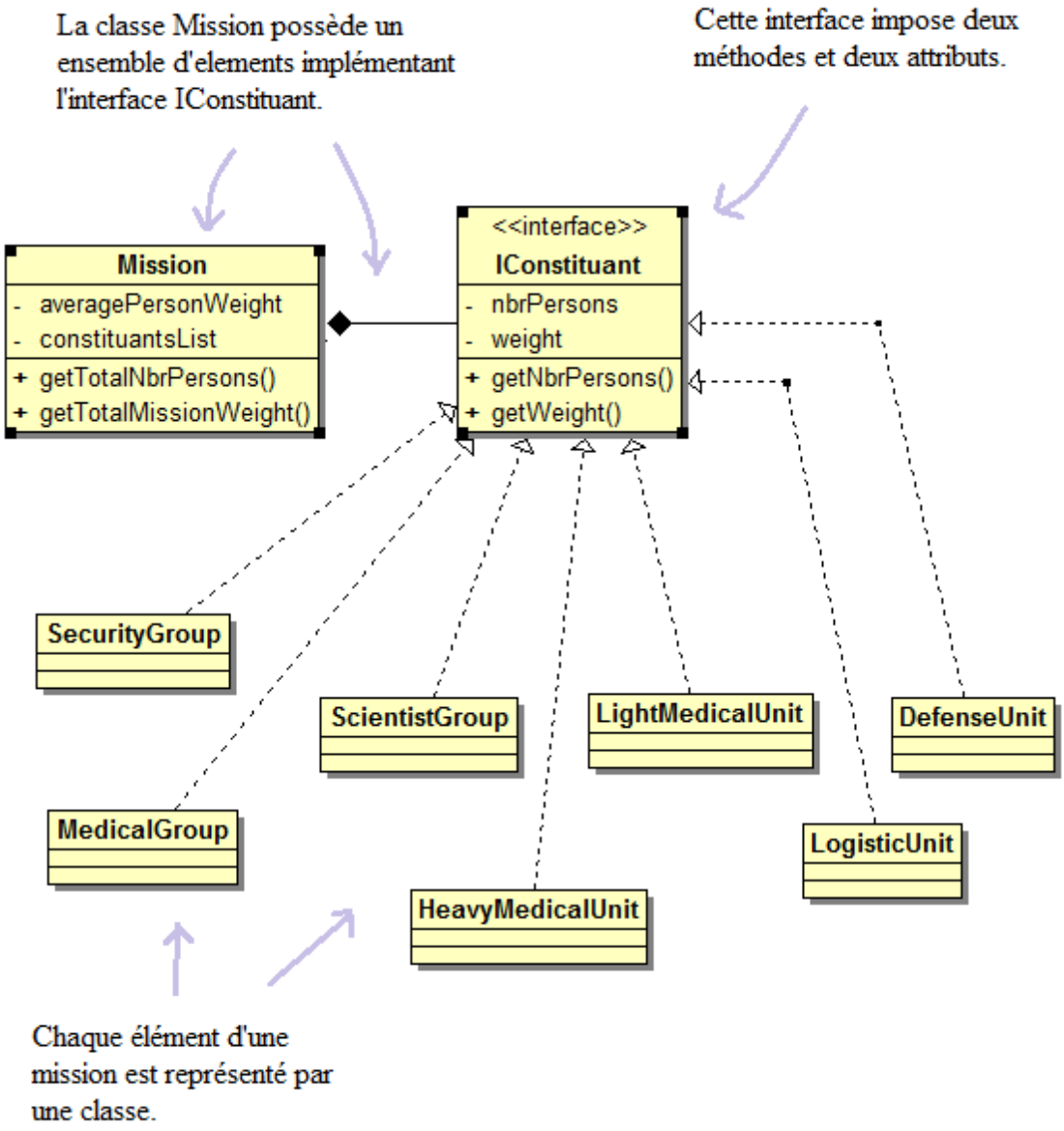
En conclusion, on peut faire mieux.

Une dernière idée ?

On pourrait garder une classe Mission allégée, et créer une classe pour chaque constituant d'une mission (équipes et unités matérielles).

Ensuite on attache les constituants à la mission, en utilisant dans la classe Mission, un ensemble de références sous forme d'une liste, qui permet à la Mission de connaître ses constituants et de les

interroger.



Cette approche est plus élaborée:

- La mission est séparée de ses éléments constitutifs.
- Les éléments constitutifs sont masqués derrière une interface, de telle sorte que la mission ne voit que des IConstituants.

Le seul petit inconvénient est que la mission doit gérer la liste de ses constituants à travers son attribut constituantsList. Malgré cela cette approche pourrait être utilisée, mais on peut faire encore mieux grâce au pattern Decorator.

4.1 Cap sur le pattern Decorator

Le pattern Decorator ressemble aux poupées Russes: Les objets sont mis les uns dans les autres pour former au final une entité composite.

Prenons par exemple une mission composée des éléments suivants:

- Une équipe de sécurité (poids: 1,5 tonne).
- Une unité matérielle médicale légère (poids: 2 tonnes).
- Une unité matérielle défensive (poids: 3 tonnes).

L'encapsulation des objets se fera comme suit:

- Un objet mission (qui est l'objet de base, ou objet "décoré"), encapsulé dans un objet Equipe de sécurité
- Le tout est encapsulé dans une Unité matérielle médicale légère.
- Le tout est encapsulé dans une Unité matérielle défensive.

En fait l'ordre d'encapsulation n'a pas d'importance dans notre exemple, mais peut en avoir selon le problème résoudre.

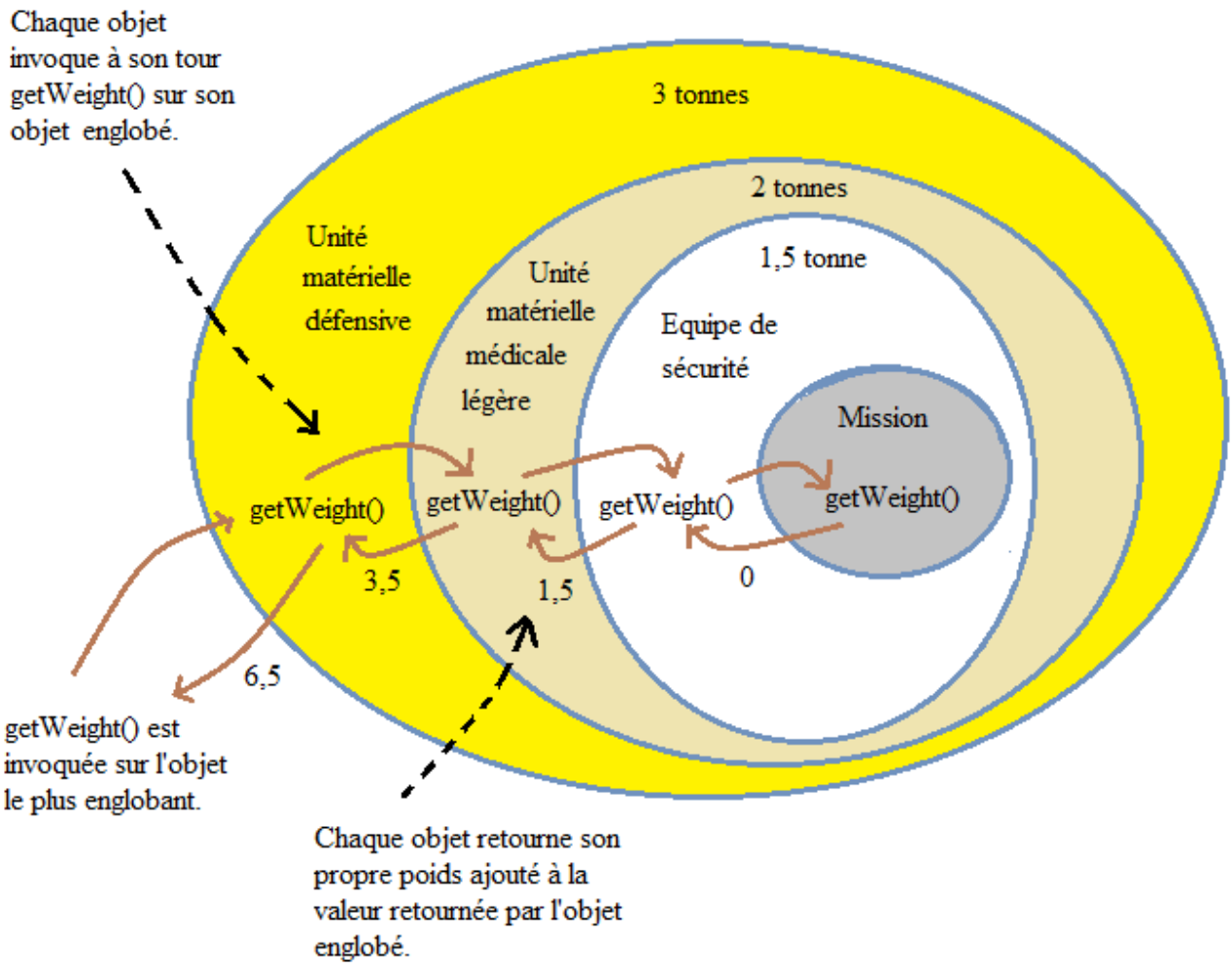
Mais ce n'est pas tout.

Il y a un second aspect important: la délégation.

Pour répondre à ces deux questions, utilisons les méthodes getNbrPersons() et getWeight. Tous les objets doivent implémenter ces méthodes.

Ces méthodes seront invoquées sur l'objet englobant, et seront automatiquement déléguées à l'objet englobé, et ainsi de suite jusqu'à l'objet le plus interne.

La réponse obtenue aura donc été complétée par chaque objet, pour former au final, une réponse globale à la mission.

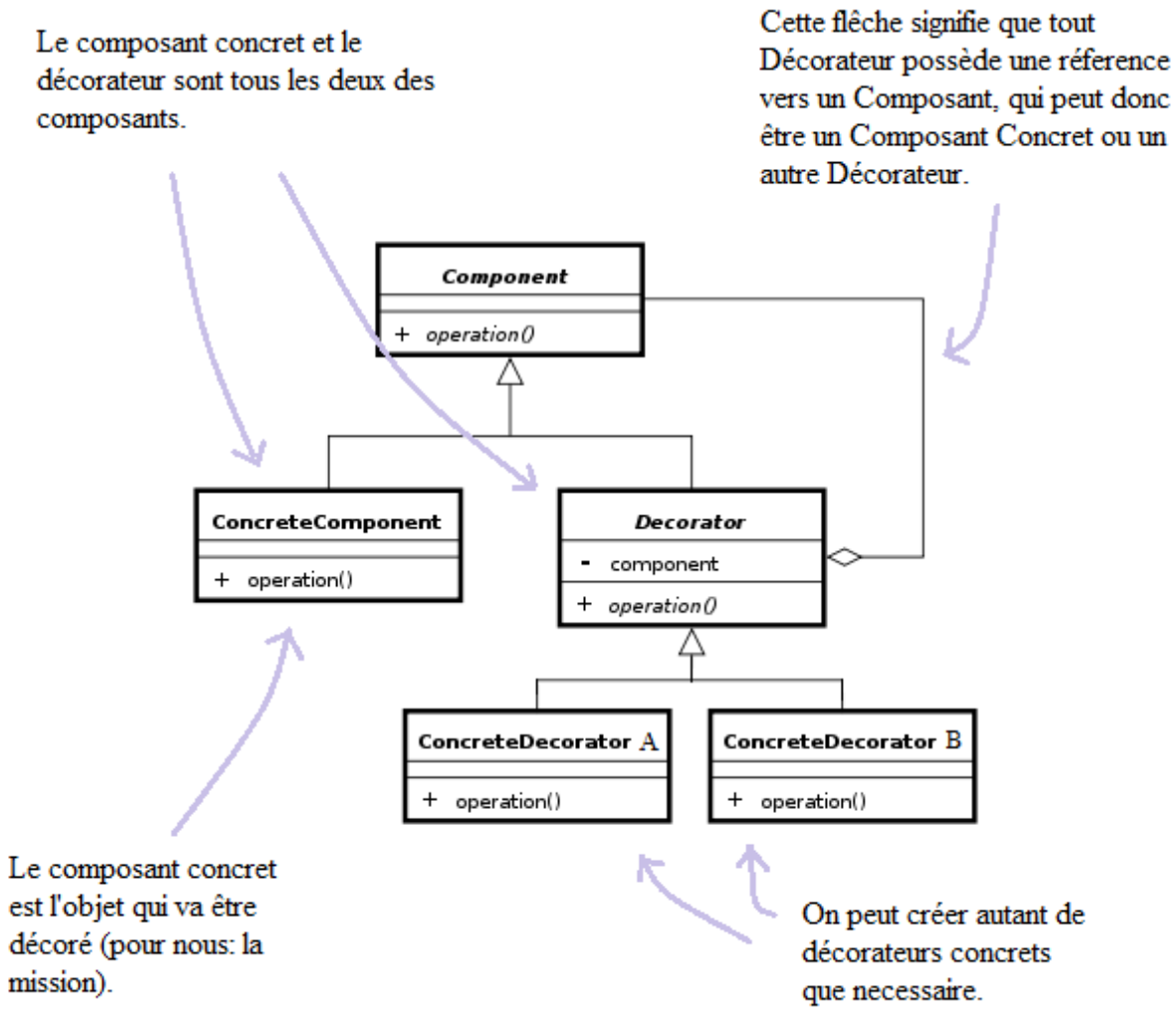


On implémentera de manière similaire une méthode getNbrPersons() pour obtenir le nombre total de personnes de la mission, et satisfaire ainsi les exigences du capitaine.

Cette façon d'appeler plusieurs fois la même méthode de manière imbriquée, est basée sur la récursivité (voir à ce sujet le chapitre: **La récursivité** en annexe).

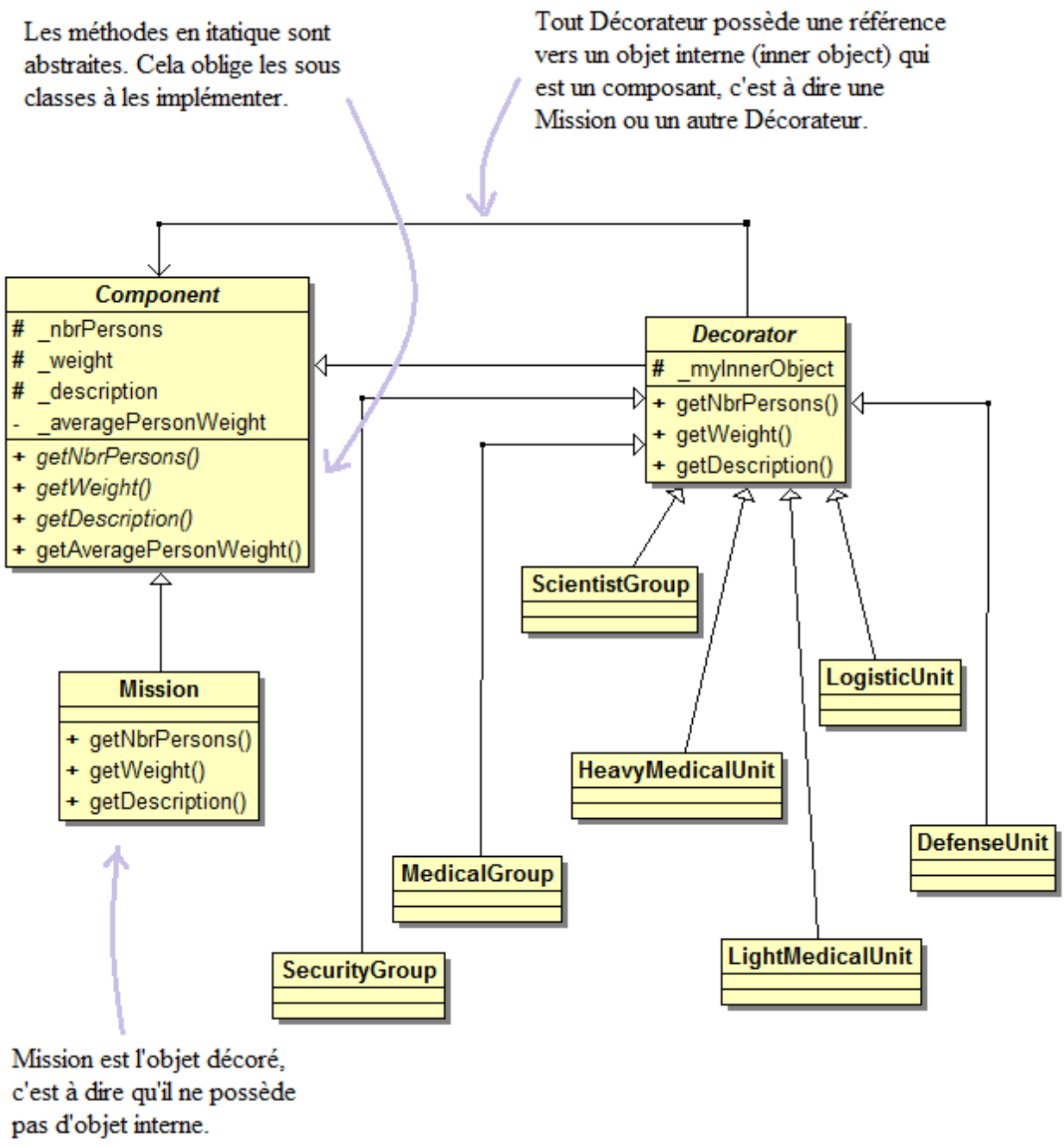
Cette technique très particulière est rarement utilisée. Elle apparaît également dans un autre pattern: Composite.

Examinons le modèle officiel du pattern Decorator:



4.2 Diagramme de classes

Nous avons tous les éléments pour définir le modèle de classes répondant au cahier des charges, en s'appuyant sur le pattern Decorator.


Les méthodes en itatique sont abstraites. Cela oblige les sous classes à les implémenter.
Tout Décorateur possède une référence vers un objet interne (inner object) qui est un composant, c'est à dire une Mission ou un autre Décorateur.
Component
_nbrPersons
_weight
_description
- _averagePersonWeight
+ getNbrPersons()
+ getWeight()
+ getDescription()
+ getAveragePersonWeight()
Decorator
_myInnerObject
+ getNbrPersons()
+ getWeight()
+ getDescription()
Mission
+ getNbrPersons()
+ getWeight()
+ getDescription()
ScientistGroup
LogisticUnit
HeavyMedicalUnit
DefenseUnit
MedicalGroup
SecurityGroup
LightMedicalUnit
Mission est l'objet décoré, c'est à dire qu'il ne possède pas d'objet interne.

4.3 Implémentation

Passons au codage des différentes classes. Le pattern Decorator est réputé pour générer de nombreuses petites classes. Effectivement il faut une classe concrète pour l'objet décoré, et une pour chaque décorateur.

4.3.1 Codage

La classe Component:

```php
<?php

abstract class Component {
  protected $_nbrPersons;
  protected $_weight;
  protected $_description;
  private $_averagePersonWeight;


  public function __construct()
  {
        $this->_averagePersonWeight = 62;

  }

  public abstract function getNbrPersons();

  public abstract function getWeight();

  public abstract function getDescription();

  public function getAveragePersonWeight()
  {
      return $this->_averagePersonWeight;
  }

}

?>
```

Attributs 'protected': hérités par les sous classes, mais de manière privée.

Méthodes abstraites: doivent être implémentées par les sous classes.

La classe Decorator:

```php
<?php

abstract class Decorator extends Component {

  protected $_myInnerObject;

  public function __construct($innerObject)
  {
      parent::__construct();
      $this->_myInnerObject = $innerObject;

  }

  public function getNbrPersons()
  {
     return $this->_nbrPersons
             + $this->_myInnerObject->getNbrPersons();
  }

  public function getWeight()
  {
     return $this->_weight + $this->_myInnerObject->getWeight();
  }

  public function getDescription()
  {
     return $this->_myInnerObject->getDescription() . '</br>'
             . '   ' .$this->_description;
  }

}
?>
```

On exécute le constructeur de la classe mère.

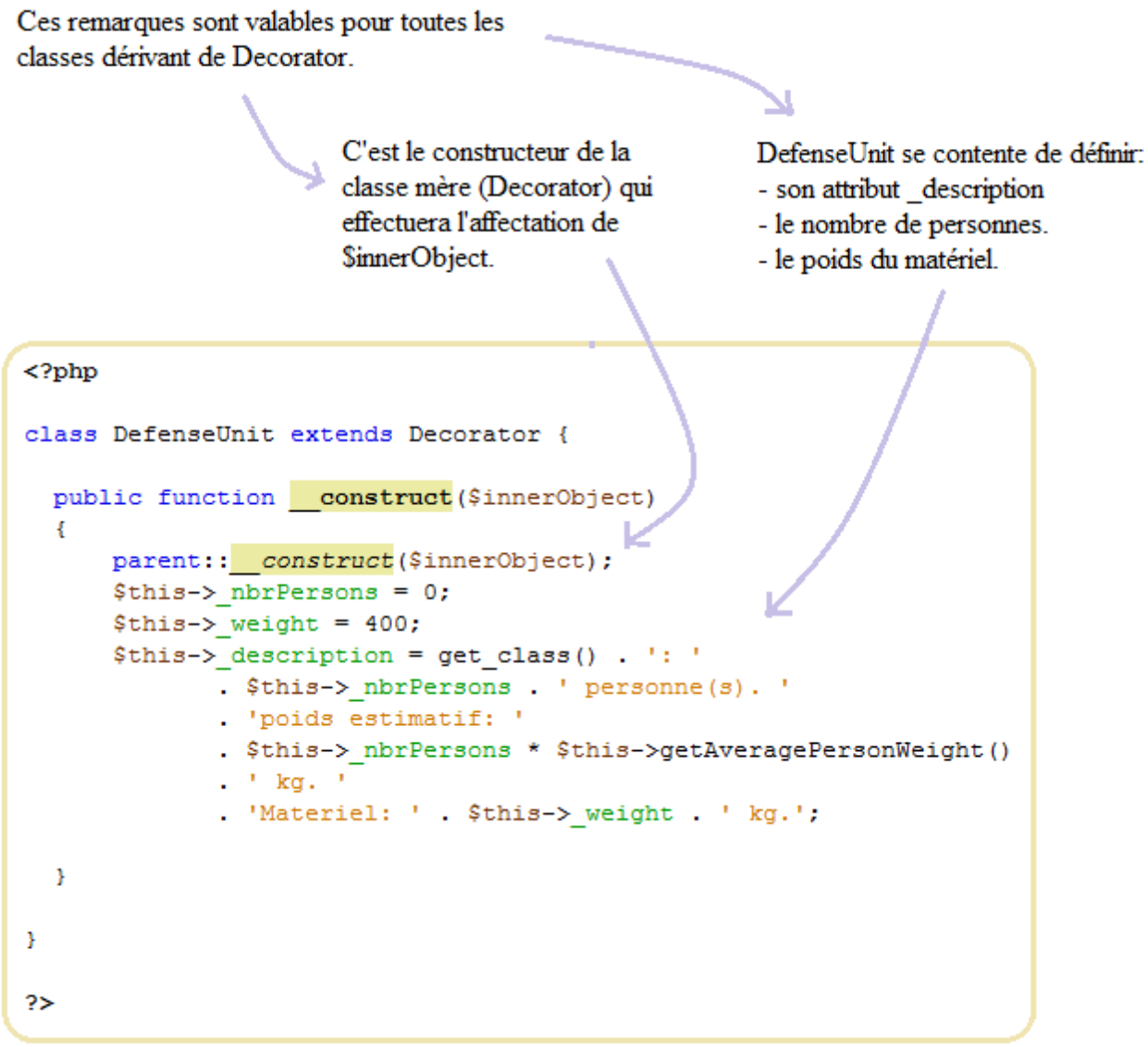
Les méthodes abstraites sont implémentées ici.

C'est ici que la récursivité entre en jeu: tout décorateur invoquera la méthode getxxxx() de son objet interne. Cette invocation sera propagée jusqu'à l'objet le plus interne: la Mission.

La classe Mission:

```php
<?php

class Mission extends Component {

  public function __construct($description)
  {
     parent::__construct();
     $this->_description = $description;
     $this->_nbrPersons = 1;
     $this->_weight = 0;
  }

  public function getNbrPersons()
  {
     return $this->_nbrPersons;
  }

  public function getWeight()
  {
     return $this->_weight;
  }

  public function getDescription() {
    return get_class() . ': ' . $this->_description . ' '
              . $this->_nbrPersons . ' chef(s) de mission. '
              . 'Poids estimatif: '
              . $this->_nbrPersons * $this->getAveragePersonWeight()
              . ' kg.';
  }

}

?>
```

La classe DefenseUnit :




```php
<?php

class DefenseUnit extends Decorator {

  public function __construct($innerObject)
  {
      parent::__construct($innerObject);
      $this->_nbrPersons = 0;
      $this->_weight = 400;
      $this->_description = get_class() . ': '
              . $this->_nbrPersons . ' personne(s). '
              . 'poids estimatif: '
              . $this->_nbrPersons * $this->getAveragePersonWeight()
              . ' kg. '
              . 'Materiel: ' . $this->_weight . ' kg.';

  }

}

?>
```

La classe HeavyMedicalUnit:

```php
<?php

class HeavyMedicalUnit extends Decorator {

  public function __construct($innerObject)
  {
      parent::__construct($innerObject);
      $this->_nbrPersons = 0;
      $this->_weight = 350;
      $this->_description = get_class() . ': '
             . $this->_nbrPersons . ' personne(s). '
             . 'poids estimatif: '
             . $this->_nbrPersons * $this->getAveragePersonWeight()
             . ' kg. '
             . 'Materiel: ' . $this->_weight . ' kg.';

  }

}

?>
```

La classe LightMedicalUnit:

```php
<?php

class LightMedicalUnit extends Decorator {

  public function __construct($innerObject)
  {
      parent::__construct($innerObject);
      $this->_nbrPersons = 0;
      $this->_weight = 230;
      $this->_description = get_class() . ': '
             . $this->_nbrPersons . ' personne(s). '
             . 'poids estimatif: '
             . $this->_nbrPersons * $this->getAveragePersonWeight()
             . ' kg. '
             . 'Materiel: ' . $this->_weight . ' kg.';

  }

}

?>
```

La classe LogisticUnit:

```php
<?php

class LogisticUnit extends Decorator {

  public function __construct($innerObject)
  {
      parent::__construct($innerObject);
      $this->_nbrPersons = 3;
      $this->_weight = 680;
      $this->_description = get_class() . ': '
             . $this->_nbrPersons . ' personne(s). '
             . 'poids estimatif: '
             . $this->_nbrPersons * $this->getAveragePersonWeight()
             . ' kg. '
             . 'Materiel: ' . $this->_weight . ' kg.';

  }

}

?>
```

La classe MedicalGroup:

```php
<?php

class MedicalGroup extends Decorator {

  public function __construct($innerObject)
  {
      parent::__construct($innerObject);
      $this->_nbrPersons = 6;
      $this->_weight = 100;
      $this->_description = get_class() . ': '
             . $this->_nbrPersons . ' personne(s). '
             . 'poids estimatif: '
             . $this->_nbrPersons * $this->getAveragePersonWeight()
             . ' kg. '
             . 'Materiel: ' . $this->_weight . ' kg.';

  }

}

?>
```

La classe ScientistGroup:

```php
<?php

class ScientistGroup extends Decorator {

  public function __construct($innerObject)
  {
      parent::__construct($innerObject);
      $this->_nbrPersons = 9;
      $this->_weight = 100;
      $this->_description = get_class() . ': '
             . $this->_nbrPersons . ' personne(s). '
             . 'poids estimatif: '
             . $this->_nbrPersons * $this->getAveragePersonWeight()
             . ' kg. '
             . 'Materiel: ' . $this->_weight . ' kg.';

  }

}

?>
```

La classe SecutityGroup:

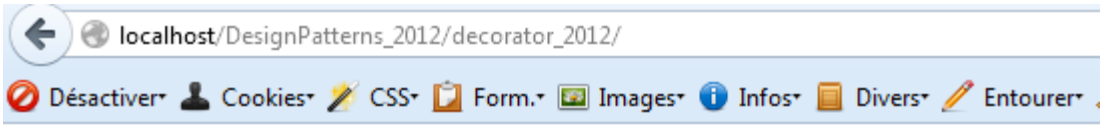
```php
<?php

class SecurityGroup extends Decorator {

  public function __construct($innerObject)
  {
      parent::__construct($innerObject);
      $this->_nbrPersons = 1;
      $this->_weight = 1200;
      $this->_description = get_class() . ': '
              . $this->_nbrPersons . ' personne(s). '
              . 'poids estimatif: '
              . $this->_nbrPersons * $this->getAveragePersonWeight()
              . ' kg. '
              . 'Materiel: ' . $this->_weight . ' kg.';
  }

}

?>
```

Nous utiliserons comme d'habitude index.php comme contrôleur de l'ensemble de notre pattern.

Le contrôleur va instancier des objets et les utiliser.

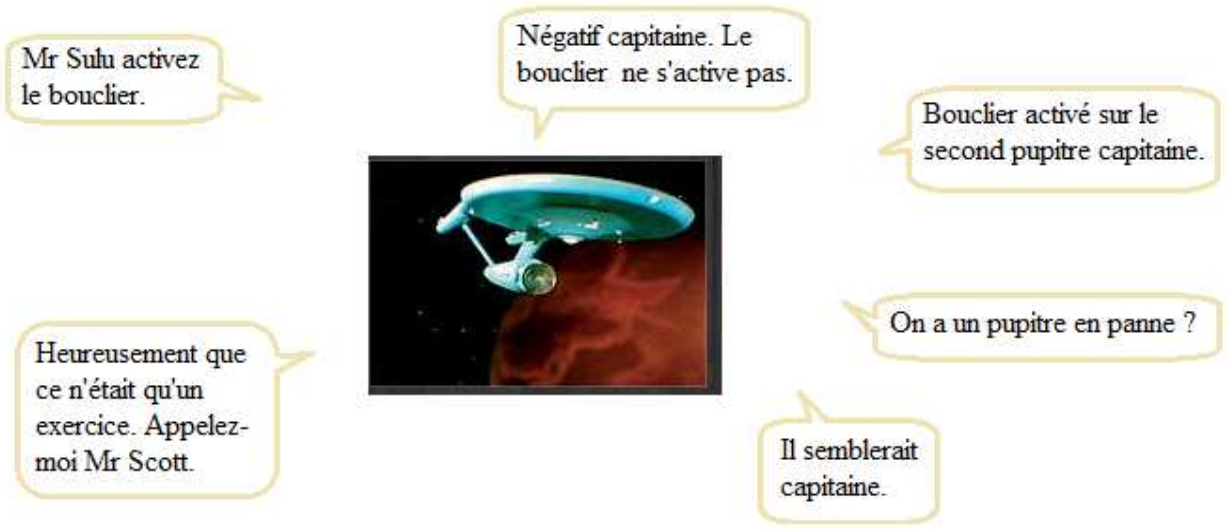
```php
<?php

//************************************************************
// DECORATOR pattern controler
//************************************************************

require_once 'includePaths.php';
$newline = "</br>";

echo 'Controleur: Debut traitement.' , $newline;

// Instanciations:
$Mission1 = new Mission("Exploration planète P731.");
$Mission1 = new DefenseUnit($Mission1);
$Mission1 = new MedicalGroup($Mission1);
$Mission1 = new SecurityGroup($Mission1);
$Mission1 = new SecurityGroup($Mission1);

$Mission2 = new Mission("Evacuation vaisseau Ravitailleur.");
$Mission2 = new MedicalGroup($Mission2);
$Mission2 = new ScientistGroup($Mission2);
$Mission2 = new SecurityGroup($Mission2);
$Mission2 = new HeavyMedicalUnit($Mission2);

// traitemment:
echo $Mission1->getDescription() , $newline;
echo 'Total mission:' , $newline;
echo 'Nbr personnes: ' , $Mission1->getNbrPersons() , $newline;
$TotalWeight = $Mission1->getWeight() + $Mission1->getNbrPersons()
        * $Mission1->getAveragePersonWeight();
echo 'Poids: ', $TotalWeight , ' kg.' , $newline;
echo '--------------' , $newline;

echo $Mission2->getDescription() , $newline;
echo 'Total mission:' , $newline;
echo 'Nbr personnes: ' , $Mission2->getNbrPersons() , $newline;
$TotalWeight = $Mission2->getWeight() + $Mission2->getNbrPersons()
        * $Mission2->getAveragePersonWeight();
echo 'Poids: ', $TotalWeight , ' kg.' , $newline;
echo '--------------' , $newline;

echo 'Controleur: Fin traitement.' , $newline;

?>
```

Le résultat de l'exécution doit ressembler à ceci:




```
Controleur: Debut traitement.
Mission: Exploration planete P731. 1 chef(s) de mission. Poids estimatif: 62 kg.
   DefenseUnit: 0 personne(s). poids estimatif: 0 kg. Materiel: 400 kg.
   MedicalGroup: 6 personne(s). poids estimatif: 372 kg. Materiel: 100 kg.
   SecurityGroup: 1 personne(s). poids estimatif: 62 kg. Materiel: 1200 kg.
   SecurityGroup: 1 personne(s). poids estimatif: 62 kg. Materiel: 1200 kg.
Total mission:
Nbr personnes: 9
Poids: 3458 kg.
--------------
Mission: Evacuation vaisseau ravitailleur. 1 chef(s) de mission. Poids estimatif: 62 kg.
   MedicalGroup: 6 personne(s). poids estimatif: 372 kg. Materiel: 100 kg.
   ScientistGroup: 9 personne(s). poids estimatif: 558 kg. Materiel: 100 kg.
   SecurityGroup: 1 personne(s). poids estimatif: 62 kg. Materiel: 1200 kg.
   HeavyMedicalUnit: 0 personne(s). poids estimatif: 0 kg. Materiel: 350 kg.
Total mission:
Nbr personnes: 17
Poids: 2804 kg.
--------------
Controleur: Fin traitement.
```

5 PATTERN SINGLETON



Le bouclier est une ressource unique qui peut être contrôlée par plusieurs pupitres. Dans ces conditions, l'état actuel du bouclier (ouvert ou fermé) doit être une information unique accessible par les pupitres. Il ne faut pas que cette information soit copiée car même avec un système de mise à jour des copies, il y a un risque pour qu'une d'entre elles ne soit pas en phase avec la réalité, à un instant t. Cela provoque évidemment des comportements imprévisibles.

Dans le cas présent nous allons faire en sorte que le bouclier soit bien une ressource unique, utilisable par plusieurs pupitres simultanément.

Pour cela nous allons utiliser le pattern Singleton qui est le plus simple des design patterns. Il se résume en effet à une seule classe.

5.1 Cap sur le pattern Singleton

Le but du pattern Singleton est de:

- Garantir qu'une classe n'est instanciée qu'une seule fois, c'est à dire qu'il ne peut y avoir que zéro ou un objet de cette classe.
- Fournir un point d'accès unique à cet objet.

Notre ressource unique est le bouclier de l'Enterprise (shield en anglais). Décidons maintenant que la classe du pattern Singleton s'appellera ShieldSingleton.

Il est clair que la classe ShieldSingleton doit contrôler le nombre d'instances d'elle-même. Il faut donc empêcher que n'importe qui puisse instancier des objets en faisant "new ShieldSingleton()".

Par conséquent la première caractéristique d'une classe de type Singleton, est d'avoir son constructeur privé, donc inaccessible de l'extérieur de la classe. Dans certains cas le constructeur sera même sans implémentation, c'est à dire vide.

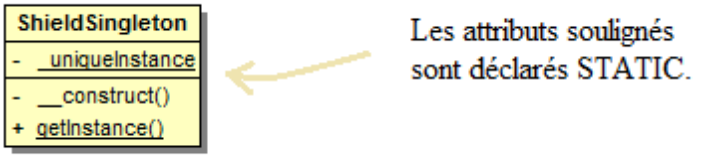
Mais dans ce cas comment instancier un objet de ShieldSingleton ?

C'est là la deuxième caractéristique d'une classe Singleton: elle doit fournir une méthode publique permettant d'obtenir l'instance unique de la classe. Mais même cette méthode doit être unique. Elle sera donc déclarée STATIC, c'est à dire qu'elle n'existera que dans la classe et pas dans l'objet instancié. Appelons cette méthode getInstance().

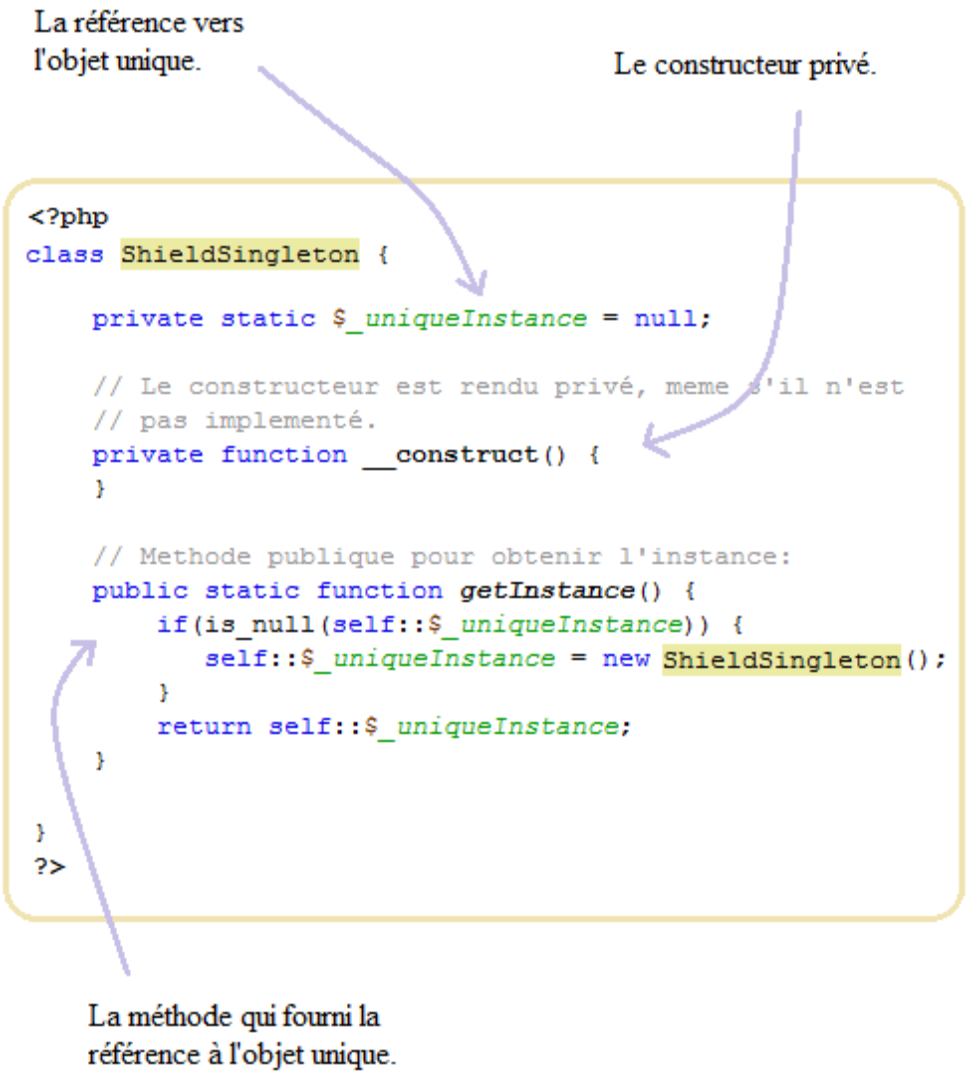
En résumé c'est donc la classe elle-même qui instancie l'unique objet, et qui en renvoi une référence via sa méthode publique getInstance(). La référence à l'objet unique sera matérialisée dans la classe ShieldSingleton grâce à un attribut lui aussi privé et statique.

Il est à noter que l'objet unique ne sera instancié qu'à la première invocation de getInstance().

Voici le modèle de classe correspondant:


ShieldSingleton
- _uniqueInstance
- __construct()
+ getInstance()
Les attributs soulignés
sont déclarés STATIC.

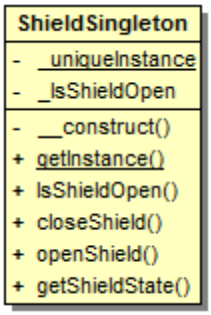

Elle est toute petite !



Le code présenté ici est requis pour toute classe qui doit être de type Singleton, mais on trouvera en général dans ces classes, d'autres attributs et méthodes qui lui sont propre.

5.2 Codage

Adaptons ce design pattern au problème de notre bouclier, en enrichissant la classe de cette manière:



- _uniqueInstance est un attribut privé et statique: il n'existera que dans la classe et pas dans l'objet.
- _isShieldOpen renvoi un booléen. C'est un attribut privé.
- Le constructeur est privé.
- getInstance() est publique mais statique.
- openShield() et closeShield() inversent la valeur de _isShieldOpen.
- getShieldState() renvoie "OPEN" ou "CLOSE" en fonction de la valeur de _ isShieldOpen.

Voici le code de la classe:

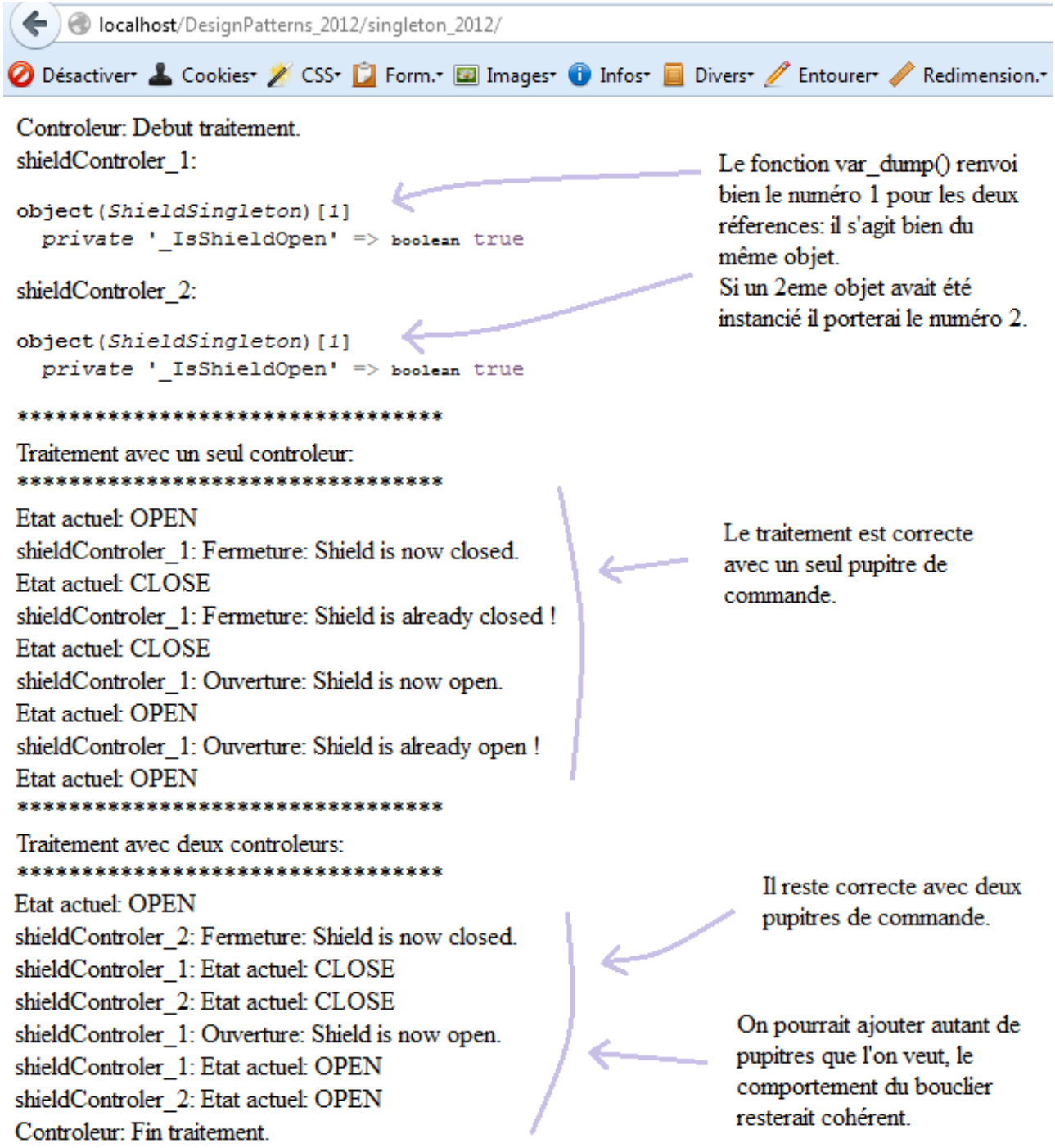
```php
<?php
class ShieldSingleton {

    private static $_uniqueInstance = null;
    private $_IsShieldOpen;

    // Le constructeur:
    private function __construct() {
        $this->_IsShieldOpen = TRUE;
    }

    // Methode publique pour obtenir l'instance:
    public static function getInstance() {
        if(is_null(self::$_uniqueInstance)) {
           self::$_uniqueInstance = new ShieldSingleton();
        }
        return self::$_uniqueInstance;
    }

    public function IsShieldOpen() {
        return $this->_IsShieldOpen;
    }

    public function getShieldState() {
        if ($this->_IsShieldOpen == FALSE ) {
            return "CLOSE";
        } else {
            return "OPEN";
        }
    }


    public function closeShield() {
        if($this->_IsShieldOpen == TRUE) {
            $this->_IsShieldOpen = FALSE;
            echo "Shield is now closed.</br>";
        } else {
            echo "Shield is already closed !</br>";
        }
    }

    public function openShield() {
        if($this->_IsShieldOpen == FALSE) {
            $this->_IsShieldOpen = TRUE;
            echo "Shield is now open.</br>";
        } else {
            echo "Shield is already open !</br>";
        }
    }

}
?>
```

Nous utiliserons comme d'habitude index.php comme contrôleur de notre pattern.

