## *El Patrón factory (fabricación)*

Muchos de los patrones de diseño en el libro original de Patrones de Diseño fomentan el acoplamiento débil (desconectado). Para entender este concepto, es más fácil hablar de una lucha por la que pasan muchos desarrolladores de sistemas de gran tamaño. El problema se produce cuando se cambia una sola pieza de código y ven como una cascada de roturas que ocurren en otras partes del sistema - las piezas que pensaba que eran completamente ajenas.

Este problema es el acoplamiento fuerte (estricto). Funciones y clases en una parte del sistema dependen demasiado de los comportamientos y estructuras en otras funciones y clases en otras partes del sistema. Se necesita un conjunto de patrones que permita que estas clases puedan hablar entre sí, pero no quieran unirlas tan fuertemente que se vuelvan entremezcladas.

En sistemas grandes, gran parte del código se basa en unas pocas clases clave. Pueden surgir dificultades cuando se necesita cambiar esas clases. Por ejemplo, suponga que tiene una clase de usuario que se lee de un archivo. ¿Quiere cambiar a una clase distinta que se lee de una base de datos, pero todo el código referencia a la clase original que se lee del archivo. Aquí es donde el patrón de fábrica viene muy bien.

El patrón fábrica es una clase que tiene algunos métodos que crean objetos para usted. En lugar de utilizar directamente el método nuevo, se utiliza la clase fábrica para crear objetos. De esta forma, si desea cambiar los tipos de objetos creados, puede cambiar sólo la fábrica. Todo el código que utiliza fábrica cambia de de forma automática.

Listado 1 muestra un ejemplo de una clase fábrica. El lado del servidor de la ecuación viene en dos piezas: la base de datos, y un conjunto de páginas PHP que permiten añadir feeds, solicitar la lista de feeds, y recibir el artículo asociado a un feed en particular.

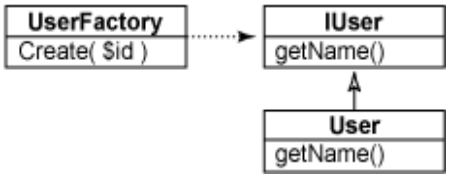
**Listado 1. Factory1.php**

```
<?php
interface IUser
{
  function getName();
}

class User implements IUser
{
  public function __construct( $id ) { }

  public function getName()
  {
    return "Jack";
  }
}

class UserFactory
```

```
{
  public static function Create( $id )
  {
    return new User( $id );
  }
}

$uo = UserFactory::Create( 1 );
echo( $uo->getName()."\n" );
?>
```

Una interfaz llamada IUSER define lo que un objeto User debe hacer. La implementación de IUSER se llama User, y una clase fábrica UserFactory crea objetos IUSER. Esta relación se muestra como UML en la Figura 1.

**Figura 1. La clase factory y sus actividades relacionadas con la interfaz IUser y la clase User**




Si ejecuta este código en la línea de comandos mediante el intérprete de PHP, se obtiene este resultado:

```
% php factory1.php
Jack
%
```

El código de prueba pide a fábrica por un objeto User e imprime el resultado del método getName.

Una variación del patrón fábrica utiliza métodos de fábrica. Estos métodos estáticos públicos en la clase construyen objetos de ese tipo. Este enfoque es útil cuando la creación de un objeto de este tipo no es trivial. Por ejemplo, supongamos que usted necesita primero crear el objeto y luego establecer muchos atributos. Esta versión del patrón fábrica encapsula ese proceso en un solo lugar para que el código de inicialización complejo no sea copiado y pegado en todo el código base.

El Listado 2 muestra un ejemplo del uso de métodos de fábrica.

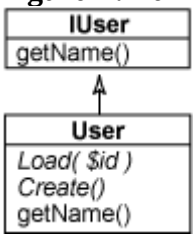
**Listado 2. Factory2.php**

```
<?php
interface IUser
{
  function getName();
}
```

```
class User implements IUser
{
  public static function Load( $id )
  {
        return new User( $id );
  }

  public static function Create( )
  {
        return new User( null );
  }

  public function __construct( $id ) { }

  public function getName()
  {
    return "Jack";
  }
}

$uo = User::Load( 1 );
echo( $uo->getName()."\n" );
?>
```

Este código es mucho más simple. Tiene una única interfaz, `IUser`, y una clase llamada `User` que implementa la interfaz. La clase de usuario tiene dos métodos estáticos que crear el objeto. Esta relación se muestra en UML en la Figura 2.

