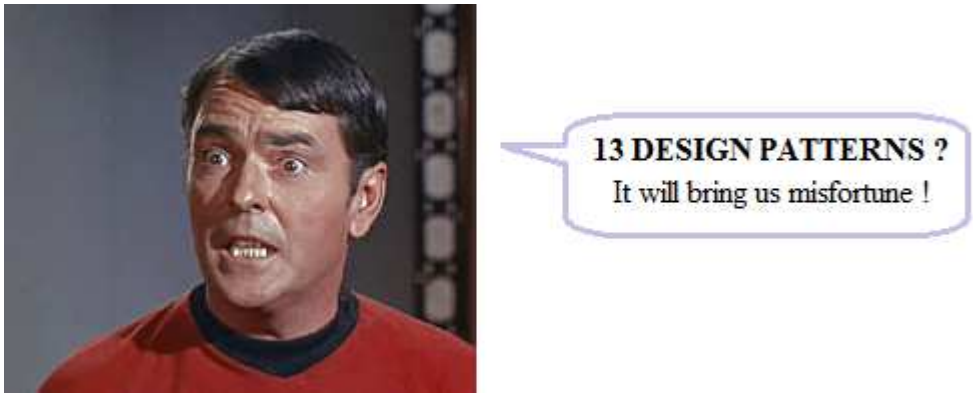
COSMIC DESIGN PATTERNS WITH PHP

A tutorial describing in PHP, 13 design patterns among the most used.



1 Preamble

This tutorial is for developers who want to discover or deepen design patterns, these programming models each addressing a particular problem.

(http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29)

Each design pattern is autonomous. It is not necessary to follow the order in which they appear in this tutorial.

However, Pseudo Factory and Abstract Factory FactoryMethod are three patterns forming a family.

Also Composite relies on Composite Iterator.

The PHP language is used. It's therefore necessary to have a small web environment and an IDE (Integrated Development Environment). See the chapter "Tools Used" in the appendix.
.

All PHP code is provided in a separate archive, available www.smaltek.fr.

Table des matières

2 STRATEGY PATTERN

The USS Enterprise uses different types of robots to perform certain tasks: exploration, intelligence, taking measurements, and even cleaning the corridors of the Enterprise.

Some walk, others run or fly.

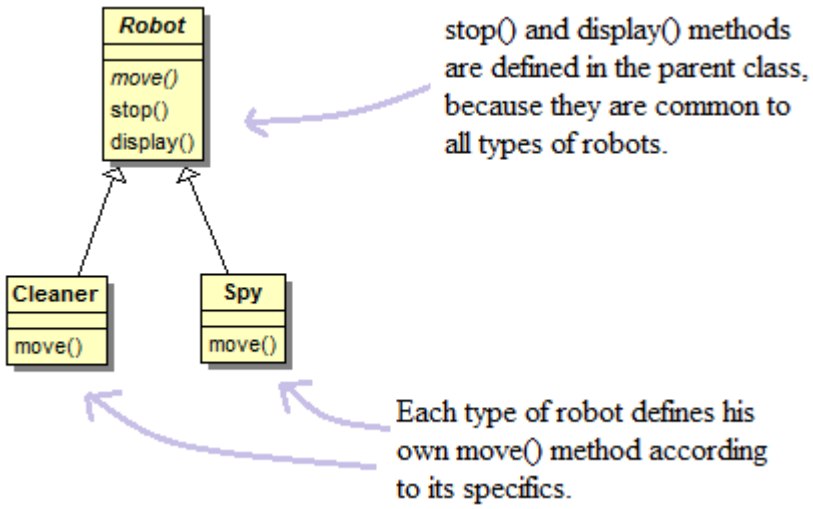
They are aware of their mission parameters including the route to perform. A program can follow the screen.

Some types of robots can now become invisible, as robots such SPY.

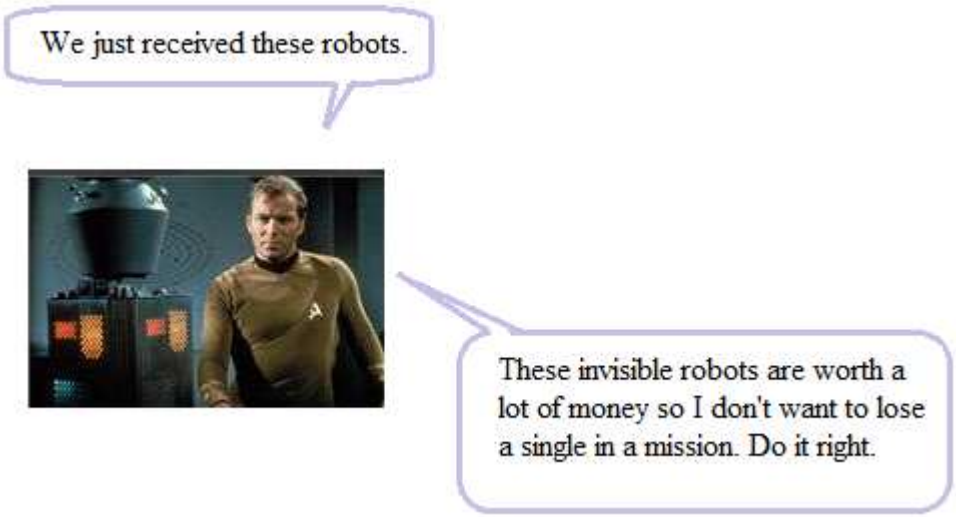
Captain Kirk says you care to change the program in order to use these new robots and their ability to become invisible.

The previous developer used object-oriented programming model with the following classes, and two implementations:

- Cleaner: cleaning robot. It moves with wheels.
- Spy: intelligence robot. It walks like a human.

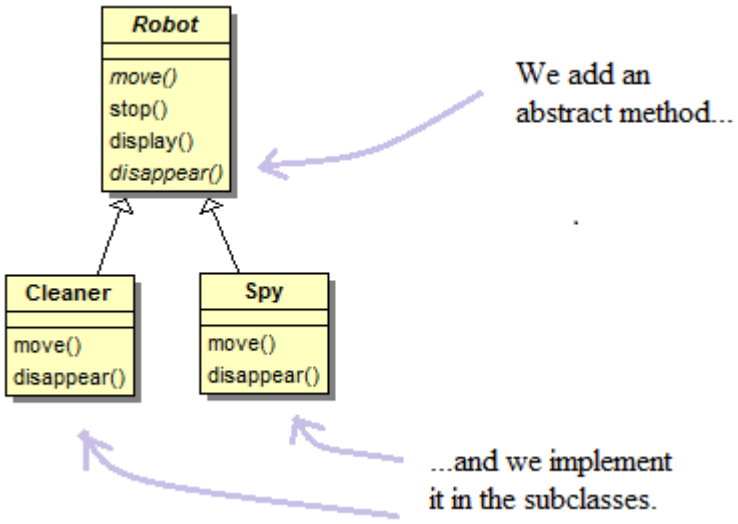


It only remains to add a Disappear() method in our model and voila. We can control the invisibility of robots involved.



2.1 First approach

Try to add disappear() method to the superclass.



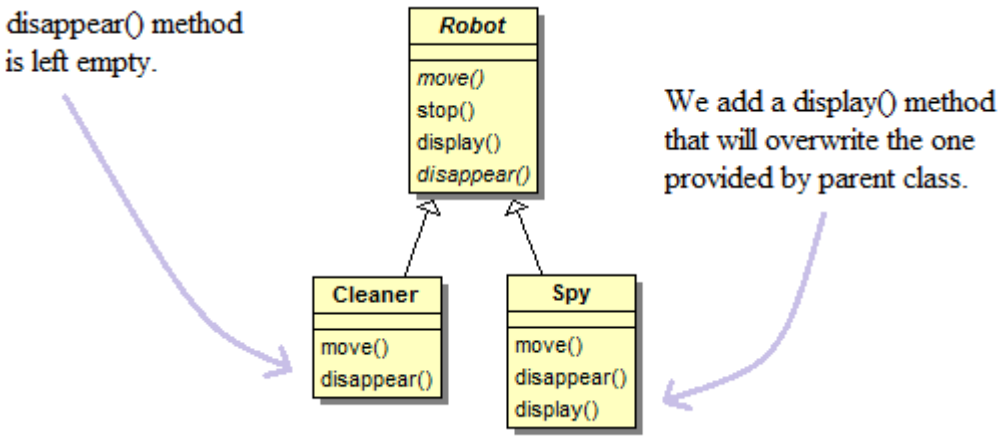
The job was not very difficult, but we quickly found some drawbacks:

1. The cleaner robot is not affected by invisibility. Yet we will be forced to add a disappear() method.
2. The display() method that is common to all types of robots, as implemented in the superclass, will have to be adapted for robots with invisibility. Indeed it will, and for them only, add a visual indicator on the screen for invisibility.

Keep on with this approach to see where it leads.

- On the downside No. 1, let's disappear() method empty in cleaner class. It's not very pretty, but it does not create any particular problem.
- On the downside No. 2, we can override the display() method (give a different implementation that provided by the superclass) in Spy class, and all the possible subclasses that require a particular view of the robot. Hmmm ... less pretty.

We would get this:



It did not look too bad, but ... it seems that something is missing. Robots measuring sensors (Sensor-type) have been forgotten. They are used to collect all kinds of information on a given site, but cannot move. You place them manually on a site, or place them by another robot.

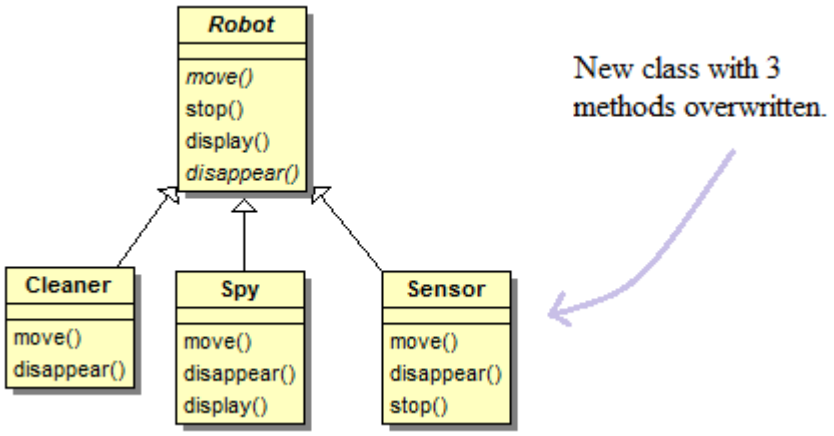
In short, the robot Sensor Type:

- is not affected by the move() and stop() methods.
- is not currently affected by the disappear() method.

No problem. We add a Sensor class in which:

- We implement the move() method leaving it empty.
- We implement the disappear() method leaving it empty.
- we overwrite stop() method leaving it empty.
- we keep the display() method from the superclass.

We get:



It is time to take evaluate our approach.

- In trying to use only the heritage, it is necessary to override certain methods in subclasses that require, in order to fit a specific behavior or to inhibit (empty implementation).
- The abstract methods as move() and disappear() must be implemented in subclasses. This leads eventually to code duplication. Imagine two types of robots that have the same way of moving. They have the same implementation of the move() method. Having to duplicate the code is often the sign of a bad approach.
- The behaviors of the various types of robots are defined by inheritance, so statically. It is impossible to change it dynamically at run-time.
- Here things are not too complicated because we have only three types of robots. But what would happen with 15 or 20 different types ? Wish good luck to the person responsible for the maintenance of such a design !

2.2 Deuxième approche

OK. I know what you think. Simply create an interface for each behavior that is not common to all robots. In this way, the subclasses can implement interfaces they just need.

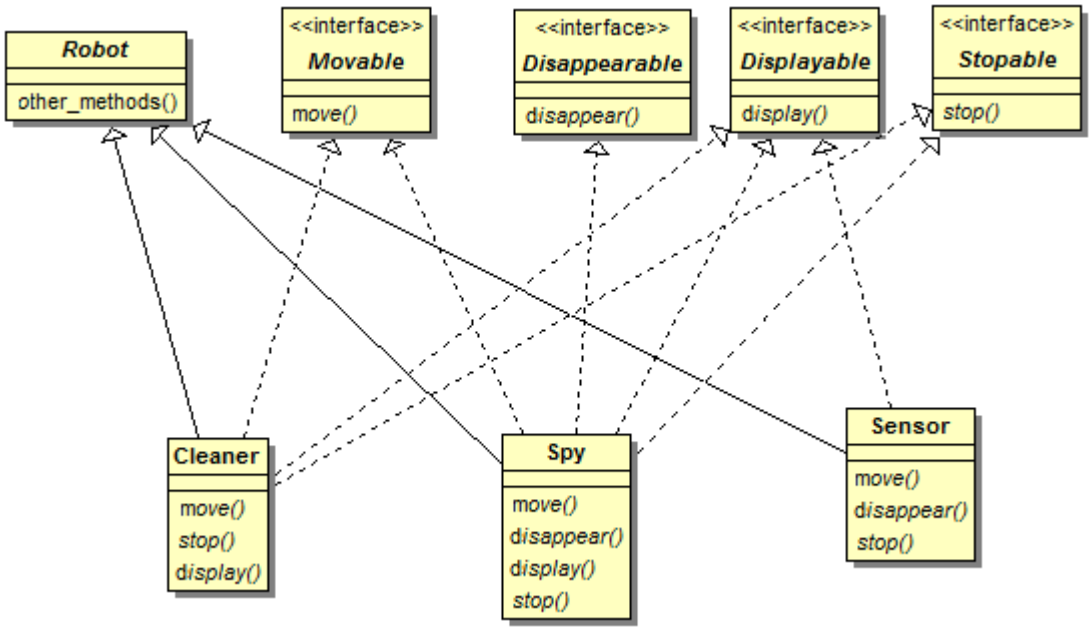
.

- Cleaner needs to be Movable, Displayable, Stopable.
- Spy needs to be Movable, Disappearable, Displayable, Stopable.
- Sensor needs to be Displayable.

Changing behaviors are now embedded by interfaces. The corresponding methods have been removed from the Robot class.

Subclasses implement the interfaces they need, and must define the concrete implementation of these methods.

We get:



Here we have four interfaces and three subclasses, so a theoretical maximum of 12 implementations of interfaces (dotted arrows). Luckily, only 8 of 12 are required.

.

Imagine this design with by example:

- 6 interfaces.
- 6 subclasses.

We would be faced with a theoretical maximum of 36 dotted arrows. Unmanageable is not it?

And Spock would not find it fascinating!

In our first approach, we relied solely on inheritance. Our second approach based on interfaces, sounded interesting, but it turns out it will quickly lead to a model difficult to maintain.

In addition, both approaches have disadvantages that we just want to avoid in object-oriented programming:

- Code duplication.
- Obligation to implement empty methods.
- Model whose complexity evolves exponentially with the number of classes or interfaces.
- Rather rigid model that will be hard to change, and is therefore difficult to maintain.
- The robots behaviors are defined in the design of classes, so it is impossible to dynamically change it at run time.

2.3 Heading Strategy pattern

Why do we have so much trouble to design a model of effective classes for our three types of robots ?

The problem is that we try to mix two hardly compatible concepts:

- On the one hand, the behaviors of robots which vary greatly from one type of robot to another. Consider for example the move() method, we know that robots dont move all in the same way. In addition, some robots dont move at all.
- On the other hand, the types of robots which are concepts that dont change or little.

Trying to directly involve behaviors that change with types of robots that dont change, we get a rigid and not evolutionary model. In the best case it will be fine only if there are only 2 or 3 types of robots and 2 or maximum 3 behaviors. And praying that we would never asked to add a fourth.

Strategy pattern is based on a strong separation between these two sets:

- The concepts that change or may change in the future (also known as variability).

- The concepts that dont change and are unlikely to change in the future (also known as commonality).

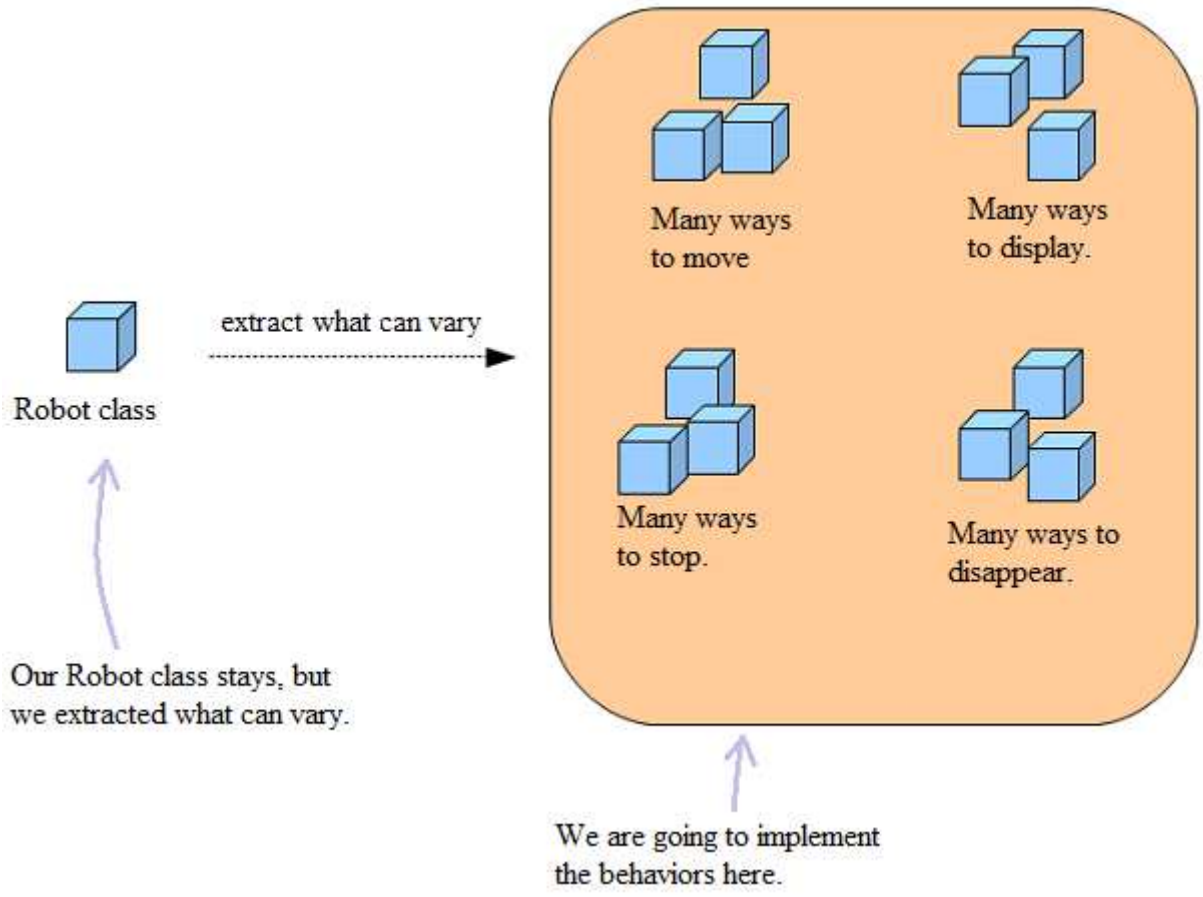
The first step is to form the two sets:

In our commonality, put concepts that dont have a changing character:

- Robots. Indeed the concept of robot is represented by a parent class Robot, and three subclasses Spy, Sensor and Cleaner. This design is unlikely to be changed. Adding additional classes as possible but that does not raise problem.

In our variability put the concepts that have a changing character, ie that must be implemented in a different way, a type of robot to another:

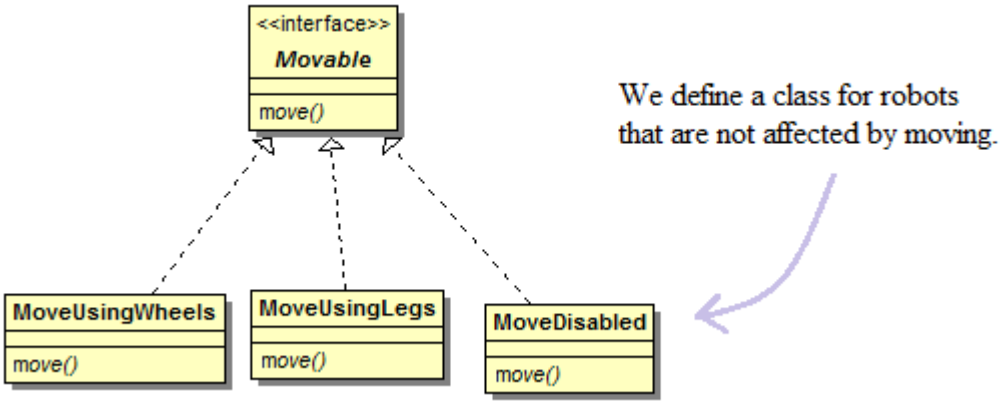
- Move: all robots do not move in the same way.
- Disappear: some robots are not affected.
- Display: The invisible robots will be displayed in a different way.
- Stop: some robots are not affected.



2.3.1 Implementing changing behaviors

The different Move behaviors are located in concrete classes that implement a new interface called Movable.

This interface contains a Move() method so it must be implemented by each class.

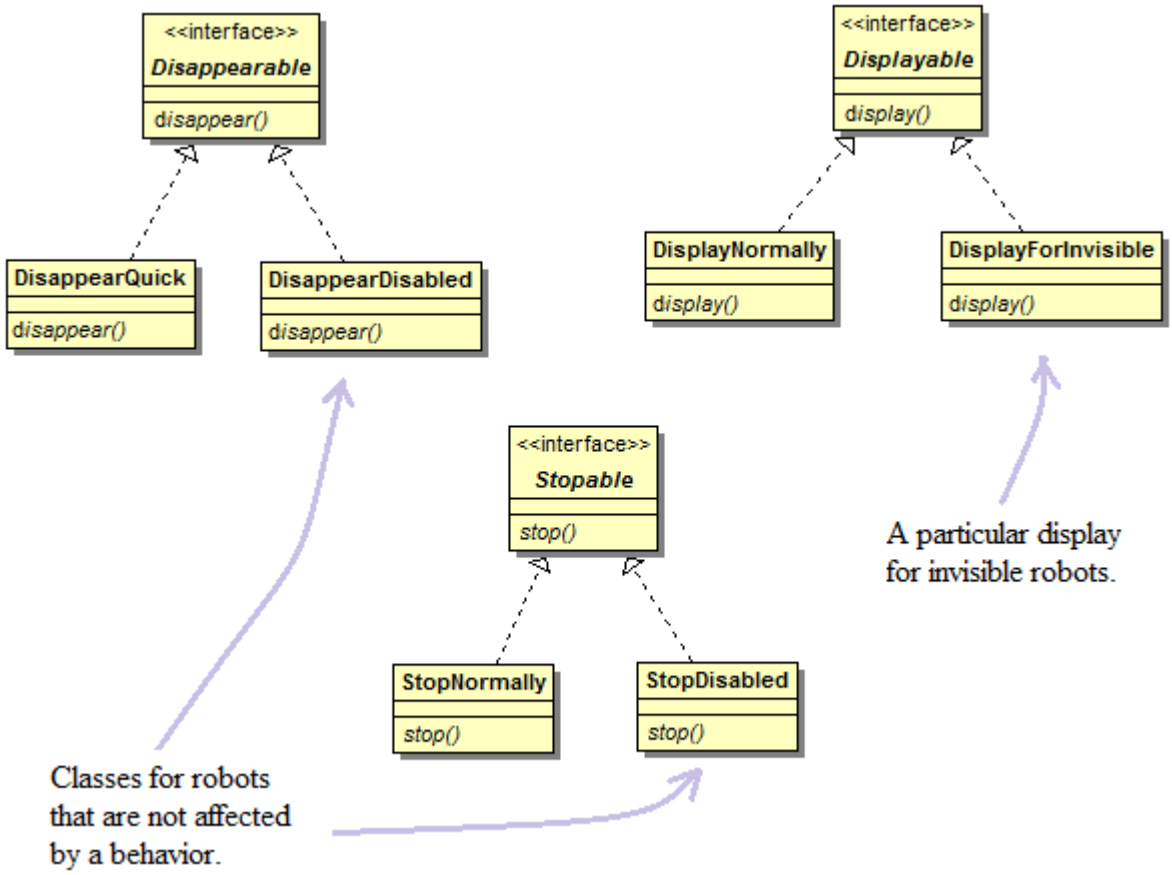


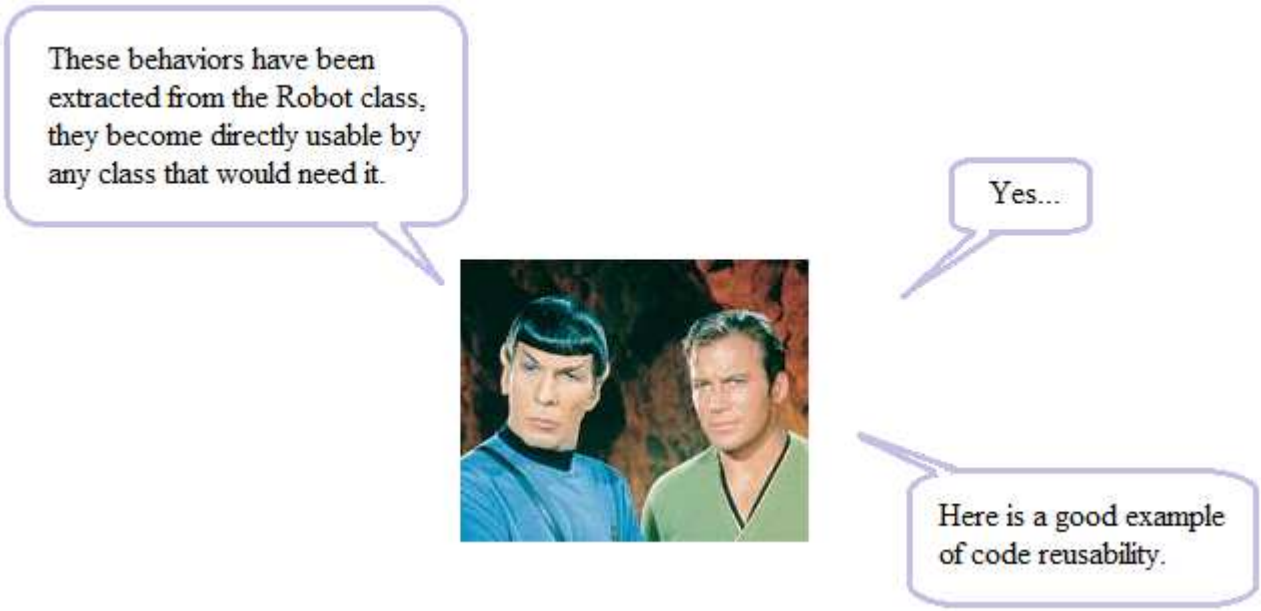
That's the way behaviors will be hidden to Robot class, which will only see Movable objects.

In other words, the robots will not know the implementation details of their move() method. They just know they have a move() method.

2.3.2 How to define the behavior of each robot

In our first attempts, the association between a behavior and a robot, was frozen in the class inheritance. This lack of flexibility was the cause of our problems.
Now that these two concepts are completely separate, it is at the instantiation of objects, ie at run time, that we assign each robot the way to move that suits him.
Do the same for other behaviors (Disappear, Display, Stop)

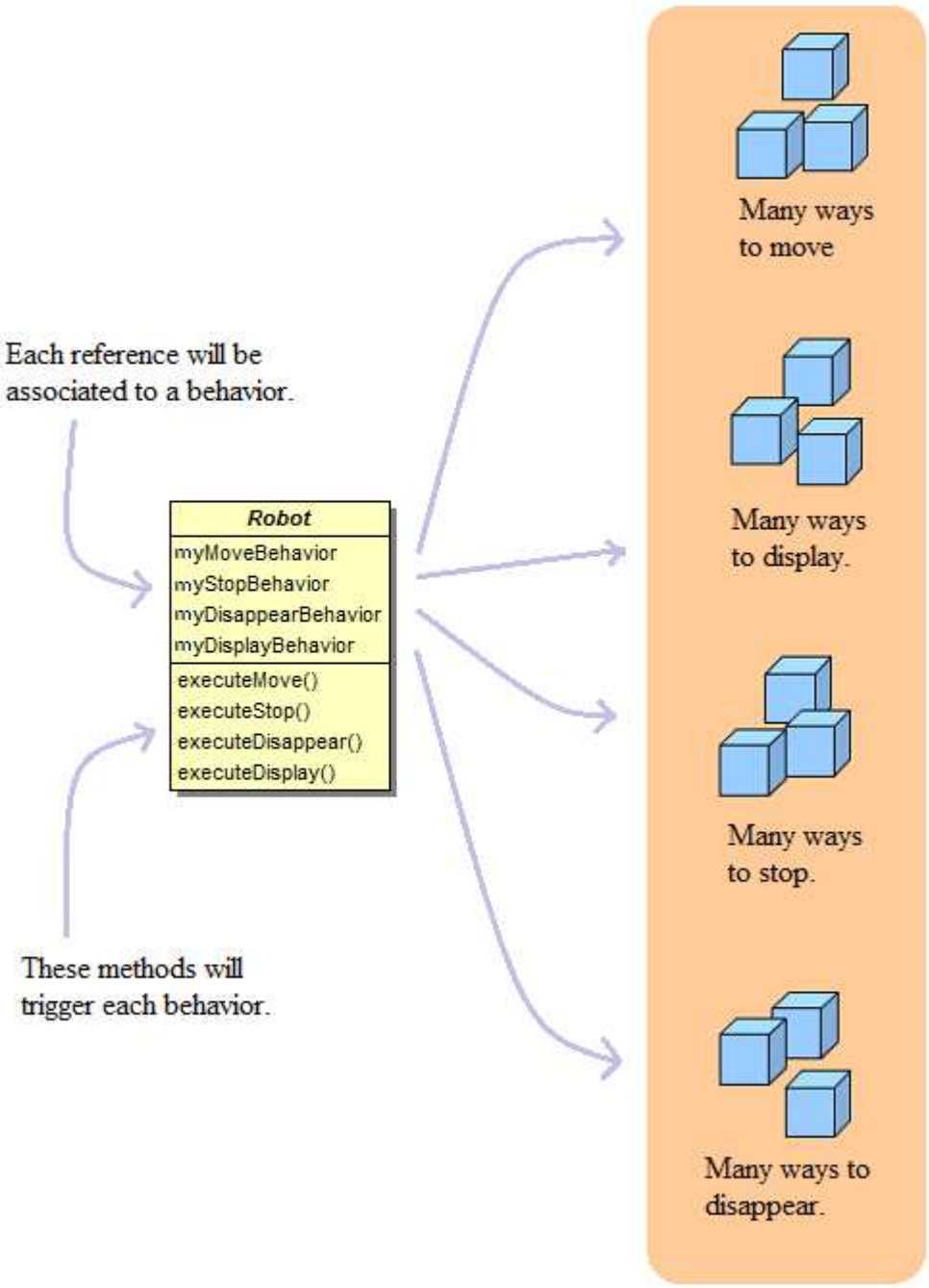

<<interface>>
Disappearable
disappear()
<<interface>>
Displayable
display()
DisappearQuick
disappear()
DisappearDisabled
disappear()
DisplayNormally
display()
DisplayForInvisible
display()
<<interface>>
Stopable
stop()
StopNormally
stop()
StopDisabled
stop()
A particular display for invisible robots.
Classes for robots that are not affected by a behavior.


These behaviors have been extracted from the Robot class, they become directly usable by any class that would need it.
Yes...
Here is a good example of code reusability.

2.3.3 Provide robots behavior

Now we create the link between a robot and behaviors that will be attributed to him.

- First, a robot must have a reference for each of its behavior. This is a rule that applies to all types of robots, we create these references in the parent class Robot.
- Finally, any robot must use behaviors it has. For this we create in the superclass Robot, and for each behavior, a public method that will be responsible for the execution of this behavior.



I would like to know how we code all of it !

2.3.4 Coding

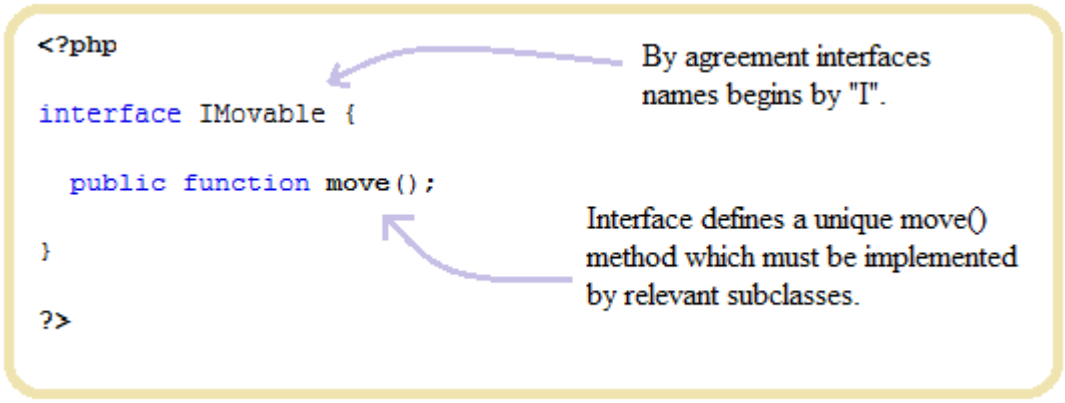
It's time to start coding PHP. Start with the behaviors.

We have 4 interfaces:

- Movable
- Displayable
- Stopable
- Disappearable

... and 2-3 classes per interface for specific behaviors.

IMovable interface:



The 3 others interfaces:

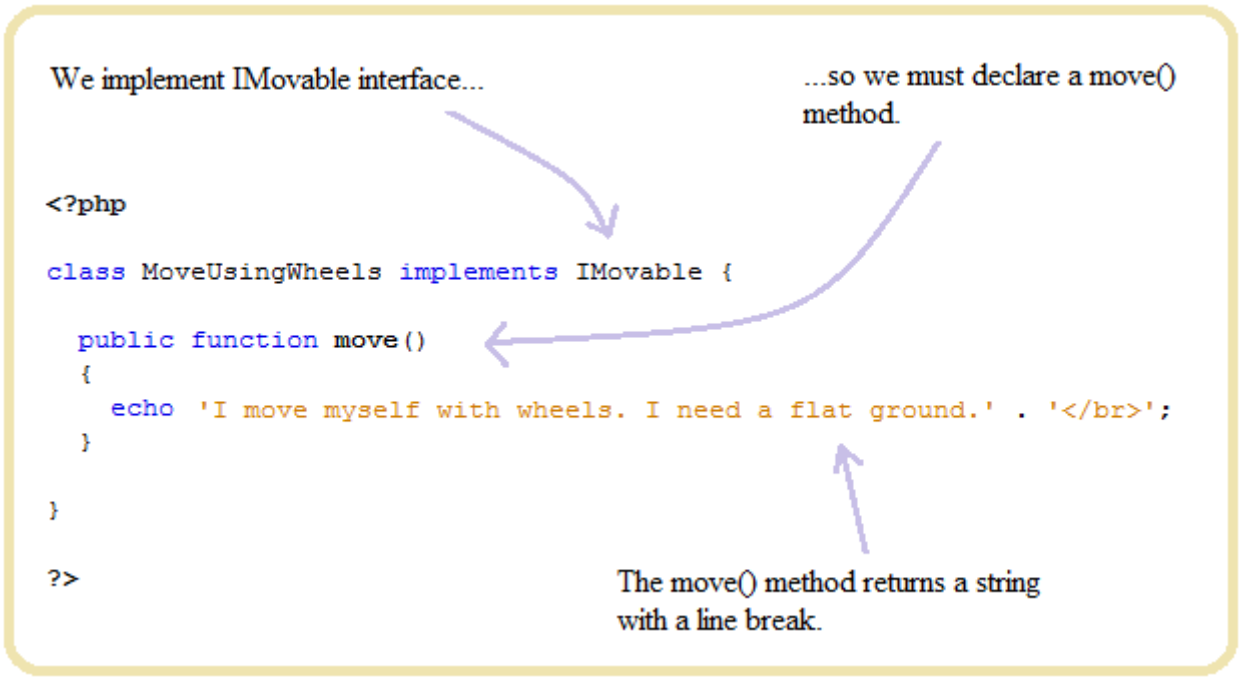
```php
<?php

interface IDisplayable {

  public function Display();

}

?>
```

```php
<?php

interface IStopable {

  public function stop();

}

?>
```

```php
<?php

interface IDisappearable {

  public function disappear();

}

?>
```

Behind IMovable interface, we have the classes:

- MoveUsingWheels
- MoveUsingLegs
- MoveDisabled



Do the same for the other two classes:

```php
<?php

class MoveUsingLegs implements IMovable {

  public function move()
  {
      echo 'Like a human, i move myself with my legs.' . '</br>';
  }

}

?>
```

```php
<?php

class MoveDisabled implements IMovable {

  public function move()
  {
    echo 'I can\'t move myself.' . '</br>';
  }

}

?>
```

Behind IDisplayable interface, we have the classes:

- DisplayNormally
- DisplayForInvisible

```php
<?php

class DisplayNormally implements IDisplayable {

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

}

?>
```

```php
<?php

class DisplayForInvisible implements IDisplayable {

  public function display()
  {
    echo 'Special display for invisible objects.' . '</br>';
  }

}

?>
```

Behind IStopable interface, we have the classes:

- StopNormally
- StopDisabled

```php
<?php

class StopNormally implements IStopable {

  public function stop()
  {
    echo 'I stop myself.' . '</br>';
  }

}

?>
```

```php
<?php

class StopDisabled implements IStopable {

  public function stop()
  {
    echo 'I don\'t have Stop function.' . '</br>';
  }

}

?>
```

Behind IDisappearable interface, we have the classes:

- DisappearQuick
- DisappearDisabled

```php
<?php

class DisappearQuick implements IDisappearable {

  public function disappear()
  {
    echo 'Fade to invisible.' . '</br>';
  }

}

?>
```

```php
<?php

class DisappearDisabled implements IDisappearable {

  public function disappear()
  {
    echo 'Invisibility not available.' . '</br>';
  }

}

?>
```

We coded all variables behavior. It only remains to encode robots classes.

Four classes are needed:

- Superclass Robot.
- Cleaner robot, extend Robot class.
- Spy robot, extend Robot class.
- Sensor robot, extend Robot class.

```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
```

The 4 references defining the behaviors, will be valued by subclasses.

A behavior execution is simply delegated to the object that is responsible for that behavior.

```php
<?php

class Cleaner extends Robot {

    function __construct() {

        echo "I'm Cleaner constructor.". '</br>';

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

    }

}

?>
```

The constructor is responsible for choosing the 4 behaviors.

We instanciate the concret behaviors for Cleaner robot.

```php
<?php

class Spy extends Robot {

    function __construct() {

        echo 'I\'m Spy constructor.'. '</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 'I\'m Sensor constructor.'. '</br>';

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

    }

}
?>
```

Our Strategy pattern is ready to be used. For this we will write a little program a few lines, which will use the Strategy pattern, ie instantiate robots, and ask them to perform actions.
This program is somewhat the user of Strategy pattern. We can also call it Controller since it controls the entire operations.

In the case of PHP, the controller can be placed in the index.php file because it is automatically executed to the invocation of a URL.

Fichier index.php:

```php
<?php

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

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

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

// Tests with a Spy instance:
$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;

?>
```

We instanciate a robot.

We make him execute his 4 behaviors.

It remains only to run our controller (index.php) via a URL in a web browser.

Mr Sulu, can you display the result ?

localhost/DesignPatterns_2012_EN/Strategy_2012/

```
*** Controller: start
I'm Spy constructor.
Like a human, i move myself with my legs.
Special display for invisible objects.
I stop myself.
Fade to invisible.
I move myself with wheels. I need a flat ground.

I'm Cleaner constructor.
I move myself with wheels. I need a flat ground.
Standard display.
I stop myself.
Invisibility not available.

I'm Sensor constructor.
I can't move myself.
Standard display.
I don't have Stop function.
Invisibility not available.

*** Controller: end.
```

We see the constructor.

Behaviors are executed.

Each robot have the behaviors we gave him.

It's really interesting.

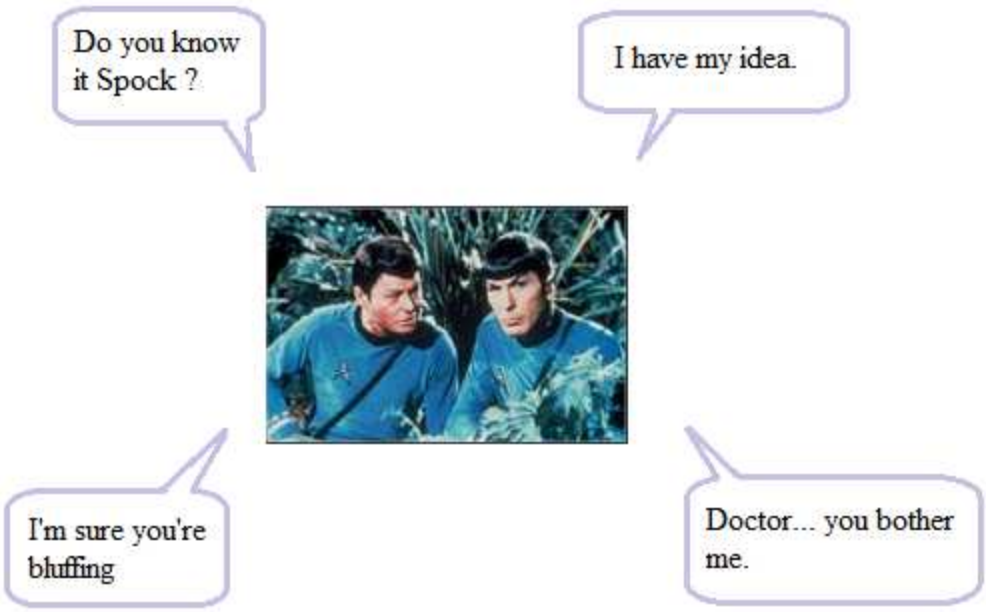
Interesting Mr Spock, but not completely satisfying: we made an effort to separate the variability and the commonality, but in the end the choice behavior of the robots is statically defined, since this

choice is made by each robot constructor.

If things are addressed in terms of responsibility, we can say that currently each robot is responsible for selecting its behaviors.

But is this really the robots to decide ?

And if not them, who must take the responsibility ?



In our little Strategy pattern, the choice behavior of the robots, can be entrusted to the head, and the head is the controller (index.php).
Not only Controller will instantiate the robots, but it may change their behavior at any time.

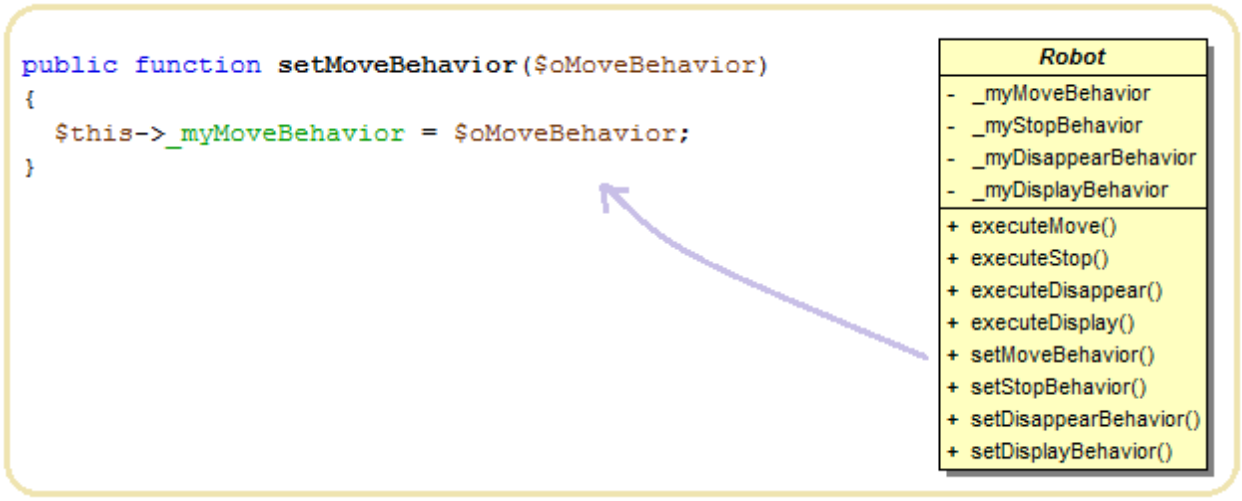
By transferring this responsibility to the controller, we reserve the right to change the behavior of robots at run time, totally dynamic, depending on the context.

2.3.5 Even more flexibility

The controller must:

- Choose (instantiate) a behavior.
- Pass this behavior to a robot so that it can adopt.

Add 4 methods to Robot class in order to change behavior:



The controller may use setMoveBehavior() method every time he wants to change the way a robot is moving. This behavior must be passed as a parameter.

Here is the code for the other three methods:

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

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

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

Slightly modify our controller:

```php
<?php

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

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

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

// Test with a Spy instance:
$oSpyRobot1 = new Spy();
$oSpyRobot1->executeMove();
$oSpyRobot1->executeDisplay();
$oSpyRobot1->executeStop();
$oSpyRobot1->executeDisappear();

// what about $oSpyRobot1 moving now with wheels ?
$oSpyRobot1->setMoveBehavior(new MoveUsingWheels);
$oSpyRobot1->executeMove();


echo $newline;
...
```

Spy constructor keep on defining default behaviors.

Let's modify a behavior of our robot.

And here is a robot on wheels !

What we get at run time:

localhost/DesignPatterns_2012_EN/Strategy_2012/

*** Controller: start
I'm Spy constructor.
Like a human, i move myself with my legs.
Special display for invisible objects.
I stop myself.
Fade to invisible.
I move myself with wheels. I need a flat ground.

What a change !

I'm Cleaner constructor.
I move myself with wheels. I need a flat ground.
Standard display.
I stop myself.
Invisibility not available.

I'm Sensor constructor.
I can't move myself.
Standard display.
I don't have Stop function.
Invisibility not available.