Le contrôleur va instancier des objets et les utiliser:

```php
<?php

//*************************************************************
// SINGLETON pattern controler
//*************************************************************

require_once 'includePaths.php';
$newline = "</br>";

echo 'Controleur: Debut traitement.' . $newline;

// Instanciations:
$shieldControler_1 = ShieldSingleton::getInstance();
$shieldControler_2 = ShieldSingleton::getInstance();

// on vérifie qu'il s'agit bien du meme objet:
echo 'shieldControler_1:' . "</br><pre>";
var_dump($shieldControler_1);
echo '</pre>';

echo 'shieldControler_2:' . "</br><pre>";
var_dump($shieldControler_2);
echo '</pre>';

echo '*********************************' . "</br>";
echo 'Traitement avec un seul controleur: ' . "</br>";
echo '*********************************' . "</br>";
echo "Etat actuel: " . $shieldControler_1->getShieldState() . "</br>";
echo "shieldControler_1: Fermeture: ";
$shieldControler_1->closeShield();
echo "Etat actuel: " . $shieldControler_1->getShieldState() . "</br>";
echo "shieldControler_1: Fermeture: ";
$shieldControler_1->closeShield();
echo "Etat actuel: " . $shieldControler_1->getShieldState() . "</br>";
echo "shieldControler_1: Ouverture: ";
$shieldControler_1->openShield();
echo "Etat actuel: " . $shieldControler_1->getShieldState() . "</br>";
echo "shieldControler_1: Ouverture: ";
$shieldControler_1->openShield();
echo "Etat actuel: " . $shieldControler_1->getShieldState() . "</br>";

echo '*********************************' . "</br>";
echo 'Traitement avec deux controleurs: ' . "</br>";
echo '*********************************' . "</br>";
echo "Etat actuel: " . $shieldControler_2->getShieldState() . "</br>";
echo "shieldControler_2: Fermeture: ";
$shieldControler_2->closeShield();
echo "shieldControler_1: Etat actuel: " .
        $shieldControler_1->getShieldState() . "</br>";
echo "shieldControler_2: Etat actuel: " .
        $shieldControler_2->getShieldState() . "</br>";
echo "shieldControler_1: Ouverture: ";
$shieldControler_1->openShield();
echo "shieldControler_1: Etat actuel: " .
        $shieldControler_1->getShieldState() . "</br>";
echo "shieldControler_2: Etat actuel: " .
        $shieldControler_2->getShieldState() . "</br>";
echo 'Controleur: Fin traitement.' . $newline;

?>
```

Et voici le résultat de l'exécution:




J'ai tout compris.
...Enfin je crois.

6 Les design patterns liés à Factory

Les design patterns ? Oui, il y en a deux. On peut même dire qu'il y en a trois, car l'un d'entre eux n'est pas vraiment un design pattern, mais on ne peut pas l'ignorer.

La notion de Factory (usine en anglais) concerne l'instanciation des objets. Certaines situations nécessitent de confier l'instanciation des objets à une classe ou un groupe de classes.

Dans ce cas, la classe souhaitant instancier un objet n'utilisera pas l'opérateur new(), mais demandera à sa Factory de lui fournir l'objet demandé.

Le but de cette décorrélation est toujours le même en POO: isoler un traitement (ici l'instanciation d'objets) pour le rendre partageable, maintenable, et lui permettre d'intégrer de la complexité. En effet l'instanciation nécessite souvent de choisir les objets à instancier, en fonction d'un contexte.

Les design patterns liés à la notion de Factory, répondent à ces contraintes en déléguant l'instanciation d'objets à des classes dont ce sera la responsabilité.

Dans les 3 chapitres suivants nous aborderons:

- La pseudo Factory: il ne s'agit pas d'un design pattern officiel, mais d'une approche simple, ressemblant à une Factory, et qui est souvent utilisée pour des problématiques peu complexes d'instanciation.
- Factory Method: design pattern dans lequel la Factory est matérialisée par une méthode abstraite.
- Abstract Factory: design pattern dans lequel la Factory est matérialisée par une abstraction qui peut être une interface ou une classe abstraite. Ici nous utiliserons une interface.

Durant ces 3 chapitres, nous allons découvrir d'où viennent et comment sont fabriqués les superbes uniformes utilisés dans la flotte de la Confédération, dont fait partie l'Enterprise.

7 PSEUDO FACTORY

Mis à part le fait qu'il y a des tenues hommes, des tenues femmes et qu'on trouve 3 coloris (jaune, bleu, rouge), ces tenues semblent banales.

Il n'en est rien. Ce sont des produits de haute technologie conçus pour affronter les dangers des voyages dans l'espace:

- Le tissu reçoit deux traitements: l'un permettant d'atténuer les rayonnements électromagnétiques, l'autre le rendant insensible aux acides les plus dangereux.
- Le vêtement contient un dispositif miniaturisé permettant de connaître sa position spatio-temporelle (et celle de son occupant).

Les étapes de la fabrication sont les suivantes:

- Assemblage des pièces par couture.
- Traitement anti rayonnement électromagnétique.
- Traitement anti acide.
- Mise en place du localisateur.
- Tests du vêtement.

La confédération souhaite bien entendu que ces étapes de fabrication soient respectées, aussi le programme de fabrication devrait posséder une méthode ressemblant à cela:

```
function makeClothe() {
    clothe = new clothe();
    clothe->assemble();
    clothe->electromagneticTreatment();
    clothe->acidTreatment();
    clothe->placeLocalisor();
    clothe->test();
    return clothe;
}
```

Malheureusement il y a plusieurs types de vêtements, il faut donc ajouter du code pour choisir le vêtement à instancier:

```php
function makeClothe($clotheModel) {

    switch ($clotheModel){
        case "STANDARD_WOMAN":
            $clothe = new(StandardWomanClothe);
            break;
        case "DELUXE_WOMAN":
            $clothe = new(DeluxeWomanClothe);
            break;
        case "STANDARD_MAN":
            $clothe = new(StandardManClothe);
            break;
        case "DELUXE_MAN":
            $clothe = new(DeluxeManClothe);
    }

    $clothe->assemble();
    $clothe->electromagneticTreatment();
    $clothe->acidTreatment();
    $clothe->placeLocalisor();
    $clothe->test();
    return $clothe;

} // end function
```

Le model de vêtement est passé à la méthode.

Selon le modèle, l'objet correspondant est instancié.

Une fois l'objet obtenu, le processus standard de fabrication est appliqué.

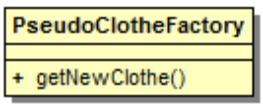
Ici le choix de l'objet à instancier représente un traitement sujet à évolution: de nouveaux modèles de vêtements peuvent apparaître, d'autres peuvent être supprimés. Dans ce cas il faudra inévitablement modifier le code de cette méthode.

A l'opposé, le processus de fabrication semble plus stable: il évoluera peu ou pas, et de plus il est indépendant du type de vêtement.

Les design patterns détestent qu'on mélange des notions variables et des notions stables dans la même classe.

Nous allons isoler dans une nouvelle classe, le code responsable du choix de l'objet à instancier. Appelons cette classe PseudoClotheFactory. Nous y créerons une méthode getNewClothe() permettant d'obtenir un nouvel objet Clothe, en lui fournissant le type de vêtement souhaité.

Voici notre Factory:



Il faut bien sûr prévoir des classes pour les vêtements. Quel que soit le modèle de vêtement, il doit

implémenter les méthodes suivantes:

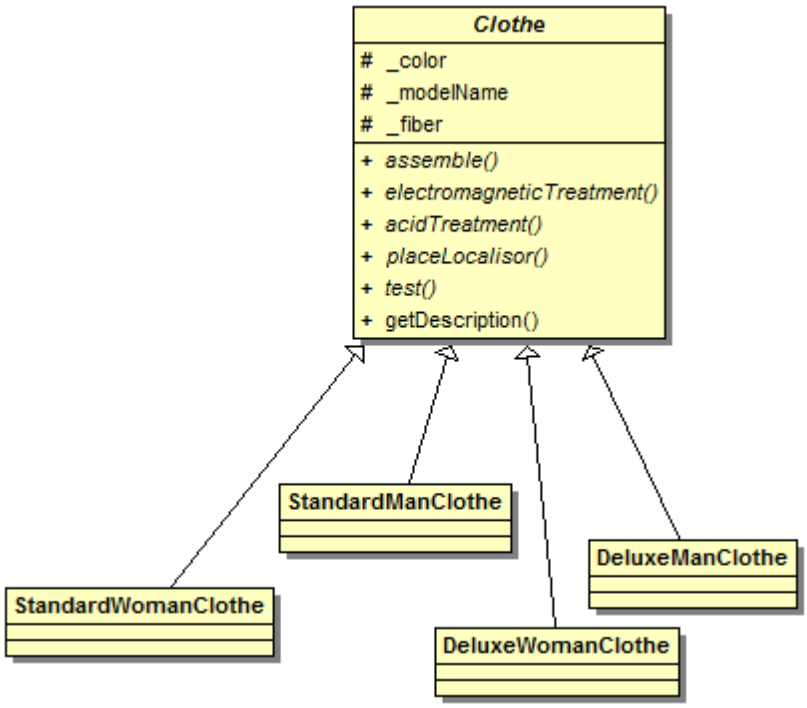
- assemble()
- electromagneticTreatment()
- acidTreatment()
- placeLocalisor()
- test()

De plus tout modèle de vêtement doit connaître:

- Sa couleur.
- Son nom de modèle.
- Son type de fibre: les modèles Deluxe ont un type de fibre encore plus résistant.

On prévoit également une méthode getDescription() qui permettra à chaque vêtement de fournir sa description détaillée.

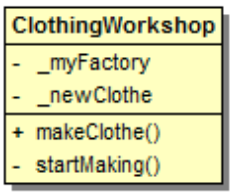
Créons la classe abstraite qui obligera tout modèle de vêtement à remplir ce contrat:



Il nous faut enfin une classe pour représenter l'entité qui décide de fabriquer un vêtement, et qui va donc utiliser notre Factory. C'est en quelque sorte l'atelier de fabrication des vêtements. Appelons cette classe ClothingWorkshop et dotons-la d'une méthode makeClothe().

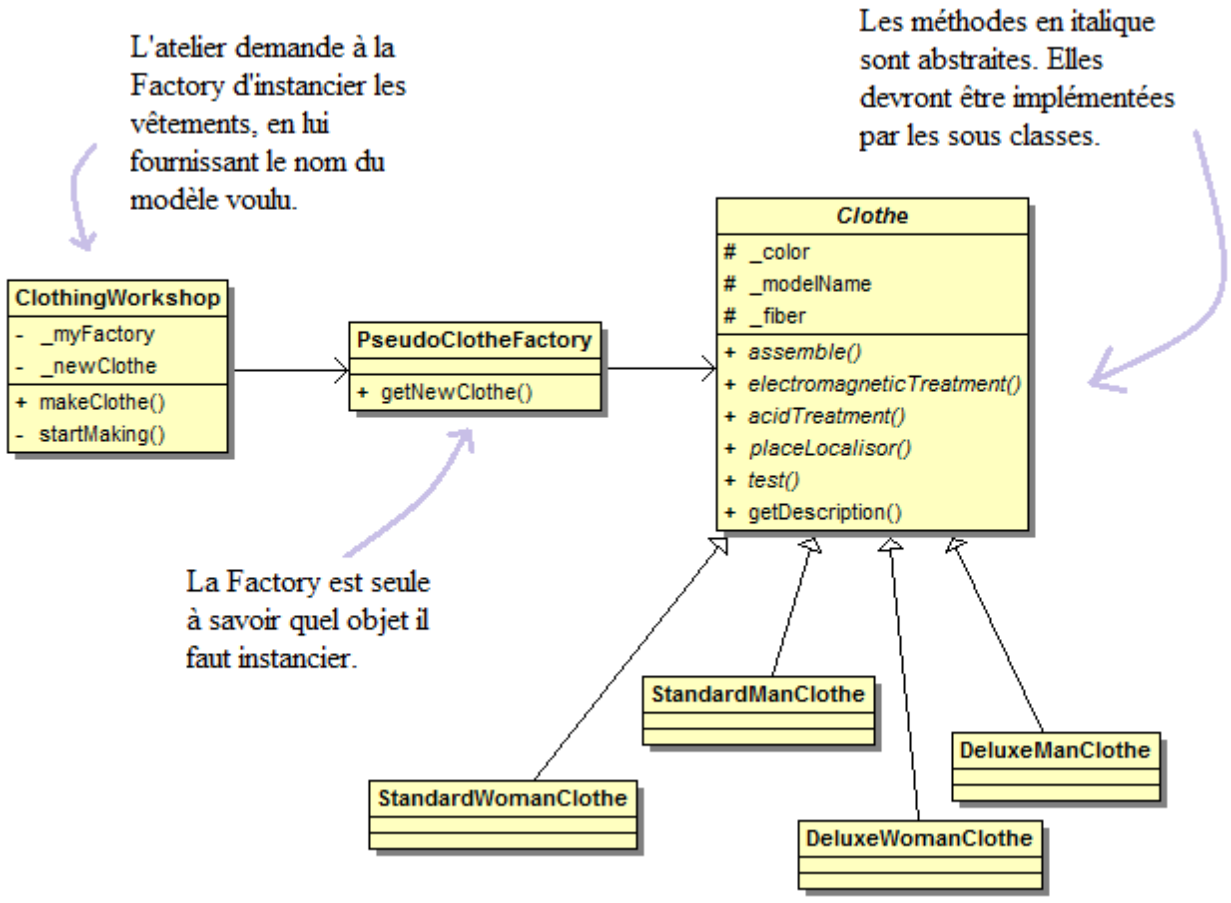
Cette classe devra connaître sa Factory.

Elle aura également une référence vers le vêtement que la Factory aura instancié pour elle.



La méthode makeClothe(), qui prendra en paramètre un modèle de vêtement et une couleur, déclenchera la fabrication du vêtement demandé en utilisant sa méthode privée startMaking().

Le montage complet nous donne ceci:



7.1 Codage

Commençons par la classe abstraite Clothe :

```php
<?php

abstract class Clothe {

    protected $_color;
    protected $_modelName;
    protected $_fiber;

    public function __construct($color) {

        switch ($color) {
            case "YEL":
                $this->_color = "Jaune";
                break;
            case "RED":
                $this->_color = "Rouge";
                break;
            case "BLU":
                $this->_color = "Bleu";
            default :
                $this->_color = "Bleu";
        }  // end switch
    }

    abstract public function assemble();
    abstract public function electromagneticTreatment();
    abstract public function acidTreatment();
    abstract public function placeLocalisor();
    abstract public function test();

    public function getDescription() {
        $description = "  " . 'Modele: ' .
                $this->_modelName . "</br>";
        $description .= "  " . 'Couleur: ' .
                $this->_color . "</br>";
        $description .= "  " . 'Fibre: ' .
                $this->_fiber . "</br>";
        return $description;
    }

}

?>
```

Le traitement de la couleur peut être implémenté dans la classe abstraite, puisque commun à toutes les sous classes.

Les méthodes abstraites doivent être implémentées par les sous classes.

La méthode getDescription() est également commune à toutes les sous classes, donc implémentée ici.

Ensuite nous avons une classe pour chacun des 4 modèles de vêtements:

```php
class DeluxeManClothe extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_modelName = "Tenue Deluxe Homme.";
        $this->_fiber = "Spéciale ultra haute résistance.";
    }

    public function assemble() {
        echo 'Assemblage Deluxe.' . "</br>";
    }

    public function electromagneticTreatment() {
        echo 'Traitement anti electromagnetic.' . "</br>";
    }

    public function acidTreatment() {
        echo 'Traitement anti acide.' . "</br>";
    }

    public function placeLocalisor() {
        echo 'Fixation du localisateur.' . "</br>";
    }

    public function test() {
        echo 'Protocole de tests standard.' . "</br>";
    }

}

?>
```

On invoque le constructeur de la classe mère, puis on complète par du traitement spécifique.

C'est ici qu'on implémente les méthodes abstraites de la classe mère.

```php
<?php

class DeluxeWomanClothe extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_modelName = "Tenue Deluxe Femme.";
        $this->_fiber = "Spéciale ultra haute résistance.";
    }

    public function assemble() {
        echo 'Assemblage Deluxe.' . "</br>";
    }

    public function electromagneticTreatment() {
        echo 'Traitement anti electromagnetic.' . "</br>";
    }

    public function acidTreatment() {
        echo 'Traitement anti acide.' . "</br>";

    }

    public function placeLocalisor() {
        echo 'Fixation du localisateur.' . "</br>";
    }

    public function test() {
        echo 'Protocole de tests standard.' . "</br>";
    }

}

?>
```

Les modèles Deluxe ont une fibre spéciale, ainsi qu'un assemblage particulier.

```php
<?php

class StandardManClothe extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_modelName = "Tenue standard Homme.";
        $this->_fiber = "Haute résistance.";
    }

    public function assemble() {
        echo 'Assemblage standard.' . "</br>";
    }

    public function electromagneticTreatment() {
        echo 'Traitement anti electromagnetic.' . "</br>";
    }

    public function acidTreatment() {
        echo 'Traitement anti acide.' . "</br>";

    }

    public function placeLocalisor() {
        echo 'Fixation du localisateur.' . "</br>";
    }

    public function test() {
        echo 'Protocole de tests standard.' . "</br>";
    }

}

?>
```

Les modèles standards ont une fibre haute résistance, et un assemblage standard.

```php
<?php

class StandardWomanClothe extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_modelName = "Tenue standard Femme.";
        $this->_fiber = "Haute résistance.";
    }

    public function assemble() {
        echo 'Assemblage standard.' . "</br>";
    }

    public function electromagneticTreatment() {
        echo 'Traitement anti electromagnetic.' . "</br>";
    }

    public function acidTreatment() {
        echo 'Traitement anti acide.' . "</br>";

    }

    public function placeLocalisor() {
        echo 'Fixation du localisateur.' . "</br>";
    }

    public function test() {
        echo 'Protocole de tests standard.' . "</br>";
    }

}

?>
```

La classe PseudoClotheFactory: elle seule sait quels objets il faut instancier:

```php
<?php

class PseudoClotheFactory {

    public function getNewClothe($clotheModel,$color) {

        $clothe = null;

        switch ($clotheModel){
            case "STANDARD_WOMAN":
                $clothe = new StandardWomanClothe($color);
                break;
            case "DELUXE_WOMAN":
                $clothe = new DeluxeWomanClothe($color);
                break;
            case "STANDARD_MAN":
                $clothe = new StandardManClothe($color);
                break;
            case "DELUXE_MAN":
                $clothe = new DeluxeManClothe($color);
        }  // end switch

        return $clothe;

    } // end function

} // end class

?>
```

Et enfin la classe ClothingWorkshop qui est le client de la Factory.

```php
<?php

class ClothingWorkshop {

    private $_myFactory;
    private $_newClothe;

    public function __construct($factory) {
        $this->_myFactory = $factory;
    }

    public function makeClothe($clotheModel, $color) {

        $this->_newClothe =
                $this->_myFactory->getNewClothe($clotheModel,$color);
        $this->startMaking();
        return $this->_newClothe;
    } // end function

    private function startMaking(){
        $this->_newClothe->assemble();
        $this->_newClothe->electromagneticTreatment();
        $this->_newClothe->acidTreatment();
        $this->_newClothe->placeLocalisor();
        $this->_newClothe->test();
    }

} // end class

?>
```

On passe la Factory au constructeur du ClothingWorkshop.

Quand le workshop doit créer un vêtement, il demande à la Factory d'en instancier un, en lui indiquant le modèle et la couleur.

Ensuite le processus de fabrication est appliqué.

ClothingWorkshop connaît le procédé de fabrication des vêtements, mais doit demander à la Factory de lui instancier les objets vêtements.

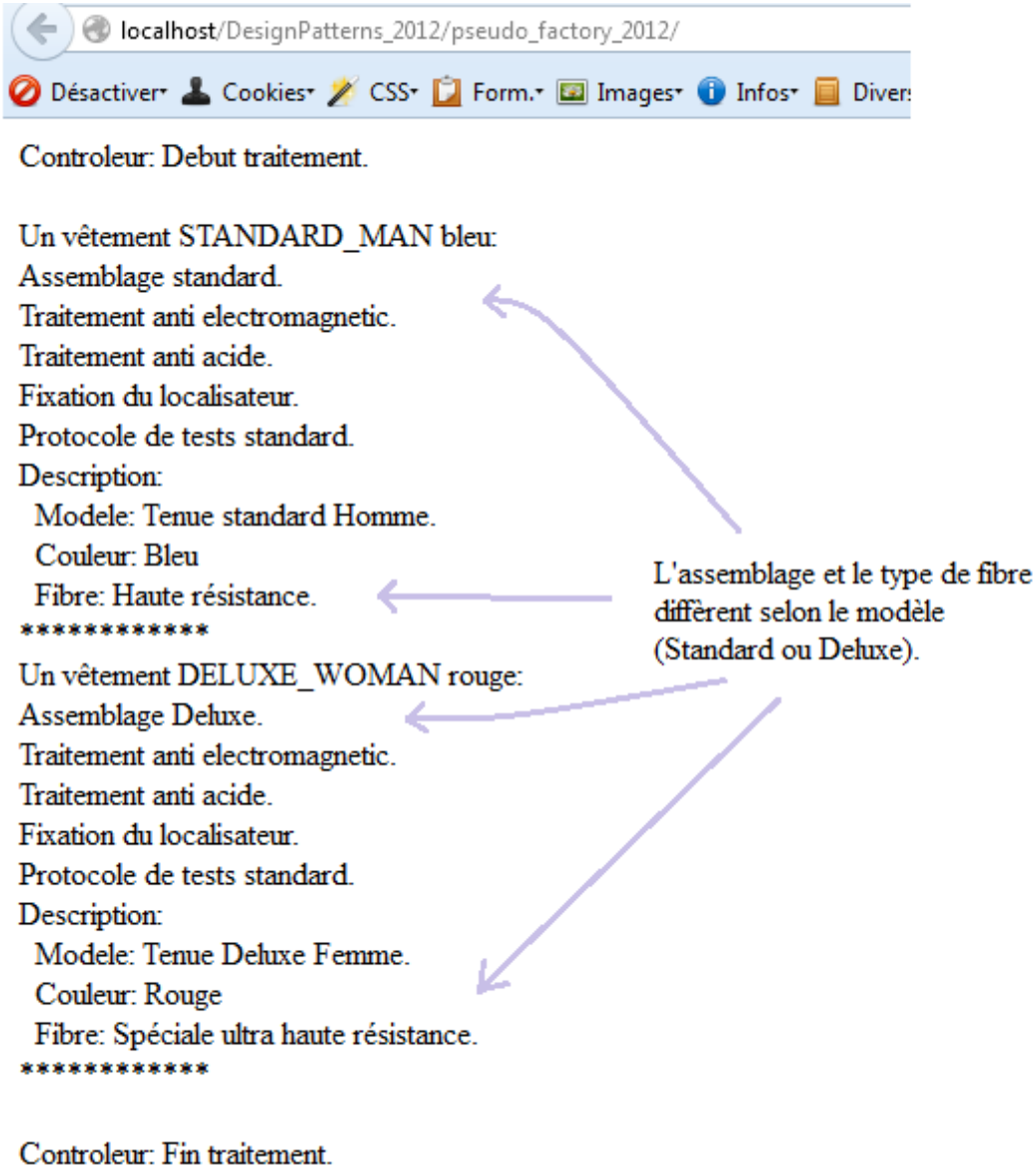
Comme d'habitude nous allons utiliser index.php comme contrôleur de l'ensemble du montage.

Index.php doit instancier un ClothingWorkshop et lui demander de fabriquer des vêtements:

```php
<?php

//************************************************************
// PSEUDO FACTORY pattern controler
//************************************************************

require_once 'includePaths.php';
$newline = "</br>";

echo 'Controleur: Debut traitement.' . $newline . $newline;

// Instanciations:
$theFactory = new PseudoClotheFactory();
$theWorkshop = new ClothingWorkshop($theFactory);

// Traitements:
echo 'Un vêtement STANDARD_MAN bleu:' . $newline;
$clothe_1 =  $theWorkshop->makeClothe("STANDARD_MAN","BLU");
echo 'Description: ' . $newline;
echo $clothe_1->getDescription();
echo '************' . $newline;

echo 'Un vêtement DELUXE_WOMAN rouge:' . $newline;
$clothe_2 = $theWorkshop->makeClothe("DELUXE_WOMAN","RED");
echo 'Description: ' . $newline;
echo $clothe_2->getDescription();
echo '************' . $newline;

echo $newline . 'Controleur: Fin traitement.' . $newline;

?>
```

Il faut donner une Factory au ClothingWorkshop.

Et voici le résultat de l'exécution:



Il n'y a rien de nouveau dans ce montage. Le choix de l'instanciation des objets a simplement été déplacé vers la Factory.

Oui nous n'avons fait qu'isoler une notion variable. Mais ce faisant la Factory devient:

- Le point d'accès unique et partageable pour obtenir des objets vêtements.
- La complexité (ici très faible) du choix des objets à instancier, existe uniquement dans la Factory. Cela facilite la maintenance.

Comme dit au début de ce chapitre, la PseudoFactory n'est pas un design pattern, mais est souvent utilisée pour des problématiques simples.

Nous allons aborder dans le prochain chapitre, le pattern Factory Méthod qui permet d'utiliser implicitement une Factory par héritage. On pourra donc avoir plusieurs Factory.

8 PATTERN FACTORY METHOD

Dans le chapitre précédent, la PseudoFactory nous a permis:

- de contrôler l'instanciation des vêtements.
- d'implémenter les spécificités des gammes Standard et Deluxe, dans les différents modèles de vêtements.

Aujourd'hui la Confédération est confrontée à un nouveau problème: certaines galaxies utilisent leur propre type de fibre pour les vêtements.

La Factory utilisée dans PseudoFactory, ne peut pas, à elle seule, gérer ces différences. Il nous faudrait maintenant une Factory par galaxie, de manière à respecter le processus de fabrication imposé par la Confédération, tout en permettant les adaptations spécifiques à chaque galaxie.

Ce problème peut être résolu par les design patterns Factory Method ou Abstract Factory.

Dans ce chapitre nous allons utiliser Factory Method, puis dans le prochain chapitre nous irons plus loin avec Abstract Factory.

Bien, prenons deux galaxies, la nôtre: la Voie Lactée (Milkyway en anglais), et NGC1313 une galaxie proche (15 millions d'années-lumière).

Milkyway utilise la fibre officielle pour fabriquer les vêtements, alors que NGC1313 utilise une fibre spéciale appelée arachno-fibre extrêmement résistante.

De plus, comme il y a maintenant plusieurs galaxies productrices de vêtements, il faut connaître l'origine de chacun d'eux.

Enfin, ce que nous avions avec la Pseudo Factory demeure: le processus de fabrication doit être le même dans toutes les galaxies, et il y a toujours les différences suivantes entre les gammes de vêtements Standard et Deluxe:

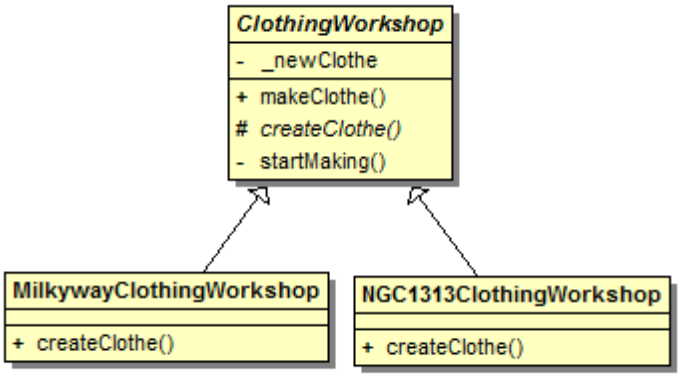
- Assemblage: Standard ou Deluxe.
- Fibre: Standard ou Deluxe (indépendamment du fait qu'il s'agit d'une fibre type Milkyway

ou NGC1313).

Le principe de la Factory Method est simple: dans une classe mère, on définit une méthode abstraite, c'est à dire non implémentée, et on laisse le soin aux sous classes de l'implémenter selon leur spécificité. Cette méthode abstraite: c'est la Factory Method. C'est elle qui jouera le rôle de Factory dans les sous classes, et qui sera donc responsable d'instancier les objets vêtements.

Chaque sous classe sera bien sûr spécialisée pour une galaxie.

Créons la classe mère et deux sous classes, une pour chaque galaxie:



ClothingWorkshop est une classe abstraite (son nom est en italique). Elle possède une méthode également abstraite createClothe() qui devra être implémentée par les sous classes. C'est la Factory Method.

Les deux sous classes implémenteront createClothe() de manière à ce que cette méthode crée les vêtements qui correspondent à la galaxie concernée.

La Factory est en quelque sorte intégrée dans le Workshop, alors qu'elle était complètement externe dans la Pseudo Factory.

La méthode makeClothe() sera invoquée pour demander au Workshop de fabriquer un vêtement, en lui passant en paramètre, le nom du modèle et la couleur souhaités. Cette méthode étant commune à toutes les galaxies, elle est implémentée dans la classe abstraite.

Et c'est bien makeClothe() qui invoquera createClothe() pour obtenir un objet vêtement correspondant à la galaxie concernée.

StartMaking() est une méthode privée, donc utilisée de manière interne par le Workshop pour fabriquer concrètement le vêtement. Cette méthode représente donc le protocole de fabrication qui doit être commun à toutes les galaxies, elle est donc implémentée dans la classe abstraite.

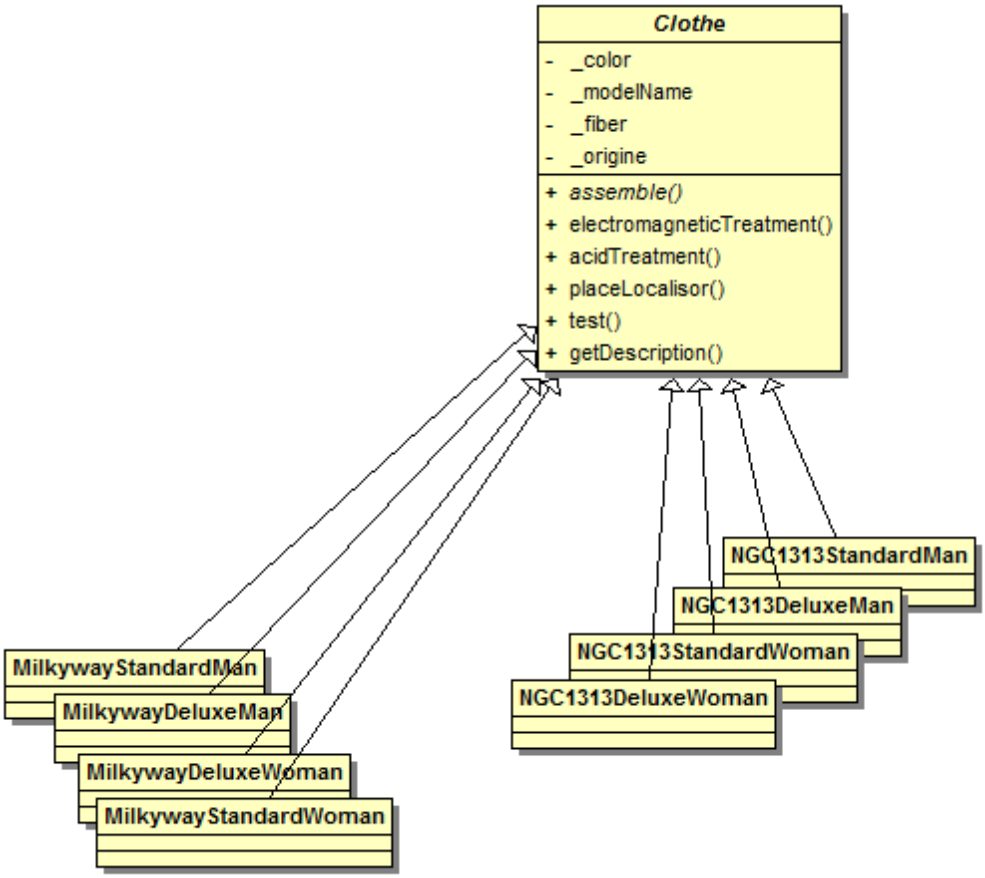
Passons maintenant aux vêtements:

Le montage utilisé dans Pseudo Factory peut être réutilisé en y apportant les modifications suivantes:

- On ajoute un attribut 'origine' à la classe Clothe: tout vêtement doit connaître sa galaxie d'origine.
- La méthode assemble() devient abstraite: chaque vêtement devra l'implémenter selon sa gamme (Standard ou Deluxe).
- Comme on a maintenant une méthode abstraite, la classe devient également abstraite.

Il faut bien sûr créer deux groupes de vêtements contenant chacun une gamme pour la galaxie concernée.

On obtient ceci:

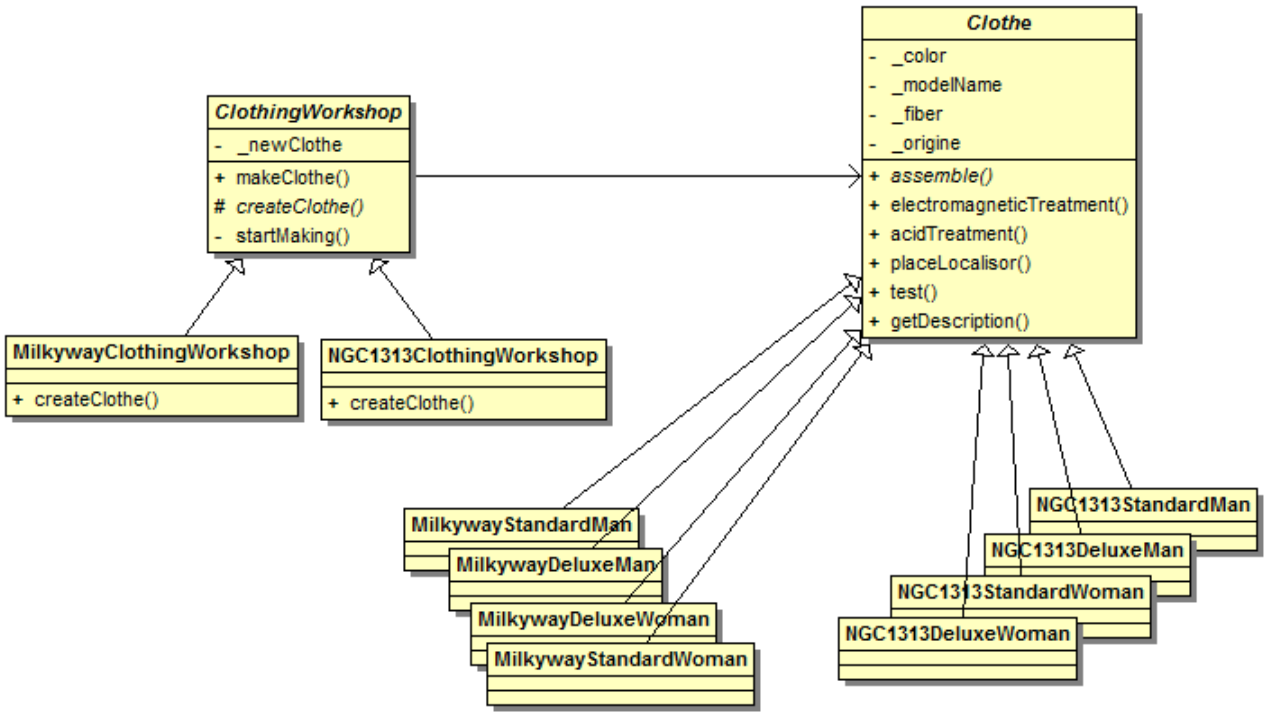


Dans la classe clothe(), nous trouvons les attributs:

- couleur
- nom du modèle de vêtement
- type de fibre
- origine.

Les méthodes sont les mêmes que dans Pseudo Factory puisque les étapes du processus de fabrication n'ont pas changé.

Le montage complet donne ceci:



Le fonctionnement est le suivant:

- Pour fabriquer un vêtement on invoque la méthode makeClothe() sur le workshop concerné (MilkywayClothingWorkshop ou NGC1313ClothingWorkshop), en lui passant le modèle et la couleur souhaités.
- makeClothe() invoque la méthode protégée createClothe() en lui repassant le modèle et la couleur.
- createClothe(), qui est la Factory Method, va instancier le vêtement de la bonne galaxie, et selon le modèle demandé, et retourner cet objet à makeClothe().
- Enfin makeClothe() invoque startMaking() qui va appliquer les différentes étapes du processus de fabrication.

Voilà, tout est dit. Il ne reste plus qu'à coder.

8.1 Codage

La classe abstraite:

```php
<?php

abstract class ClothingWorkshop {

    private $_newClothe;

    public function makeClothe($clotheModel, $color) {
        $this->_newClothe = $this->createClothe($clotheModel, $color);
        $this->startMaking();
        return $this->_newClothe;
    }

    private function startMaking() {
        $this->_newClothe->assemble();
        $this->_newClothe->electromagneticTreatment();
        $this->_newClothe->acidTreatment();
        $this->_newClothe->placeLocalisor();
        $this->_newClothe->test();
    }

    // This is the Factory Method:
    protected abstract function createClothe($clotheModel, $color);
}

?>
```

On invoque la Factory Method.

On applique le processus de fabrication.

Factory Method implémentée dans les sous classes.

MilkywayClothingWorkshop :

```php
<?php

class MilkywayClothingWorkshop extends ClothingWorkshop {

  // This is the Factory Method:
  public function createClothe($clotheModel, $color)
  {
        $clothe = null;

        switch ($clotheModel){
            case "STANDARD_WOMAN":
                $clothe = new MilkywayStandardWoman($color);
                break;
            case "DELUXE_WOMAN":
                $clothe = new MilkywayDeluxeWoman($color);
                break;
            case "STANDARD_MAN":
                $clothe = new MilkywayStandardMan($color);
                break;
            case "DELUXE_MAN":
                $clothe = new MilkywayDeluxeMan($color);
        }  // end switch

        return $clothe;
  }

}

?>
```

C'est ici qu'on choisit le vêtement à instancier

NGC1313ClothingWorkshop:

```php
<?php

class NGC1313ClothingWorkshop extends ClothingWorkshop {

    // This is the Factory Method:
    public function createClothe($clotheModel, $color) {
        $clothe = null;

        switch ($clotheModel) {
            case "STANDARD_WOMAN":
                $clothe = new NGC1313StandardWoman($color);
                break;
            case "DELUXE_WOMAN":
                $clothe = new NGC1313DeluxeWoman($color);
                break;
            case "STANDARD_MAN":
                $clothe = new NGC1313StandardMan($color);
                break;
            case "DELUXE_MAN":
                $clothe = new NGC1313DeluxeMan($color);
        }  // end switch

        return $clothe;
    }

}

?>
```

Clothe :

```php
<?php

abstract class Clothe {

    protected $_color;
    protected $_modelName;
    protected $_fiber;
    protected $_origine;

    public function __construct($color) {

        switch ($color) {
            case "YEL":
                $this->_color = "Jaune";
                break;
            case "RED":
                $this->_color = "Rouge";
                break;
            case "BLU":
                $this->_color = "Bleu";
            default :
                $this->_color = "Bleu";
        }  // end switch
    }

    abstract function assemble();

    public function electromagneticTreatment() {
        echo 'Traitement anti champs electromagnetiques.' . "</br>";
    }

    public function acidTreatment() {
        echo 'Traitement anti acide.' . "</br>";
    }

    public function placeLocalisor() {
        echo 'Positionnement du Localisor.' . "</br>";
    }

    public function test() {
        echo 'Protocole de tests.' . "</br>";
    }

    public function getDescription() {
        $description = "  " . 'Modele: ' .
                $this->_modelName . "</br>";
        $description .= "  " . 'Couleur: ' .
                $this->_color . "</br>";
        $description .= "  " . 'Fibre type: ' .
                $this->_fiber . "</br>";
        $description .= "  " . 'Origine: ' .
                $this->_origine . "</br>";
        return $description;
    }

}

?>
```

MilkywayDeluxeMan:

```php
<?php

class MilkywayDeluxeMan extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_fiber = "Spéciale ultra haute résistance.";
        $this->_modelName = "Tenue Deluxe Homme.";
        $this->_origine = "Milkyway.";
    }

    public function assemble() {
        echo 'Assemblage: Deluxe.' . "</br>";
    }

}

?>
```

Chaque vêtement défini sa fibre, son nom et son origine.

La méthode assemble () est implémentée dans chaque classe vêtement.

MilkywayDeluxeWoman:

```php
<?php

class MilkywayDeluxeWoman extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_fiber = "Spéciale ultra haute résistance.";
        $this->_modelName = "Tenue de luxe Femme.";
        $this->_origine = "Milkyway.";
    }

    public function assemble() {
        echo 'Assemblage: Deluxe.' . "</br>";
    }

}

?>
```

MilkywayStandardMan:

```php
<?php

class MilkywayStandardMan extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_fiber = "Haute résistance.";
        $this->_modelName = "Tenue Standard Homme.";
        $this->_origine = "Milkyway.";
    }

    public function assemble() {
        echo 'Assemblage: Standard.' . "</br>";
    }

}

?>
```

MilkywayStandardWoman:

```php
<?php

class MilkywayStandardWoman extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_fiber = "Haute résistance.";
        $this->_modelName = "Tenue Standard Femme.";
        $this->_origine = "Milkyway.";
    }

    public function assemble() {
        echo 'Assemblage: Standard.' . "</br>";
    }

}

?>
```

NGC1313DeluxeMan:

```php
<?php

class NGC1313DeluxeMan extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_fiber = "Arachno-fibre ultra haute résistance.";
        $this->_modelName = "Tenue de luxe Homme.";
        $this->_origine = "NGC1313.";
    }

    public function assemble() {
        echo 'Assemblage: Deluxe.' . "</br>";
    }

}

?>
```

NGC1313DeluxeWoman:

```php
<?php

class NGC1313DeluxeWoman extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_fiber = "Arachno-fibre ultra haute résistance.";
        $this->_modelName = "Tenue Deluxe Femme.";
        $this->_origine = "NGC1313.";
    }

    public function assemble() {
        echo 'Assemblage: Deluxe.' . "</br>";
    }

}

?>
```

NGC1313StandardMan:

```php
<?php

class NGC1313StandardMan extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_fiber = "Arachno-fibre haute résistance.";
        $this->_modelName = "Tenue Standard Homme.";
        $this->_origine = "NGC1313.";
    }

    public function assemble() {
        echo 'Assemblage: Standard.' . "</br>";
    }

}

?>
```

NGC1313StandardWoman:

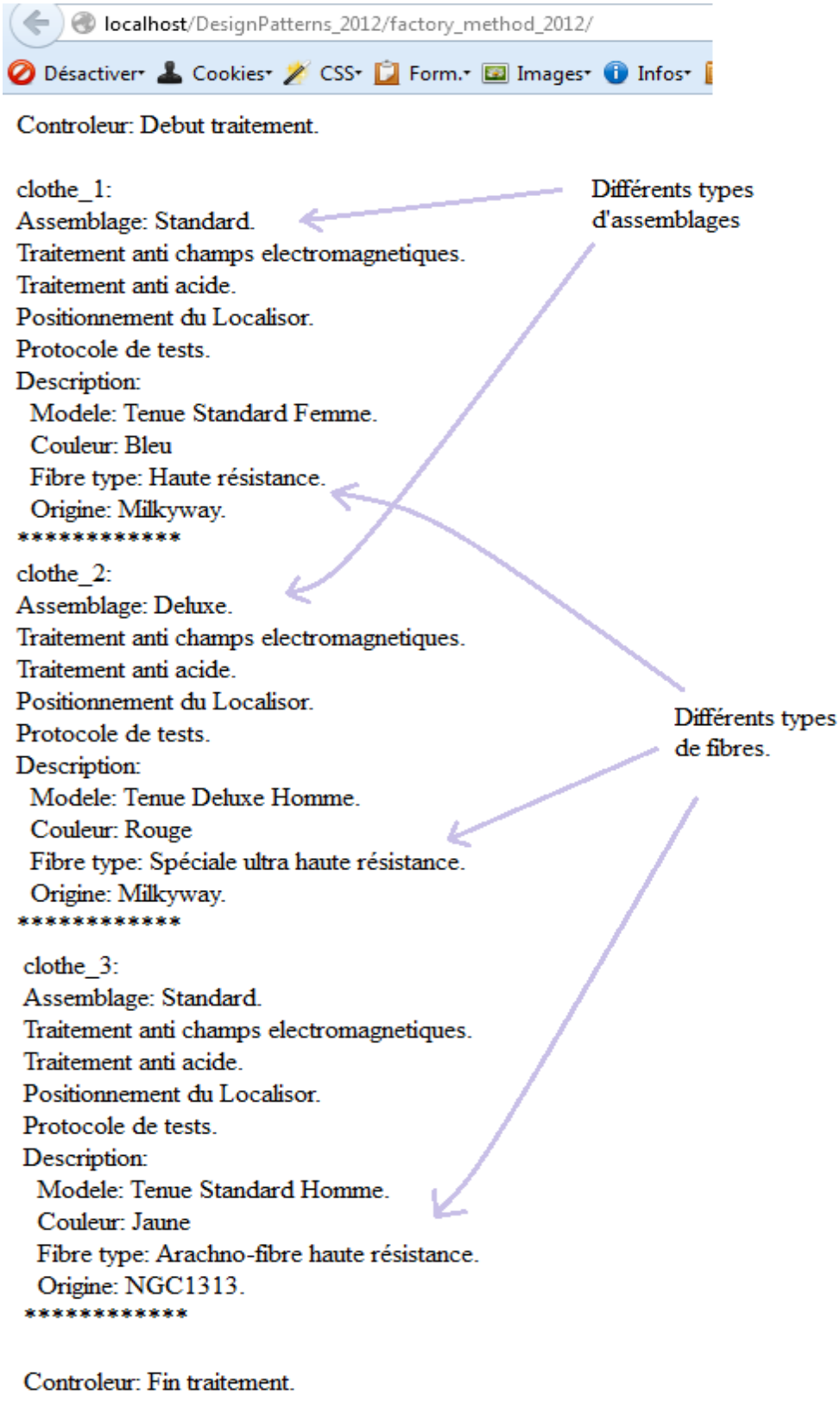
```php
<?php

class NGC1313StandardWoman extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_fiber = "Arachno-fibre haute résistance.";
        $this->_modelName = "Tenue Standard Femme.";
        $this->_origine = "NGC1313.";
    }

    public function assemble() {
        echo 'Assemblage: Standard.' . "</br>";
    }

}

?>
```

8.2 Tests

Nous utiliserons index.php comme contrôleur de l'ensemble de notre pattern.

```php
<?php

//*************************************************************
// FACTORY METHOD pattern controler
//*************************************************************

require_once 'includePaths.php';
$newline = "</br>";

echo 'Controleur: Debut traitement.' . $newline . $newline;

// Instanciations:
$MilkyWayWorkshop = new MilkywayClothingWorkshop;
$NGC1313Workshop = new NGC1313ClothingWorkshop;


// Traitements:
echo 'clothe_1:' . $newline;
$clothe_1 = $MilkyWayWorkshop->makeClothe('STANDARD_WOMAN', 'BLU');
echo 'Description: ' . $newline;
echo $clothe_1->getDescription();
echo '************' . $newline;

echo 'clothe_2:' . $newline;
$clothe_2 = $MilkyWayWorkshop->makeClothe('DELUXE_MAN', 'RED');
echo 'Description: ' . $newline;
echo $clothe_2->getDescription();
echo '************' . $newline;

echo 'clothe_3:' . $newline;
$clothe_3 = $NGC1313Workshop->makeClothe('STANDARD_MAN', 'YEL');
echo 'Description: ' . $newline;
echo $clothe_3->getDescription();
echo '************' . $newline;

echo $newline . 'Controleur: Fin traitement.' . $newline;
?>
```

Le résultat de l'exécution nous donne ceci:



En conclusion on peut dire que vu du contrôleur, le montage est assez simple à utiliser car il suffit de demander un vêtement à un workshop pour l'obtenir complètement fabriqué. Il ne reste qu'à lui demander sa description via la méthode getDescription().