**Figura 2. La interfaz `IUser` y la clase `User` con los métodos de la fábrica**




La ejecución del script en la línea de comandos produce el mismo resultado que el código en el Listado 1, como se muestra aquí:

```
% php factory2.php
Jack
%
```

Como se ha visto, a veces aplicar tales patrones puede parecer excesivo en situaciones pequeñas. Sin embargo, sigue siendo bueno aprender formas sólidas de codificación como estas para su uso en cualquier tamaño de proyecto.

## *El Patrón singleton*

Algunos recursos de la aplicación son exclusivos en que hay uno y sólo uno de este tipo de recursos. Por ejemplo, la conexión a una base de datos a través del handle (manija) de la base de datos es exclusiva. Quiere compartir el handle de la base de datos en una aplicación, para evitar que la apertura y cierre de las conexiones cause sobrecarga, sobre todo durante obtención de información en una sola página.

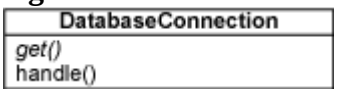
El patrón singleton cubre esta necesidad. Un objeto es un singleton si la aplicación puede incluir uno y sólo uno de ese objeto a la vez. El código del listado 3 muestra una conexión de base de datos singleton en PHP V5

**Listado 3. Singleton.php**

```
<?php
require_once("DB.php");

class DatabaseConnection
{
  public static function get()
  {
    static $db = null;
    if ( $db == null )
      $db = new DatabaseConnection();
    return $db;
  }

  private $_handle = null;

  private function __construct()
  {
    $dsn = 'mysql://root:password@localhost/photos';
    $this->_handle =& DB::Connect( $dsn, array() );
  }

  public function handle()
  {
    return $this->_handle;
  }
}

print( "Handle = ".DatabaseConnection::get()->handle()."\n" );
print( "Handle = ".DatabaseConnection::get()->handle()."\n" );
?>
```

Este código muestra una clase llamada DatabaseConnection. No puede crear sus propios DatabaseConnection porque el constructor es privado. Pero puede conseguir uno y sólo un objeto DatabaseConnection con el método estático get. El modelo de este código se muestra en la Figura 3.

**Figura 3. La conexión a base de datos singleton**




La comprobación es que el handle de la base de datos devuelto por el método es el mismo entre las dos llamadas. Se puede ver esto ejecutando el código en la línea de comandos.

```
% php singleton.php
Handle = Object id #3
Handle = Object id #3
%
```

Los dos handles devueltos son el mismo objeto. Si utiliza la conexión a base de datos singleton en la aplicación, puede reusar el mismo handle (identificador) en todas partes.

Se podría utilizar una variable global para almacenar el handle a la base de datos, pero ese enfoque sólo funciona para pequeñas aplicaciones. En aplicaciones más grandes, hay que evitar las globales, e ir con objetos y métodos para obtener acceso a los recursos.

## *El Patrón observer (observador)*

El patrón Observer (observador) brinda otra manera de evitar el acoplamiento fuerte (conexión estrecha) entre los componentes. Este modelo es simple: Un objeto se hace observable mediante la adición de un método que permite a otro objeto, el observador, registrarse a si mismo. Cuando cambia el objeto observable, envía un mensaje a los observadores registrados. Lo que los observadores hacen con esa información no es relevante o importante para el objeto observable. El resultado permite que los objetos puedan hablar unos con otros sin necesidad de entender por qué.

Un ejemplo sencillo es una lista de usuarios en un sistema. El código del listado 4 muestra una lista de usuarios que envía un mensaje cuando los usuarios se agregan. Esta lista es vista por un observador de registro que emite un mensaje cuando un usuario se agrega.