*** Controller: end.

He really changed his behavior. It's fascinating !

2.3.6 A global vision

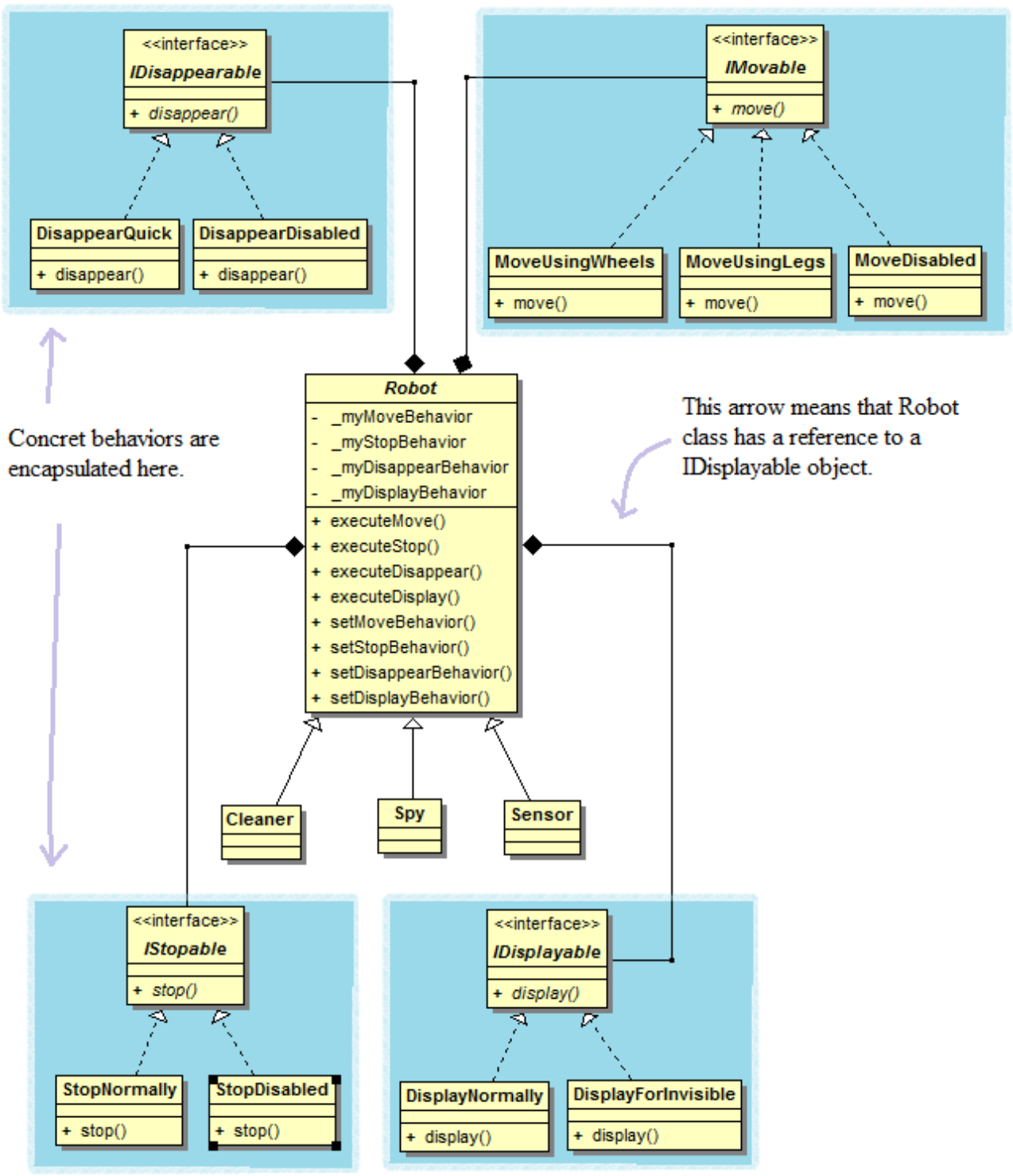
We coded well, and our Strategy pattern works fine. The original design has been completely transformed, but what have we done exactely ?

We extracted variable concepts and we have encapsulated them in families of behaviors.

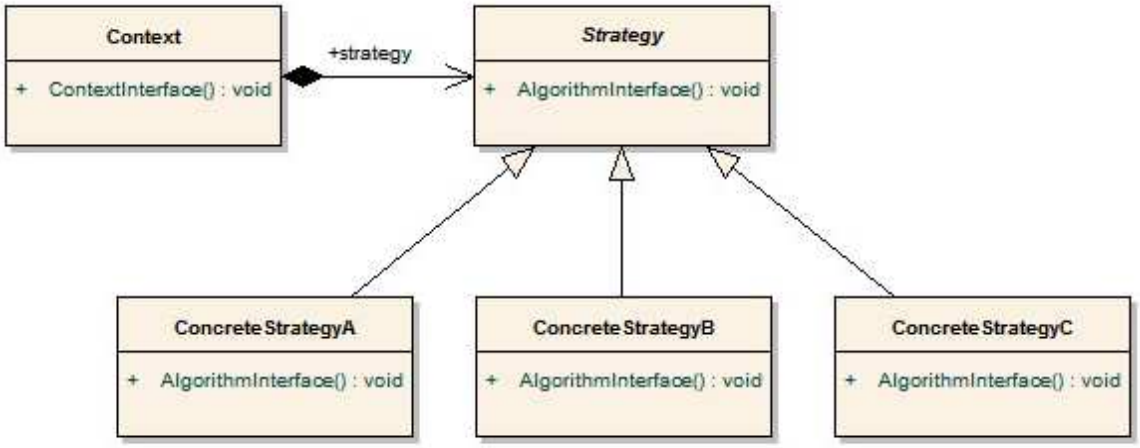
We kept the parent class Robot and its subclasses, but robots do not directly control their behaviors because they have been delegated to concrete classes that encapsulate behaviors.

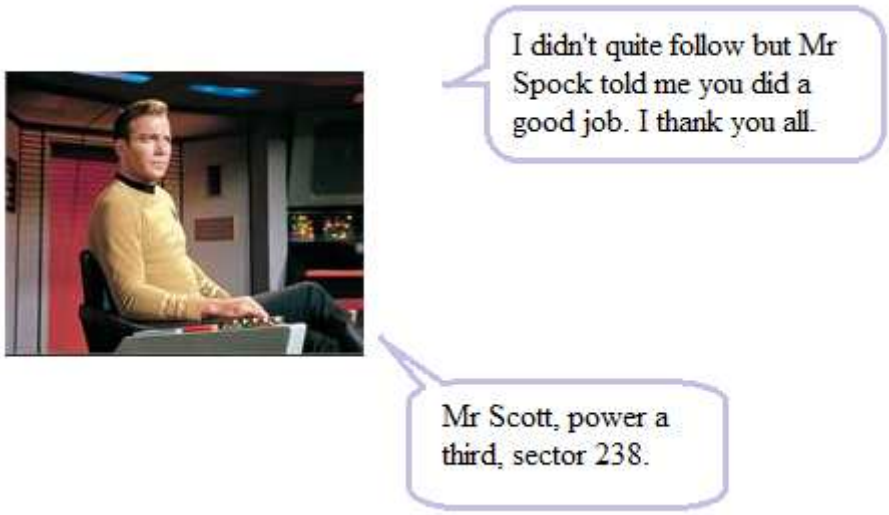
Finally we added methods to change the behavior of robots at runtime. This was absolutely impossible in the initial approach.

Family picture:

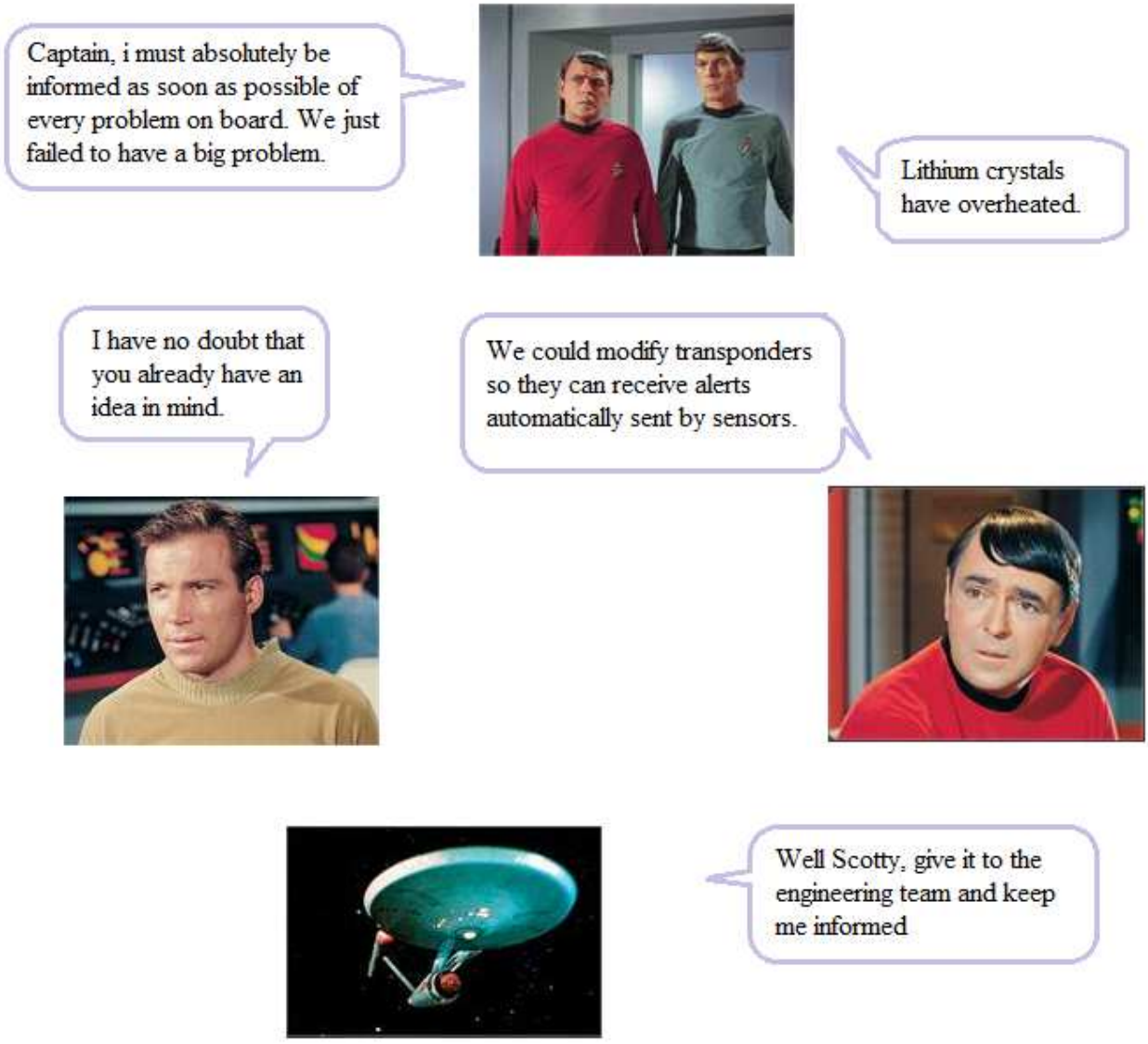


Here is a standard representation of the Strategy pattern that can be found on the internet:





3 OBSERVER PATTERN



Transponders are these small handheld devices that members of the Enterprise have to the belt. They use it to communicate, much like a ... cell phone.

We are asked to add a feature allowing transponders receive alerts when certain events occur on the propulsion system of the vessel.

It can be:

- The temperature of the crystal battery (matter that supply energy to the vessel). The warning threshold is set to 400 degrees centigrade.
- The level of vibration of the propulsion system. On a scale of 0-7, the warning threshold is set at 5.

- The speed of the propulsion system. On a scale of 0 to 100, the alert threshold is set at 95.

The transponders will take the initiative to be notified and to stop being notified.

Luckily, what it is required is exactly what can be handled by the Observer pattern. Let's go.

3.1 Heading Observer Pattern

People who love gardening, can subscribe to a specialized magazine. They will then receive a regular issue of the magazine, until they decide to end their subscription.

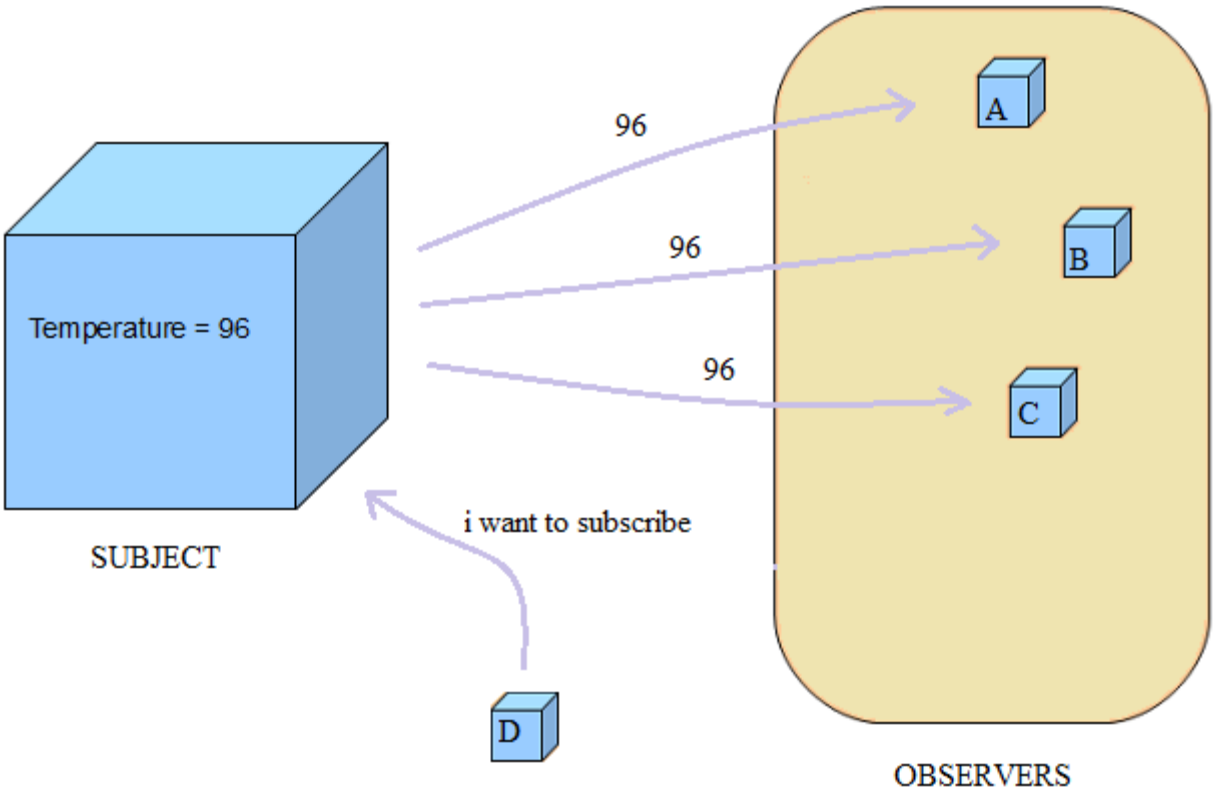
The Observer pattern works on the same principle:

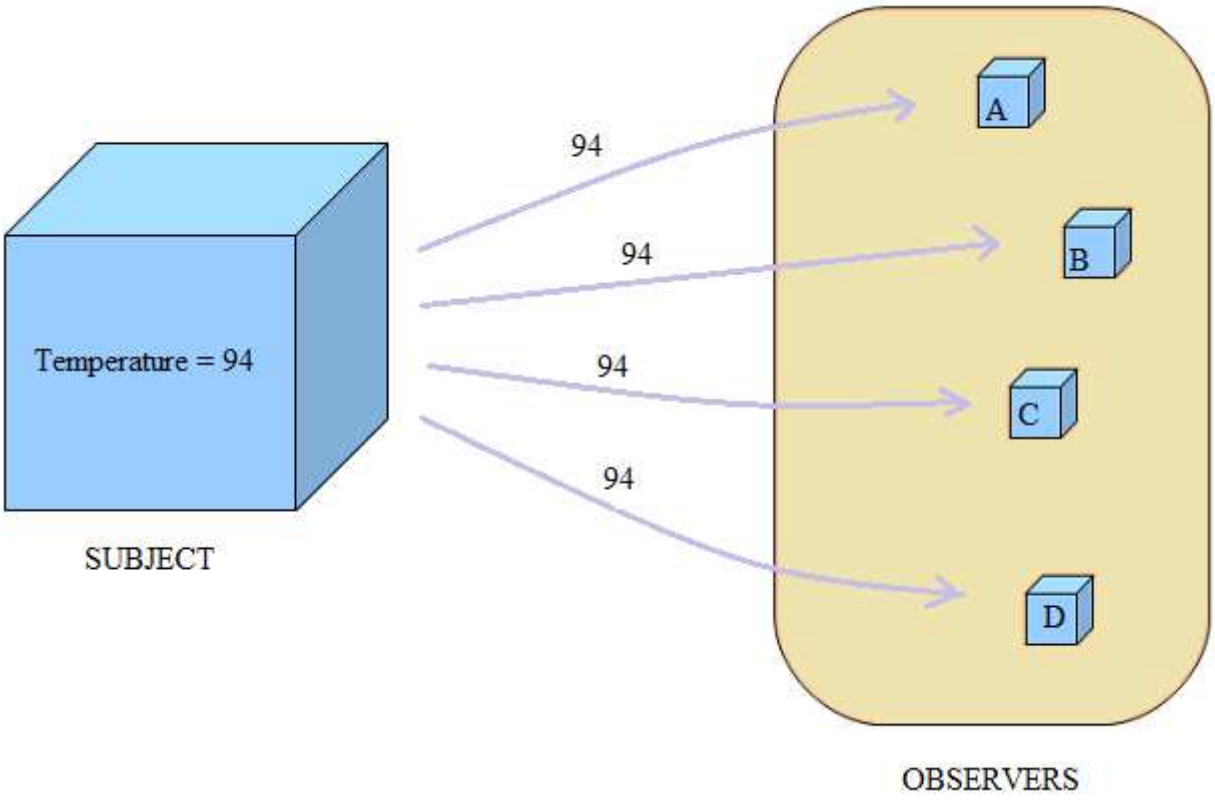
A resource is available through a subscription. This resource is called the Subject.

People interested in this resource subscribe to it. These people are called the Observers.

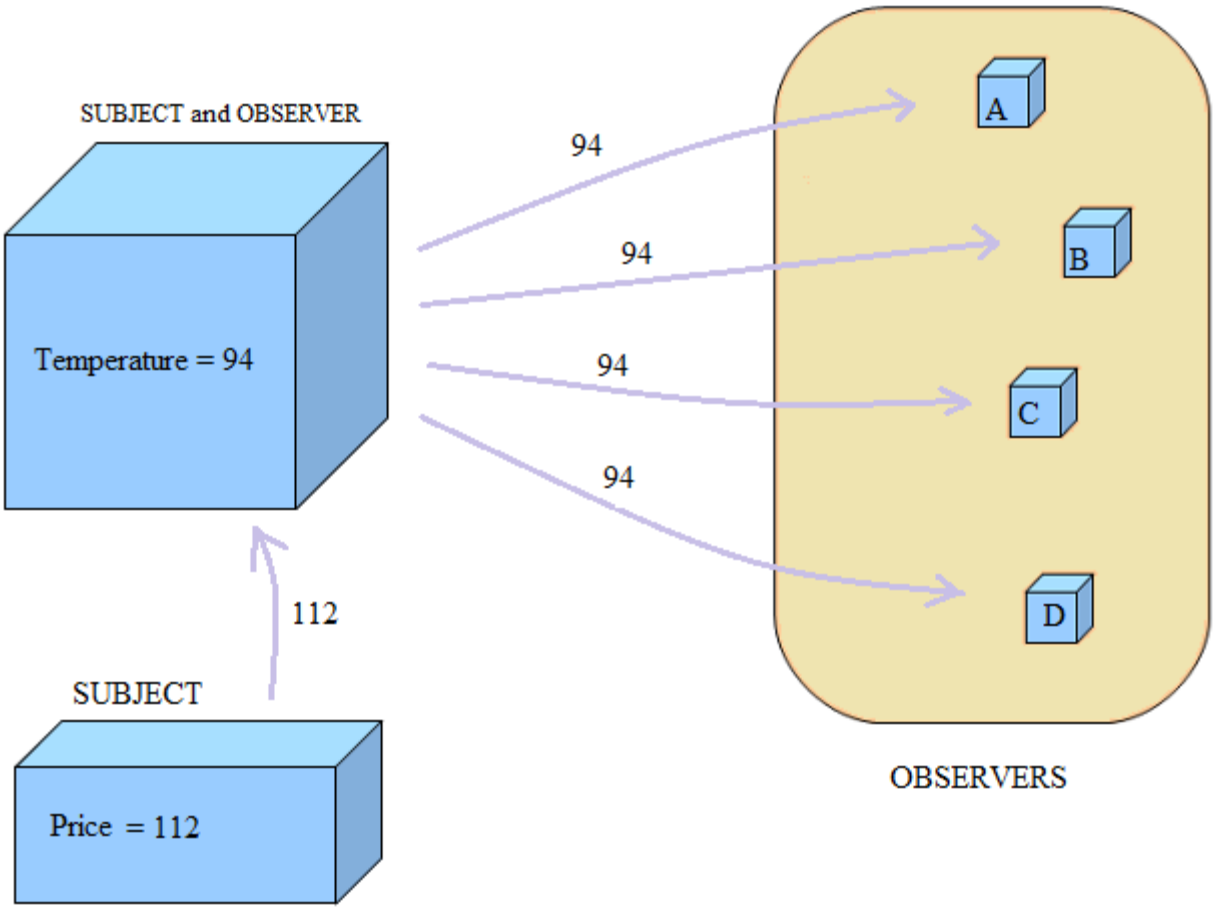
As long as the Observer is a subscriber, he receives notifications from the Subject.

Any observer can end his subscription. It will stop receiving notifications from the Subject.



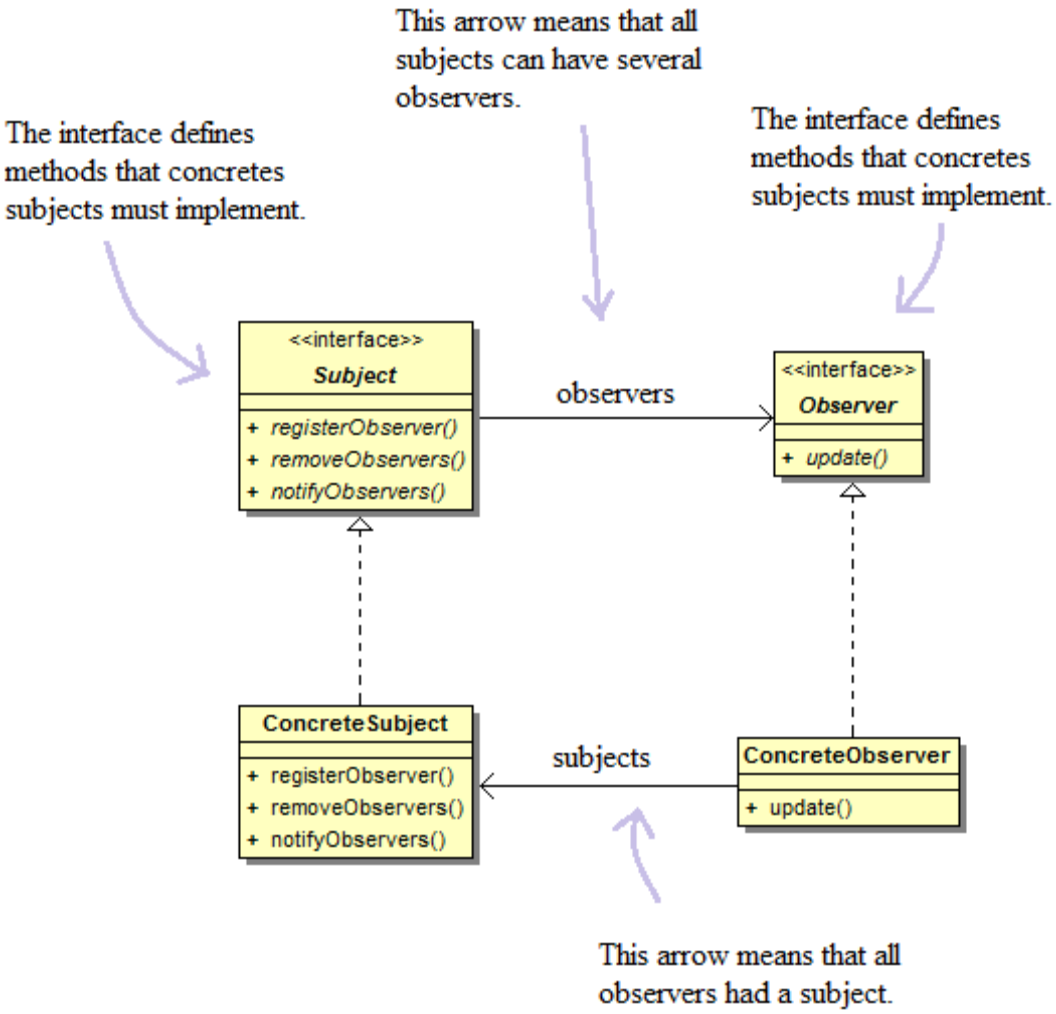


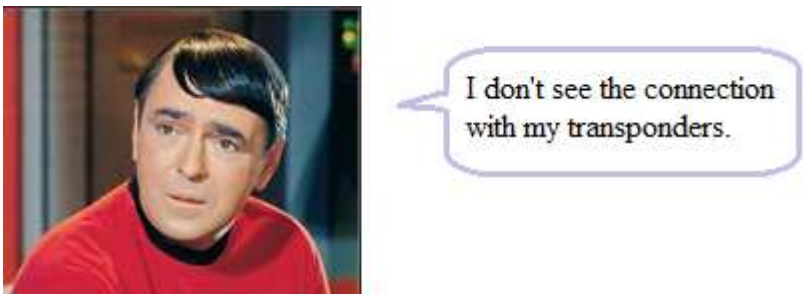
A Subject can also be itself Observer to another Subject:


SUBJECT and OBSERVER
Temperature = 94
94
94
94
94
A
B
C
D
112
SUBJECT
Price = 112
OBSERVERS

3.1.1 Class Diagram

Here is the standard class diagram of the Observer pattern:





3.2 Implementation

Here we are Mr Scott.

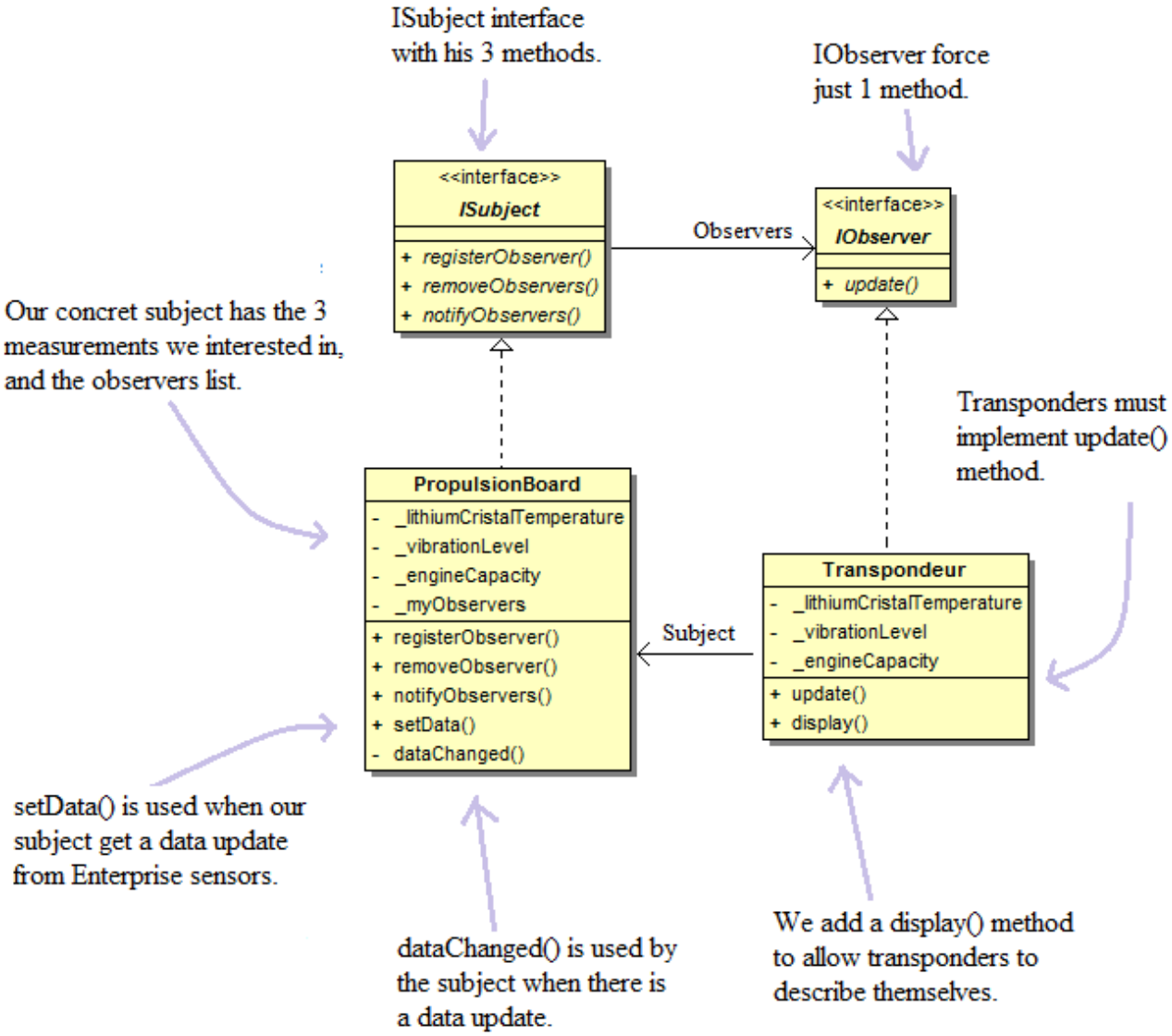
Our subject has to manage the information that interests Observers:

- Lithium crystal temperature.
- Vibration level of the propulsion system.
- Energy level of the propulsion system.

The subject gets this information directly from the various sensors of the Enterprise. No matter how it was obtained, the thing is that he has, and they are constantly updated.
The concret subject is like a dashboard of the Enterprise propulsion system. Therefore we can call it "PropulsionBoard".

Transponders are the concrete observers. They must be able to subscribe and unsubscribe from the subject.

This gives us the following class diagram:


ISubject interface with his 3 methods.
IObserver force just 1 method.
<<interface>>
ISubject
+ registerObserver()
+ removeObservers()
+ notifyObservers()
Observers
<<interface>>
IObserver
+ update()
Our concret subject has the 3 measurements we interested in, and the observers list.
Transponders must implement update() method.
PropulsionBoard
- _lithiumCristalTemperature
- _vibrationLevel
- _engineCapacity
- _myObservers
+ registerObserver()
+ removeObserver()
+ notifyObservers()
+ setData()
- dataChanged()
Subject
Transpondeur
- _lithiumCristalTemperature
- _vibrationLevel
- _engineCapacity
+ update()
+ display()
setData() is used when our subject get a data update from Enterprise sensors.
dataChanged() is used by the subject when there is a data update.
We add a display() method to allow transponders to describe themselves.

3.2.1 Coding

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);

}

?>
```

Transponder class:

```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

?>
```

We implement IObserver interface.

Constructor do the registering to the subject.

PropulsionBoard class:

```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() . ": One more observer .</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

?>
```

Observers are handled in an array.

To add an observer means add it in the array.

Here we unsubscribe an observer.

We notify all observers

That's where PropulsionBoard receives data updates.

We will use index.php as the controller of our pattern.

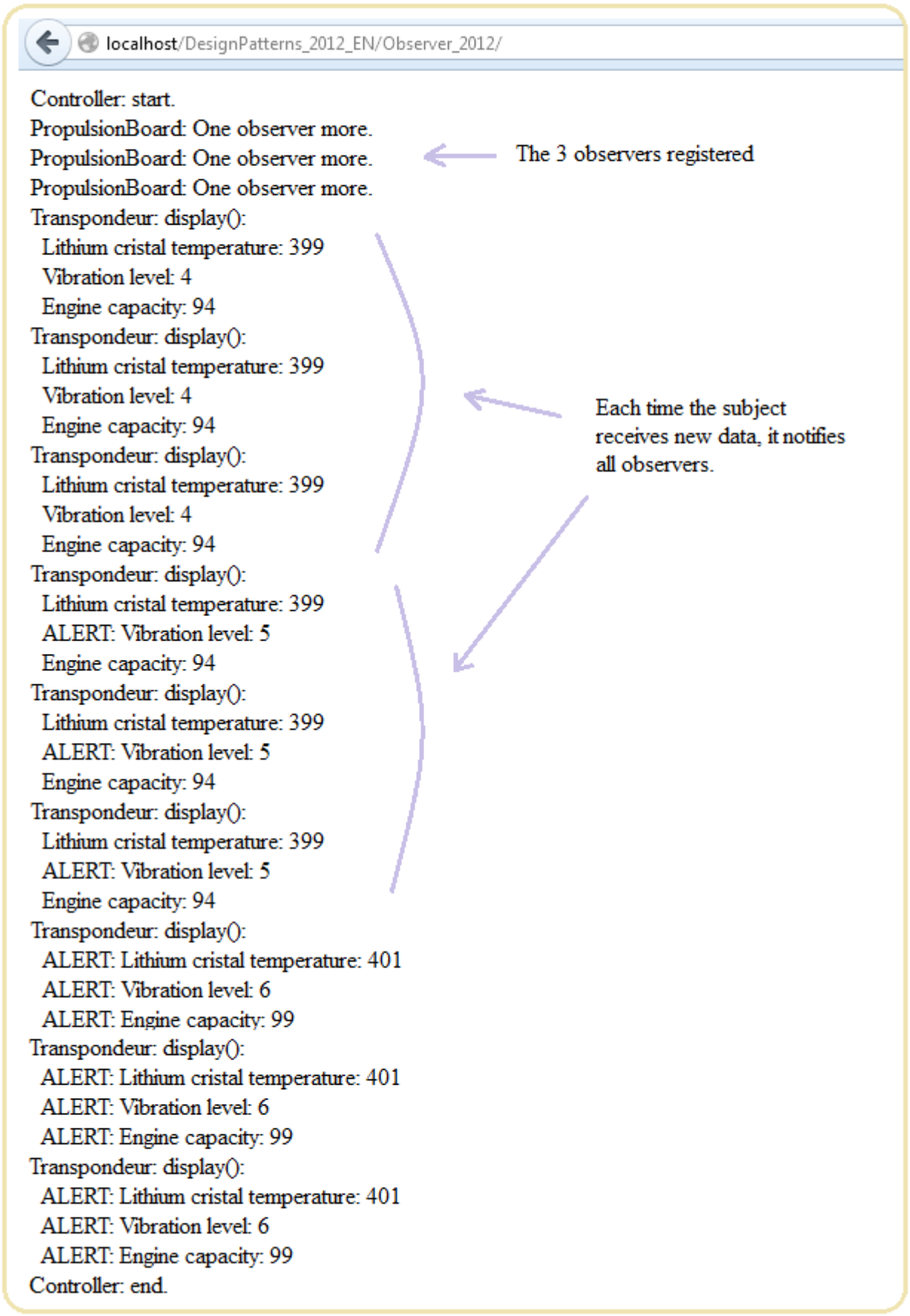
The controller will instantiate objects and use them.

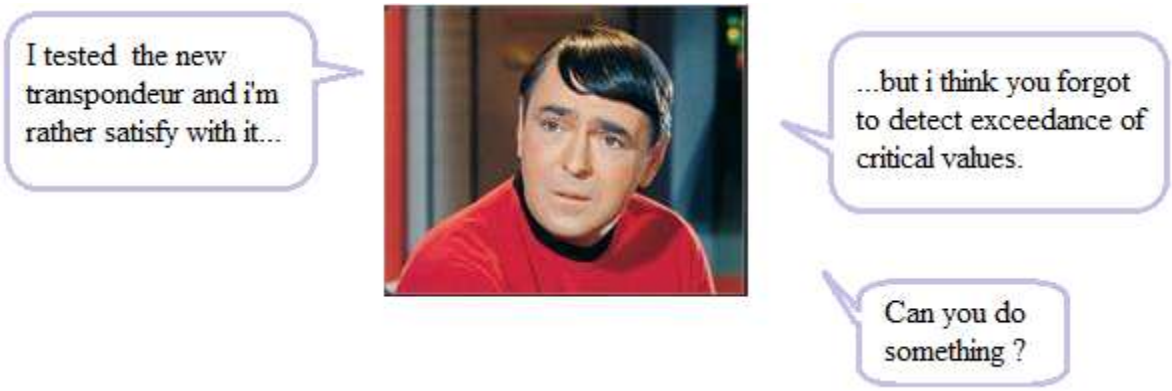
```php
<?php

//************************************************************
// 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;

?>
```

We instanciate 1 subject and 3 observers.

Sending new data to the subject will trigger notifications to all observers.

The result of the execution of the controller should look like this:



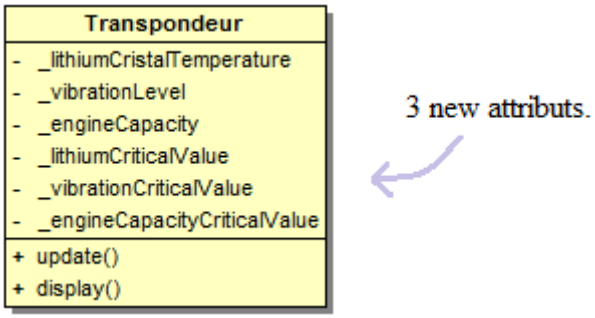


You are absolutely right Mr Scott. We will fix it right away.
Overflow detection of critical values is a responsibility of the transponder. It must know these critical values to be able to react when they are reached or exceeded.

Let's add attributes in the transponder class for storing the critical values.

These values will be transmitted to the transponder during their instantiation.

We also modify the display() method to adjust the display when the values are greater than or equal to their critical values.



```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

?>
```

New attributs are initialised by constructor.

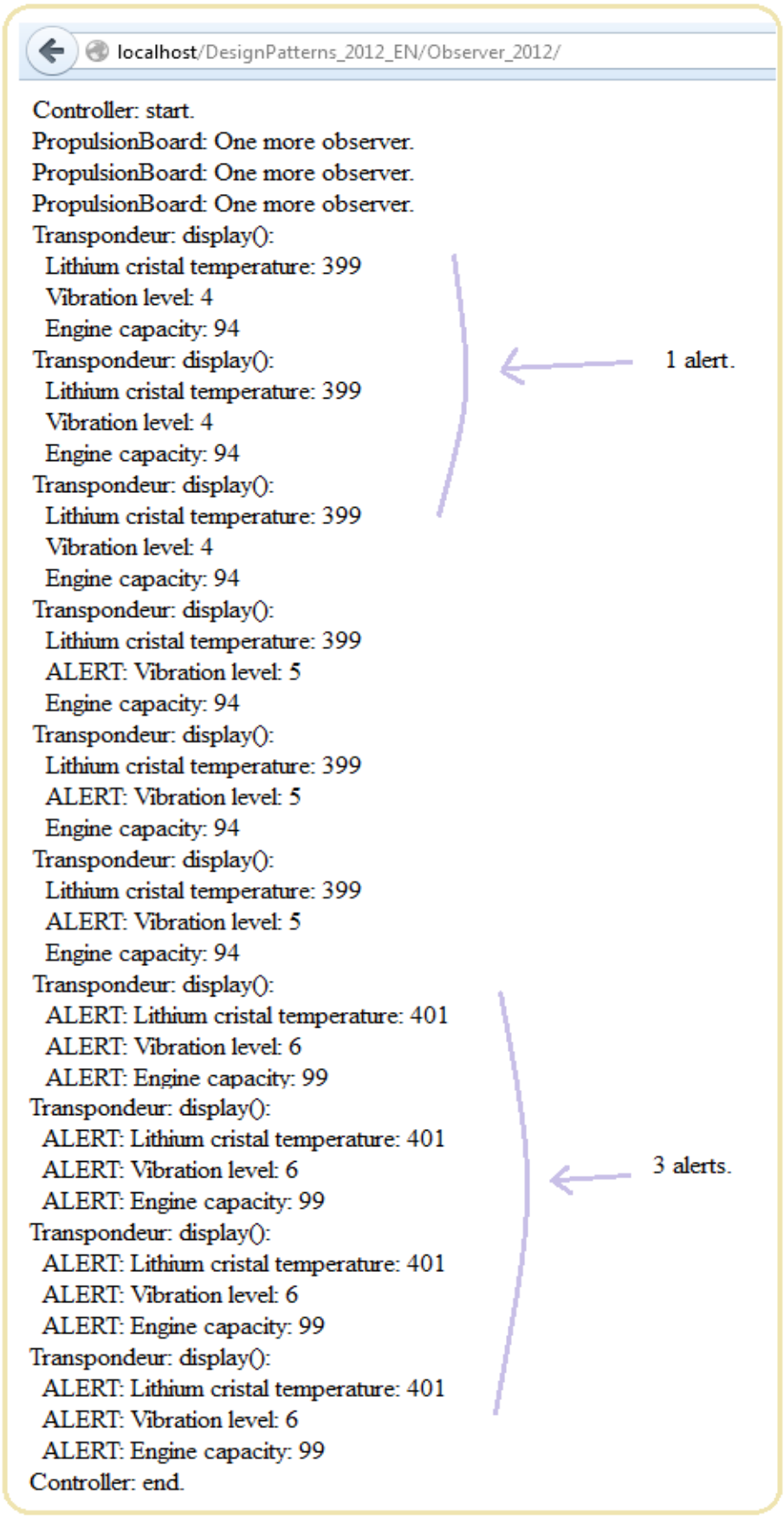
display() methof knows when it must throw an alert.

Finally, we must modify our controller:

```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;

