Implementación

Capítulo 9
Implementación

9.1. Entorno de programación

La definición del entorno tecnológico del nuevo sistema influye de manera decisiva en el desarrollo del proyecto, por ello esta tarea se realiza en paralelo a otras actividades y/o fases del proyecto. En esta tarea será necesaria la participación activa del director del proyecto, para ayudar a definir el entorno óptimo para el nuevo sistema.

La especificación del entorno tecnológico del nuevo sistema se describe a continuación en la siguiente tabla:

Elemento	**Tipo**	**Detalle**	**Versión**
Hardware	Ordenador	Intel Pentium 4 3GHz - 512MB - 60Gb	
Software	Sistema Operativo	Windows XP	2002 SP1
	Servidor Web	Apache	1.3.33
	Gestión Web BD	PHPMyAdmin	2.6.1
	Entorno de Desarrollo	Eclipse	3.0
	Navegadores Web	Mozilla Firefox	1.0.4
		Internet Explorer	6.0
		Opera	8.0
	Gestor Base de Datos	MySQL	4.1.9
	Leng. de Programación	PHP	4.3.10
	Editor de Textos	TeXnicCenter	6.01 (beta)

Cuadro 9.1: Especificación del Entorno Tecnológico del Sistema

Como conclusión podemos indicar que para la instalación del servidor Apache, PHP y MySQL existen una gran cantidad de instaladores. De entre los existentes, hemos empleado el paquete EasyPHP 1.8, que nos instala fácilmente dichos elementos. Por último, para la representación de gráficas necesitamos la librería de PHP *php_gd2*.

Elección del lenguaje de programación

La elección del lenguaje PHP (siglas que originalmente signifcaban *Personal Home Page*) se basa fundamentalmente en las siguiente cinco grandes características:

Velocidad No solo la velocidad de ejecución, la cual es importante, sino además no crear demoras en la máquina. Por esta razón no debe requerir demasiados recursos de sistema. PHP se integra muy bien junto a otro software, especialmente bajo ambientes Unix, cuando se configura como módulo de Apache, está listo para ser utilizado.

Estabilidad PHP utiliza su propio sistema de administración de recursos y dispone de un sofisticado método de manejo de variables, conformando un sistema robusto y estable.

Seguridad El sistema debe poseer protecciones contra ataques. PHP provee diferentes niveles de seguridad, estos pueden ser configurados desde el archivo *php.ini*

Simplicidad Se les debe permitir a los programadores generar código productivamente en el menor tiempo posible.

Conectividad PHP dispone de una amplia gama de librerías, y agregarle extensiones es muy fácil. Está compuesto de un sistema principal (escrito por Zend), un conjunto de módulos y una variedad de extensiones de código.

Otras ventajas son:

- funciona en (casi) cualquier plataforma utilizando el mismo código fuente, pudiendo ser compilado y ejecutado en aproximadamente 25 plataformas, incluyendo diferentes versiones de Unix, Windows y Macs.
- posee muchas interfaces distintas para cada tipo de servidor. PHP actualmente se puede ejecutar bajo Apache, IIS, AOLServer, Roxen y THTTPD. Otra alternativa es configurarlo como modulo CGI.
- puede interactuar con muchos motores de bases de datos tales como MySQL, MS SQL, Oracle, Informix, PostgreSQL, y otros muchos.
- PHP es *Open Source*, lo cual significa que el usuario no depende de una compañía específica, además no se está forzado a pagar actualizaciones anuales para tener una versión que funcione.

Pero también posee inconvenientes:

- El manejo de errores no es tan sofisticado como Cold Fusion o ASP.
- No existe IDE o Debugger.

9.2. Aspectos relevantes de la implementación

Como resumen del proceso de implementación de la aplicación, expondremos los aspectos más importantes y destacados que hemos tenido que resolver a la hora de crear este Portal Web.