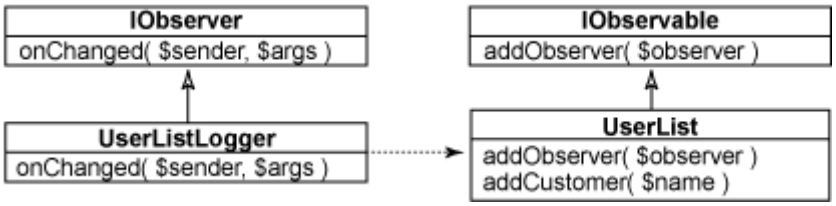
**Listado 4. Observer.php**

```
<?php
interface IObserver
{
  function onChanged( $sender, $args );
}

interface IObservable
{
  function addObserver( $observer );
}
```

```
class UserList implements IObservable
{
  private $_observers = array();

  public function addCustomer( $name )
  {
    foreach( $this->_observers as $obs )
      $obs->onChanged( $this, $name );
  }

  public function addObserver( $observer )
  {
    $this->_observers []= $observer;
  }
}

class UserListLogger implements IObserver
{
  public function onChanged( $sender, $args )
  {
    echo( "'$args' added to user list\n" );
  }
}

$ul = new UserList();
$ul->addObserver( new UserListLogger() );
$ul->addCustomer( "Jack" );
?>
```

Este código define cuatro elementos: dos interfaces y dos clases. La interfaz IObservable define un objeto que puede ser observado, y `UserList` implementa la interfaz para registrarse como observable. La lista IObserver define lo que se necesita para ser un observador, y la UserListLogger implementa esa interfaz IObserver. Esto se muestra en la Figura 4.

**Figura 4. La lista de usuarios observables y el registrador de la lista de usuarios de eventos**




Si ejecuta esto en la línea de comandos, verá la siguiente salida:

```
% php observer.php
'Jack' added to user list
%
```

El código de prueba crea un UserList y agrega el observador UserListLogger a ella. A continuación, el código añade un cliente, y el UserListLogger es notificado del cambio.

Es muy importante darse cuenta de que `UserList` no sabe lo que el registrador va a hacer. Podría haber uno o más oyentes que hacen otras cosas. Por ejemplo, puede tener un observador que envía un mensaje al nuevo usuario, dándole la bienvenida al sistema. El valor de este enfoque es que `UserList` ignora todos los objetos que dependen de ella, se enfoca en su tarea de mantener la lista de usuarios y el envío de mensajes cuando cambia la lista.

Este patrón no se limita a los objetos en la memoria. Es el fundamento de los sistemas de base de datos con colas de mensajes (database-driven message queuing systems) utilizados en aplicaciones de gran tamaño.

## *El Patrón chain-of-command (cadena de mando)*

Basándose en el principio de acoplamiento débil, el patrón *chain-of-command* (cadena de mando) dirige un comando de mensaje, solicitud, o cualquier otra cosa a través de un conjunto de manejadores (handlers). Cada controlador decide por sí mismo si se puede atender la solicitud. Si se puede, se trata la solicitud y se detiene el proceso. Se pueden agregar o quitar handlers del sistema, sin influir en otros handlers. El listado 5 muestra un ejemplo de este patrón.

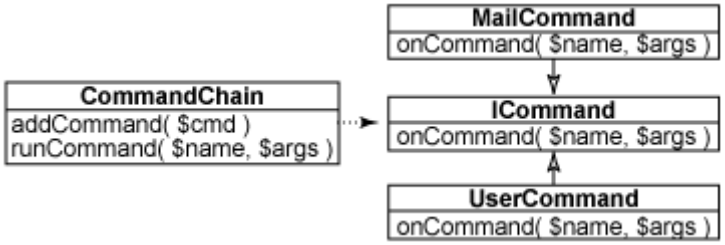
**Listing 5. Chain.php**

```
<?php
interface ICommand
{
  function onCommand( $name, $args );
}

class CommandChain
{
  private $_commands = array();

  public function addCommand( $cmd )
  {
    $this->_commands []= $cmd;
  }

  public function runCommand( $name, $args )
  {
    foreach( $this->_commands as $cmd )
    {
      if ( $cmd->onCommand( $name, $args ) )
        return;
    }
  }
}

class UserCommand implements ICommand
{
  public function onCommand( $name, $args )
  {
    if ( $name != 'addUser' ) return false;
    echo( "UserCommand handling 'addUser'\n" );
    return true;
```

```
  }
}

class MailCommand implements ICommand
{
  public function onCommand( $name, $args )
  {
    if ( $name != 'mail' ) return false;
    echo( "MailCommand handling 'mail'\n" );
    return true;
  }
}

$cc = new CommandChain();
$cc->addCommand( new UserCommand() );
$cc->addCommand( new MailCommand() );
$cc->runCommand( 'addUser', null );
$cc->runCommand( 'mail', null );
?>
```

Este código define una clase CommandChain que mantiene una lista de objetos ICommand. Dos clases implementan la interfaz ICommand – una que responde a las solicitudes de correo electrónico y otra que responde a añadir usuarios. Se muestra en la Figura 5.