?>
```

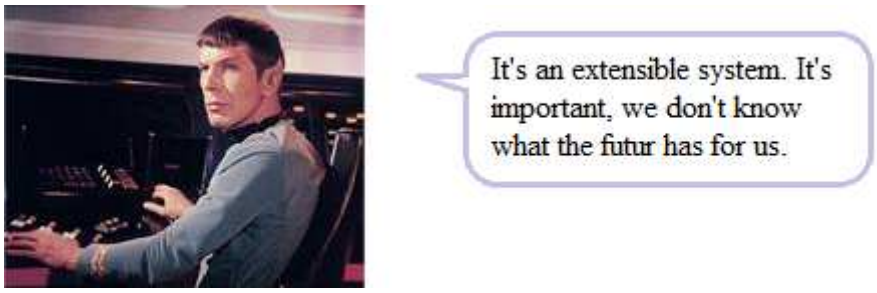
Controler knows the critical values and pass them to Observers at their creation time.

Let’s see the controller running:


localhost/DesignPatterns_2012_EN/Observer_2012/
Controller: start.
PropulsionBoard: One more observer.
PropulsionBoard: One more observer.
PropulsionBoard: One more observer.
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
1 alert.
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
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
Transpondeur: display():
ALERT: Lithium cristal temperature: 401
ALERT: Vibration level: 6
ALERT: Engine capacity: 99
3 alerts.
Controller: end.

In our design, the transponder is the only observer of the PropulsionBoard subject. But suppose we are asked a voice server recites PropulsionBoard values in speakers.

Like transponders, this new class should implement the interface IObserver and subscribe to PropulsionBoard. It would merely have a different use of data obtained.



4 DECORATOR PATTERN

The Enterprise conducts regular missions involve sending teams and equipment to the outside of the vessel to perform a task.

It may include exploration missions, control, supply, evacuation...

Captain Kirk wants to have a system to identify the components of a mission (teams and hardware devices) and know the weight of these components, and the total weight and the number of people.

The teams are as follow:

- Medical group (4 people).
- Security group (6 people).
- Scientific group (3 people).

The hardware units are as follows:

- Light medical hardware unit.
- Heavy medical hardware unit.
- Defensive hardware unit (weapons).
- Logistics hardware unit (shelter, bedding, food) .

A mission consists of any combination of teams and hardware units.

Example 1:

1 Medical group.

1 Heavy medical hardware unit.

2 Security group.

Exemple 2:

1 Logistics hardware unit.

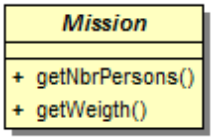
2 Scientific group.

At work !

The difficulty here is that we must both:

- "compose" a mission with a set of elements that are the teams and the hardware units.
- Ask each element, especially for its weight or the number of people.
- Ask the mission as a whole, to get the total weight and the total number of people.

One could imagine an abstract class Mission, and a set of subclasses, each representing a combination of elements:



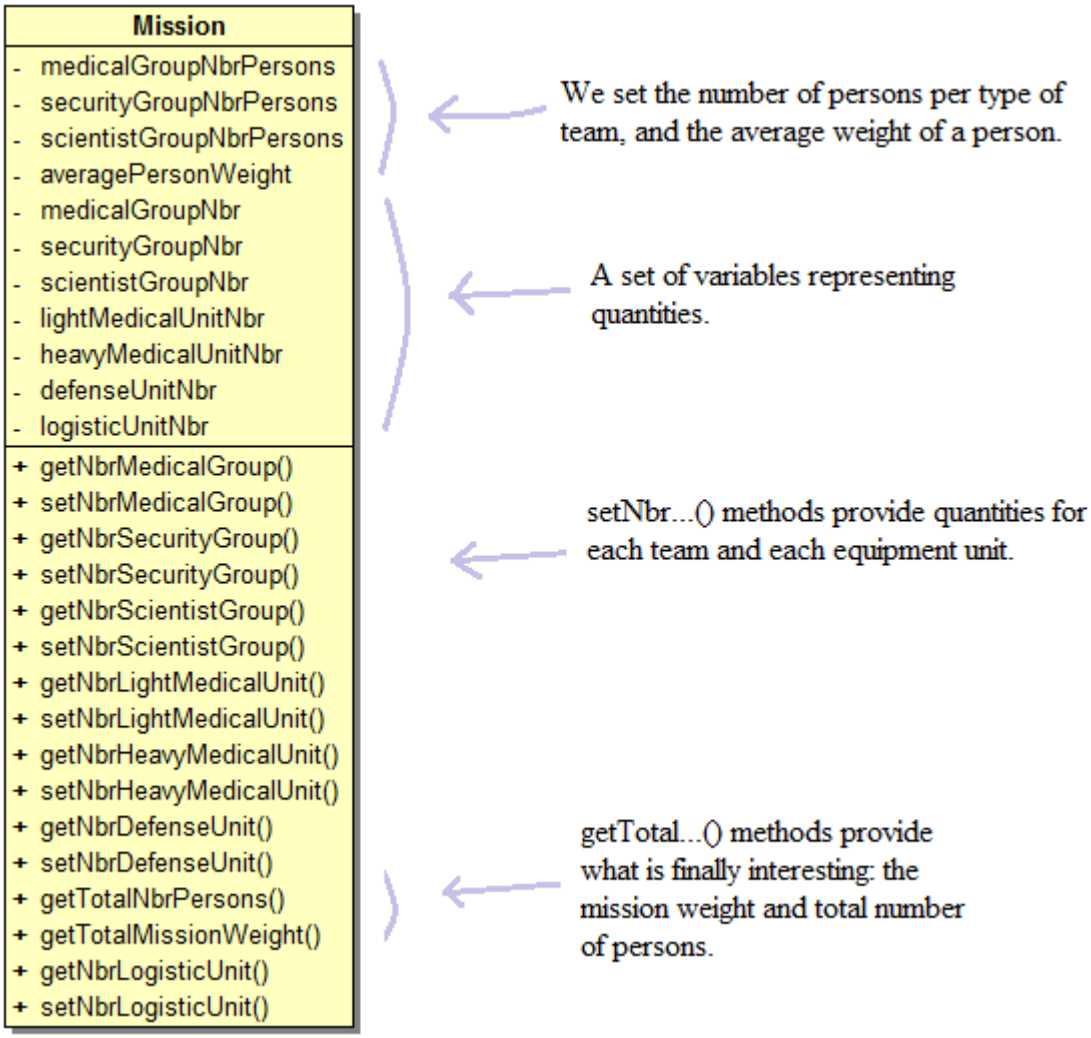
But knowing:

- That we have 3 types of teams and four types of hardware units.
- that for a mission, each of these elements can be doubled, tripled (there is actually no limit).

The number of classes to create would be virtually unlimited. This approach is definitely abandoned.

OK it was not a good idea. Try something simpler:

We could keep the Mission class, but the composition of the mission would be represented by attributes that indicate their amounts:



We get a single class which contains all the information necessary for a mission.

This approach is feasible but has drawbacks:

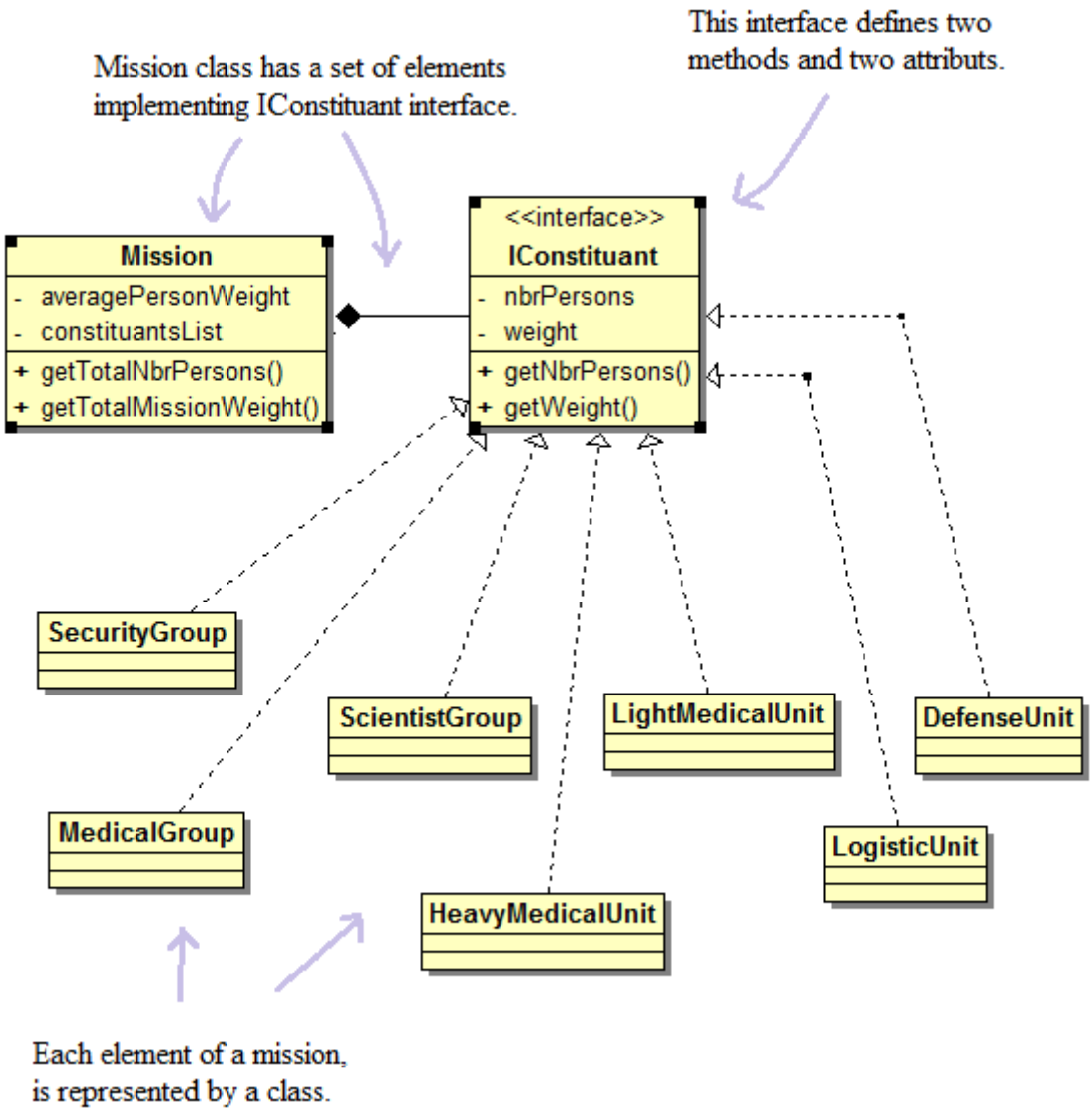
- It's a big class and it contains information of different natures.
- Adding a new team or hardware device types, requires modifying the class.
- A mission not using a component, will implement all the same the methods of this component.

In conclusion, we can do better.

A last idea ?

We could keep a light Mission class, and create a class for each component of a mission (teams and hardware units).

Then we attach the components to the Mission using in the Mission class, a set of references in the form of a list, which allows the Mission to know its components and ask them.



This approach is more elaborate:

- The mission is separated from its components.
- The components are hidden behind an interface, so that the mission sees only IConstituants.

The only slight drawback is that the mission must manage its constituents through its constituantsList attribute. Nevertheless this approach could be used, but we can do even better with the Decorator pattern.

4.1 Heading Decorator pattern

The Decorator pattern is similar to Russian dolls: The objects are placed into each other to form the final composite entity.

Consider a mission composed of the following:

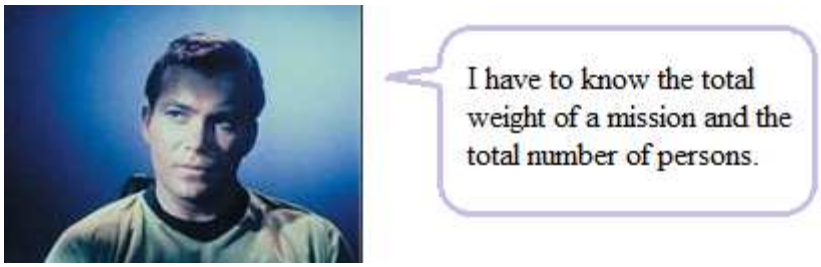
- A Security group (weight: 1.5 tons).
- A light Medical unit (weight: 2 tons).
- A Defense unit (weight: 3 tons).

The encapsulation of objects will be as follows:

- A Mission object (which is the base object, or "decorated" object), encapsulated in a Security group object.
- The whole is encapsulated in a light Medical unit.
- The whole is encapsulated in a Defense unit.

In fact the order of encapsulation does not matter in this example, but may be depending on the problem to solve.

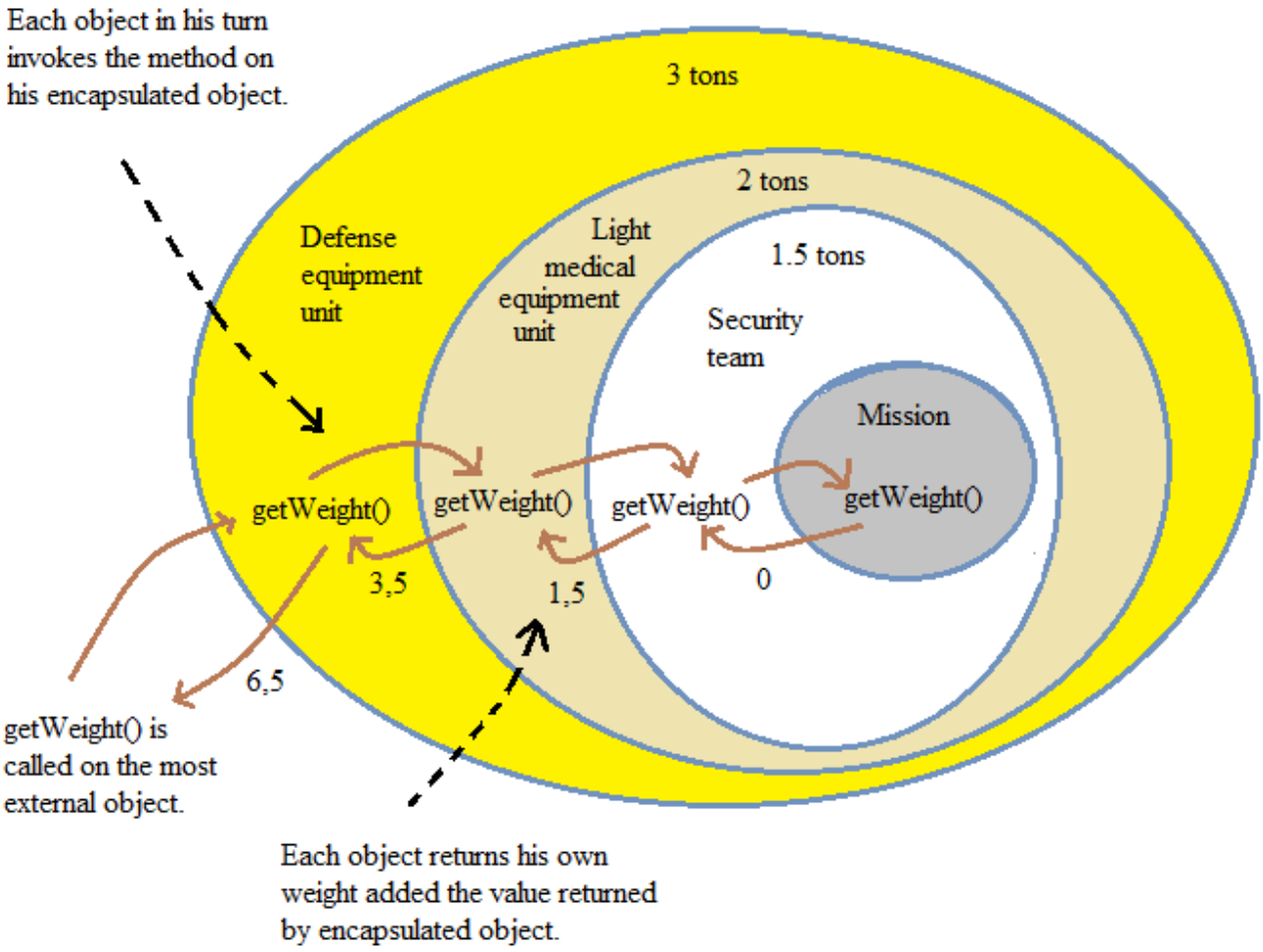
But that's not all.
There is a second important aspect: delegation.



To answer these two questions, use the getNbrPersons() and getWeight() methods. All objects must implement these methods.

These methods are invoked on the object encompassing, and will be automatically delegated to the included object, and so on until the innermost object.

The response will have been completed by each object to form at the end, a comprehensive response from the mission.

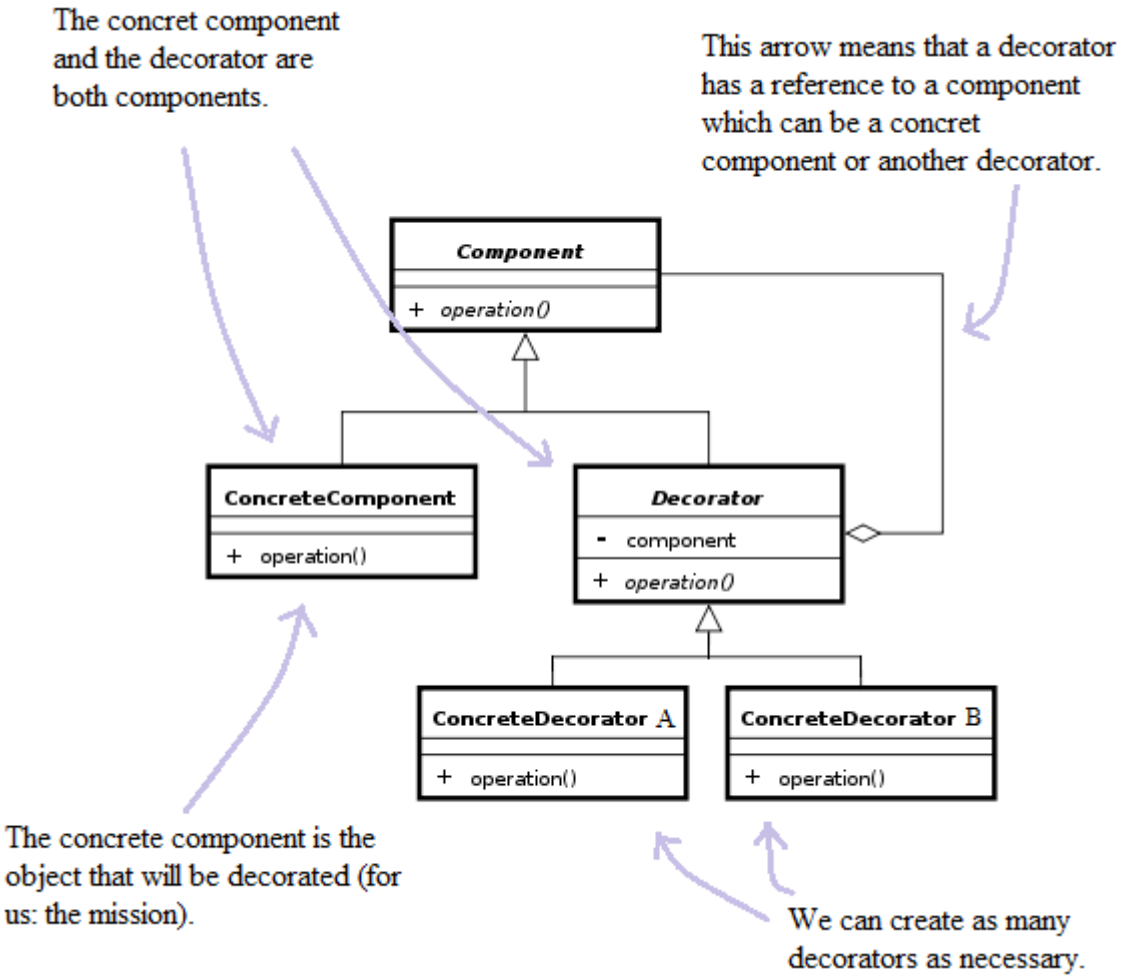


We will similarly implement getNbrPersons() method to get the total number of people in the mission and satisfy the captain requirements.

This way to call several times the same method in an interleaved manner, is based on recursion (see the chapter: Recursion in Apendix).

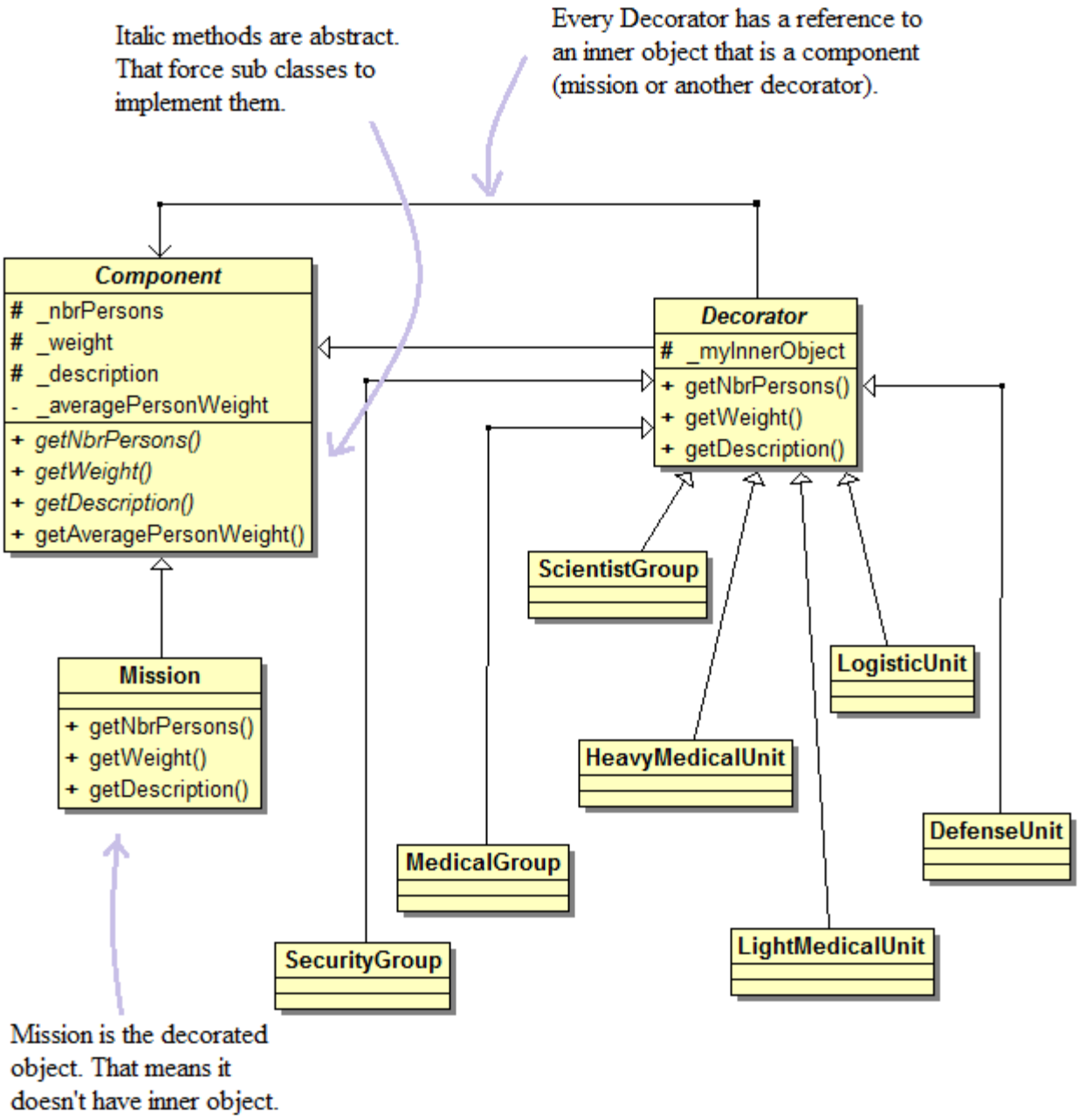
This special technique is rarely used. It also appears in another pattern: Composite.

Consider the official model of Decorator pattern:



4.2 Class Diagram

We have all the elements to define the class model that meet the specifications, based on the Decorator pattern.


Italic methods are abstract. That force sub classes to implement them.
Every Decorator has a reference to an inner object that is a component (mission or another decorator).
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 is the decorated object. That means it doesn't have inner object.

4.3 Implementation

Let us code the different classes. The Decorator pattern is known to generate many small classes. Sure it takes a concrete class for the decorated object, and one for each designer.

4.3.1 Coding

The Component class:

```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;
  }

}

?>
```

protected = inherited by subclasses, as a private attribut.

Abstract methods must be implemented by subclasses.

The Decorator class:

```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;
  }

}
?>
```

We call the parent class constructor.

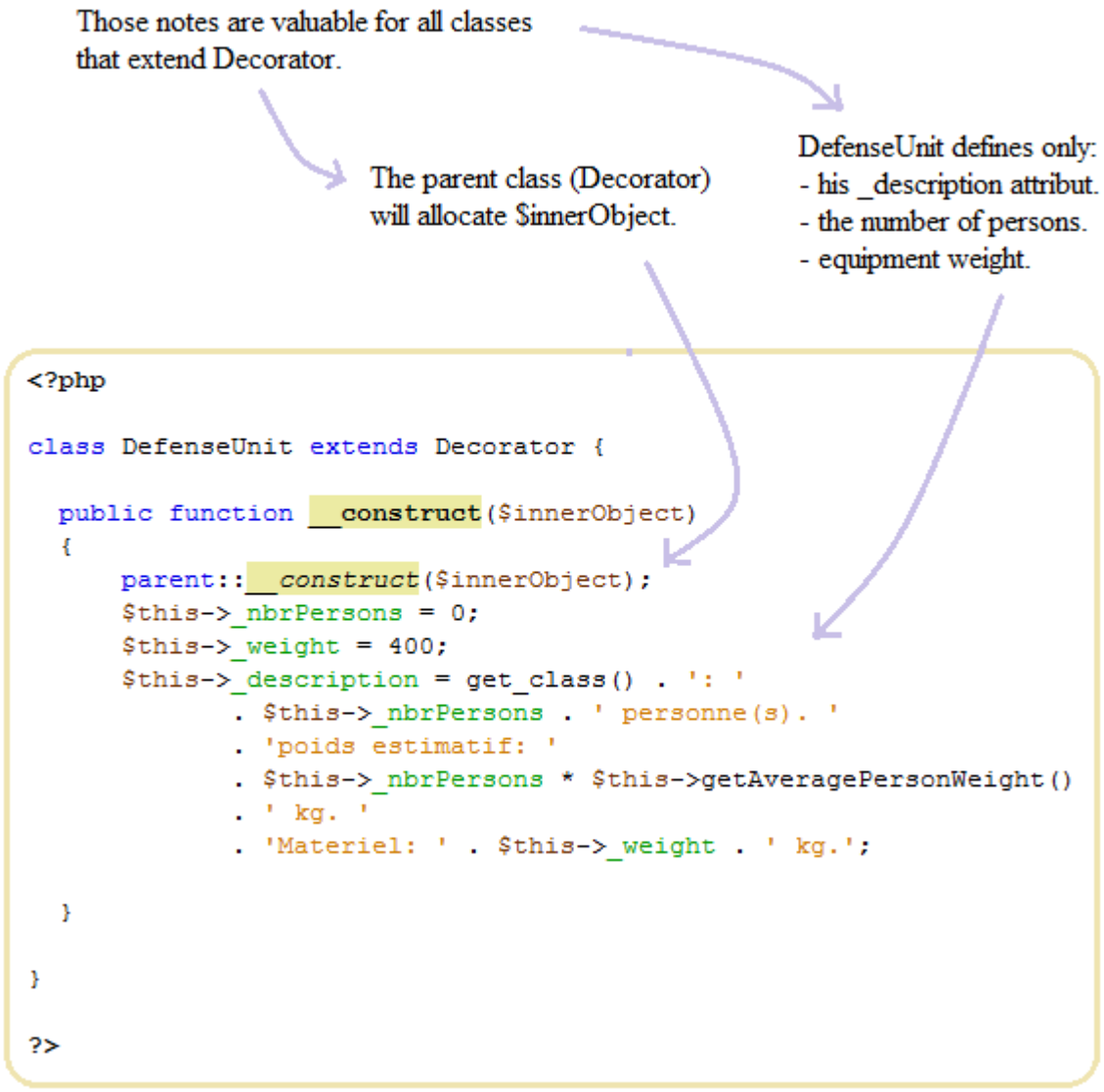
Abstract methods are implemented here.

That's where recursion start working: every Decorator will call getxxx() method of his inner object. This call will be propagated until the most internal object: the mission.

The Mission class:

```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.';
  }

}

?>
```

The DefenseUnit class :




```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.';

  }

}

?>
```

The HeavyMedicalUnit class:

```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.';

  }

}

?>
```

The LightMedicalUnit class:

```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.';

  }

}

?>
```

The LogisticUnit class:

```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.';

  }

}

?>
```

The MedicalGroup class:

```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.';

  }

}

?>
```

The ScientistGroup class:

```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.';

  }

}

?>
```

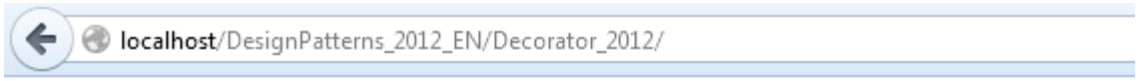
The SecutityGroup class:

```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.';
  }

}

?>
```

We will use the usual index.php as the controller of our pattern. The controller will instantiate objects and use them.

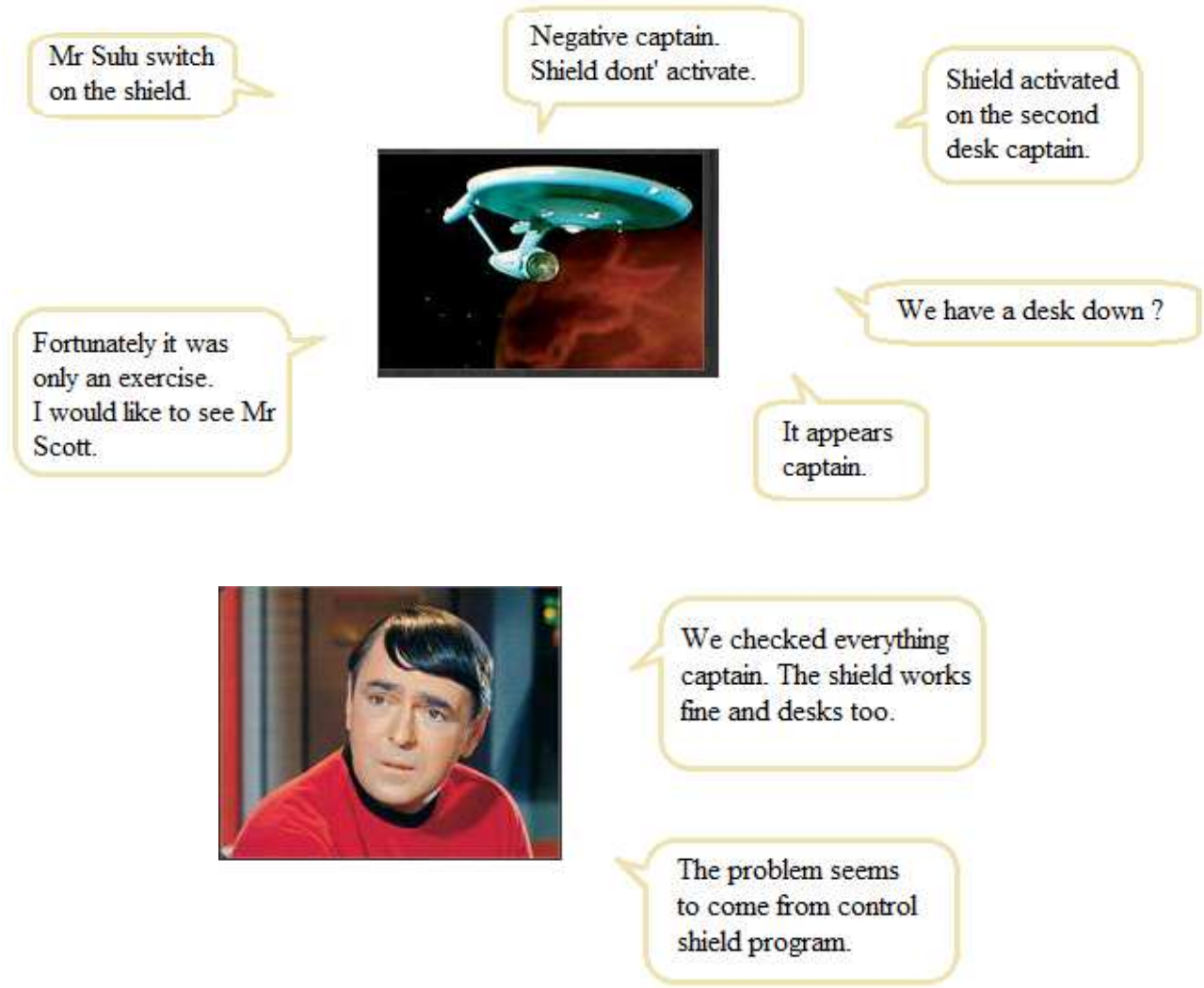
```php
<?php

//************************************************************
// DECORATOR pattern controller
//************************************************************

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

echo 'Controller: start.' , $newline;

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

$Mission2 = new Mission("Resupplier vessel evacuation.");
$Mission2 = new MedicalGroup($Mission2);
$Mission2 = new ScientistGroup($Mission2);
$Mission2 = new SecurityGroup($Mission2);
$Mission2 = new HeavyMedicalUnit($Mission2);

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

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

echo 'Controller: end.' , $newline;

?>
```

The execution result should look like this:




```
Controller: start.
Mission: Planet P731 exploration. 1 Head(s) of Mission. Estimated weight: 62 kg.
   DefenseUnit: 0 person(s). Estimated weight: 0 kg. Equipment: 400 kg.
   MedicalGroup: 6 person(s). Estimated weight: 372 kg. Equipment: 100 kg.
   SecurityGroup: 1 person(s). Estimated weight: 62 kg. Equipment: 1200 kg.
   SecurityGroup: 1 person(s). Estimated weight: 62 kg. Equipment: 1200 kg.
Total mission:
Nbr persons: 9
Weight: 3458 kg.
--------------
Mission: Resupplier vessel evacuation. 1 Head(s) of Mission. Estimated weight: 62 kg.
   MedicalGroup: 6 person(s). Estimated weight: 372 kg. Equipment: 100 kg.
   ScientistGroup: 9 person(s). Estimated weight: 558 kg. Equipment: 100 kg.
   SecurityGroup: 1 person(s). Estimated weight: 62 kg. Equipment: 1200 kg.
   HeavyMedicalUnit: 0 person(s). Estimated weight: 0 kg. Equipment: 350 kg.
Total mission:
Nbr persons: 17
Weight: 2804 kg.
--------------
Controller: end.
```

5 SINGLETON PATTERN



The shield is a unique resource that can be controlled by several desks. Under these conditions, the current state of the shield (open or closed) must be accessible from a single information. It is not necessary that this information is copied because even with a update system, there is a risk that a copy is not in tune with reality, at a moment. This obviously causes unpredictable behavior.

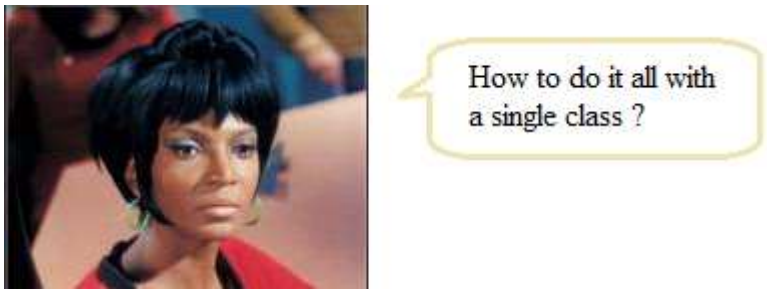
In this example we will ensure that the shield is indeed a unique resource that can be used by several desks simultaneously.

For this we will use the Singleton pattern, the simplest design patterns. It consists to a single class.

5.1 Heading Singleton pattern

The goal of the Singleton pattern is:

- Ensure that a class is instantiated only once, ie there can be only zero or one object of this class.
- Provide a single point of access to the object.



Our unique resource is the shield of the Enterprise. Decide now that the pattern class will be called ShieldSingleton.

It is clear that the ShieldSingleton class must control the number of instances of itself. It is therefore necessary to prevent anyone can instantiate objects using "new ShieldSingleton()".

Therefore the first characteristic of a Singleton type class, is having its constructor as private, so inaccessible from outside the class. In some cases the constructor will even be without implementation, ie empty.

But in this case how to instantiate a ShieldSingleton object ?

This is the second characteristic of a Singleton class: it must provide a public method to obtain the single instance of the class. But even this method must be unique. So it will be declared STATIC, ie it only exists in the class and not in the instantiated object. Call this method getInstance ().

In short it is the class itself that instantiates the single object, and it returns a reference through its public method getInstance (). The reference to the unique object will be materialized in ShieldSingleton class with an attribute also private and static.

It should be noted that the unique object will be instantiated at the first invocation of getInstance().

Here is the corresponding class model:

The reference to the unique object.

The private constructor.

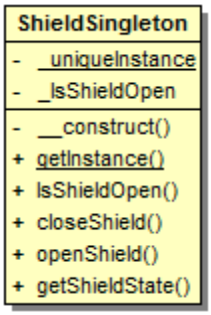
```php
<?php
class ShieldSingleton {

    private static $_uniqueInstance = null;

    // Le constructeur est rendu privé, meme s'il n'est
    // pas implementé.
    private function __construct() {
    }

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

}
?>
```

The method that will provide the reference to the single object.

The code presented here is required for every class to be Singleton type, but generally we can find in these classes, other attributes and methods that are her own.

5.2 Coding

Adapt the design pattern to the problem of our shield, enriching the class in this way:



- _uniqueInstance is a private static attribute: it will exist only in the classroom and not in the object.
- _isShieldOpen returns a boolean. This is a private attribute.
- The private constructor.
- getInstance () is public but static.
- openShield () and closeShield () inverts the value of _isShieldOpen.
- getShieldState () returns "OPEN" and "CLOSE" depending the value of _ isShieldOpen.

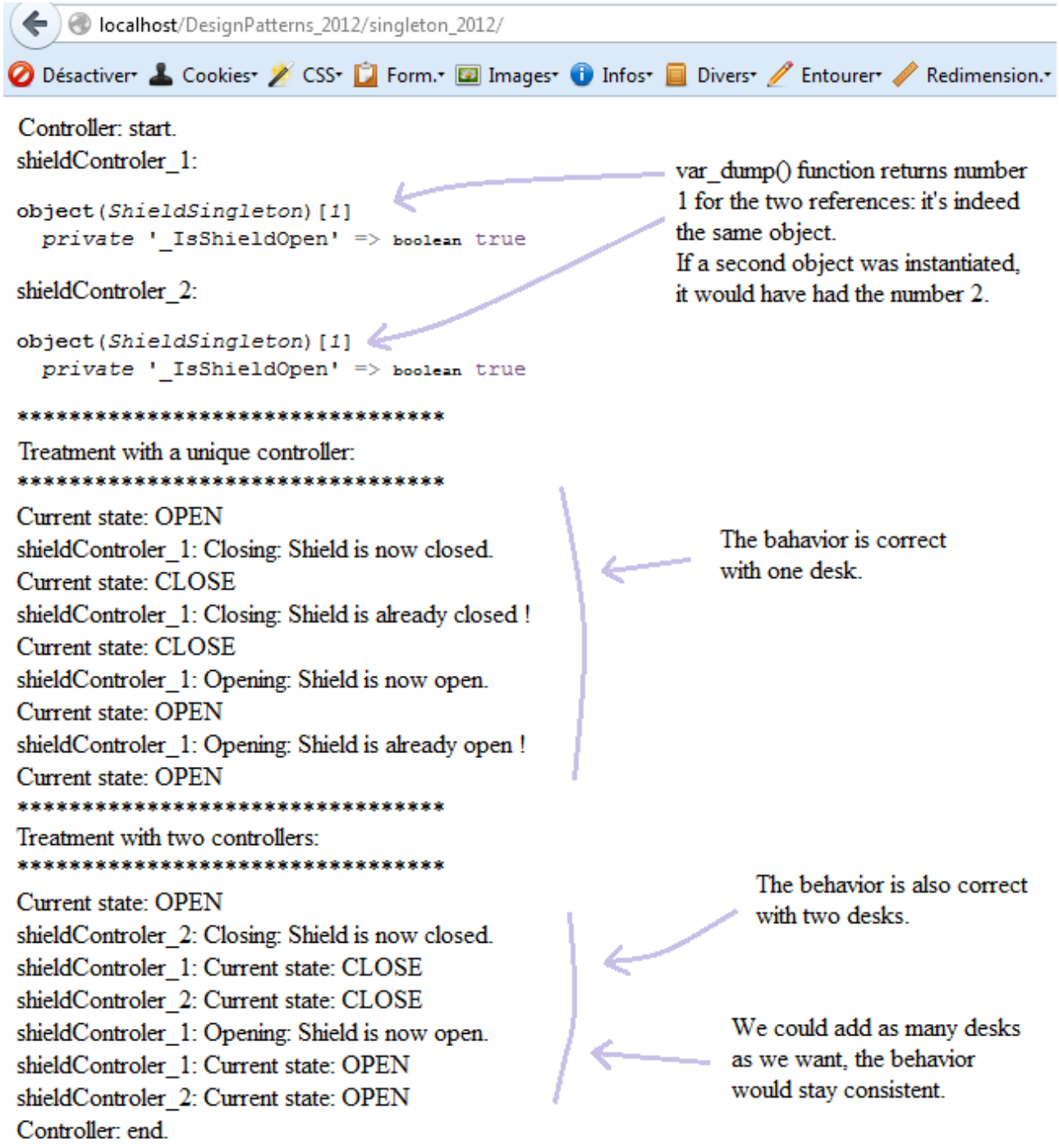
Here is the class:

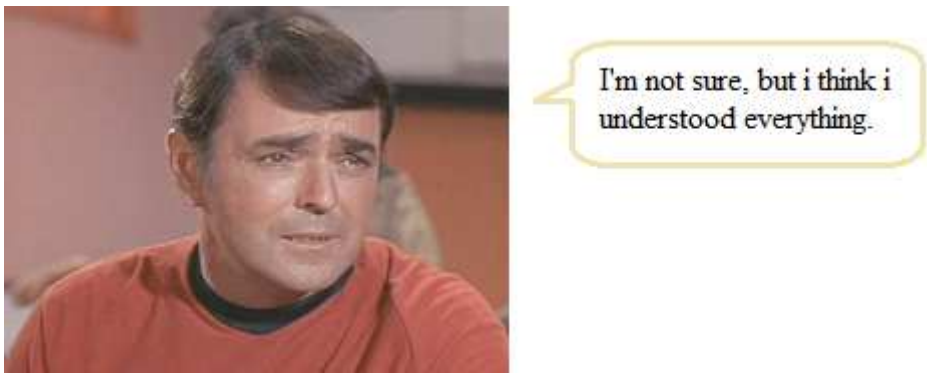
```php
<?php
class ShieldSingleton {

    private static $_uniqueInstance = null;
    private $_IsShieldOpen;

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

    // Public method to get 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>";
        }
    }

}
?>
```

We will use index.php as usual as our controller. The controller will instantiate objects and use them.

```php
<?php

//************************************************************
// SINGLETON pattern controller
//************************************************************

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

echo 'Controller: start.' . $newline;

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

// We verify it's the same object:
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 'Treatment with a unique controller: ' . "</br>";
echo '**********************************' . "</br>";
echo "Current state: " . $shieldControler_1->getShieldState() . "</br>";
echo "shieldControler_1: Closing: ";
$shieldControler_1->closeShield();
echo "Current state: " . $shieldControler_1->getShieldState() . "</br>";
echo "shieldControler_1: Closing: ";
$shieldControler_1->closeShield();
echo "Current state: " . $shieldControler_1->getShieldState() . "</br>";
echo "shieldControler_1: Opening: ";
$shieldControler_1->openShield();
echo "Current state: " . $shieldControler_1->getShieldState() . "</br>";
echo "shieldControler_1: Opening: ";
$shieldControler_1->openShield();
echo "Current state: " . $shieldControler_1->getShieldState() . "</br>";

echo '**********************************' . "</br>";
echo 'Treatment with two controllers: ' . "</br>";
echo '**********************************' . "</br>";
echo "Current state: " . $shieldControler_2->getShieldState() . "</br>";
echo "shieldControler_2: Closing: ";
$shieldControler_2->closeShield();
echo "shieldControler_1: Current state: " .
        $shieldControler_1->getShieldState() . "</br>";
echo "shieldControler_2: Current state: " .
        $shieldControler_2->getShieldState() . "</br>";
echo "shieldControler_1: Opening: ";
$shieldControler_1->openShield();
echo "shieldControler_1: Current state: " .
        $shieldControler_1->getShieldState() . "</br>";
echo "shieldControler_2: Current state: " .
        $shieldControler_2->getShieldState() . "</br>";
echo 'Controller: end.' . $newline;

?>
```

And here is the result of executing:




I'm not sure, but i think i understood everything.

6 Design patterns related to Factory

Design patterns? Yes, there are two. We can even say that there are three, because one of them is not really a design pattern, but we can't ignore.

The concept of Factory is for instantiating objects. Some situations require that instantiation of objects is assigned to a class or group of classes.

In this case, the class wishing instantiate an object will not use the new() operator, but will ask the factory to provide the requested object.

The purpose of this decorrelation is the same in OOP: isolate treatment (here instantiating objects) to make it shareable, maintainable, and allow it to integrate complexity. Indeed instantiation often requires choosing the objects to instantiate, based on the context.

Design patterns related to the concept of Factory meet these constraints by delegating the instantiation of objects to classes which it will be the responsibility.

In the following three chapters we will focus on:

- The "pseudo Factory": it is not a formal design pattern, but a simple instantiation approach, resembling a Factory, which is often used for simple problems.
- Factory Method: design pattern in which the factory is represented by an abstract method.
- Abstract Factory: design pattern in which the factory is materialized by an abstraction which can be an interface or an abstract class. Here we use an interface.

During these three chapters, we will find out how are made those superb uniforms used in the fleet of the Confederation, which includes the Enterprise.

7 PSEUDO FACTORY

Aside from the fact that there are outfits for men, women and there are three colors (yellow, blue, red), these outfits seem trivial.

They don't. These are high-tech products designed to meet the dangers of space travel:

- The fabric gets two treatments: one to mitigate electromagnetic radiation, the other making insensitive the most dangerous acids.
- The garment contains a miniaturized device to know its spatio-temporal position (and that of its occupant).

The manufacturing steps are as follows:

- Assembly parts for sewing.
- Anti electromagnetic radiation treatment.
- Anti Acid treatment.
- Putting the locator.
- Tests of the garment.

Confederation sure wishes that these manufacturing steps are followed, as the production program should have a method like this:

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

Unfortunately there are many types of clothes, so you have to add code to choose the garment to instantiate:

```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
```

Clothe model is passed to the method.

Depending the model, the object is instanciated.

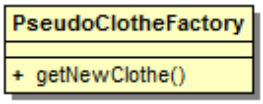
Once we get the object, the standard process is applied.

Here the choice of the object to instantiate is a treatment is subject to evolution: new models of clothes may appear, others may be deleted. In this case it will inevitably change the code of this method.

In contrast, the manufacturing process seems more stable, it will change little or not, and moreover it is independent of the type of clothing.

Design patterns hate to mix variable and stable concepts in the same class.

We will isolate a new class, the code responsible for the selection of the object to instantiate. Call this class PseudoClotheFactory. We will create a getNewClothe() method to obtain a new Clothe object by providing the type of garment we want.
Here is our Factory:



We must of course provide classes for clothes. Whatever garment model, it must implement the following methods:

- assemble()

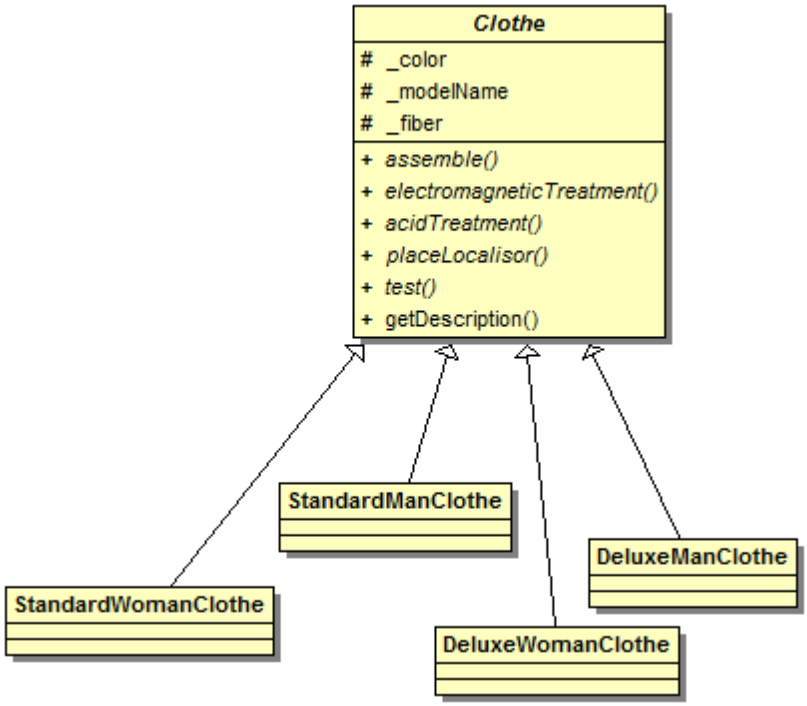
- electromagneticTreatment()
- acidTreatment()
- placeLocalisor()
- test()

And any garment model must know:

- His colour.
- His model name.
- The fiber type: Deluxe models have a more resistant type of fiber.

We also provide a getDescription() method that allows each garment to provide a detailed description.

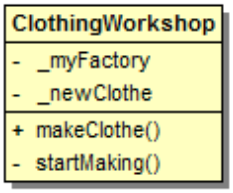
Create an abstract class that require any model of clothing to complete the contract:



Finally, we need a class to represent the entity that decides to make a garment, and will therefore use our Factory. This is somehow the fabrication clothes shop. Call this class ClothingWorkshop and give it a makeClothe() method.

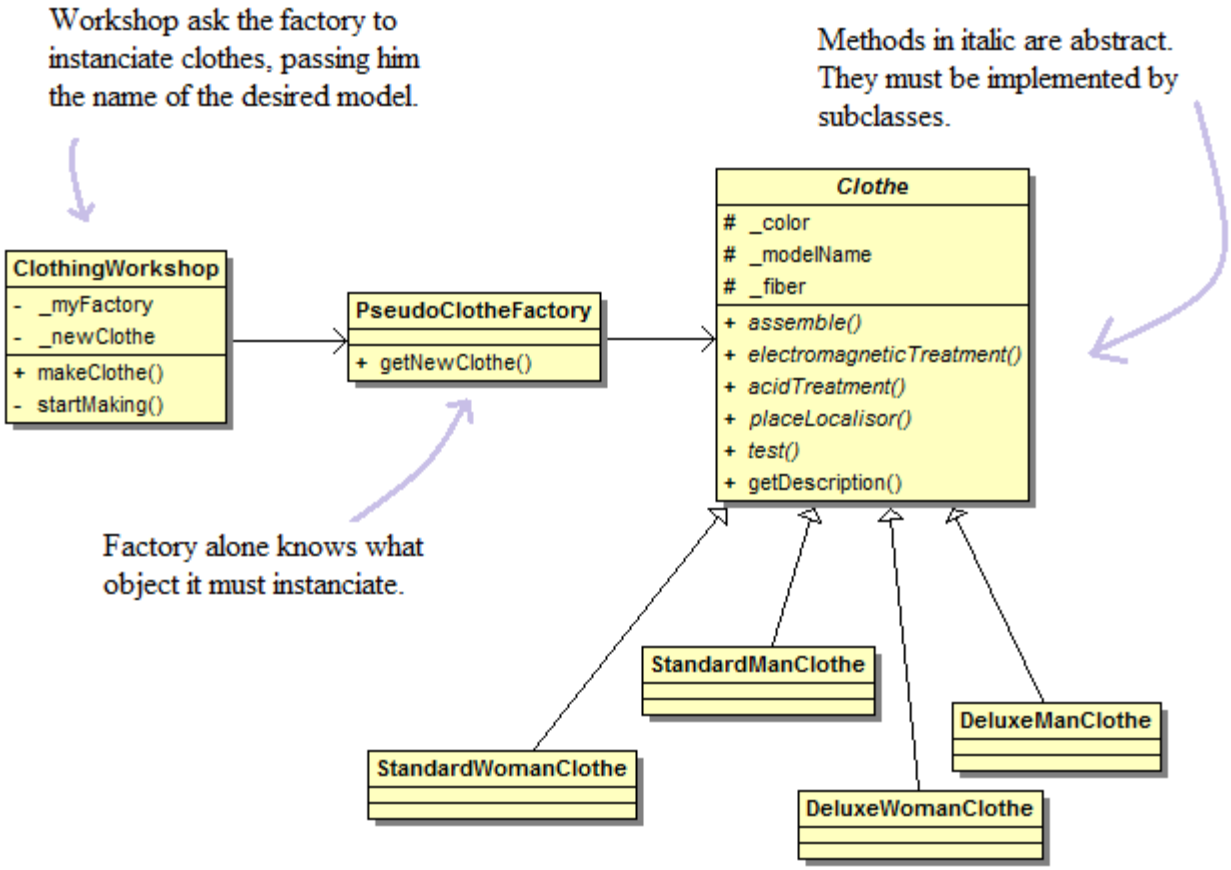
This class will have to know his Factory.

It will also have a reference to the clothing that will be instantiated for her by the Factory.



The makeClothe() method, which takes as parameter a garment model and color, triggers the production of the requested garment using its private method startMaking().

The whole assembly gives us this:



7.1 Coding

Let's start with the abstract class 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 = "  " . 'Model: '  .
                $this->_modelName . "</br>";
        $description .= "  " . 'Colour:  ' .
                $this->_color . "</br>";
        $description .= "  " . 'Fiber: ' .
                $this->_fiber . "</br>";
        return $description;
    }

}

?>
```

The color treatment can be implemented in the abstract class, for it's common to all subclasses.