Inicialmente describiremos brevemente cómo se han codificado las diferentes capas de nuestro diseño, para luego centrarnos en los problemas comunes a cualquier aplicación web, dando nuestra solución particular

9.2.1. Implementación de la Capa de Presentación

Como hemos visto en los diagramas de diseño, la capa de presentación posee una clase superior denominada VistaGeneral que define los aspectos estáticos o fijos de la aplicación, como son

Cabecera definida mediante el método *VistaGeneral::crearCabecera()*

Menú Principal (que varía según los permisos del usuario actual), , definida mediante el método *VistaGeneral::crearMenuPrincipal()*

Menú Secundario definida mediante el método *VistaGeneral::crearMenuSecundario()*

Pié de Página definida mediante el método *VistaGeneral::crearPiePagina()*

También define una serie de métodos para ayudar a las subclases que heredan de ella para crear los títulos de los formularios (*VistaGeneral::crearTitulo()*) y la navegabilidad de los mismos (*VistaGeneral::crearNavegacion()*), basándose este ultimo en un array cuyo cada elemento es un item en la navegabilidad, y en el método final que genera el contenido de la aplicación basándose en las plantillas definidas con la clase *FastTemplate*. Para mantener un orden a la hora de generación, tenemos el siguiente orden de llamadas:

```
// VistaGeneral::mostrar(), muestra el esquema de generación de la vista de la aplicación

function mostrar(){
  $this->crearCabecera();             // definida en VistaGeneral
  $this->crearMenuPrincipal();        // definida en VistaGeneral
  $this->crearContenido();            // redefinida por cada subclase
  $this->crearMenuSecundario();       // definida en VistaGeneral
  $this->crearPiePagina();            // definida en VistaGeneral
  $this->imprimir();                  // definida en VistaGeneral
}
```

La importancia de esta clase reside en que sus subclases heredan dichos métodos, de tal forma que éstas adaptan el método *crearContenido()* a sus necesidades, puesto que es en esencia lo que cambia en cada parte de la aplicación.

Debido a que cada subvista tiene que generar básicamente formularios de alta, baja, modificación y consulta, se ha tenido que añadir tanto crearContenido como hayan sido necesarios, de tal forma que disponemos de un adaptador de cada vista al contenido que nos interesa, veamos ejemplos:

```
require_once( getenv('CLASES_PRESENTACION').'vistaGeneral.class.php');
require_once( getenv('CLASES_SISTEMA').'usuario.class.php');
require_once( getenv('CLASES_SISTEMA').'perfil.class.php');

class VistaGestUsuarios extends VistaGeneral{

// VistaGestUsuario::mostrarFormularioRegistrarUsuario(),
// genera el formulario de alta de usuarios

function mostrarFormularioRegistrarUsuario(){
  $this->crearCabecera();
  $this->crearMenuPrincipal();
  $this->crearContenidoRegistrarUsuario();
  $this->crearMenuSecundario();
  $this->crearPiePagina();
  $this->imprimir();
}

// VistaGestUsuario::mostrarFormularioModificarPerfilUsuario(),
// genera el formulario de modificación de perfiles

function mostrarFormularioModificarPerfilUsuario($login,$perfil){
  $this->crearCabecera();
  $this->crearMenuPrincipal();
  $this->crearContenidoModificarPerfilUsuario($login,$perfil);
  $this->crearMenuSecundario();
  $this->crearPiePagina();
  $this->imprimir();
}

...
}
```

En resumen y como norma de codificación empleada a la hora de implementar las clases de la vista, hemos realizado lo siguiente:

1. definir la clase Vista a necesitar, por ejemplo *VistaGestUsuarios*
2. incluir, si no se ha incluido anteriormente, la superclase *VistaGeneral* y las clases del sistema que vayamos a necesitar
3. heredar de la superclase *VistaGeneral* para acceder a los métodos del esquema de generación de la aplicación
4. para cada formulario a generar o información a mostrar, definimos dos métodos:
   - mostrarFormulario-*acción*-(), que define el esquema de generación, incluyendo el *crearContenido* asociado
   - crearContenido-*acción*-(), *crearContenido* asociado que define el formulario a generar