9 PATTERN ABSTRACT FACTORY

Dans le précédent chapitre (Factory Method) nous avons laissé certaines initiatives aux vêtements. En effet ils étaient responsables de déterminer les attributs suivants:

- type de fibre
- nom du modèle
- origine
- type d'assemblage

Quant au protocole de test, il était laissé à la discrétion du workshop et on n'avait aucun contrôle sur cet élément.

La Confédération souhaite par conséquent durcir le contrôle de la fabrication, en interdisant aux vêtements et aux workshops de définir ou de choisir des éléments par eux même.

Les décisions suivantes ont été prises:

- Le type de fibre sera imposé aux vêtements selon la galaxie et la gamme du vêtement.
- L'origine sera imposée aux vêtements selon leur galaxie.
- Le type d'assemblage sera imposé aux vêtements selon leur gamme.
- Le protocole de test sera imposé aux vêtements selon leur galaxie.
- Le type de localisor sera imposé aux vêtements selon leur galaxie.

Le seul attribut qui continue d'être défini par le vêtement est son nom de modèle, c'est à dire son propre nom.

Ces contraintes semblent complexes à mettre en œuvre. Il n'en est rien.

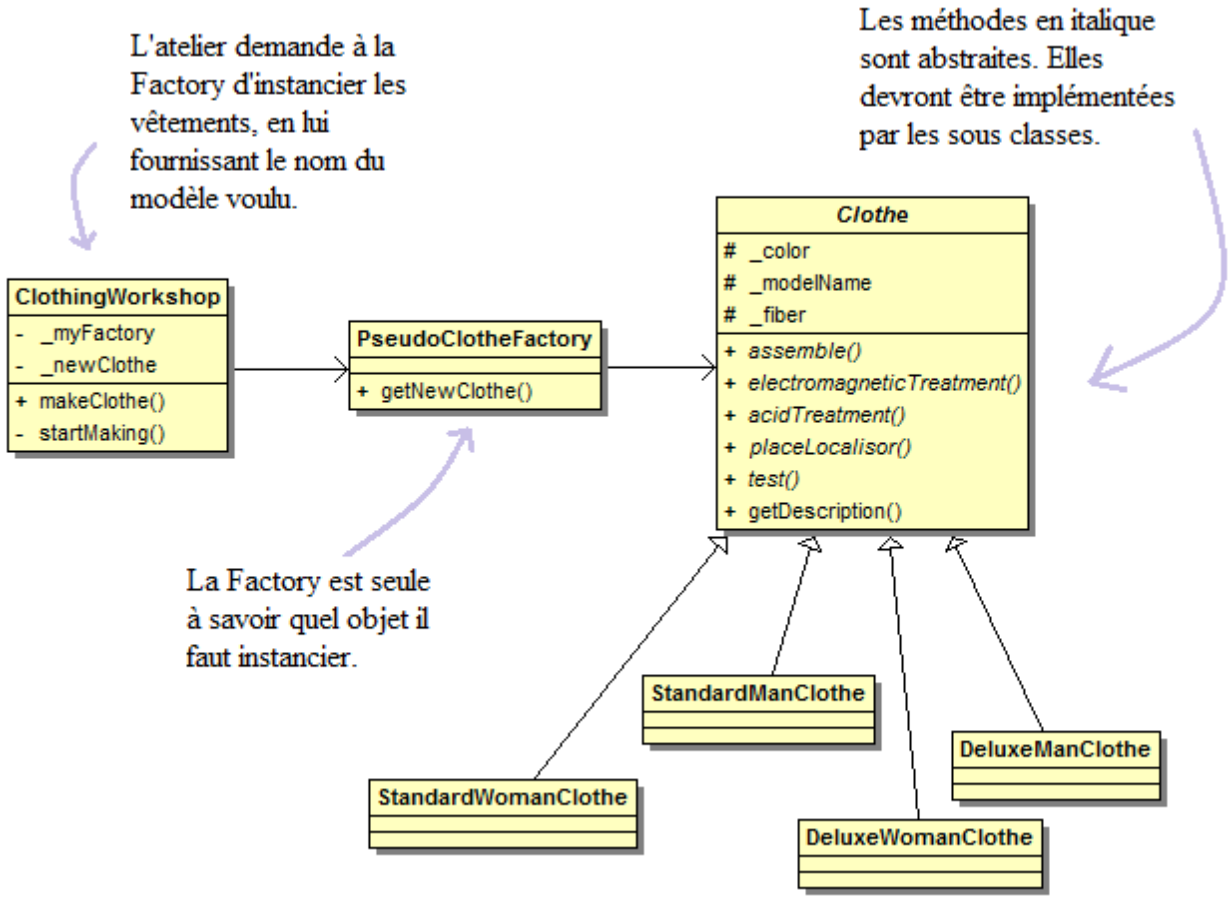
Grâce à la notion de Factory, et à condition que le montage soit bien fait, c'est un jeu d'enfant.

Nous pourrions adapter notre Factory Method pour intégrer ces nouvelles contraintes. Mais profitons de l'occasion pour refaire complètement notre montage en utilisant le pattern Abstract Factory.

La différence essentielle entre les deux design patterns est que Abstract Factory, étant externe, peut être utilisée par n'importe quelle classe de l'application, alors que Factory Method, étant une méthode, doit être vue davantage comme un service privé dédié à la classe qui implémente cette méthode.

Mais avant de commencer le travail, quelques mots sur ce pattern.

Dans le chapitre consacré à Pseudo Factory, nous avions le montage suivant:

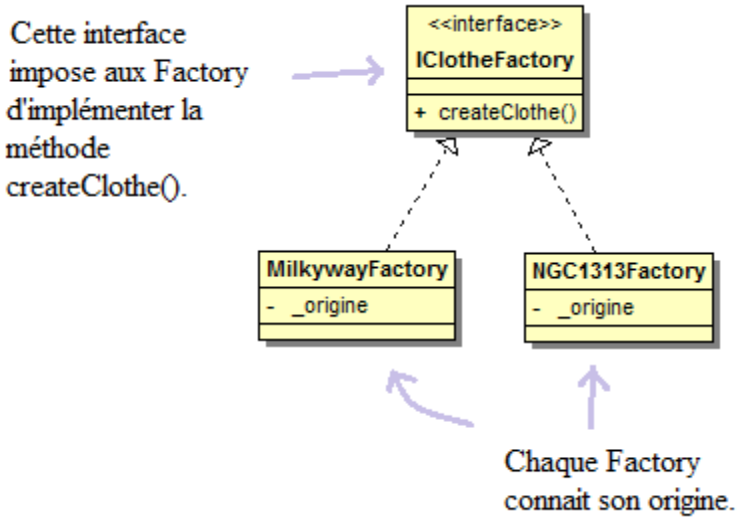


La Factory était externe à ClothingWorkshop.

On peut considérer que le pattern Abstract Factory est une extension de PseudoFactory, car nous allons simplement permettre l'existence de plusieurs Factory, mais en leur imposant une interface

commune, dans laquelle nous définissons une simple méthode createClothe() qui sera responsable de l'instanciation de tout ce qui est nécessaire pour fabriquer le vêtement, en tenant compte de toutes les contraintes évoquées.

Nous obtenons ceci:



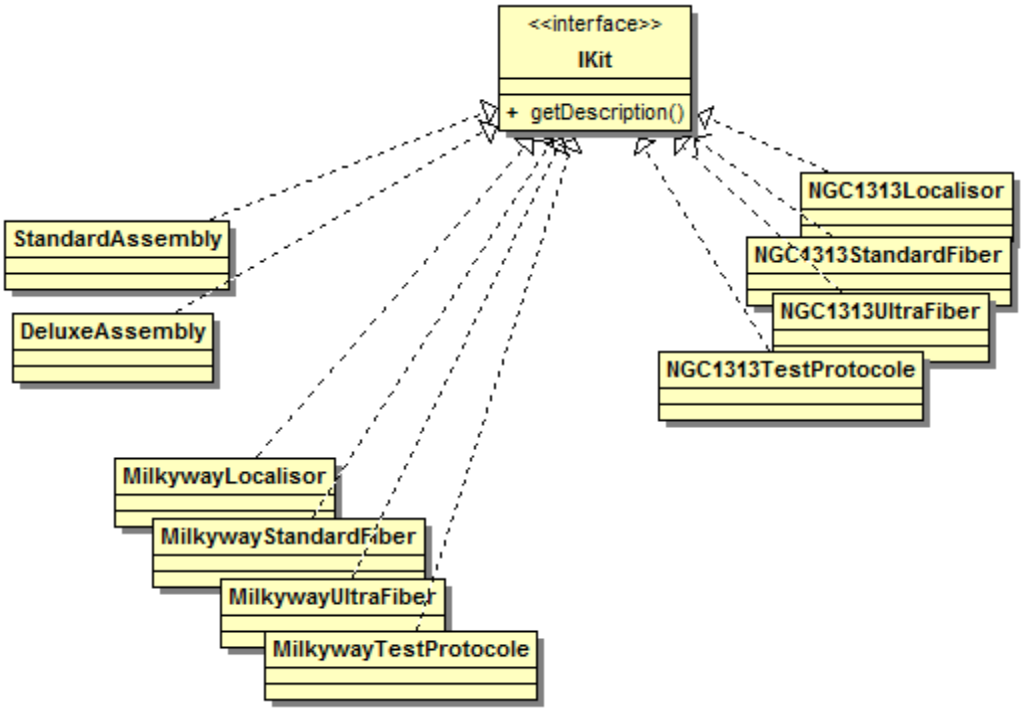
Intéressons-nous maintenant aux éléments nécessaires pour fabriquer les vêtements. Comme indiqué au début du chapitre, nous devons imposer les éléments suivants aux vêtements:

- Le type de fibre.
- L'origine.
- Le type d'assemblage.
- Le protocole.
- Le type de localisor.

Ces éléments forment une sorte de kit de fabrication, et c'est bien entendu notre Factory qui va se charger de choisir les bons éléments du kit, selon le contexte.

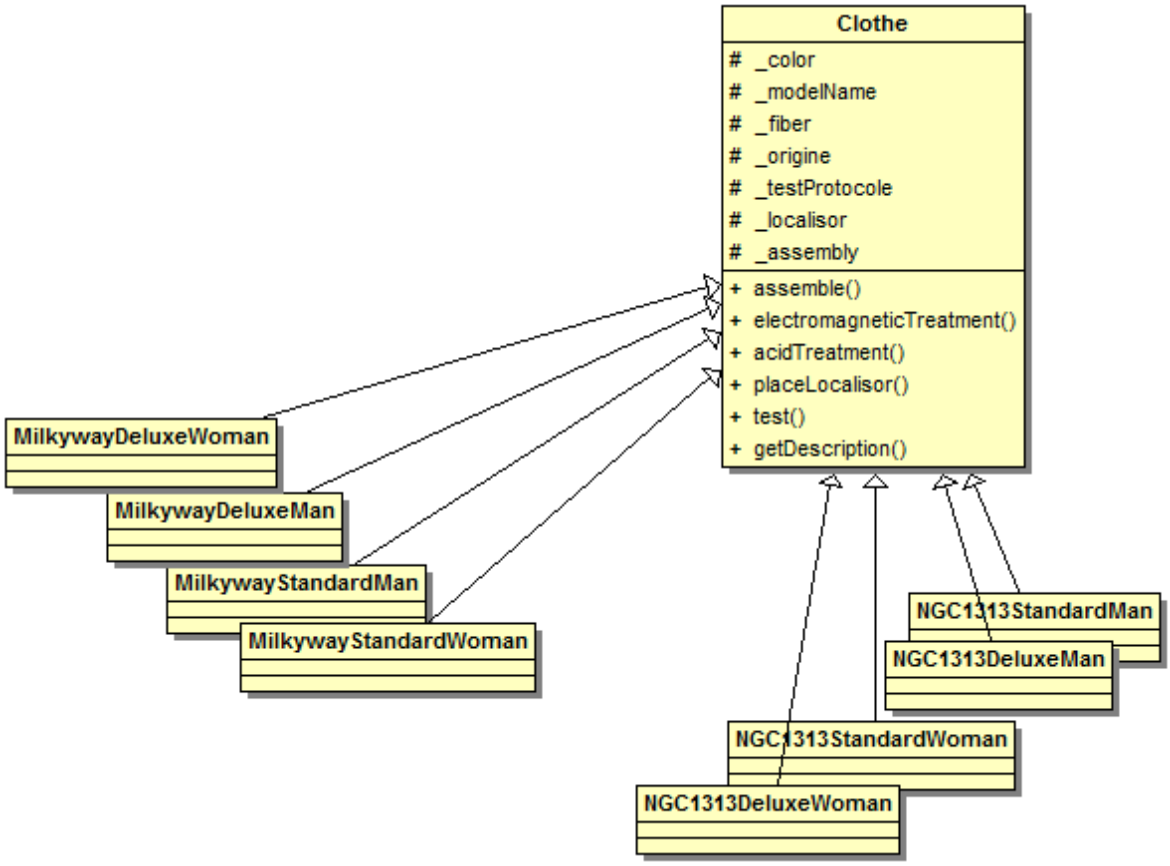
Cela signifie que nous devons disposer de classes pour les éléments du kit. Nous n'aurons qu'une chose à demander à un élément: sa description. Nous prévoyons par conséquent une méthode getDescription().

Comme tous les éléments du kit doivent implémenter cette méthode, nous l'imposons via une petite interface. Ce qui nous donne:



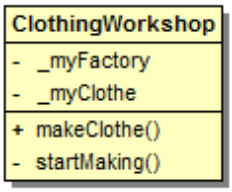
Tous les éléments du kit sont spécifiques à chaque galaxie, sauf le type d'assemblage qui ne dépend que de la gamme (Standard ou Deluxe).

Concernant les vêtements: peu de changements par rapport à Factory Method:



La classe Clothe possède tous ses attributs, elle est donc capable de se décrire elle-même complètement, mais ce n'est pas elle qui définira ses attributs. Comme décidé, cette classe n'a plus aucune initiative. On lui demandera simplement d'exécuter une à une les étapes de sa fabrication.

Le workshop est peu modifié également:

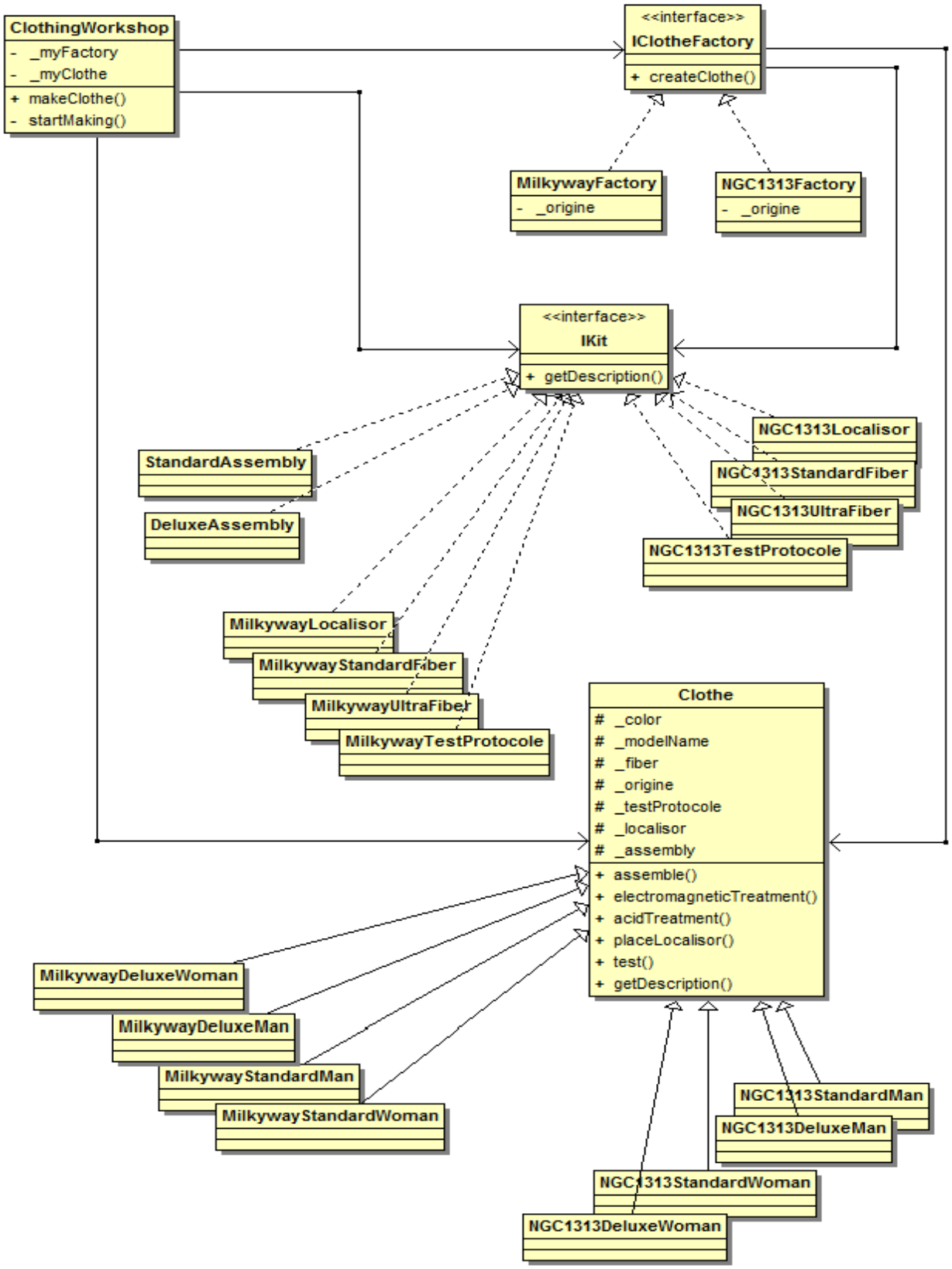


Il possède une référence vers la Factory qui lui sera attribuée.

La méthode makeClothe() sert à demander au workshop de fabriquer un vêtement.

startMaking() est la méthode qui connaît le processus de fabrication.

Le montage complet nous donne ceci:



Principe de fonctionnement:

- Le contrôleur (non représenté sur le diagramme de classes) invoque la méthode ClothingWorkshop.makeClothe() pour fabriquer un vêtement, en lui passant le modèle et la couleur.
- MakeClothe() invoque la Factory qui va instancier le vêtement ainsi que les éléments du kit, et retourner le tout au workshop.
- Le workshop invoque sa méthode privée startMaking() qui va appliquer les étapes de fabrication au vêtement.
- Enfin le vêtement fabriqué est retourné au contrôleur.

9.1 Codage

ClothingWorkshop:

```php
<?php

class ClothingWorkshop {

    private $_myFactory;
    private $_myClothe;

    public function __construct($factory) {
        $this->_myFactory = $factory;
    }

    public function makeClothe($clotheModel, $color) {
        $this->_myClothe = $this->_myFactory->createClothe($clotheModel,
                                                           $color);

        $this->startMaking();
        return $this->_myClothe;
    }

    private function startMaking() {
        $this->_myClothe->assemble();
        $this->_myClothe->electromagneticTreatment();
        $this->_myClothe->acidTreatment();
        $this->_myClothe->placeLocalisor();
        $this->_myClothe->test();
    }

}

?>
```

Il faudra passer une Factory au constructeur du Workshop.

On utilise la Factory pour obtenir le vêtement.

C'est ici que se trouve le processus de fabrication.

L'interface IclotheFactory:

```php
<?php

interface IClotheFactory {

    public function createClothe($clotheModel, $color);
}

?>
```

MilkywayFactory:

```php
<?php

class MilkywayFactory implements IClotheFactory {

    private $_origine = 'Milkyway';

    public function createClothe($clotheModel, $color) {
        $clothe = null;

        switch ($clotheModel) {
            case "STANDARD_WOMAN":
                $clothe = new MilkywayStandardWoman($color,
                                new MilkywayStandardFiber(),
                                $this->_origine,
                                new MilkywayLocalisor(),
                                new MilkywayTestProtocole(),
                                new StandardAssembly());
                break;
            case "DELUXE_WOMAN":
                $clothe = new MilkywayDeluxeWoman($color,
                                new MilkywayUltraFiber(),
                                $this->_origine,
                                new MilkywayLocalisor(),
                                new MilkywayTestProtocole(),
                                new DeluxeAssembly());
                break;
            case "STANDARD_MAN":
                $clothe = new MilkywayStandardMan($color,
                                new MilkywayStandardFiber(),
                                $this->_origine,
                                new MilkywayLocalisor(),
                                new MilkywayTestProtocole(),
                                new StandardAssembly());
                break;
            case "DELUXE_MAN":
                $clothe = new MilkywayDeluxeMan($color,
                                new MilkywayUltraFiber(),
                                $this->_origine,
                                new MilkywayLocalisor(),
                                new MilkywayTestProtocole(),
                                new DeluxeAssembly());
        }  // end switch

        return $clothe;
    }

}

?>
```

Chaque Factory défini son origine.

La Factory sait ce qu'il faut instancier selon le contexte.

NGC1313Factory:

```php
<?php

class NGC1313Factory implements IClotheFactory {

    private $_origine = 'NGC1313';

    public function createClothe($clotheModel, $color) {
        $clothe = null;

        switch ($clotheModel) {
            case "STANDARD_WOMAN":
                $clothe = new NGC1313StandardWoman($color,
                                new NGC1313StandardFiber(),
                                $this->_origine,
                                new NGC1313Localisor(),
                                new NGC1313TestProtocole(),
                                new StandardAssembly());
                break;
            case "DELUXE_WOMAN":
                $clothe = new NGC1313DeluxeWoman($color,
                                new NGC1313UltraFiber(),
                                $this->_origine,
                                new NGC1313Localisor(),
                                new NGC1313TestProtocole(),
                                new DeluxeAssembly());
                break;
            case "STANDARD_MAN":
                $clothe = new NGC1313StandardMan($color,
                                new NGC1313StandardFiber(),
                                $this->_origine,
                                new NGC1313Localisor(),
                                new NGC1313TestProtocole(),
                                new StandardAssembly());
                break;
            case "DELUXE_MAN":
                $clothe = new NGC1313DeluxeMan($color,
                                new NGC1313UltraFiber(),
                                $this->_origine,
                                new NGC1313Localisor(),
                                new NGC1313TestProtocole(),
                                new DeluxeAssembly());
        }  // end switch

        return $clothe;
    }

}

?>
```

IKit:

```php
<?php

interface IKit {
    public function getDescription();
}

?>
```

DeluxeAssembly:

```php
<?php

class DeluxeAssembly implements IKit {

    public function getDescription() {
        return 'Deluxe assembly style.' . "</br>";
    }

}

?>
```

StandardAssembly:

```php
<?php

class StandardAssembly implements IKit {

    public function getDescription() {
        return 'Standard assembly style.' . "</br>";
    }

}

?>
```

NGC1313UltraFiber:

```php
<?php

class NGC1313UltraFiber implements IKit {

    public function getDescription() {
        return 'NGC1313 arachno-fibre ultra haute résistance.';
    }

}

?>
```

NGC1313Localisor:

```php
<?php

class NGC1313Localisor implements IKit {

    public function getDescription() {
        return 'NGC1313 localisor model.' . "</br>";
    }

}

?>
```

NGC1313TestProtocole:

```php
<?php

class NGC1313TestProtocole implements IKit {

    public function getDescription() {
        return 'NGC1313 test protocole.' . "</br>";
    }

}

?>
```

NGC1313StandardFiber:

```php
<?php

class NGC1313StandardFiber implements IKit {

    public function getDescription() {
        return 'NGC1313 arachno-fibre haute résistance.';
    }

}

?>
```

MilkywayLocalisor:

```php
<?php

class MilkywayLocalisor implements IKit {

    public function getDescription() {
        return 'Milkyway localisor model.' . "</br>";
    }

}

?>
```

MilkywayStandardFiber:

```php
<?php

class MilkywayStandardFiber implements IKit {

    public function getDescription() {
        return 'Milkyway fibre standard haute résistance.';
    }

}

?>
```

MilkywayTestProtocole:

```php
<?php

class MilkywayTestProtocole implements IKit {

    public function getDescription() {
        return 'Milkyway test protocole.' . "</br>";
    }

}

?>
```

MilkywayUltraFiber:

```php
<?php

class MilkywayUltraFiber implements IKit {

    public function getDescription() {
        return 'Milkyway fibre standard ultra haute résistance.';
    }

}

?>
```

Clothe:

```php
<?php

class Clothe {

    protected $_color = null;            // Chaine.
    protected $_fiber = null;            // Object.
    protected $_origine = null;          // Chaine.
    protected $_localisor = null;        // Object.
    protected $_testProtocole = null;    // Object.
    protected $_assembly = null;         // Object
    protected $_modelName = null;        // Chaine.

    public function __construct($color, $fiber, $origine,
                $localisor, $testProtocole, $assembly) {

        $this->_fiber = $fiber;
        $this->_origine = $origine;
        $this->_localisor = $localisor;
        $this->_testProtocole = $testProtocole;
        $this->_assembly = $assembly;

        switch ($color) {
            case "YEL":
                $this->_color = "Jaune";
                break;
            case "RED":
                $this->_color = "Rouge";
                break;
            case "BLU":
                $this->_color = "Bleu";
            default :
                $this->_color = "Bleu";
        }  // end switch
    }

    public function assemble() {
        echo 'Assemblage: ' . $this->_assembly->getDescription();
    }

    public function electromagneticTreatment() {
        echo 'Traitement anti champs electromagnetiques.' . "</br>";
    }

    public function acidTreatment() {
        echo 'Traitement anti acide.' . "</br>";
    }

    public function placeLocalisor() {
        echo 'Positionnement du Localisor: ' .
                $this->_localisor->getDescription();
    }

    public function test() {
        echo 'Protocole de tests: ' .
                $this->_testProtocole->getDescription();
    }

    public function getDescription() {
        $description = "  " . 'Modele: ' .
                $this->_modelName . "</br>";
        $description .= "  " . 'Couleur: ' .
                $this->_color . "</br>";
        $description .= "  " . 'Fibre type: ' .
                    $this->_fiber->getDescription() . "</br>";
        $description .= "  " . 'Origine: ' .
                    $this->_origine . "</br>";
        $description .= "  " . 'Assemblage: ' .
                    $this->_assembly->getDescription();
        $description .= "  " . 'Protocole de tests: ' .
                    $this->_testProtocole->getDescription();
        return $description;
    }

}

?>
```

C'est la Factory qui fourni au vêtement, tous les éléments du kit de fabrication.

MilkywayStandardMan:

```php
<?php

class MilkywayStandardMan extends Clothe {

    public function __construct($color, $fiber, $origine,
            $localisor, $testProtocole, $assembly) {

        parent::__construct($color, $fiber, $origine,
            $localisor, $testProtocole, $assembly);
        $this->_modelName = "Tenue Standard Homme.";

    }

}

?>
```

On appelle le constructeur de la classe mère.

.. et on complète en définissant le nom du modèle de vêtement, seul attribut qui est défini par le vêtement lui-même.

MilkywayDeluxeWoman:

```php
<?php

class MilkywayDeluxeWoman extends Clothe {

    public function __construct($color, $fiber, $origine,
            $localisor, $testProtocole, $assembly) {

        parent::__construct($color, $fiber, $origine,
            $localisor, $testProtocole, $assembly);
        $this->_modelName = "Tenue Deluxe Femme.";

    }

}

?>
```

MilkywayStandardWoman:

```php
<?php

class MilkywayStandardWoman extends Clothe {

    public function __construct($color, $fiber, $origine,
            $localisor, $testProtocole, $assembly) {

        parent::__construct($color, $fiber, $origine,
            $localisor, $testProtocole, $assembly);
        $this->_modelName = "Tenue Standard Femme.";

    }

}

?>
```

MilkywayDeluxeMan:

```php
<?php

class MilkywayDeluxeMan extends Clothe {

    public function __construct($color, $fiber, $origine,
            $localisor, $testProtocole, $assembly) {

        parent::__construct($color, $fiber, $origine,
                $localisor, $testProtocole, $assembly);
        $this->_modelName = "Tenue Deluxe Homme.";
    }

}

?>
```

NGC1313StandardWoman:

```php
<?php

class NGC1313StandardWoman extends Clothe {

    public function __construct($color, $fiber, $origine,
            $localisor, $testProtocole, $assembly) {

        parent::__construct($color, $fiber, $origine,
            $localisor, $testProtocole, $assembly);
        $this->_modelName = "Tenue Standard Femme.";

    }

}

?>
```

NGC1313StandardMan:

```php
<?php

class NGC1313StandardMan extends Clothe {

    public function __construct($color, $fiber, $origine,
            $localisor, $testProtocole, $assembly) {

        parent::__construct($color, $fiber, $origine,
            $localisor, $testProtocole, $assembly);
        $this->_modelName = "Tenue Standard Homme.";

    }

}

?>
```

NGC1313DeluxeWoman:

```php
<?php

class NGC1313DeluxeWoman extends Clothe {

    public function __construct($color, $fiber, $origine,
            $localisor, $testProtocole, $assembly) {

        parent::__construct($color, $fiber, $origine,
            $localisor, $testProtocole, $assembly);
        $this->_modelName = "Tenue Deluxe Femme.";

    }

}

?>
```

NGC1313DeluxeMan:

```php
<?php

class NGC1313DeluxeMan extends Clothe {

    public function __construct($color, $fiber, $origine,
            $localisor, $testProtocole, $assembly) {

        parent::__construct($color, $fiber, $origine,
            $localisor, $testProtocole, $assembly);
        $this->_modelName = "Tenue Deluxe Homme.";

    }

}

?>
```

9.2 Tests

Comme d'habitude nous allons utiliser index.php comme contrôleur de l'ensemble du montage:

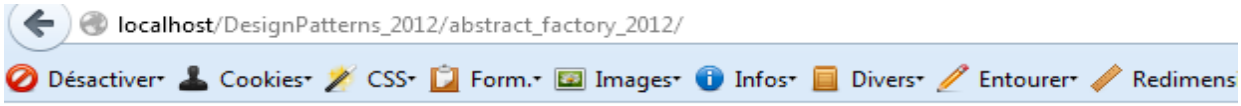
```php
<?php

//************************************************************
// ABSTRACT FACTORY pattern controler
//************************************************************

require_once 'includePaths.php';
$newline = "</br>";

echo 'Controleur: Debut traitement.' . $newline . $newline;

// Instanciations des factory:
$myMilkyWayFactory = new MilkywayFactory;
$myNGC1313Factory = new NGC1313Factory;

// Instanciations des workshops:
$myMilkyWayWorkshop = new ClothingWorkshop($myMilkyWayFactory);
$myNGC1313Workshop = new ClothingWorkshop($myNGC1313Factory);

// Traitements:
echo "clothe_1:" . $newline;
$clothe_1 = $myMilkyWayWorkshop->makeClothe("STANDARD_MAN", "YEL");
echo 'Description: ' . $newline;
echo $clothe_1->getDescription() . $newline;
echo "************" . $newline;

echo "clothe_2:" . $newline;
$clothe_2 = $myNGC1313Workshop->makeClothe("DELUXE_WOMAN", "BLU");
echo 'Description: ' . $newline;
echo $clothe_2->getDescription() . $newline;
echo "************" . $newline;

echo "clothe_3:" . $newline;
$clothe_3 = $myNGC1313Workshop->makeClothe("STANDARD_MAN", "RED");
echo 'Description: ' . $newline;
echo $clothe_3->getDescription() . $newline;
echo "************" . $newline;

echo 'Controleur: Fin traitement.';

?>
```

Instanciation des Factory.

Instanciation des Workshop en leur passant chacun sa Factory.

La création des vêtements est très simple.

Voici le résultat de l'exécution:



Controleur: Debut traitement.

clothe_1:
Assemblage: Standard assembly style.
Traitement anti champs electromagnetiques.
Traitement anti acide.
Positionnement du Localisor: Milkyway localisor model.
Protocole de tests: Milkyway test protocole.
Description:
  Modele: Tenue Standard Homme.
  Couleur: Jaune
  Fibre type: Milkyway fibre standard haute résistance.
  Origine: Milkyway
  Assemblage: Standard assembly style.
  Protocole de tests: Milkyway test protocole.

Les phases du processus de fabrication sont contrôlées par le Workshop, mais avec des spécificités exprimées par le vêtement, et imposées par la Factory.

Le vêtement fourni la description de ses éléments de fabrication, dont certains ont été déterminés par la Factory.

****
clothe_2:
Assemblage: Deluxe assembly style.
Traitement anti champs electromagnetiques.
Traitement anti acide.
Positionnement du Localisor: NGC1313 localisor model.
Protocole de tests: NGC1313 test protocole.
Description:
  Modele: Tenue Deluxe Femme.
  Couleur: Bleu
  Fibre type: NGC1313 arachno-fibre ultra haute résistance.
  Origine: NGC1313
  Assemblage: Deluxe assembly style.
  Protocole de tests: NGC1313 test protocole.

****
clothe_3:
Assemblage: Standard assembly style.
Traitement anti champs electromagnetiques.
Traitement anti acide.
Positionnement du Localisor: NGC1313 localisor model.
Protocole de tests: NGC1313 test protocole.
Description:
  Modele: Tenue Standard Homme.
  Couleur: Rouge
  Fibre type: NGC1313 arachno-fibre haute résistance.
  Origine: NGC1313
  Assemblage: Standard assembly style.
  Protocole de tests: NGC1313 test protocole.

****
Controleur: Fin traitement.

Avec ce nouveau système ils pourront sortir régulièrement de nouvelles collections.

Bon assez parlé chiffon. Maintenant j'aimerai une vrai mission.

10 PATTERN COMMAND

Généralement, à une demande correspond une action qui est toujours la même. C'est un lien fort et immuable, donc considéré comme statique.

Mais il est des cas où ce lien doit pouvoir être modifié en fonction d'un contexte. Par exemple:

- Une demande doit déclencher une autre action que celle prévue initialement.
- Une demande doit déclencher non plus une, mais plusieurs actions.

Le pattern Command, en décorrélant la demande de l'action, va permettre d'intégrer facilement ce type de contraintes, en ajoutant de l'intelligence entre les deux.

Pour résumer on peut voir ce pattern comme une télécommande programmable: chaque bouton déclenche une ou plusieurs actions, et on peut reprogrammer chaque bouton à tout moment.

Il ne reste plus qu'à attendre la prochaine occasion pour utiliser ce merveilleux pattern.

Voici ce que le capitaine souhaite contrôler à distance:

- La vitesse de propulsion de l'Enterprise: sur une échelle de 0 à 5 (5 étant l'hyper-vitesse).
- Le bouclier (ouvert/fermé).
- L'invisibilité de l'Enterprise (activée/désactivée).
- L'écran de la salle de contrôle (allumé/éteint).

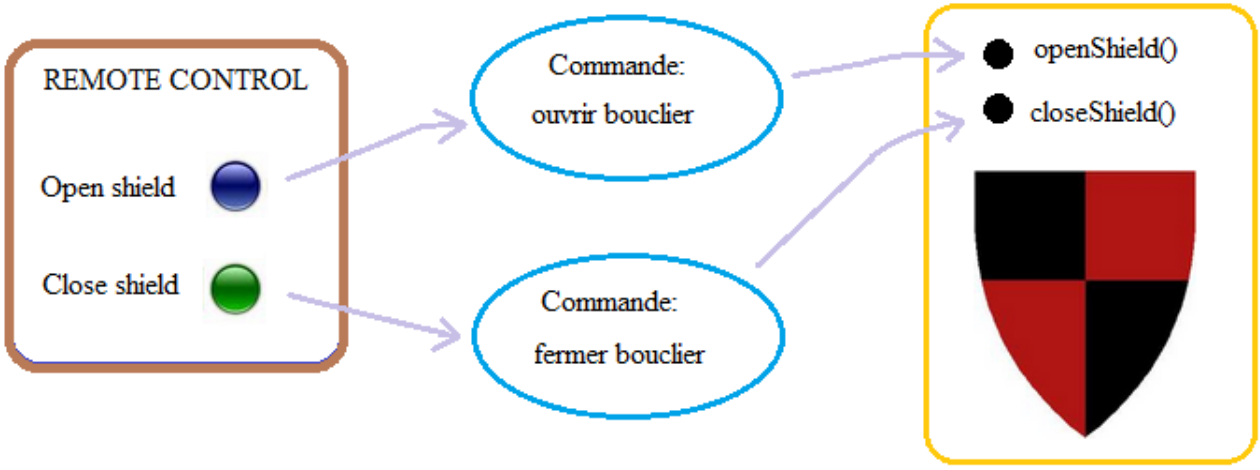
Pour comprendre le fonctionnement du pattern Command, prenons par exemple le bouclier de l'Enterprise (en anglais: shield).

Imaginons une télécommande à 2 boutons:

- Un bouton déclenche l'ouverture du bouclier.
- Un bouton déclenche la fermeture du bouclier.

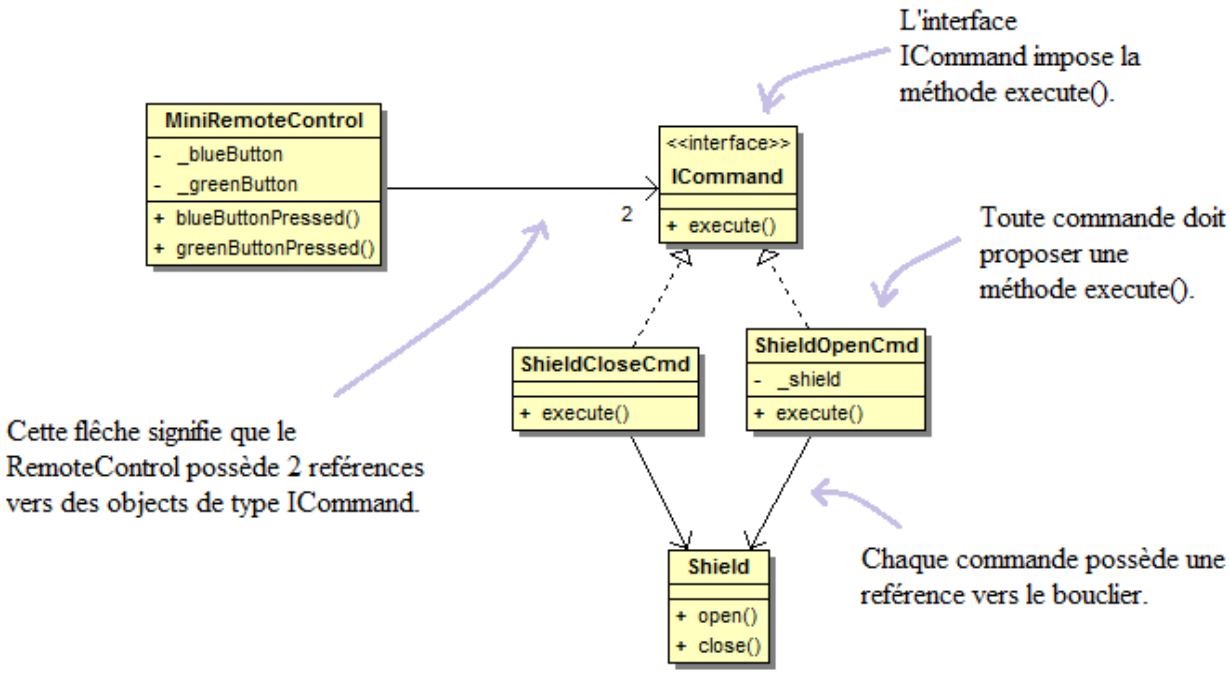
Le jeu se joue à 3:

- La télécommande.
- La commande (c'est à dire l'action déclenchée).
- L'appareil contrôlé (ici le bouclier)



- Chaque bouton de la télécommande est lié à une et une seule commande.
- Chaque commande sait quel appareil elle doit contrôler et quelle action elle doit déclencher.
- La télécommande n'a aucune connaissance de l'appareil qui est contrôlé.

Le principe est simple. Voyons maintenant comment transformer cela en modèle de classes.



MiniRemoteControl:

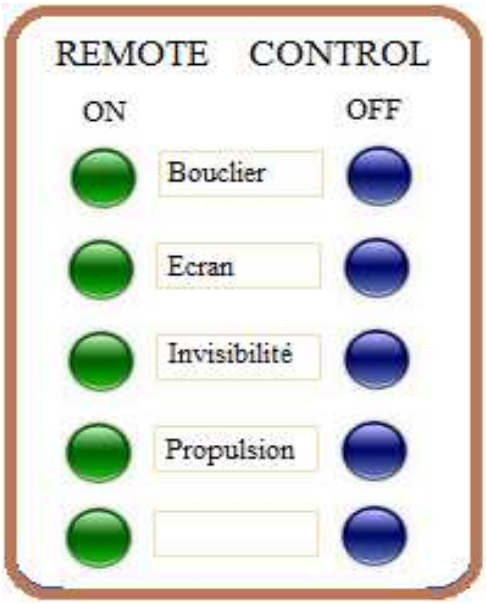
- Les boutons sont matérialisés par deux attributs privés: _blueButton et _greenButton.
- Chaque bouton est lié à un objet de type ICommand.
- Pour appuyer sur un bouton il faudra invoquer blueButtonPressed() ou greenButtonPressed().

Voyons ce qui se passe si on appuie sur le bouton bleu, qui correspond à l'ouverture du bouclier:

- L'événement déclencheur est donc l'invocation de la méthode blueButtonPressed().
- La méthode blueButtonPressed() invoque à son tour la méthode execute() du bouton bleu _blueButton. En effet _blueButton n'est autre qu'une référence vers un objet de type ICommand, et plus précisément d'un objet ShieldOpenCmd.
- C'est donc en fait la méthode execute() de la commande ShieldOpenCmd, qui a été exécutée. Cette méthode agit directement sur le bouclier en invoquant la méthode open().