Abstract methods must be implemented by subclasses.

getDescription() method is also common to all subclasses. So it's implemented here.

Then we have a class for each of the four clothe models:

```php
<?php

class DeluxeManClothe extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_modelName = "Deluxe man suit.";
        $this->_fiber = "Special ultra high resistance.";
    }

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

    public function electromagneticTreatment() {
        echo 'Anti electromagnetic fields treatment.' . "</br>";
    }

    public function acidTreatment() {
        echo 'Anti acid treatment.' . "</br>";
    }

    public function placeLocalisor() {
        echo 'Put localisor.' . "</br>";
    }

    public function test() {
        echo 'Standard test protocol.' . "</br>";
    }

}

?>
```

We call the parent class constructor, then we continue with specific treatment.

That's where we implement abstract methods of the parent class.

```php
<?php

class DeluxeWomanClothe extends Clothe {

    public function __construct($color) {
        parent::  construct($color);
        $this->_modelName = "Deluxe woman suit";
        $this->_fiber = "Special ultra high resistance.";
    }

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

    public function electromagneticTreatment() {
        echo 'Anti electromagnetic fields treatment.' . "</br>";
    }

    public function acidTreatment() {
        echo 'Anti acid treatment.' . "</br>";

    }

    public function placeLocalisor() {
        echo 'Put localisor.' . "</br>";
    }

   public function test() {
       echo 'Standard test protocol.' . "</br>";
   }

}

?>
```

Deluxe models have a special fiber and assembly.

```php
<?php

class StandardManClothe extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_modelName = "Standard man suit.";
        $this->_fiber = "High resistance.";
    }

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

    public function electromagneticTreatment() {
        echo 'Anti electromagnetic fields treatment.' . "</br>";
    }

    public function acidTreatment() {
        echo 'Anti acid treatment.' . "</br>";

    }

    public function placeLocalisor() {
        echo 'Put localisor.' . "</br>";
    }

    public function test() {
        echo 'Standard test protocol.' . "</br>";
    }

}

?>
```

Standard models have high resistance fiber and standard assembly.

```php
<?php

class StandardWomanClothe extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_modelName = "Standard woman suit.";
        $this->_fiber = "High resistance.";
    }

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

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

    public function acidTreatment() {
        echo 'Anti acid treatment.' . "</br>";

    }

    public function placeLocalisor() {
        echo 'Put localisor.' . "</br>";
    }

    public function test() {
        echo 'Standard test protocol.' . "</br>";
    }

}

?>
```

The PseudoClotheFactory class: it only knows what objects must be instantiated:

```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

?>
```

And finally ClothingWorkshop class that is the customer of the 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

?>
```

We pass the Factory to the ClothingWorkshop.

When the workshop wants to create a clothe, he ask Factory to instanciate one, passing it the model and colour.

Then the manufactoring process is applied.

ClothingWorkshop knows the process of making clothes, but it must ask the factory to instantiate clothes objects for it.

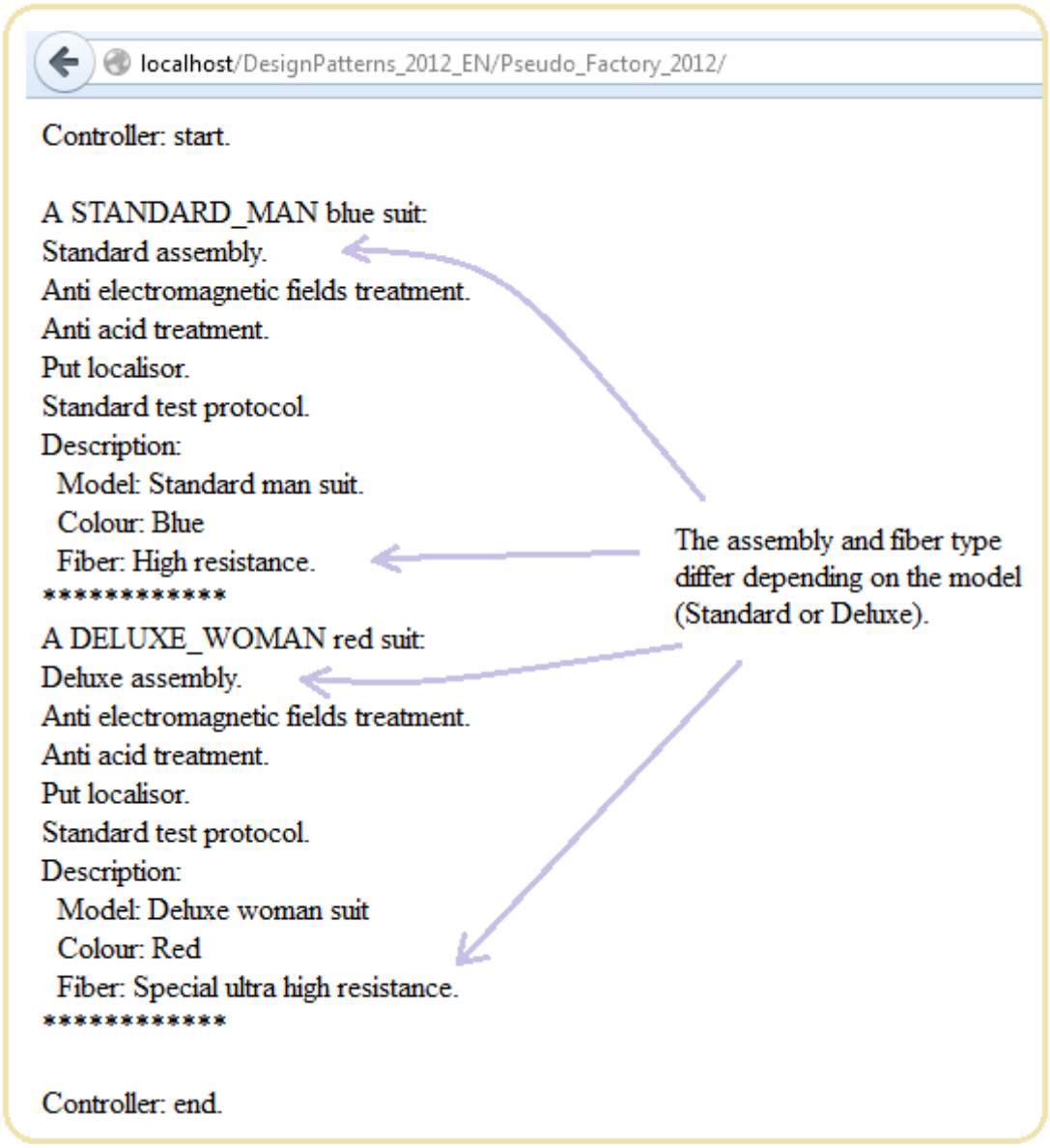
As usual we will use index.php as the controller of the whole design.

Index.php must instantiate a ClothingWorkshop and ask him to make clothes:

```php
<?php

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

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

echo 'Controller: start.' . $newline . $newline;

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

// Treatments:
echo 'A STANDARD_MAN blue suit:' . $newline;
$clothe_1 =  $theWorkshop->makeClothe("STANDARD_MAN","BLU");
echo 'Description: ' . $newline;
echo $clothe_1->getDescription();
echo '************' . $newline;

echo 'A DELUXE_WOMAN red suit:' . $newline;
$clothe_2 = $theWorkshop->makeClothe("DELUXE_WOMAN","RED");
echo 'Description: ' . $newline;
echo $clothe_2->getDescription();
echo '************' . $newline;

echo $newline . 'Controller: end.' . $newline;

?>
```

We must give a Factory to ClothingWorkshop.

And here is the result:



There's nothing new in this design. Objects instanciation choice has simply been moved to the Factory.

Yes we just isolated a variable concept. But in doing so the Factory becomes:

- The unique and sharable access point for clothing items.

- The complexity (here very low) of choosing objects to instantiate, exists only in the Factory. This facilitates maintenance.

As mentioned earlier in this chapter, the PseudoFactory is not a design pattern, but is often used for simple problems.

In the next chapter we will discuss the Factory Méthod pattern which allows the use of Factory by inheritance. So there can be several Factories.

8 FACTORY METHOD PATTERN

In the previous chapter, PseudoFactory allowed us:

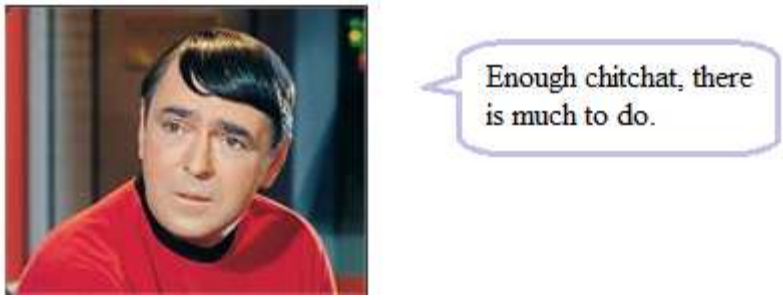
- To control the instantiation of clothes.
- Implement the specificities of Standard and Deluxe ranges in different styles of clothes.

Today Confederation is facing a new problem: some galaxies using their own type of fiber for clothing.

PseudoFactory can not by itself, manage those differences. We would need now a Factory for each galaxy, in order to meet the production process imposed by the Confederation, while allowing specific adaptations in each galaxy.

This problem can be solved by Factory Method and Abstract Factory design patterns.

In this chapter we will use Factory Method, and then in the next chapter we will go further with Abstract Factory.



Well, consider two galaxies, ours: the Milky Way, and a nearby galaxy NGC1313 (15 million light years).

Milkyway uses the official fiber to make clothes, while NGC1313 uses a special fiber called arachno-fiber extremely resistant.

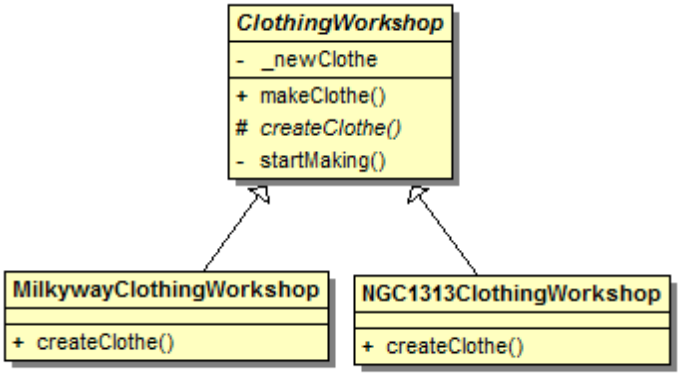
Moreover, as there are now several galaxies producing clothes, we must know the origin of each.

Finally, what we had with the Pseudo Factory remains: the manufacturing process must be the same in all galaxies, and there is always the following differences between Standard and Deluxe ranges of clothing:

- Assembly: Standard or Deluxe.
- Fiber: Standard or Deluxe (regardless of whether it is Milkyway or NGC1313 fiber type).

The principle of the Factory Method is simple: in a parent class, we define an abstract method that is not implemented, and it is left to subclasses to implement it according to their specificity. This abstract method: it is the Factory Method. It will play the role of Factory in subclasses, and will be responsible for instantiating Clothe objects.
Each subclass will of course be specialized for a Galaxy.

Create the parent class and two subclasses, one for each galaxy:



ClothingWorkshop is an abstract class (its name is in italics). It also has an abstract method createClothe() to be implemented by subclasses. This is the Factory Method.

The two subclasses will implement createClothe() so that it can create the clothes that meet each galaxy requirements.

The Factory is somehow integrated into the Workshop. It was completely outside with Pseudo Factory.

The makeClothe() method is invoked to ask the Workshop to produce a clothe, passing it as a parameter, the desired model name and color. This method is common to all galaxies, so it is implemented in the abstract class.

And it is makeClothe() which will invoke createClothe() to get a clothe object corresponding to the galaxy.

StartMaking() is a private method, used internally by the Workshop to make clothing. This method represents the manufacturing protocol that must be common to all galaxies, so it is implemented in

the abstract class.

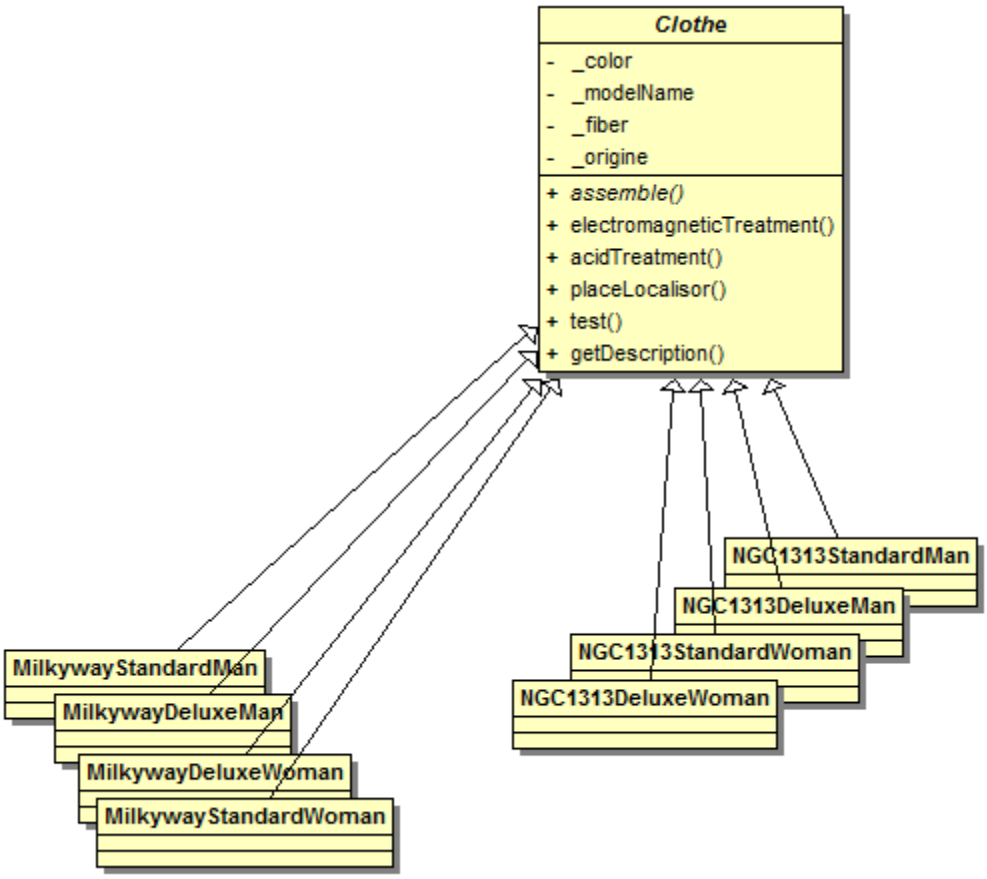
Now for clothing:

The design used in Pseudo Factory can be reused with the following changes:

- We add an “origin” attribute to Clothe class: all clothing needs to know its original Galaxy.
- The assemble() method is abstract: each clothe will implement depending its range (Standard or Deluxe).
- As there is now an abstract method, the class also becomes abstract.

We must of course create two groups each containing a clothing range for the Galaxy.

We ge this:



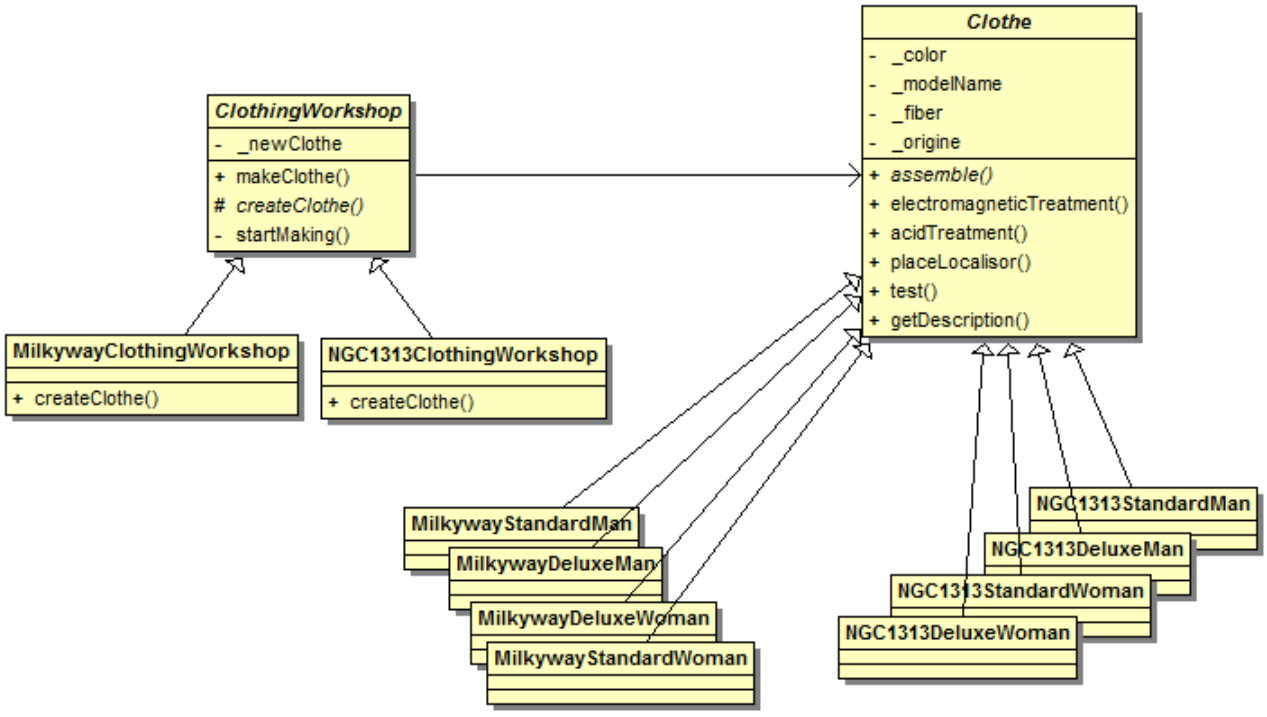
In Clothe class, we find the attributes:

- Colour.
- Clothe name.

- Fiber type.
- Origin.

The methods are the same as in Pseudo Factory since the steps of the manufacturing process have not changed.

The complete design is as follow:



How does it work ?

- To make a clothe we invoque makeClothe() method on the workshop concerned (MilkywayClothingWorkshop or NGC1313ClothingWorkshop), passing it the model and the desired color.
- makeClothe() invokes the protected createClothe() method, passing it the model and color.
- createClothe(), which is the Factory Method, will instantiate the clothe of the galaxy, according to the model requested, and return this object to makeClothe().
- Finally makeClothe() invokes startMaking() which will implement the various steps of the manufacturing process.

All is said. It only remains to code.

8.1 Coding

The abstract class:

```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);
}

?>
```

We call the Factory Method.

We apply the manufacturing process.

Factory Method implemented in subclasses.

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;
  }

}

?>
```

That's where we choose the clothe to instanciate.

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 = "Yellow";
                break;
            case "RED":
                $this->_color = "Red";
                break;
            case "BLU":
                $this->_color = "Blue";
            default :
                $this->_color = "Blue";
        }  // end switch
    }

    abstract function assemble();

    public function electromagneticTreatment() {
        echo 'Anti electromagnetic fields treatment.' . "</br>";
    }

    public function acidTreatment() {
        echo 'Anti acid treatment.' . "</br>";
    }

    public function placeLocalisor() {
        echo 'Put localisor.' . "</br>";
    }

    public function test() {
        echo 'Test protocol.' . "</br>";
    }

    public function getDescription() {
        $description = "  " . 'Model: ' .
                $this->_modelName . "</br>";
        $description .= "  " . 'Colour: ' .
                $this->_color . "</br>";
        $description .= "  " . 'Fiber type: ' .
                $this->_fiber . "</br>";
        $description .= "  " . 'Origin: ' .
                $this->_origine . "</br>";
        return $description;
    }

}

?>
```

MilkywayDeluxeMan:

```php
<?php

class MilkywayDeluxeMan extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_fiber = "Special ultra high resistance.";";
        $this->_modelName = "Deluxe man suit.";
        $this->_origine = "Milkyway.";
    }

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

}

?>
```

Each clothe defines his fiber, his name and origin.

assemble() method is implemented in each clothe class.

MilkywayDeluxeWoman:

```php
<?php

class MilkywayDeluxeWoman extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_fiber = "Special ultra high resistance.";
        $this->_modelName = "Deluxe woman suit.";
        $this->_origine = "Milkyway.";
    }

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

}

?>
```

MilkywayStandardMan:

```php
<?php

class MilkywayStandardMan extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_fiber = "High resistance.";
        $this->_modelName = "Standard man suit.";.";
        $this->_origine = "Milkyway.";
    }

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

}

?>
```

MilkywayStandardWoman:

```php
<?php

class MilkywayStandardWoman extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_fiber = "High resistance.";
        $this->_modelName = "Standard woman suit.";;
        $this->_origine = "Milkyway.";
    }

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

}

?>
```

NGC1313DeluxeMan:

```php
<?php

class NGC1313DeluxeMan extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_fiber = "Arachno-fiber ultra high resistance.";
        $this->_modelName = "Deluxe man suit.";
        $this->_origine = "NGC1313.";
    }

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

}

?>
```

NGC1313DeluxeWoman:

```php
<?php

class NGC1313DeluxeWoman extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_fiber = "Arachno-fiber ultra high resistance.";
        $this->_modelName = "Deluxe woman suit.";
        $this->_origine = "NGC1313.";
    }

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

}

?>
```

NGC1313StandardMan:

```php
<?php

class NGC1313StandardMan extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_fiber = "Arachno-fiber high resistance.";;
        $this->_modelName = "Standard man suit.";
        $this->_origine = "NGC1313.";
    }

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

}

?>
```

NGC1313StandardWoman:

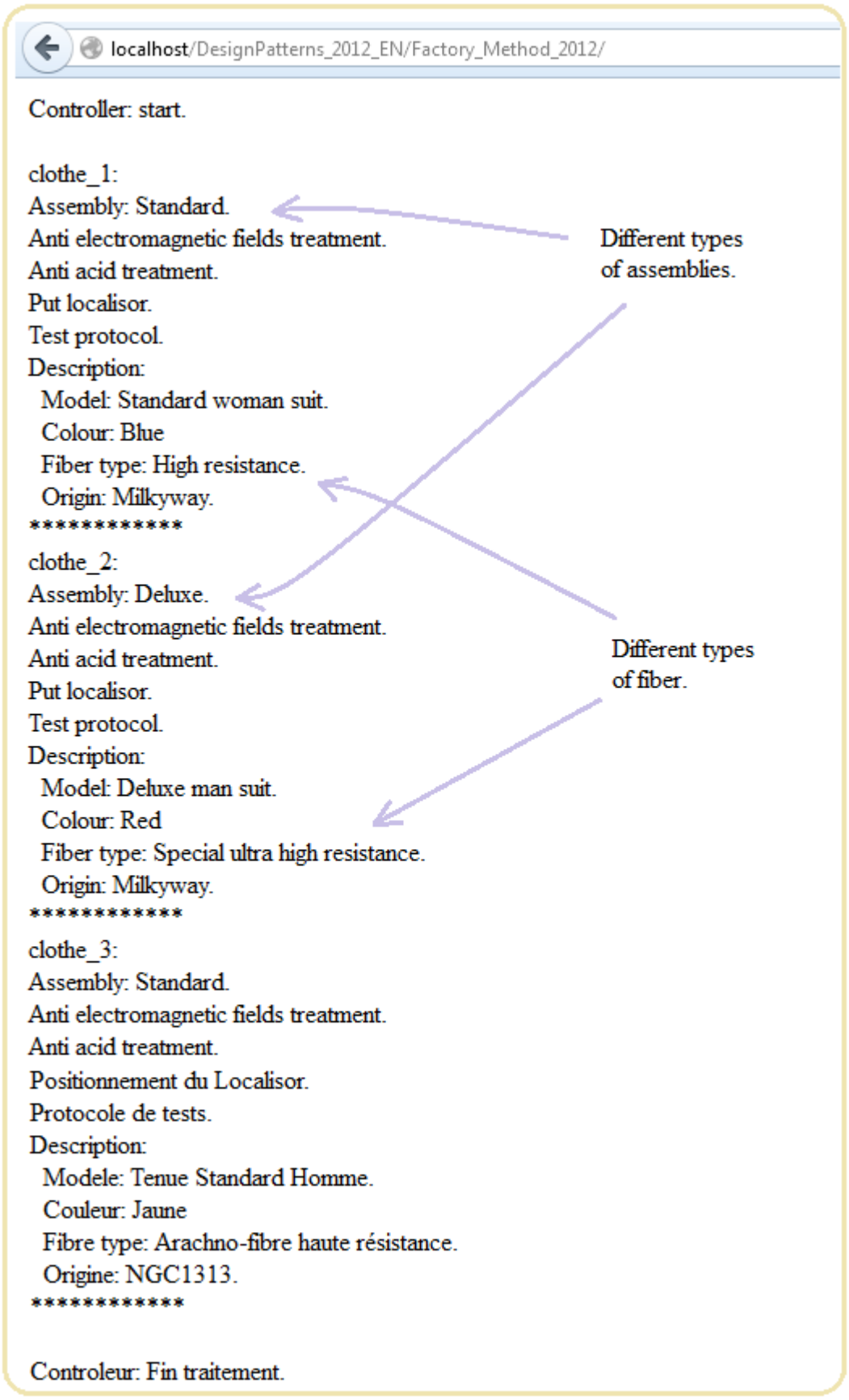
```php
<?php

class NGC1313StandardWoman extends Clothe {

    public function __construct($color) {
        parent::__construct($color);
        $this->_fiber = "Arachno-fiber high résistance.";:
        $this->_modelName = "Standard woman suit.";
        $this->_origine = "NGC1313.";
    }

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

}

?>
```

8.2 Tests

We will use index.php as the controller of all of our design:

```php
<?php

//*************************************************************
// FACTORY METHOD pattern controller
//*************************************************************

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

echo 'Controller: start.' . $newline . $newline;

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


// Treatments:
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 . 'Controller: end.' . $newline;
?>
```

The result of executing gives us this:



In conclusion we can say that from the controller, the design is fairly simple to use because you just ask for a clothe to a workshop to get it completely fabricated. It remains only to ask his description with getDescription() method.

9 ABSTRACT FACTORY PATTERN

In the previous chapter (Factory Method) we left some initiatives to clothes. In fact they were responsible for determining the following attributes:

- Fiber type.
- Model name.
- Origin.
- Assembly type.

As for the test protocol, it was left to the workshop responsibility and we had no control over this element.

Confederation therefore wishes harden production control, banning clothes and workshops to define or select elements by themselves.

The following decisions were taken:

- The fiber type will be forced according to the clothe galaxy and range.
- The origin will be forced to clothes according to Galaxy.
- The type of assembly will be forced according to the clothe range.
- The test protocol will be forced according to the clothe Galaxy.
- The type of localisor will be forced to clothes according to Galaxy.

The only attribute that continues to be defined by the clothe is the model name, ie his own name.

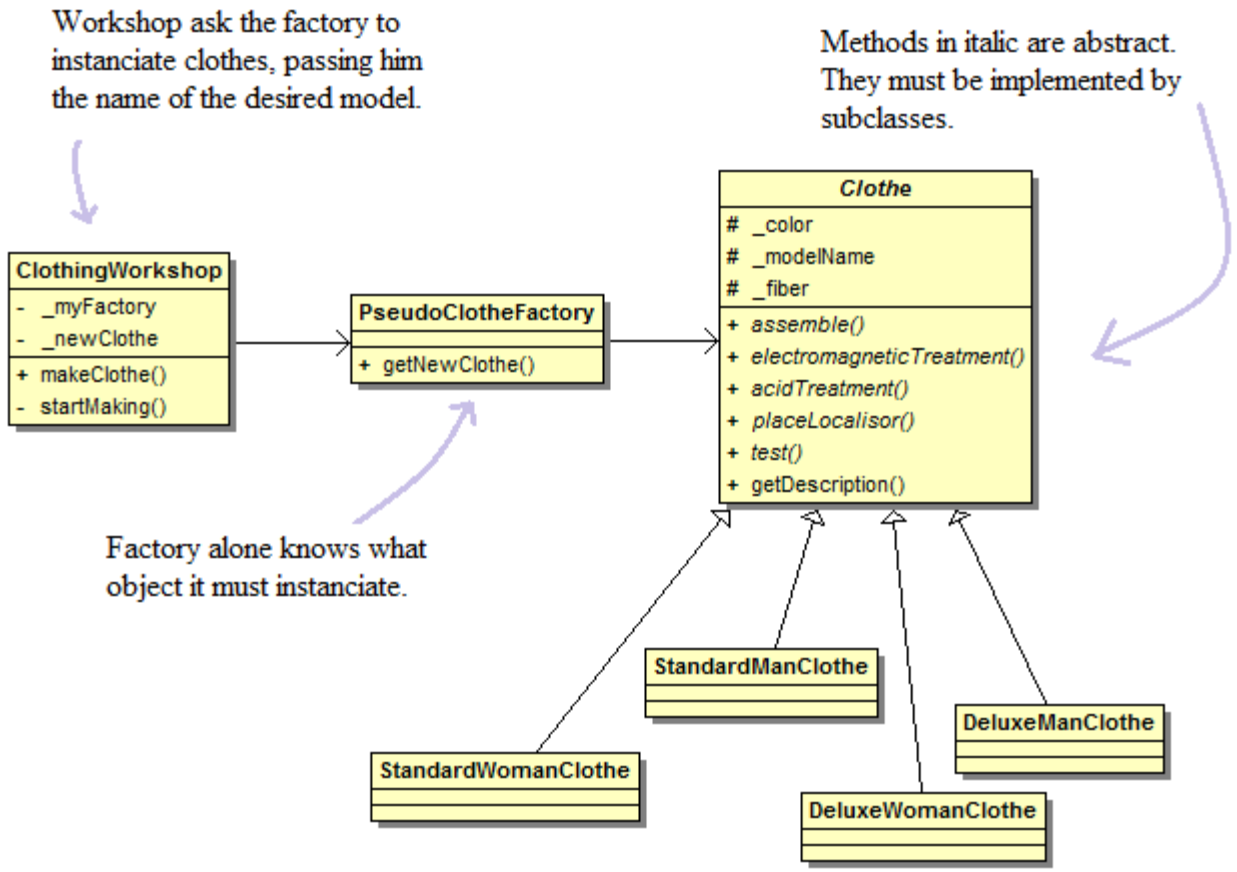
These constraints seem complex to implement. They don't.
With the concept of Factory, and provided that the design is done right, it's simple.

We can adapt our Factory Method to integrate these new constraints. But take this opportunity to completely redo our design using the Abstract Factory pattern.

The essential difference between the two patterns is that Abstract Factory, being external, can be used by any application class, whereas Factory Method, as a method, should be seen more as a dedicated private service to the class that implements this method.

But before starting work, a few words about this pattern.

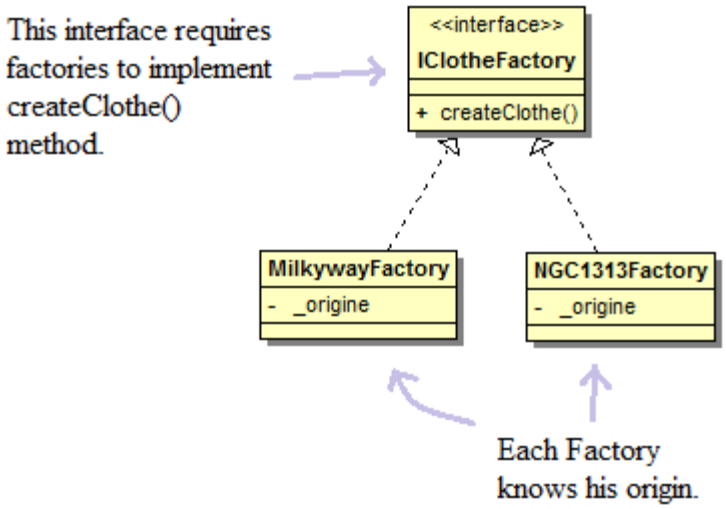
In the chapter Pseudo Factory, we had the following design:



The Factory was external to ClothingWorkshop.

We can consider that the Abstract Factory pattern is an extension of PseudoFactory because we simply allow the existence of several Factories, but by imposing a common interface, in which we define a simple createClothe() method that will be responsible for the instantiation of all that is needed to make the clothe, taking into account all the constraints mentioned.

We get this:

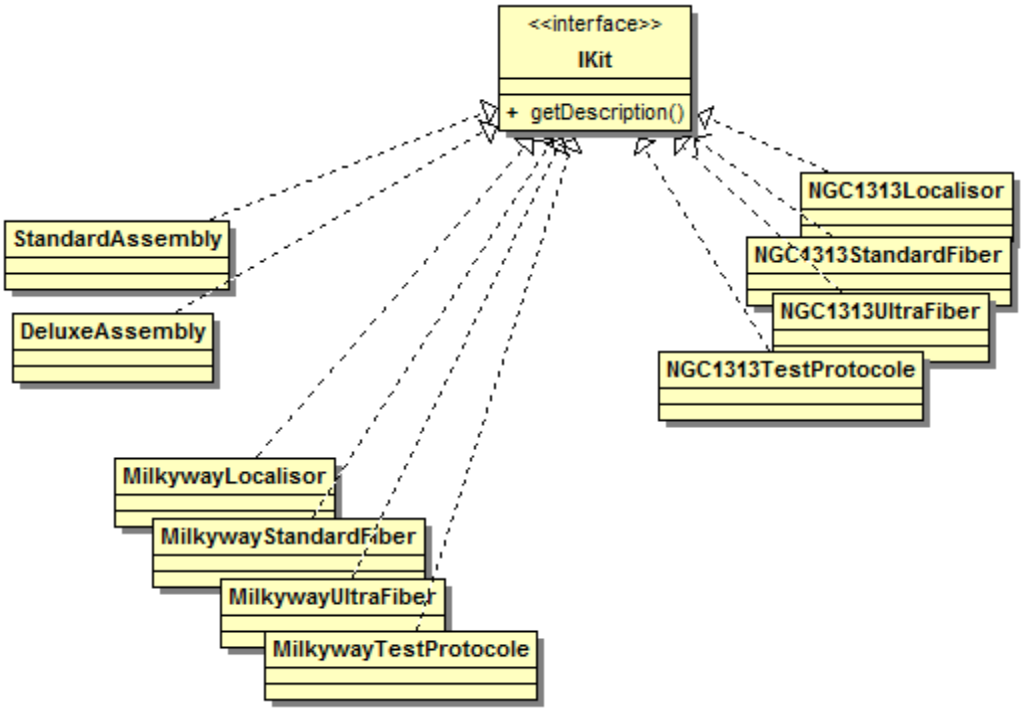


Let's see now the elements needed to make the clothe. As indicated earlier in this chapter, we must impose the following to clothe:

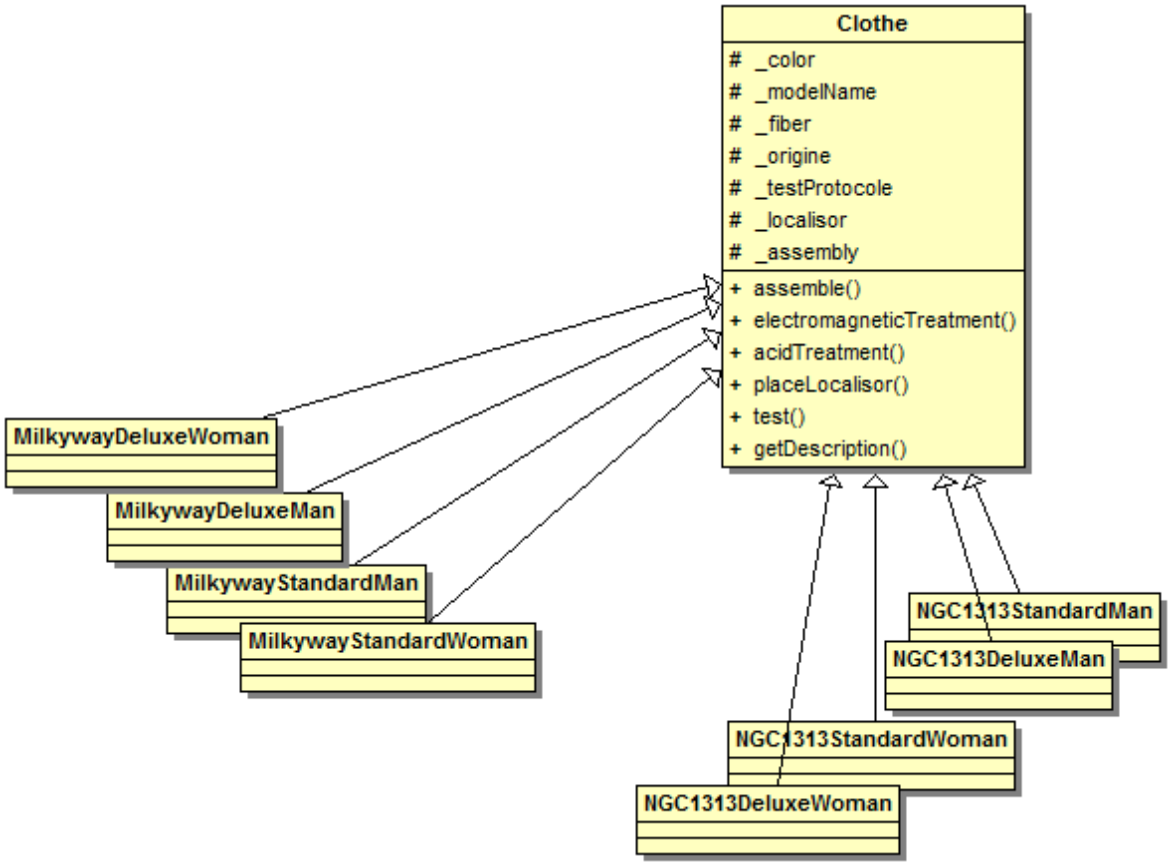
- Fibre type.
- Origin.
- Assembly type.
- Test protocole.
- Localisor type.

These elements form a kind of construction kit, and it is understood that our factory will be in charge of choosing the right components of the kit, depending the context.
This means that we have classes for kit components. We have one thing to ask an element: his description. We therefore expect a getDescription() method.

Like all kit components should implement this method, we impose itvia a small interface. Which gives us:

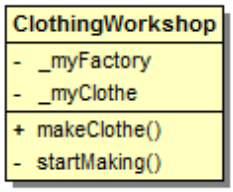


All kit components are specific to each galaxy, except the assembly type that depends only on the range (Standard or Deluxe).
About clothes: little changes from Factory Method:



Clothe class has all its attributes, so it is able to describe itself completely, but it not defines its own attributes. As agreed, this class has no more initiative. We simply ask it to run its manufacturing stages.

The workshop is also slightly modified:

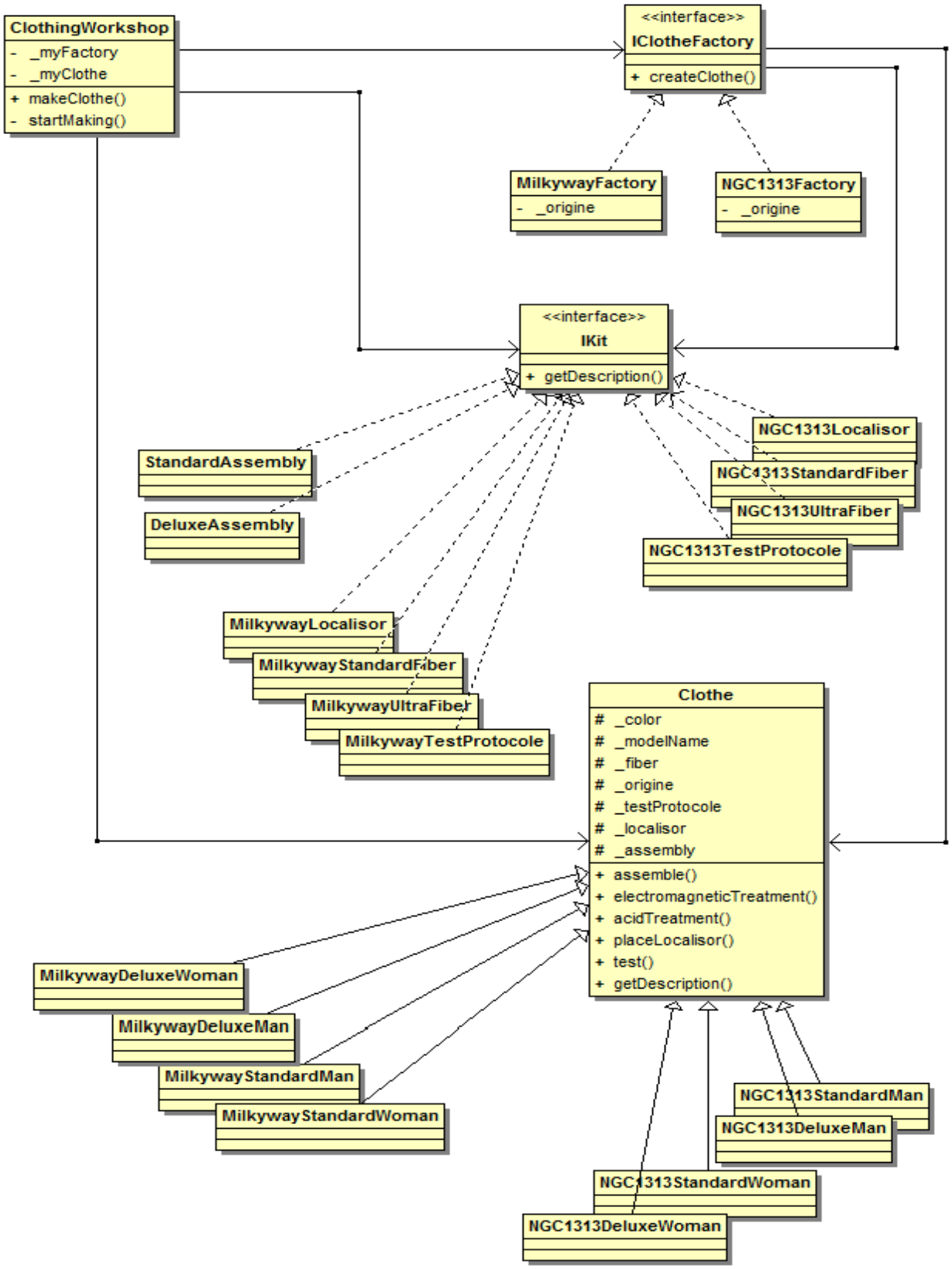


It has a reference to the Factory it is assigned.

The makeClothe() method is used to ask the workshop to produce a clothe.

startMaking() is the method that knows the manufacturing process.

The complete design gives us this:



Principle of operation:

- The controller (not shown in the class diagram) invokes ClothingWorkshop.makeClothe() method to make a clothe, passing the model and color.
- MakeClothe() invokes the factory that will instantiate the clothing and components kit, and return it to the workshop.
- The workshop invoke its private method startMaking() which will apply manufacturing steps to the clothe.
- Finally, the clothe made is returned to the controller.

9.1 Coding

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();
    }

}

?>
```

We have to pass a Factory to the Workshop constructor.