9.2.2. Implementación de la Capa de Aplicación

En la capa de aplicación hemos implementado el patrón fachada para poder tener un único punto de recepción de solicitudes del cliente, para luego filtrarla al subcontrolador asignado al mismo. El punto de entrada se realiza siempre al fichero *controlador.php* que se encarga de instanciar la clase controladora principal, *Controlador* y llamar al método general *procesar()*:

```
<?php
/* PARAMETROS QUE RECIBE
 *  $mod:  módulo al que hacer referencia
 *  $accion: acción a realizar dentro del módulo
 */
require_once('./comun/config.php');
require_once( getenv('CLASES_APLICACION').'controlador.class.php' );
$controlador =& Controlador::getInstancia();
$controlador->procesar($mod, $accion);
?>
```

La clase *Controlador* recibe dos parámetros: módulo y acción, indicando el módulo o subcontrolador a utilizar y un código de la acción a realizar, respectivamente (por ejemplo, para enviar un mensaje a un foro en cuestión, tendríamos que indicar el módulo *foros* y la acción *6a*).

Cada acción está codificada con el par (Nº de acción,paso de la acción), de tal forma que la acción *6a* del módulo *foros* indica *mostrar el formulario de registro del mensaje en el foro* y la acción *6b* indica *registrar mensaje en el sistema*. De esta forma separamos aún más la lógica de proceso, de tal forma que tenemos completamente delimitados todos los pasos para realizar las acciones dentro de cada módulo del sistema.

Tras esto, la clase *Controlador*, en función del módulo, instancia el subcontrolador indicado y llama a su método *procesar* sólamente con la acción que recibió:

```
// método Controlador::procesar(), instancia subcontroladores y les asigna acciones

function procesar($mod, $accion=""){
  switch($mod){
    case 'index':
      $vista =& VistaGeneral::getInstancia();
      $vista->mostrar(); break;
    case 'usuarios':
      require_once( getenv('CLASES_APLICACION') . 'controladorGestUsuarios.class.php');
      $subcontrolador =& ControladorGestUsuarios::getInstancia();
      $subcontrolador->procesar($accion); break;
    case 'videojuegos':
      require_once( getenv('CLASES_APLICACION') . 'controladorGestVideojuegos.class.php');
      $subcontrolador =& ControladorGestVideojuegos::getInstancia();
      $subcontrolador->procesar($accion); break;
    ...
    case 'encuestas':
      require_once( getenv('CLASES_APLICACION') . 'controladorGestEncuestas.class.php');
      $subcontrolador =& ControladorGestEncuestas::getInstancia();
      $subcontrolador->procesar($accion); break;
    default: Error::Error("No se reconoce el módulo indicado"); exit;
  }
}
```