Le principe de base est simple. Adaptons le maintenant à la demande du capitaine.

Voici à quoi devrait ressembler notre télécommande:



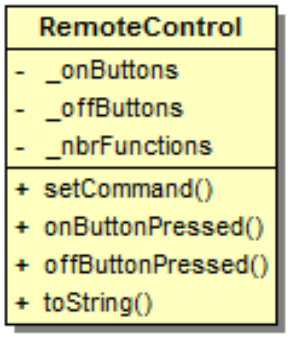
On peut faire les remarques suivantes:

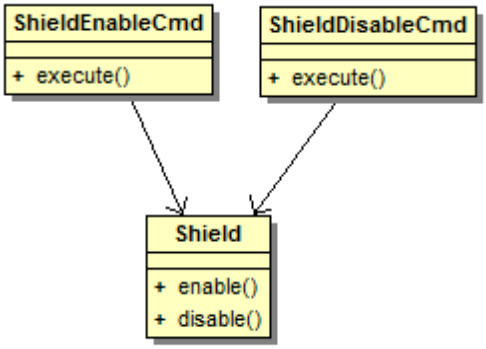
- Il peut y avoir des boutons de la télécommande qui sont inutilisés. Il faudra prévoir cela.
- La propulsion n'est pas une commande de type on/off, mais peut prendre une valeur sur une échelle de 0 à 4. Il faudra un traitement spécial.
- Il nous faudra quelque chose pour gérer les liaisons entre boutons et commandes.

Nous allons enrichir notre classe RemoteControl:

- Pour gérer les liens bouton/commande, nous utiliserons deux tableaux en mémoire centrale: onButtons et offButtons. Ces tableaux contiendront les références vers les objets ICommand, pour chaque bouton.
- Nous aurons un attribut qui indiquera le nombre de fonctions que la télécommande peut gérer (c'est à dire le nombre de couples de boutons on/off).
- Nous aurons besoin d'une méthode permettant d'attribuer une commande à un bouton. Appelons-là setCommand(). Cette méthode permettra de programmer ou de reprogrammer la télécommande.
- Enfin il faut pouvoir signifier l'appui sur un bouton. Pour cela nous ajoutons deux méthodes publiques onButtonPressed() et offButtonPressed(). Chacune prendra en paramètre l'indice du bouton concerné, dans les tableaux en mémoire centrale.
- A titre utilitaire nous ajoutons une méthode toString() qui affichera la configuration de la télécommande.

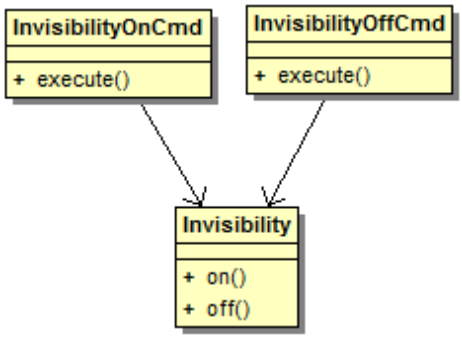
Notre classe RemoteControl devient:



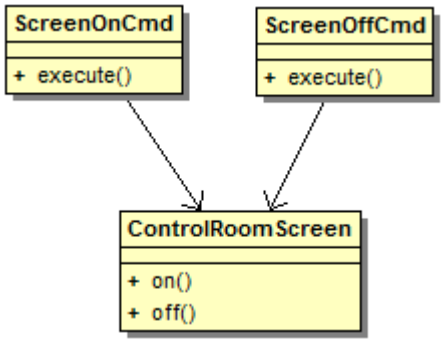


Les deux commandes qui contrôlent le bouclier:

Les deux commandes qui contrôlent l'invisibilité:



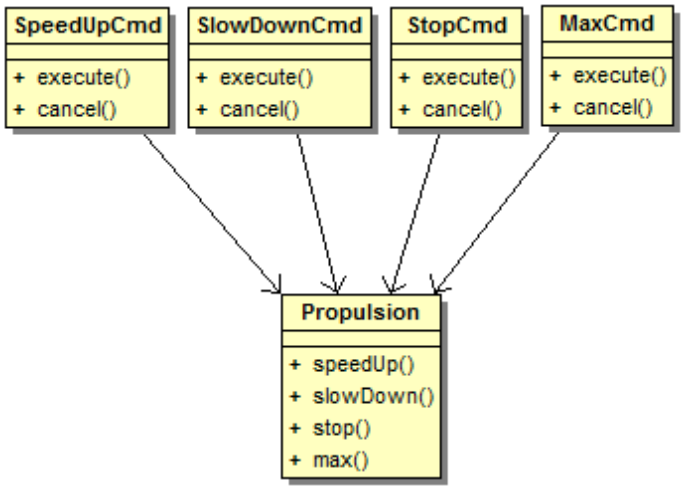
Les deux commandes qui contrôlent l'écran de la salle de contrôle:



Concernant la vitesse de l'Enterprise, la classe Propulsion nous propose quatre méthodes:

- speedUp() augmente la vitesse d'un niveau.
- slowDown() diminue la vitesse d'un niveau.
- stop() diminue la vitesse plusieurs fois jusqu'à l'arrêt.
- max() augmente la vitesse plusieurs fois jusqu'à atteindre la vitesse maximum.

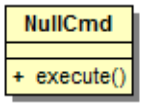
Nous créons donc quatre commandes.



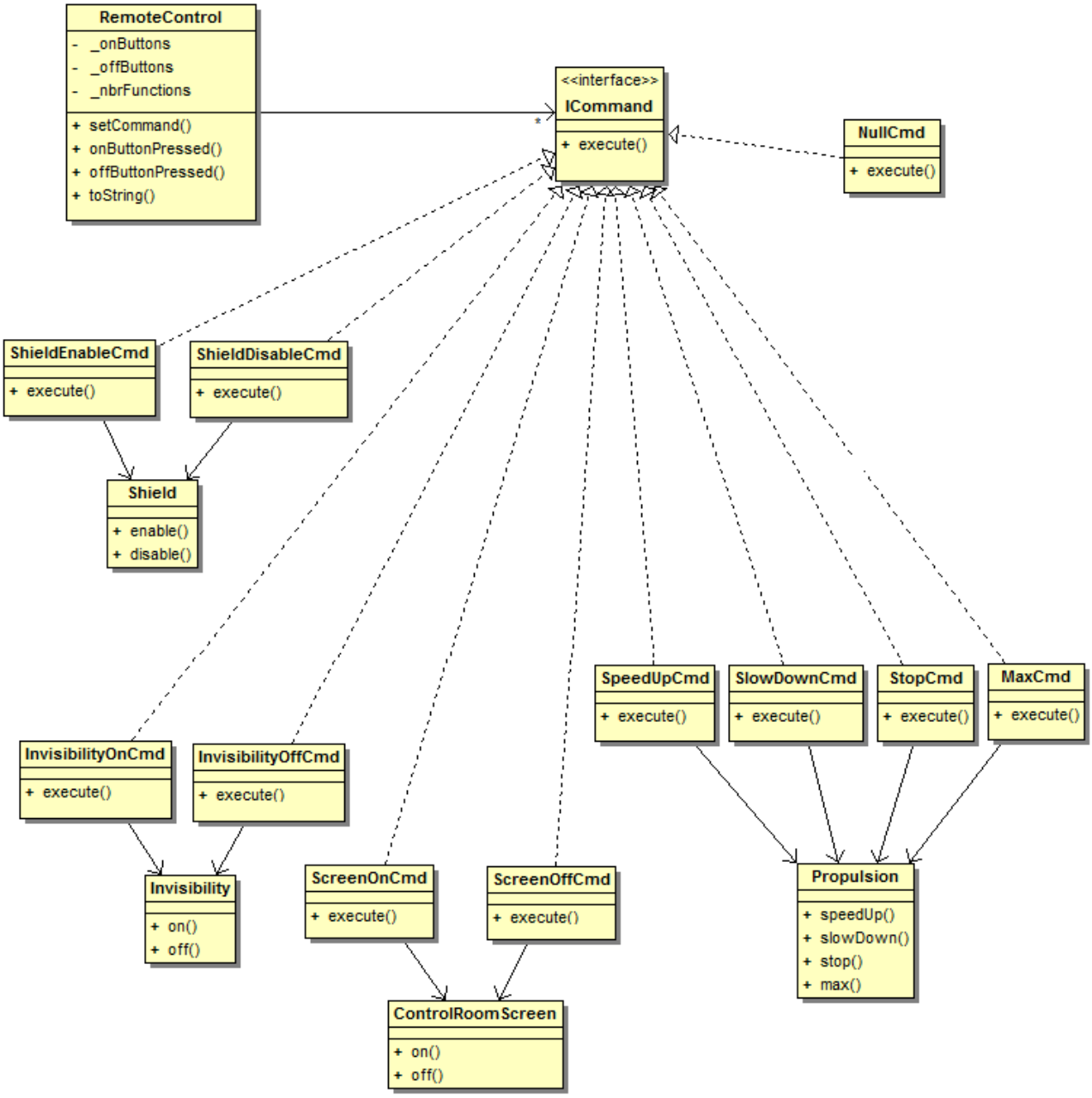
Il y a maintenant le problème des boutons inutilisés. Si on appuie sur un bouton qui n'est pas programmé, il ne doit rien se passer. Hors dans notre pattern tout appui sur un bouton invoque la méthode execute() de la commande associée à ce bouton.

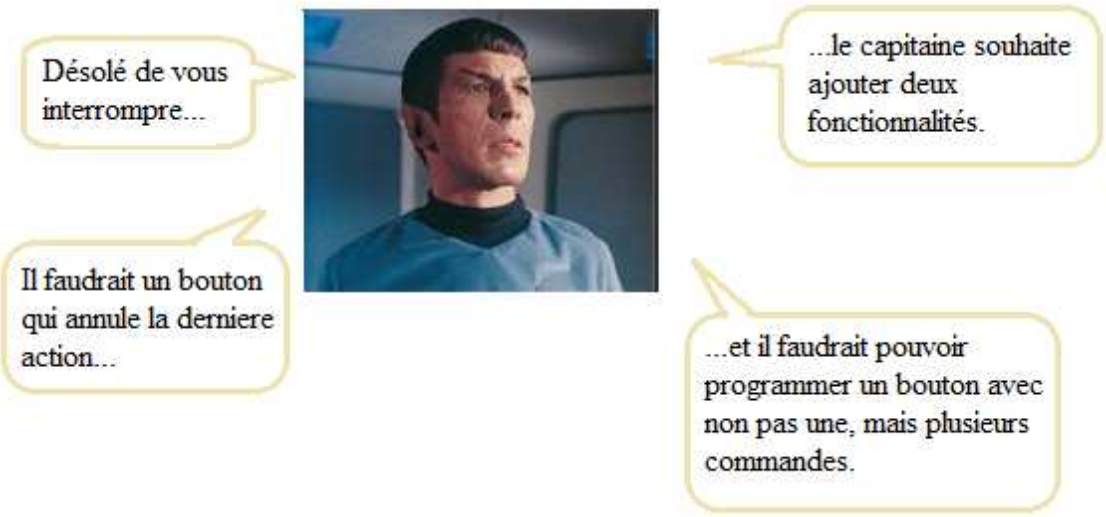
Une première approche serait de tester si le bouton possède bien une commande, avant d'invoquer cette commande.

Mais il y a mieux: créons une commande qui ne fait rien, et nommons-la NullCmd. Comme toute commande, elle possède une méthode execute(), mais celle-ci ne fera rien. Il n'y a donc pas de test à mettre en place, à condition que tout bouton inutilisé soit bien associé à la commande NullCmd.



Voici le montage complet:





Bien bien...

Après discussion avec le capitaine, il s'agit de programmer un bouton qui doit mettre l'Enterprise en mode "danger", c'est à dire:

- Vitesse maximum.
- Bouclier activé.
- Invisibilité activée.

L'inverse du mode "danger" est le mode "normal":

- Vitesse lente.
- Bouclier désactivé.
- Invisibilité désactivée.

Quant au bouton "Annulation" il doit simplement effectuer l'inverse de la dernière commande exécutée.

Heureusement, grâce au design pattern Command, ces deux nouvelles fonctionnalités ne nous poserons aucun problème.

Commençons par le bouton d'annulation (en anglais: cancel):

- La télécommande doit mémoriser la dernière commande exécutée pour être capable de l'annuler. Nous ajoutons donc un attribut privé lastCommand qui contiendra la référence de la dernière commande exécutée. De ce fait chaque exécution de commande devra mettre à jour lastCommand.
- La télécommande doit également disposer d'un bouton "Annuler", qui sera matérialisé par

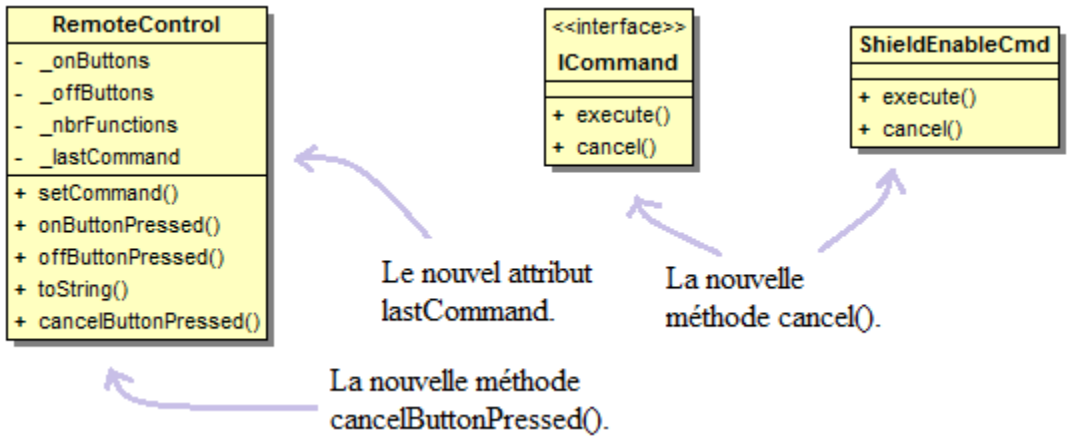
une méthode cancelButtonPressed().

- Qui doit savoir ce qu'il faut faire pour annuler une commande ? C'est la commande elle-même. Nous devons faire en sorte que chaque commande sache comment annuler son action. Pour cela ajoutons une méthode cancel() à l'interface ICommand. Cette méthode devra être implémentée par toutes les commandes.

Voici ce qui doit se passer quand on appuie sur le bouton Cancel de la télécommande:

- La méthode cancelButtonPressed() de la télécommande, est invoquée.
- Cette méthode invoque à son tour la méthode cancel() sur l'attribut lastCommand. En effet cet attribut n'est autre qu'une référence sur un objet de type ICommand.
- La méthode cancel() de la commande, agit sur l'appareil contrôlé, de manière à annuler son action.

Voici les modifications apportées aux classes. Ici une seule commande est représentée, mais il est bien entendu que toutes les commandes doivent implémenter la nouvelle méthode cancel().



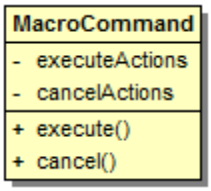
Intéressons-nous maintenant à la commande devant lancer plusieurs actions.

L'adaptation pour atteindre ce résultat, est extrêmement simple. Il suffit de créer une commande dont les méthodes execute() et cancel() effectuent plusieurs actions.

Pour que cette commande sache ce qu'elle doit faire pour chacune des deux méthodes, nous là dotons de deux attributs privés de type tableau, qui contiendront les références des commandes à exécuter.

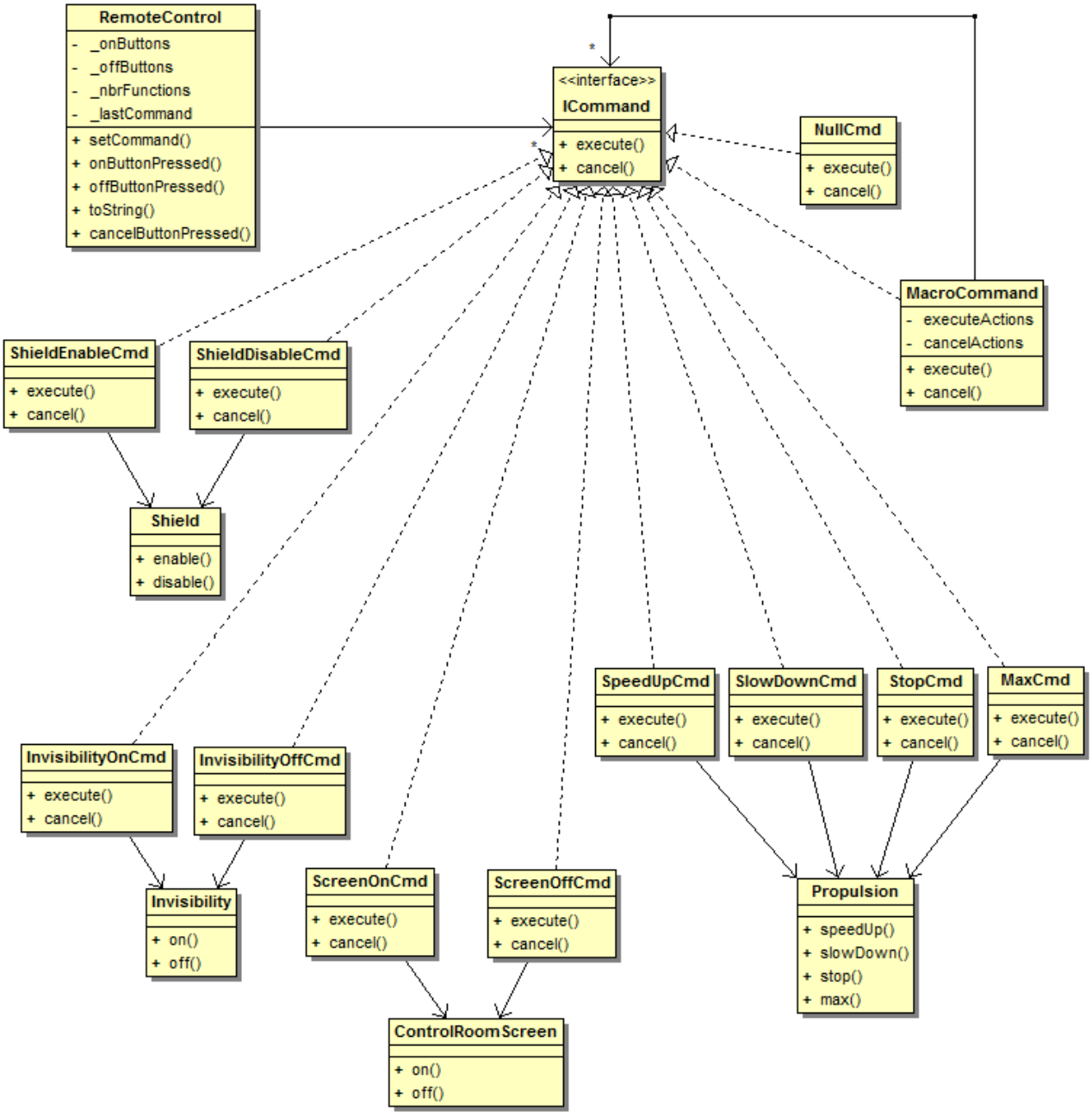
En résumé cette macro commande ne fait qu'exécuter d'autres commandes. Etant elle-même une commande, elle propose les méthodes execute() et cancel().

Voici cette nouvelle classe:



Avant de passer au codage, voici le modèle de classes final:

10.1 Diagramme de classe



Quelques remarques:

- Chaque commande contrôle directement un appareil.
- La macro commande ne contrôle pas directement un appareil, car elle ne fera qu'exécuter d'autres commandes dont la liste sera indiquée dans les tableaux executeActions et cancelActions. C'est ce qu'indique la flèche continue reliant MacroCommand à ICommand.

Cette flèche signifie que MacroCommand possède plusieurs références vers des objets de type ICommand.

- La classe Propulsion proposant quatre fonctions, nous avons créé quatre commandes.

10.2 Codage

La classe Invisibility:

```php
<?php

class Invisibility {

    private $_name = '';

    public function __construct($aName) {
        $this->_name = $aName;
    }

    public function on() {
        echo get_class() . ': ' . $this->_name . ' activé. </br>';
    }

    public function off() {
        echo get_class() . ': ' . $this->_name . ' désactivé. </br>';
    }

}

?>
```

L'attribut _name permet de nommer l'objet instancié.

La classe ControlRoomScreen:

```php
<?php

class ControlRoomScreen {

    private $_name = '';

    public function __construct($aName) {
        $this->_name = $aName;
    }

    public function on() {
        echo get_class() . ': ' . $this->_name . ' allumé. </br>';
    }

    public function off() {
        echo get_class() . ': ' . $this->_name . ' éteint. </br>';
    }

}

?>
```

La classe Propulsion:

```php
<?php

class Propulsion {

    private $_name = '';
    private $_currentSpeed;
    private $_speedLevels = array("OFF", "Slow", "Medium", "Fast", "Light speed");
    private $_maxSpeed;

    public function __construct($aName) {
        $this->_name = $aName;
        $this->_currentSpeed = 0;
        $this->_maxSpeed = count($this->_speedLevels) - 1;
    }

    public function speedUp() {
        $this->_currentSpeed++;
        $alert = "";
        if ($this->_currentSpeed > $this->_maxSpeed) {
            $this->_currentSpeed = $this->_maxSpeed;
            $alert = ". Impossible d'accélérer.";
        }
        $this->log($alert);
    }

    public function slowDown() {
        $this->_currentSpeed--;
        $alert = "";
        if ($this->_currentSpeed < 0) {
            $this->_currentSpeed = 0;
            $alert = ". Impossible de ralentir.";
        }
        $this->log($alert);
    }

    public function stop() {
        while ($this->_currentSpeed > 0) {
            $this->slowDown();
        }
    }

    public function max() {
        while ($this->_currentSpeed < $this->_maxSpeed) {
            $this->speedUp();
        }
    }

    private function log($alert) {
        echo get_class() . ': ' . $this->_name . ": "
        . $this->_speedLevels[$this->_currentSpeed]
        . $alert . '</br>';
    }

}

?>
```

Un tableau matérialise les différentes vitesses.

Le constructeur fait quelques initialisations.

Pour accélerer on incrémente currentSpeed, en vérifiant si on atteint la limite.

Idem pour la décélération.

La classe Shield:

```php
<?php

class Shield {

    private $_name = '';

    public function __construct($aName) {
        $this->_name = $aName;
    }

    public function enable() {
        echo get_class() . ': ' . $this->_name . ' activé. </br>';
    }

    public function disable() {
        echo get_class() . ': ' . $this->_name . ' désactivé. </br>';
    }

}

?>
```

La classe MacroCommandCmd:

```php
<?php

class MacroCommandCmd implements ICommand {

    private $_executeCommands = array();
    private $_cancelCommands = array();

    public function __construct($executeActions, $cancelActions) {
        $this->_executeCommands = $executeActions;
        $this->_cancelCommands = $cancelActions;
    }

    public function execute() {
        foreach ($this->_executeCommands as $i => $command) {
            echo "** Macro: ";
            $command->execute();
        }
    }

    public function cancel() {
        foreach ($this->_cancelCommands as $i => $command) {
            echo "** Macro: ";
            $command->execute();
        }
    }

}

?>
```

On passe au constructeur, les deux listes d'actions.

On execute simplement la liste de commandes.

La classe NullCmd:

```php
<?php

class NullCmd implements ICommand {

    public function execute() {
        echo "Je suis une commande qui ne fait rien.";
    }

    public function cancel() {
        echo "Je suis une commande qui ne fait rien.";
    }

}

?>
```

L'interface ICommand:

```php
<?php

interface ICommand {

    public function execute();
    public function cancel();
}

?>
```

La commande ShieldEnableCmd:

```php
<?php

class ShieldEnableCmd implements ICommand {

    private $_myReceiver;

    public function __construct($aReceiver) {
        $this->_myReceiver = $aReceiver;
    }

    public function execute() {
        $this->_myReceiver->enable();
    }

    public function cancel() {
        $this->_myReceiver->disable();
    }

}

?>
```

L'appareil contrôlé (receiver) est passé au controleur de la commande.

La commande sait quelle action doit être executée sur l'appareil contrôlé.

La commande ShieldDisableCmd :

```php
<?php

class ShieldDisableCmd implements ICommand {

    private $_myReceiver;

    public function __construct($aReceiver) {
        $this->_myReceiver = $aReceiver;
    }

    public function execute() {
        $this->_myReceiver->disable();
    }

    public function cancel() {
        $this->_myReceiver->enable();
    }

}

?>
```

La classe ScreenOnCmd:

```php
<?php

class ScreenOnCmd implements ICommand {

    private $_myReceiver;

    public function __construct($aReceiver) {
        $this->_myReceiver = $aReceiver;
    }

    public function execute() {
        $this->_myReceiver->on();
    }

    public function cancel() {
        $this->_myReceiver->off();
    }

}

?>
```

La classe ScreenOffCmd:

```php
<?php

class ScreenOffCmd implements ICommand {

    private $_myReceiver;

    public function __construct($aReceiver) {
        $this->_myReceiver = $aReceiver;
    }

    public function execute() {
        $this->_myReceiver->off();
    }

    public function cancel() {
        $this->_myReceiver->on();
    }

}

?>
```

La commande InvisibilityOnCmd:

```php
<?php

class InvisibilityOnCmd implements ICommand {

    private $_myReceiver;

    public function __construct($aReceiver) {
        $this->_myReceiver = $aReceiver;
    }

    public function execute() {
        $this->_myReceiver->on();
    }

    public function cancel() {
        $this->_myReceiver->off();
    }

}

?>
```

La commande InvisibilityOffCmd:

```php
<?php

class InvisibilityOffCmd implements ICommand {

    private $_myReceiver;

    public function __construct($aReceiver) {
        $this->_myReceiver = $aReceiver;
    }

    public function execute() {
        $this->_myReceiver->off();
    }

    public function cancel() {
        $this->_myReceiver->on();
    }

}

?>
```

La commande MaxCmd:

```php
<?php

class MaxCmd implements ICommand {

    private $_myReceiver;

    public function __construct($aReceiver) {
        $this->_myReceiver = $aReceiver;
    }

    public function execute() {
        $this->_myReceiver->max();
    }

    public function cancel() {

    }

}

?>
```

La commande StopCmd:

```php
<?php

class StopCmd implements ICommand {

    private $_myReceiver;

    public function __construct($aReceiver) {
        $this->_myReceiver = $aReceiver;
    }

    public function execute() {
        $this->_myReceiver->stop();
    }

    public function cancel() {

    }

}

?>
```

La commande SpeedUpCmd:

```php
<?php

class SpeedUpCmd implements ICommand {

    private $_myReceiver;

    public function __construct($aReceiver) {
        $this->_myReceiver = $aReceiver;
    }

    public function execute() {
        $this->_myReceiver->speedUp();
    }

    public function cancel() {
        $this->_myReceiver->slowDown();
    }

}

?>
```

La commande SlowDownCmd:

```php
<?php

class SlowDownCmd implements ICommand {

    private $_myReceiver;

    public function __construct($aReceiver) {
        $this->_myReceiver = $aReceiver;
    }

    public function execute() {
        $this->_myReceiver->slowDown();
    }

    public function cancel() {
        $this->_myReceiver->speedUp();
    }

}

?>
```

La classe RemoteControl:

```php
<?php

class RemoteControl {

    private $_onButtons = array();
    private $_offButtons = array();
    private $_nbrFunctions;
    private $_lastCommand;

    public function __construct($nbrFunctions) {
        $this->_nbrFunctions = $nbrFunctions;
        $myNullCommand = new NullCmd();
        for ($i = 0; $i < $this->_nbrFunctions; $i++) {
            $this->_onButtons[$i] = $myNullCommand;
            $this->_offButtons[$i] = $myNullCommand;
        }
    }

    public function setCommand($functionIndex, $OnCommand, $OffCommand) {
        if ($functionIndex >= 0 && $functionIndex < $this->_nbrFunctions) {
            $this->_onButtons[$functionIndex] = $OnCommand;
            $this->_offButtons[$functionIndex] = $OffCommand;
        } else {
            echo get_class() . '->setCommand(): N° fonction inconnu: '
            . $functionIndex . '. Admis: de 0 à '
            . ($this->$_nbrFunctions - 1) . '.</br>';
        }
    }

    public function onButtonPressed($functionIndex) {
        if ($functionIndex >= 0 && $functionIndex < $this->_nbrFunctions) {
            $this->_onButtons[$functionIndex]->execute();
            $this->_lastCommand = $this->_onButtons[$functionIndex];
        } else {
            echo get_class() . '->onButtonPressed(): N° fonction inconnu: '
            . $functionIndex . '. Admis: de 0 à '
            . ($this->$_nbrFunctions - 1) . '.</br>';
        }
    }

    public function offButtonPressed($functionIndex) {
        if ($functionIndex >= 0 && $functionIndex < $this->_nbrFunctions) {
            $this->_offButtons[$functionIndex]->execute();
            $this->_lastCommand = $this->_offButtons[$functionIndex];
        } else {
            echo get_class() . '->offButtonPressed(): N° fonction inconnu: '
            . $functionIndex . '. Admis: de 0 à '
            . ($this->$_nbrFunctions - 1) . '.</br>';
        }
    }
```

Les 2 tableaux pour le stockage des commandes.

Le constructeur initialise toutes les commandes avec NullCommand.

setCommand() permet de définir une paire de commandes.

Quand on appuie sur un bouton, la commande correspondante est executée.

```
    public function __toString() {
        $strRetour = '</br>*****  Remote Control *****</br>';
        foreach ($this->_onButtons as $key => $value) {
            $strRetour .= 'function ' . $key . ' :  ON = '
                    . get_class($value) . ' OFF= '
                    . get class($this-> offButtons[$key]) . '<br/>';
        }
        return $strRetour;
    }

    public function cancelButtonPressed() {
        $this->_lastCommand->cancel();
    }

}

?>
```

toString() affiche la configuration de la télécommande.

Le bouton Cancel execute l'inverse de la derniere commande.

10.3 TESTS

Comme d'habitude nous allons utiliser index.php comme contrôleur de l'ensemble du montage:

```php
<?php

//************************************************************
// COMMAND pattern controler
//************************************************************

require_once 'includePaths.php';
$newline = "</br>";

echo 'Controleur: Debut traitement.', $newline;
echo '--------------', $newline;

// Instanciations:
$myRemoteControl = new RemoteControl(8);

$myNullCommand = new NullCmd();

$invisibilityGenerator = new Invisibility("InvisibilityGenerator");
$invisibilityOffCommand = new InvisibilityOffCmd($invisibilityGenerator);
$invisibilityOnCommand = new InvisibilityOnCmd($invisibilityGenerator);

$shield = new Shield("bouclier");
$shieldDisableCommand = new ShieldDisableCmd($shield);
$shieldEnableCommand = new ShieldEnableCmd($shield);

$screen = new ControlRoomScreen("écran");
$screenOnCommand = new ScreenOnCmd($screen);
$screenOffCommand = new ScreenOffCmd($screen);

$propulsion = new Propulsion("vitesse");
$slowDownCommand = new SlowDownCmd($propulsion);
$speedUpCommand = new SpeedUpCmd($propulsion);
$stopCommand = new StopCmd($propulsion);
$maxCommand = new MaxCmd($propulsion);

$dangerActionsList = array($shieldEnableCommand, $invisibilityOnCommand,
    $maxCommand);
$dangerCancelList = array($shieldDisableCommand, $invisibilityOffCommand,
    $stopCommand, $speedUpCommand);
$myOnDangerMacroCommand = new MacroCommandCmd($dangerActionsList,
        $dangerCancelList);
$myOffDangerMacroCommand = new MacroCommandCmd($dangerCancelList,
        $dangerActionsList);
```

On indique le nombre de fonctions progammables souhaité.

Les 2 listes d'actions pour la macro commande.

```php
// Configuration du RemoteControl:
$myRemoteControl->setCommand(0, $invisibilityOnCommand,
        $invisibilityOffCommand);
$myRemoteControl->setCommand(1, $shieldEnableCommand,
        $shieldDisableCommand);
$myRemoteControl->setCommand(2, $screenOnCommand, $screenOffCommand);
$myRemoteControl->setCommand(3, $speedUpCommand, $slowDownCommand);
$myRemoteControl->setCommand(4, $myOnDangerMacroCommand,
        $myOffDangerMacroCommand);

// traitemment:
$myRemoteControl->onButtonPressed(0); // invisibility
$myRemoteControl->offButtonPressed(0); // invisibility

$myRemoteControl->onButtonPressed(1); // bouclier
$myRemoteControl->offButtonPressed(1); // bouclier

$myRemoteControl->onButtonPressed(2); // écran
$myRemoteControl->offButtonPressed(2); // écran

$myRemoteControl->onButtonPressed(3); // vitesse
$myRemoteControl->onButtonPressed(3); // vitesse
$myRemoteControl->onButtonPressed(3); // vitesse
$myRemoteControl->onButtonPressed(3); // vitesse
$myRemoteControl->onButtonPressed(3); // vitesse

$myRemoteControl->offButtonPressed(3); // vitesse
$myRemoteControl->offButtonPressed(3); // vitesse
$myRemoteControl->offButtonPressed(3); // vitesse
$myRemoteControl->offButtonPressed(3); // vitesse
$myRemoteControl->offButtonPressed(3); // vitesse

$myRemoteControl->onButtonPressed(4);  // macro commande
$myRemoteControl->offButtonPressed(4);  // macro commande

echo "** Annulation: ";
$myRemoteControl->cancelButtonPressed();  // cancel button.

echo $myRemoteControl;

echo '--------------', $newline;
echo 'Controleur: Fin traitement.', $newline;
?>
```

On programme la télécommande.

On s'amuse à appuyer sur des boutons !

On accélere jusqu'à atteindre la limite.

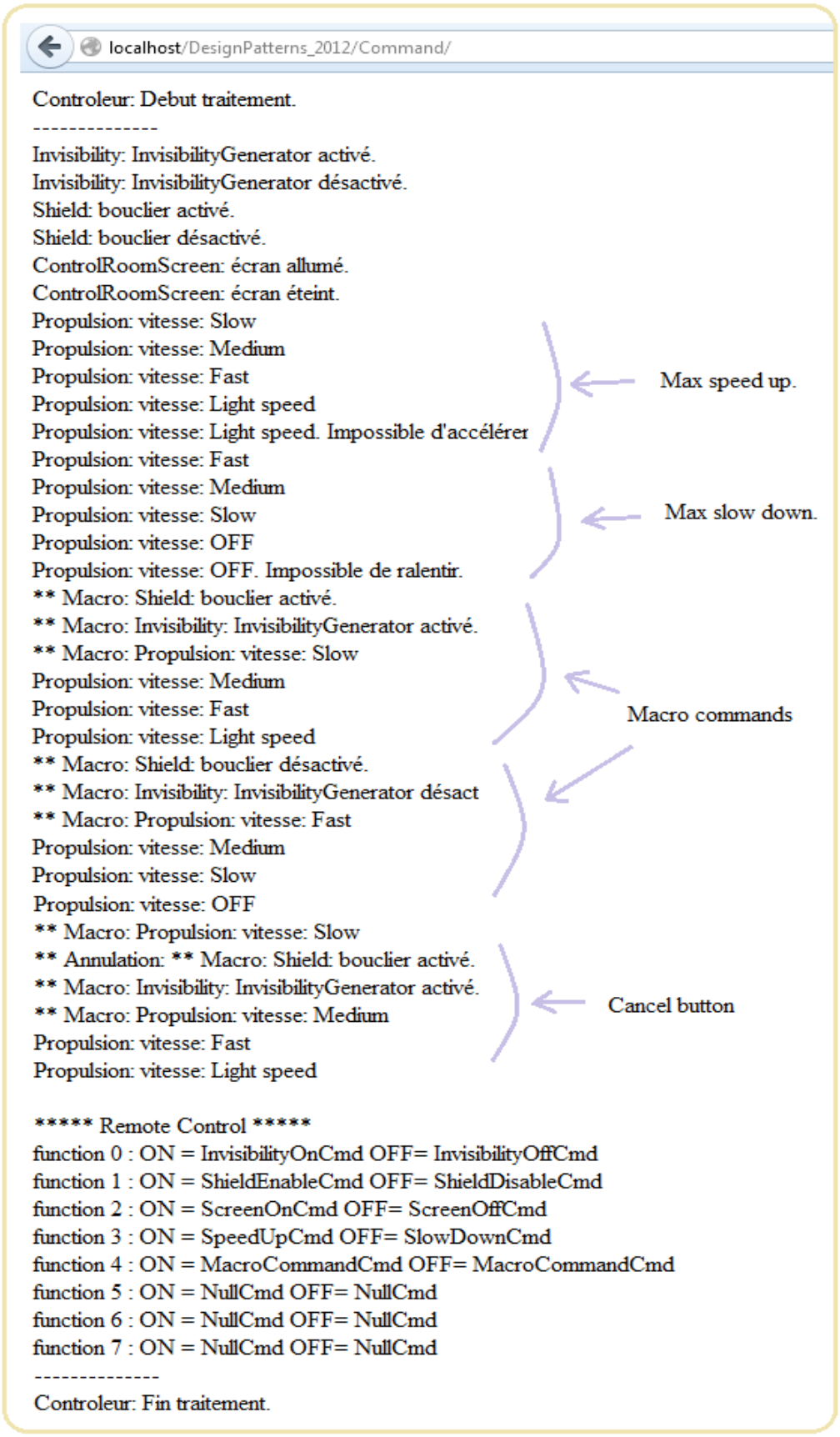
On décelère jusqu'à atteindre la limite.

Utilisons notre macro commande.

...et notre bouton d'annulation.

Pour terminer, affichons la configuration de la télé commande.

Le résultat de l'exécution :



10.4 Conclusion

Comme nous l'avons vu, le pattern Command sépare deux notions qui sont habituellement confondues:

- La demande d'une action à exécuter, c'est à dire l'invocation d'une méthode qui va effectuer un travail. Cette notion devient une Commande.
- L'exécution du travail, c'est à dire le fait d'exécuter le travail défini par la commande.

Les demandes d'actions étant encapsulées dans des objets normalisés (commandes), il est devient facile de les stocker pour une exécution ultérieure.

De ce fait le pattern Command est souvent utilisé pour traiter des files d'attentes de travaux: Un ou plusieurs processus sont chargés alors d'exécuter une file de commandes les unes après les autres. Le processus, qui n'a aucune idée de ce que fait réellement la commande, se contente d'invoquer la méthode execute() de la commande. Une fois le travail effectué, le processus passe à la commande suivante.

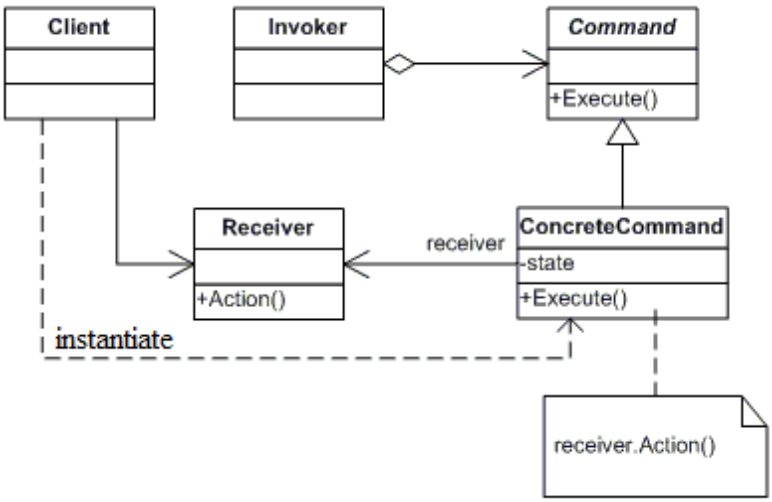
Le pattern peut également être utilisé pour historiser l'activité d'un système entre deux sauvegardes. Si un crash survient, après la restauration de la dernière sauvegarde, il sera possible de rejouer les commandes historisées pour retrouver l'état du système au moment du crash.

Les utilisations d'un même pattern peuvent être très différentes les unes des autres.

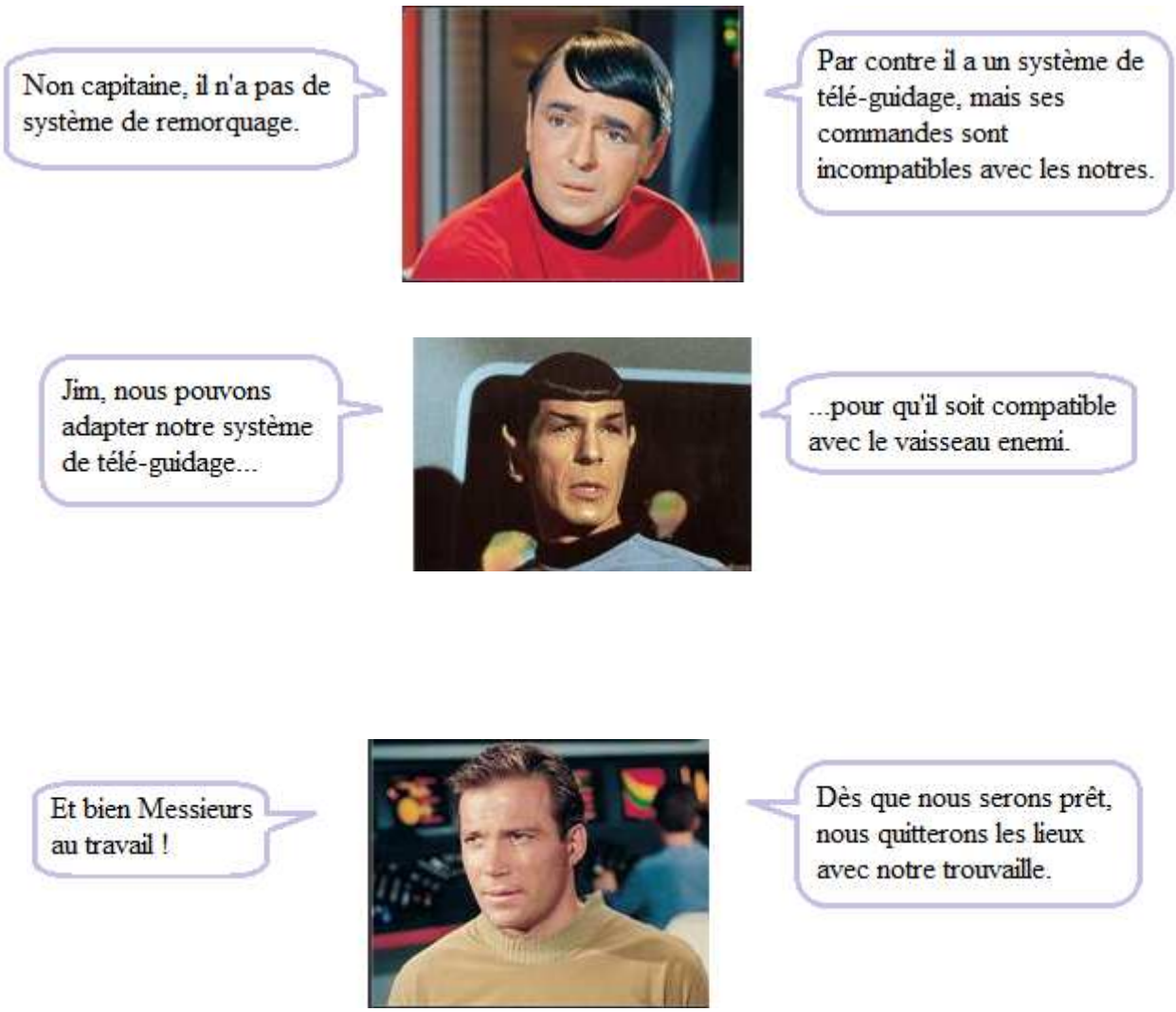
Ce pattern, une fois encore, est basé sur la composition et non pas l'implémentation. C'est à dire que les traitements sont encapsulés dans de petites classes spécialisées qui sont associées par le contrôleur, en utilisant la composition (c'est à dire les références entre objets) au moment de l'exécution du programme. Cela signifie, dans notre exemple, que le contrôleur pourrait reconfigurer la télécommande à n'importe quel moment.

Par opposition, un montage classique, basé sur l'implémentation, nous donnerait des classes plus grosses, moins nombreuses certes, mais contenant des traitements davantage figés dans le code. Toute modification nécessiterait souvent de retoucher le code source. Le contrôleur, pendant l'exécution du programme, n'aurait que peu de marge de manœuvre.

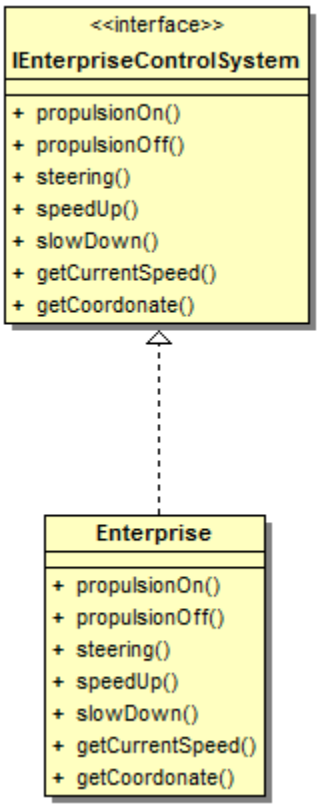
Voici le diagramme de classe officiel du pattern Command:


Client
Invoker
Command
+Execute()
Receiver
+Action()
receiver
ConcreteCommand
-state
+Execute()
instantiate
receiver.Action()

11 PATTERN ADAPTER



Le système de pilotage de l'Enterprise se présente comme suit:



L'interface IEnterpriseControlSystem impose un ensemble de méthodes nécessaires au pilotage de l'Enterprise.