**Figura 5. La cadena de mando y sus comandos relacionados**




Si se ejecuta el script, que contiene un código de prueba, verá la siguiente salida:

```
% php chain.php
UserCommand handling 'addUser'
MailCommand handling 'mail'
%
```

El código crea primero un objeto CommandChain y añade instancias de dos objetos de comando para la misma. A continuación, ejecuta dos comandos para ver quién responde a los comandos. Si el nombre del comando se ajusta a UserCommand o MailCommand, el código sigue y no pasa nada.

El patrón chain-of-command puede ser útil en la creación de una arquitectura extensible para el procesamiento de solicitudes, que se puede aplicar a muchos problemas.

## *El Patrón strategy (estrategia)*

En este patrón, los algoritmos son extraídos de las clases complejas para que puedan ser reemplazados fácilmente. Por ejemplo, el patrón strategy (estrategia) es una opción si se desea cambiar el modo en que las páginas se ordenan en un motor de búsqueda. Piense en un motor de búsqueda en varias partes - una que itera a través de las páginas, una que clasifica cada página, y otra que ordena los resultados basados en esa clasificación. En un ejemplo complejo, todas las partes estarían en la misma clase. Utilizando el patrón de estrategia, la parte de clasificación se puede disponer en otra clase para que se pueda cambiar como se clasifican las páginas, sin interferir con el resto del código del motor de búsqueda.

Como un ejemplo más simple, el Listado 6 muestra una clase de lista de usuarios que proporciona un método para encontrar un conjunto de usuarios sobre la base de un conjunto de estrategias plug-and-play.

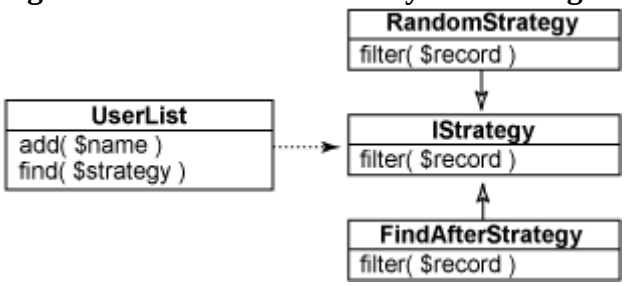
**Listado 6. Strategy.php**

```
<?php
interface IStrategy
{
  function filter( $record );
}

class FindAfterStrategy implements IStrategy
{
  private $_name;

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

  public function filter( $record )
  {
    return strcmp( $this->_name, $record ) <= 0;
  }
}

class RandomStrategy implements IStrategy
{
  public function filter( $record )
  {
    return rand( 0, 1 ) >= 0.5;
  }
}

class UserList
{
  private $_list = array();

  public function __construct( $names )
  {
    if ( $names != null )
    {
```

```
      foreach( $names as $name )
      {
        $this->_list []= $name;
      }
    }
  }

  public function add( $name )
  {
    $this->_list []= $name;
  }

  public function find( $filter )
  {
    $recs = array();
    foreach( $this->_list as $user )
    {
      if ( $filter->filter( $user ) )
        $recs []= $user;
    }
    return $recs;
  }
}

$ul = new UserList( array( "Andy", "Jack", "Lori", "Megan" ) );
$f1 = $ul->find( new FindAfterStrategy( "J" ) );
print_r( $f1 );

$f2 = $ul->find( new RandomStrategy() );
print_r( $f2 );
?>
```

El modelo UML de este código se muestra en la Figura 6.

**Figura 6. La lista de usuarios y las estrategias para la selección de los usuarios**




La clase UserList es una envoltura alrededor de un vector de nombres. Implementa un método que toma una de varias estrategias para la selección de un subconjunto de esos nombres. Esas estrategias están definidas por la interfaz IStrategy, que tiene dos implementaciones: Una elige usuarios al azar y la otra elige todos los nombres después de un nombre especificado. Cuando se ejecuta el código de prueba, se obtiene el resultado siguiente:

```
% php strategy.php
Array
(
    [0] => Jack
    [1] => Lori
    [2] => Megan
)
Array
```

```
(
    [0] => Andy
    [1] => Megan
)
%
```

El código de prueba ejecuta las mismas listas de usuarios contra dos estrategias y muestra los resultados. En el primer caso, la estrategia busca cualquier nombre que esté oredenado después de la J, para que pueda obtener Jack, Lori, y Megan. La segunda estrategia recoge los nombres al azar y con resultados diferentes cada vez. En este caso, los resultados son Andy y Megan.