Por último, para cada subcontrolador se define de nuevo el método *procesar()* solamente con la acción indicada, que se vuelve a filtrar según el par que define la acción:

```
// método ControladorGestForos::procesar(), define la lógica de negocio a cada acción

function procesar($accion){

  switch ($accion) {

    case '1a' : // 1a.- Mostrar formulario registro de foro
      $this->mostrarFormularioRegistrarForo(); break;
    case '1b' : // 1b.- Registrar Foro
      $this->registrarForo(); break;

    case '2a' : // 2a.- Consultar datos del foro
      $this->mostrarFormularioConsultarForo($GLOBALS['idforo']); break;

    case '3a' : // 3a.- Mostrar formulario modificar foro
      $this->mostrarFormularioModificarForo($GLOBALS['idforo']); break;
    case '3b' : // 3b.- Modificar datos del foro
      $this->modificarForo($GLOBALS['idforo']); break;

    case '4a' : // 4a.- Mostrar formulario eliminar foro
      $this->mostrarFormularioEliminarForo($GLOBALS['idforo']); break;
    case '4b' : // 4b.- Eliminar foro
      $this->eliminarForo($GLOBALS['idforo']); break;

    case '5a' : // 5a.- Mostrar formulario búsqueda de foros
      $this->mostrarFormularioBuscarForos(); break;

    case '6a' : // 6a.- Mostrar formulario registrar mensaje
      $this->mostrarFormularioRegistrarMensaje($GLOBALS['idforo'],$GLOBALS['idmsj']); break;
    case '6b' : // 6b.- Registrar Mensaje
      $this->registrarMensaje($GLOBALS['idforo'],$GLOBALS['idmsj']); break;

    case '7a' : // 7a.- Consultar mensaje
      $this->mostrarFormularioConsultarMensaje($GLOBALS['idmsj']); break;

    case '8a' : // 8a.- Mostrar formulario eliminar mensaje
      $this->mostrarFormularioEliminarMensaje($GLOBALS['idmsj']); break;
    case '8b' : // 8b.- Eliminar mensaje
      $this->eliminarMensaje($GLOBALS['idmsj']); break;

    case '9a' : // 9a.- Mostrar formulario búsqueda de mensajes
      session_start();
      $this->mostrarFormularioBuscarMensajes($_SESSION['idforo']); break;

    default:
      Error :: Error('No se reconoce la acción indicada');

  }

}
```

Y finalmente, los métodos asignados a cada acción definen la lógica de negocio, que se encargarán de acceder a las capas de datos y presentación, según la acción, además del uso de las clases del sistema, empleadas en todo el sistema.

9.2.3. Implementación de la Capa de Datos

La última capa de la aplicación es similar a la capa de presentación, posee una superclase denominada *AbstracDAO* que define los métodos más comunes a cualquier acceso a una Base de Datos, como es la ejecución de sentencias, crear conexiones, obtener campos, iterar sobre las diferentes tuplas del *resultset*, etc.

```
// método AbstractDAO::ejecutar(), ejecuta sentencias SQL en la Base de Datos

function ejecutar($sql,$mensaje=""){

  $conexion =& ConexionDAO::getInstancia();
  $link = $conexion->getLink();

  $link->StartTrans();                      // Inicio Transacción

  $this->result =& $link->Execute($sql);    // ejecución de la sentencia

  if( ! $this->result ){
    $msjError = $link->ErrorMsg();
    $link->FailTrans();                     // RollBack de la Transacción
    if( trim($mensaje) == "" )
      Error::Error("Ha ocurrido un error<br>$msjError<br>$sql");
    else
      Error::Error("$mensaje<br>$msjError<br>$sql");
  }

  $this->id = $link->Insert_ID();           // almacenamos el id para los INSERT

  $link->CompleteTrans();                   // Commit de la Transacción

  $link->Close();
}
```

Como caracterísitica fundamental podemos decir que se usan transacciones para todas las sentencias SQL y que las conexiones son *No Permanentes* de tal forma que en cada ejecución podemos comprobar si el usuario que ejecutó la acción tiene permiso para acceder a la información que solicita.

Para el caso de las conexiones, instanciamos un objeto de la clase *ConexionDAO* que se encarga de la configuración para establecer la conexión a la Base de Datos (define el usuario y el password, tipo de bd, nombre del mismo y la posibilidad de establecer un debug de las consultas ejecutadas):

```
require_once( getenv('CLASES_DATOS'). 'adodb/adodb.inc.php');

class ConexionDAO {

  // CONFIGURACION DE LA BASE DE DATOS
  var $dbtype = 'mysqlt';                   // MySQL con soporte para transacciones
  var $dbhost = 'localhost';
  var $dbuser = 'anonimo';                  // usuario por defecto
  var $dbpassword = '';
  var $dbdatabase = 'portalwebvideojuegos'; // nombre de la base de datos
  var $dbdebug = false;                     // por defecto sin opciones de debug
  var $link;                                // variable de conexión
```

```
  // método ConexioDAO::getLink(), ejecuta sentencias SQL en la Base de Datos

  function getLink() {
    $this->link =& ADONewConnection($this->dbtype);
    $this->link->debug=$this->dbdebug;
    $this->link->NConnect($this->dbhost,$this->dbuser,$this->dbpassword,$this->dbdatabase);
    if( !$this->link->IsConnected() )
      Error::Error('fallo de conexión');
    return $this->link;
  }
 ...
}
```