La classe Enterprise implémente donc ces méthodes.

Utilisons un peu cette classe avant d'aller plus loin. Voici le code de l'interface:

```php
<?php

interface IEnterpriseControlSystem {

    public function propulsionOn();

    public function propulsionOff();

    public function steering($x, $y, $z);

    public function speedUp();

    public function slowDown();
}

?>
```

Et celui de la classe Enterprise:

```php
<?php

class Enterprise implements IEnterpriseControlSystem {

    private $_vesselName = '';
    private $_currentSpeed;
    private $_propulsionStatus;
    private $_speedLevels = array("OFF", "Slow", "Medium",
        "Fast", "Light speed");
    private $_maxSpeed;
    private $_x, $_y, $_z;  // coordonate

    public function __construct($vesselName) {
        $this->_vesselName = $vesselName;
        $this->_currentSpeed = 0;
        $this->_propulsionStatus = FALSE;
        $this->_maxSpeed = count($this->_speedLevels) - 1;
        $this->_x = 0;
        $this->_y = 0;
        $this->_z = 0;
    }

    public function propulsionOn() {
        $this->_propulsionStatus = TRUE;
        $this->log("propulsionOn()", "propulsion On");
    }

    public function propulsionOff() {
        $this->_currentSpeed = 0;
        $this->_propulsionStatus = FALSE;
        $this->log("propulsionOff()", "propulsion Off");
    }

    public function steering($x, $y, $z) {
        $this->_x = $x;
        $this->_y = $y;
        $this->_z = $z;
    }

    public function speedUp() {
        $message1 = "";
        $message2 = "Current speed: " . $this->getCurrentSpeed();
        if ($this->_propulsionStatus == TRUE) {
            if ($this->_currentSpeed >= $this->_maxSpeed) {
                $message1 = "Impossible d'accélérer.";
            } else {
                $this->_currentSpeed++;
                $message2 = "Current speed: " . $this->getCurrentSpeed();
            }
        } else {
            $message1 = "Impossible: propulsion off.";
        }
        $this->log("speedUp()", $message1 . " " . $message2);
    }
```

Quelques attributs privés.

Un système de coordonnées à 3 dimensions.

Le constructeur initialise les attributs privés.

Allumage et extinctions du système de propulsion.

Pour changer de cap, on affecte de nouvelles coordonnées.

Accélérer.

```php
    public function slowDown() {
        $message1 = "";
        $message2 = "Current speed: " . $this->getCurrentSpeed();
        if ($this->_propulsionStatus == TRUE) {
            if ($this->_currentSpeed <= 0) {
                $message1 = "Impossible de ralentir.";
            } else {
                $this->_currentSpeed--;
                $message2 = "Current speed: " . $this->getCurrentSpeed();
            }
        } else {
            $message1 = "Impossible: propulsion Off.";
        }
        $this->log("slowDown()", $message1 . " " . $message2);
    }

    private function log($method, $message) {
        echo "Class: " . get_class()
        . ". Method: " . $method
        . ". Vaisseau: " . $this->_vesselName . ": "
        . $message . '</br>';
    }

    public function getCurrentSpeed() {
        return $this->_speedLevels[$this->_currentSpeed];
    }

    public function getCoordonate() {
        return "Coordonate: x= " . $this->_x
                . "; y= " . $this->_y
                . "; z= " . $this->_z
                . "<br>";
    }

}

?>
```

Ralentir

Une méthode utilitaire pour afficher des messages.

Obtenir la vitesse ou les coordonnées actuelles.

Voici un exemple d'utilisation de la classe Enterprise:

```php
<?php

//**************************************************************
// ADAPTER pattern controler
//**************************************************************

require_once 'includePaths.php';
$newline = "</br>";

echo 'Controleur: Debut traitement.', $newline;
echo '--------------', $newline;

// Instanciations:
$myEnterprise = new Enterprise("enterprise");



// traitemment sur Enterprise:
echo $myEnterprise->getCoordonate();
$myEnterprise->steering(25,65,33);
echo $myEnterprise->getCoordonate();

$myEnterprise->propulsionOn();
$myEnterprise->propulsionOff();
$myEnterprise->speedUp();

$myEnterprise->propulsionOn();
$myEnterprise->speedUp();
$myEnterprise->speedUp();
$myEnterprise->speedUp();
$myEnterprise->speedUp();
$myEnterprise->speedUp();

$myEnterprise->slowDown();
$myEnterprise->slowDown();
$myEnterprise->slowDown();
$myEnterprise->slowDown();
$myEnterprise->slowDown();

echo '--------------', $newline;
echo 'Controleur: Fin traitement.', $newline;
?>
```

On fixe un cap.

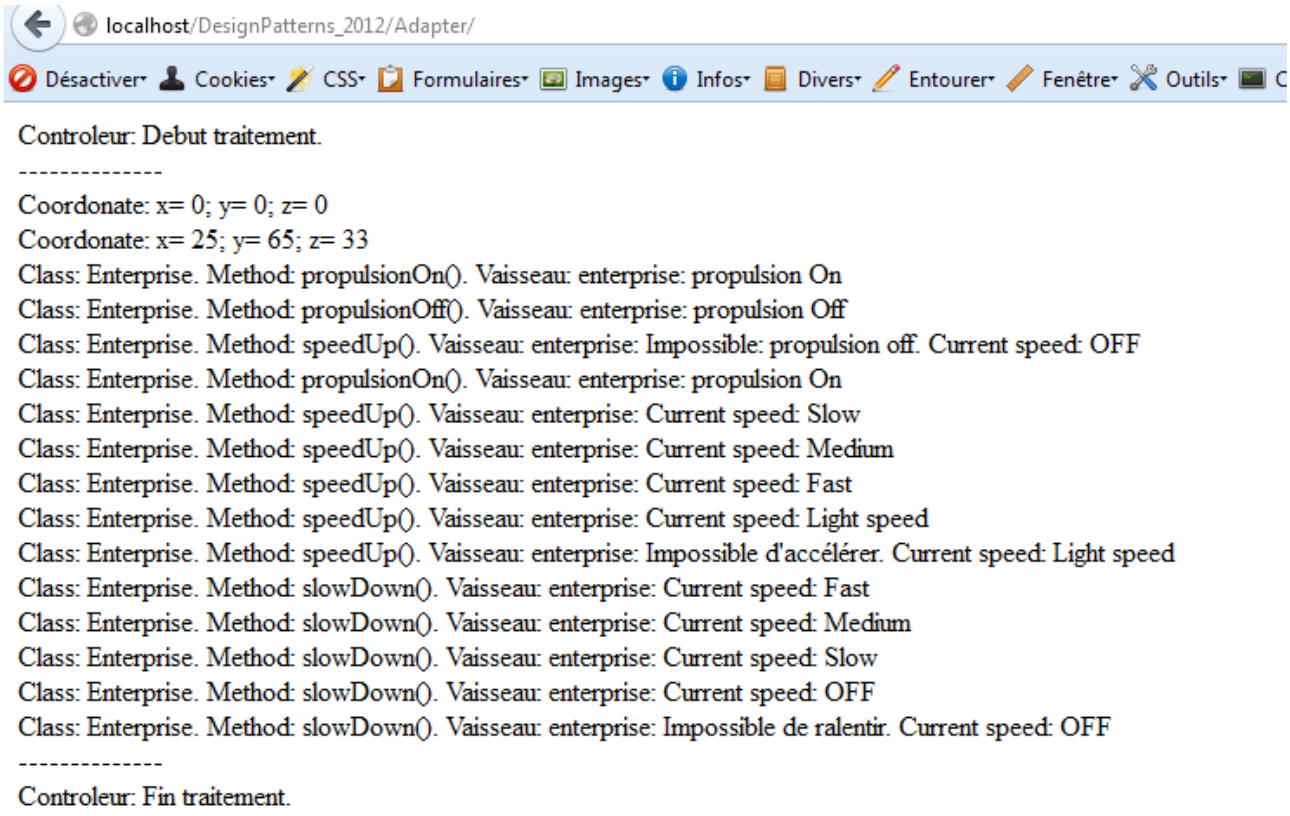
On teste l'allumage et l'arret du moteur.

On teste l'accélération quand le moteur est éteint.

Accélération au maximum.

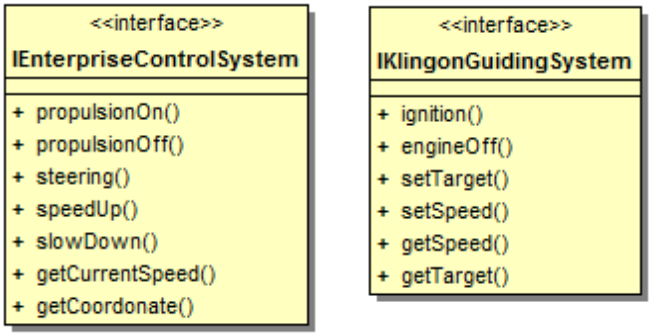
Décélération complète.

Et voici le résultat de l'exécution:



Effectivement Mr Sulu. Mais les choses intéressantes arrivent maintenant.

Le vaisseau Klingon possède son propre système de contrôle défini par l'interface IKlingonGuidingSystem.



Comme on le voit, elle n'a pas grand-chose en commun avec celle de l'Enterprise. Il va pourtant falloir rendre ces deux systèmes compatibles.

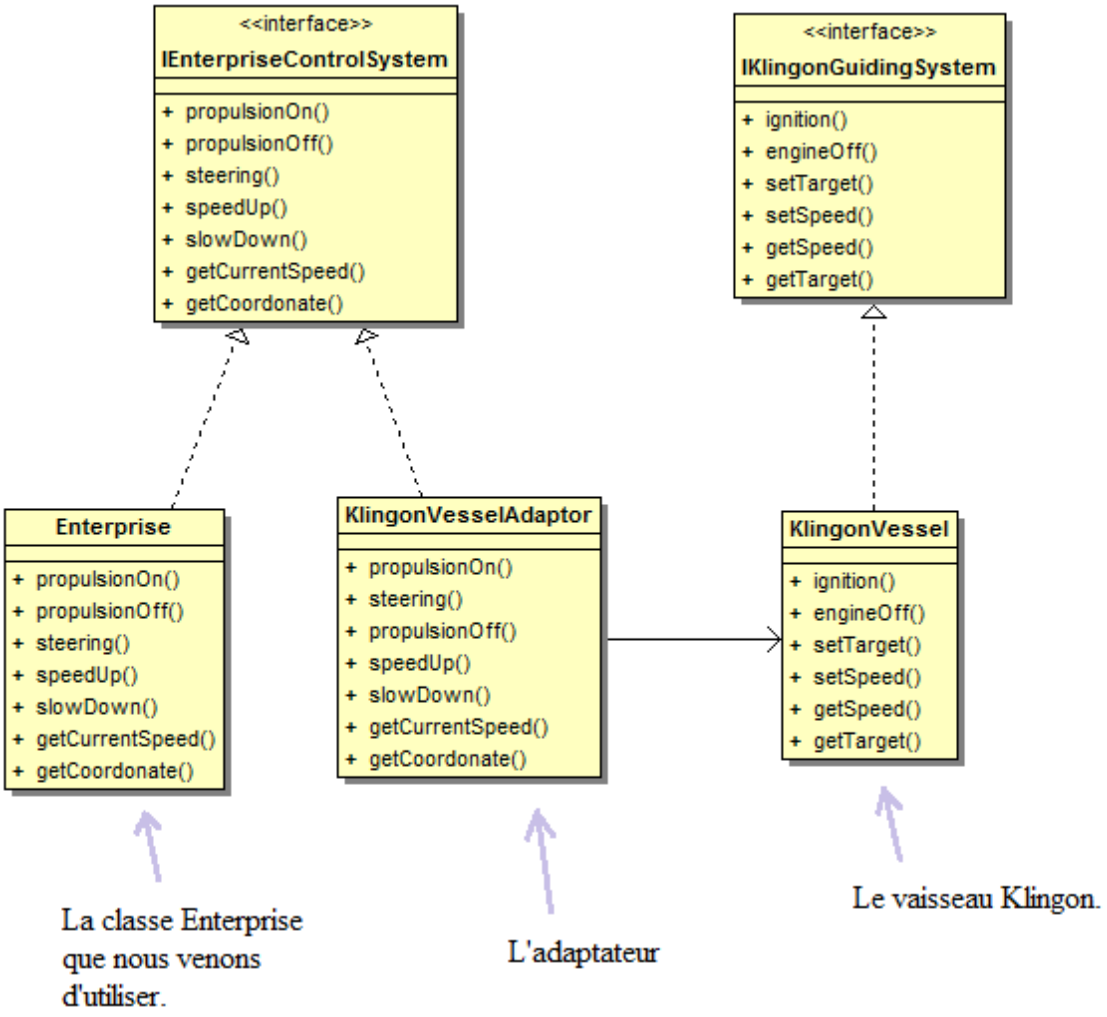
Le pattern Adapter est l'un des plus intuitifs. Il agit réellement comme un adaptateur, permettant à deux systèmes à priori incompatibles, de communiquer. Exactement comme un adaptateur permet à un appareil électrique de se brancher sur une prise électrique d'un pays étranger.

Notre adaptateur:

- Sera matérialisé par une classe dérivant de IEnterpriseControlSystem. Elle possédera donc toutes les méthodes de pilotage d'un vaisseau de type Enterprise.
- Mais aura une référence vers un vaisseau de type Klingon.
- Pour chaque méthode de type Enterprise, exécutera l'équivalent sur le vaisseau Klingon.

C'est cette classe qui possède l'intelligence permettant l'adaptation entre les deux systèmes.

Voici le modèle de classes correspondant:



Voyons comment fonctionne la classe KlingonVessel:

```php
class KlingonVessel implements IKlingonGuidingSystem {

    private $_currentSpeed = 0;
    private $_maxSpeed = 10;
    private $_alpha, $_beta, $_gama;  // coordonate
    private $_vesselName = '';

    public function __construct($vesselName) {
        $this->_vesselName = $vesselName;
        $this->_currentSpeed = 0;
        $this->_alpha = 0;
        $this->_beta = 0;
        $this->_gama = 0;
    }

    public function ignition() {
        $this->log("ignition()", "Ignition");
    }

    public function engineOff() {
        $this->log("engineOff()", "Engine off");
    }

    public function setTarget($alpha, $beta, $gama) {
        $this->_alpha = $alpha;
        $this->_beta = $beta;
        $this->_gama = $gama;
        $message = "alpha: " . $this->_alpha
                . " beta: " . $this->_beta
                . " gama: " . $this->_gama;
        $this->log("setTarget()", $message);
    }

    public function setSpeed($newSpeed) {
        if ($newSpeed < 0) {
            $newSpeed = 0;
        }
        if ($newSpeed > $this->_maxSpeed) {
            $newSpeed = $this->_maxSpeed;
        }
        $this->_currentSpeed = $newSpeed;
        $message = "Current speed: " . $this->_currentSpeed;
        $this->log("setSpeed()", $message);
    }
```

Des attributs privés.

Un constructeur qui initialise les attributs privés.

Allumage et extinction du moteur.

Définition d'un nouveau cap.

Définition de la nouvelle vitesse.

```
    public function getSpeed() {
        return $this->_currentSpeed;
    }

    public function getTarget() {
        return "Target: alpha= " . $this->_alpha
                . "; beta= " . $this->_beta
                . "; gama= " . $this->_gama
                . "<br>";
    }

    private function log($method, $message) {
        echo "Class: " . get_class()
        . ". Method: " . $method
        . ". Vaisseau: " . $this->_vesselName . ": "
        . $message . '</br>';
    }

}

?>
```

Obtenir la vitesse ou le cap actuel.

Méthode utilitaire pour loguer des infos.

Il faut maintenant déterminer la manière dont nous allons adapter chaque méthode de type Enterprise en une méthode de type Klingon.

- La méthode ignition() allume le moteur Klingon. Cette méthode correspond donc directement à la méthode propulsionOn() de l'Enterprise.
- De manière similaire, engineOff() correspond à propulsionOff().
- La méthode setTarget() correspond à steering(). Il s'agit de fournir 3 coordonnées spatiales définissant le nouveau cap. Mais le système référentiel Klingon n'a pas la même unité de mesure. Il faudra multiplier les coordonnées par un coefficient.
- La méthode setSpeed() défini la vitesse du vaisseau Klingon. Elle a un fonctionnement différent de l'Enterprise car on définit ici directement la vitesse à atteindre, alors que sur l'Enterprise, on accélère (speedUp()) ou on décélère (slowDown()). Ces deux méthodes devrons utiliser setSpeed() en augmentant ou en diminuant la vitesse Klingon d'une unité, par rapport à sa vitesse actuelle fournie par getSpeed().
- La méthode getSpeed() correspond à getCurrentSpeed().
- La méthode getTarget() correspond à getCoordonate().

Il n'y a plus qu'à implémenter la classe KlingonVesselAdapter:

```php
<?php

class KlingonVesselAdapter implements IEnterpriseControlSystem {

    private $_myVessel;

    public function __construct($vessel) {
        $this->_myVessel = $vessel;
    }

    public function propulsionOn() {
        $this->_myVessel->ignition();
    }

    public function propulsionOff() {
        $this->_myVessel->engineOff();
    }

    public function steering($x, $y, $z) {
        $coef = 2.734;
        $this->_myVessel->setTarget($x * $coef, $y * $coef, $z * $coef);
    }

    public function speedUp() {
        $this->_myVessel->setSpeed($this->_myVessel->getSpeed() + 1);
    }

    public function slowDown() {
        $this->_myVessel->setSpeed($this->_myVessel->getSpeed() - 1);
    }

    public function getCurrentSpeed() {
        return $this->_myVessel->getSpeed();
    }

    public function getCoordonate() {
        return $this->_myVessel->getTarget();
    }

}

?>
```

La reference vers le vaisseau Klingon.

Allumage et extinction du moteur: on invoque la bonne méthode du vaisseau Klingon.

On applique le coeficient pour le changement de referentiel.

Accélération/décélération.

Obtention de la vitesse et des coordonnées actuelles.

Comme on le voit, pour chaque traitement, notre adaptateur invoque la méthode adéquate sur le vaisseau Klingon, de manière directe pour les cas simples, ou en adaptant les données si nécessaire.

11.1 Tests

Comme d'habitude nous allons utiliser index.php comme contrôleur de l'ensemble du montage:

```php
<?php

//*************************************************************
// ADAPTER pattern controler
//*************************************************************

require_once 'includePaths.php';
$newline = "</br>";

echo 'Controleur: Debut traitement.', $newline;
echo '--------------', $newline;

// Instanciations:
$myKlingonVessel = new KlingonVessel("klingon");
$myKlingonVesselAdaptor = new KlingonVesselAdapter($myKlingonVessel);

// traitemment:
$myKlingonVesselAdaptor->propulsionOn();
$myKlingonVesselAdaptor->propulsionOff();
$myKlingonVesselAdaptor->propulsionOn();
$myKlingonVesselAdaptor->speedUp();
$myKlingonVesselAdaptor->speedUp();
$myKlingonVesselAdaptor->speedUp();
$myKlingonVesselAdaptor->speedUp();
$myKlingonVesselAdaptor->speedUp();
$myKlingonVesselAdaptor->speedUp();
$myKlingonVesselAdaptor->speedUp();
$myKlingonVesselAdaptor->speedUp();
$myKlingonVesselAdaptor->speedUp();
$myKlingonVesselAdaptor->speedUp();
$myKlingonVesselAdaptor->speedUp();

$myKlingonVesselAdaptor->slowDown();
$myKlingonVesselAdaptor->slowDown();
$myKlingonVesselAdaptor->slowDown();
$myKlingonVesselAdaptor->slowDown();
$myKlingonVesselAdaptor->slowDown();
$myKlingonVesselAdaptor->slowDown();
$myKlingonVesselAdaptor->slowDown();
$myKlingonVesselAdaptor->slowDown();
$myKlingonVesselAdaptor->slowDown();
$myKlingonVesselAdaptor->slowDown();

$myKlingonVesselAdaptor->steering(10, 200, 600);
$myKlingonVesselAdaptor->getCoordonate();

echo '--------------', $newline;
echo 'Controleur: Fin traitement.', $newline;
?>
```

On instancie les 2 classes necessaires.

tests allumages/extinctions.

Accélération jusqu'au maximum.

Décélération jusqu'au minimum.

Changement de cap.

Affichage des coordonnées.

Et voici le résultat de l'exécution:

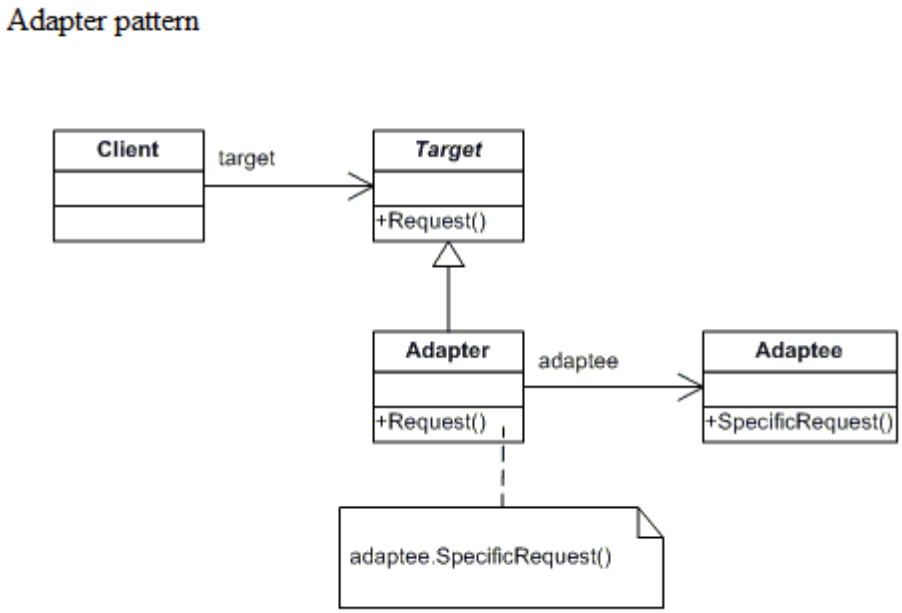
Controleur: Debut traitement.

Class: KlingonVessel. Method: ignition(). Vaisseau: klingon: Ignition
Class: KlingonVessel. Method: engineOff(). Vaisseau: klingon: Engine off
Class: KlingonVessel. Method: ignition(). Vaisseau: klingon: Ignition
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 1
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 2
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 3
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 4
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 5
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 6
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 7
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 8
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 9
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 10
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 10
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 9
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 8
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 7
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 6
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 5
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 4
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 3
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 2
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 1
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 0
Class: KlingonVessel. Method: setSpeed(). Vaisseau: klingon: Current speed: 0
Class: KlingonVessel. Method: setTarget(). Vaisseau: klingon: alpha: 27.34 beta: 546.8 gama: 1640.4

Controleur: Fin traitement.

Le travail est terminé. Notre adaptateur converti les commandes envoyées par le système de pilotage de l'Enterprise et le vaisseau Klingon répond correctement.

Le pattern Adapter officiel:



12 PATTERN FACADE

Tous les systèmes ont été mis en service capitaine.

C'est trop long Mr Spock.

Il y a plusieurs opérations à effectuer. Cela prend du temps Jim.

Voyez s'il est possible d'automatiser ces opérations. Je suis persuadé qu'on peut encore réduire la durée.

L'Enterprise possède différents systèmes qui doivent tous être activés individuellement pour que le vaisseau soit pleinement opérationnel:

Système	Description
Communication	Un réseau interne qui peut être activé ou désactivé. Un réseau externe qui peut être activé ou désactivé.
ElectricNetwork	Peut être mis dans l'un des modes: Off, Stand by, Maintenance, On.
Engine	Peut être mis dans l'un des modes: Off, Stand by, Maintenance, On.
Gravity	Peut être activé ou désactivé.
Teleportation	Peut être mis dans l'un des modes: Off, Stand by, Maintenance, On.
Weapon	Peut être mis dans l'un des modes: Off, Stand by, Maintenance, On.

Pour mettre l'Enterprise en service, il faut appliquer la procédure suivante:

- Activer les réseaux de communication interne et externe.
- Mettre le réseau électrique sur le mode ON.
- Activer la gravité artificielle.
- Mettre la téléportation sur le mode ON.
- Mettre l'armement sur le mode ON.

Il y a également une procédure pour mettre l'Enterprise en mode Maintenance:

- Activer le réseau de communication interne.
- Désactiver le réseau de communication externe.
- Mettre le réseau électrique sur le mode Maintenance.
- Activer la gravité artificielle.
- Mettre la téléportation sur le mode Off.
- Mettre l'armement sur le mode Maintenance.

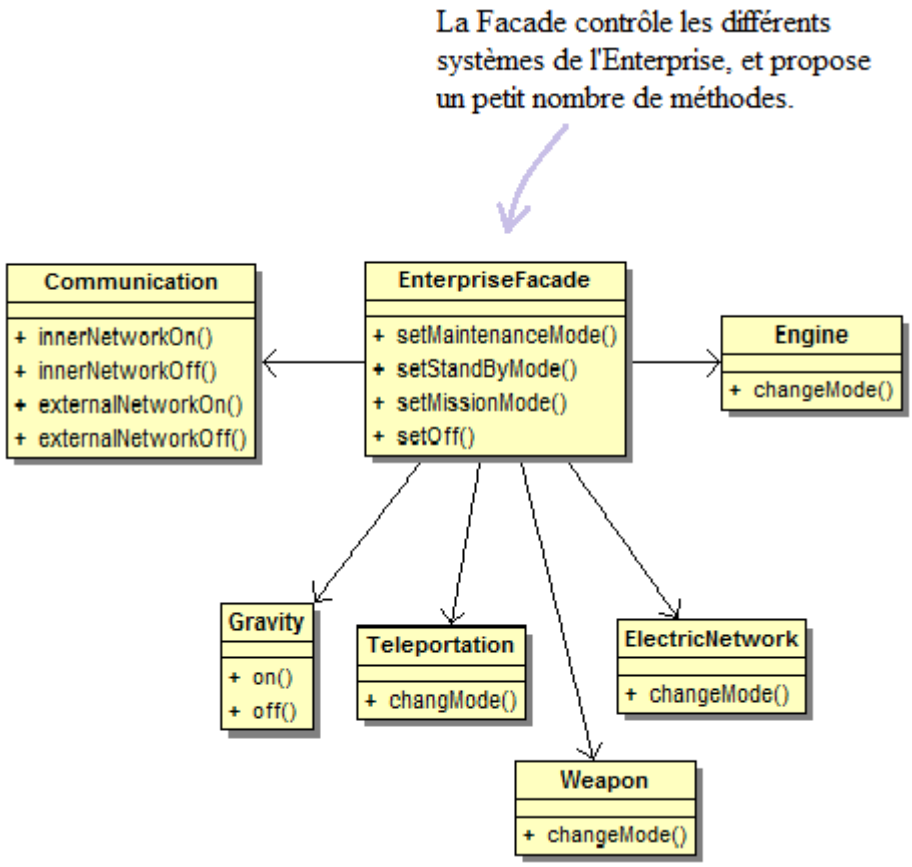
Une autre procédure pour mettre le vaisseau en mode Stanby, et une autre pour le mettre en mode Off.

Pour chaque procédure, il faut instancier les objets et invoquer les méthodes adéquates dans un ordre précis.

Il y a là une certaine complexité dont on aimerait bien se débarrasser. C'est là qu'intervient le design pattern Facade.

Le pattern Facade va prendre en charge toute cette complexité et automatiser les opérations pour nous, en nous proposant un nombre réduit de méthodes, permettant de contrôler simplement le système complexe qu'est l'Enterprise.

Concrètement le pattern Facade se présente comme une classe venant s'intercaler entre le système complexe, et l'utilisateur du système complexe:



Pour contrôler l'Enterprise il n'est plus nécessaire de connaitre la complexité des différents systèmes. Il suffit d'utiliser les méthodes proposées par la Facade. Ce sont elles qui vont faire tout le travail.

Puisque tout est clair, passons au codage.

12.1 Codage

La classe Communication:

```php
<?php

class Communication {

    private $_innerNetworkStatus;
    private $_externalNetworkStatus;

    public function __construct() {
        $this->_innerNetworkStatus = FALSE;
        $this->_externalNetworkStatus = FALSE;
    }

    public function innerNetworkOn() {
        $this->_innerNetworkStatus = TRUE;
        $this->log("innerNetworkOn()", "innerNetwork: ON");
    }

    public function innerNetworkOff() {
        $this->_innerNetworkStatus = FALSE;
        $this->log("innerNetworkOff()", "innerNetwork: OFF");
    }

    public function externalNetworkOn() {
        $this->_externalNetworkStatus = TRUE;
        $this->log("externalNetworkOn()", "externalNetwork: ON");
    }

    public function externalNetworkOff() {
        $this->_externalNetworkStatus = FALSE;
        $this->log("externalNetworkOff()", "externalNetwork: OFF");
    }

    public function getExternalNetworkStatus() {
        if( $this->_externalNetworkStatus == TRUE ) {
            return "1";
        } else {
            return "0";
        }
    }

    public function getInnerNetworkStatus() {
        if( $this->_innerNetworkStatus == TRUE ) {
            return "1";
        } else {
            return "0";
        }
    }

    private function log($method, $message) {
        echo "Class: " . get_class()
        . ". Method: " . $method . " "
        . $message . '</br>';
    }

}
?>
```

L'état des réseaux internes et externes.

Gestion du réseau interne.

Gestion du réseau externe.

Une méthode utilitaire pour afficher des infos.

La classe Gravity:

```php
<?php

class Gravity {

    private $_isGravityEnabled;

    public function __construct() {
        $this->_isGravityEnabled = TRUE;
    }

    public function on() {
        $this->_isGravityEnabled = TRUE;
        $this->log("on()", "Gravity ON");
    }

    public function off() {
        $this->_isGravityEnabled = FALSE;
        $this->log("off()", "Gravity OFF");
    }

    public function getGravity() {
        return $this->_isGravityEnabled;
    }

    private function log($method, $message) {
        echo "Class: " . get_class()
        . ". Method: " . $method . " "
        . $message . '</br>';
    }

}

?>
```

La Classe Teleportation:

```php
<?php

class Teleportation {

    private $_modeList = array("OFF" => "OFF", "STANDBY" => "Standby",
        "MAINT" => "Maintenance", "ON" => "ON");
    private $_currentMode;

    public function __construct() {
        $this->_currentMode = "OFF";
    }

    public function changeMode($newMode) {
        $this->_currentMode = $newMode;
        $this->log("changeMode()", "CurrentMode: "
                . $this->_modeList[$this->getCurrentMode()]);
    }

    public function getCurrentMode() {
        return $this->_currentMode;
    }

    private function log($method, $message) {
        echo "Class: " . get_class()
        . ". Method: " . $method . " "
        . $message . '</br>';
    }

}
?>
```

Les différents modes possibles.

La classe Weapon:

```php
<?php

 class Weapon {

    private $_modeList = array("OFF" => "OFF", "STANDBY" => "Standby",
        "MAINT" => "Maintenance", "ON" => "ON");
    private $_currentMode;

    public function __construct() {
        $this->_currentMode = "OFF";
    }

    public function changeMode($newMode) {
        $this->_currentMode = $newMode;
        $this->log("changeMode()", "CurrentMode: "
                . $this->_modeList[$this->getCurrentMode()]);
    }

    public function getCurrentMode() {
        return $this->_currentMode;
    }

    private function log($method, $message) {
        echo "Class: " . get_class()
        . ". Method: " . $method . " "
        . $message . '</br>';
    }

}
?>
```

Les différents modes possibles.

La classe ElectricNetwork:

```php
<?php

class ElectricNetwork {

    private $_modeList = array("OFF" => "OFF", "STANDBY" => "Standby",
        "MAINT" => "Maintenance", "ON" => "ON");
    private $_currentMode;

    public function __construct() {
        $this->_currentMode = "OFF";
    }

    public function changeMode($newMode) {
        $this->_currentMode = $newMode;
        $this->log("changeMode()", "CurrentMode: "
                . $this->_modeList[$this->getCurrentMode()]);
    }

    public function getCurrentMode() {
        return $this->_currentMode;
    }

    private function log($method, $message) {
        echo "Class: " . get_class()
        . ". Method: " . $method . " "
        . $message . '</br>';
    }

}
?>
```

Les différents modes possibles.

La classe Engine:

```php
<?php

 class Engine {

    private $_modeList = array("OFF" => "OFF", "STANDBY" => "Standby",
        "MAINT" => "Maintenance", "ON" => "ON");
    private $_currentMode;

    public function __construct() {
        $this->_currentMode = "OFF";
    }

    public function changeMode($newMode) {
        $this->_currentMode = $newMode;
        $this->log("changeMode()", "CurrentMode: "
                . $this->_modeList[$this->getCurrentMode()]);
    }

    public function getCurrentMode() {
        return $this->_currentMode;
    }

    private function log($method, $message) {
        echo "Class: " . get_class()
        . ". Method: " . $method . " "
        . $message . '</br>';
    }

}
?>
```

Les différents modes possibles.

La classe EnterpriseFacade:

```php
<?php

class EnterpriseFacade {

    private $_teleportation;
    private $_engine;
    private $_weapon;
    private $_electricNetwork;
    private $_gravity;
    private $_communication;

    public function __construct($teleportation, $engine, $weapon,
            $electricNetwork, $gravity, $communication) {
        $this->_teleportation = $teleportation;
        $this->_engine = $engine;
        $this->_weapon = $weapon;
        $this->_electricNetwork = $electricNetwork;
        $this->_gravity = $gravity;
        $this->_communication = $communication;
    }

    public function setMaintenanceMode() {
        $this->_communication->innerNetworkOn();
        $this->_communication->externalNetworkOff();
        $this->_electricNetwork->changeMode("MAINT");
        $this->_gravity->on();
        $this->_teleportation->changeMode("OFF");
        $this->_weapon->changeMode("MAINT");
    }

    public function setStandByMode() {
        $this->_communication->innerNetworkOff();
        $this->_communication->externalNetworkOff();
        $this->_electricNetwork->changeMode("STANDBY");
        $this->_gravity->off();
        $this->_teleportation->changeMode("OFF");
        $this->_weapon->changeMode("STANDBY");
    }

    public function setMissionMode() {
        $this->_communication->innerNetworkOn();
        $this->_communication->externalNetworkOn();
        $this->_electricNetwork->changeMode("ON");
        $this->_gravity->on();
        $this->_teleportation->changeMode("ON");
        $this->_weapon->changeMode("ON");
    }
```

Les références vers les différents systèmes de l'Enterprise.

C'est ici qu'on gère toute la compléxité du système.

```php
    public function setOff() {
        $this->_communication->innerNetworkOff();
        $this->_communication->externalNetworkOff();
        $this->_electricNetwork->changeMode("OFF");
        $this->_gravity->off();
        $this->_teleportation->changeMode("OFF");
        $this->_weapon->changeMode("OFF");
    }

}
?>
```

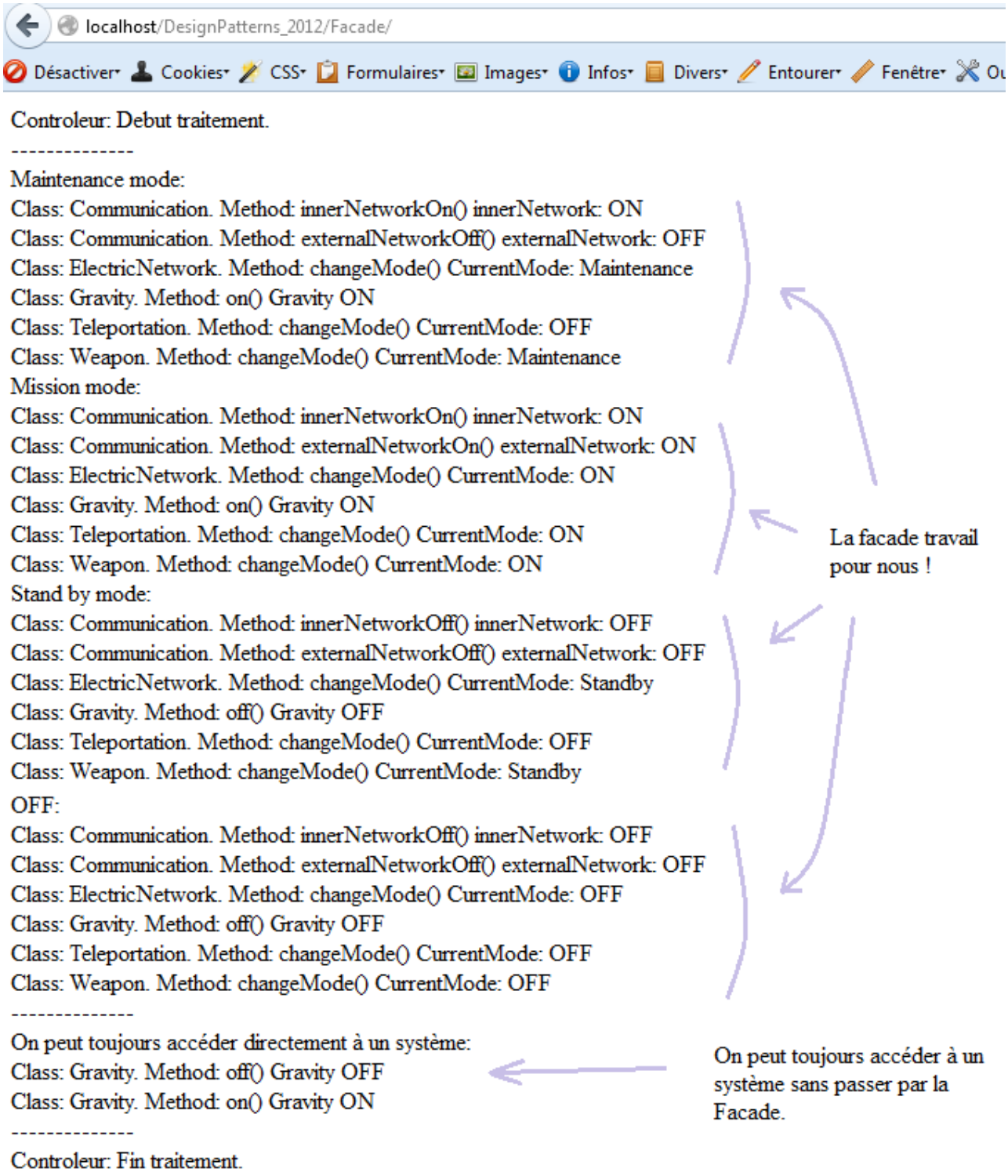
12.2 Tests

Comme d'habitude nous allons utiliser index.php comme contrôleur de l'ensemble du montage:

```php
<?php

//*************************************************************
// FACADE pattern controler
//*************************************************************

require_once 'includePaths.php';
$newline = "<br>";

echo 'Controleur: Debut traitement.', $newline;
echo '--------------', $newline;

// Instanciations:
$myTeleportation = new Teleportation();
$myGravity = new Gravity();
$myCommunication = new Communication();
$myElectricNetwork = new ElectricNetwork();
$myEngine = new Engine();
$myWeapon = new Weapon();
$myFacade = new EnterpriseFacade($myTeleportation,$myEngine,$myWeapon,
        $myElectricNetwork,$myGravity,$myCommunication);

echo "Maintenance mode:" . $newline;
$myFacade->setMaintenanceMode();

echo "Mission mode:" . $newline;
$myFacade->setMissionMode();

echo "Stand by mode:" . $newline;
$myFacade->setStandByMode();

echo "OFF:" . $newline;
$myFacade->setOff();

echo '--------------', $newline;
echo "On peut toujours accéder directement à un système:" . "<br>";
$myGravity->off();
$myGravity->on();

echo '--------------', $newline;
echo 'Controleur: Fin traitement.', $newline;
?>
```

On utilise les méthodes de la Facade.

Voici le résultat de l'exécution:



Excellent Mr Spock. Le vaisseau n'a jamais été opérationnel aussi vite.

Comme on a un peu de temps je vais essayer mon nouveau fusil laser portatif.

13 PATTERN TEMPLATE METHOD

Pour illustrer ce pattern, nous allons nous intéresser à la nourriture servie à bord de l'Enterprise.

Une nourriture moderne et équilibrée, qui fait rêver.

Toutes sortes de plats sont préparés à la demande, mais ils sont tous élaborés à partir d'un seul ingrédient: des granulés contenant tous ce qui est nécessaire à l'organisme.

On y trouve des lipides, des glucides, des protéines, des sels minéraux, des vitamines. Ils sont sans saveur particulière, ce qui permet de les transformer à volonté en leur donnant la texture, la couleur et la saveur voulue.

3 textures sont possibles:

- Les granulés sont laissés en l'état (pâtes, petits morceaux de légumes ou de viande etc...)
- Les granulés sont mixé avec le l'eau: (soupes, purées, crèmes...)
- Les granulés sont pétris avec de l'eau, moulés, partiellement asséchés, démoulés. On obtient des blocs de différentes formes: pâtés, pâtisseries, charcuterie...

Préparer un plat revient donc à suivre les étapes suivantes:

- Prendre une quantité de granulés.
- Transformer en l'une des 3 textures.
- Ajouter le colorant souhaité.
- Ajouter la saveur souhaitée.
- Monter en température.
- Ajouter optionnellement une sauce.
- Ajouter optionnellement un complément en vitamines.

Toutes sortes de plats sont réalisables. Il suffit de choisir la texture, la saveur et le couleur.

Mais examinons plus en détail les recettes pour chacune des 3 textures:

Textures granulaires:
- prendre du granulat
- appliquer la couleur
- appliquer la saveur
- mélanger
- monter en température
- appliquer l'option sauce
- appliquer l'option complement vitamines

Textures fluides:
- prendre du granulat
- ajouter de l'eau chaude
- appliquer la couleur
- appliquer la saveur
- mixer
- monter en température
- appliquer l'option sauce
- appliquer l'option complement vitamines

Textures solides:
- prendre du granulat
- ajouter de l'eau chaude
- appliquer la couleur
- appliquer la saveur
- pétrir
- mettre en moule
- sécher
- démouler
- monter en température
- appliquer l'option sauce
- appliquer l'option complement vitamines

Comme on le voit les 3 recettes sont très ressemblantes: certaines étapes sont communes, d'autres sont spécifiques.

On serait bien tenté de factoriser ce qui peut l'être, mais comment factoriser certaines parties d'un algorithme ?

Pour ce type de problème, la meilleure solution est le design pattern Template Method.

Ce pattern va nous permettre de définir un algorithme général imposé, tout en laissant la porte ouverte à une certaine variabilité pour les étapes qui le nécessitent.

Commençons par mettre les 3 recettes en parallèle pour identifier les opérations communes:

	Textures granulaires:		Textures fluides:		Textures solides:
1	- prendre du granulat	=	- prendre du granulat	=	- prendre du granulat
2			- ajouter de l'eau chaude	=	- ajouter de l'eau chaude
3	- appliquer la couleur	=	- appliquer la couleur	=	- appliquer la couleur
4	- appliquer la saveur	=	- appliquer la saveur	=	- appliquer la saveur
5	- mélanger		- mixer		- pétrir
6					- mettre en moule
7					- sécher
8					- démouler
9	- monter en température	=	- monter en température	=	- monter en température
10	- appliquer l'option sauce	=	- appliquer l'option sauce	=	- appliquer l'option sauce
11	- appliquer l'option complement vitamines	=	- appliquer l'option complement vitamines	=	- appliquer l'option complement vitamines

On peut séparer les opérations en 3 groupes:

- Groupe A: opérations étant identiques dans les 3 recettes (opérations 1,3,4,9,10,11). Ces opérations seront définies et implémentées dans notre Template Method. Elles seront par conséquent obligatoires et non modifiables.
- Groupe B: opérations identiques mais qui ne concernent pas les 3 recettes. L'opération est donc optionnelle (opérations 2,6,7,8). Ces opérations seront proposées par la Template Method, sous forme de crochets (hook): la classe utilisatrice aura le choix de l'exécuter ou pas et pourra même définir son implémentation.
- Groupe C: opérations qui représentent conceptuellement la même étape, mais dont la manière de faire est différente dans les 3 recettes (opération 5). Ces opérations seront obligatoires mais il reviendra aux classes utilisatrices de les implémenter.

Définissons maintenant ce qui va devenir notre Template Method: la recette unique pouvant s'adapter aux 3 recettes.