El patrón strategy es ideal para sistemas de gestión de datos complejos o de procesamiento de datos que necesitan una gran flexibilidad en cómo los datos son filtrados, buscados, o procesados.

## *El Patrón adapter (adaptador)*

Utilice el patrón adapter (adaptador) cuando necesite convertir un objeto de un tipo a un objeto de otro tipo. Normalmente, los desarrolladores manejan este proceso a través de un montón de código de asignación, como se muestra en el Listado 1. El patrón adapter es una buena manera de limpiar este tipo de código y reutilizar todo el código de asignación en otros lugares. Además, se esconde el código de asignación, lo que puede simplificar las cosas un poco si también está haciendo algún formateo en el camino.

**Listado 1. Uso de código para asignar valores entre los objetos**

```
class AddressDisplay
{
    private $addressType;
    private $addressText;

    public function setAddressType($addressType)
    {
        $this->addressType = $addressType;
    }

    public function getAddressType()
    {
        return $this->addressType;
    }

    public function setAddressText($addressText)
    {
        $this->addressText = $addressText;
    }

    public function getAddressText()
    {
        return $this->addressText;
    }
}
```

```
class EmailAddress
{
    private $emailAddress;

    public function getEmailAddress()
    {
        return $this->emailAddress;
    }

    public function setEmailAddress($address)
    {
        $this->emailAddress = $address;
    }
}

$emailAddress = new EmailAddress();
/* Populate the EmailAddress object */
$address = new AddressDisplay();
/* Aquí está el código de asignación, donde estoy asignando valores
de un objeto a otro... */
$address->setAddressType("email");
$address->setAddressText($emailAddress->getEmailAddress());
```

Este ejemplo utiliza un objeto AddressDisplay para mostrar una dirección a un usuario. El objeto AddressDisplay tiene dos partes: el tipo de dirección y una cadena de dirección con formato.

Después de implementar el patrón (ver Listado 2), el script PHP no tiene que preocuparse sobre la manera exacta en que el objeto EmailAddress se convierte en el objeto AddressDisplay. Eso es bueno, especialmente si cambia el objeto AddressDisplay o las reglas que rigen la forma en que un objeto EmailAddress se convierte en un objeto AddressDisplay.

Recuerde, una de las principales ventajas del diseño modular es tener que cambiar tan poco código como sea posible si cambia algo en el dominio del negocio o es necesario agregar una nueva funcionalidad al software. Piense en esto, incluso cuando usted está haciendo las tareas cotidianas, tales como la asignación de valores de las propiedades de un objeto a otro.

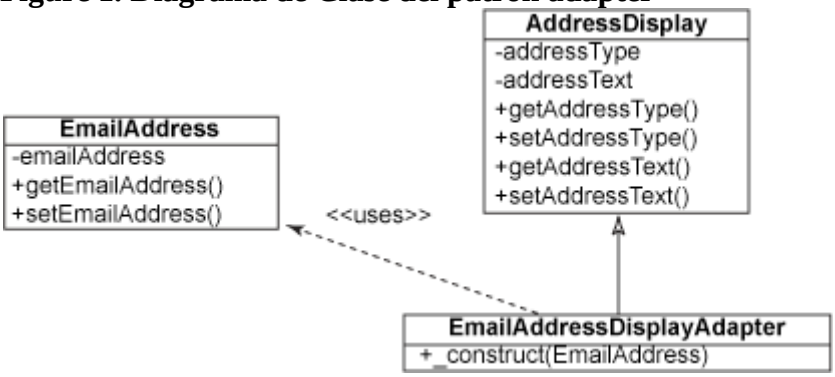
**Listado 2. Usando el patrón adapter**

```
class EmailAddressDisplayAdapter extends AddressDisplay
{
    public function __construct($emailAddr)
    {
        $this->setAddressType("email");
        $this->setAddressText($emailAddr->getEmailAddress());
    }
}

$email = new EmailAddress();
$email->setEmailAddress("user@example.com");

$address = new EmailAddressDisplayAdapter($email);
```

```
echo($address->getAddressType() . "\n") ;
echo($address->getAddressText());
```

Figure 1 shows a class diagram of the adapter pattern.