Con este diseño que hemos planteado hemos querido independizar el acceso a la base de datos, de tal forma que por un lado tenemos la clase que se encarga de conectar a una BD, independientemente del tipo de BD que sea[1], y por otro, hemos definido una superclase que implementa los métodos comunes de acceso a la BD, que dependerán del tipo de BD.

El resto de clases DAO heredan de *AbstractDAO* para poder emplear dichos métodos comunes, y éstas subclases definirán tantos métodos como operaciones básicas se necesiten para el que fué creado, como por ejemplo *VideojuegoDAO*:

```
require_once( getenv('CLASES_DATOS').'abstractDAO.class.php');

class VideojuegoDAO extends AbstractDAO \{

var $daoGenero;
var $daoEmpresa;
var $daoPlataforma;
var $daoUsuario;

// métodos getVideojuego(), crea un videojuego con el id dado

function getVideojuego($idvid) \{

  // obtenemos toda la información del videojuego
  $sql = "SELECT * FROM pwv_videojuegos WHERE idvideojuego=$idvid";
  $this->ejecutar($sql);

  $nombre = $this->getCampo('nombre');
  $desc = $this->getCampo('descripcion');
  $fecha = $this->getCampo('fecha');
  $imagen = $this->getCampo('imagen');
  $url = $this->getCampo('url');
  $precio = $this->getCampo('precio');
  $moneda = $this->getCampo('moneda');
  $idemp = $this->getCampo('idempresa');
  $empresa = $this->daoEmpresa->getEmpresa($idemp);
  $idgen = $this->getCampo('idgenero');
  $genero = $this->daoGenero->getGenero($idgen);

  // obtenemos las plataformas del videojuego
  $sql = "SELECT idplataforma FROM pwv_videojuegoplataformas WHERE idvideojuego=$idvid";
  $this->ejecutar($sql);
```

[1] Esto no es 100 % cierto, depende tanto del tipo de BD como de la implementación dada de AdoDB

```
  // almacenamos sus identificadores
  $idplats = array();
  while( !$this->esUltimo() ) {
    array_push($idplats,$this->getCampo('idplataforma'));
    $this->next();
  }
  $plataformas = array();
  foreach( $idplats as $idplat ) {
   // instanciamos PlataformaDAO para obtener la plataforma por el id dado
    array_push($plataformas,$this->daoPlataforma->getPlataforma($idplat));
  }

  // creamos el objeto Videojuego
  $videojuego = new Videojuego($nombre,$desc,$fecha,$imagen,$url,$precio,$moneda,$empresa,
    $genero,$plataformas);
  $videojuego->setId($idvid);

  return $videojuego;
\}
...
\}
```

9.2.4. Implementación de las Clases del Sistema

Una vez definidos las clases en la fase de análisis, se proceden a completarse para dar lugar a las clases de diseño, las cuales se definen en el paquete *Clases del Sistema*, descritos en la etapa Diseño.