We use the Factory to get the clothe.

This is where the manufacturing process is.

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;
    }

}

?>
```

Each Factory defines his origin.

The Factory knows what to instanciate according the context.

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-fiber ultra high resistance.';
    }

}

?>
```

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-fiber high resistance.';
    }

}

?>
```

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 standard fiber high 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 standard fiber ultra high 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 = "Yellow";
                break;
            case "RED":
                $this->_color = "Red";
                break;
            case "BLU":
                $this->_color = "Blue";
            default :
                $this->_color = "Blue";
        }  // end switch
    }

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

    public function electromagneticTreatment() {
        echo 'Anti electromagnetic fields treatment.' . "</br>";
    }

    public function acidTreatment() {
        echo 'Anti acid treatment.' . "</br>";
    }

    public function placeLocalisor() {
        echo 'Put localisor: ' .
                $this->_localisor->getDescription();
    }

    public function test() {
        echo 'Tests protocol: ' .
                $this->_testProtocole->getDescription();
    }

    public function getDescription() {
        $description = "  " . 'Model: ' .
                $this->_modelName . "</br>";
        $description .= "  " . 'Color: ' .
                $this->_color . "</br>";
        $description .= "  " . 'Fiber type: ' .
                    $this->_fiber->getDescription() . "</br>";
        $description .= "  " . 'Origin: ' .
                    $this->_origine . "</br>";
        $description .= "  " . 'Assembly: ' .
                    $this->_assembly->getDescription();
        $description .= "  " . 'Tests protocol: ' .
                    $this->_testProtocole->getDescription();
        return $description;
    }

}

?>
```

The Factory provide Clothe all the elements of the manufactoring kit.

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 = "Standard man suit.";

    }

}

?>
```

we call the superclass constructor.

... and we define the clothe model name, unique attribute defined by clothe himself.

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 = "Deluxe woman suit.";

    }

}

?>
```

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 = "Standard woman suit.";

    }

}

?>
```

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 = "Deluxe man suit.";
    }

}

?>
```

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 = "Standard woman suit.";

    }

}

?>
```

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 = "Standard man suit.";

    }

}

?>
```

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 = "Deluxe woman suit.";

    }

}

?>
```

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 = "Deluxe man suit.";

    }

}

?>
```

9.2 Tests

As usual we will use index.php as the controller of the whole design:

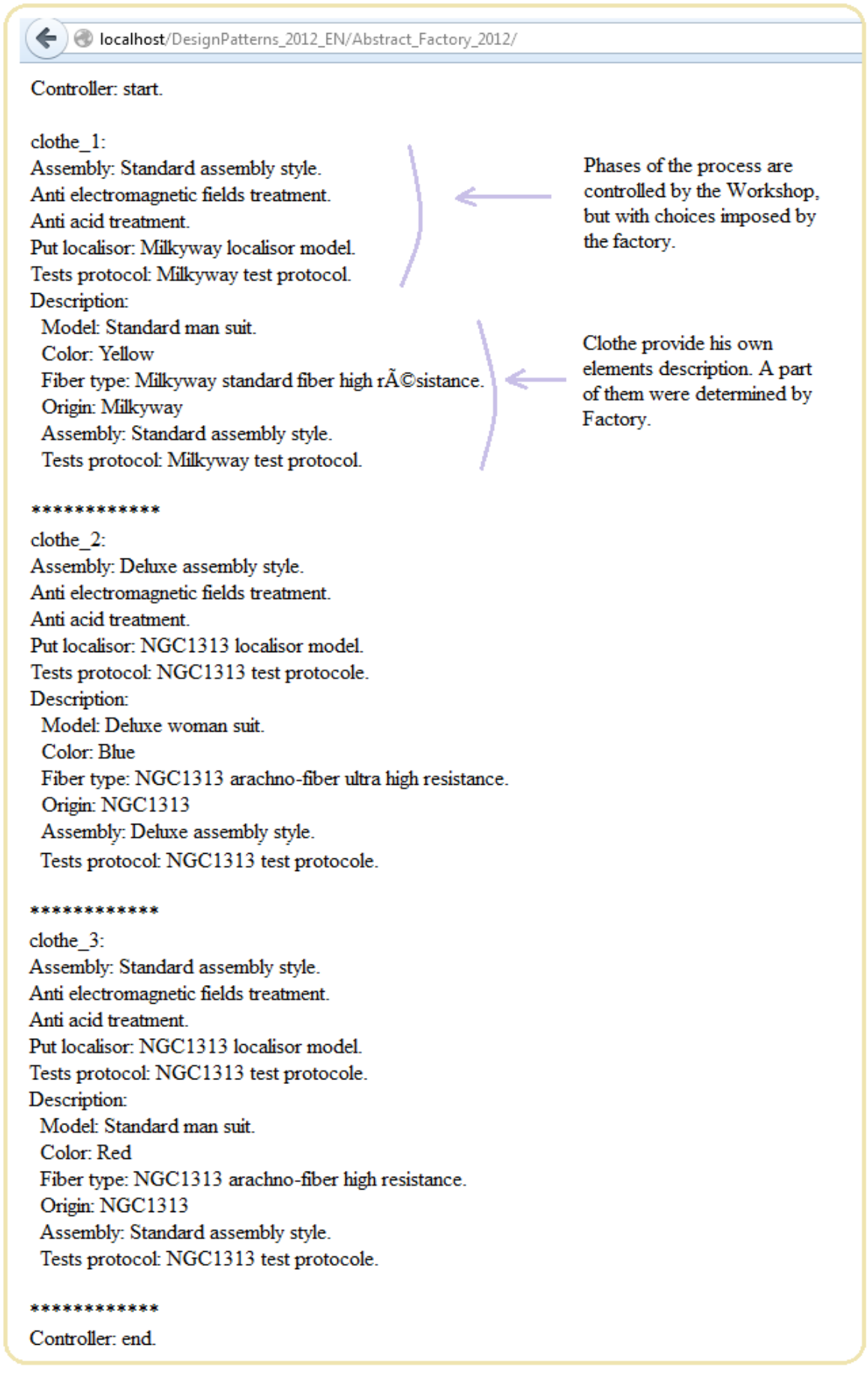
```php
<?php

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

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

echo 'Controller: start.' . $newline . $newline;

// Factory instanciations:
$myMilkyWayFactory = new MilkywayFactory;
$myNGC1313Factory = new NGC1313Factory;

// Workshops instanciations:
$myMilkyWayWorkshop = new ClothingWorkshop($myMilkyWayFactory);
$myNGC1313Workshop = new ClothingWorkshop($myNGC1313Factory);

// Treatements:
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 'Controller: end.';

?>
```

Factory instanciation.

Workshop instanciation, passing each of them his Factory.

Clothe creation is very simple.

Here is the result:


localhost/DesignPatterns_2012_EN/Abstract_Factory_2012/
Controller: start.
clothe_1:
Assembly: Standard assembly style.
Anti electromagnetic fields treatment.
Anti acid treatment.
Put localisor: Milkyway localisor model.
Tests protocol: Milkyway test protocol.
Description:
Model: Standard man suit.
Color: Yellow
Fiber type: Milkyway standard fiber high rÃ©sistance.
Origin: Milkyway
Assembly: Standard assembly style.
Tests protocol: Milkyway test protocol.
Phases of the process are controlled by the Workshop, but with choices imposed by the factory.
Clothe provide his own elements description. A part of them were determined by Factory.
****
clothe_2:
Assembly: Deluxe assembly style.
Anti electromagnetic fields treatment.
Anti acid treatment.
Put localisor: NGC1313 localisor model.
Tests protocol: NGC1313 test protocole.
Description:
Model: Deluxe woman suit.
Color: Blue
Fiber type: NGC1313 arachno-fiber ultra high resistance.
Origin: NGC1313
Assembly: Deluxe assembly style.
Tests protocol: NGC1313 test protocole.
****
clothe_3:
Assembly: Standard assembly style.
Anti electromagnetic fields treatment.
Anti acid treatment.
Put localisor: NGC1313 localisor model.
Tests protocol: NGC1313 test protocole.
Description:
Model: Standard man suit.
Color: Red
Fiber type: NGC1313 arachno-fiber high resistance.
Origin: NGC1313
Assembly: Standard assembly style.
Tests protocol: NGC1313 test protocole.
****
Controller: end.

With this new system, they will able to produce sometimes new collections

Stop chattering. Now i would like a real mission.

10 COMMAND PATTERN

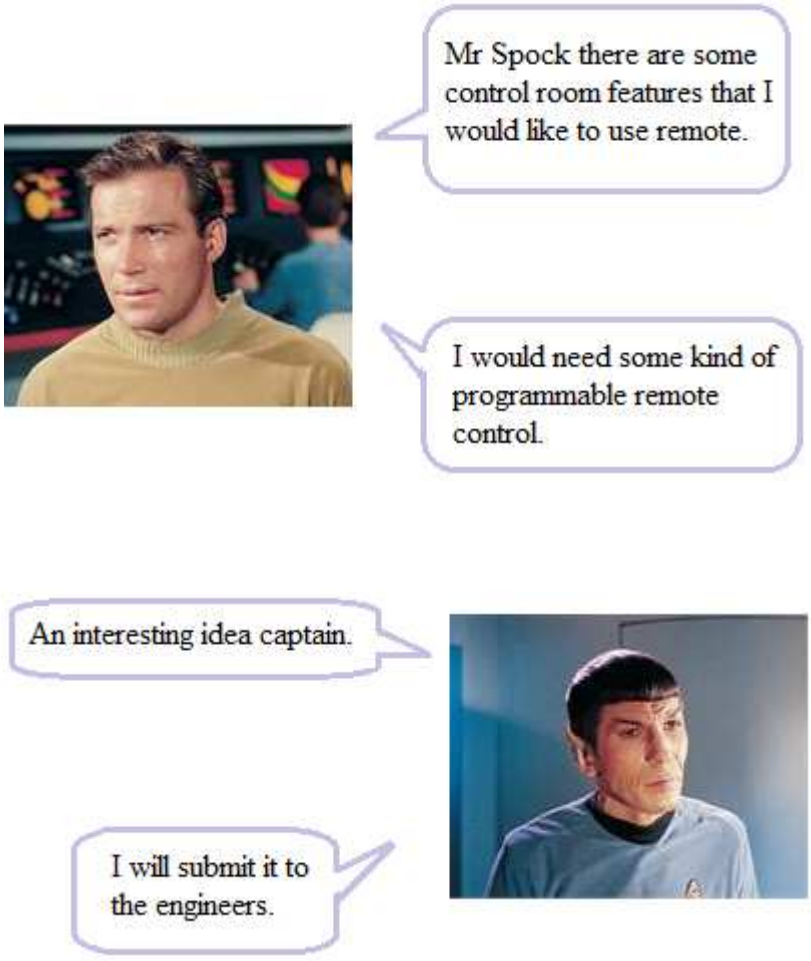
Generally, a request has an action that is always the same. It is a strong and enduring relationship, therefore considered static.
But there are cases where this link must be changed according to context. For example:

- A request must trigger another action than the one initially planned.
- A request must trigger not one, but several actions.

The Command pattern in decorrelating requests and actions, will allow to easily integrate this kind of constraint, adding intelligence between the two.

To summarize we can see this pattern as a programmable remote control: each button triggers one or more actions, each button can be reprogrammed at any time.

It only remains to wait for the next opportunity to use this wonderful pattern.



Here's what the captain wishes to remote control:

- The Enterprise propulsion speed: on a scale of 0-5 (5 being the hyper-speed).

- The Shield (open / closed).
- The invisibility of the Enterprise (on / off).
- The display of the control room (on / off).

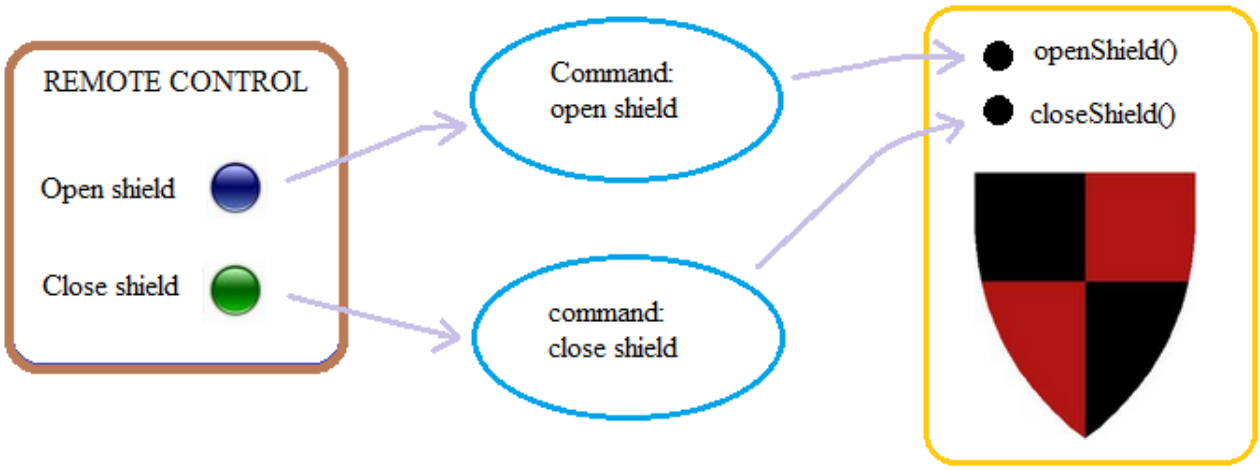
To understand how does the Command pattern work, for example, take the shield of the Enterprise).

Imagine a 2-buttons remote control:

- A button triggers the opening of the shield.
- A button triggers closure of the shield.

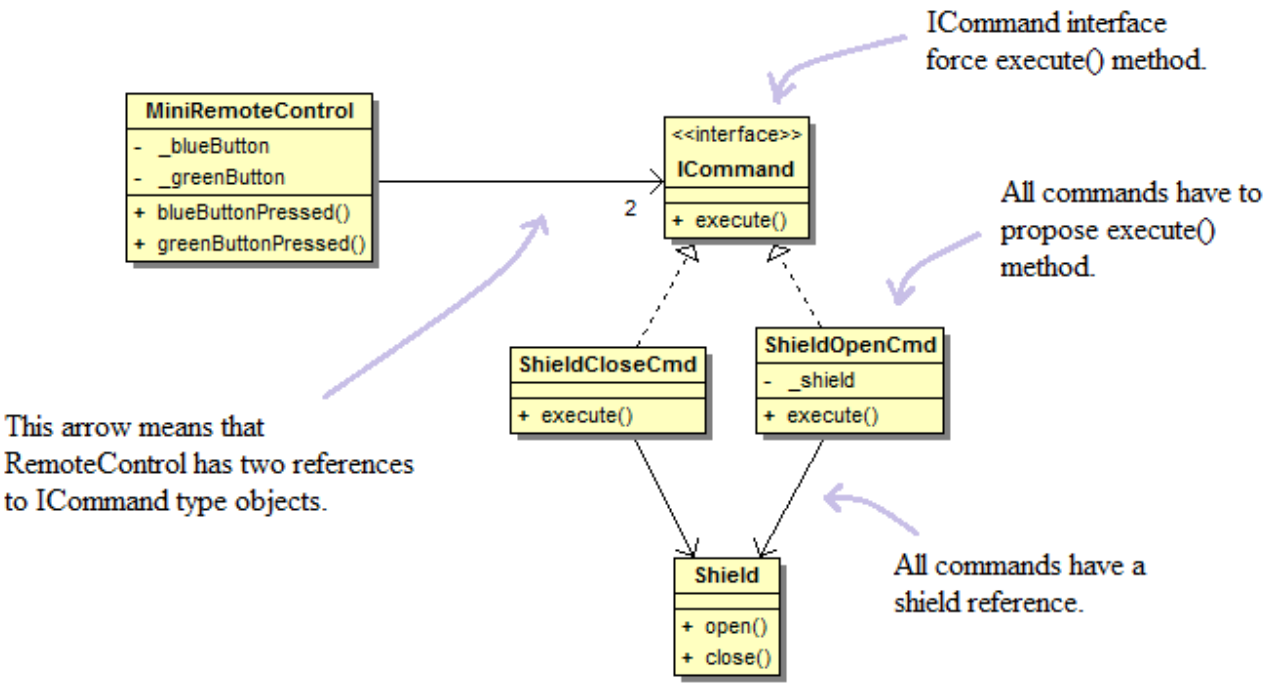
The game is played with three players:

- The remote control.
- The command (ie the action triggered).
- The controlled device (here the shield)



- Each button on the remote control is connected to one and only one command.
- Each command knows what device it must control and what action it should trigger on the device.
- The remote control has no knowledge of the device controlled.

The principle is simple. Let's see how to turn this into class model



MiniRemoteControl:

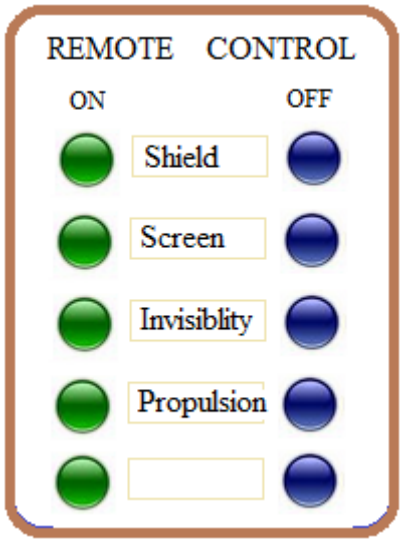
- The buttons are represented by two private attributes _blueButton and _greenButton.
- Each button is linked to a ICommand type object.
- To press a button it will invoke blueButtonPressed () or greenButtonPressed ().

Let's see what happens if you press the blue button, which corresponds to the opening of the shield:

- The triggering event is the invocation of blueButtonPressed() method.
- The blueButtonPressed() method in his turn invokes the execute() method of the blue button _blueButton. Indeed _blueButton is simply a reference to a ICommand object, and more precisely a ShieldOpenCmd object.
- This is actually the execute() method of the ShieldOpenCmd command that was executed. This method works directly on the shield by invoking the open() method.

The basic principle is simple. Adapt now at the of Captain request.

Here is what our remote control should look like:



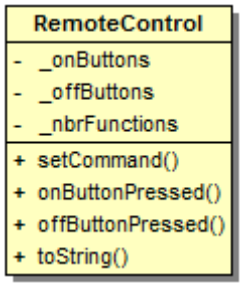
We can make the following remarks:

- There may be unused buttons on the remote control. We will provide this.
- The drive is not a control “on/off” type, but may take a value on a scale of 0 to 4. It will require a special treatment.
- We will need something to manage links between buttons and controls.

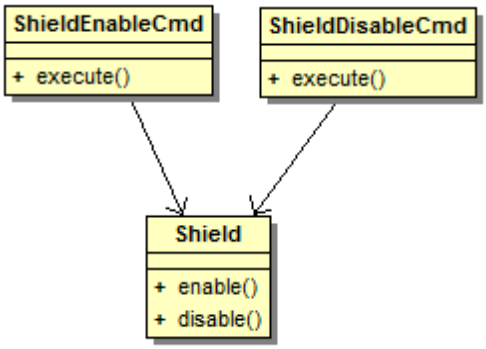
We will enrich our RemoteControl class:

- To manage the links buttons/commands, we will use two arrays: onButtons and offButtons. These arrays will contain references to the ICommand objects for each button.
- We will have an attribute that indicate the number of functions the remote can handle (ie the number of pairs of “on/off” buttons).
- We will need a method to assign a command to a button. Call it setcommand(). This method will allow us to program or reprogram the remote control.
- Finally, we must be able to handle the push of a button. To do this we add two public methods onButtonPressed() and offButtonPressed(). Each will take as a parameter the index of the button in the arrays.
- As a tool we add a toString() method that displays the configuration of the remote control.

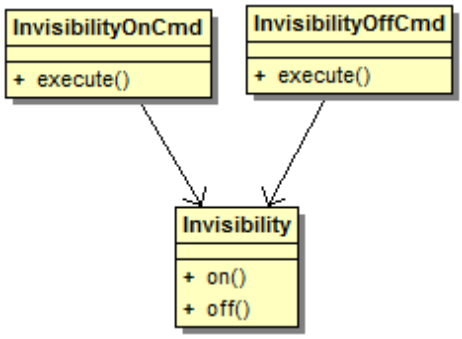
Our RemoteControl class becomes:



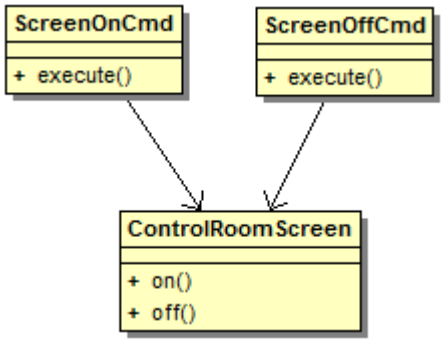
The two commands that control the shield:



The two commands that control the invisibility:



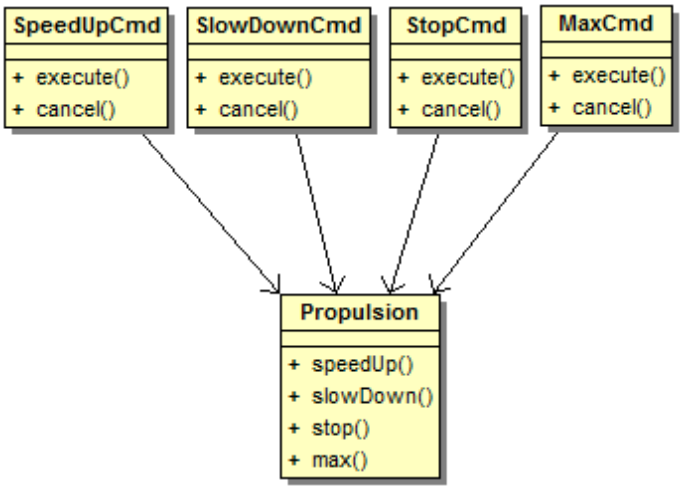
The two commands that control the display of the control room:



On the speed of the Enterprise, the Engine class provides four methods to us:

- speedUp(): increases the speed of a level.
- slowDown(): decreasing the speed of a level.
- stop() : decreases the speed repeatedly until stop.
- max(): Increases the speed several times until you reach the maximum speed.

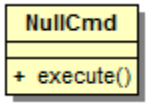
So we create four commands.



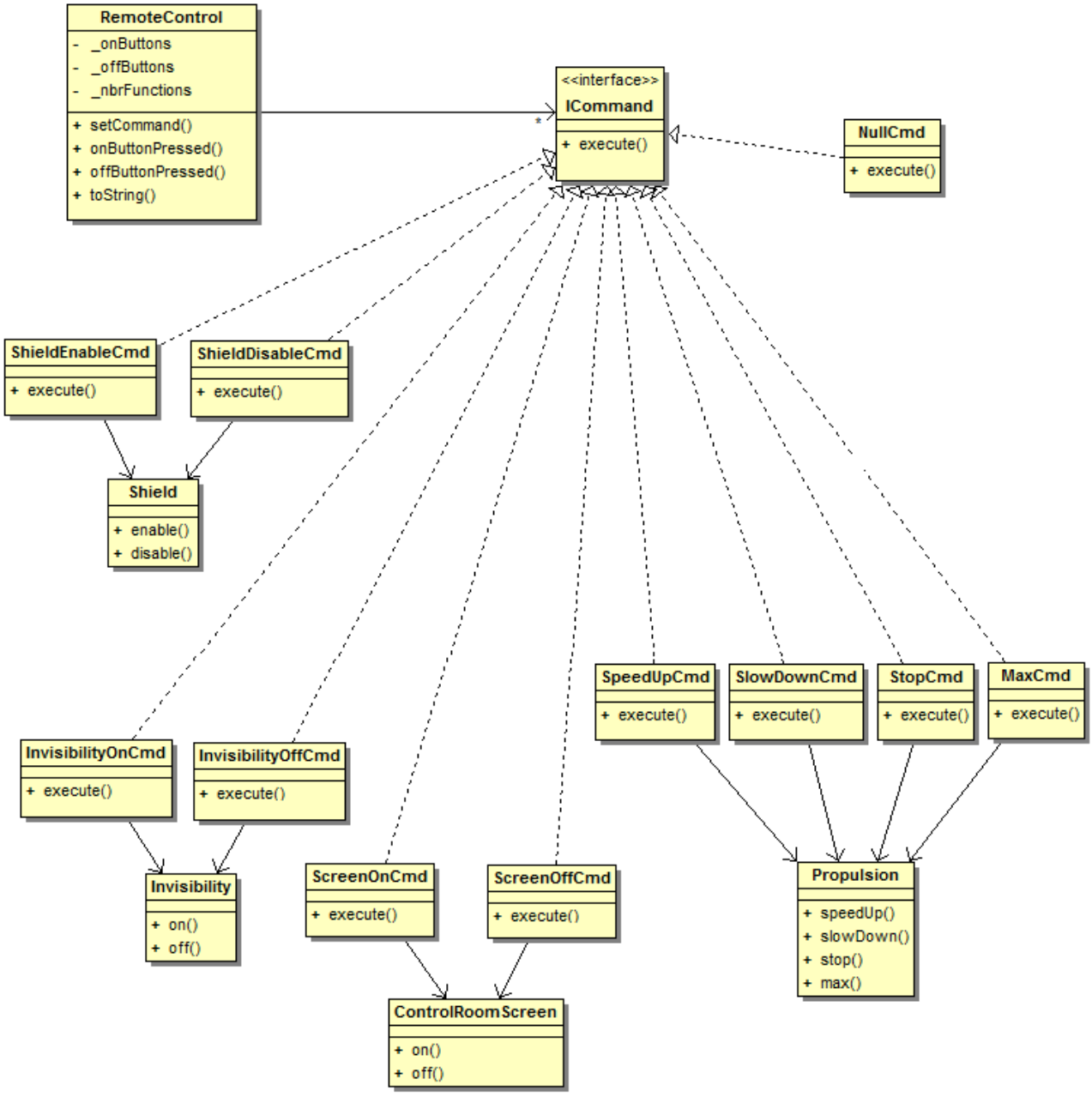
There is now the problem of unused buttons. If you press a button that is not programmed, it must do nothing. But in our pattern, pressing a button invokes the execute() method of the command associated with the button.

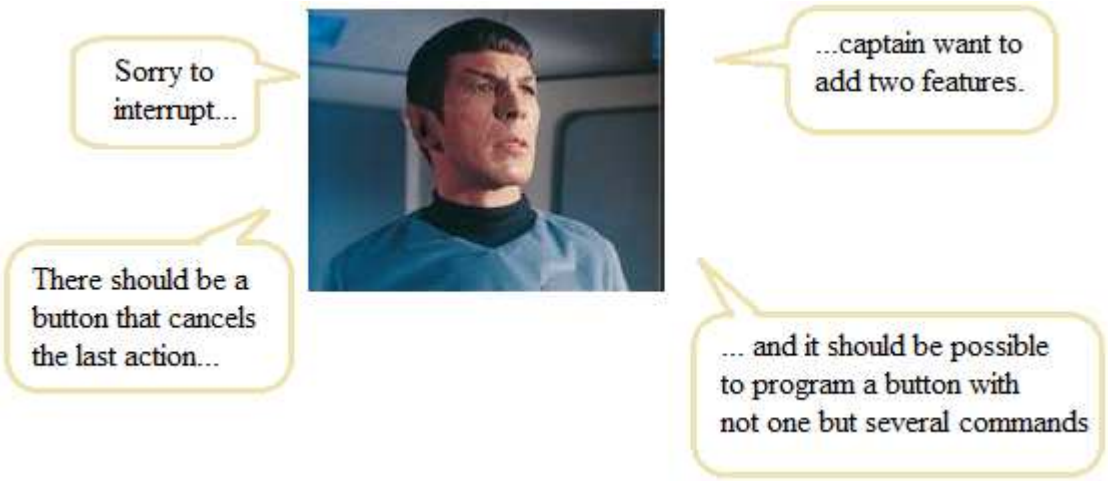
A first approach would be to test if the button has a good command before invoking this command.

But there is better: create a command that does nothing, and call it NullCmd. Like any command, it has an execute() method, but it will do nothing. So there is no test to do. We must unsure that any unused button is well associated with the command NullCmd.



Here is the complete design:





Well well...

After a meeting with the captain, it’s a button that puts Enterprise in "danger mode”, ie:

- Maximum speed.
- Shield on.
- Invisibility on.

The opposite of "danger mode” is the "normal mode":

- Low speed.
- Shield off.
- Invisibility off.

As for the "Cancel" button it should just do the reverse of the last command executed.

Fortunately, thanks to the Command design pattern, these new features will give us no problems.

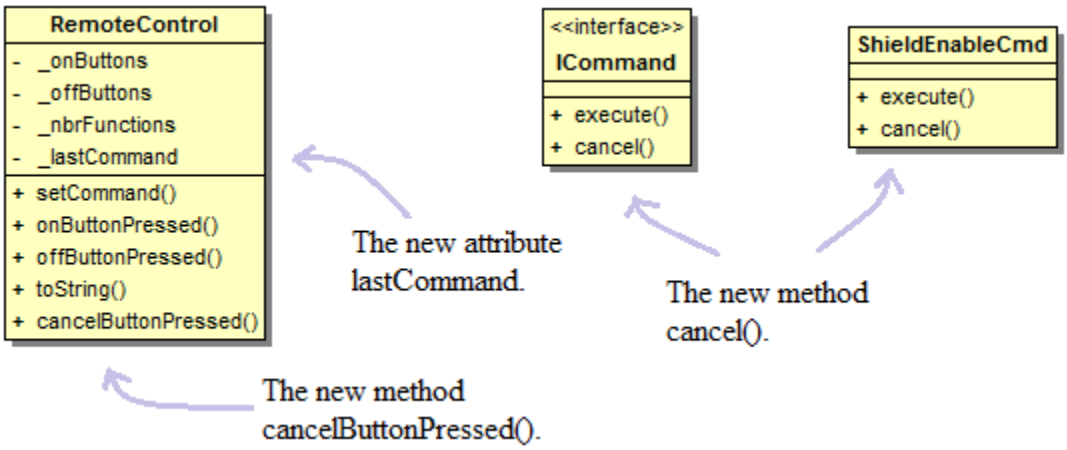
Let's start with the cancel button:

- The remote control must store the last command to be able to cancel it. So we add a private attribute _lastCommand which contain the reference of the last command executed. Thus each command will have to update _lastCommand.
- The remote control should also have a "Cancel" button, which will be realized by a cancelButtonPressed() method.
- Who must know what it takes to cancel a command ? This is the command itself. We must ensure that each command knows how to cancel the action. To do this add a cancel() method to the ICommand interface. This method must be implemented by all commands.

Here's what should happen when you press the Cancel button on the remote control:

- The cancelButtonPressed() method of the remote control is invoked.
- This method in turn invokes the cancel() method on the _lastCommand attribute. Indeed this attribute is simply a reference to an object of type ICommand.
- The cancel() method of the command acts on the controlled device, so as to cancel its action.

Here are the changes to classes. Here only one command is represented, but it is understood that all commands must implement the new cancel() method.

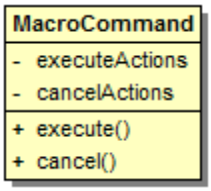


Now let’s see the command which triggers several actions.

Adaptation to achieve this result is extremely simple. Simply create a command whose methods execute() and cancel() perform several actions.

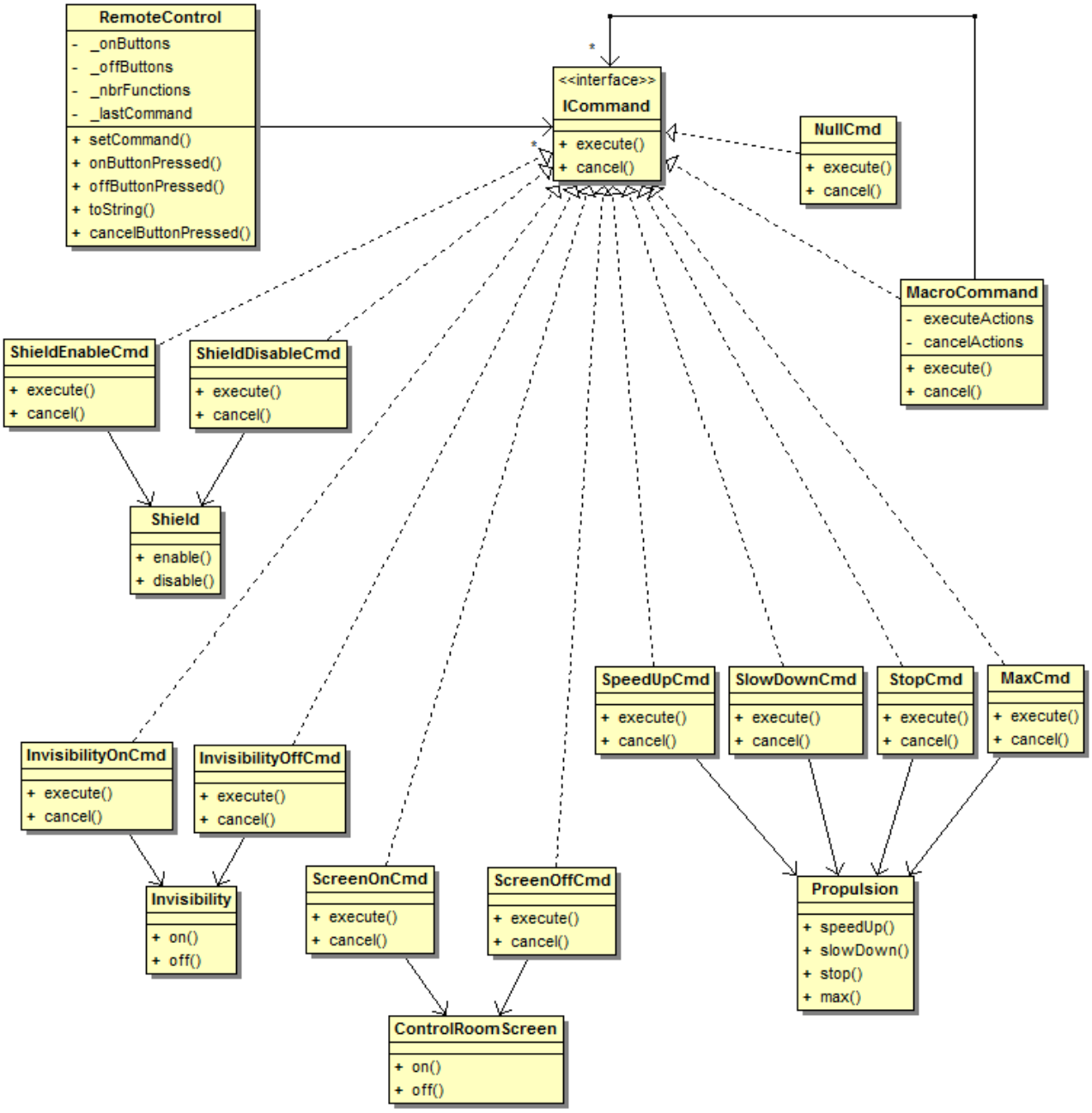
For this command to know what to do for each of the two methods, we add two private attributes of array type, which contain references to commands to execute.

In summary this macro command only executes other commands. Being itself a command, it provides the execute() and cancel() methods.
Here is this new class:



Before moving on to coding, here is the final classes model:

10.1 Class Diagram



Some remarks:

- Each command directly control a device.
- The macro command does not directly control a device, as it will only execute other commands listed in the executeActions and cancelActions arrays. This is indicated by the solid arrow connecting MacroCommand to ICommand. This arrow means that MacroCommand has several references to ICommand objects.

- The Engine class offers four functions, we created four commands.

10.2 Coding

Invisibility class:

```php
<?php

class Invisibility {

    private $_name = '';

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

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

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

}

?>
```

ControlRoomScreen class:

```php
<?php

class ControlRoomScreen {

    private $_name = '';

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

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

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

}

?>
```

Propulsion class:

```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 to speed up.";
        }
        $this->log($alert);
    }

    public function slowDown() {
        $this->_currentSpeed--;
        $alert = "";
        if ($this->_currentSpeed < 0) {
            $this->_currentSpeed = 0;
            $alert = ". Impossible to slow down.";
        }
        $this->log($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>';
    }

}

?>
```

An array to set different speed.

The constructor make some initializations.

To speed up, we increment currentSpeed, checking if we reach the limit.

Ditto for deceleration.

Shield class:

```php
<?php

class Shield {

    private $_name = '';

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

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

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

}

?>
```

MacroCommandCmd class:

```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();
        }
    }

}

?>
```

We give the two actions lists to the constructor.

We simply execute the command list.

NullCmd class:

```php
<?php

class NullCmd implements ICommand {

    public function execute() {
        echo "I'm a command that does nothing.";
    }

    public function cancel() {
        echo "I'm a command that does nothing.";
    }

}

?>
```

ICommand interface:

```php
<?php

interface ICommand {

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

?>
```

ShieldEnableCmd command:

```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();
    }

}

?>
```

The controlled device (receiver) is passed to the command controller.

The command knows what action should be performed on the controlled device.

ShieldDisableCmd command:

```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();
    }

}

?>
```

ScreenOnCmd command:

```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();
    }

}

?>
```

ScreenOffCmd command:

```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();
    }

}

?>
```

InvisibilityOnCmd command:

```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();
    }

}

?>
```

InvisibilityOffCmd command:

```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();
    }

}

?>
```

MaxCmd command:

```php
<?php

class MaxCmd implements ICommand {

    private $_myReceiver;

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

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

    public function cancel() {

    }

}

?>
```

StopCmd command:

```php
<?php

class StopCmd implements ICommand {

    private $_myReceiver;

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

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

    public function cancel() {

    }

}

?>
```

SpeedUpCmd command:

```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();
    }

}

?>
```

RemoteControl class:

```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(): unknown function: '
            . $functionIndex . '. Admited: from 0 to '
            . ($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(): unknown function: '
            . $functionIndex . '. Admited: from 0 to '
            . ($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(): unknown function: '
            . $functionIndex . '. Admited: from 0 to '
            . ($this->$_nbrFunctions - 1) . '.</br>';
        }
    }
```

The two arrays to store commands.

The constructor initialize all commands with NullCommand.

setCommand() allow to define a pair of controls.

When we press a button, the corresponding command is executed

```
    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() display the remote control configuration.

The cancel button execute the inverse of the last command.

10.3 Tests

As usual we will use index.php as the controller of the entire design

```php
<?php

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

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

echo 'Controller: start.', $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("shield");
$shieldDisableCommand = new ShieldDisableCmd($shield);
$shieldEnableCommand = new ShieldEnableCmd($shield);

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

$propulsion = new Propulsion("speed");
$slowDownCommand = new SlowDownCmd($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);
```

We indicate the number of commands required

The two action lists for macro command.

```
// RemoteControl Configuration:
$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);

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

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

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

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

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

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

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

echo $myRemoteControl;

echo '--------------', $newline;
echo 'Controller: End.', $newline;
?>
```

Remote control progamming.

Press buttons, have fun !

Max speed up.

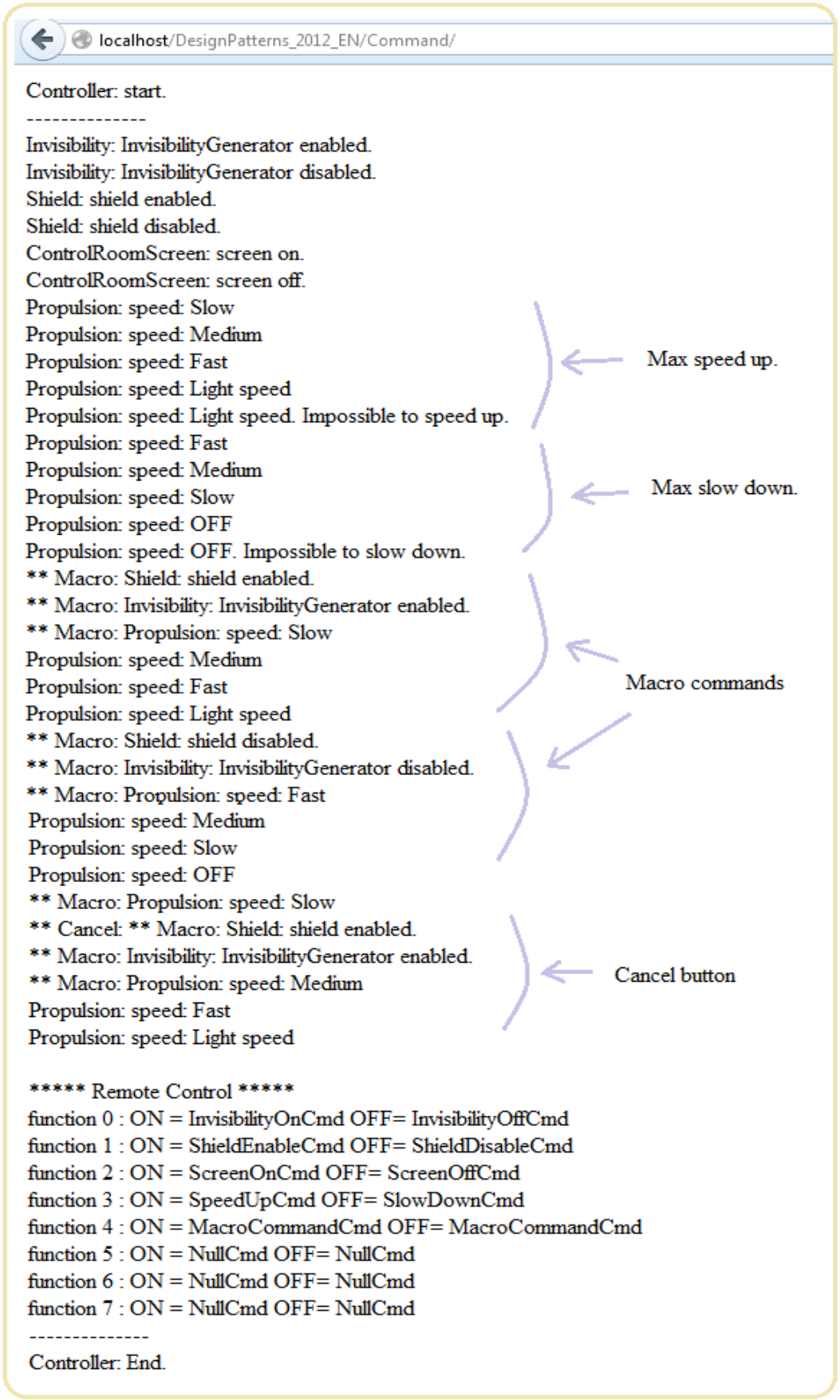
Max slow down.

Let's use our macro command.

...and our cancel button.

To finish, display the remote control configuration.

Here is the result:



10.4 Conclusion

As we have seen, the Command pattern separate two concepts that are usually combined:

- The request for an action to perform, ie the invocation of a method that will do a job. This notion becomes an Command.
- The execution of the work, ie the fact of performing the work defined by the command.

Action requests being encapsulated in standard objects (commands), it is easy to store them for later execution.

Thus the Command pattern is often used to treat work queues: One or more processes are responsible for executing a command one after the other. The process, which has no idea of what the command really does, simply invoke the execute() method of the command. Once the work is completed, the process takes the next command.

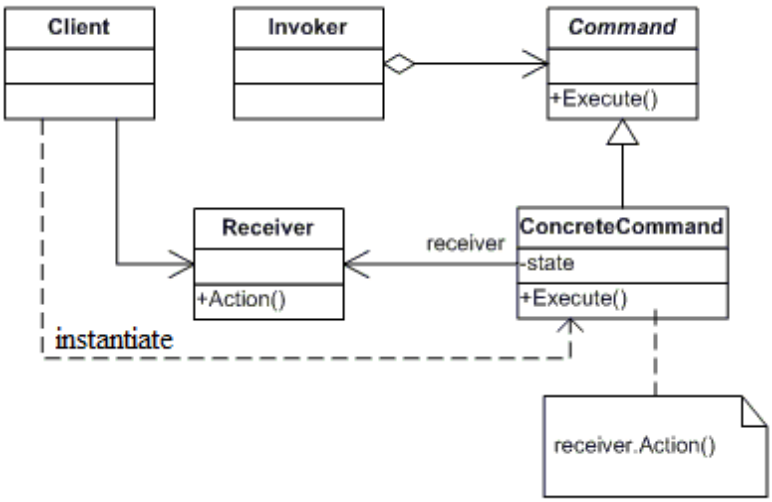
The pattern can also be used to historicize the activity of a system between two backups. If a crash occurs after restoring the last backup, it will be possible to replay historized commands to find the state of the system at the time of crash.

Uses the same pattern can be very different from each other.

This pattern, again, is based on the composition and not the implementation. This means that treatments are encapsulated in small, specialized classes that are associated by the controller, using the composition (ie references between objects) at run time. This means, in our example, the controller can reconfigure the remote control at any time.

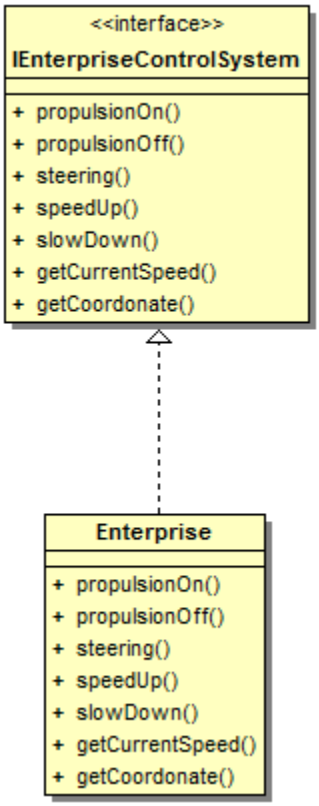
In contrast, a conventional design, based on implementation, we would certainly give us less numerous but larger classes containing hard-coded treatments. Any changes would often require to touch the classes. At run time the controller would have little leeway.

Here is the official Command pattern class diagram:


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

11 ADAPTER PATTERN

The control system of the Enterprise is as follows:



The IEnterpriseControlSystem interface imposes a set of methods necessary to manage the Enterprise.

The Enterprise class then implements these methods.

Use this class a little before going further. Here is the code of the interface:

```php
<?php

interface IEnterpriseControlSystem {

    public function propulsionOn();

    public function propulsionOff();

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

    public function speedUp();

    public function slowDown();
}

?>
```

And that of the Enterprise class:

```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);
    }
```

Some private attributs.

A 3D coordinates system.

Constructor initialize private attributs

Switch on and off of the propelling system.

To change course, we set new coordinates.