**Figure 1. Diagrama de Clase del patrón adapter**

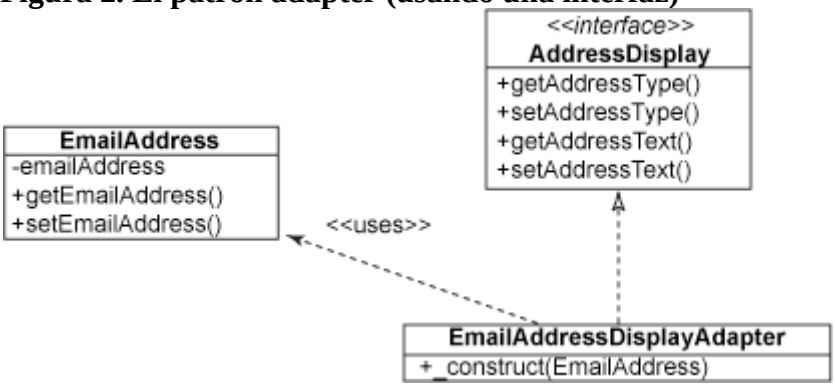



**Metodo Alternativo**

Un método alternativo de escribir un adapter es implementar una interfaz para adaptar el comportamiento, en lugar de extender un objeto.

Esta es una manera muy limpia para la creación de un adapter y no tiene los inconvenientes de extender el objeto. Una de las desventajas del uso de la interfaz es que es necesario agregar la implementación en la clase de adaptador, como se muestra en la Figura 2.

**Figura 2. El patrón adapter (usando una interfaz)**




## *El Patrón iterator (iterador)*

El patrón iterator (iterador) proporciona una forma de encapsular bucles a través de una colección o un vector de objetos. Es especialmente útil si desea recorrer los diferentes tipos de objetos de la colección.

Vuelva a mirar el ejemplo en el Listado 1: correo electrónico y dirección física. Antes de añadir un patrón iterator, si estamos recorriendo las direcciones de la persona, es posible recorrer las direcciones físicas y mostrarlas, y a continuación, recorrer las direcciones de correo electrónico de la persona y mostrarlas, entonces recorrer un bucle a través de direcciones de mensajería instantánea de la persona y mostrarlas . Eso es un bucle sucio!

En su lugar, mediante la implementación de un iterador, todo lo que hay que hacer es llamar while($ ITR-> hasNext ()) y tratar con lo que retorna el siguiente item $ ITR-> next ().

Un ejemplo de uno de los iterator se muestra en el Listado 3.

Esto permite añadir nuevos tipos de elementos sobre los que se pueda iterar, sin tener que cambiar el código que recorre los elementos. En el ejemplo de persona, por ejemplo, se podría agregar un vector de direcciones de mensajería instantánea, simplemente actualizando el iterador, sin tener que modificar el código que recorre las direcciones a mostrar.

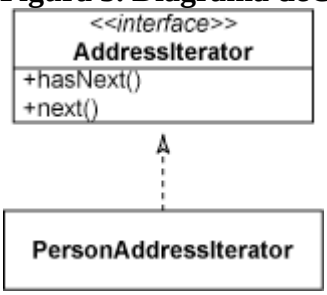
**Listado 3. Usando el patrón iterator para recorrer objectos**

```
class PersonAddressIterator implements AddressIterator
{
    private $emailAddresses;
    private $physicalAddresses;
    private $position;

    public function __construct($emailAddresses)
    {
        $this->emailAddresses = $emailAddresses;
        $this->position = 0;
    }

    public function hasNext()
    {
        if ($this->position >= count($this->emailAddresses) ||
            $this->emailAddresses[$this->position] == null) {
            return false;
        } else {
            return true;
        }
    }

    public function next()
    {
        $item = $this->emailAddresses[$this->position];
        $this->position = $this->position + 1;
        return $item;
    }

}
```

Si el objeto Person se modifica para devolver una implementación de la interfaz AddressIterator, el código de la aplicación que utiliza el iterador no necesita ser modificado si la aplicación se extiende para recorrer objetos adicionales. Se

puede utilizar un iterador compuesto que envuelve a los iteradores que recorren cada tipo de dirección