Estas clases no tienen mucha complejidad, salvo en ciertas clases donde se requiere el uso de *Factorías*, las cuales se encargan de generar los objetos. Éstas factorías afectan a las clases que generen tipos, como son *TipoConcepto*, *TipoOpcion* y *TipoRecurso*, empleándose las factorías *FactoriaTipoConcepto*, *FactoriaTipoOpcion* y *FactoriaTipoRecurso*:

```
require_once( getenv('CLASES_DATOS').'tipoRecursoDAO.class.php');

class FactoriaTipoRecurso {

var $daoTipoRecurso;

function FactoriaTipoRecurso() {
  $this->daoTipoRecurso =& TipoRecursoDAO::getInstancia();
}

function getTipoRecursoPorId($id) {
  $tipo= new TipoRecurso();
  if( $this->daoTipoRecurso->existeId($id) ){
    $tipo->setId($id);
    $nombre = $this->daoTipoRecurso->getNombreRecurso($id);
    $tipo->setNombre($nombre);
  } else {
    return null;
  }
  return $recurso;
}
```

```
function getTodosTiposRecurso() {
  $array = $this->daoTipoRecurso->getTodosTiposRecurso();
  $arrayRecursos = array();
  foreach( $array as $tipoRecurso )
    foreach( $tipoRecurso as $id => $nombre ){
      $recurso =& new TipoRecurso();
      $recurso->setId($id);
      $recurso->setNombre($nombre);
      array_push($arrayRecursos,$recurso);
    }
  return $arrayRecursos;
}
}
```

Por motivos de eficiencia, hemos tenido que adoptar una serie de medidas en ciertas clases del sistema. Éstos cambios son:

1. Los usuarios tienen asignado un perfil. Debido a que los usuarios son muy consultados y no tanto su perfil, hemos optado por no crear el perfil del usuario cuando se crea el objeto, sino hasta que realmente fuera necesario, en la llamada al método *PerfilDAO::getPerfil($login)*.

   Esta estrategia se corresponde con el patrón de diseño *Proxy Virtual*, pero debido a las limitaciones de la versión 4 de PHP en cuanto a programación orientada a objetos, se hacia complejo la adaptación.

2. Los videojuegos tienen una serie de conceptos asociados como son el género del mismo, la empresa que lo desarrolló, las plataformas en las que se puede jugar y una serie de aspectos no tan ligados al videojuego en sí, como son las opiniones, las ventas de usuarios y los recursos asociados en el repositorio; es por esto que estas tres colecciones de objetos (*Opinion*, *Venta* y *Recurso*) se tratarán como objetos independientes, aunque relacionados en el modelo del dominio con los videojuegos.

   La decisión que hemos tomado es la de eliminar la relación a nivel de dominio de los objetos *venta*, *opinion* y *recurso* del *videojuego*, (no así de los objetos *empresa*, *genero* y *plataformas*) de tal forma que liberamos la carga de los objetos videojuego para su procesado y/o consulta, de tal forma que el rendimiento no se verá tan mermado en uno de los aspectos vitales de este Portal Web. Pero aunque hayamos eliminado estas relaciones, el controlador asociado a los videojuegos las mantiene a nivel de la capa de datos, mediante las relaciones de la base de datos existentes entre los videojuegos y las ventas, opiniones y recursos, y es más, éstos objetos no se crearán hasta que realmente fueran necesarios, con la consiguiente reducción de carga de objetos en memoria.

9.2.5. Implementación del Control de Acceso al Sistema

El control de acceso al sistema se rige mediante la tabla de privilegios de MySQL, agrupándose bajo una misma base de datos llamada *mysql*.

Para gestionar los diferentes permisos habremos de introducir modificaciones en dicha base de datos. Puesto que se trata de una base de datos más, MySQL puede trabajar con ella como si de una base de datos *normal* se tratase: es decir, podremos utilizar diferentes sentencias SQL para añadir, actualizar o borrar entradas (permisos). Sin embargo, este método resulta obsoleto. MySQL introduce un segundo método para gestionar los permisos, que es el recomendado, basado en el uso de dos comandos especiales: *grant* y *revoke*, siendo éste el método que emplearemos y estudiaremos a continuación.