	Template Method	Type de méthode	Commentaire	Méthode
1	- prendre du granulat	statique	imposée	getGrainFood()
2	- ajouter de l'eau chaude	- protégée - non implémentée	hook	addHotWater()
3	- appliquer la couleur	statique	imposée	applyColor()
4	- appliquer la saveur	statique	imposée	applyFlavour()
5	- mélanger	abstraite	déléguée	blend()
6	- mettre en moule	- protégée - non implémentée	hook	mould()
7	- assécher	- protégée - non implémentée	hook	dryUp()
8	- démouler	- protégée - non implémentée	hook	outMould()
9	- monter en température	statique	imposée	applyTemperature()
10	- appliquer l'option sauce	statique	imposée	addSauce()
11	- appliquer l'option complement vitamines	statique	imposée	addVitamins()

Les méthodes imposées (en bleue) seront définies comme statiques dans la classe mère c'est à dire non héritées par les sous classes. Ces dernières ne pourront donc pas les redéfinir.

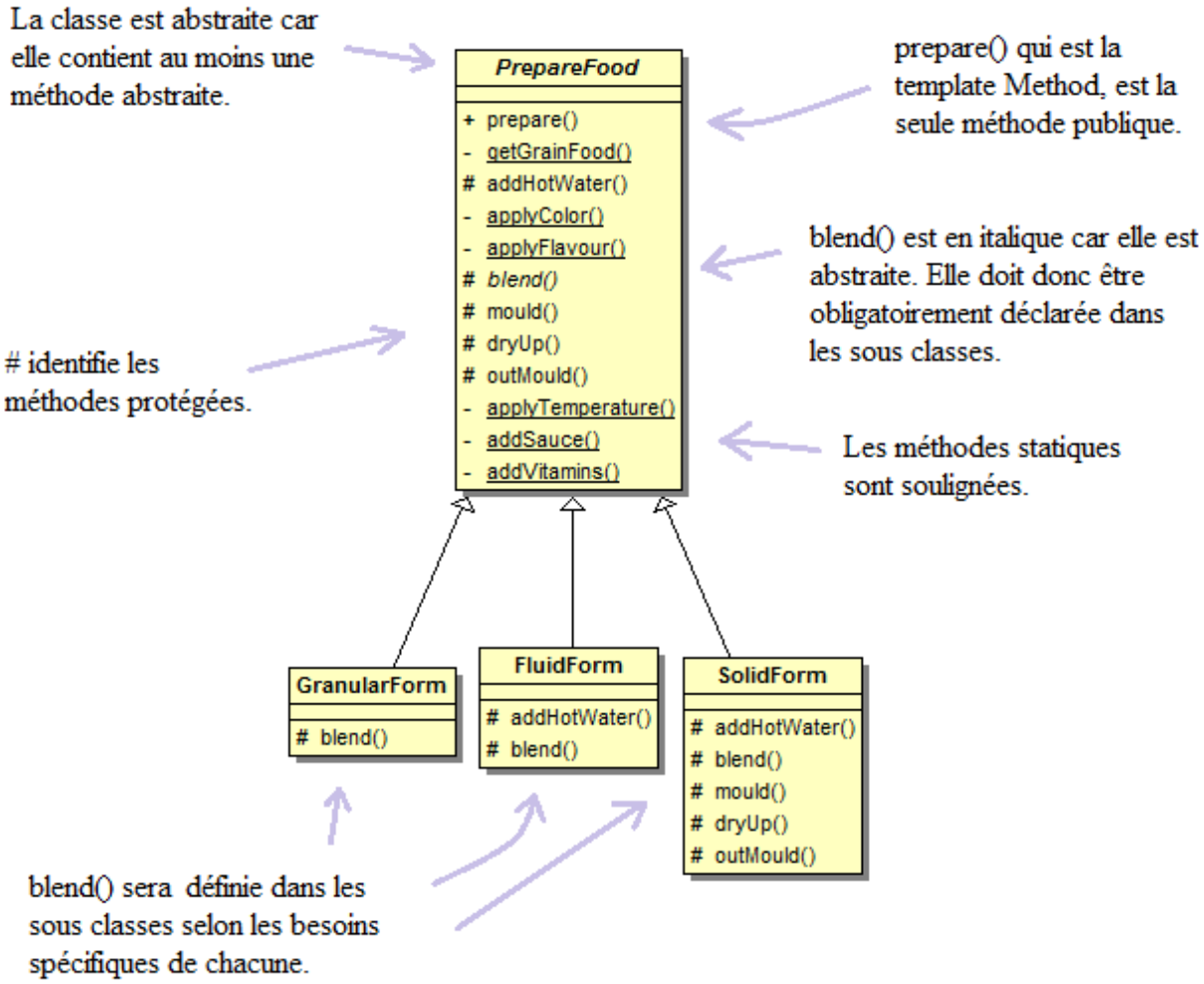
Les méthodes de type "crochet" (hook), seront définies comme protégées pour n'être accessible que par les sous classes, et non implémentées car ce sont précisément les sous classes qui sont chargées de les implémenter selon leur besoins spécifiques.

La méthode dite déléguée, sera définie comme abstraite. Elle correspond à une étape obligatoire dans la recette, mais doit être implémentée par les sous classes car la manière de faire varie d'une recette à l'autre.

13.1 Le modèle de classes

Le pattern Template Method se présente sous la forme d'une classe abstraite contenant la Template Method ainsi qu'un certain nombre d'autres méthodes nécessaires.

Ceci fait, il est possible de créer des classes dérivant de la classe abstraite. Dans notre cas nous allons créer une sous classe par recette:



13.2 Codage

La classe PrepareFood:

```php
<?php

abstract class PrepareFood {

    final public function prepare() {
        $this->getGrainFood();          // mandatory
        $this->addHotWater();           // hook
        $this->applyColor();            // mandatory
        $this->applyFlavour();          // mandatory
        $this->blend();                 // delegate
        $this->mould();                 // hook
        $this->dryUp();                 // hook
        $this->outMould();              // hook
        $this->applyTemperature();      // mandatory
        $this->addSauce();              // mandatory
        $this->addVitamins();           // mandatory
    }

    private static function getGrainFood() {
        // Mandatory: implemented in abstract class as static.
        echo "Je prend une certaine quantité de granulés." . "<br>";
    }

    private static function applyFlavour() {
        // Mandatory: implemented in abstract class as static.
        echo "J'applique la saveur demandée." . "<br>";
    }

    private static function applyColor() {
        // Mandatory: implemented in abstract class as static.
        echo "J'applique la couleur demandée." . "<br>";
    }

    private static function applyTemperature() {
        // Mandatory: implemented in abstract class as static.
        echo "Je met le plat à la température demandée." . "<br>";
    }

    private static function addSauce() {
        // Mandatory: implemented in abstract class as static.
        echo "J'ajoute la sauce demandée." . "<br>";
    }
```

Le mot clef "final" interdit aux sous classes de redéfinir la méthode.

La Template Method définit toutes les séquences de l'algorithme.

```
    private static function addVitamins() {
        // Mandatory: implemented in abstract class as static.
        echo "J'ajoute les vitamines demandées." . "<br>";
    }

    protected function addHotWater() {
        // Hook: stay empty.
        // Overloaded by subclasses if necessary.
    }

    protected function mould() {
        // Hook: stay empty.
        // Implemented by subclasses IF NECESSARY.
    }

    protected function dryUp() {
        // Hook: stay empty.
        // Implemented by subclasses IF NECESSARY.
    }

    protected function outMould() {
        // Hook: stay empty.
        // Implemented by subclasses IF NECESSARY.
    }

    // Delegated: stay mandatory but implemented by subclasses:
    protected abstract function blend();

}
?>
```

La classe GranularForm ne doit implémenter que la méthode blend():

```
<?php

class GranularForm extends PrepareFood {

    protected function blend() {
        echo "Je mélange doucement les granulés." . "<br>";
    }

}

?>
```

La classe FluidForm ne doit implémenter que les méthodes blend() et addHotWater():

```php
<?php

class FluidForm extends PrepareFood {

    protected function addHotWater() {
        echo "J'ajoute de l'eau chaude." . "<br>";
    }

    protected function blend() {
        echo "Je mixe les granulés pour obtenir une substance fluide." . "<br>";
    }

}

?>
```

La classe SolidForm doit implémenter plusieurs méthodes:

```php
<?php

class SolidForm extends PrepareFood {

    protected function addHotWater() {
        echo "J'ajoute de l'eau chaude." . "<br>";
    }

    protected function blend() {
        echo "Je pétris pour obtenir une pâte homogène." . "<br>";
    }

    protected function mould() {
        echo "Je met la pâte dans un moule." . "<br>";
    }

    protected function dryUp() {
        echo "J'extrais l'humidité." . "<br>";
    }

    protected function outMould() {
        echo "Je démoule." . "<br>";
    }

}

?>
```

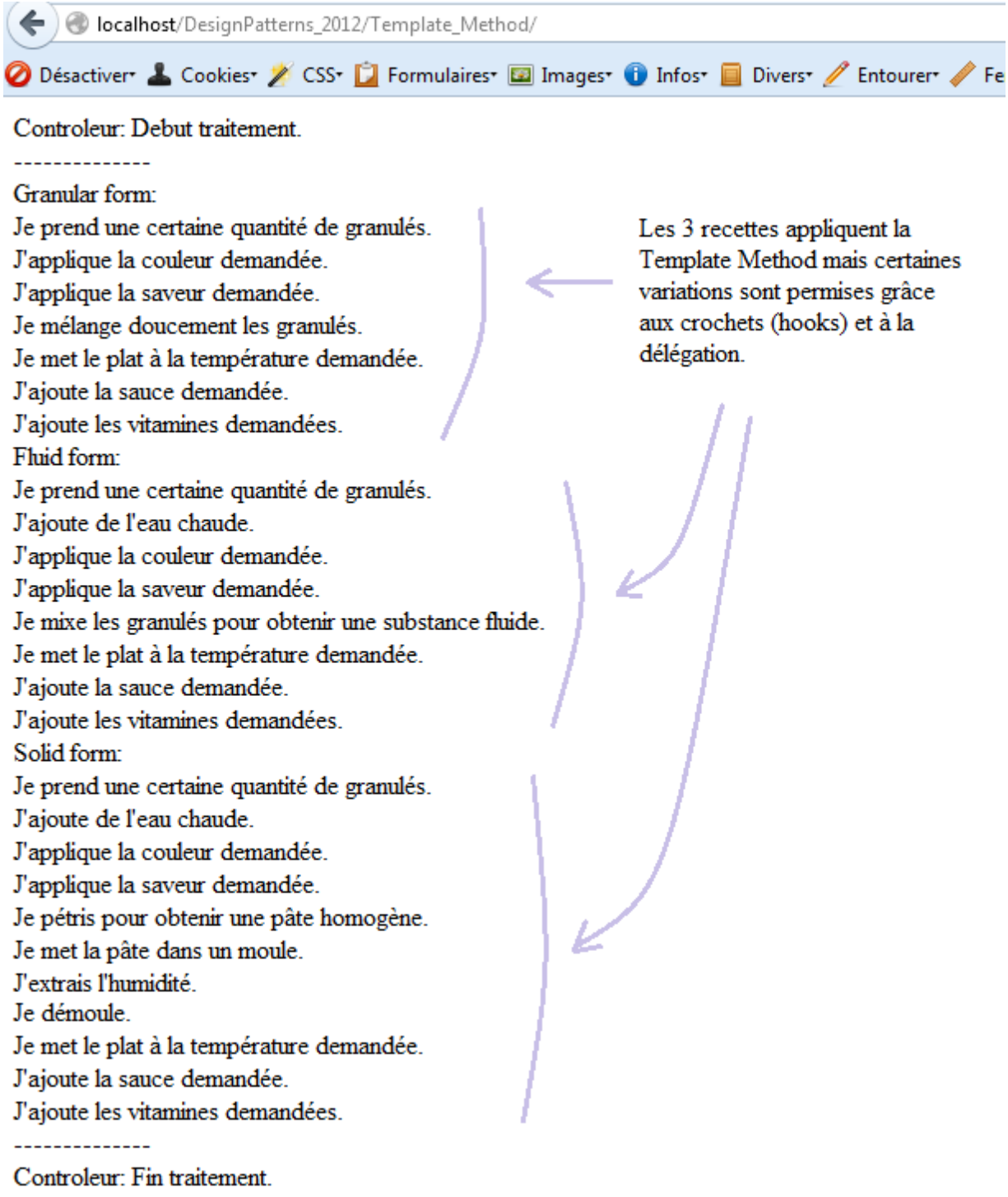
13.3 TESTS

Comme d'habitude nous allons utiliser index.php comme contrôleur de l'ensemble du montage.

Le code est extrêmement simple:

```php
<?php

//*************************************************************
// Template_Method pattern controler
//*************************************************************

require_once 'includePaths.php';
$newline = "</br>";

echo 'Controleur: Debut traitement.', $newline;
echo '--------------', $newline;

// Instanciations:
$myGranularForm = new GranularForm();
$myFluidForm = new FluidForm();
$mySolidform = new SolidForm();


// traitemment:
echo "Granular form:" . "<br>";
$myGranularForm->prepare();
echo "Fluid form:" . "<br>";
$myFluidForm->prepare();
echo "Solid form:" . "<br>";
$mySolidform->prepare();

echo '--------------', $newline;
echo 'Controleur: Fin traitement.', $newline;
?>
```

Voici le résultat de l'exécution:




Le goût de ces granulés n'est rien à coté d'un bon verre de scotch !

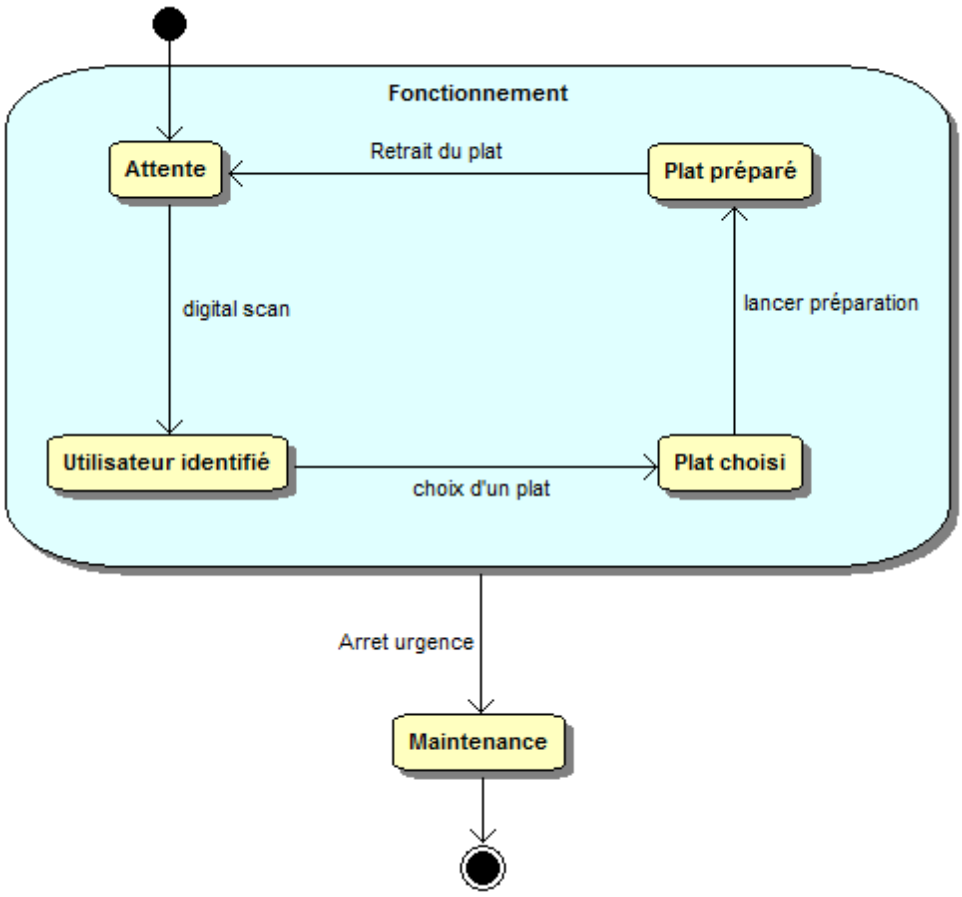
14 PATTERN STATE

Continuons à nous intéresser à la nutrition à bord de l'Enterprise. Avec le pattern Template Method, nous avons vu qu'un ingrédient unique permettait de préparer n'importe quel plat.

Voyons maintenant comment fonctionne les machines qui préparent les plats à la demande. Le fonctionnement de ces distributeurs automatiques est très simple:

- La personne appose la main sur un capteur optique pour être identifiée.
- La personne indique verbalement le plat désiré.
- La machine prépare le plat.
- La machine indique que le plat est prêt.
- La personne retire le plat.
- A tout moment on peut appuyer sur le bouton d'arrêt d'urgence qui met le distributeur en attente de maintenance.

Ce fonctionnement peut être représenté selon le diagramme d’état suivant:



Le distributeur peut avoir cinq états:

- **Attente**: la machine ne fait rien et attend qu'une personne se présente.
- **Utilisateur identifié**: une personne s'est identifiée sur le capteur digital.
- **Plat choisi:** la personne a choisi un plat.
- **Plat préparé**: la personne peut retirer le plat.
- **Maintenance**: on a appuyé sur le bouton d'arrêt d'urgence.

On comprend que pour répondre favorablement à une action, la machine doit se trouver dans l'état correspondant à cette action.

Si, par exemple la machine est dans l'état "plat choisi", elle ne peut traiter une action "digital scan". La seule action possible dans ce cas est "lancer préparation".

On devine tout de suite la complexité de notre future classe représentant le distributeur automatique: pour chaque action il faudrait tester l'état actuel du distributeur pour savoir comment traiter cette action. Nous n'allons évidemment pas nous engager dans cette voie.

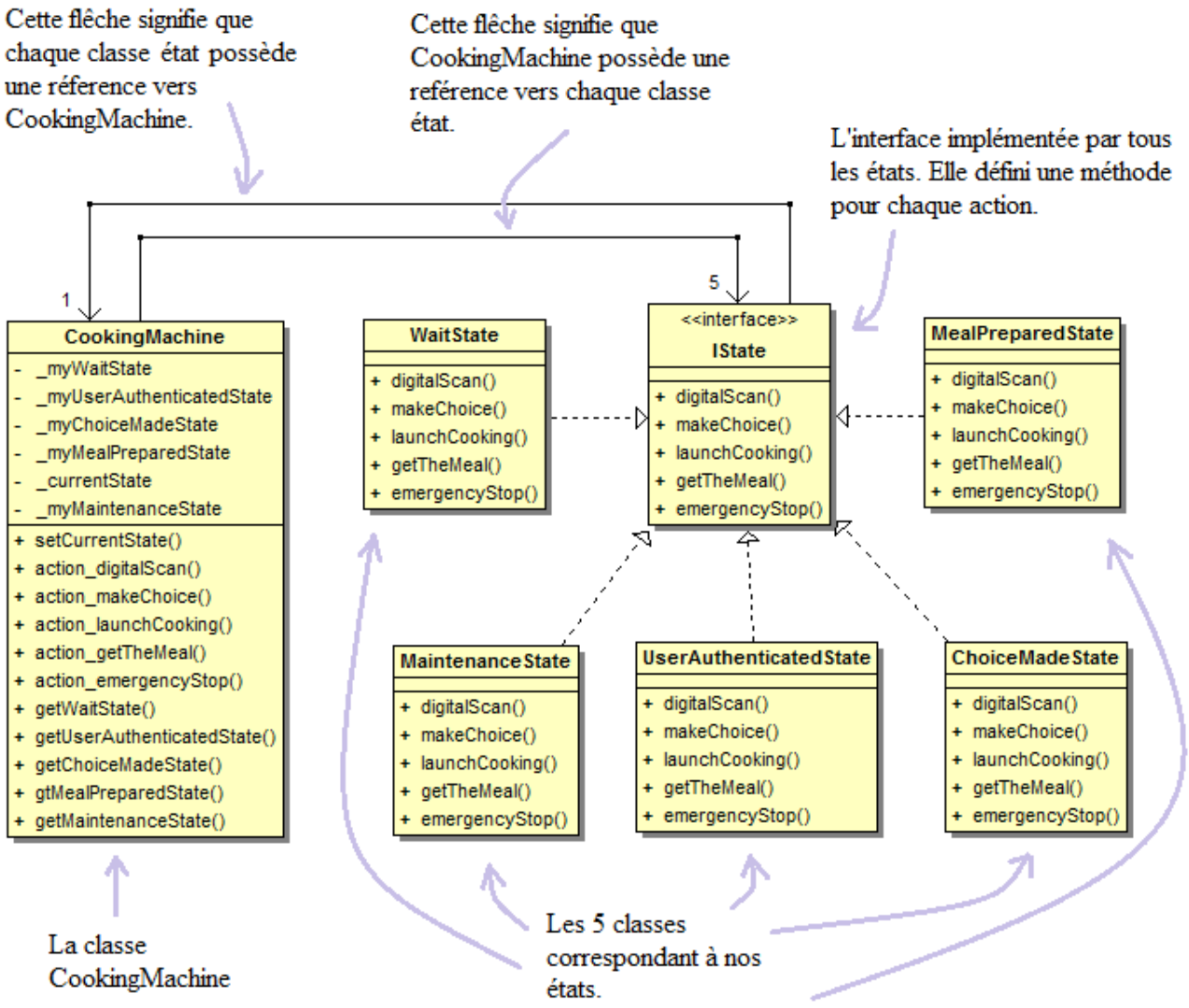
Le pattern State répond parfaitement à cette problématique avec une approche très simple:

- Nous aurons bien une classe pour notre distributeur automatique. Appelons la

CookingMachine.

- Chaque action devient une méthode dans la classe CookingMachine.
- Chaque état devient une classe implémentant une interface obligeant la classe à déclarer une méthode pour chaque action.

Voici le modèle de classe correspondant:



Voyons comment le montage fonctionne:

- C'est CookingMachine qui reçoit les actions des utilisateurs.
- CookingMachine possède une référence vers la classe représentant son état actuel, grâce à son attribut _currentState.
- Quand CookingMachine reçoit une action, elle ne fait que la transmettre (c'est à dire la

déléguer) à la classe désignée par son attribut _currentState.

- C'est donc bien la classe désignée par _currentState qui traitera concrètement l'action.
- De plus si l'action implique un changement d'état de CookingMachine, c'est encore la classe désignée par _currentState qui effectuera ce changement.
- Les classes représentant les états, implémentent toutes les actions possibles. Généralement, dans chaque classe, une seule action (c'est à dire une seule méthode) effectue un traitement significatif. Les autres, dans notre cas, affichent un message indiquant que l'action n'est pas recevable pour cet état.

On voit que CookingMachine n'est qu'une boite aux lettres: elle transmet les actions à son état actuel.

Il est aisé de modifier le comportement d'un état en n'intervenant que sur la classe concernée. Il est de même facile d'ajouter un nouvel état: il suffit de créer une nouvelle classe implémentant l'interface IState.

14.1 Codage

L'interface IState:

```php
<?php

interface IState {

    public function digitalScan();
    public function makeChoice();
    public function launchCooking();
    public function getTheMeal();
    public function emergencyStop();
}

?>
```

La classe ChoiceMadeState:

```php
<?php

class ChoiceMadeState implements IState {

    private $_myCookingMachine;

    public function __construct($CookingMachine) {
        $this->_myCookingMachine = $CookingMachine;
    }

    public function digitalScan() {
        echo "Action: digitalScan() impossible dans l'état actuel." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function makeChoice() {
        echo "Action: makeChoice() impossible dans l'état actuel." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function launchCooking() {
        echo "Action: launchCooking()..." . "<br>";
        echo "Le plat est prêt." . "<br>";
        $this->_myCookingMachine->setCurrentState($this->_myCookingMachine
                                        ->getMealPreparedState());
        $this->_myCookingMachine->displayCurrentState();
    }

    public function getTheMeal() {
        echo "Action: getTheMeal() impossible dans l'état actuel." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function emergencyStop() {
        echo "Action: emergencyStop()." . "<br>";
        $this->_myCookingMachine->setCurrentState($this->_myCookingMachine
                                        ->getMaintenanceState());
        $this->_myCookingMachine->displayCurrentState();
    }

}

?>
```

La référence vers CookingMachine, est passée au constructeur.

L'action est effectuée ici, et on modifie l'état de CookingMachine.

L'arret d'urgence sera implémenté dans tous les états.

La classe MaintenanceState:

```php
<?php

class MaintenanceState implements IState {

    private $_myCookingMachine;

    public function __construct($CookingMachine) {
        $this->_myCookingMachine = $CookingMachine;
    }

    public function digitalScan() {
        echo "Action: digitalScan() impossible. Je suis en maintenance." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function makeChoice() {
        echo "Action: makeChoice() impossible. Je suis en maintenance." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function launchCooking() {
        echo "Action: launchCooking() impossible. Je suis en maintenance." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function getTheMeal() {
        echo "Action: getTheMeal() impossible. Je suis en maintenance." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function emergencyStop() {
        echo "Action: emergencyStop()." . "<br>";
        echo "Je suis déjà en maintenance !" . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

}

?>
```

La classe MealPreparedState:

```php
<?php

class MealPreparedState implements IState {

    private $_myCookingMachine;

    public function __construct($CookingMachine) {
        $this->_myCookingMachine = $CookingMachine;
    }

    public function digitalScan() {
        echo "Action: digitalScan() impossible dans l'état actuel." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function makeChoice() {
        echo "Action: makeChoice() impossible dans l'état actuel." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function launchCooking() {
        echo "Action: launchCooking() impossible dans l'état actuel." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function getTheMeal() {
        echo "Action: getTheMeal()." . "<br>";
        echo "Veuillez retirer le plat." . "<br>";
        $this->_myCookingMachine->setCurrentState($this->_myCookingMachine
                                                  ->getWaitState());
        $this->_myCookingMachine->displayCurrentState();
    }

    public function emergencyStop() {
        echo "Action: emergencyStop()." . "<br>";
        $this->_myCookingMachine->setCurrentState($this->_myCookingMachine
                                                  ->getMaintenanceState());
        $this->_myCookingMachine->displayCurrentState();
    }

}

?>
```

La classe UserAuthenticatedState:

```php
<?php

class UserAuthenticatedState implements IState {

    private $_myCookingMachine;

    public function __construct($CookingMachine) {
        $this->_myCookingMachine = $CookingMachine;
    }

    public function digitalScan() {
        echo "Action: digitalScan() impossible dans l'état actuel." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function makeChoice() {
        echo "Action: makeChoice()." . "<br>";
        echo "Choix validé." . "<br>";
        $this->_myCookingMachine->setCurrentState($this->_myCookingMachine
                                                  ->getChoiceMadeState());
        $this->_myCookingMachine->displayCurrentState();
    }

    public function launchCooking() {
        echo "Action: launchCooking() impossible dans l'état actuel." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function getTheMeal() {
        echo "Action: getTheMeal() impossible dans l'état actuel." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function emergencyStop() {
        echo "Action: emergencyStop()." . "<br>";
        $this->_myCookingMachine->setCurrentState($this->_myCookingMachine
                                                  ->getMaintenanceState());
        $this->_myCookingMachine->displayCurrentState();
    }

}

?>
```

La classe WaitState:

```php
<?php

class WaitState implements IState {

    private $_myCookingMachine;

    public function __construct($CookingMachine) {
        $this->_myCookingMachine = $CookingMachine;
    }

    public function digitalScan() {
        echo "Action: digitalScan()." . "<br>";
        echo "Identification effectuée." . "<br>";
        $this->_myCookingMachine->setCurrentState($this->_myCookingMachine
                                                  ->getUserAuthenticatedState());
        $this->_myCookingMachine->displayCurrentState();
    }

    public function makeChoice() {
        echo "Action: makeChoice() impossible dans l'état actuel." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function launchCooking() {
        echo "Action: launchCooking() impossible dans l'état actuel." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function getTheMeal() {
        echo "Action: getTheMeal() impossible dans l'état actuel." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function emergencyStop() {
        echo "Action: emergencyStop()." . "<br>";
        $this->_myCookingMachine->setCurrentState($this->_myCookingMachine
                                                  ->getMaintenanceState());
        $this->_myCookingMachine->displayCurrentState();
    }

}

?>
```

La classe CookingMachine:

```php
<?php

class CookingMachine {

    private $_myWaitState;
    private $_myUserAuthenticatedState;
    private $_myChoiceMadeState;
    private $_myMealPreparedState;
    private $_myMaintenanceState;
    private $_myCurrentState;

    public function __construct() {
        $this->_myWaitState = new WaitState($this);
        $this->_myUserAuthenticatedState = new UserAuthenticatedState($this);
        $this->_myChoiceMadeState = new ChoiceMadeState($this);
        $this->_myMealPreparedState = new MealPreparedState($this);
        $this->_myMaintenanceState = new MaintenanceState($this);

        // Définition de l'état par defaut:
        $this->_myCurrentState = $this->_myWaitState;

        // On affiche l'état actuel:
        $this->displayCurrentState();
    }

    public function setCurrentState($newState) {
        $this->_myCurrentState = $newState;
    }

    public function action_digitalScan() {
        $this->_myCurrentState->digitalScan();
    }

    public function action_makeChoice() {
        $this->_myCurrentState->makeChoice();
    }

    public function action_launchCooking() {
        $this->_myCurrentState->launchCooking();
    }

    public function action_getTheMeal() {
        $this->_myCurrentState->getTheMeal();
    }

    public function action_emergencyStop() {
        $this->_myCurrentState->emergencyStop();
    }
```

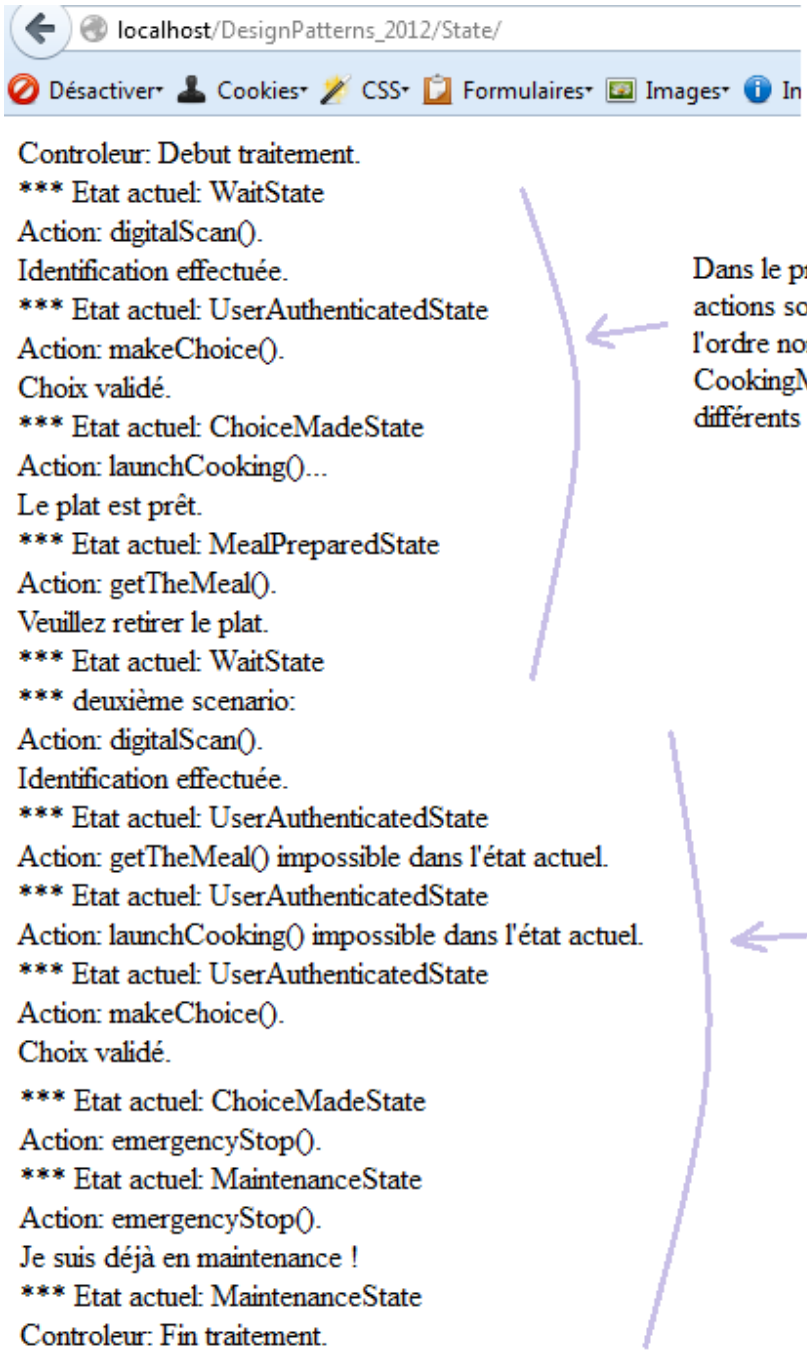
```
    public function getWaitState() {
        return $this->_myWaitState;
    }

    public function getUserAuthenticatedState() {
        return $this->_myUserAuthenticatedState;
    }

    public function getChoiceMadeState() {
        return $this->_myChoiceMadeState;
    }

    public function getMaintenanceState() {
        return $this->_myMaintenanceState;
    }

    public function getMealPreparedState() {
        return $this->_myMealPreparedState;
    }

    public function displayCurrentState() {
        echo "*** Etat actuel: " . get_class($this->_myCurrentState) . "<br>";
    }

}

?>
```

14.2 TESTS

Comme d'habitude nous allons utiliser index.php comme contrôleur de l'ensemble du montage.

```php
<?php

//*************************************************************
// STATE pattern controler
//*************************************************************

require_once 'includePaths.php';
$newline = "</br>";

echo 'Controleur: Debut traitement.' . $newline;

// Instanciations:
$myCookingMachine = new CookingMachine();

// traitemment (scenario normal):
$myCookingMachine->action_digitalScan();
$myCookingMachine->action_makeChoice();
$myCookingMachine->action_launchCooking();
$myCookingMachine->action_getTheMeal();

// traitemment (scenario avec des actions non attendues):
echo "*** deuxième scenario:" . "<br>";
$myCookingMachine->action_digitalScan();
$myCookingMachine->action_getTheMeal();
$myCookingMachine->action_launchCooking();
$myCookingMachine->action_makeChoice();
$myCookingMachine->action_emergencyStop();
$myCookingMachine->action_emergencyStop();

echo 'Controleur: Fin traitement.' . $newline;

?>
```

Voici le résultat de l'exécution:



15 PATTERN ITERATOR

Alors Mr Sulu, expliquez-moi pourquoi je ne peux pas avoir la liste de toutes les salles du vaisseau.

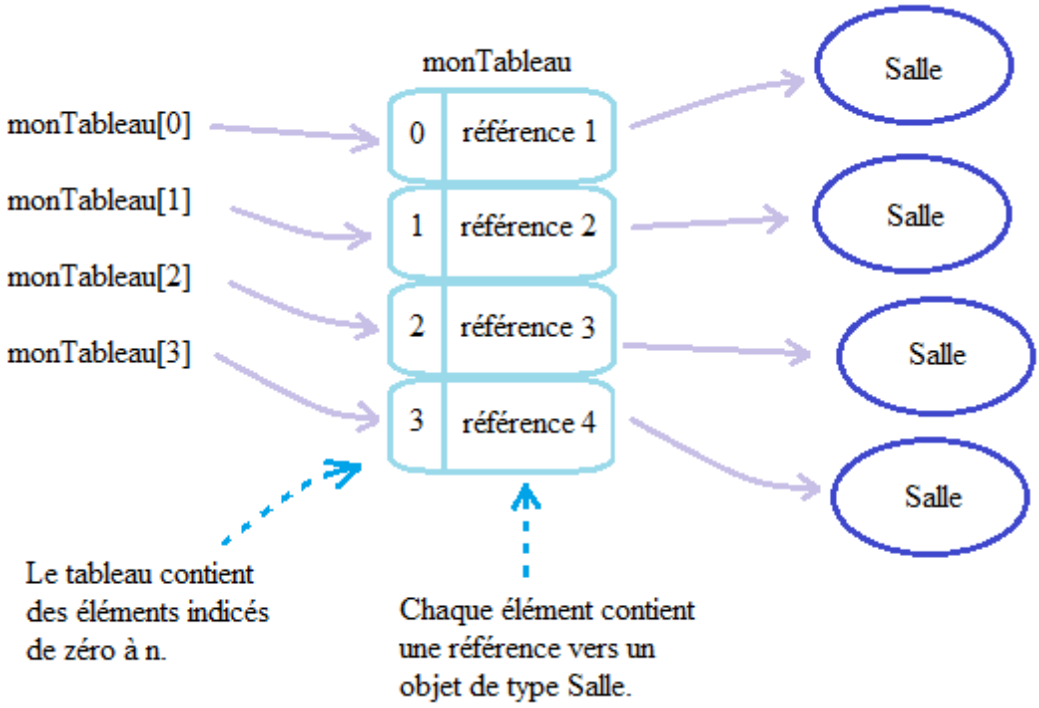
Le système nous donne les salles des ponts A et B, mais pas celles des ponts C et D.

Il est possible que les ponts C et D fonctionnent encore avec l'ancien système informatique.

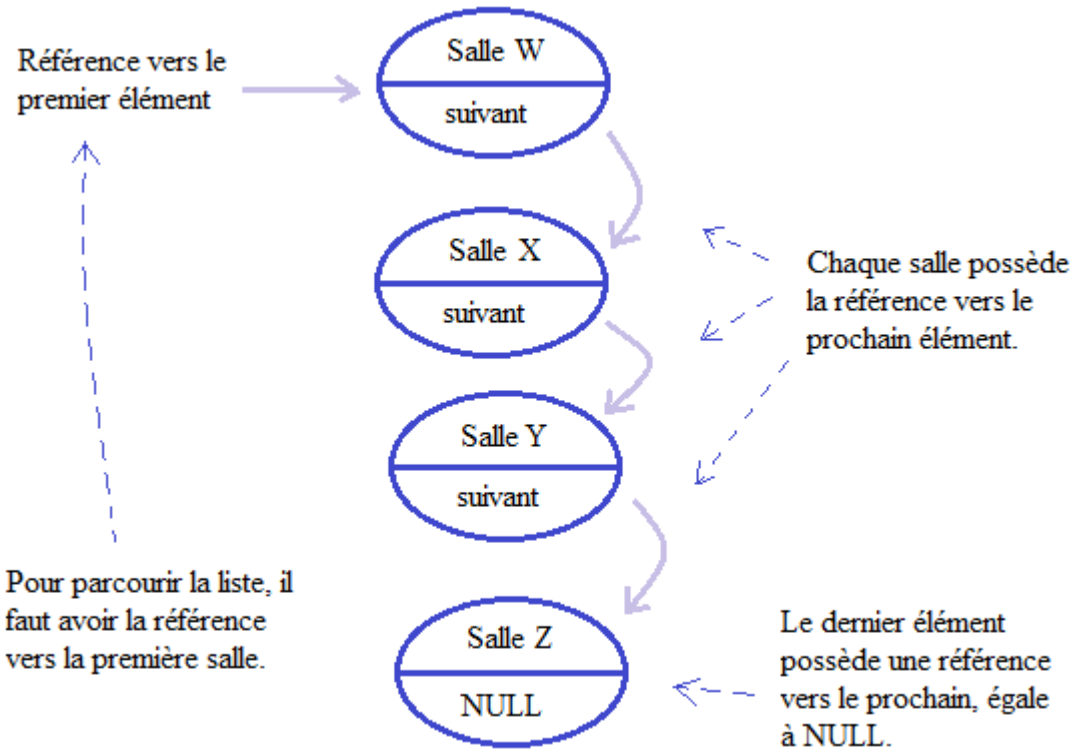
On ne va pas se laisser embêter par un petit problème de compatibilité, n'est ce pas Mr Sulu ?

Le problème a rapidement été identifié.

Les ponts A et B sont gérés par le nouveau programme qui stocke les salles sous forme d'un tableau de références. Pour obtenir la liste des salles il faut parcourir les éléments du tableau en faisant varier l'indice de zéro à n:



Les ponts C et D sont gérés par l'ancien programme qui stocke les salles sous forme d'une liste chaînée. Pour obtenir la liste des salles il faut partir du premier élément et suivre les références jusqu'au dernier élément:



Notre objectif étant de disposer d'un programme capable de fournir la liste de toutes les salles de l'Enterprise (c'est à dire les ponts A, B, C et D), nous sommes confrontés à un problème d'incompatibilité.

Pour résoudre cette incompatibilité, une première approche nous inciterait à envisager deux traitements:

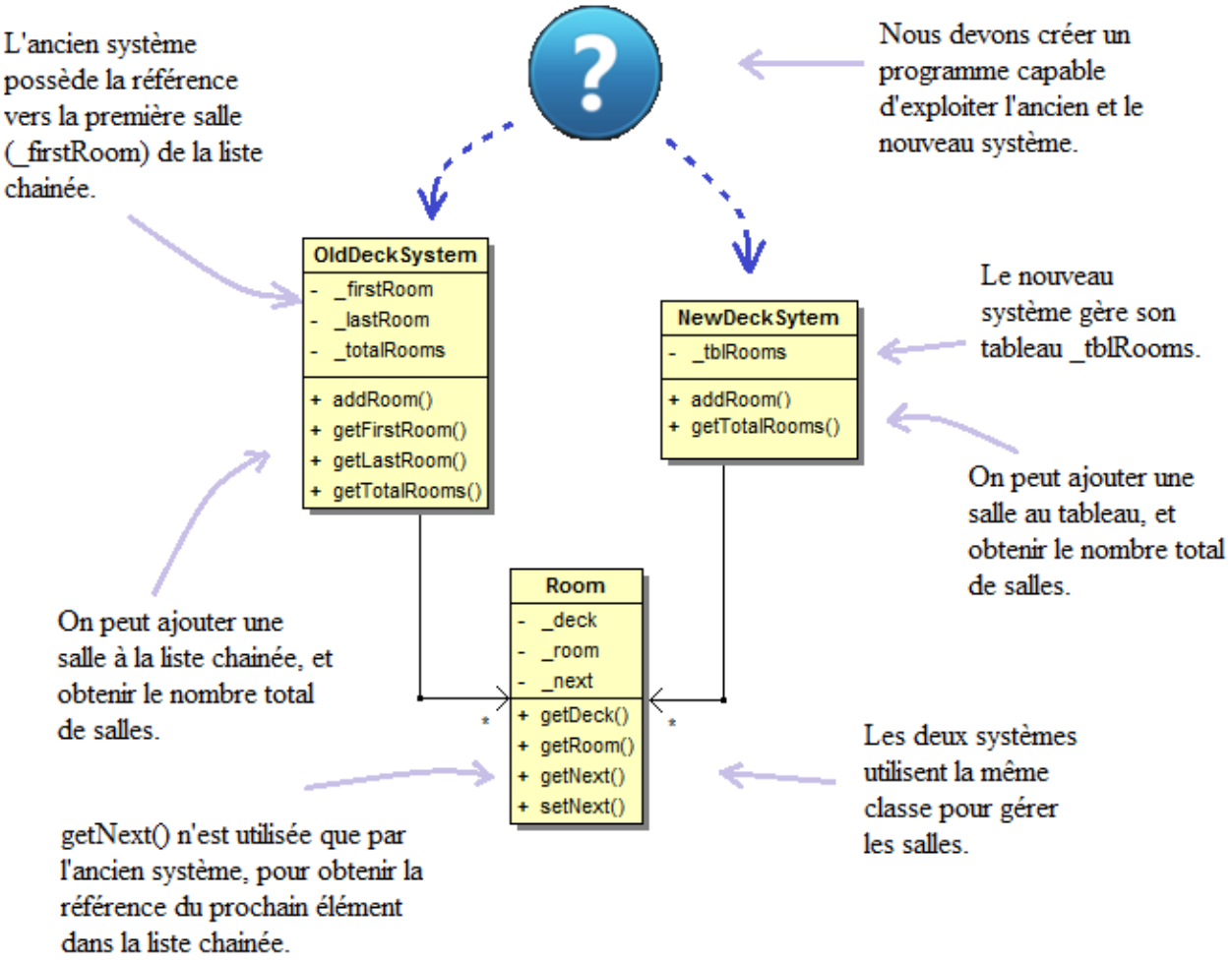
- Un traitement pour les ponts A et B, sachant itérer sur une structure de type Tableau (c'est le traitement existant).
- Un traitement pour les ponts C et D, sachant itérer sur une structure de type Liste Chaînée.

Nous devinons aisément les inconvénients de cette approche:

- Si la logique de parcours des salles change à l'avenir, il faudrait modifier les deux traitements.
- Si une nouvelle structure de stockage des salles apparaît, il faudrait ajouter un troisième traitement.

Pour ces raisons nous ne nous engagerons pas sur cette voie.

Faisons un point sur le modèle de classes actuel:


L'ancien système possède la référence vers la première salle (_firstRoom) de la liste chainée.
Nous devons créer un programme capable d'exploiter l'ancien et le nouveau système.
OldDeckSystem
- _firstRoom
- _lastRoom
- _totalRooms
+ addRoom()
+ getFirstRoom()
+ getLastRoom()
+ getTotalRooms()
NewDeckSytem
- _tblRooms
+ addRoom()
+ getTotalRooms()
Le nouveau système gère son tableau _tblRooms.
On peut ajouter une salle au tableau, et obtenir le nombre total de salles.
Room
- _deck
- _room
- _next
+ getDeck()
+ getRoom()
+ getNext()
+ setNext()
On peut ajouter une salle à la liste chainée, et obtenir le nombre total de salles.
Les deux systèmes utilisent la même classe pour gérer les salles.
getNext() n'est utilisée que par l'ancien système, pour obtenir la référence du prochain élément dans la liste chainée.