```
    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>";
    }

}

?>
```

Utility method to display messages.

Get current speed or coordinates

Here is an example of using the Enterprise class:

```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;
?>
```

We set a course.

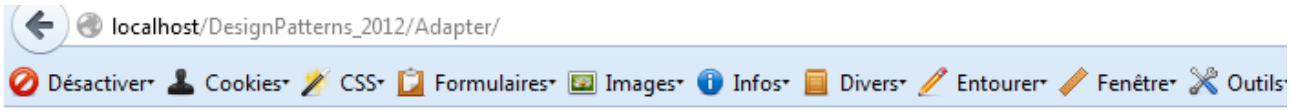
We test engine start and stop.

We test speed up when engine is off.

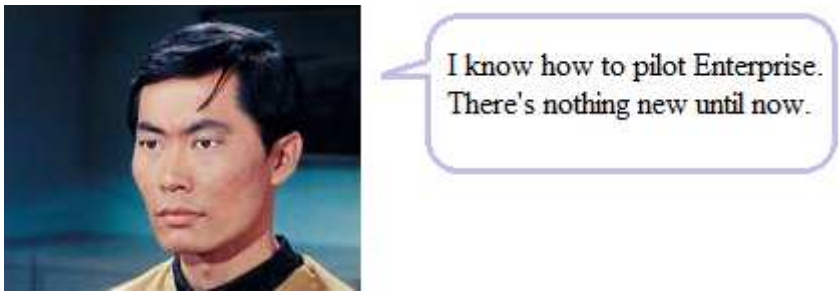
Max speed up.

Full slow down.

And here is the result of executing:

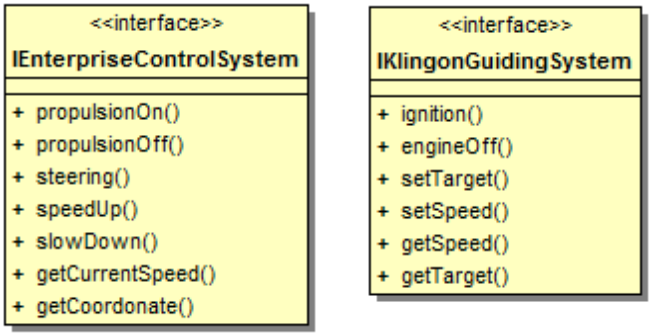



```
Controller: start.
--------------
Coordonate: x= 0; y= 0; z= 0
Coordonate: x= 25; y= 65; z= 33
Class: Enterprise. Method: propulsionOn(). Vaisseau: enterprise: propulsion On
Class: Enterprise. Method: propulsionOff(). Vaisseau: enterprise: propulsion Off
Class: Enterprise. Method: speedUp(). Vaisseau: enterprise: Impossible: propulsion off. Current speed: OFF
Class: Enterprise. Method: propulsionOn(). Vaisseau: enterprise: propulsion On
Class: Enterprise. Method: speedUp(). Vaisseau: enterprise: Current speed: Slow
Class: Enterprise. Method: speedUp(). Vaisseau: enterprise: Current speed: Medium
Class: Enterprise. Method: speedUp(). Vaisseau: enterprise: Current speed: Fast
Class: Enterprise. Method: speedUp(). Vaisseau: enterprise: Current speed: Light speed
Class: Enterprise. Method: speedUp(). Vaisseau: enterprise: Impossible d'accélérer. Current speed: Light speed
Class: Enterprise. Method: slowDown(). Vaisseau: enterprise: Current speed: Fast
Class: Enterprise. Method: slowDown(). Vaisseau: enterprise: Current speed: Medium
Class: Enterprise. Method: slowDown(). Vaisseau: enterprise: Current speed: Slow
Class: Enterprise. Method: slowDown(). Vaisseau: enterprise: Current speed: OFF
Class: Enterprise. Method: slowDown(). Vaisseau: enterprise: Impossible de ralentir. Current speed: OFF
--------------
Controller: end.
```



Indeed Mr Sulu. But interesting things happen now.

The Klingon ship has its own control system defined by the interface IKlingonGuidingSystem.



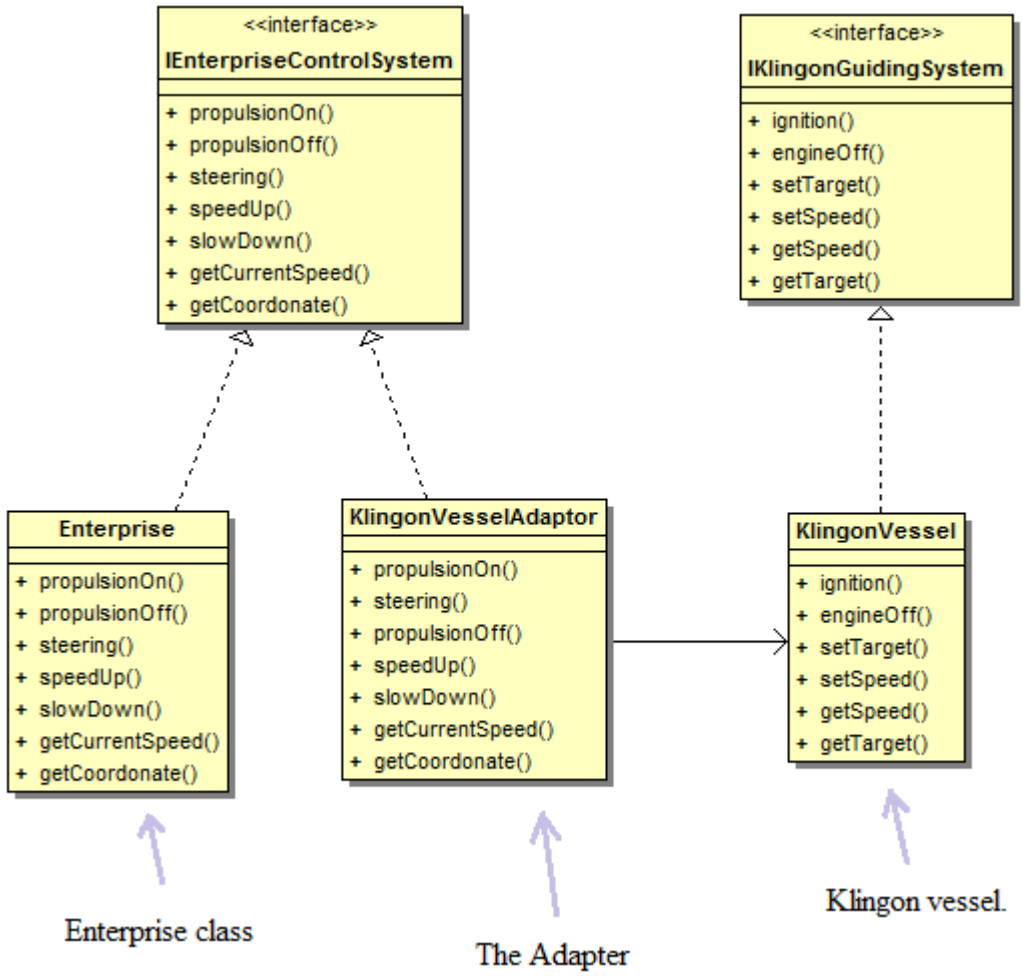
As can be seen, it has not much in common with that of the Enterprise. It will however be necessary to make these systems compatible.

The Adapter pattern is one of the most intuitive. It actually acts as an adapter, allowing two systems seem to be incompatible, to communicate. Just as an adapter allows an electrical device to plug into an electrical outlet in a foreign country.
Our adapter:

- Will be materialized by a class derived from IEnterpriseControlSystem. It will therefore have all the methods of controlling a spaceship Enterprise Type.
- But will have a reference to a Klingon vessel type.
- For each Enterprise type method, perform the equivalent of the Klingon ship.

It is this class that has the intelligence to adapt between the two systems.

Here is the corresponding class model:



Let's see how does KlingonVessel class work:

```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);
    }
```

Some private attributs.

Constructor initialize private attributs

Switch on and off of the propelling system.

Set a new course.

```
    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>';
    }

}

?>
```

Get current speed or coordinates

Utility method to display messages.

We must now determine how we adapt each Enterprise method to a Klingon method.

- The ignition() method turns the Klingon engine on. So this method directly corresponds to the propulsionOn() method of the Enterprise.
- Similarly, engineOff() is corresponding to propulsionOff().
- The setTarget() method corresponds to steering(). These provide three spatial coordinates defining the new course. But the Klingon reference system is not the same unit of measurement. It will be necessary to multiply the coordinates by a factor.
- The SetSpeed() method set the speed of the Klingon ship. It has a different operation of the Enterprise because here we define directly the target speed, while the Enterprise is accelerated (SpeedUp()) or decelerated (Slowdown()). Both methods must use SetSpeed() by increasing or decreasing the speed of a Klingon vessel from its current speed provided by getSpeed().
- The getSpeed() method corresponds to getCurrentSpeed().
- The getTarget() method corresponds to getCoordonate().

We just have to implement the KlingonVesselAdapter class:

```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();
    }

}

?>
```

Reference to Klingon vessel.

Switching on and off the engine: we call the appropriate method of the Klingon vessel.

The coeficient is applied to convert data.

Get current speed or coordinates

As can be seen, for each treatment, our adapter invokes the appropriate method on the Klingon ship, directly for simple cases, or adapting the data if necessary.

11.1 Tests

As usual we will use index.php as the controller of the entire design:

```php
<?php

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

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

echo 'Controller: start.', $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->slowDown();

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

echo '--------------', $newline;
echo 'Controller: End.', $newline;
?>
```

We instanciate the 2 necessary classes.

Swith on/off test.

Max speed up.

Max slow down.

Change course.

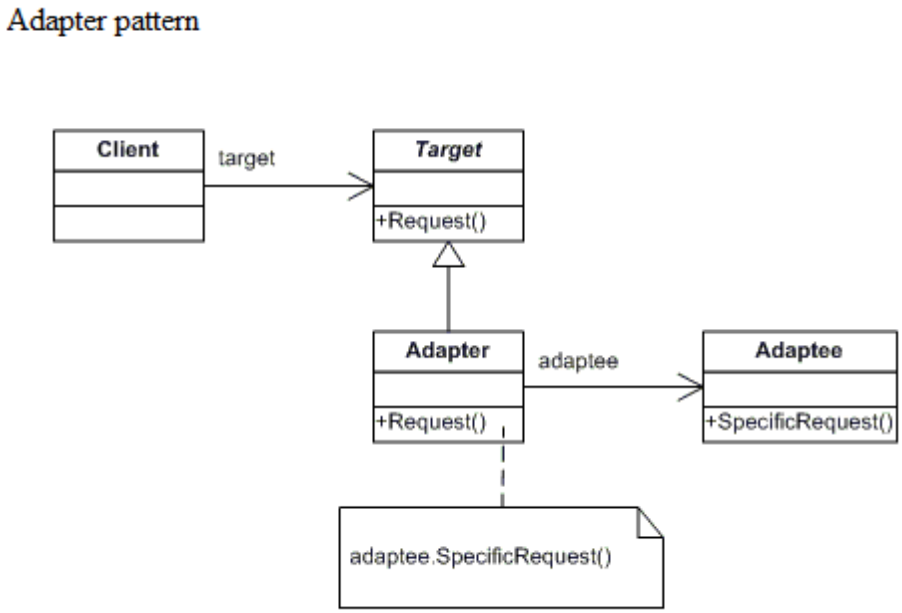
Display coordinates.

Here is the result:


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

The work is done. We converted the commands sent by the control system of the Enterprise and the Klingon ship adapter responds correctly.

The official Adapter pattern:



12 FACADE PATTERN

All systems are up and running captain.

It's too long Mr Spock.

There are many operations to do. That takes time Jim.

See if it's possible to get those operations automatic. I'm sure we can reduce the time.

The Enterprise has different systems which must be activated individually to get the vessel fully operational:

Système	Description
Communication	An internal network that can be enabled or disabled. An external network which may be enabled or disabled.
ElectricNetwork	Can be put in one of the modes: Off, Standby, Maintenance, On.
Engine	Can be put in one of the modes: Off, Standby, Maintenance, On.
Gravity	Can be enabled or disabled.
Teleportation	Can be put in one of the modes: Off, Standby, Maintenance, On.
Weapon	Can be put in one of the modes: Off, Standby, Maintenance, On.

To get Enterprise into service, it must use the following procedure:

- Enable internal and external communication networks.
- Set eletrical network to ON.
- Set artificial gravity to ON.
- Set teleportation to ON.
- Set weapons to ON.

There is also a procedure to get Enterprise in Maintenance Mode:

- Set internal communication network to ON.
- Set external communication network to OFF.
- Set electrical network to MAINTENANCE.
- Set artificial gravity to ON.
- Set teleportation to OFF.
- Set weapons to MAINTENANCE.

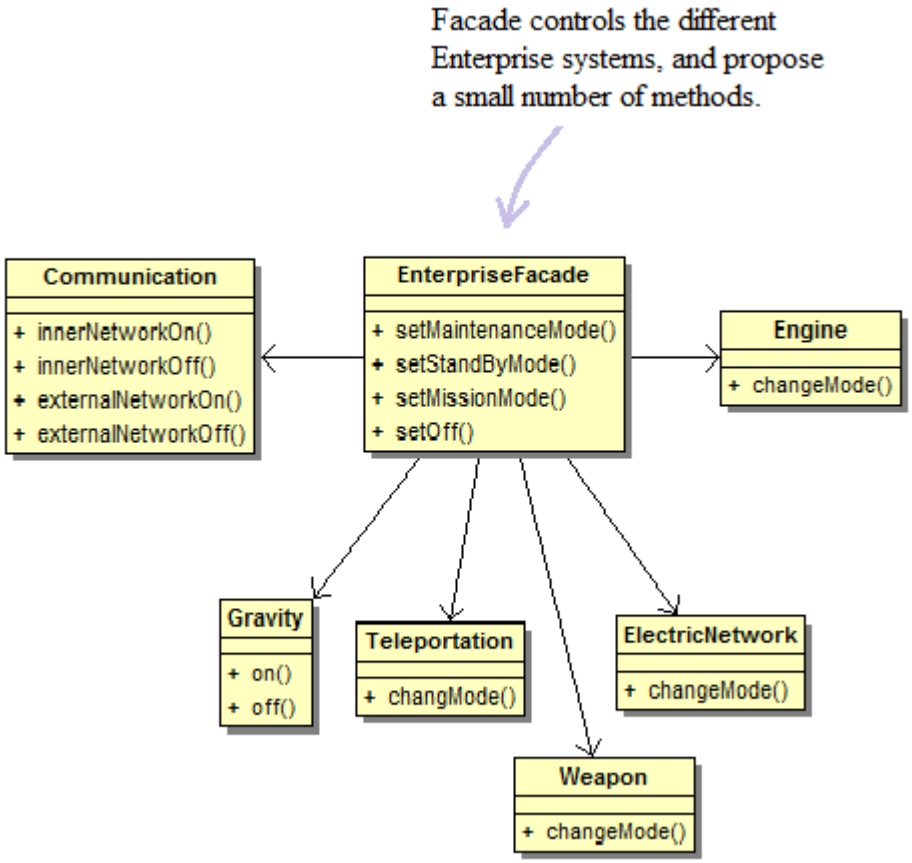
Another procedure to turn the vessel to Standby mode, and another to put it in Off mode.

For each procedure, we must instantiate objects and invoke the appropriate methods in a specific order.

There is a certain complexity that we would like to get rid of. This is where the design pattern Façade can be usefull.

The Facade pattern will take care of all this complexity and automate operations for us. We are offering a limited number of methods to simply control the complex Enterprise system.

Specifically the Facade pattern is a class that fit between the complex system, and the user of the complex system:



To control the Enterprise is no longer necessary to know the complexity of the various systems. Just use the methods proposed by the Facade. It will do the job for us.

As everything is clear, let's coding.

12.1 Coding

Communication: class

```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>';
    }

}
?>
```

Internal and external networks state.

Internal network management.

External network management.

A utility method to display infos.

Gravity class:

```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>';
    }

}

?>
```

Teleportation class:

```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>';
    }

}
?>
```

The different possible modes.

Weapon class:

```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>';
    }

}
?>
```

The different possible modes.

ElectricNetwork class:

```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>';
    }

}
?>
```

The different possible modes.

Engine class:

```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>';
    }

}
?>
```

The different possible modes.

EnterpriseFacade class:

```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");
    }
```

References to the different Enterprise systems.

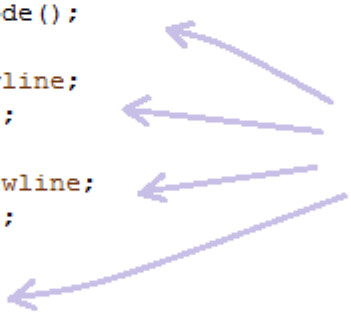
That's where we manage all the system complexity.

```
    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

As usual we will use index.php as the controller of the entire design:

```php
<?php

//*************************************************************
// FACADE pattern controller
//*************************************************************

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

echo 'Controller: start.', $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();


// treatment:
//$myTeleportation->changeMode("MAINT");
//$myTeleportation->changeMode("STANDBY");

echo '--------------', $newline;
echo "You can always go directly to a system:" . "<br>";
$myGravity->off();
$myGravity->on();

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



Here is the result:

Excellent Mr Spock. The vessel was never ready as fast.

As we have a little time, i'll try my new portable laser gun

13 TEMPLATE METHOD PATTERN

To illustrate this pattern, we will look at the food served on board.
A modern and balanced food that make us dream.

All kinds of dishes are cooked on demand, but they are all made from a single ingredient: granules containing all that is needed by the body.

There are lipids, carbohydrates, proteins, minerals, vitamins. They have no particular flavor, which allow to transform them at will by giving them the texture, color and flavor desired.

3 textures are possible:

- The granules are left intact (pasta, small pieces of vegetables or meat etc....)
- The granules are mixed with water (soups, purees, creams...)
- The granules are kneaded with water, molded, partially dried, stripped. We obtain blocks of different shapes: pies, pastries, meats...

Prepare a dish thus means following the steps:

- Take a quantity of granules.
- Transform into one of three textures.
- Add the desired dye.
- Add the desired flavor.
- Set temperature.
- Add optionally sauce.
- Add optionally a vitamin supplement.

All kinds of dishes are achievable. Simply choose the texture, flavor and color.

But look more closely at the recipes for each of the three textures:

Granular textures:
- Take some granules
- apply color
- apply flavor
- blend
- set temperature
- apply the sauce option
- apply vitamin supplement option

Fluid textures:
- Take some granules
- add hot water
- apply color
- apply flavor
- mix
- set temperature
- apply the sauce option
- apply vitamin supplement option

Solid textures:
- Take some granules
- add hot water
- apply color
- apply flavor
- mix
- mold
- dry
- unmold
- set temperature
- apply the sauce option
- apply vitamin supplement option

As can be seen the three recipes are very similar: some steps are common, others are specific.

We would be tempted to factor what may be, but how factoring parts of an algorithm ?

For this kind of problem, the best solution is the Template Method design pattern.

This pattern will allow us to define a general algorithm imposed, while leaving the door open to some variability for steps that require.

Let's start by putting the 3 recipes in parallel to identify common operations:

	Granular textures:		Fluid textures:		Solid textures:
1	- take some granules	=	- take some granules	=	- take some granules
2			- add hot water	=	- add hot water
3	- apply color	=	- apply color	=	- apply color
4	- apply flavor	=	- apply flavor	=	- apply flavor
5	- blend		- mix		- mix
6					- mold
7					- dry
8					- unmold
9	- set temperature	=	- set temperature	=	- set temperature
10	- apply the sauce option	=	- apply the sauce option	=	- apply the sauce option
11	- apply vitamin supplement option	=	- apply vitamin supplement option	=	- apply vitamin supplement option

We can separate operations into 3 groups:

- Group A: operations are identical in the three recipes (1,3,4,9,10,11 operations). These operations will be defined and implemented in our Template Method. They will therefore be mandatory and not modifiable.
- Group B: identical operations but does not involve the 3 recipes. The operation is optional (2,6,7,8 operations). These operations will be proposed by the Template Method in the form of hooks: the user class has the option to run or not and may even define its implementation.
- Group C: operations that are conceptually the same step, but the way to do it is different in the three recipes (step 5). These operations will be required, but subclasses will have to implement them.

We now define what will become our Template Method: the unique recipe that can be adapted to 3 recipes.

	Template Method	Method type	Comment	Method
1	- take some granules	static	mandatory	getGrainFood()
2	- add hot water	- protected - non implemented	hook	addHotWater()
3	- apply color	static	mandatory	applyColor()
4	- apply flavor	static	mandatory	applyFlavour()
5	- mix	abstract	delegated	blend()
6	- mold	- protected - non implemented	hook	mould()
7	- dry	- protected - non implemented	hook	dryUp()
8	- unmold	- protected - non implemented	hook	outMould()
9	- set temperature	static	mandatory	applyTemperature()
10	- apply the sauce option	static	mandatory	addSauce()
11	- apply vitamin supplement option	static	mandatory	addVitamins()

Methods imposed (in blue) will be defined as static in the superclass that is not inherited by subclasses. The latter therefore can not redefine.

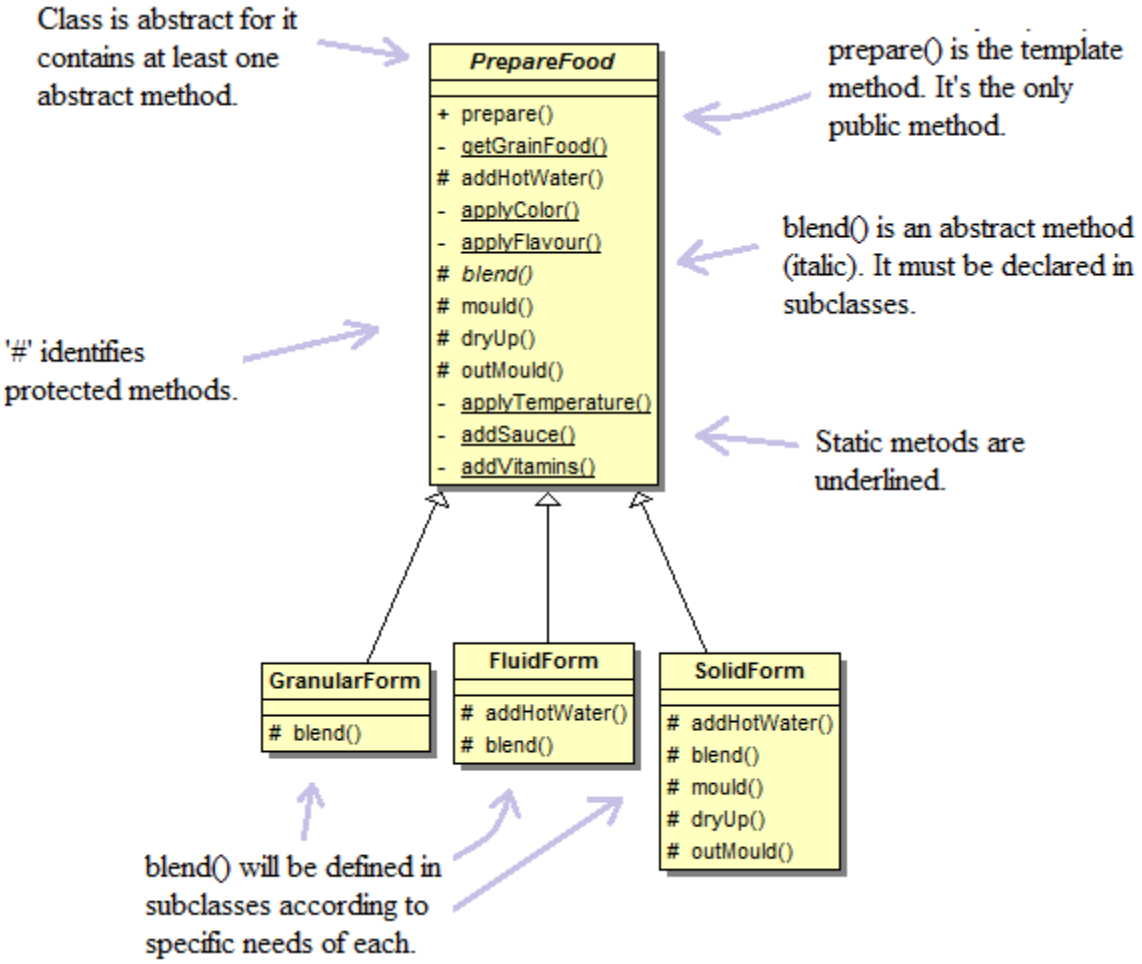
Methods "hook" type will be defined as protected to be accessible only by subclasses, not implemented because it is precisely the subclasses which are responsible to implement them according to their specific needs.

The delegate method is defined as abstract. It is a mandatory step in the recipe, but must be implemented by subclasses as how to vary from one recipe to another.

13.1 Class model

The Template Method pattern is in the form of an abstract class containing the Template Method and a number of other methods needed.

it is possible to create classes derived from the abstract class. In our case we will create a subclass for each recipe:



13.2 Coding

PrepareFood class:

```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 "I take a certain amount of granules." . "<br>";
    }

    private static function applyFlavour() {
        // Mandatory: implemented in abstract class as static.
        echo "I apply the requested flavor." . "<br>";
    }

    private static function applyColor() {
        // Mandatory: implemented in abstract class as static.
        echo "I apply the requested colour." . "<br>";
    }

    private static function applyTemperature() {
        // Mandatory: implemented in abstract class as static.
        echo "I put the dish at the required temperature." . "<br>";
    }

    private static function addSauce() {
        // Mandatory: implemented in abstract class as static.
        echo "I add the sauce required." . "<br>";
    }
```

The word 'final' prohib
subclasses to overwrit
the method.

Template meth
defines all step
the algorithm.

```
    private static function addVitamins() {
        // Mandatory: implemented in abstract class as static.
        echo "I add vitamins required." . "<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();

}
?>
```

GranularForm class must only implement the blend() method:

```php
<?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 "I add hot water ." . "<br>";
    }

    protected function blend() {
        echo "I mix the granules to obtain a fluid substance." . "<br>";
    }

}

?>
```

The SolidForm class must implement several methods:

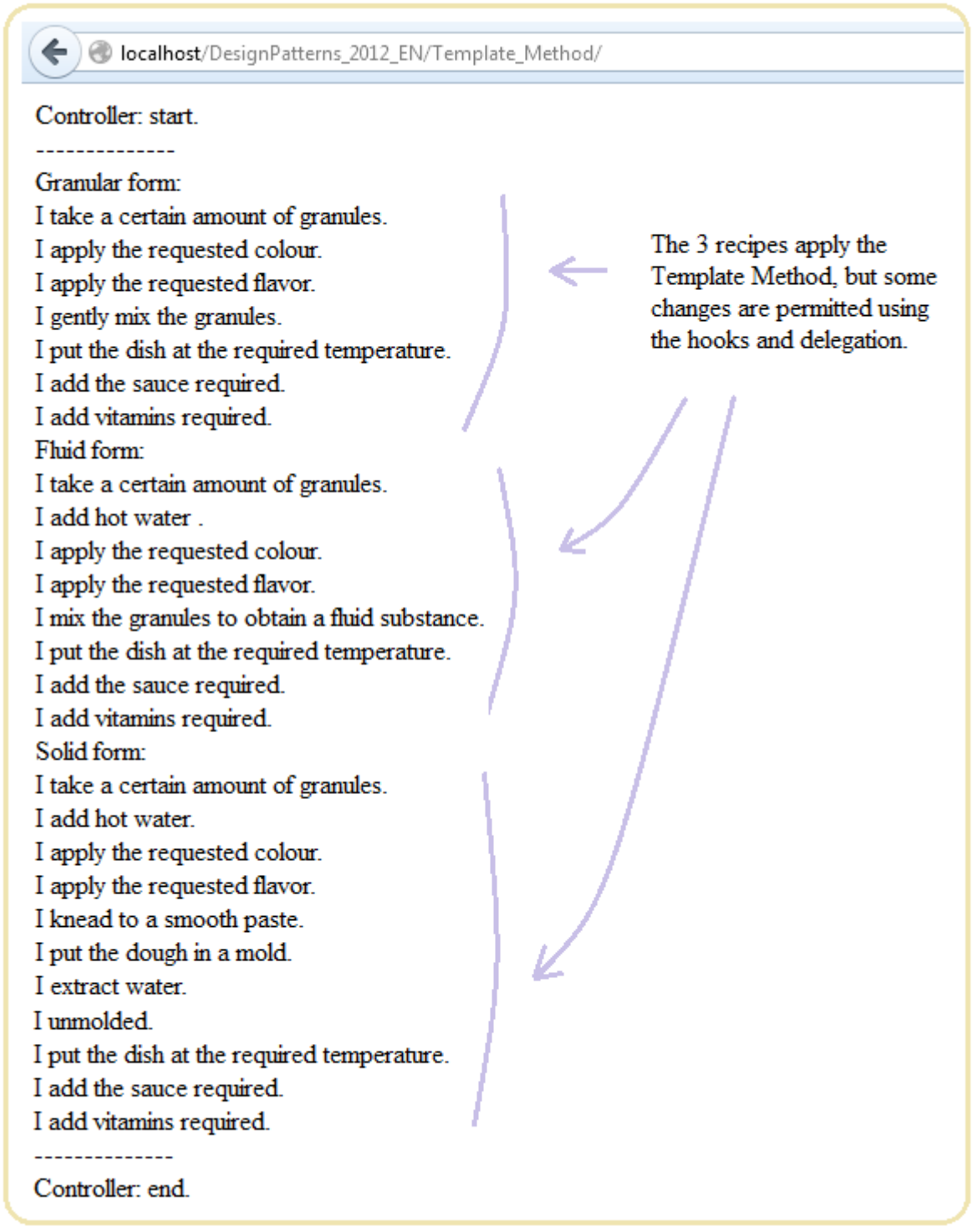
```php
<?php

class SolidForm extends PrepareFood {

    protected function addHotWater() {
        echo "I add hot water." . "<br>";
    }

    protected function blend() {
        echo "I knead to a smooth paste." . "<br>";
    }

    protected function mould() {
        echo "I put the dough in a mold." . "<br>";
    }

    protected function dryUp() {
        echo "I extract water." . "<br>";
    }

    protected function outMould() {
        echo "I unmolded." . "<br>";
    }

}

?>
```

13.3 TESTS

As usual we will use index.php as the controller of the whole design.
The code is extremely simple:

```php
<?php

//*************************************************************
// Template_Method pattern controler
//*************************************************************

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

echo 'Controller: start.', $newline;
echo '--------------', $newline;

// Instanciations:
$myGranularForm = new GranularForm();
$myFluidForm = new FluidForm();
$mySolidform = new SolidForm();


// treatment:
echo "Granular form:" . "<br>";
$myGranularForm->prepare();
echo "Fluid form:" . "<br>";
$myFluidForm->prepare();
echo "Solid form:" . "<br>";
$mySolidform->prepare();

echo '--------------', $newline;
echo 'Controller: end.', $newline;
?>
```

Here is the result:



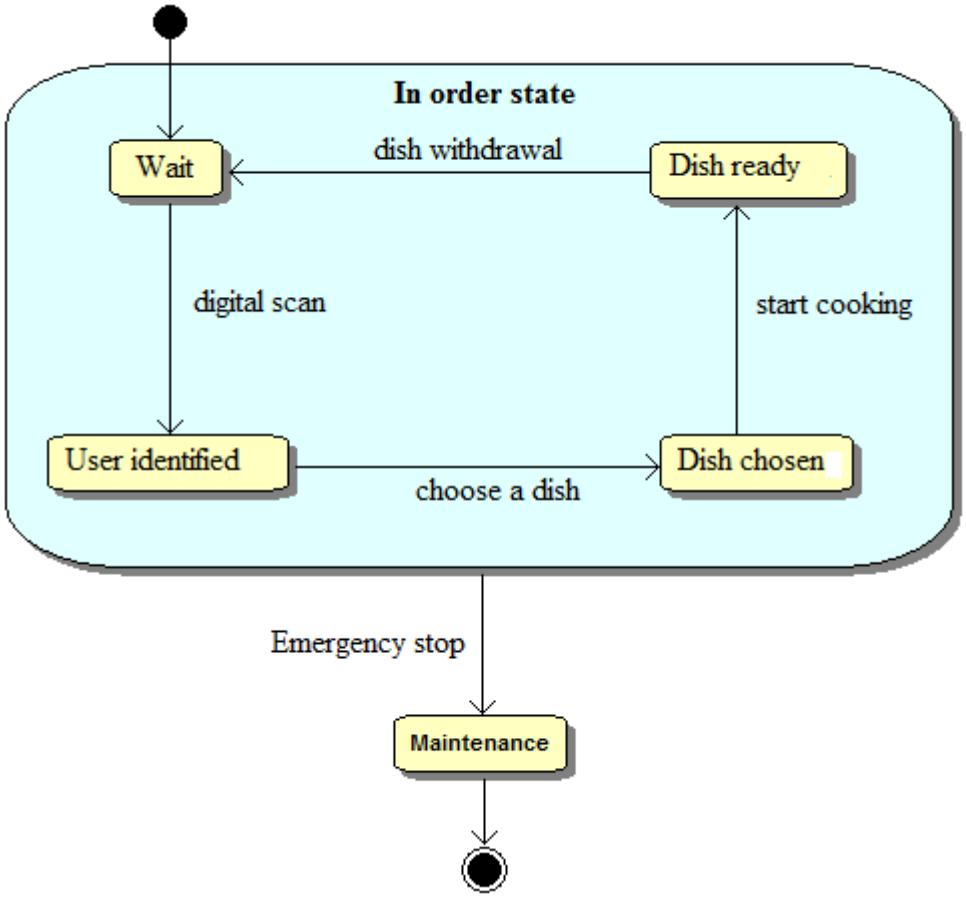

The taste of these granules is nothing compare to a good scotch.

14 STATE PATTERN

Continue to take an interest in nutrition aboard the Enterprise. With the Template Method pattern, we saw a single ingredient allowed to prepare any dish.
Now let's see how the machines that prepare dishes on demand works. The operation of these vending machines is very simple:

- The person affixes his hand on an optical sensor to be identified.
- The person verbally indicates the desired dish.
- The machine prepares the dish.
- The machine indicates that the dish is ready.
- The person removed the dish.
- At any time you can press the emergency stop button that puts the distributor pending maintenance.

This operation can be represented by the following state diagram:



The distributor may have five states:

- **Wait**: the machine does nothing and waits for an order.
- **User identified** in: a person has been identified on the digital sensor.
- **Dish chosen:** the person chose a dish.
- **Dish ready**: the person can remove the dish.
- **Maintenance**: the emergency stop button was pressed.

We understand that to respond favorably to an action, the machine must be in the state corresponding to this action.

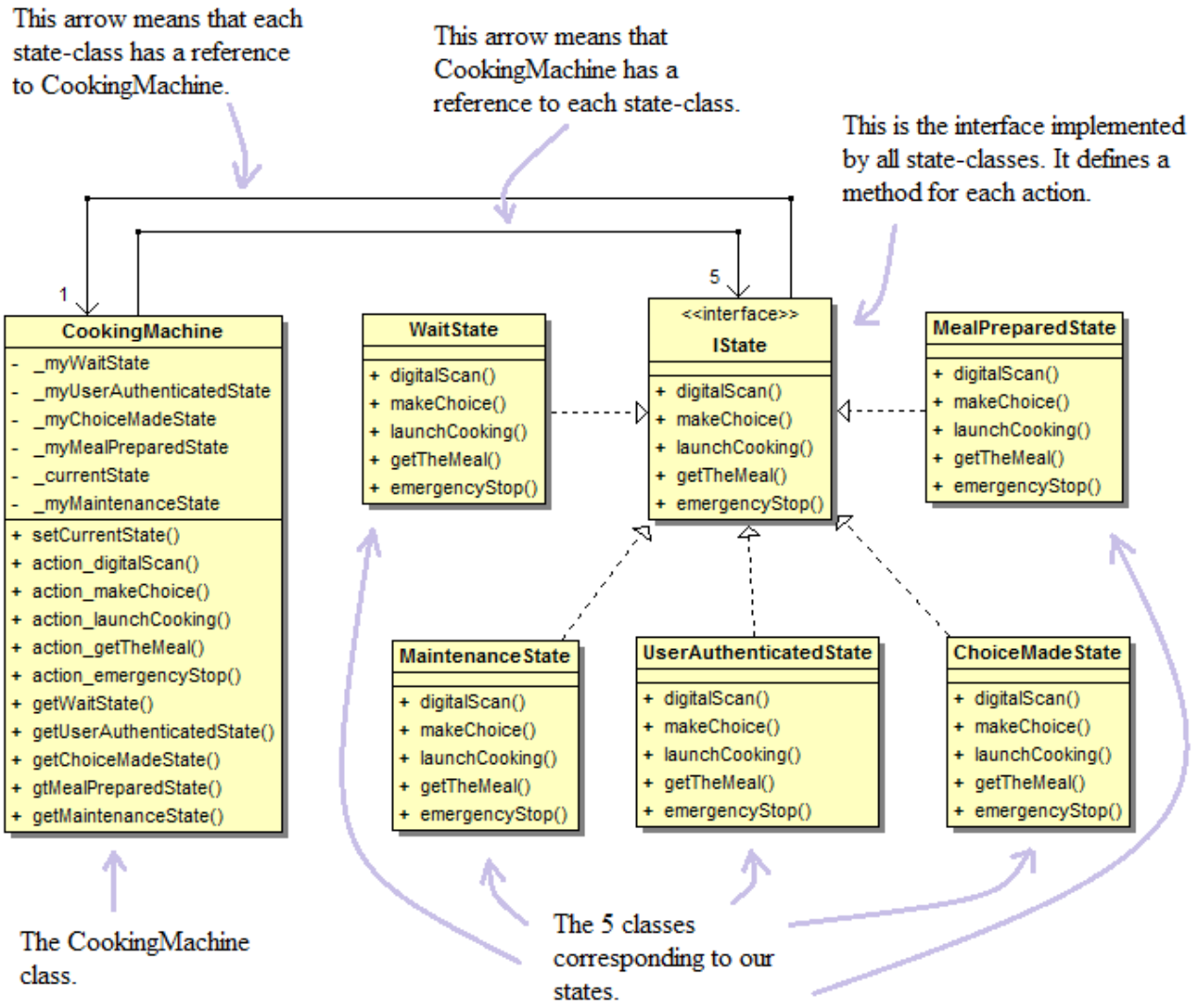
If, for example, the machine is in state "dish chosen", it cannot process action "digital scan". The only action in this case is "start cooking".

We guess right away the complexity of our future class representing the vending machine: for every action there should test the current state of the distributor to find out how to treat this action. We obviously do not going to engage in this way.

The State pattern perfectly meets this problem with a very simple approach:

- We'll have a class for our vending machine. Call it CookingMachine.
- Each action become a method in the CookingMachine class.
- Each state is a class that implements an interface requiring the class to declare a method for each action.

Here is the corresponding class model:



See how the design works:

- It is CookingMachine receiving user actions.
- CookingMachine has a reference to the class representing its current state, with its attribute _currentState.
- When CookingMachine receives an action, it only transmit it (ie delegate) to the object referenced by the class attribute _currentState.
- It is therefore _ currentState that concretely deals with the action.
- In addition, if the action involves a change of CookingMachine state, it is still _currentState object that will make this change.
- Classes representing the states, implement all possible actions. Generally, in each class, one action (ie a single method) performs a significant treatment. Others, in our case, display a message indicating that the action is not admissible for this state.

We see that CookingMachine is just a mailbox: it transmits actions to its current state.

It is easy to change the behavior of a state intervening only on the relevant class. It is also easy to add a new state: just create a new class that implements the interface IState.

14.1 Coding

IState interface:

```php
<?php

interface IState {

    public function digitalScan();
    public function makeChoice();
    public function launchCooking();
    public function getTheMeal();
    public function emergencyStop();
}

?>
```

ChoiceMadeState class:

```php
<?php

class ChoiceMadeState implements IState {

    private $_myCookingMachine;

    public function __construct($CookingMachine) {
        $this->_myCookingMachine = $CookingMachine;
    }

    public function digitalScan() {
        echo "Action: digitalScan() impossible with current state." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function makeChoice() {
        echo "Action: makeChoice() impossible with current state." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function launchCooking() {
        echo "Action: launchCooking()..." . "<br>";
        echo "Meal is ready." . "<br>";
        $this->_myCookingMachine->setCurrentState($this->_myCookingMachine
                                                  ->getMealPreparedState());
        $this->_myCookingMachine->displayCurrentState();
    }

    public function getTheMeal() {
        echo "Action: getTheMeal() impossible with current state." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function emergencyStop() {
        echo "Action: emergencyStop()." . "<br>";
        $this->_myCookingMachine->setCurrentState($this->_myCookingMachine
                                                  ->getMaintenanceState());
        $this->_myCookingMachine->displayCurrentState();
    }

}

?>
```

The reference to CookingMachine is passed to the constructor.

Action is done here, and we modify CookingMachine state.

Emergency action will be implemented in all states.

MaintenanceState class:

```php
<?php

class MaintenanceState implements IState {

    private $_myCookingMachine;

    public function __construct($CookingMachine) {
        $this->_myCookingMachine = $CookingMachine;
    }

    public function digitalScan() {
        echo "Action: digitalScan() impossible. I'm in maintenance state." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function makeChoice() {
        echo "Action: makeChoice() impossible.  I'm in maintenance state." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function launchCooking() {
        echo "Action: launchCooking() impossible. I'm in maintenance state." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function getTheMeal() {
        echo "Action: getTheMeal() impossible. I'm in maintenance state." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function emergencyStop() {
        echo "Action: emergencyStop()." . "<br>";
        echo "I'm already in maintenance state !" . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

}

?>
```

MealPreparedState class:

```php
<?php

class MealPreparedState implements IState {

    private $_myCookingMachine;

    public function __construct($CookingMachine) {
        $this->_myCookingMachine = $CookingMachine;
    }

    public function digitalScan() {
        echo "Action: digitalScan() impossible in current state." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function makeChoice() {
        echo "Action: makeChoice() impossible in current state." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function launchCooking() {
        echo "Action: launchCooking() impossible in current state." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function getTheMeal() {
        echo "Action: getTheMeal()." . "<br>";
        echo "Please remove the meal." . "<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();
    }

}

?>
```

UserAuthenticatedState class:

```php
<?php

class UserAuthenticatedState implements IState {

    private $_myCookingMachine;

    public function __construct($CookingMachine) {
        $this->_myCookingMachine = $CookingMachine;
    }

    public function digitalScan() {
        echo "Action: digitalScan() impossible in current state." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function makeChoice() {
        echo "Action: makeChoice()." . "<br>";
        echo "Choice validated." . "<br>";
        $this->_myCookingMachine->setCurrentState($this->_myCookingMachine
                                                  ->getChoiceMadeState());
        $this->_myCookingMachine->displayCurrentState();
    }

    public function launchCooking() {
        echo "Action: launchCooking() impossible in current state." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function getTheMeal() {
        echo "Action: getTheMeal() impossible in current state." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function emergencyStop() {
        echo "Action: emergencyStop()." . "<br>";
        $this->_myCookingMachine->setCurrentState($this->_myCookingMachine
                                                  ->getMaintenanceState());
        $this->_myCookingMachine->displayCurrentState();
    }

}

?>
```

WaitState class:

```php
<?php

class WaitState implements IState {

    private $_myCookingMachine;

    public function __construct($CookingMachine) {
        $this->_myCookingMachine = $CookingMachine;
    }

    public function digitalScan() {
        echo "Action: digitalScan()." . "<br>";
        echo "Authentication complete." . "<br>";
        $this->_myCookingMachine->setCurrentState($this->_myCookingMachine
                                                  ->getUserAuthenticatedState());
        $this->_myCookingMachine->displayCurrentState();
    }

    public function makeChoice() {
        echo "Action: makeChoice() impossible in current state." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function launchCooking() {
        echo "Action: launchCooking() impossible in current state." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function getTheMeal() {
        echo "Action: getTheMeal() impossible in current state." . "<br>";
        $this->_myCookingMachine->displayCurrentState();
    }

    public function emergencyStop() {
        echo "Action: emergencyStop()." . "<br>";
        $this->_myCookingMachine->setCurrentState($this->_myCookingMachine
                                                  ->getMaintenanceState());
        $this->_myCookingMachine->displayCurrentState();
    }

}

?>
```

CookingMachine class:

```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);

        // Default state:
        $this->_myCurrentState = $this->_myWaitState;

        // we display the current state:
        $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();
    }
```

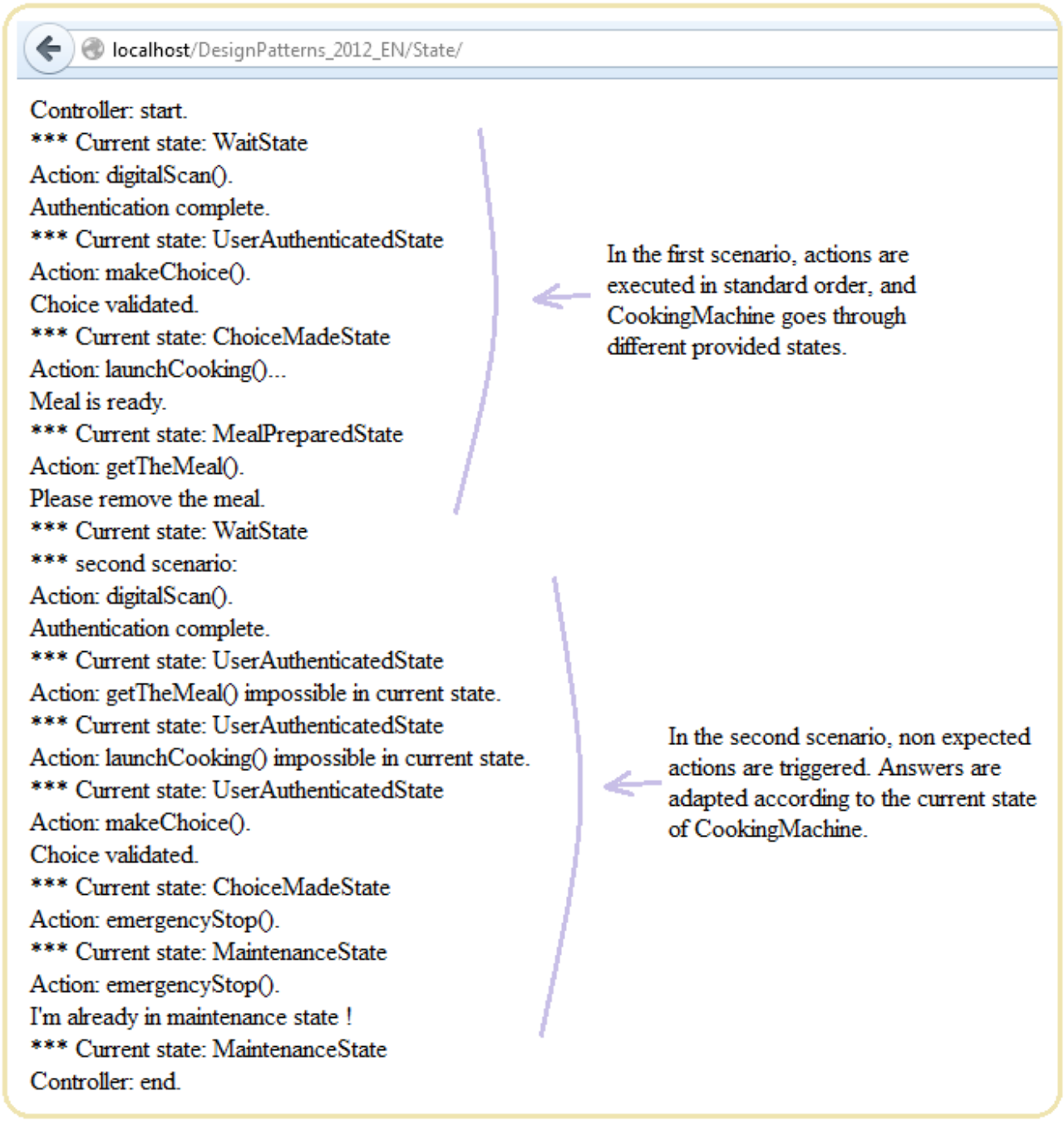
```
    public function getWaitState() {
        return $this->_myWaitState;
    }

    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 "*** Current state: " . get_class($this->_myCurrentState) . "<br>";
    }

}

?>
```

14.2 TESTS

As usual we will use index.php as the controller of the whole design.