Para realizar este control hemos definido los siguientes tipos de usuarios con sus permisos sobre la base de datos en base a sus necesidades de acceso:

Tipo de Usuario	**Usuario BD**	**Permisos**
Usuario Anónimo	anonimo	SELECT
Usuario Registrado	user_reg	SELECT, INSERT, UPDATE, DELETE
Moderador de Foro	mod_foro	SELECT, INSERT, UPDATE, DELETE
Gestor de Contenidos	gest_contenidos	SELECT, INSERT, UPDATE, DELETE
Administrador	admin	SELECT, INSERT, UPDATE, DELETE

Cuadro 9.2: Usuarios y Permisos de Acceso a la Base de Datos

Para llevar a acabo esta política de acceso hemos empleado dos clases:

La clase controladora *ControladorGestUsuarios*, que valida al usuario y establece los permisos de acceso a la base de datos. Veamos los métodos que intervienen en este proceso:

```
// ControladorGestUsuarios::entrarSistema(), encargado de la autentificación de usuarios

function entrarSistema(){

  /* PARAMETROS QUE RECIBE
   * $login: login del usuario
   * $password: password */
  if( isset($_POST) ){
    $login = $_POST['login'];
    $password = $_POST['pwd'];
    $permisos = $this->daoUsuario->Autentificar($login,$password);
    $conexion =& ConexionDAO::getInstancia();
    switch($permisos){
      case 'usuario': $conexion->setUser('user_reg'); break;
      case 'gestor de contenidos': $conexion->setUser('gest_contenidos'); break;
      case 'moderador de foro': $conexion->setUser('mod_foro'); break;
      case 'administrador': $conexion->setUser('admin'); break;
    }
```

```
    if( $permisos != false ){
      session_start();
      $_SESSION["login"]=$login;
      $_SESSION["permisos"]=$permisos;
    } else
      Error::Error('Password incorrecto');
  }
  $vista =& VistaGestUsuarios::getInstancia();
  $vista->mostrar();
}

// ControladorGestUsuarios::desconectar(), encargado de cerrar la sesión del usuario

function desconectar(){
  session_start();

  if(isset($_SESSION)){
    session_unset();
    session_destroy();
  }

  $conexion =& ConexionDAO::getInstancia();
  $conexion->setUser('anonimo');
  $vista =& VistaGestUsuarios::getInstancia();
  $vista->mostrar();
}
```

La clase *ConexionDAO* para establecer el usuario a conectarse en cada momento. Veamos los métodos que intervienen en este proceso:

```
// ConexionDAO::getLink(), devuelve una conexión a la Base de Datos

function getLink(){

  $this->link =& ADONewConnection($this->dbtype);
  $this->link->debug=$this->dbdebug;
  $this->link->NConnect($this->dbhost,$this->dbuser,$this->dbpassword,$this->dbdatabase);
  if( !$this->link->IsConnected() )
    Error::Error('fallo de conexión');
  return $this->link;
}

// ConexionDAO::setUser(), establece el usuario para la siguiente conexión a la BD

function setUser($usuario){
  if( trim($usuario)=='mod_foro' || trim($usuario)=='anonimo' || trim($usuario)=='user_reg' ||
      trim($usuario)=='gest_contenidos' || trim($usuario)=='admin' )
    $this->dbuser=$usuario;
  else
    Error::Error('No se reconoce el usuario');
}
```

Básicamente, cada vez que se identifique un usuario, accedemos a la base de datos según el tipo de usuario y en las siguientes conexiones se realizarán con el usuario indicado. Cuando cierre la sesión con la aplicación, el usuario por defecto es el anónimo con sólamente poder consultar la información de la base de datos.