15.1 Présentation du pattern Iterator

On peut résumer le pattern Iterator en deux points:

- Il sépare le traitement d'itération, de la chose sur laquelle on itère. Cela signifie que la classe qui contient la collection sur laquelle on veut itérer, ne propose pas de traitement permettant d'itérer sur cette collection. Ce traitement sera isolé dans une autre classe appelée Itérateur.
- L'itérateur aura une interface unifiée, c'est à dire qu'elle proposera toujours au moins les méthodes standard suivantes:
  - hasNext(): cette méthode renvoi VRAI tant qu'il y a encore au moins un élément à parcourir dans la collection.
  - next(): cette méthode renvoi la référence vers le prochain élément à parcourir.

Comment utilise-t-on l'itérateur ?

- Le programme souhaitant itérer sur une collection, doit demander à la classe gérant cette collection, de lui fournir la référence de son itérateur.
- Une fois l'itérateur obtenu, il suffit de l'utiliser dans une boucle standard:

```
Tant que monItérateur.hasNext()
        monItérateur.next()
        // on traite l'élément
        // ...
Fin
```

On comprend que le programme qui souhaite itérer sur la collection, ne sait absolument pas comment est gérée cette collection ni comment on itère concrètement dessus.

Elle doit simplement disposer de l'itérateur correspondant à cette collection, et utiliser les deux méthodes hasNext() et next(). C'est l'itérateur qui effectue concrètement le parcours de la collection.

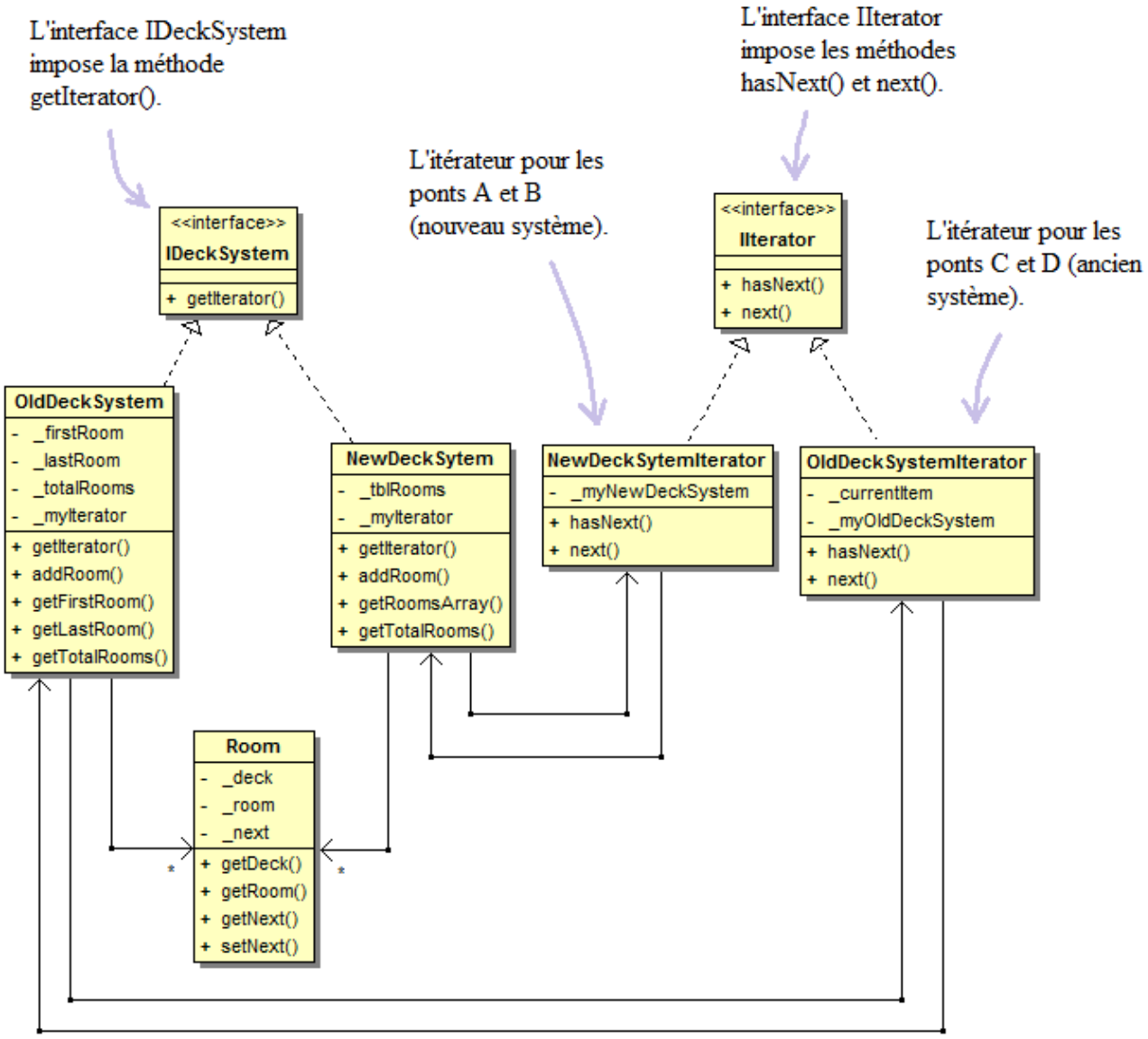
Il va falloir maintenant faire évoluer notre modèle de classe pour en faire un véritable pattern Iterator. Pour cela nous devons effectuer les modifications suivantes:

- Nous allons faire en sorte que les classes OldDeckSystem et NewDeckSytem implémentent une interface IDeckSystem, les obligeant à déclarer une méthode getIterator(). En effet ces

deux classes seront tenues de fournir leurs itérateurs respectifs.

- Nous allons créer une interface IIterator obligeant à déclarer les deux méthodes hasNext() et next().
- Nous allons créer une classe NewDeckSytemIterator implémentant l'interface IIterator. Ce sera notre itérateur pour le nouveau système (celui des ponts A et B).
- Nous allons créer une classe OldDeckSystemIterator implémentant l'interface IIterator. Ce sera notre itérateur pour l'ancien système (celui des ponts C et D).

Voici le nouveau modèle de classes:



N'oublions pas que nous voulons faire un programme qui affiche la liste de toutes les salles de l'Enterprise.

Le programme devra utiliser le montage de la manière suivante:

- Disposer d'une référence sur un objet de type OldDeckSystem.
- Disposer d'une référence sur un objet de type NewDeckSystem.
- Obtenir auprès de l'objet OldDeckSystem, une référence vers son itérateur en utilisant la méthode getIterator().

- Parcourir les salles gérées par cet itérateur en utilisant les méthodes hasNext() et next().
- Obtenir auprès de l'objet NewDeckSystem, une référence vers son itérateur en utilisant la méthode getIterator().
- Parcourir les salles gérées par cet itérateur en utilisant les méthodes hasNext() et next().

Le code sera finalement assez simple car toute la complexité est gérée par les itérateurs.

Passons donc au codage.

15.2 Codage

L'interface IDeckSystem:

```php
<?php

interface IDeckSystem {

    public function getIterator();
}
?>
```

L'interface IIterator:

```php
<?php

interface IIterator {

    public function next();
    public function hasNext();

}
?>
```

La classe Room:

```php
<?php

class Room {

    private $_deck;
    private $_room;
    private $_next;

    public function __construct($deck = "", $room = "") {
        $this->_deck = $deck;
        $this->_room = $room;
        $this->_next = NULL;
    }

    public function getDeck() {
        return $this->_deck;
    }

    public function getRoom() {
        return $this->_room;
    }

    public function getNext() {
        return $this->_next;
    }

    public function setNext($NextNode) {
        $this->_next = $NextNode;
    }

}

?>
```

L'attribut _next, n'est utilisé que par l'ancien système (ponts C et D).

Méthode utilisée par l'ancien système pour définir l'élément suivant dans la liste chainée.

La classe OldDeckSytem:

```php
<?php

class OldDeckSystem implements IDeckSystem {

    private $_firstRoom;
    private $_lastRoom;
    private $_totalRooms;
    private $_myIterator;

    public function __construct() {
        echo "je suis le constructeur de OldDeckSystem", "<br>";
        $this->_firstRoom = NULL;
        $this->_lastRoom = NULL;
        $this->_totalRooms = 0;

        // On ajoute des elements dans la liste chainée:
        $this->addRoom("A", "Salle de commandement");
        $this->addRoom("A", "Salle de repos");
        $this->addRoom("A", "Salle des communications");
        $this->addRoom("B", "Local technique");

        // on instancie l'itérateur:
        $this->_myIterator = new OldDeckSystemIterator($this);
    }

    public function getFirstRoom() {
        return $this->_firstRoom;
    }

    public function getLastRoom() {
        return $this->_lastRoom;
    }

    public function addRoom($deck, $room) {
        $newRoom = new Room($deck, $room);
        $this->_totalRooms++;

        if ($this->_firstRoom == NULL) {
            // It's the first room:
            $this->_firstRoom = $newRoom;
        } else {
            // It's not the first room:
            $this->_lastRoom->setNext($newRoom);
        }

        $this->_lastRoom = $newRoom;
    }
```

_firstRoom réference le premier élément de la liste chainée.

$this permettra à l'itérateur d'accéder à OldDeckSytem.

C'est ici que les éléments de la liste sont chainés entre eux.

```
    public function getTotalRooms() {
        return $this->_totalRooms;
    }

    public function getIterator() {
        return $this->_myIterator;
    }

}

?>
```

La classe NewDeckSytem:

```php
<?php

class NewDeckSytem implements IDeckSystem {

    private $_tblRooms;
    private $_myIterator;

    public function __construct() {
        echo "je suis le constructeur de NewDeckSystem", "<br>";

        // On ajoute des salles dans le tableau:
        $this->addRoom("C", "Infirmerie");
        $this->addRoom("C", "Restaurant");
        $this->addRoom("D", "Salle de réunion");

        // on instancie l'itérateur:
        $this->_myIterator = new NewDeckSytemIterator($this);
    }

    public function getIterator() {
        return $this->_myIterator;
    }

    public function addRoom($deck, $room) {
        $this->_tblRooms[] = new Room($deck, $room);
    }

    public function getRoomsArray() {
        return $this->_tblRooms;
    }

    public function getTotalRooms() {
        return count($this->_tblRooms);
    }

}

?>
```

On ajoute une salle au tableau.

La classe NewDeckSytemIterator:

```php
<?php

class NewDeckSytemIterator implements IIterator {

    private $_currentIndex;
    private $_totalRooms;
    private $_tblRooms;
    private $_myNewDeckSystem;

    public function __construct($NewDeckSystem) {
        echo "je suis le constructeur de NewDeckSytemIterator", "<br>";
        $this->_myNewDeckSystem = $NewDeckSystem;
        $this->_tblRooms = $this->_myNewDeckSystem->getRoomsArray();
        $this->_totalRooms = count($this->_tblRooms);

        if ($this->_totalRooms > 0) {
            $this->_currentIndex = 0;  // first room.
        } else {
            $this->_currentIndex = -1; // no rooms
        }
    }

    public function hasNext() {
        if ($this->_currentIndex != -1) {
            return TRUE;
        } else {
            return FALSE;
        }
    }

    public function next() {
        $returnedItem = $this->_tblRooms[$this->_currentIndex];

        if ($this->_currentIndex < $this->_totalRooms - 1) {
            $this->_currentIndex++;
        } else {
            // It was the last Room:
            $this->_currentIndex = -1;
        }
        return $returnedItem;
    }

}

?>
```

On passe la référence vers NewDeckSystem à l'itérateur.

L'itérateur récupère le tableau des salles auprès de NewDeckSystem.

Les deux méthodes hasNext() et next().

La classe OldDeckSystemIterator:

```php
<?php

class OldDeckSystemIterator implements IIterator {

    private $_currentItem;
    private $_myOldDeckSystem;


    public function __construct($oldDeckSystem) {
        $this->_myOldDeckSystem = $oldDeckSystem;
        $this->_currentItem = $this->_myOldDeckSystem->getFirstRoom();
        echo "je suis le constructeur de OldDeckSystemIterator","<br>";
    }


    public function hasNext() {
        if($this->_currentItem != NULL){
            return TRUE;
        } else {
            return FALSE;
        }
    }

    public function next() {
        $returnedItem = $this->_currentItem;
        $this->_currentItem = $this->_currentItem->getNext();
        return $returnedItem;
    }

}

?>
```

15.3 Tests

Comme d'habitude nous allons utiliser index.php comme contrôleur de l'ensemble du montage:

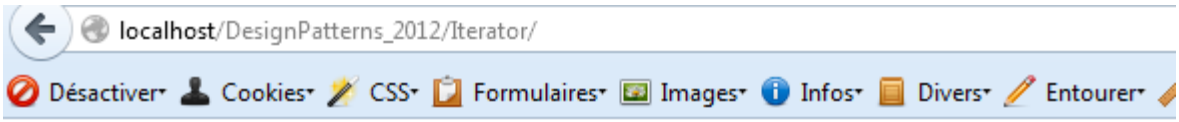
```php
<?php

//*************************************************************
// ITERATOR pattern controler
//*************************************************************

require_once 'includePaths.php';
$newline = "</br>";

echo 'Controleur: Debut traitement.' . $newline;

// Instanciations:
$myOldDeckSystem = new OldDeckSystem();
$myNewDeckSystem = new NewDeckSytem();

echo "*** Old deck system:", "<br>";
echo "nbr rooms: " . $myOldDeckSystem->getTotalRooms() . "<br>";

// On parcour la liste chainée via l'itérateur:
if ($myOldDeckSystem->getTotalRooms() > 0) {
    DisplayRoomsList($myOldDeckSystem->getIterator());
}

echo "<br>";
echo "*** New deck system:", "<br>";
echo "nbr rooms: " . $myNewDeckSystem->getTotalRooms() . "<br>";

// On parcour le tableau via l'itérateur:
if ($myNewDeckSystem->getTotalRooms() > 0) {
    DisplayRoomsList($myNewDeckSystem->getIterator());
}

function DisplayRoomsList($iterator) {
    // on utilise l'itérateur passé en parametre:
    while ($iterator->hasNext() == TRUE) {
        $currentRoom = $iterator->next();
        $deck = $currentRoom->getDeck();
        $room = $currentRoom->getRoom();
        echo "Pont ", $deck, ": ", $room, "<br>";
    }
}

echo 'Controleur: Fin traitement.' . $newline;
?>
```

On instancie les deux systèmes.

On appelle DisplayRoomsList() en lui passant l'itérateur de l'ancien ou du nouveau système.

DisplayRoomsList() est capable d'utiliser l'itérateur de l'ancien ou du nouveau système.

Voici le résultat de l'exécution:



Controleur: Debut traitement.
je suis le constructeur de OldDeckSystem
je suis le constructeur de OldDeckSystemIterator
je suis le constructeur de NewDeckSystem
je suis le constructeur de NewDeckSytemIterator

Chaque itérateur est instancié par le constructeur du système auquel il appartient.

*** Old deck system:
nbr rooms: 4
Pont A: Salle de commandement
Pont A: Salle de repos
Pont A: Salle des communications
Pont B: Local technique

Les itérateurs proposent une interface simple pour parcourir les salles, et gèrent en interne la compléxité du parcours.

*** New deck system:
nbr rooms: 3
Pont C: Infirmerie
Pont C: Restaurant
Pont D: Salle de réunion
Controleur: Fin traitement.

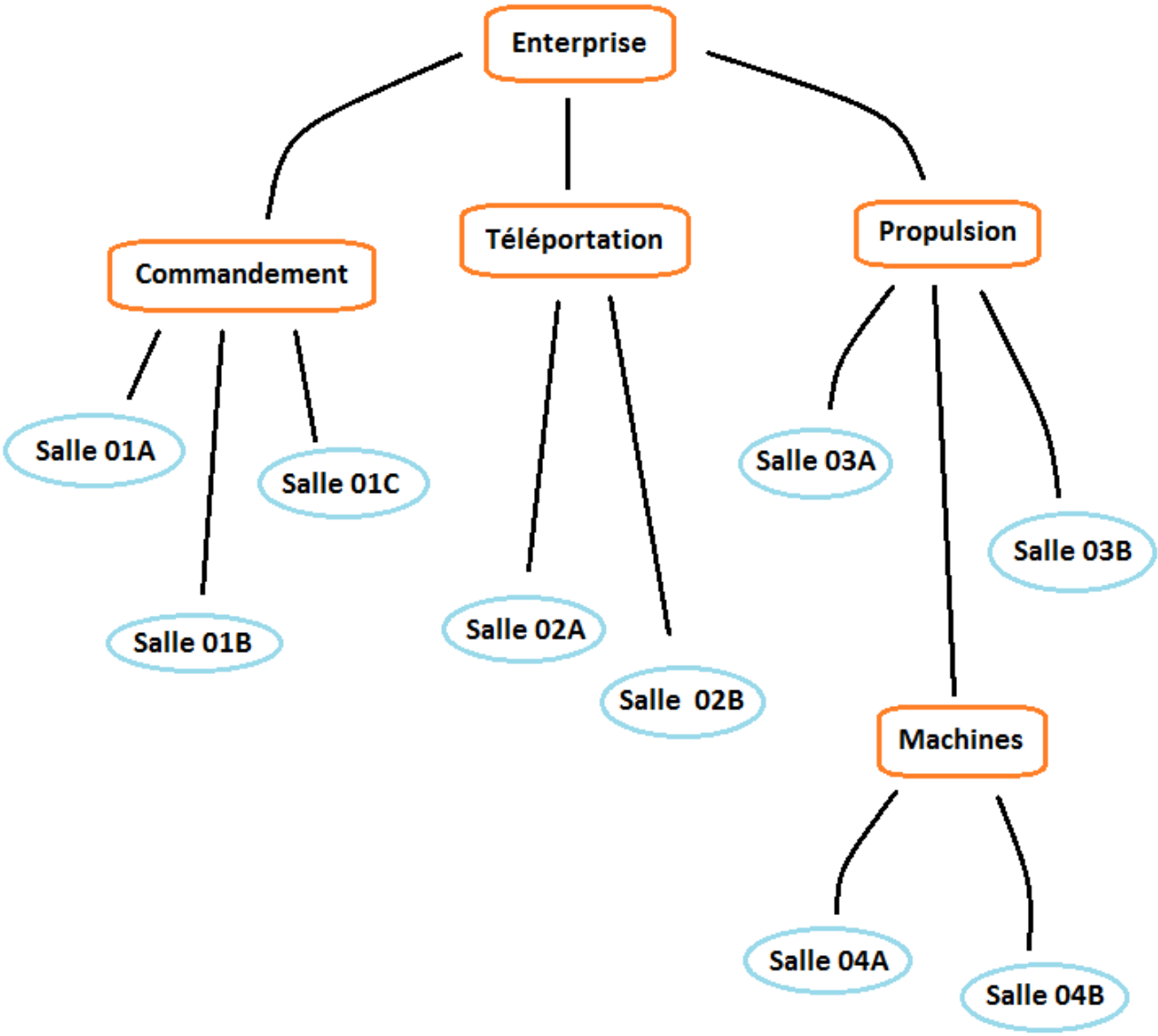
Merci Mr Sulu.
Voila ce qui me semble une solution des plus élégantes.

16 PATTERN COMPOSITE

Avec le pattern Iterator nous avons vu comment encapsuler les opérations d'itérations sur une collection, et offrir une interface unifiée facilitant ces opérations.

Le pattern Composite quant à lui, va nous permettre d'itérer sur des structures arborescentes composites. Il faut entendre par là que chaque élément de l'arbre peut être un élément simple appelé feuille, ou un élément composite, ce dernier pouvant à son tour contenir des feuilles ou d'autres éléments composites.

Voici l'arborescence définie par le capitaine :



Nous trouvons les secteurs:

- Commandement
- Téléportation
- Propulsion
- Machines

Le secteur Machine est inclus dans le secteur Propulsion.

Les salles sont des éléments simples appelés feuilles.

Il y a un élément racine nommé Enterprise et qui représente l'ensemble du vaisseau.

Le parcours d'une structure arborescente fait appel à la récursivité. Il s'agit d'un type de programmation particulier dans lequel un traitement peut s'exécuter lui-même de manière imbriquée. Lire à ce sujet l'annexe "La récursivité" en fin de document.

16.1 Comment parcourir un arbre

Les structures arborescentes ont leur terminologie:

- Tous les éléments de l'arbre sont appelés des nœuds (en anglais: node).
- Il y a toujours un nœud racine qui contient tous les autres. Dans notre cas c'est le nœud Enterprise.
- Les nœuds qui ne contiennent rien sont appelés des feuilles. Pour nous ce sont les salles.
- Les enfants sont les nœuds immédiatement situés sous un nœud parent. Par exemple, le nœud Propulsion a 3 enfants: Salle 03A, Salle 03B, et Machines.

Dans l'arbre de l'Enterprise, nous trouvons deux types d'éléments:

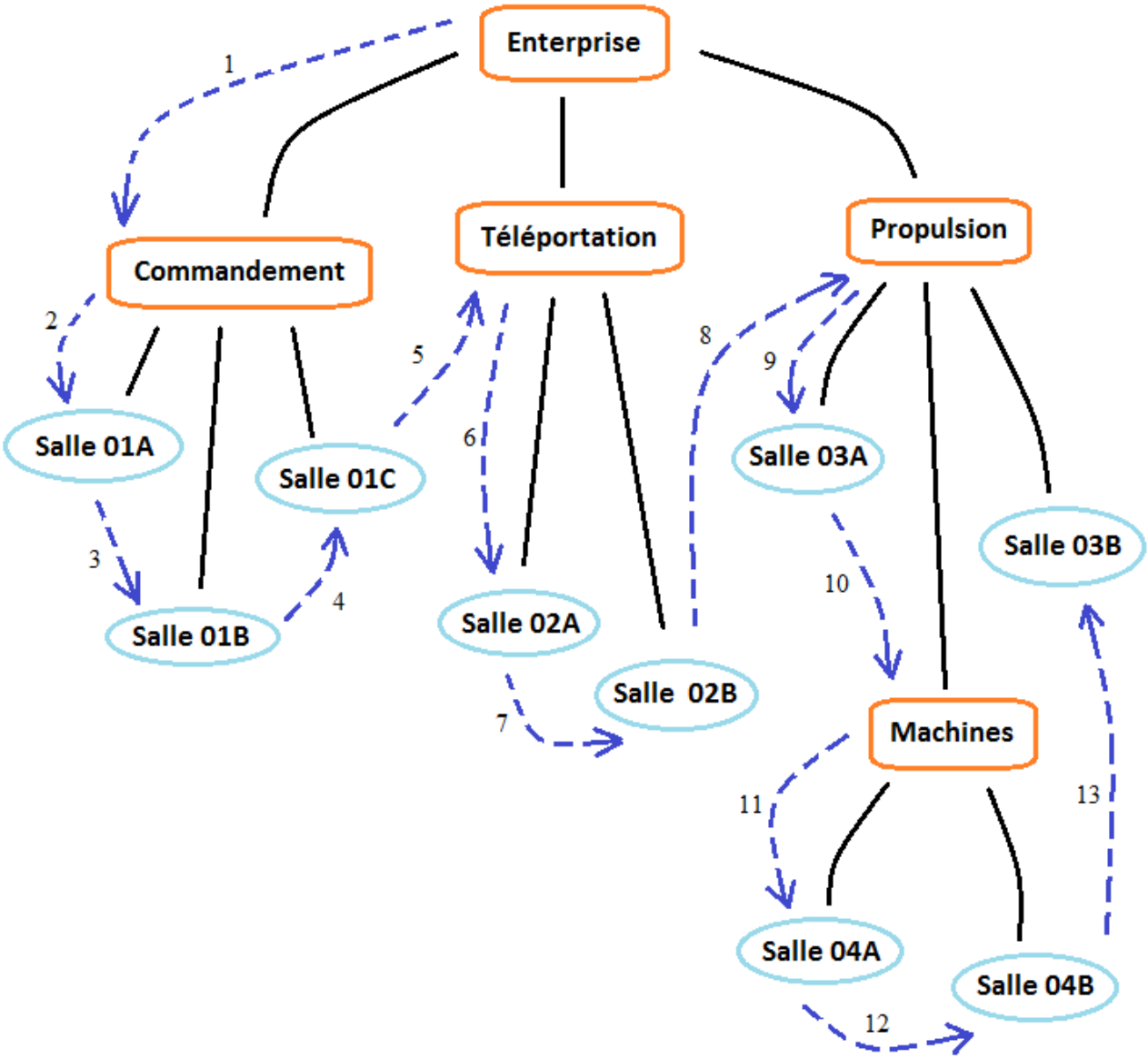
- Les salles: il s'agit de nœuds simples qui ne contiennent rien (feuilles).
- Les secteurs: les enfants de ces nœuds peuvent être des salles ou d'autres secteurs. Par exemple le secteur Téléportation contient deux salles. Le secteur Propulsion contient deux salles et un secteur.

Le parcours d'un arbre débute à la racine.

On parcourt les enfants de la racine et, si un enfant est un composite (un secteur), alors on l'explore immédiatement en appliquant la même approche.

Chaque fois qu'un secteur est exploré, c'est un nouveau processus d'exploration qui démarre, et le processus d'exploration du parent est mis en attente. C'est le principe de la récursivité.

Selon les règles que nous venons d'évoquer, le parcours de notre arbre doit être le suivant (suivre les flèches bleues):



Voici l'objectif du pattern Composite: parcourir une structure arborescente de cette manière. Finalement ce n'est pas si compliqué !

16.2 Comment fonctionne le pattern Composite

Pour ce design pattern, deux solutions seront proposées au niveau de l'implémentation du parcours: A et B.

Il s'agit de deux variantes aussi valables l'une que l'autre en terme de performance.

La variante B étant un peu plus simple, nous commencerons par elle. Dans cette variante, toutes les classes seront suffixées par _B.

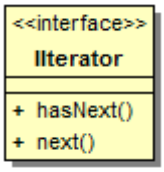
Il est très important d'avoir assimilé le pattern Iterator, vu précédemment, car il est utilisé dans le pattern Composite.

En effet, quand le pattern Composite parcourt l'arbre et qu'il rencontre un secteur, il va lui demander de fournir son itérateur. Cet itérateur n'est autre qu'un pattern Iterator.

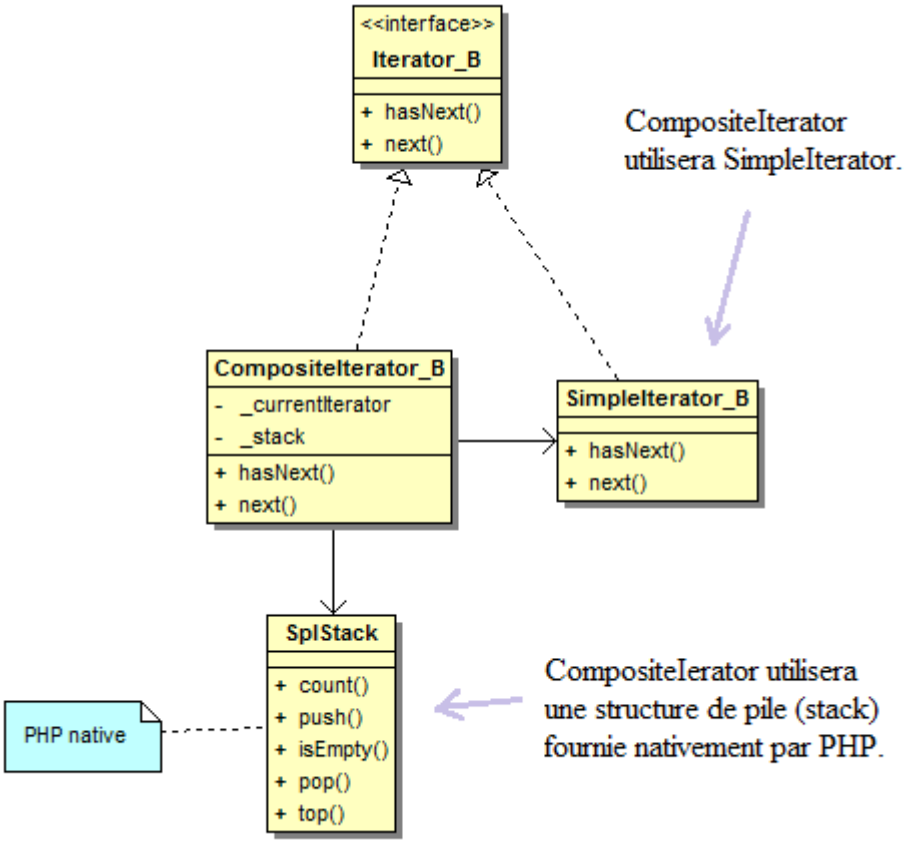
On devine donc l'utilisation de deux types d'itérateurs dans le pattern Composite:

- L'itérateur fourni par un secteur: nous l'appellerons "SimpleIterator".
- L'itérateur de l'arbre dans sa globalité: nous l'appellerons CompositeIterator.

Souvenez-vous dans le pattern Iterator, nous avions défini une interface pour imposer les méthodes hasNext() et next():



Nous reprenons cette interface dans le pattern Composite, et nous définissons deux classes qui l'implémentent:



SplStack est une classe fournie par le langage PHP. Elle sera utilisée ici pour empiler les processus successifs d'exploration des secteurs. Ceci est directement lié à la récursivité.

A la place d'une pile, il aurait été possible d'utiliser un simple tableau. Mais les occasions d'utiliser une pile sont rares. Il ne faut pas s'en priver !

Rappel sur les piles:

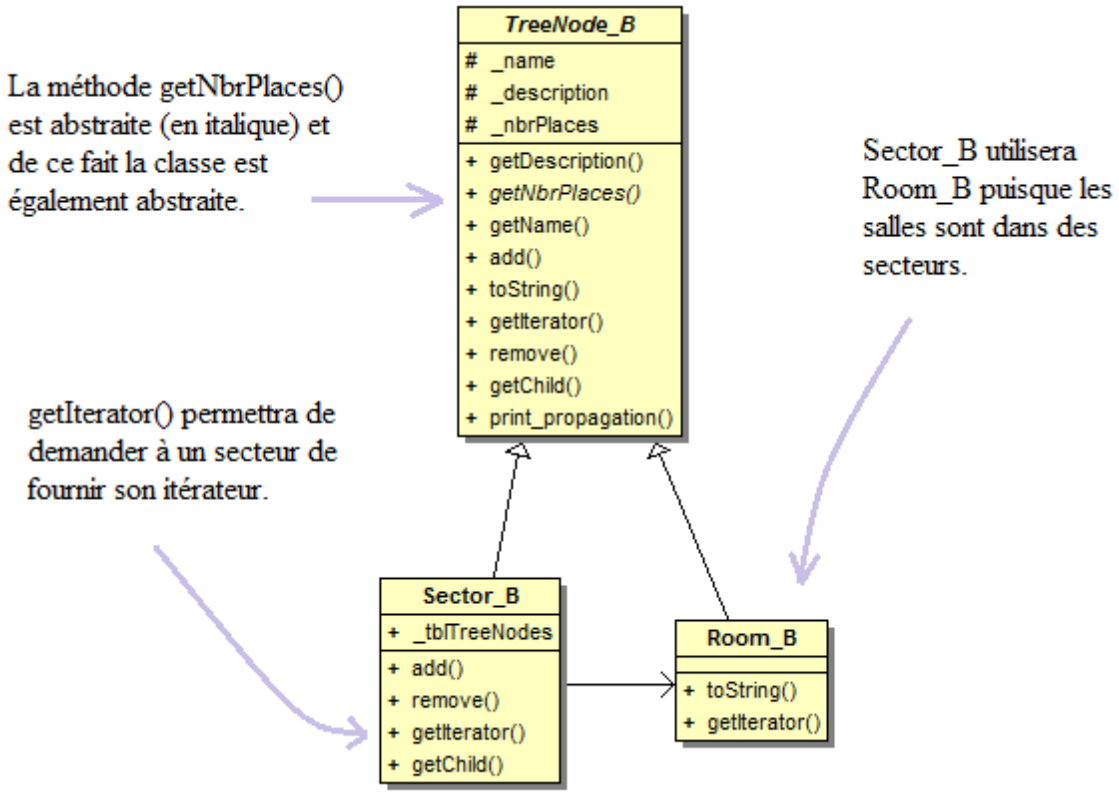
Les piles font partie des structure de données en informatique, au même titre que les variables, les tableaux, ou les listes chaînées…

La pile se comporte comme une pile d'assiettes:

- On peut empiler un élément supplémentaire: **push()**.
- On peut dépiler (supprimer) l'élément du haut: **pop()**.
- On peut lire l'élément du haut sans le dépiler: **top()**.
- On peut savoir si la pile est vide: **isEmpty()**.
- On peut savoir combien d'éléments il y a dans la pile: **count()**.

Nous avons également besoin de classes pour matérialiser les nœuds de l'arbre. Définissons une

classe TreeNode_B dont dérivent Sector_B et Room_B.

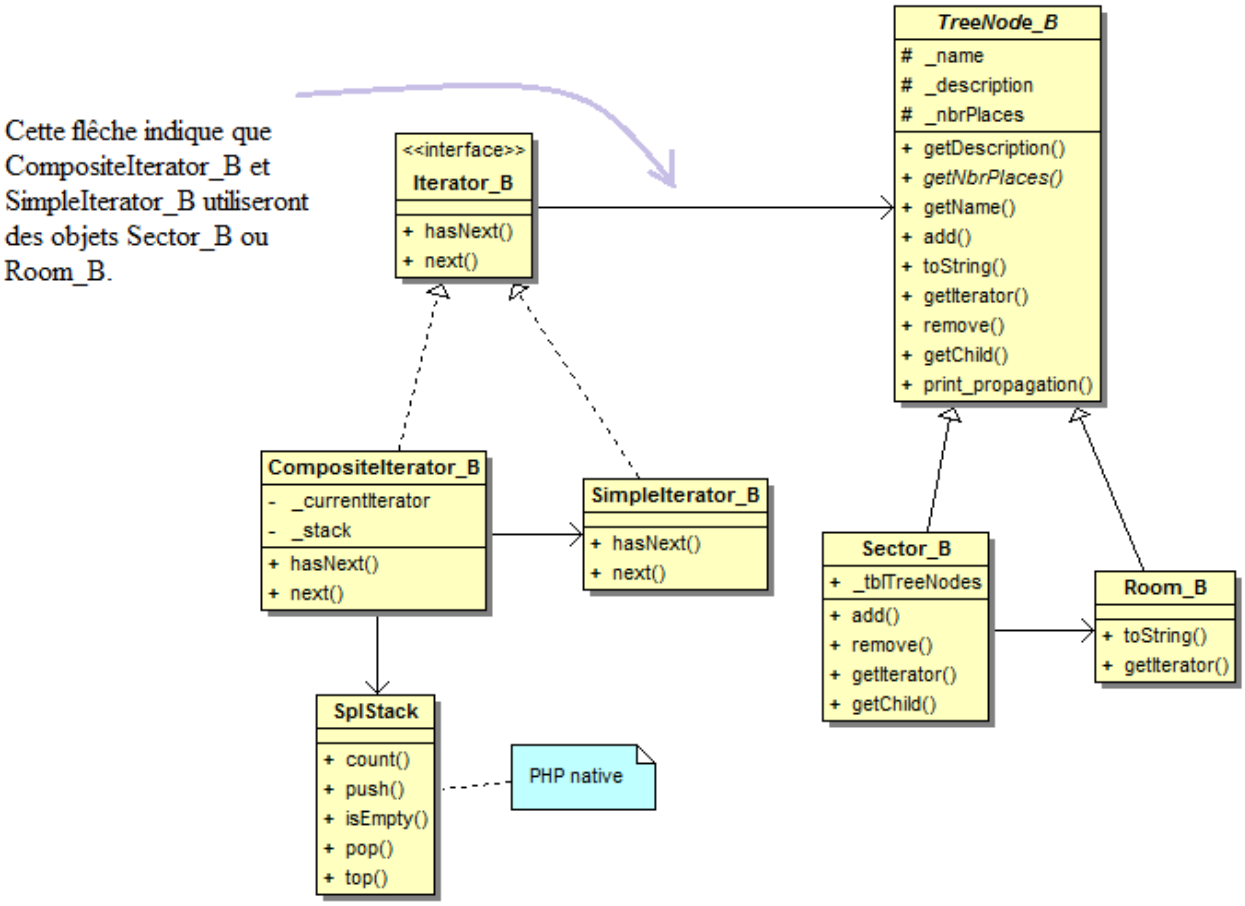


Sector_B représentera un secteur, et Room_B, une salle.

Certaines méthodes définies au niveau de TreeNode_B, ne s'appliquent qu'à Sector_B.

Par exemple la méthode add() qui permet d'ajouter une salle à un secteur, n'a pas de sens pour une salle (on ne peut pas ajouter une salle à une salle). Pour pallier ce petit problème, un comportement par défaut sera défini dans la classe TreeNode_B pour certaines méthodes, et ces dernières seront surchargées, si nécessaire, dans les sous classes. Nous y reviendrons en examinant le code.

Voici le montage complet:



16.3 Codage de la variante B

Il faut bien plonger dans le code à un moment donné. Nous allons commencer par les classes de base qui ne posent pas de problèmes particuliers.

Nous terminerons par le code de la classe CompositeIterator_B qui gère la récursivité et qui représente la partie difficile de ce pattern.

La classe TreeNode_B:

```php
<?php

abstract class TreeNode_B {
    const DEFAULT_ERROR_MESSAGE = "Methode non supportée par cette classe.";

    protected $_name;
    protected $_description;

    public function __construct($name = "", $description = "") {
        $this->_description = $description;
        $this->_name = $name;
    }

    public abstract function getNbrPlaces();
    public abstract function getIterator();

    public function getDescription() {
        return $this->_description;
    }

    public function getName() {
        return $this->_name;
    }

    public function add($treeNode) {
        throw new Exception(TreeNode_B::DEFAULT_ERROR_MESSAGE);
    }

    public function remove($treeNode) {
        throw new Exception(TreeNode_B::DEFAULT_ERROR_MESSAGE);
    }

    public function getChild($indice) {
        throw new Exception(TreeNode_B::DEFAULT_ERROR_MESSAGE);
    }

    public function print propagation() {
        echo $this->__toString();
        // Il faut aussi invoquer la méthode print_propagation() des enfants:
        foreach ($this->_tblTreeNodes as $key => $component) {
            $component->print_propagation();
        }
    }

    public function __toString() {
        // Affichag par défaut:
        return $this->_name . ": " . $this->_description . '</br>';
    }

}

?>
```

A la création d'un noeud, il faut lui donner un nom et une description.

Ces méthodes sont abstraites car elles auront des implémentations différentes dans Sector_B et Room_B.

Ces méthodes ne concernent que la classe Sector_B.

Nous parlerons de cette méthode ultérieurement.

Les méthodes add(), remove() et getChild() ont une implémentation par défaut qui renvoi une erreur. Cette implémentation sera celle héritée par Room_B.

Par contre dans Sector_B nous redéfinirons ces 3 méthodes puisqu'elles n'ont de sens que dans cette classe.

La classe Sector_B:

```php
<?php

class Sector_B extends TreeNode_B {

    public $_tblTreeNodes = array();

    public function add($treeNode) {
        $this->_tblTreeNodes[] = $treeNode;
    }

    public function getIterator() {
        return new SimpleIterator_B($this);
    }

    public function remove($treeNode) {
        foreach (array_keys($this->_tblTreeNodes) as $key) {
            if ($this->_tblTreeNodes[$key] === $treeNode) {
                // suppression de la cellule du tableau:
                unset($this->_tblTreeNodes[$key]);
            }
        } // end foreach
    }

    public function getChild($indice) {
        return $this->_tblTreeNodes[$indice];
    }

    // On parcour les enfants pour cumuler le nombre de places.
    // Cette méthode est récursive si plusieurs Secteurs sont
    // imbriqués.
    public function getNbrPlaces() {
        $Compteur = 0;
        foreach (array_keys($this->_tblTreeNodes) as $key) {
            $Compteur = $Compteur + $this->_tblTreeNodes[$key]->getNbrPlaces();
        } // end foreach
        return $Compteur;
    }

    // on redéfini la méthode pour avoir un affichage personnalisé:
    public function __toString() {
        return "*** " . $this->_name . ": " . $this->_description . " " .
                $this->getNbrPlaces() . " places" . '</br>';
    }

}

?>
```

Le Secteur gère ses enfants dans un simple tableau.

Voila comment le Secteur fourni son itérateur.

Ces méthodes, bien qu'utiles, ne seront pas utilisées dans notre exemple.

Voila comment le Secteur cumule le nombre de places de ses enfants.

La classe Room_B:

```php
<?php

class Room_B extends TreeNode_B {

    private $_nbrPlaces;

    public function __construct($name = "", $description = "", $nbrPlaces = 0) {
        parent::__construct($name, $description);
        $this->_nbrPlaces = $nbrPlaces;
    }

    // on redéfini la méthode pour avoir un affichage personnalisé:
    public function __toString() {
        return "   " . "* " . $this->_name . ": " .
                $this->_description . " " . $this->_nbrPlaces .
                " places" . '</br>';
    }

    // on redéfini la méthode:
    public function print_propagation() {
        echo $this->__toString();
    }

    public function getNbrPlaces() {
        return $this->_nbrPlaces;
    }

    public function getIterator() {
        return NULL;
    }

}

?>
```

La salle se construit avec un nom, une description, et un nombre de places.

Nous reviendrons sur cette méthode ultérieurement.

Le fait de renvoyer NULL, permettra de distinguer les salles des secteurs.

L'interface Iterator_B: il s'agit de la même interface que dans le pattern Iterator.

```php
<?php

interface Iterator_B {

    public function hasNext();

    public function next();
}

?>
```

La classe SplStack: voir L'URL suivante:

http://php.net/manual/fr/class.splstack.php

La classe SimpleIterator_B: classe quasiment identique au pattern Iterator.

```php
<?php

class SimpleIterator_B implements Iterator_B {

    private $_curentPosition = 0; // indice du tableau. Démarre à zéro.
    private $_mySecteur;

    public function __construct($secteur) {
        $this->_mySecteur = $secteur;
    }

    public function hasNext() {
        if ($this->_curentPosition >= count($this->_mySecteur->_tblTreeNodes)) {
            return false;
        } else {
            return true;
        } // end if
    }

    public function next() {
        $treeNode = $this->_mySecteur->_tblTreeNodes[$this->_curentPosition];
        $this->_curentPosition++; // on incrémente
        return $treeNode;
    }

}

?>
```

Le SimpleIterator se construit en lui passant le Secteur sur lequel il doit itérer.

Le SimpleIterator accède directement au tableau des enfants du Secteur.

La classe CompositeIterator_B:

```php
<?php

class CompositeIterator_B implements Iterator_B {

    private $_myStack;

    public function __construct($simpleIterator) {
        $this->_myStack = new SplStack();
        $this->_myStack->push($simpleIterator);
        echo "Je suis CompositeIterator_B" . "<br>";
    }

    public function hasNext() {

        if ($this->_myStack->isEmpty()) {
            echo "La pile est vide.", "<br>";
            return FALSE;
        } else {
            echo $this->_myStack->count() .
                    " élément(s) dans la pile." . "<br>";
            if ($this->_myStack->top()->hasNext() == FALSE) {
                echo "Terminé pour ce simpleItérateur.", "<br>";
                // on dépile l'element du haut:
                $this->_myStack->pop();
                // la recursivité est ici:
                return $this->hasNext();
            } else {
                return TRUE;
            }
        } // if
    }

    public function next() {
        if ($this->hasNext() == TRUE) {

            $myTreeNode = $this->_myStack->top()->next();

            // on regarde si on a affaire à une feuille ou pas:
            if (!$myTreeNode->getIterator() == NULL) {
                // c'est un secteur, donc
                // on empile le nouveau SimpleItérateur:
                $this->_myStack->push($myTreeNode->getIterator());
            }
            return $myTreeNode;
        } else {
            return null;
        } // if
    }

}

?>
```

Un seul CompositeIterator sera instancié pour le parcours de l'arbre. On lui passe le SimpleIterator de la racine.

_myStack ne contient que des références vers des SimpleIterators. Donc ici on appel la méthode next() sur le SimpleIterator qui se trouve sur le dessus de la pile.