**Figura 3. Diagrama deClases del patrón iterator**




## *El Patrón decorator (decorador)*

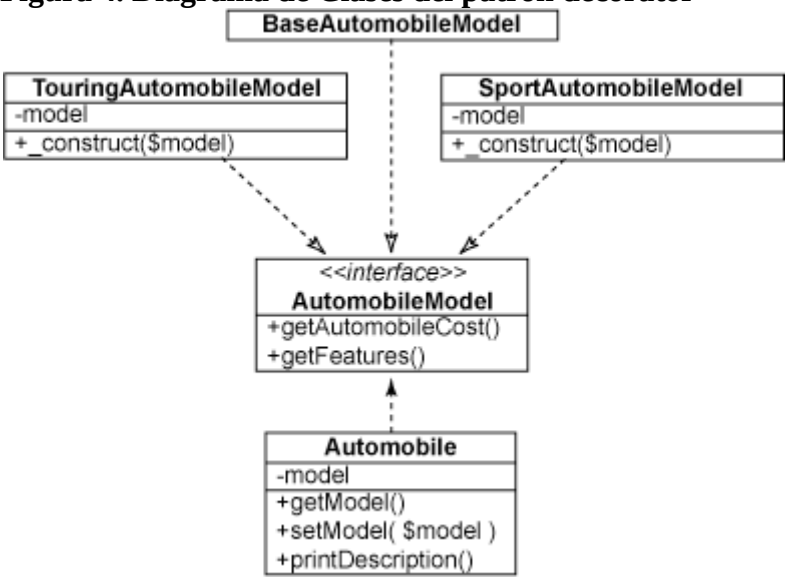
Considere el ejemplo de código en el Listado 4. El propósito de este código consiste en añadir un montón de características en un coche por un sitio *Construya su propio coche*. Cada modelo de coche tiene más características y un costo asociado. Con sólo dos modelos, sería bastante trivial agregar estas características con declaraciones `if then`. Sin embargo, si se incorpora un nuevo modelo, tendría que volver a pasar por el código y asegurarse que las declaraciones trabajan para el nuevo modelo.

**Listing 4. Using the decorator pattern to add features**

```
require('classes.php');

$auto = new Automobile();

$model = new BaseAutomobileModel();

$model = new SportAutomobileModel($model);

$model = new TouringAutomobileModel($model);

$auto->setModel($model);

$auto->printDescription();
```

Introduzca el patrón decorator (decorador), que permite agregar esta funcionalidad en AutomobileModel en una clase agradable y limpia. Cada clase mantiene su incumbencia en su precio y características y cómo son agregadas al modelo base.

**Figura 4. Diagrama de Clases del patrón decorator**




Una de las ventajas del patrón decorador es que se puede añadir fácilmente más de un decorador a la base.

Si se ha hecho mucho trabajo con objetos de flujo (stream objects), ha utilizado un decorador. La mayoría de las construcciones de flujo, como un flujo de salida, son decoradores que toman un flujo de entrada de base, y luego lo decoran añadiendo funcionalidad adicional - como uno que ingresa flujos de archivos, uno que ingresa flujos de buffers, etc.

## *El Patrón delegate (delegado)*

El patrón delegate (delegado) proporciona una forma de delegar el comportamiento basado en diferentes criterios. Considere el código en el Listado 5. Este código contiene una serie de condiciones. Sobre la base de la condición, el código selecciona el tipo adecuado de objeto para manejar la petición.

**Listado 5. Usando declaraciones condicionales para redirigir requerimientos de embarque**

```
pkg = new Package("Heavy Package");
$pkg->setWeight(100);

if ($pkg->getWeight() > 99)
{
        echo( "Shipping " . $pkg->getDescription() . " by rail.");
} else {
        echo("Shipping " . $pkg->getDescription() . " by truck");
}
```

Con un patrón delegate, un objeto internaliza este proceso de enrutamiento mediante el establecimiento de una referencia interna al objeto apropiado cuando un método es llamado, como useRail () en el Listado 6. Esto es especialmente útil si los criterios cambian para el manejo de varios paquetes, o si un nuevo tipo de transporte está disponible.