```php
<?php

//*************************************************************
// STATE pattern controler
//*************************************************************

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

echo 'Controller: start.' . $newline;

// Instanciations:
$myCookingMachine = new CookingMachine();

// treatment (standard scenario):
$myCookingMachine->action_digitalScan();
$myCookingMachine->action_makeChoice();
$myCookingMachine->action_launchCooking();
$myCookingMachine->action_getTheMeal();

// treatment (scenario with unexpected actions):
echo "*** second scenario:" . "<br>";
$myCookingMachine->action_digitalScan();
$myCookingMachine->action_getTheMeal();
$myCookingMachine->action_launchCooking();
$myCookingMachine->action_makeChoice();
$myCookingMachine->action_emergencyStop();
$myCookingMachine->action_emergencyStop();

echo 'Controller: end.' . $newline;

?>
```

And here is the result:



15 ITERATOR PATTERN

So Mr Sulu, can you explain why i can't have the list of the vessel's rooms.

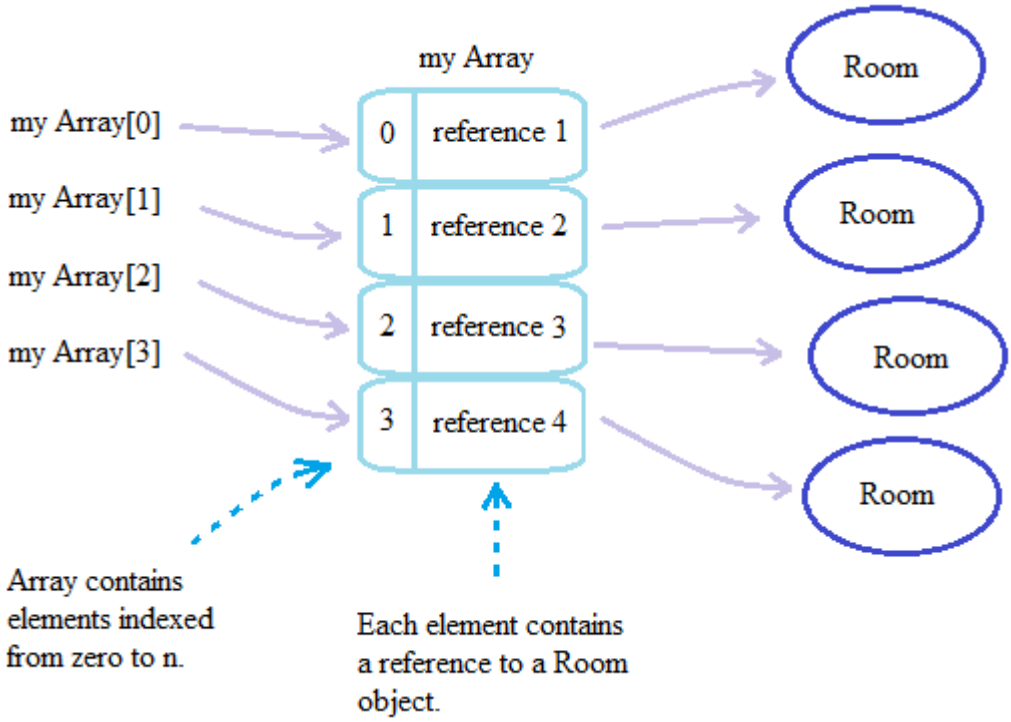
The system give us the A and B decks rooms, but not those of C and D decks.

It's possible that C and D decks work with the former program.

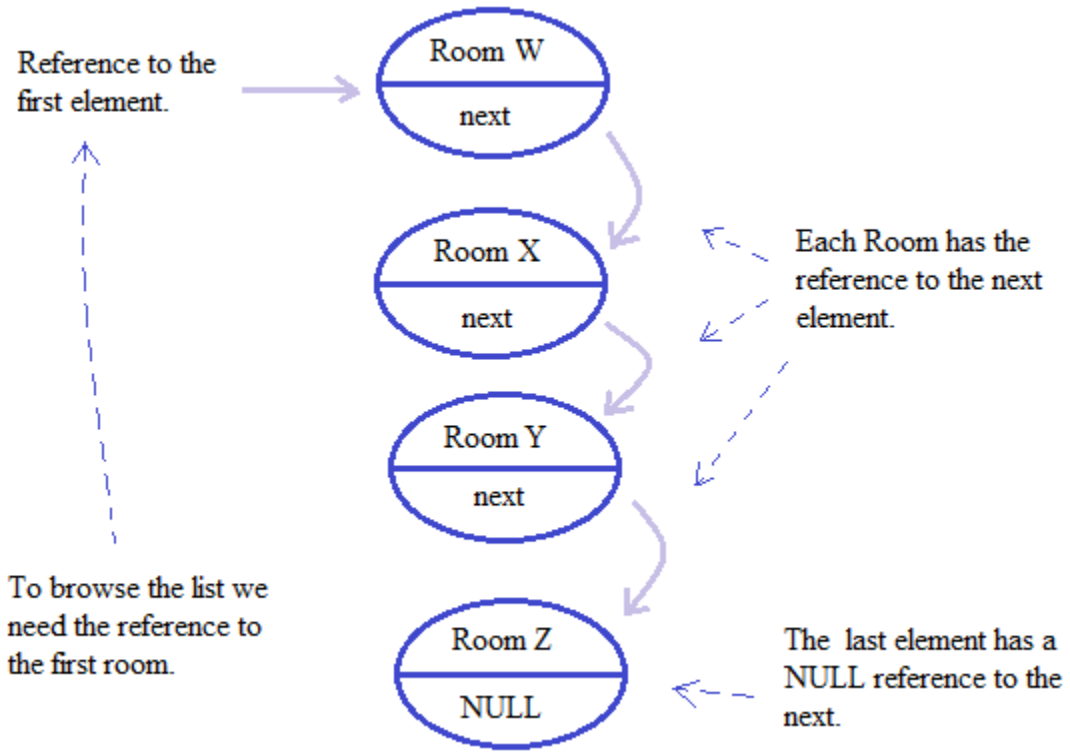
Are we gonna get annoyed by a little compatibility problem Mr Sulu ?

The problem was quickly identified.

Decks A and B are managed by the new program that stores the rooms as references in an array. To get the rooms list you have to walk the array elements by varying the index from zero to n:



Decks C and D are managed by former program that stores the rooms as a linked list. To get the rooms list you must start from the first element and follow the references to the last element:



As our goal is to have a program that can provide a list of all the rooms of the Enterprise (ie bridges A, B, C and D), we are facing a problem of incompatibility.

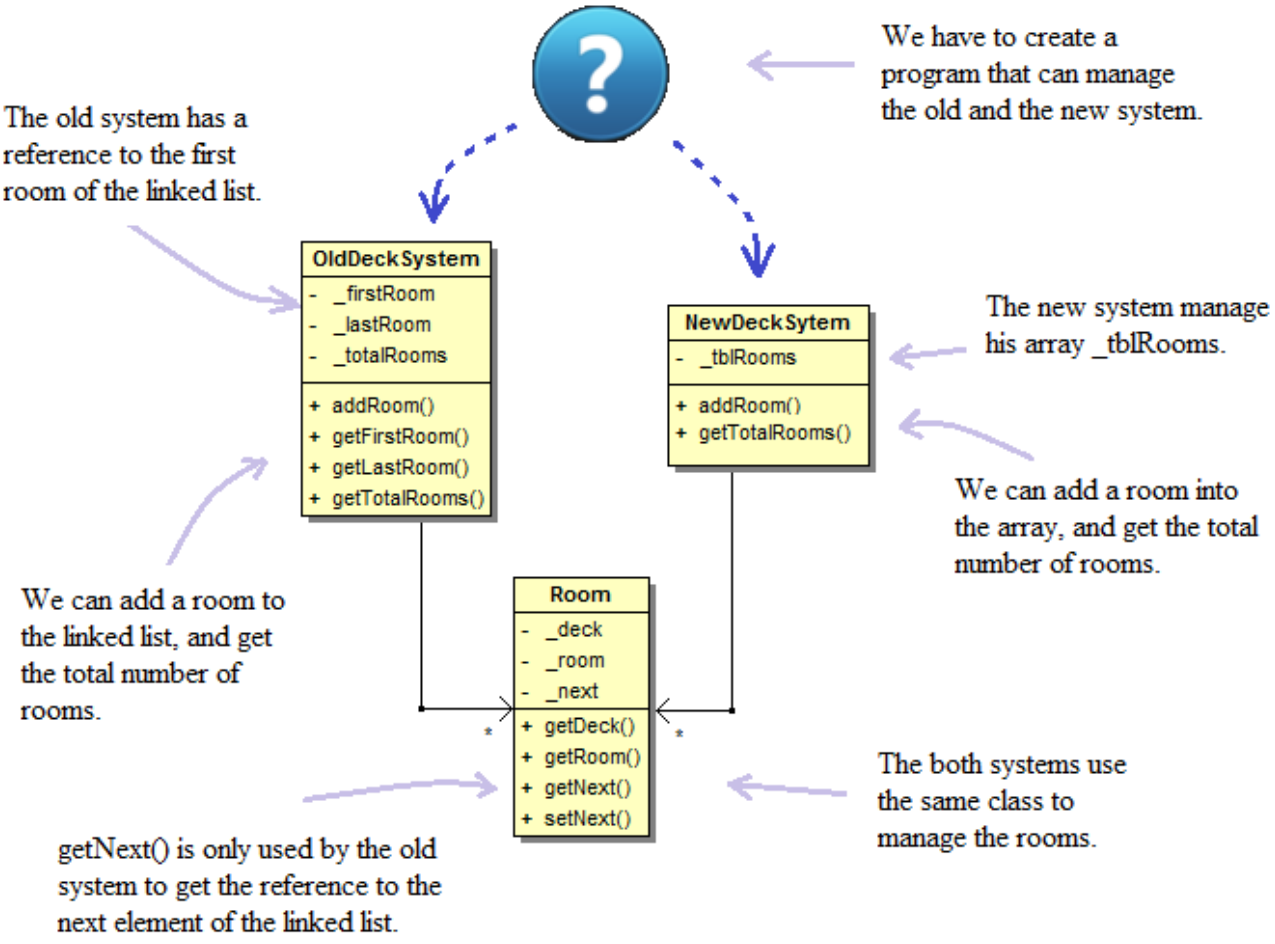
To solve this incompatibility, a first approach lead us to consider two treatments:

- A treatment for decks A and B, knowing iterate over an array structure (the existing treatment).
- Treatment for decks C and D, knowing iterate over a linked list structure.

We can easily guess the drawbacks of this approach:

- If the logic of room scanning change in the future, it would change the two treatments.
- If a new rooms storage structure appears we should add a third treatment.

For these reasons we will not engage in this direction.
Make a point about the current class model:


We have to create a program that can manage the old and the new system.
The old system has a reference to the first room of the linked list.
OldDeckSystem
- _firstRoom
- _lastRoom
- _totalRooms
+ addRoom()
+ getFirstRoom()
+ getLastRoom()
+ getTotalRooms()
NewDeckSytem
- _tblRooms
+ addRoom()
+ getTotalRooms()
The new system manage his array _tblRooms.
We can add a room into the array, and get the total number of rooms.
We can add a room to the linked list, and get the total number of rooms.
Room
- _deck
- _room
- _next
+ getDeck()
+ getRoom()
+ getNext()
+ setNext()
The both systems use the same class to manage the rooms.
getNext() is only used by the old system to get the reference to the next element of the linked list.

15.1 Presentation of the Iterator pattern

We can summarize the Iterator pattern at two points:

- It separates the iteration process, from the thing that is iterated. This means that the class that contains the collection that you want to iterate, does not offer treatment to iterate through this collection. This treatment will be isolated in another class called Iterator.
- The iterator will have a unified interface, ie it always offer at least the following standard methods:
  - hasNext (): This method return TRUE as long as there is still at least one item in the collection.
  - next (): This method return the reference to the next item to go.

How do we use the iterator**?**

- The program wishing to iterate over a collection, must ask the class managing this collection, to provide the reference of its iterator.
- Once the iterator obtained, simply use a standard loop:

```
While myIterator.hasNext()
    myIterator.next()
    // we process the element
    // ...
End
```

It is understood that the program that wants to iterate over the collection, has no idea of how this collection is managed and how it actually iterates over.

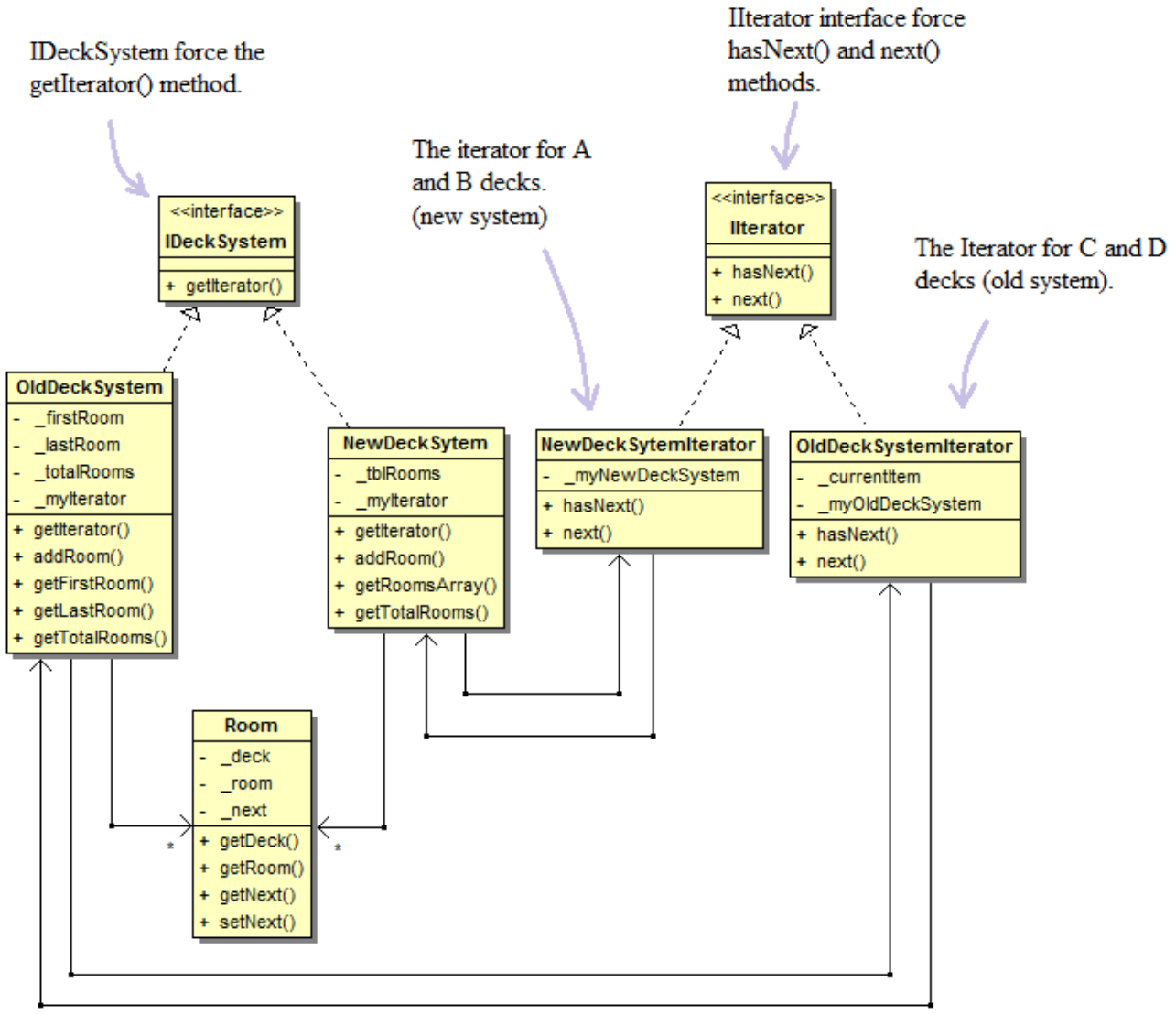
It must simply have the iterator for this collection, and use both hasNext() and next() methods. This is the iterator that actually performs the course of collection.

We now need to change our class model to make it a real Iterator pattern. To do this we must make the following changes:

- We will ensure that OldDeckSystem and NewDeckSytem classes implement a IDeckSystem interface, forcing them to declare a getIterator() method. Indeed, these two classes will have to provide their iterators.

- We will create a IIterator interface forcing to declare the two methods hasNext() and next().
- We will create a NewDeckSytemIterator class that implements the IIterator interface. This will be our iterator to the new system (decks A and B).
- We will create an OldDeckSystemIterator class that implements the IIterator interface. This will be our iterator to the old system (decks C and D).

Here is the new class model:



Remember that we want to make a program that displays a list of all rooms in the Enterprise.

The program should use the desing in the following manner:

- Have a reference to an OldDeckSystem object.
- Have a reference to a NewDeckSystem object.
- Get from OldDeckSystem object, a reference to its iterator using getIterator() method.
- Scan the rooms managed by this iterator using the hasNext() and next() methods.
- Get from NewDeckSystem object, a reference to its iterator using getIterator() method.
- Scan the rooms managed by this iterator using the hasNext() and next() methods.

The code is actually quite simple because the complexity is managed by the iterators.
Let's go to the coding.

15.2 Coding

IDeckSystem interface:

```php
<?php

interface IDeckSystem {

    public function getIterator();
}
?>
```

IIterator interface:

```php
<?php

interface IIterator {

    public function next();
    public function hasNext();

}
?>
```

Room class:

```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;
    }

}

?>
```

The _next attribut is only used by old system (decks C and D).

Method used by old system to define next element in the linked list.

OldDeckSytem class:

```php
<?php

class OldDeckSystem implements IDeckSystem {

    private $_firstRoom;
    private $_lastRoom;
    private $_totalRooms;
    private $_myIterator;

    public function __construct() {
        echo "I'm OldDeckSystem constructor", "<br>";
        $this->_firstRoom = NULL;
        $this->_lastRoom = NULL;
        $this->_totalRooms = 0;

        // We add elements to the linked list:
        $this->addRoom("A", "Control room");
        $this->addRoom("A", "Rest room");
        $this->addRoom("A", "Communication room");
        $this->addRoom("B", "Technical room");

        // We instantiate iterator:
        $this->_myIterator = new OldDeckSystemIterator($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 references the first element in the linked list.

$this will allow iterator to access OldDeckSytem.

That's where list elements are linked together.

```
    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 "I'm NewDeckSystem constructor", "<br>";

        // We add rooms in array:
        $this->addRoom("C", "Infirmary");
        $this->addRoom("C", "Restaurant");
        $this->addRoom("D", "Meeting room");

        // We instantiate iterator:
        $this->_myIterator = new NewDeckSytemIterator($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);
    }

}

?>
```

We add a room to the array.

La classe NewDeckSytemIterator:

```php
<?php

class NewDeckSytemIterator implements IIterator {

    private $_currentIndex;
    private $_totalRooms;
    private $_tblRooms;
    private $_myNewDeckSystem;

    public function __construct($NewDeckSystem) {
        echo "I'm NewDeckSytemIterator constructor", "<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;
    }

}

?>
```

We pass the NewDeckSystem reference to the iterator.

Iterator get the rooms array from NewDeckSystem.

The two methods hasNext() and 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 "I'm OldDeckSystemIterator constructor","<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

As usual we will use index.php as the controller of the whole design.

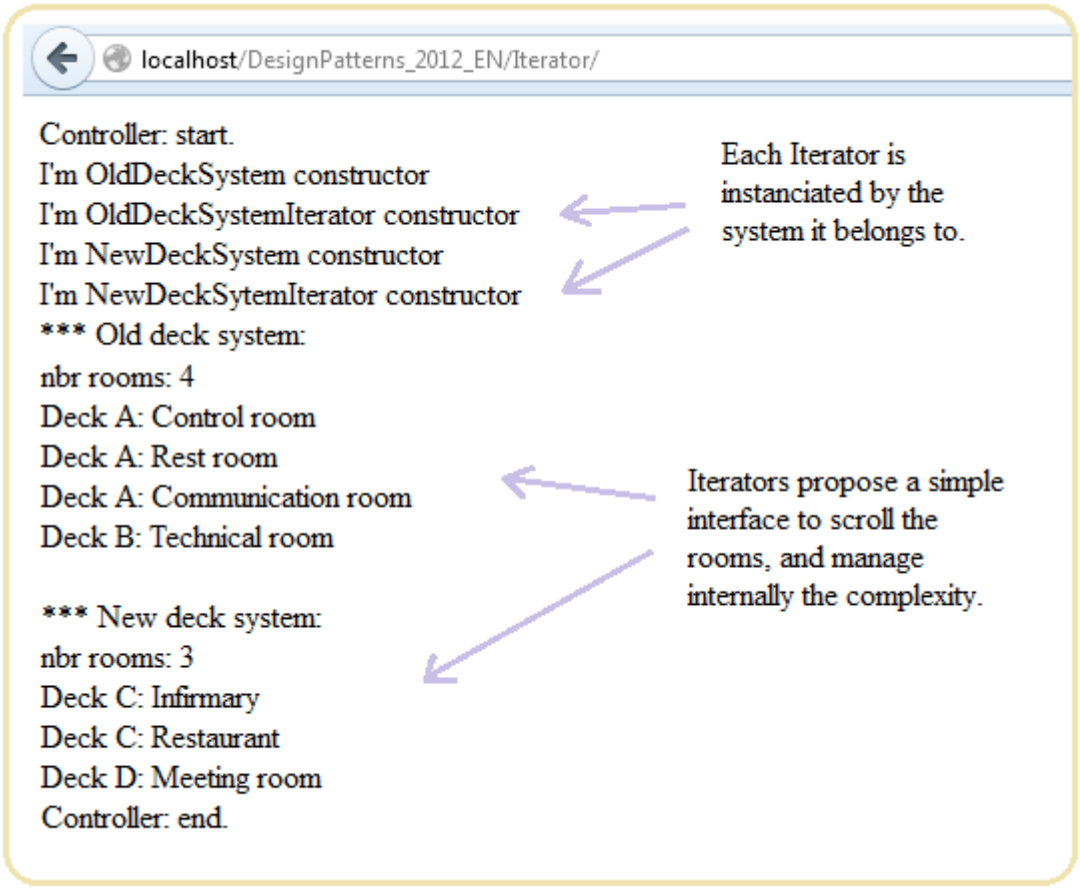
```php
<?php

//*************************************************************
// ITERATOR pattern controler
//*************************************************************

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

echo 'Controller: start.' . $newline;

// Instanciations:
$myOldDeckSystem = new OldDeckSystem();
$myNewDeckSystem = new NewDeckSytem();

echo "*** Old deck system:", "<br>";
echo "nbr rooms: " . $myOldDeckSystem->getTotalRooms() . "<br>";

// We scan the linked list through the iterator:
if ($myOldDeckSystem->getTotalRooms() > 0) {
    DisplayRoomsList($myOldDeckSystem->getIterator());
}

echo "<br>";
echo "*** New deck system:", "<br>";
echo "nbr rooms: " . $myNewDeckSystem->getTotalRooms() . "<br>";

// We scan the array through the iterator:
if ($myNewDeckSystem->getTotalRooms() > 0) {
    DisplayRoomsList($myNewDeckSystem->getIterator());
}

function DisplayRoomsList($iterator) {
    // we use the iterator passed as a parameter:
    while ($iterator->hasNext() == TRUE) {
        $currentRoom = $iterator->next();
        $deck = $currentRoom->getDeck();
        $room = $currentRoom->getRoom();
        echo "Pont ", $deck, ": ", $room, "<br>";
    }
}

echo 'Controller: end.' . $newline;
?>
```

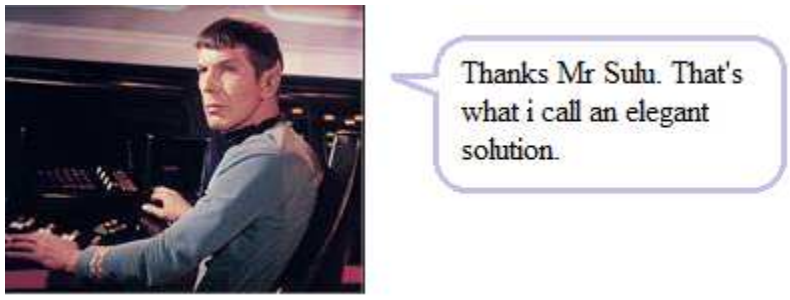
We instanciate the two systems.

We call DisplayRoomsList() passing him old and new systems iterators.

DisplayRoomsList() can use old or new system iterators.

Here is the result at run time:



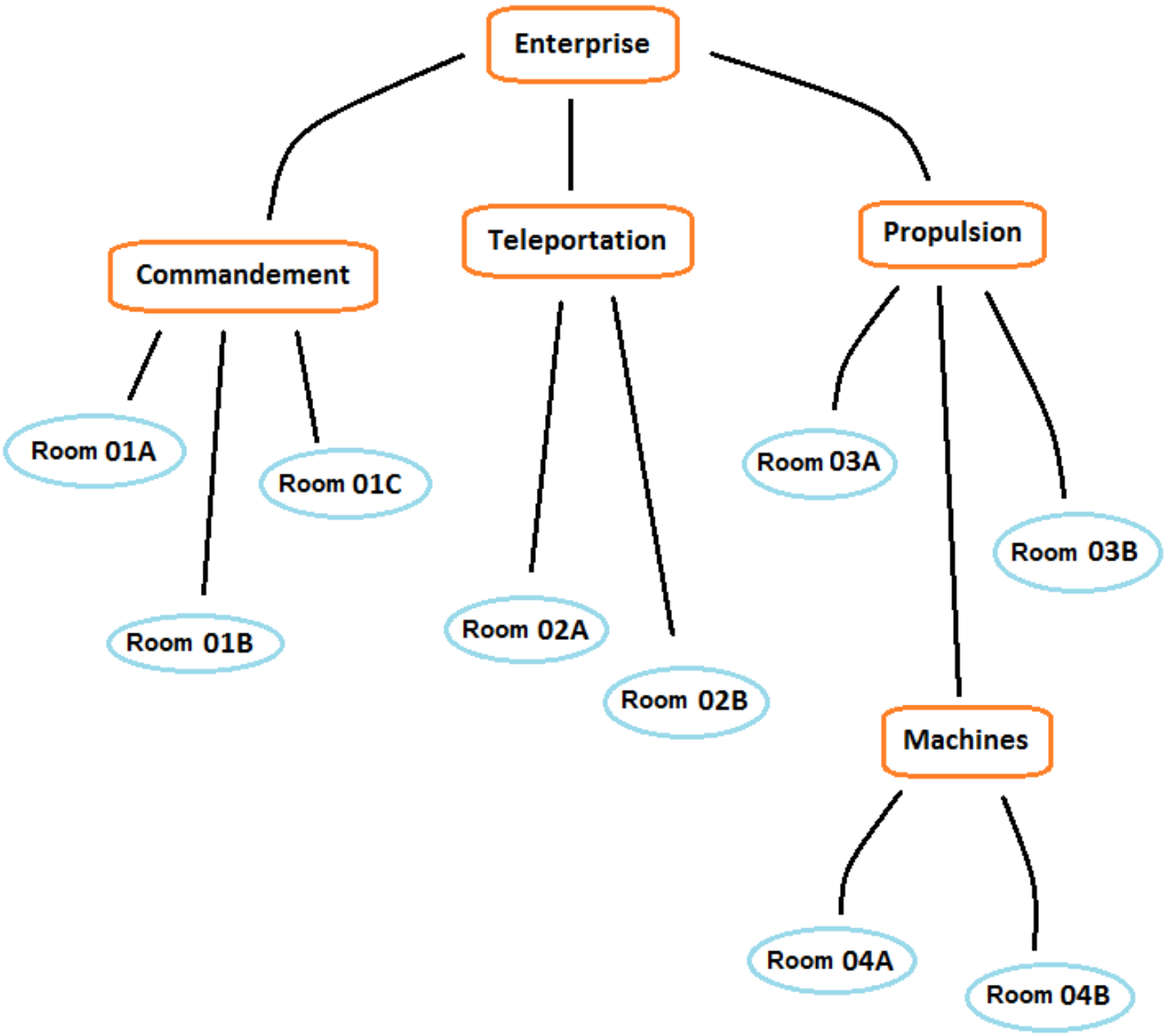


16 COMPOSITE PATTERN

With the Iterator pattern we saw how to encapsulate iteration operations over a collection, and provide a unified interface to facilitate these operations.

Composite pattern in turn, will allow us to iterate on composite tree structures. That means that each element of the tree can be a simple element called leaves or a composite element, the latter may in turn contain leaves or other composite elements.

Here is the tree structure defined by the captain:



The sectors are:

- Commandement
- Teleportation
- Propulsion
- Machines

The Machines sector is included in Propulsion sector.

The rooms are simple elements called leaves.

There is a root element named Enterprise that represents the entire vessel.

Browsing a tree structure uses recursion. This is a particular type of programming in which a

treatment can be performed by itself in a nested manner. See the appendix "Recursion" at the end of document.

16.1 How to browse a tree

Tree structures have their terminology:

- All elements of the tree are called nodes.
- There is always a root node that contains all others. For us it is the Enterprise node.
- Nodes that contain nothing are called leaves. Here they are rooms.
- Children are the nodes immediately beneath a parent node. For example, the Propulsion node has 3 children: Room 03A, Room 03B, and Machines. But not Room 04A and Room 04B.

In the Enterprise tree, we find two types of elements:

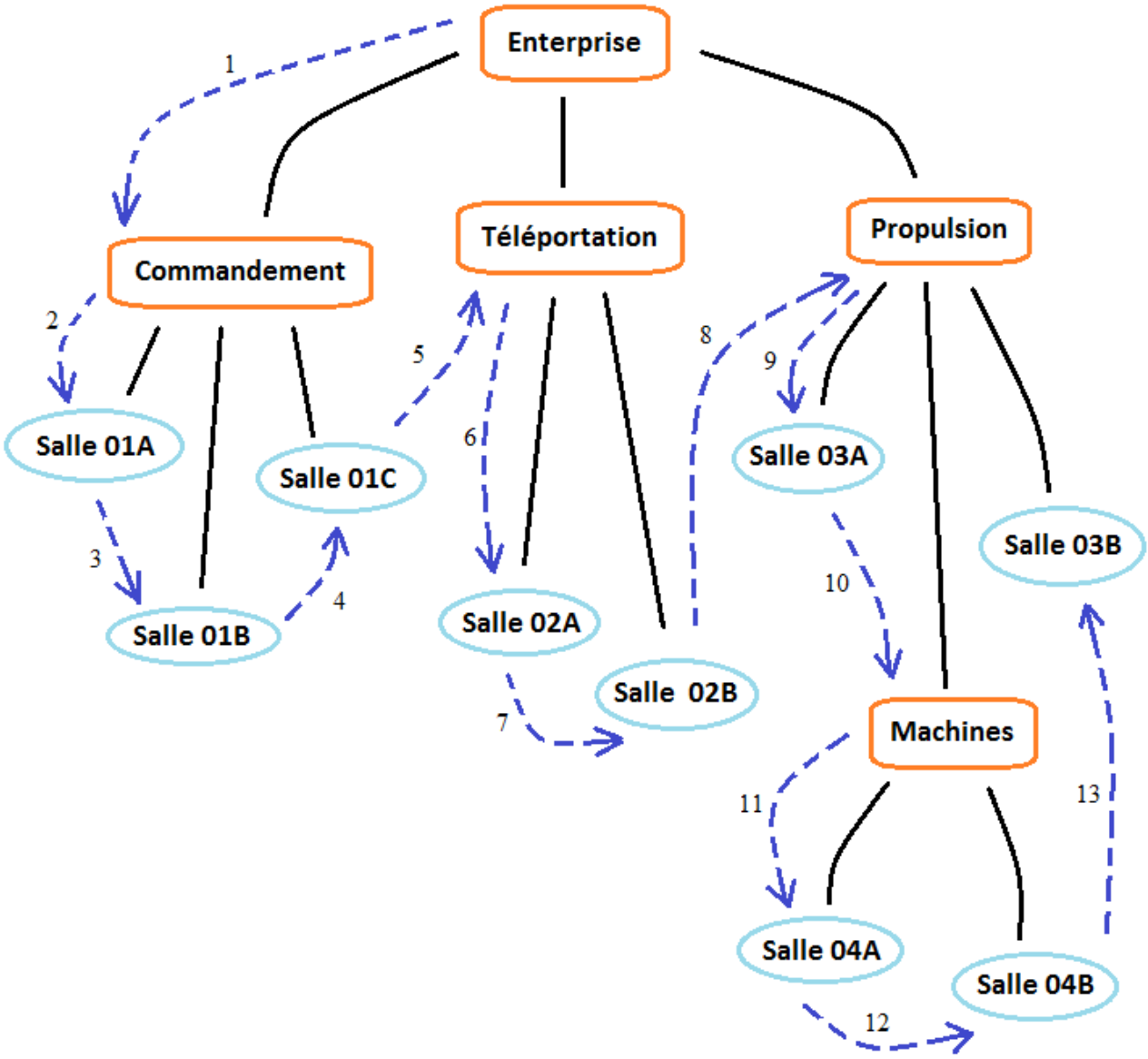
- Rooms: they are simple nodes that contain nothing (leaves).
- Sectors: children of these nodes can be rooms or other sectors. For example, the Teleportation sector contains two rooms. The Propulsion sector contains two rooms and a sector.

Tree scanning starts from the root node.

Children of the root node is traversed, and if a child is a composite (a sector), then we explore immediately by applying the same approach.

Whenever an sector is explored, it is a new exploration process that starts, and the exploration of the parent process is put on wait state. This is the principle of recursion.

According to the rules we have just mentioned, the course of our tree should be the following (follow the blue arrows):



Here is the objective of Composite pattern: browse a tree structure in this way. Finally it is not so complicated !

16.2 How does the Composite pattern work

In this design pattern, two solutions will be proposed for the tree scanning implementation: A and B.

They are two variants equally valid in terms of performance.

Alternative B is a little simpler, so we start with it. In this approach, all classes will be suffixed by "_B".

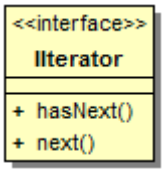
It is very important to have assimilated the Iterator pattern, seen above, because it is used in the Composite pattern.

Indeed, when the Composite pattern traverses the tree and it encounters a sector, it will ask to provide its iterator. This iterator is simply an Iterator pattern.

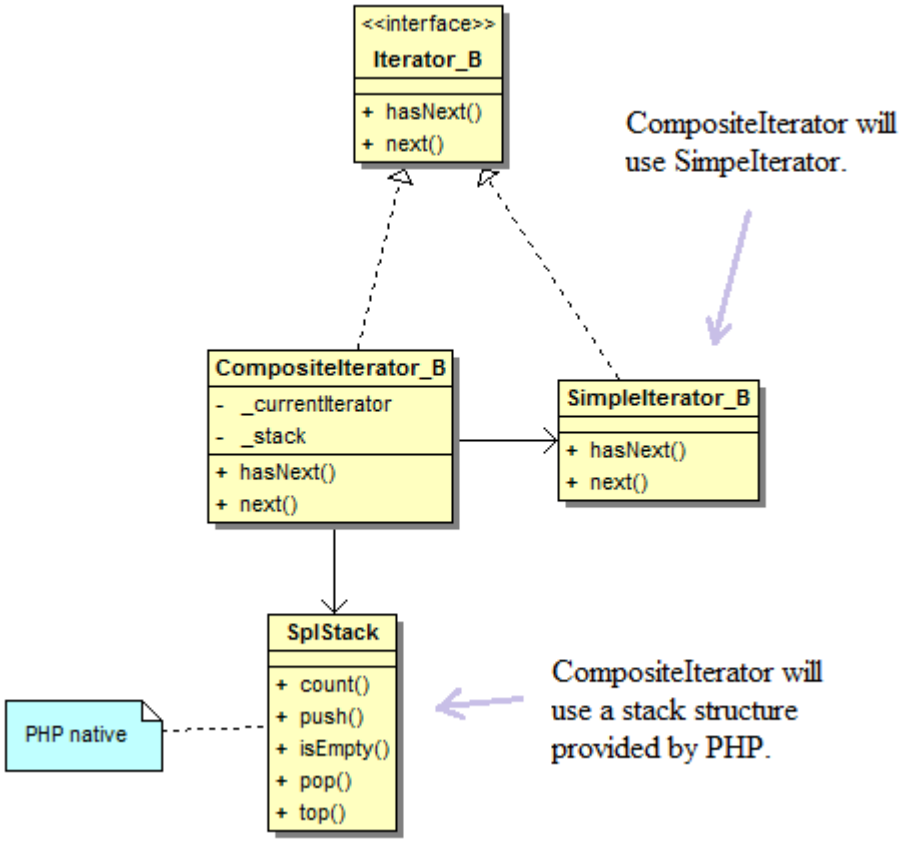
So we guess the use of two types of iterators in the composite pattern:

- The iterator provided by a sector, we call it "SimpleIterator".
- The iterator of the tree as a whole: we call it "CompositeIterator".

Remember in the Iterator pattern, we defined an interface to impose hasNext() and next() methods:



We take this interface in the Composite pattern, and we define two classes that implement it:



SplStack is a class provided by PHP. It will be used here to stack the successive processes of sectors exploration. This is directly related to the recursion.

Instead of a stack, it would have been possible to use a simple array. But the opportunities to use a stack are rare. Do not deprive !

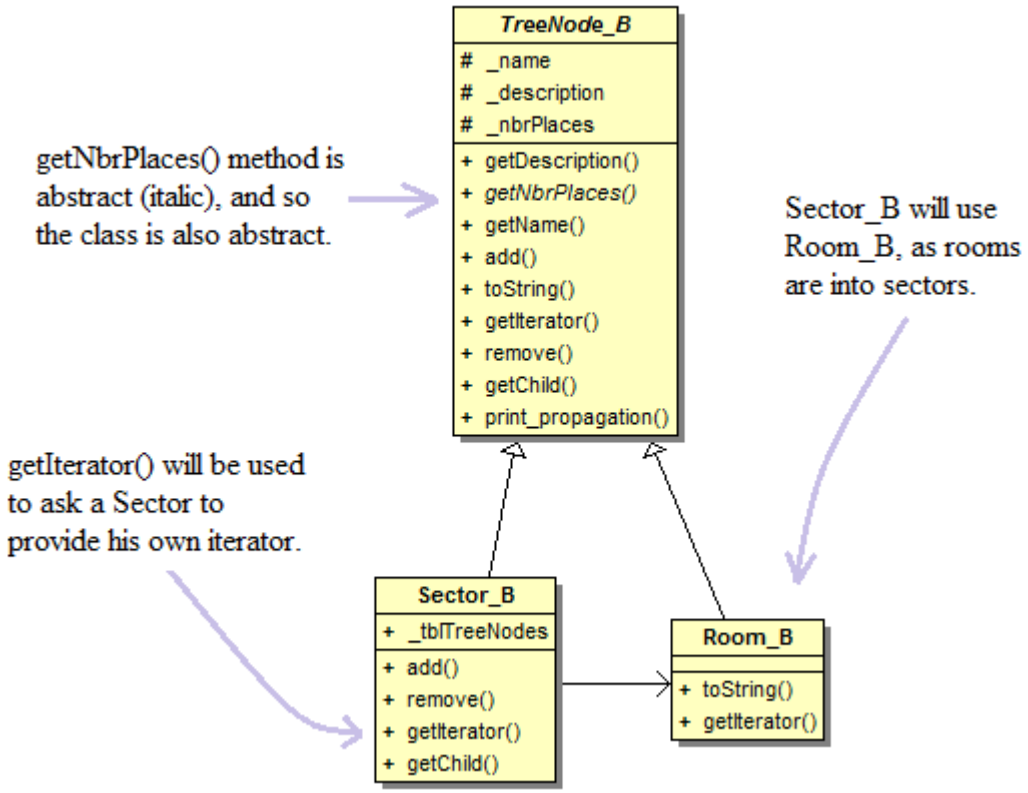
Reminder stacks**:**

The stacks are part of data structure in computer science, as well as variables, arrays or linked lists…

The stack behaves like a stack of plates:

- You can stack an additional element: **push()**.
- You can unstack (remove) the top element: **pop()**.
- You can read the top element without unstacking: **top()**.
- You can tell if the stack is empty: **isEmpty()**.
- You can find out how many elements are in the stack: **count()**.

We also need classes to materialize the tree nodes. Define a derived a TreeNode_B class whose derive Sector_B and Room_B.