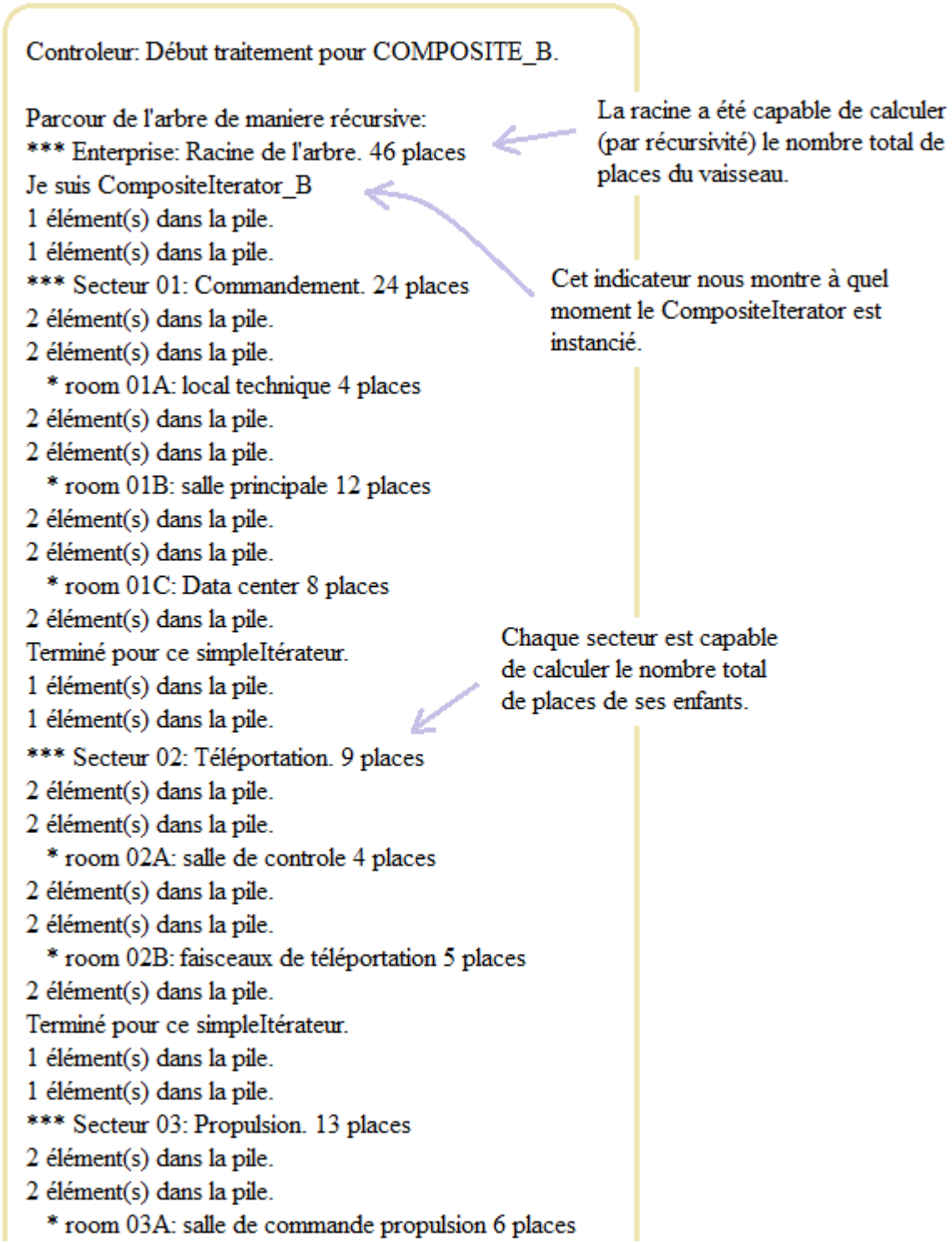
Le code de CompositeIterator_B est le cœur du pattern Composite. C'est lui qui gère la complexité du parcours récursif.

Voyons maintenant le code qui va utiliser l'ensemble du montage. Nous utilisons comme d'habitude index.php:

```php
<?php

//************************************************************
// COMPOSITE_B pattern controler
//************************************************************

require_once 'includePaths.php';
$newline = "<br>";
echo 'Controleur: Début traitement pour COMPOSITE_B.', $newline;

// Instanciations des secteurs:
$racine = new Sector_B("Enterprise", "Racine de l'arbre.");
$secteur_01 = new Sector_B("Secteur 01", "Commandement.");
$secteur_02 = new Sector_B("Secteur 02", "Téléportation.");
$secteur_03 = new Sector_B("Secteur 03", "Propulsion.");
$secteur_04 = new Sector_B("Secteur 04", "Moteurs.");

$racine->add($secteur_01);
$racine->add($secteur_02);
$racine->add($secteur_03);

// Instanciations des salles:
$secteur_01->add(new Room_B("room 01A", "local technique", 4));
$secteur_01->add(new Room_B("room 01B", "salle principale", 12));
$secteur_01->add(new Room_B("room 01C", "Data center", 8));

$secteur_02->add(new Room_B("room 02A", "salle de controle", 4));
$secteur_02->add(new Room_B("room 02B", "faisceaux de téléportation", 5));

$secteur_03->add(new Room_B("room 03A", "salle de commande propulsion", 6));
$secteur_03->add($secteur_04);
$secteur_03->add(new Room_B("room 03B", "salle de test", 6));

$secteur_04->add(new Room_B("room 04A", "compartiment cristal de Lithium", 0));
$secteur_04->add(new Room_B("room 04B", "Salle de décontamination", 1));

//*********************************
// Parcour de l'arbre en récursif:
//*********************************

echo '</br>', "Parcour de l'arbre de maniere récursive:", $newline;
echo $racine; // appelle la méthode __toString().

$myCompositeIterator = new CompositeIterator_B($racine->getIterator());
while ($myCompositeIterator->hasNext()) {
    $myNode = $myCompositeIterator->next();
    echo $myNode; // appelle la méthode __toString().
} // end while

echo '--------------', $newline;
echo 'Controleur: Fin traitement.', $newline;

?>
```

Le CompositeIterator s'utilise de manière très simple.

Et voici le résultat de l'exécution:



```
2 élément(s) dans la pile.
2 élément(s) dans la pile.
*** Secteur 04: Moteurs. 1 places
3 élément(s) dans la pile.
3 élément(s) dans la pile.
  * room 04A: compartiment cristal de Lithium 0 places
3 élément(s) dans la pile.
3 élément(s) dans la pile.
  * room 04B: Salle de décontamination 1 places
3 élément(s) dans la pile.
Terminé pour ce simpleItérateur.
2 élément(s) dans la pile.
2 élément(s) dans la pile.
  * room 03B: salle de test 6 places
2 élément(s) dans la pile.
Terminé pour ce simpleItérateur.
1 élément(s) dans la pile.
Terminé pour ce simpleItérateur.
La pile est vide.
--------------
Controleur: Fin traitement.
```

Pour bien comprendre le fonctionnement du code de CompositeIterator, le mieux est de l'exécuter "à la main":

- Dessinez un arbre sur une feuille de papier, avec des secteurs et des salles, ou prenez l'arbre donné en exemple au début du pattern Composite.
- Vérifiez qu'il y ait au moins un secteur qui contienne un sous-secteur, car c'est un cas de test important.
- Réservez un coin de la feuille pour représenter la pile. Son contenu évoluera pendant le parcours de l'arbre.
- Suivez le code des différentes classes comme si vous étiez l'ordinateur.
- Armez-vous de patience car cette partie est particulière ardue.

16.4 Peut-on aller plus loin ?

Comme on le voit dans notre contrôleur index.php, l'utilisation de CompositeIterator est très simple.

Puisque que chaque nœud est réellement récupéré par le contrôleur, nous pouvons imaginer toutes sortes de traitement sur eux.

Si nous voulons par exemple afficher la liste des salles qui ont 5 places ou plus, il suffit d'ajouter un test dans la boucle de parcours de l'arbre, dans index.php:

```
//*********************************************************
// Liste des salles ayant plus de 5 places (en récursif):
//*********************************************************

echo '</br>', "Liste des salles ayant plus de 5 places:", $newline;
echo $racine;  // appelle la méthode __toString().

$myCompositeIterator = new CompositeIterator_B($racine->getIterator());
while ($myCompositeIterator->hasNext()) {
    $myNode = $myCompositeIterator->next();
    if ($myNode->getIterator() == NULL) {  // si c'est une salle.
        if ($myNode->getNbrPlaces() >= 5) {
            echo $myNode;  // appelle la méthode __toString().
        }
    }
} // end while
```

Nous avons terminé l'étude de la variante B du pattern Composite. Avant de passer à la variante A (un peu plus compliquée), examinons un type de parcours "automatique" de l'arbre qui demande très peu de code et qui peut, dans certains cas, être suffisant.

16.5 Parcours par propagation automatique

Dans ce type de parcours, nous appelons une méthode sur le nœud racine, et nous faisons en sorte que cet appel soit propagé à tous les nœuds de l'arbre.

C'est une forme de récursivité "automatique" qui se propage en s'appuyant sur les liens parent-enfants de l'arbre.

Les avantages:

- Il y a très peu de code à écrire, et il est de plus très simple.
- On n'utilise pas du tout les itérateurs.

Les inconvénients:

- Le contrôleur qui lance ce traitement (pour nous index.php), ne récupère pas chaque nœud

de l'arbre comme dans la variante B vue précédemment, ou la variante A qui sera présentée plus loin. Comme les nœuds ne sont pas récupérés, aucun traitement ni filtrage n'est possible.

- Il faut ajouter une méthode dans les classes Sector_B et Room_B.

Voyons comment fonctionne ce type de parcours.

La méthode print_propagation() a été ajoutée aux classes TreeNode_B et Room_B:

Dans TreeNode_B:

```php
public function print_propagation() {
    echo $this->__toString();
    // Il faut aussi invoquer la méthode print_propagation() des enfants:
    foreach ($this->_tblTreeNodes as $key => $component) {
        $component->print_propagation();
    }
}
```

Dans Room_B:

```php
// on redéfini la méthode:
public function print_propagation() {
    echo $this->__toString();
}
```

C'est ici que se produit la récursivité "automatique", lorsque l'enfant est un secteur.

Cette méthode, définie dans TreeNode_B, est héritée par Sector_B.

...puis et redéfinie dans Room_B.

Pour exécuter ce parcours par propagation, il suffit d'ajouter ce code dans notre contrôleur index.php:

```php
//*********************************************************
// Liste des salles par propagation d'un appel de méthode:
//*********************************************************
echo '</br>', "Parcour de l'arbre par propagation d'un appel de méthode:", '</br>';
$racine->print_propagation();
```

Et voilà ce qu'on obtient à l'exécution:

Passons maintenant à la variante A du pattern Composite.

16.6 La variante A

Dans la variante A, les mêmes classes sont reprises et sont suffixées avec _A.

Deux classes sont modifiées:

- CompositeIterator_A: méthodes hasNext() et next().
- Sector_A: méthode getIterator().

Le parcours de l'arbre n'utilise plus de pile. A la place, plusieurs CompositeIterator sont instanciés et chaînés, au fur et à mesure de la progression dans l'arbre.

La modification de Sector_A concerne getIterator(). Dans la variante B, cette méthode renvoyait un nouveau SimpleIterator. Dans la variante A, elle doit renvoyer un CompositeIterator_A qui encapsule lui-même un SimpleIterator.

Nous y reviendrons lors de l'examen du code. Avant cela, examinons comment fonctionne cette variante A.

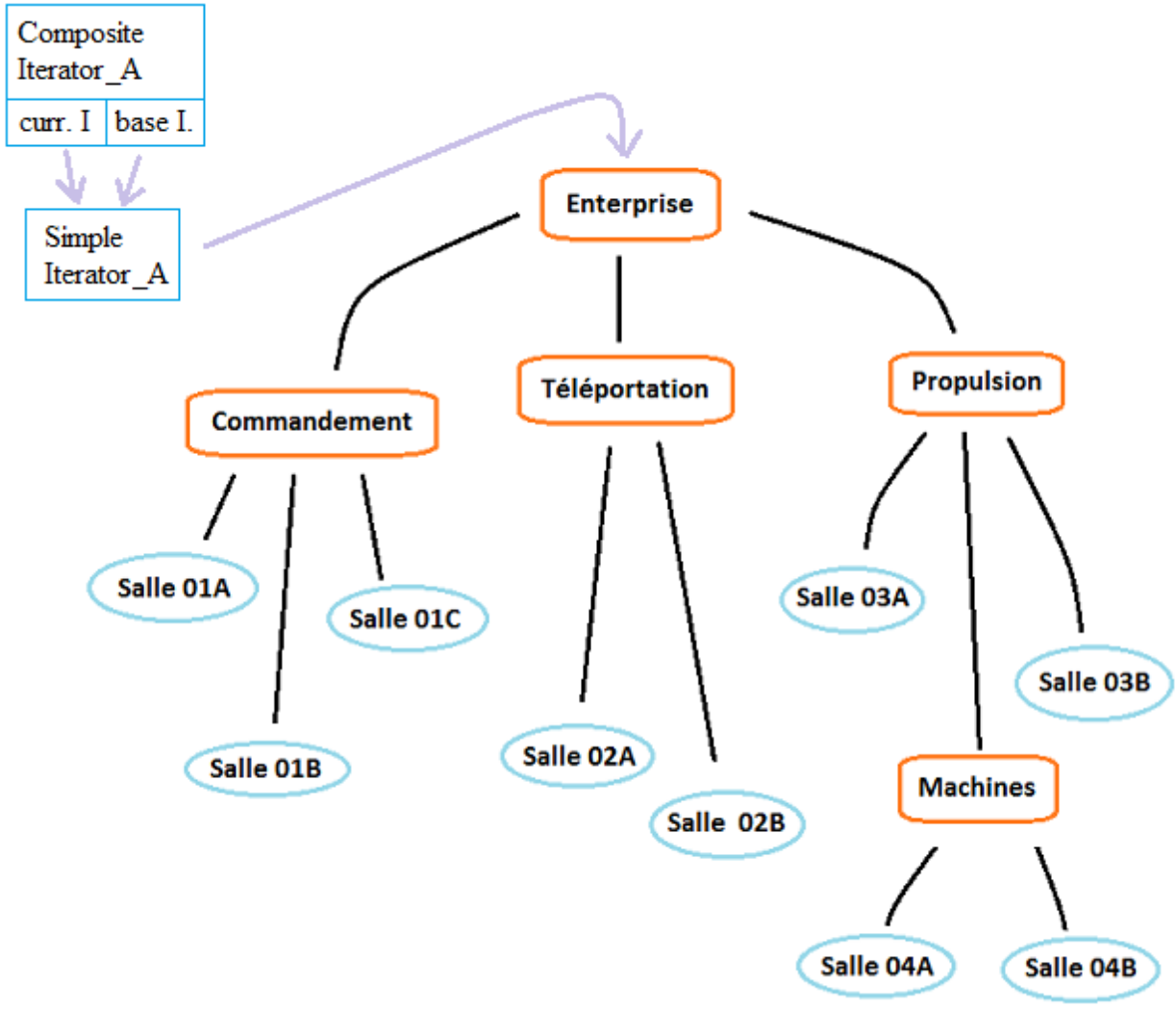
La figure ci-dessous représente la situation après l'instanciation du premier CompositeIterator_A pointant sur la racine.

Un CompositeIterator_A est toujours accompagné d'un SimpleIterator_A, puisqu'ils sont instanciés en même temps par Sector_A.

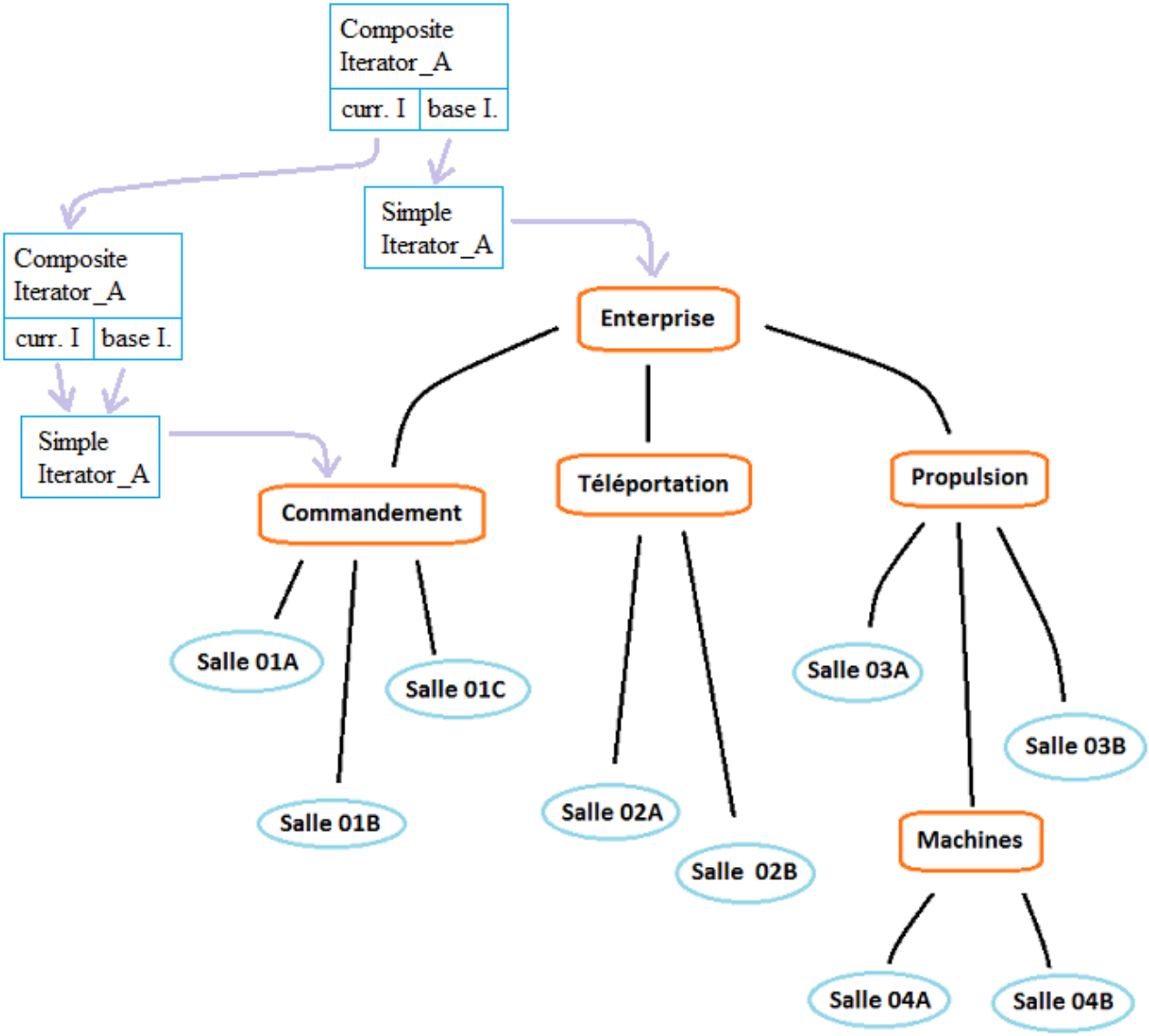
"base I." représente la variable privée $_baseIterator.

"cur. I." représente la variable privée $_currentIterator.

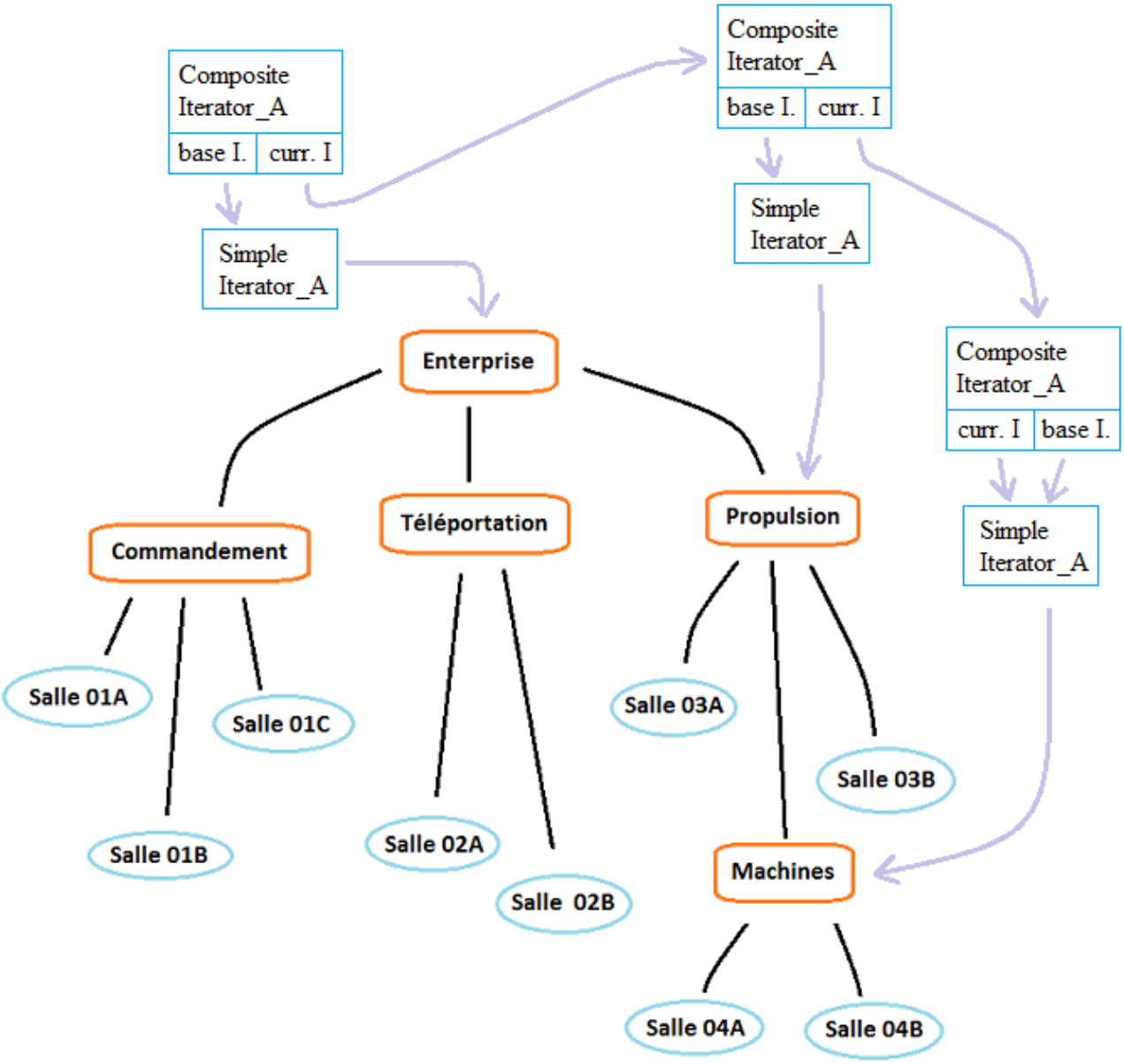
Ces deux variables sont utilisées par CompositeIterator_A. Nous y reviendrons lors de l'étude du code.



La figure ci-dessous représente la situation après l'instanciation du second CompositeIterator_A pointant sur le nœud "Commandement".



La figure ci-dessous représente la situation vers la fin du parcours de l'arbre, lorsque le dernier CompositeIterator_A pointe sur le nœud "Machine".



16.7 Codage de la variante A

L'interface Iterator_A: pas de changements.

```php
<?php

interface Iterator_A {

    public function hasNext();

    public function next();
}

?>
```

La classe TreeNode_A: pas de changements:

```php
<?php

abstract class TreeNode_A {
    const DEFAULT_ERROR_MESSAGE = "Methode non supportée par cette classe.";

    protected $_name;
    protected $_description;

    public function __construct($name = "", $description = "") {
        $this->_description = $description;
        $this->_name = $name;
    }

    public abstract function getNbrPlaces();

    public function getDescription() {
        return $this->_description;
    }

    public function getName() {
        return $this->_name;
    }

    public function add() {
        // Comportement par défaut:
        throw new Exception(TreeNode_A::DEFAULT_ERROR_MESSAGE);
    }

    public function getIterator() {
        // Comportement par défaut:
        throw new Exception(TreeNode_A::DEFAULT_ERROR_MESSAGE);
    }

    public function remove() {
        // Comportement par défaut:
        throw new Exception(TreeNode_A::DEFAULT_ERROR_MESSAGE);
    }

    public function getChild() {
        // Comportement par défaut:
        throw new Exception(TreeNode_A::DEFAULT_ERROR_MESSAGE);
    }
```

```
    public function print_propagation() {
        $this->__toString();

        // Il faut aussi invoquer la méthode print_propagation() des enfants:
        foreach ($this->_tblTreeNodes as $key => $component) {
            $component->print_propagation();
        }
    }

    public function __toString() {
        echo $this->_name, ": ", $this->_description, '</br>';
    }

}

?>
```

La classe Room_B: pas de changements:

```php
<?php

class Room_A extends TreeNode_A {

    private $_nbrPlaces;

    public function __construct($name = "", $description = "", $nbrPlaces = 0) {
        parent::__construct($name, $description);
        $this->_nbrPlaces = $nbrPlaces;
    }

    // on redéfini la méthode pour avoir un affichage personnalisé:
    public function __toString() {
        return "   " . "* " . $this->_name . ": " .
                $this->_description . " " . $this->_nbrPlaces .
                " places" . '</br>';
    }

    // on redéfini la méthode:
    public function print_propagation() {
        echo $this->__toString();
    }

    public function getNbrPlaces() {
        return $this->_nbrPlaces;
    }

    public function getIterator() {
        return NULL;
    }

}

?>
```

La classe Sector_A. La méthode getIterator() est modifiée.

```php
<?php

class Sector_A extends TreeNode_A {

    public $_tblTreeNodes = array();

    public function add($treeNode) {
        $this->_tblTreeNodes[] = $treeNode;
    }

    public function getIterator() {
        $returnedIterator = new CompositeIterator_A(new SimpleIterator_A($this));
        return $returnedIterator;
    }

    public function remove($treeNode) {
        foreach (array_keys($this->_tblTreeNodes) as $key) {
            if ($this->_tblTreeNodes[$key] === $treeNode) {
                // suppression de la cellule du tableau:
                unset($this->_tblTreeNodes[$key]);
            }
        } // end foreach
    }

    public function getChild($indice) {
        return $this->_tblTreeNodes[$indice];
    }

    // On parcour les enfants pour cumuler le nombre de places.
    // Cette méthode est récursive si plusieurs Secteurs sont
    // imbriqués.
    public function getNbrPlaces() {
        $Compteur = 0;
        foreach (array_keys($this->_tblTreeNodes) as $key) {
            $Compteur = $Compteur + $this->_tblTreeNodes[$key]->getNbrPlaces();
        } // end foreach
        return $Compteur;
    }

    // on redéfini la méthode pour avoir un affichage personnalisé:
    public function __toString() {
        return "*** " . $this->_name . ": " . $this->_description . " " .
                $this->getNbrPlaces() . " places" . '</br>';
    }

}

?>
```

Cette méthode renvoi maintenant un CompositeIterator qui encapsule un SimpleIterator.

La classe SimpleIterator_A: pas de changements:

```php
<?php

class SimpleIterator_A implements Iterator_A {

    private $_curentPosition = 0; // indice du tableau. Démarre à zéro.
    private $_mySecteur;

    public function __construct($secteur) {
        $this->_mySecteur = $secteur;
    }

    public function hasNext() {
        if ($this->_curentPosition >= count($this->_mySecteur->_tblTreeNodes)) {
            return false;
        } else {
            return true;
        } // end if
    }

    public function next() {
        $treeNode = $this->_mySecteur->_tblTreeNodes[$this->_curentPosition];
        $this->_curentPosition++; // on incrémente
        return $treeNode;
    }

    public function getSecteurName() {
        return $this->_mySecteur->getName();
    }

}

?>
```

La classe CompositeIterator_A: plusieurs changements:

```php
<?php

class CompositeIterator_A implements Iterator_A {

    private $_currentIterator;
    private $_baseIterator;

    public function __construct($simpleIterator) {
        $this->_baseIterator = $simpleIterator;
        $this->_currentIterator = $simpleIterator;
        echo "Je suis CompositeIterator_A pour " .
            $this->getSecteurName() . "<br>";
    }

    public function hasNext() {
        // Cette méthode est récursive si plusieurs
        // CompositeIterator sont liés.
        if ($this->_currentIterator->hasNext()) {
            return true;  // on quitte la méthode.
        }

        if ($this->_currentIterator == $this->_baseIterator) {
            echo "terminé pour ce SimpleIterator: " .
                $this->_currentIterator->getSecteurName() . "<br>";
            return false; // on quitte la méthode.
        }
        // Si on arrive ici, c'est que l'exploration du sous-secteur
        // est terminée. On continue donc l'exploration du secteur courant:
        $this->_currentIterator = $this->_baseIterator;
        return $this->_currentIterator->hasNext();
    }

    public function next() {
        // Cette méthode est récursive si plusieurs
        // CompositeIterator sont liés.
        if ($this->hasNext()) {
            echo "hasNext(): secteur " . $this->getSecteurName() .
                ": il en reste !" . "<br>";
            $myTreeNode = $this->_currentIterator->next();
            if ($myTreeNode instanceof Sector_A) {   // si c'est un secteur.
                if ($this->_currentIterator == $this->_baseIterator) {
                    // la récursivité est ici:
                    $this->_currentIterator = $myTreeNode->getIterator();
                }
            }
            return $myTreeNode;
        } else {
            return null;
        } // end if
    }
```

Quelques affichages ont été insérés dans le code pour suivre la progression du parcour de l'arbre.

currentIterator référence soit un autre CompositeIterator, soit un SimpleIterator si on arrive en bout de chaine.

getIterator() va créer un nouveau couple CompositeIterator - SimpleIterator.

```
    public function getSecteurName() {
        return $this->_currentIterator->getSecteurName();
    }

}

?>
```

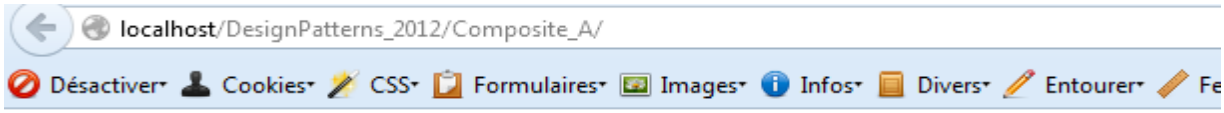
Là encore pour bien comprendre le processus de parcours de l'arbre, le mieux est de l'exécuter à la main sur une feuille de papier, avec là encore une bonne dose de patience.

Voyons maintenant le code qui va utiliser l'ensemble du montage. Nous utilisons comme d'habitude index.php:

```php
<?php

//************************************************************
// COMPOSITE_A pattern controler
//************************************************************

require_once 'includePaths.php';
$newline = "</br>";

echo 'Controleur: Début traitement pour COMPOSITE_A.', $newline;

// Instanciations des secteurs:
$racine = new Sector_A("Enterprise", "Racine de l'arbre.");
$secteur_01 = new Sector_A("Secteur 01", "Commandement.");
$secteur_02 = new Sector_A("Secteur 02", "Téléportation.");
$secteur_03 = new Sector_A("Secteur 03", "Propulsion.");
$secteur_04 = new Sector_A("Secteur 04", "Moteurs.");

$racine->add($secteur_01);
$racine->add($secteur_02);
$racine->add($secteur_03);

// Instanciations des salles:
$secteur_01->add(new Room_A("room 01A", "local technique", 4));
$secteur_01->add(new Room_A("room 01B", "salle principale", 12));
$secteur_01->add(new Room_A("room 01C", "Data center", 8));

$secteur_02->add(new Room_A("room 02A", "salle de controle", 4));
$secteur_02->add(new Room_A("room 02B", "faisceaux de téléportation", 5));

$secteur_03->add(new Room_A("room 03A", "salle de commande propulsion", 6));
$secteur_03->add($secteur_04);
$secteur_03->add(new Room_A("room 03B", "salle de test", 6));

$secteur_04->add(new Room_A("room 04A", "compartiment cristal de Lithium", 0));
$secteur_04->add(new Room_A("room 04B", "Salle de décontamination", 1));

//*********************************
// Parcour de l'arbre en récursif:
//*********************************

echo '</br>', "Parcour de l'arbre de maniere récursive:", $newline;
echo $racine; // appelle la méthode __toString().

$myCompositeIterator = new CompositeIterator_A(new SimpleIterator_A($racine));
while ($myCompositeIterator->hasNext()) {
    $myNode = $myCompositeIterator->next();
    echo $myNode; // appelle la méthode __toString().
} // end while

echo '--------------', $newline;
echo 'Controleur: Fin traitement.', $newline;
?>
```

Parcourir l'arbre est très simple.

Et voici le résultat de l'exécution:




```
Controleur: Début traitement pour COMPOSITE_A.

Parcour de l'arbre de maniere récursive:
*** Enterprise: Racine de l'arbre. 46 places
Je suis CompositeIterator_A pour Enterprise
hasNext(): secteur Enterprise: il en reste !
Je suis CompositeIterator_A pour Secteur 01
*** Secteur 01: Commandement. 24 places
hasNext(): secteur Secteur 01: il en reste !
hasNext(): secteur Secteur 01: il en reste !
  * room 01A: local technique 4 places
hasNext(): secteur Secteur 01: il en reste !
hasNext(): secteur Secteur 01: il en reste !
  * room 01B: salle principale 12 places
hasNext(): secteur Secteur 01: il en reste !
hasNext(): secteur Secteur 01: il en reste !
  * room 01C: Data center 8 places
terminé pour ce SimpleIterator: Secteur 01
hasNext(): secteur Enterprise: il en reste !
Je suis CompositeIterator_A pour Secteur 02
*** Secteur 02: Téléportation. 9 places
hasNext(): secteur Secteur 02: il en reste !
hasNext(): secteur Secteur 02: il en reste !
  * room 02A: salle de controle 4 places
hasNext(): secteur Secteur 02: il en reste !
hasNext(): secteur Secteur 02: il en reste !
  * room 02B: faisceaux de téléportation 5 places
terminé pour ce SimpleIterator: Secteur 02
hasNext(): secteur Enterprise: il en reste !
Je suis CompositeIterator_A pour Secteur 03
*** Secteur 03: Propulsion. 13 places
hasNext(): secteur Secteur 03: il en reste !
hasNext(): secteur Secteur 03: il en reste !
  * room 03A: salle de commande propulsion 6 places
hasNext(): secteur Secteur 03: il en reste !
hasNext(): secteur Secteur 03: il en reste !
Je suis CompositeIterator_A pour Secteur 04
*** Secteur 04: Moteurs. 1 places
hasNext(): secteur Secteur 04: il en reste !
hasNext(): secteur Secteur 04: il en reste !
hasNext(): secteur Secteur 04: il en reste !
  * room 04A: compartiment cristal de Lithium 0 places
hasNext(): secteur Secteur 04: il en reste !
```

Ici hasNext() s'affiche deux fois car 2 CompositeIterator sont liés.

Ici hasNext() s'affiche trois fois car 3 CompositeIterator sont liés.

```
hasNext(): secteur Secteur 04: il en reste !
hasNext(): secteur Secteur 04: il en reste !
  * room 04B: Salle de décontamination 1 places
terminé pour ce SimpleIterator: Secteur 04
hasNext(): secteur Secteur 03: il en reste !
hasNext(): secteur Secteur 03: il en reste !
  * room 03B: salle de test 6 places
terminé pour ce SimpleIterator: Secteur 03
--------------
Controleur: Fin traitement.
```

La room 03B est bien affichée après les salles du secteur 04 (Moteurs).

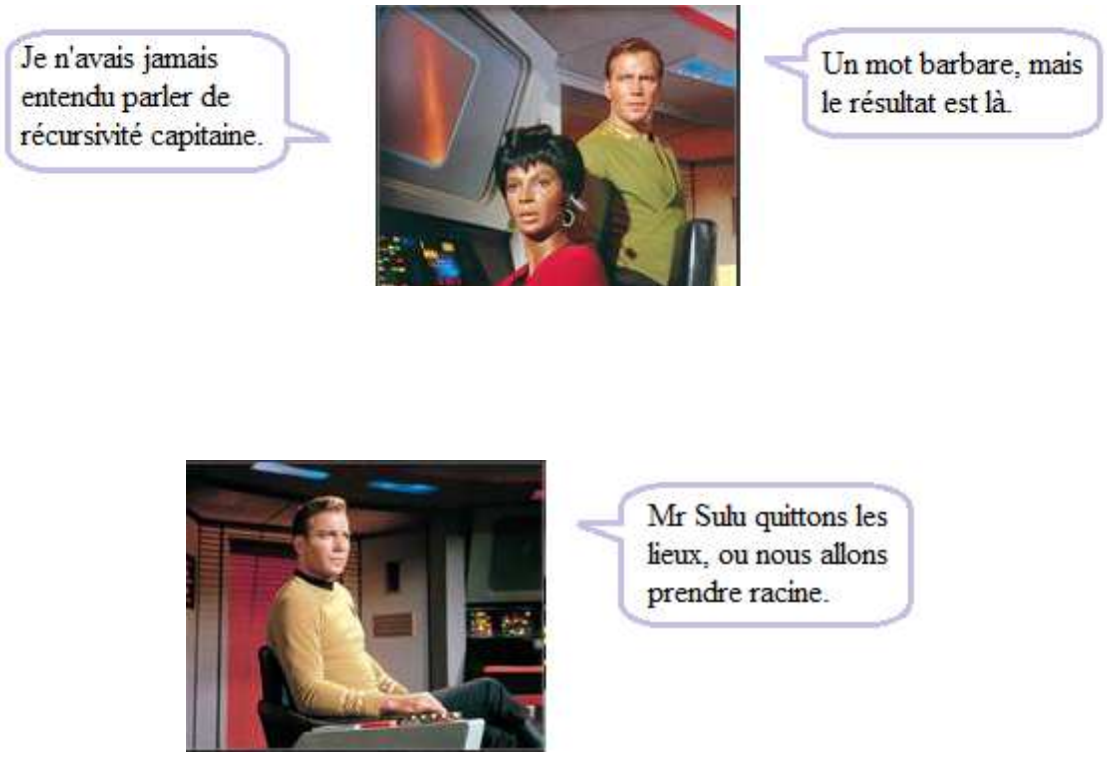
Nous disposons toujours de la possibilité d'effectuer des tests sur chaque nœud de l'arbre comme nous l'avons fait dans la variante B: afficher toutes les salles ayant 5 places ou plus.

Voici la boucle à ajouter à index.php:

```php
//*****************************************
// Liste des salles ayant plus de 5 places:
//*****************************************

echo '</br>', "Liste des salles ayant plus de 5 places:", '</br></br>';
echo $racine; // appelle la méthode __toString().
$myCompositeIterator = new CompositeIterator_A(new SimpleIterator_A($racine));
while ($myCompositeIterator->hasNext()) {
    $myNode = $myCompositeIterator->next();
    if ($myNode instanceof Room_A) {
        if ($myNode->getNbrPlaces() >= 5) {
            echo $myNode->__toString(), '</br>';
        }
    }
} // end while
```

Enfin nous disposons toujours du parcours par propagation d'un appel de méthode comme vu dans la variante B. Le code reste le même:

```php
//************************************************************
// Liste des salles par propagation d'un appel de méthode:
//************************************************************
echo '</br>', "Parcour de l'arbre par propagation d'un appel de méthode:", '</br>';
$racine->print_propagation();
```



17 Notions complémentaires

17.1 Couplage faible

Se dit d'une relation de dépendance entre 2 ou plusieurs classes, quand cette relation de dépendance est réduite à son minimum.

On recherche systématiquement à définir des couplages faibles entre les classes, et les design patterns vont tous dans ce sens. Les conceptions objets obtenues sont plus aptes à affronter les évolutions à venir.

17.2 Principe ouvert/fermé

Ce principe veut qu'une classe soit fermée à toute modification à l'intérieure de la classe, mais ouverte aux modifications par extension de la classe.

Le fait d'interdire la modification d'une classe, limite le risque d'introduction de nouvelles erreurs.

Mais en permettant d'étendre les fonctionnalités d'une classe par extension, on préserve l'évolutivité d'un modèle.

Les design patterns respectent ce principe, en particulier le pattern Decorator, sur lequel il est exclusivement basé.

17.3 La récursivité

En Informatique, un traitement récursif est un traitement qui contient un appel vers lui-même.

Le critère d'arrêt d'un traitement récursif est fondamental. Si la condition d'arrêt ne se produit pas, les appels récursifs vont s'enchaîner indéfiniment, ce qui peut conduire à un blocage complet de l'ordinateur, du fait de l'occupation de la totalité de la mémoire centrale.

17.3.1 La fonction récursive.

Il s'agit d'une simple fonction qui contient un appel vers elle-même. L'exemple classique est la fonction permettant de calculer la factorielle de x:

```php
function factorielle($x)
{
    if($x == 0)
        return 1;
    else
        return $x*factorielle($x-1);
}
```

Ce type de traitement s'appuie sur un mécanisme de pile qui sera géré automatiquement par le langage, et qui, à chaque nouvel appel de la fonction "factorielle", va empiler le contexte d'exécution de la fonction. Ce contexte ici, se résume à la valeur de x.

Lorsque le critère d'arrêt se produit, le langage va dépiler les contextes, et terminer l'exécution de chacun d'eux, c'est à dire multiplier sa propre valeur de x, à la valeur qui lui a été retournée par le contexte précédent.

De dépilement en dépilement, les valeurs de x vont se multiplier entre elles pour fournir au final la factorielle de x.

17.3.2 Variantes de récursivité:

Récursivité par propagation d'un message dans une structure arborescente.

Une méthode est appelée sur l'élément racine de l'arbre. Cette méthode appelle à son tour son homologue sur ses enfants, et ainsi de suite.

Il n'y a pas critère d'arrêt, le traitement s'arrête lorsque tous les nœuds de l'arbre ont été parcourus.

Ce sont les liens entre les nœuds de l'arbre, qui provoquent le comportement récursif.

Récursivité par instanciations récursives.

En programmation objet: l'appel récursif est effectué sur le même nom de méthode, mais sur un objet qui est instancié au moment de l'appel récursif. Les instanciations récursives cessent lorsque le critère d'arrêt se produit. L'empilement ici est matérialisé par les objets instanciés.

18 Annexes

18.1 Conventions de nommage

Les conventions de nommages suivantes ont été utilisées dans ce tutoriel:

Les classes: *UpperCamelCase*

Les interfaces : *UpperCamelCase*
Les fonctions et méthodes: *camelCase*
Les variables: *camelCase*
Les constantes: *ALL_CAPS*

18.2 Code source en PHP

L'ensemble du code PHP est fourni dans une archive à part, disponible sur smaltek.fr.

18.3 Outils utilisés

J'ai utilisé les outils "open source" suivants:

WAMP: *www.wampserver.com*

Wamp installe les éléments suivants:

- Un serveur Apache avec le module PHP.
- Un serveur de base de données Mysql. Ce serveur Mysql n'est pas utilisé dans le cadre de ce tutoriel.

NETBEANS: *http://netbeans.org/*

Netbeans est un IDE (Environnement de Développement Intégré). Il permet de gérer les codes sources sous forme de projets, pour différents langages dont PHP.

Eclipse (*www.eclipse.org*) est un autre IDE comparable, également open source.

Configuration du projet dans NetBeans:

- J'ai réuni tous les design patterns dans un unique projet Netbeans.
- Chaque design pattern se trouve dans un sous répertoire qui porte son nom.
- Dans chaque sous répertoire j'ai mis les deux fichiers suivants:
  - autoloader.php
  - includePaths.php

Le contenu de autoloader.php est le même dans tous les design patterns:

```php
<?php

function __autoload($class) {
    require_once $class . '.class.php';
}

?>
```

La fonction PHP __autoload(), va automatiser le chargement des classes.

Ici, toutes les classes et interfaces doivent être suffixées avec ".class.php" pour que cela fonctionne.

Contenu de includePaths.php:

```php
<?php

/**
 * Configuration des chemins d'acces internes.
 * Internal path configuration.
 *
* Ajoute les chemins necessaires, dans la variable include_path.
* Add required paths into include_path variable.
*/

$NewPath = $_SERVER['DOCUMENT_ROOT'] . 'DesignPatterns_2012/Composite_A/class';
set_include_path(get_include_path() . PATH_SEPARATOR . $NewPath);

$NewPath = $_SERVER['DOCUMENT_ROOT'] . 'DesignPatterns_2012/Composite_A/interfaces';
set_include_path(get_include_path() . PATH_SEPARATOR . $NewPath);

//echo $NewPath;
//echo get_include_path();

require_once 'autoloader.php';

?>
```

Ce script va ajouter dans les chemins de recherches du système d'exploitation, les répertoires dans lesquels se trouvent les classes et les interfaces du design pattern.

Les chemins indiqués dans ce script sont donc à modifier pour chaque pattern.

19 Conclusion

Ce tutoriel ne présente qu'une partie des 23 design patterns originaux du GOF (Gang Of Four) http://fr.wikipedia.org/wiki/Patron_de_conception

D'ailleurs cette liste n'est pas figée. De nouveaux patterns voient le jour. Souvent ces derniers n'étant que des variantes de patterns existants.

Un pattern peut en cacher un autre.

Un pattern n'est pas monolithique. Certains patterns s'associent pour former un nouveau pattern. Nous avons vu par exemple que Composite utilisait un Iterator.

Les patterns associés de cette manière sont des Compound Patterns.
http://serviceorientation.com/soaglossary/compound_design_pattern

Du pattern à tous prix.

Quelques pièges à éviter :

- Mettre du design pattern quand ce n'est pas justifié, ne peut qu'alourdir inutilement un projet. Cela équivaut à prendre l'avion pour se rendre à la boulangerie du quartier.
- Transformer sa problématique pour la rendre compatible avec un pattern : Ce n'est pas le cahier des charges qui s'adapte aux design patterns mais l'inverse.
- Parer un projet à toutes les évolutions futures possibles et imaginables : C'est une utopie qui peut considérablement alourdir le projet en code et en tests. Il conviendra plutôt de définir avec le client la « probabilité raisonnable » de telle évolution future et de préparer le projet en conséquence, éventuellement à l'aide de design patterns.

Fin du document