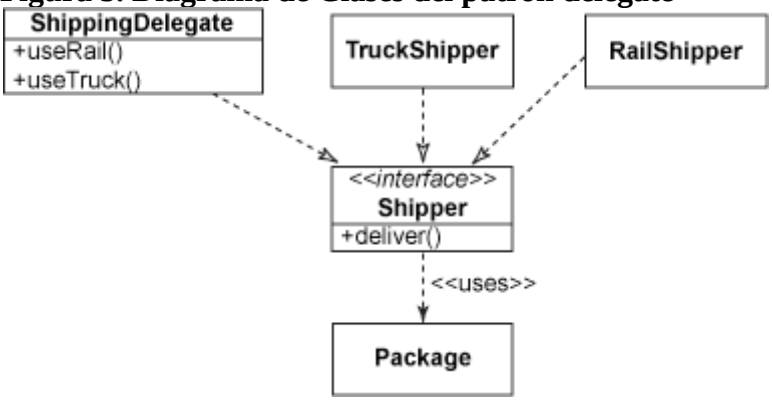
**Listado 6. Usando el patrón delegate para reasignar pedidos de embarque**

```
require_once('classes.php');

$pkg = new Package("Heavy Package");
$pkg->setWeight(100);

$shipper = new ShippingDelegate();

if ($pkg->getWeight() > 99)
{
        $shipper->useRail();
}

$shipper->deliver($pkg);
```

El delegado tiene la ventaja de que el comportamiento puede cambiar dinámicamente llamando a los métodos useRail () o useTruck () para cambiar la clase que se ocupa del trabajo

Figua 5 muestra un diagrama de clases del patrón delegate.

**Figura 5. Diagrama de Clases del patrón delegate**




## *El Patrón state (estado)*

El patrón state (Estado) es un patrón similar a los patrones command, pero la intención es muy diferente. Considere el código de abajo

**Listado 7. Usande codigo para construir un robot**

```
class Robot
{
```

```
        private $state;

        public function powerUp()
        {
                if (strcmp($state, "poweredUp") == 0)
                {
                        echo("Already powered up...\n");
                        /* Implementation... */
                } else if ( strcmp($state, "powereddown") == 0) {
                        echo("Powering up now...\n");
                        /* Implementation... */
                }
        }

        public function powerDown()
        {
                if (strcmp($state, "poweredUp") == 0)
                {
                        echo("Powering down now...\n");
                        /* Implementation... */
                } else if ( strcmp($state, "powereddown") == 0) {
                        echo("Already powered down...\n");
                        /* Implementation... */
                }
        }

        /* etc... */

}
```

En este listado, el código PHP representa el sistema operativo de un poderoso robot que se convierte en un coche. El robot puede encender, apagar, convertirse a su vez en un robot, cuando es un vehículo, y se convierten en un vehículo cuando se trata de un robot. El código está bien ahora, pero se ve que puede llegar a ser complejo si alguna de las reglas cambia, o si otro estado entra en escena.

Ahora mire el listado 8, que tiene la misma lógica para el manejo de los estados del robot, pero esta vez ponga a la lógica en el patrón de estado. El código en el Listado 8 hace lo mismo que el código original, pero la lógica del manejo de los estados se ha puesto en un objeto para cada estado. Para ilustrar las ventajas de utilizar este patrón de diseño, imagine que después de un tiempo, estos robots han descubierto que no se deben apagar mientras se está en modo de robot. De hecho, si se apagan, deben cambiar primero al modo de vehículo. Si ya está en el modo de vehículo, el robot sólo se apaga. Con el patrón state, los cambios son bastante triviales.

**Listado 8. Usando el patrón state para manejar los estados del robot**

```
$robot = new Robot();
echo("\n");
$robot->powerUp();
echo("\n");
$robot->turnIntoRobot();
echo("\n");
$robot->turnIntoRobot(); /* This one will just give me a message */
```

```
echo("\n");
$robot->turnIntoVehicle();
echo("\n");
```

**Listado 9. Pequeños cambios a uno de los objetos state**

```
class NormalRobotState implements RobotState
{
    private $robot;

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

    public function powerUp()
    {
        /* implementation... */
    }
    public function powerDown()
    {
        /* First, turn into a vehicle */
        $this->robot->setState(new VehicleRobotState($this->robot));
        $this->robot->powerDown();
    }

    public function turnIntoVehicle()
    {
        /* implementation... */
    }

    public function turnIntoRobot()
    {
        /* implementation... */
    }
}
```

Algo que no parece obvio si observamos la Figura 6 es que cada objeto en el patrón state tiene una referencia al objeto de contexto (el robot), por lo que cada objeto puede alcanzar el estado apropiado.

**Figura 6. Diagrama de Clases del patrón state**