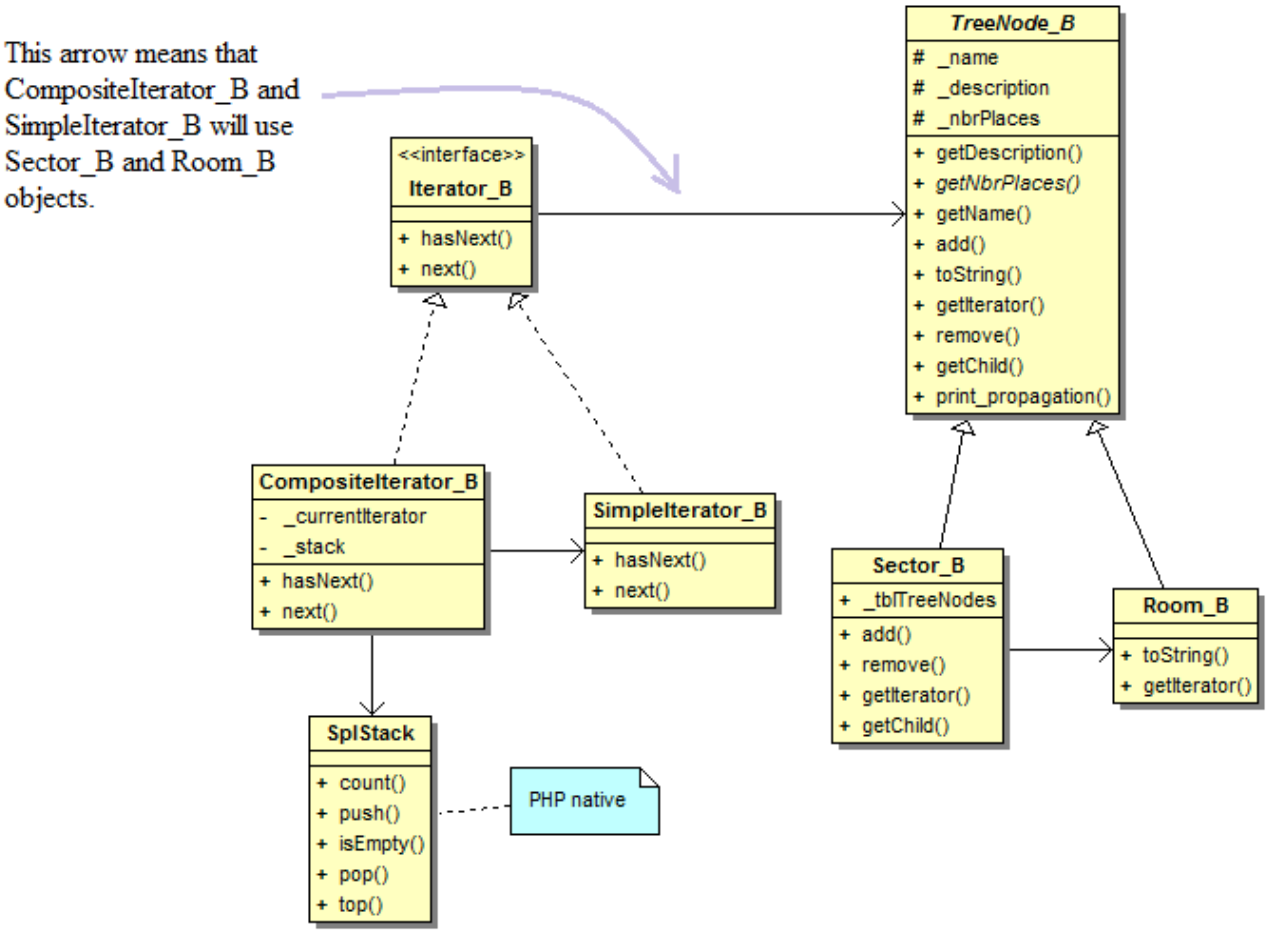


Sector_B represent a sector and Room_B a room.

Some methods defined in TreeNode_B, apply only to Sector_B.

For example, the add() method adds a room to a sector, has no meaning for a room (you can not add a room to another room). To overcome this little problem, a default behavior will be defined in TreeNode_B class for some methods, and these will be overridden, if necessary, in subclasses. We will return by examining the code.

Here is the whole design:



16.3 The B alternative coding

We must dive into the code at some point. We'll start with base classes that do not give any particular problems.
We conclude with CompositeIterator_B class that handles recursion and is the hardest part of this pattern.

TreeNode_B class:

```php
<?php

abstract class TreeNode_B {
    const DEFAULT_ERROR_MESSAGE = "Method not supported by this class."; .";

    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 $this->__toString();
        // We must invoke print_propagation() method against children:
        foreach ($this->_tblTreeNodes as $key => $component) {
            $component->print_propagation();
        }
    }

    public function __toString() {
        // Default behavior:
        return $this->_name . ": " . $this->_description . '</br>';
    }

}

?>
```

When creating a node, we give him a name and description.

Those methods are abstract cause they will have different implementations in Sector_B and Room_B.

Those methods are only relevant to Sector_B.

We'll describe this method later.

The add(), remove() and getChild() methods have a default implementation which returns an error. This implementation will be the inherited in Room_B.

But in Sector_B we redefine these three methods because they make sense in this class.

Sector_B class :

```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] === $treeNode) {
                // array cell deletion:
                unset($this->_tblTreeNodes[$key]);
            }
        } // end foreach
    }

    public function getChild($indice) {
        return $this->_tblTreeNodes[$indice];
    }

    // We scan children to cumulate number of places:
    // This method is recursive if many sectors are
    // nested.
    public function getNbrPlaces() {
        $Compteur = 0;
        foreach (array_keys($this->_tblTreeNodes) as $key) {
            $Compteur = $Compteur + $this->_tblTreeNodes[$key]->getNbrPlaces();
        } // end foreach
        return $Compteur;
    }

    // we overwrite the method for a custom view:
    public function __toString() {
        return "*** " . $this->_name . ": " . $this->_description . " " .
                $this->getNbrPlaces() . " places" . '</br>';
    }

}

?>
```

Sector handle his children into a simple array.

That's how Sector provide his iterator.

Those methods are usefull but will not be used in our example.

That's how Sector cumulate number of places of his children.

Room_B class:

```php
<?php

class Room_B extends TreeNode_B {

    private $_nbrPlaces;

    public function __construct($name = "", $description = "", $nbrPlaces = 0) {
        parent::__construct($name, $description);
        $this->_nbrPlaces = $nbrPlaces;
    }

    // we overwrite the method for a custom view:
    public function __toString() {
        return "   " . "* " . $this->_name . ": " .
                $this->_description . " " . $this->_nbrPlaces .
                " places" . '</br>';
    }

    // we overwrite the method
    public function print_propagation() {
        echo $this->__toString();
    }

    public function getNbrPlaces() {
        return $this->_nbrPlaces;
    }

    public function getIterator() {
        return NULL;
    }

}

?>
```

Room is build with a name, description, and number of places.

We'll come back to this method later.

Returning NULL will allow to identify Sectors from Rooms.

The Iterator_B interface: it is the same interface as the Iterator pattern.

```php
<?php

interface Iterator_B {

    public function hasNext();

    public function next();
}

?>
```

SplStack class: see the following URL:

http://php.net/manual/en/class.splstack.php

SimpleIterator_B class: almost identical to Iterator pattern class.

```php
<?php

class SimpleIterator_B implements Iterator_B {

    private $_curentPosition = 0; // array index. Start at 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++; // we increment
        return $treeNode;
    }

}

?>
```

SimpleIterator is build passing him the Sector on which he must itérate.

SimpleIterator has direct access to Sector children array.

CompositeIterator_B class:

```php
<?php

class CompositeIterator_B implements Iterator_B {

    private $_myStack;

    public function __construct($simpleIterator) {
        $this->_myStack = new SplStack();
        $this->_myStack->push($simpleIterator);
        echo "I'm CompositeIterator_B" . "<br>";
    }

    public function hasNext() {

        if ($this->_myStack->isEmpty()) {
            echo "Stack is empty.", "<br>";
            return FALSE;
        } else {
            echo $this->_myStack->count() .
                    " element(s) in stack." . "<br>";
            if ($this->_myStack->top()->hasNext() == FALSE) {
                echo "Finish for this simpleIterator.", "<br>";
                // we delete top element:
                $this->_myStack->pop();
                // recursion is here:
                return $this->hasNext();
            } else {
                return TRUE;
            }
        } // if
    }

    public function next() {
        if ($this->hasNext() == TRUE) {

            $myTreeNode = $this->_myStack->top()->next();

            // Is it a leaf or not ?
            if (!$myTreeNode->getIterator() == NULL) {
                // It's a sector, so
                // we add a new SimpleItérateur in stack:
                $this->_myStack->push($myTreeNode->getIterator());
            }
            return $myTreeNode;
        } else {
            return null;
        } // if
    }

}

?>
```

A unique CompositeIterator will be instanciated to scan the tree. We give him the SimpleIterator of the tree root.

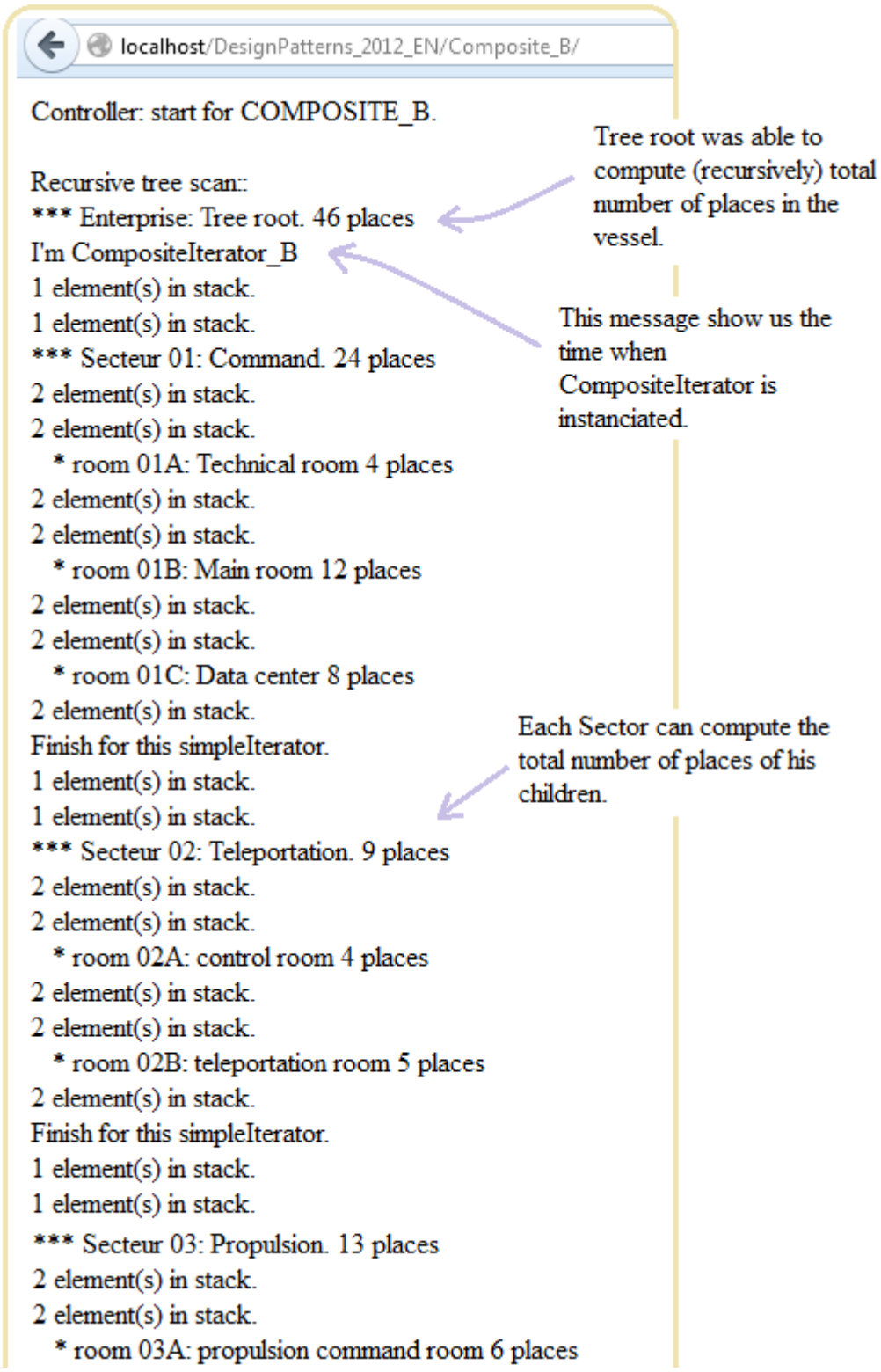
_myStack contains only references to SimpleIterators. So here we call the next() method on the SimpleIterator which is on the top of the stack.

CompositeIterator_B code is the heart of Composite pattern. It manage the complexity of recursive tree scanning.
Now see the code that will use the entire design. As usual we use index.php:

```php
<?php

//************************************************************
// COMPOSITE_B pattern controller
//************************************************************

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

echo 'Controller: start for COMPOSITE_B.', $newline;

// Sectors Instanciations:
$racine = new Sector_B("Enterprise", "Tree root.");
$secteur_01 = new Sector_B("Secteur 01", "Command.");
$secteur_02 = new Sector_B("Secteur 02", "Teleportation.");
$secteur_03 = new Sector_B("Secteur 03", "Propulsion.");
$secteur_04 = new Sector_B("Secteur 04", "Engines.");

$racine->add($secteur_01);
$racine->add($secteur_02);
$racine->add($secteur_03);

// Room instanciations:
$secteur_01->add(new Room_B("room 01A", "Technical room", 4));
$secteur_01->add(new Room_B("room 01B", "Main room", 12));
$secteur_01->add(new Room_B("room 01C", "Data center", 8));

$secteur_02->add(new Room_B("room 02A", "control room", 4));
$secteur_02->add(new Room_B("room 02B", "teleportation room", 5));

$secteur_03->add(new Room_B("room 03A", "propulsion command room", 6));
$secteur_03->add($secteur_04);
$secteur_03->add(new Room_B("room 03B", "test room", 6));

$secteur_04->add(new Room_B("room 04A", "Lithium cristal compartment", 0));
$secteur_04->add(new Room_B("room 04B", "decontamination room", 1));

//*********************************
// Recursive tree scan:
//*********************************

echo '</br>', "Recursive tree scan::", $newline;
echo $racine; // calls __toString() method.

$myCompositeIterator = new CompositeIterator_B($racine->getIterator());
while ($myCompositeIterator->hasNext()) {
    $myNode = $myCompositeIterator->next();
    echo $myNode; // calls __toString() method.
} // end while

echo '--------------', $newline;
echo 'Controller: end.', $newline;
?>
```

It's very simple to use the CompositeIterator.

And here is the result:



```
   * room 03A: propulsion command room 6 places
2 element(s) in stack.
2 element(s) in stack.
*** Secteur 04: Engines. 1 places
3 element(s) in stack.
3 element(s) in stack.
   * room 04A: Lithium cristal compartment 0 places
3 element(s) in stack.
3 element(s) in stack.
   * room 04B: decontamination room 1 places
3 element(s) in stack.
Finish for this simpleIterator.
2 element(s) in stack.
2 element(s) in stack.
   * room 03B: test room 6 places
2 element(s) in stack.
Finish for this simpleIterator.
1 element(s) in stack.
Finish for this simpleIterator.
Stack is empty.
--------------
Controller: end.
```

To understand the way CompositeIterator works, the better is to run it "by hand":

- Draw a tree on a piece of paper, with sectors and rooms, or take the tree exemplary early in Composite pattern.
- Make sure there is at least one sector that contains a sub-sector, as this is a significant test case.
- Save a corner of the paper to represent the stack. Its content will evolve during the course of the tree.
- Follow the code of the different classes as if you were the computer.
- Arm yourself with patience because this part is particularly difficult.

16.4 Can we go further?

As seen in our index.php controller using CompositeIterator is very simple.

Since each node is really recovered by the controller, we can imagine all sorts of treatment on them.

If we want for example, display the list of rooms that have 5 or more places, just add a test in the

loop in index.php:

```
//*********************************************************
// Rooms list having more than 5 places:
//*********************************************************

echo '</br>', "// Rooms list having more than 5 places: ", $newline;
echo $racine;  // calls __toString() method.

$myCompositeIterator = new CompositeIterator_B($racine->getIterator());
while ($myCompositeIterator->hasNext()) {
    $myNode = $myCompositeIterator->next();
    if ($myNode->getIterator() == NULL) {  // if it's a room.
        if ($myNode->getNbrPlaces() >= 5) {
            echo $myNode;  // calls __toString() method.
        }
    }
} // end while
```

We completed the study of variant B Composite pattern. Before turning to alternative A (slightly more complicated) let's examine a kind of "automatic" tree scanning that requires very little code and can, in some cases, be sufficient.

16.5 Automatic propagation scanning

In this type of scanning, we call a method on the root node, and we ensure that this appeal be propagated to all nodes of the tree.
It is a form of "automatic" recursion that spreads based on parent-child relations.

Advantages:

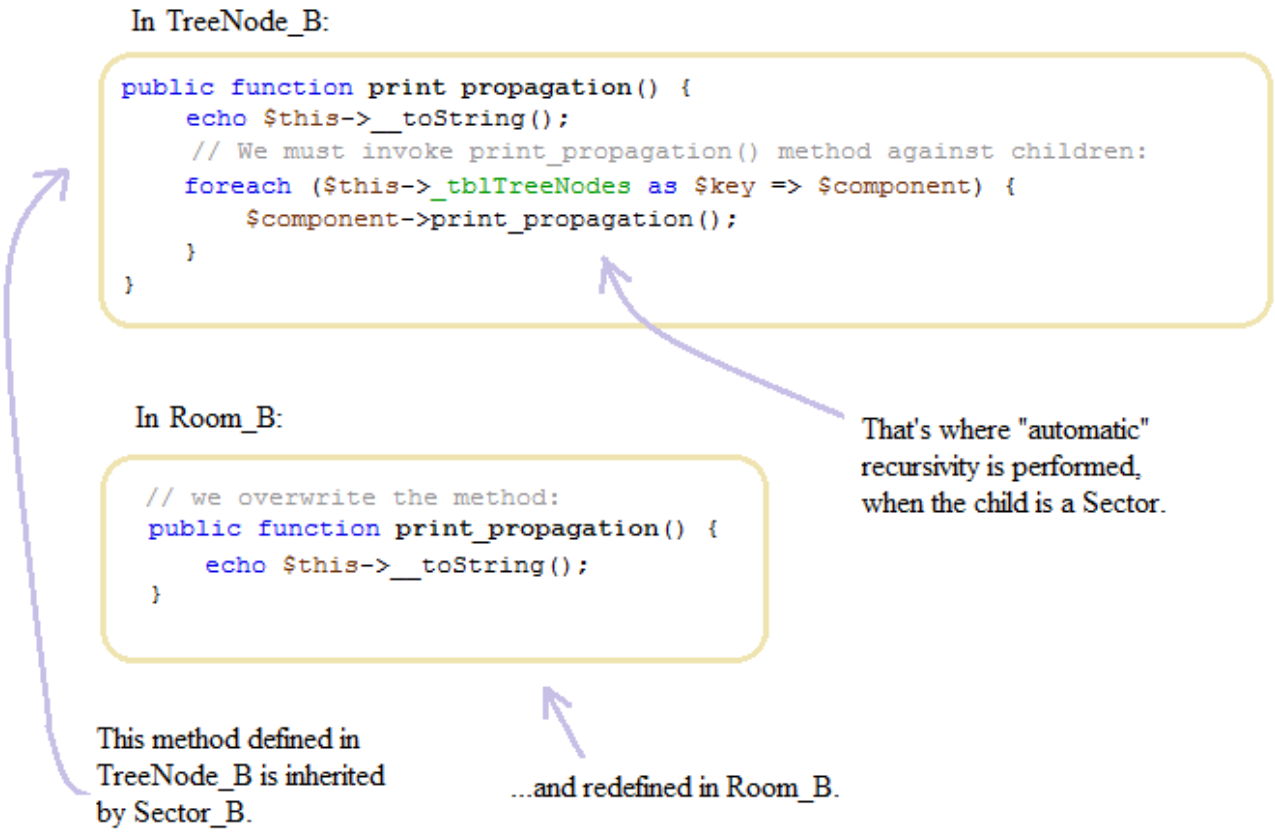
- There is very little code to write, and it is very simple.
- We do not use any iterators.
-

The disadvantages:

- The controller that run this treatment (for us index.php) does not recover each node of the tree as in variant B above or variant A which will be presented later. As nodes are not recovered, no treatment or filtering is possible on them.
- We must add a method in Sector_B and Room_B classes.

Let's see how does this type of scanning works.

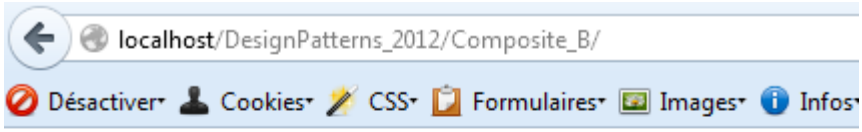
The print_propagation() method has been added to TreeNode_B and Room_B classes:



To run this propagation scanning, just add this code in our index.php controller:

```
//*************************************************************
// Rooms list by method call propagation:
//*************************************************************
echo '</br>', "Tree scan by method call propagation:", '</br>';
$racine->print_propagation();
```

And here is the result:




```
Controller: start for COMPOSITE_B.

Tree scan by method call propagation:
*** Enterprise: Tree root. 46 places
*** Secteur 01: Command. 24 places
  * room 01A: Technical room 4 places
  * room 01B: Main room 12 places
  * room 01C: Data center 8 places
*** Secteur 02: Teleportation. 9 places
  * room 02A: control room 4 places
  * room 02B: teleportation room 5 places
*** Secteur 03: Propulsion. 13 places
  * room 03A: propulsion command room 6 places :s
*** Secteur 04: Engines. 1 places
  * room 04A: Lithium cristal compartment 0 places :es
  * room 04B: decontamination room 1 places
  * room 03B: test room 6 places
--------------
Controller: end.
```

Now let’s go to the A variant of the Composite pattern.

16.6 The A alternative

In A alternative, the same classes are used and are suffixed with "_A".

Two classes are changed:

- CompositeIterator_A: hasNext() and next() method.
- Sector_A: getIterator() method.

The tree scanning no longer uses stack. Instead, several CompositeIterators are instantiated and linked along the progress in the tree.

The change in Sector_A is related to getIterator() method. In B alternative, this method returns a new SimpleIterator. In A , it must return a CompositeIterator_A that encapsulates itself a SimpleIterator.

We will come back to it when examining the code. Before that, look at how does A alternative works.

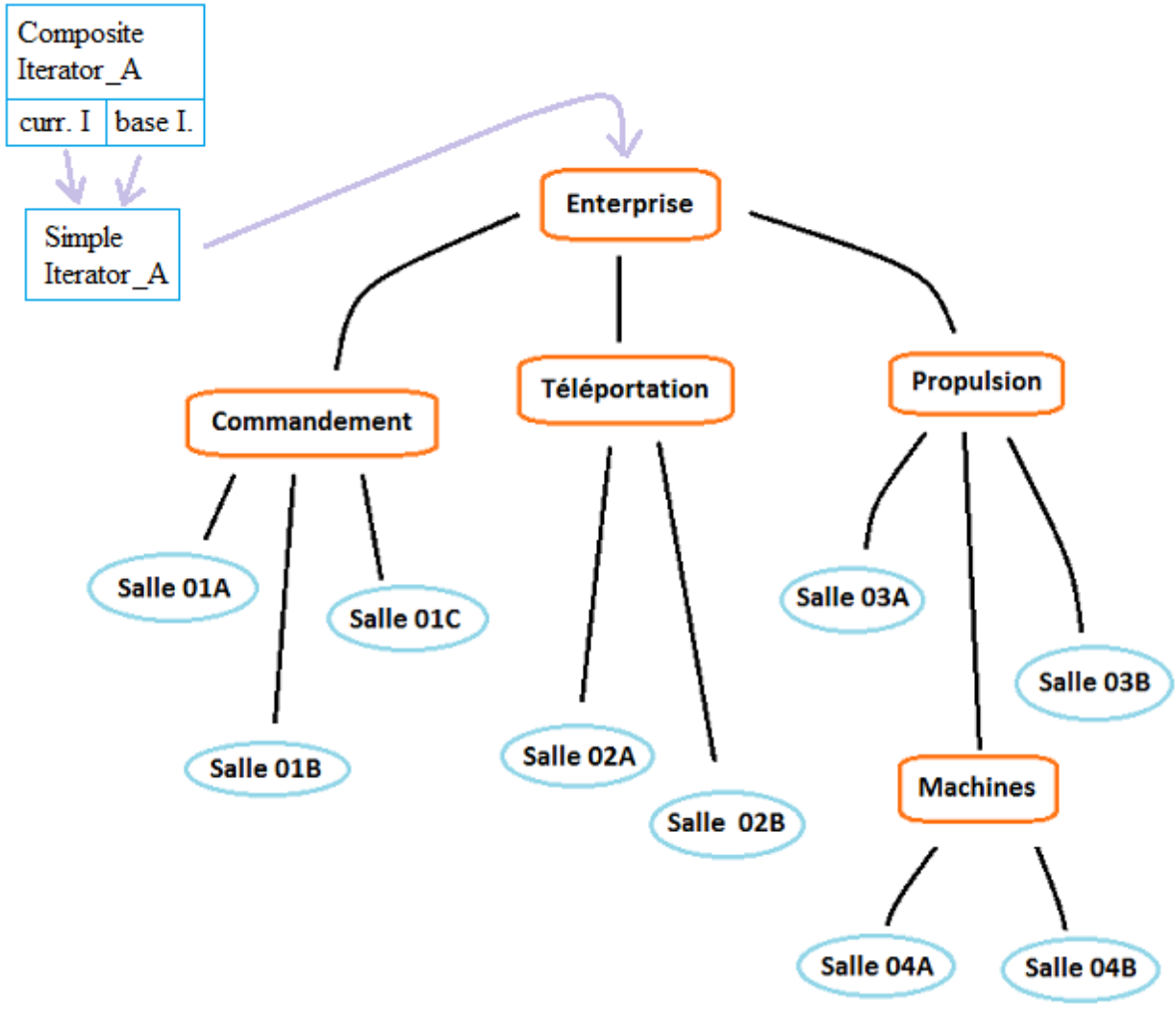
The figure below shows the situation after the first instantiation of CompositeIterator_A pointing to the root.

A CompositeIterator_A is always accompanied by a SimpleIterator_A, they are instantiated at the same time by Sector_A.

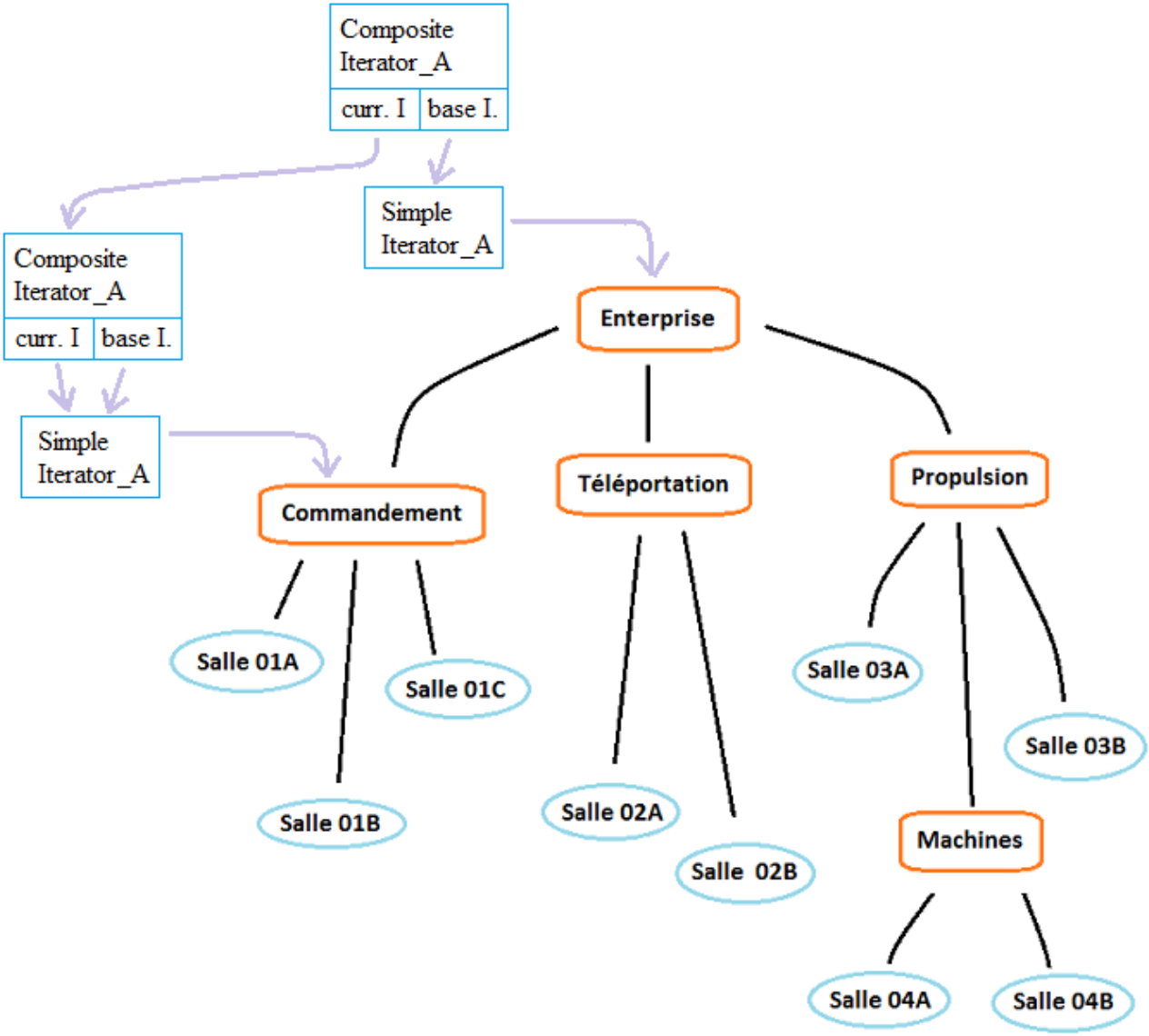
"base I." represents the private variable $ _baseIterator.

"cur. I." represents the private variable $_currentIterator.

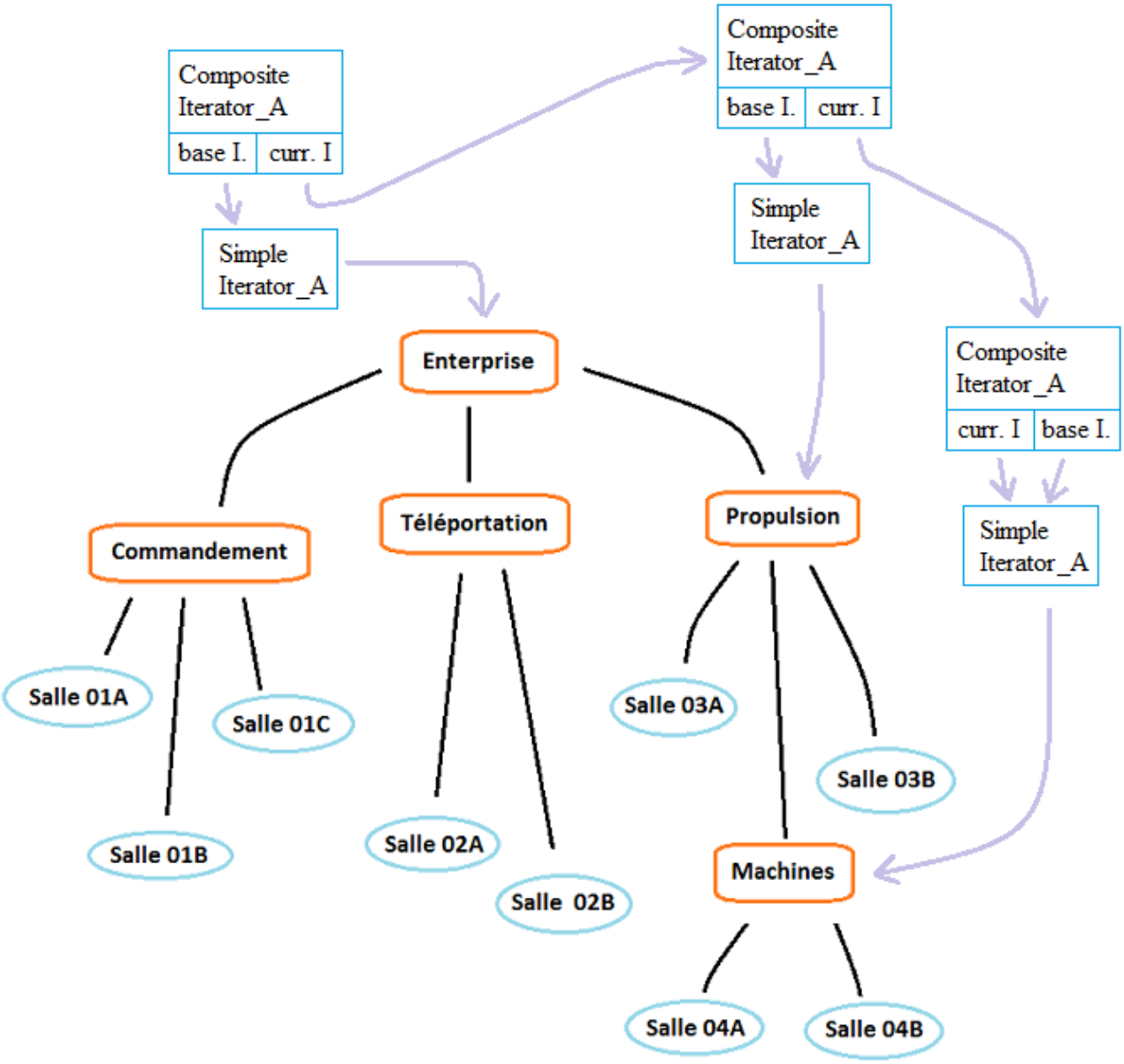
These two variables are used by CompositeIterator_A. We will come back to it when examining code.



The figure below shows the situation after the instantiation of the second CompositeIterator_A pointing to the node "Commandement ".



The figure below shows the situation towards the end of the tree scanning, when the last CompositeIterator_A is pointing on the “Machine” node.



16.7 A alternative coding

The Iterator_A interface: no change.

```php
<?php

interface Iterator_A {

    public function hasNext();

    public function next();
}

?>
```

The TreeNode_A class: no change:

```php
<?php

abstract class TreeNode_A {
    const DEFAULT_ERROR_MESSAGE = "Method not supported by this class.";

    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() {
        // Default behavior:
        throw new Exception(TreeNode_A::DEFAULT_ERROR_MESSAGE);
    }

    public function getIterator() {
        // Default behavior:
        throw new Exception(TreeNode_A::DEFAULT_ERROR_MESSAGE);
    }

    public function remove() {
        // Default behavior:
        throw new Exception(TreeNode_A::DEFAULT_ERROR_MESSAGE);
    }

    public function getChild() {
        // Default behavior:
        throw new Exception(TreeNode_A::DEFAULT_ERROR_MESSAGE);
    }

    public function print_propagation() {
        $this->__toString();
                __
        // We must invoke print_propagation() method against children:
        foreach ($this->_tblTreeNodes as $key => $component) {
            $component->print_propagation();
        }
    }

    public function __toString() {
        echo $this->_name, ": ", $this->_description, '</br>';
    }

}

?>
```

The Room_B class: no change:

```php
<?php

class Room_A extends TreeNode_A {

    private $_nbrPlaces;

    public function __construct($name = "", $description = "", $nbrPlaces = 0) {
        parent::__construct($name, $description);
        $this->_nbrPlaces = $nbrPlaces;
    }

    // we overwrite the method for a custom view:
    public function __toString() {
        return "   " . "* " . $this->_name . ": " .
                $this->_description . " " . $this->_nbrPlaces .
                " places" . '</br>';
    }

    // we overwrite the method:
    public function print_propagation() {
        echo $this->__toString();
    }

    public function getNbrPlaces() {
        return $this->_nbrPlaces;
    }

    public function getIterator() {
        return NULL;
    }

}

?>
```

The Sector_A class: getIterator() method is changed.

```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] === $treeNode) {
                // array cell deletion:
                unset($this->_tblTreeNodes[$key]);
            }
        } // end foreach
    }

    public function getChild($indice) {
        return $this->_tblTreeNodes[$indice];
    }

    // We scan children to cumulate number of places:
    // This method is recursive if many sectors are
    // nested.
    public function getNbrPlaces() {
        $Compteur = 0;
        foreach (array_keys($this->_tblTreeNodes) as $key) {
            $Compteur = $Compteur + $this->_tblTreeNodes[$key]->getNbrPlaces();
        } // end foreach
        return $Compteur;
    }

    // we overwrite the method for a custom view:
    public function __toString() {
        return "*** " . $this->_name . ": " . $this->_description . " " .
                $this->getNbrPlaces() . " places" . '</br>';
    }

}

?>
```

This method returns now a CompositeIterator which encapsulate a SimpleIterator.

The SimpleIterator_A class: no change:

```php
<?php

class SimpleIterator_A implements Iterator_A {

    private $_curentPosition = 0; // array index. Start at 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++; // we increment
        return $treeNode;
    }

    public function getSecteurName() {
        return $this->_mySecteur->getName();
    }

}

?>
```

CompositeIterator_A class: several changes:

```php
<?php

class CompositeIterator_A implements Iterator_A {

    private $_currentIterator;
    private $_baseIterator;

    public function __construct($simpleIterator) {
        $this->_baseIterator = $simpleIterator;
        $this->_currentIterator = $simpleIterator;
        echo "I'm CompositeIterator_A for " .
            $this->getSecteurName() . "<br>";
    }

    public function hasNext() {
        // This method is recursive if several
        // CompositeIterator are linked.
        if ($this->_currentIterator->hasNext()) {
            return true;  // on quitte la méthode.
        }

        if ($this->_currentIterator == $this->_baseIterator) {
            echo "Finish for this SimpleIterator: " .
                $this->_currentIterator->getSecteurName() . "<br>";
            return false; // on quitte la méthode.
        }
        // If we get here is that the exploration of the sub-sector
        // is complete. So we continue the exploration for current sector:
        $this->_currentIterator = $this->_baseIterator;
        return $this->_currentIterator->hasNext();
    }

    public function next() {
        // This method is recursive if several
        // CompositeIterator are linked.
        if ($this->hasNext()) {
            echo "hasNext(): secteur " . $this->getSecteurName() .
                ": there are more !" . "<br>";
            $myTreeNode = $this->_currentIterator->next();
            if ($myTreeNode instanceof Sector_A) {    // if it's a sector.
                if ($this->_currentIterator == $this->_baseIterator) {
                    // recursion is here:
                    $this->_currentIterator = $myTreeNode->getIterator();
                }
            }
            return $myTreeNode;
        } else {
            return null;
        } // end if
    }
```

Some displays are included in the code to follow the tree scan progress.

currentIterator references either another CompositeIterator, or a SimpleIterator if we are at the chain end.

getIterator() will create a new couple CompositeIterator - SimpleIterator.

```
    public function getSecteurName() {
        return $this->_currentIterator->getSecteurName();
    }

}

?>
```

Again to understand the process through the tree, the best is to run it by hand on a sheet of paper, with a good dose of patience.

Now let's see the code that will use the whole design. We use the usual index.php:

```php
<?php

//************************************************************
// COMPOSITE_A pattern controler
//************************************************************

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

echo 'Controller: start run for COMPOSITE_A.', $newline;

// Sectors instanciations:
$racine = new Sector_A("Enterprise", "Tree root.");
$secteur_01 = new Sector_A("Secteur 01", "Command.");
$secteur_02 = new Sector_A("Secteur 02", "Teleportation.");
$secteur_03 = new Sector_A("Secteur 03", "Propulsion.");
$secteur_04 = new Sector_A("Secteur 04", "Engines.");

$racine->add($secteur_01);
$racine->add($secteur_02);
$racine->add($secteur_03);

// Room instanciations:
$secteur_01->add(new Room_A("room 01A", "Technical room", 4));
$secteur_01->add(new Room_A("room 01B", "Main room", 12));
$secteur_01->add(new Room_A("room 01C", "Data center", 8));

$secteur_02->add(new Room_A("room 02A", "control room", 4));
$secteur_02->add(new Room_A("room 02B", "teleportation room", 5));

$secteur_03->add(new Room_A("room 03A", "propulsion command room", 6));
$secteur_03->add($secteur_04);
$secteur_03->add(new Room_A("room 03B", "test room", 6));

$secteur_04->add(new Room_A("room 04A", "Lithium cristal compartment", 0));
$secteur_04->add(new Room_A("room 04B", "decontamination room", 1));

//*********************************
// Recursive tree scan:
//*********************************

echo '</br>', "Recursive tree scan:", $newline;
echo $racine; // calls __toString() method.

$myCompositeIterator = new CompositeIterator_A(new SimpleIterator_A($racine));
while ($myCompositeIterator->hasNext()) {
    $myNode = $myCompositeIterator->next();
    echo $myNode; // calls __toString() method.
} // end while

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

Here is the result:

```
hasNext(): sector Secteur 04: there are more !
hasNext(): sector Secteur 04: there are more !
hasNext(): sector Secteur 04: there are more !
  * room 04B: decontamination room 1 places
Finish for this SimpleIterator: Secteur 04
hasNext(): sector Secteur 03: there are more !
hasNext(): sector Secteur 03: there are more !
  * room 03B: test room 6 places
Finish for this SimpleIterator: Secteur 03
--------------
Controller: end.
```

Here, three CompositeIterator are linked.

Room 03B is displayed after Sector 04 rooms.

We always have the possibility of testing on each node of the tree as we did in Alternative B: list all rooms having 5 or more seats.

Here the loop to add into index.php:

```php
//*****************************************
// Rooms list having more than 5 places:
//*****************************************

echo '</br>', "Rooms list having more than 5 places:", '</br></br>';
echo $racine; // calls __toString() method.
$myCompositeIterator = new CompositeIterator_A(new SimpleIterator_A($racine));
while ($myCompositeIterator->hasNext()) {
    $myNode = $myCompositeIterator->next();
    if ($myNode instanceof Room_A) {
        if ($myNode->getNbrPlaces() >= 5) {
            echo $myNode->__toString(), '</br>';
        }
    }
} // end while
```

Finally we still have the tree scanning with propagation of a method call, as seen in Alternative B. The code remains the same:

```php
//*************************************************************
// Rooms list by method call propagation:
//*************************************************************
echo '</br>', "Tree scan by method call propagation:", '</br>';
$racine->print_propagation();
```

17 Additional Notions

17.1 Loose Coupling

Refers to a dependency relationship between two or more classes, when this dependency relationship is reduced to a minimum.

We systematically search to define loose coupling between classes and design patterns are all in this direction. Designs obtained are more likely to face future developments.

17.2 The open / closed principle

This principle means that a class is closed to any changes inside the class, but open to changes by extending the class.
Prohibiting the change inside a class, limit the risk of introducing new errors.

But allowing extend the functionality of a class, we preserve the design scalability.
Design patterns respect this principle, especially the Decorator pattern, on which it is based exclusively.

17.3 Recursion

In computer science, a recursive process is a process that contains a call to itself.
The stop recursive processing criterion is fundamental. If the stop condition does not occur, the recursive calls will be performed indefinitely, which can lead to a complete blockage of the computer, due to the occupation of the entire main memory.

17.3.1 The recursive function.

This is a simple function that contains a call to itself. The classic example is the function to calculate the factorial of x:

```php
function factorielle($x)
{
    if($x == 0)
        return 1;
    else
        return $x*factorielle($x-1);
}
```

This type of treatment is based on a stack mechanism to be handled automatically by the language, and at each new call of "factorial" function will stack the execution context of the function. The context here boils down to the value of x.

When the stopping criterion occurs, the language will pop the contexts and complete execution of each, ie multiply its own value of x, with the value that was returned by the previous context. Unstacking the contexts, the values of x will multiply together to provide the final factor of the x.

17.3.2 Other kind of recursion:

Recursion by propagating a message in a tree structure.

A method is called on the root element of the tree. This method calls his counterpart on his children around, and so on.
There is no stopping criterion, processing stops when all nodes of the tree were covered.
These are the links between tree nodes, causing the recursive behavior.

Recursion with recursive instantiations.

In Object Oriented Programming: the recursive call is made on the same method name, but on an object that is instantiated when the recursive call occurs. Recursive instantiations stop when the stopping criterion occurs. The stack here is represented by the instanced objects.

18 Annexes

18.1 Naming Conventions

The following naming conventions were used in this tutorial:

Classes: *UpperCamelCase*

Iinterfaces : *UpperCamelCase*
Functions et methods: *camelCase*
Variables: *camelCase*
Constantes: *ALL_CAPS*

18.2 PHP source code

All PHP source code is provided in a separate archive, available on www.smaltek.fr.

18.3 Tools used

I used the following open source tools:

WAMP: *www.wampserver.com*

Wamp installs the following:

- An Apache server with PHP module.
- A Mysql database server. This MySQL server is not used in this tutorial.

NETBEANS: *http://netbeans.org/*

Netbeans is an IDE (Integrated Development Environment). It allows you to manage projects as sources codes for different languages including PHP.

Eclipse (*www.eclipse.org*) is another comparable IDE, also open source.

Configuration of the project in NetBeans:

- I grouped all design patterns in a single Netbeans project.
- Each design pattern is in a subdirectory of his name.
- In each subdirectory I put the following two files:
  - autoloader.php
  - includePaths.php

Autoloader.php content is the same in all design patterns:

```php
<?php

function __autoload($class) {
    require_once $class . '.class.php';
}

?>
```

PHP __ autoload() function will automatically load classes.
Here, all the classes and interfaces must be suffixed with ". Class.php" for this to work.

The includePaths.php content:

```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';

?>
```

This script will add in operating system research paths, the directories where are the classes and interfaces of the design pattern.
The paths in this script have to be changed for each design pattern.

19 Conclusion

This tutorial shows only a part of the 23 original design patterns from GOF (Gang Of Four)
http://fr.wikipedia.org/wiki/Patron_de_conception

Moreover, this list is not fixed. New patterns emerge. The latter often being merely variations of existing patterns.

A pattern can hide another.

A pattern is not monolithic. Some patterns combine to form a new pattern. We have seen for example that Composite used an Iterator.
Patterns associated in this way are called Compound Patterns.
http://serviceorientation.com/soaglossary/compound_design_pattern

Pattern at all costs.

Some pitfalls to avoid:

- Place some design pattern when it is not justified, can only add unnecessary weight to the project. This is equivalent to take a plane to go to the local bakery.
- Transform the problem to make it compatible with a pattern: It is not the specification that fits the design patterns but the reverse.
- Prepare a project to all conceivable future developments: It is a utopia that can add significantly weight to the project in coding and tests. Rather should be defined with the customer the "reasonable probability" of such future developments and prepare the project accordingly, possibly using design patterns.

End of document