9.2.6. Implementación de la Validación de Datos

La validación de datos se realiza a tres niveles fundamentalmente:

- Nivel Local: antes de enviar los datos al servidor, se usa una función *JavaScript*[2] que valida los campos del formulario. Para ello usamos una función que recibe dos arrays: la lista de campos a comprobar, y la lista de filtros a aplicar a cada campo. Por ejemplo: obligatorios = ['login','password1','password2','email'] y filtros = ['STR','PWD','PWD','MAIL'], de tal forma que en cada posición se aplica la validación indicada en filtros[].
- Nivel de Clases del Sistema, es decir, son las clases del sistema las encargadas de realizar la comprobación de datos, como por ejemplo la validación del correo electrónico o la fecha:

```
// Usuario::setCorreo(), establece el correo electrónico del usuario
function setCorreo($correo) {

  if( ereg("^([a-zA-Z0-9_.-])+@(([a-zA-Z0-9-])+.)+([a-zA-Z0-9]{2,4})$",$correo)
      || $correo=="" )
    $this->correo = $correo;
  else
    Error::Error('No es correcto el correo electrónico');
}
```

- Nivel del Controlador asociado, es decir, antes de crear los objetos, se deben realizar comprobaciones de validación de campos de los formularios mostrados, como por ejemplo la introducción de contraseñas o precios

```
// porción de código de ControladorGestUsuarios::registrarUsuario(), registra usuarios

$password = $_POST['password1'];
$passwordRep = $_POST['password2'];

if( trim($password)!='' and trim($passwordRep)!='' and $password == $passwordRep ) {
  $usuario = new Usuario($login,$password,$imagen,$correo,$tipo);
  $this->daoUsuario->crearUsuario($usuario);
  // si ha ido bien consultamos su ficha
  $this->mostrarFormularioDatosUsuario($usuario->getLogin());
} else {
  if( $imagen!=null )
    eliminarFichero($imagen);
    Error::Error('Las contraseñas no coinciden');
}
```

Cuando se produce un error, mantenemos los valores introducidos en el formulario, indicando los campos erróneos, para ello usamos la variable del sistema *$_POST[]*. Para el caso de las fechas, la decisión que hemos tomado es la de utilizar un calendario, en el que el usuario de forma directa pueda introducir la fecha sin errores de formato. Para ello hemos tenido que emplear código *JavaScript* que realice todo el proceso, además de establecer el campo de la fecha desactivado para impedir al usuario que pueda escribir directamente en él.

[2]tiene el inconveniente de no ser soportado por el navegador utilizado

9.2.7. Implementación de la Internacionalización

Para aplicar un idioma (actualmente español e inglés), tenemos que definir para cada elemento a traducir una clave que se asignarán en las clases de la capa de presentación y que al mostrarse finalmente, éstas se reemplazan mediante un array que se genera dinámicamente en función del idioma.

El código encargado de traducir las claves al idioma indicado mediante la variable de entorno *IDIOMA* es el siguiente:

```
// traducción al idioma
$idiomaDAO =& IdiomaDAO::getInstancia();

// comprobamos que existe el archivo con las claves y si hemos cambiado de idioma
if( !file_exists($idiomaDAO->getDirArchivoIdioma()) ||
  (trim(getenv("IDIOMA"))!=trim($idiomaDAO->getIdioma())) )
{
   $idiomaDAO->setIdioma(getenv("IDIOMA"));   // obtenemos el idioma a cambiar
   $dir = $idiomaDAO->getClaves();            // generamos la traducción de las claves
} else
   $dir = $idiomaDAO->getDirArchivoIdioma();  // obtenemos el nombre del fichero

require($dir);   // cargamos el array con las claves, llamado $traduccion

// obtenemos la traducción de la clave 'MC_INICIO'
$html .= "<a href='$dir'>".strtoupper($traduccion['MC_INICIO'])."</a> $sep ";
```

El método *getClaves()* escribe en un fichero (*idioma.php*) un array asociativo donde se asigna a cada clave la cadena de texto en el idioma indicado, de tal forma que para el caso del español tendríamos:

```
$traduccion['MC_INICIO'] = 'Inicio'        // traducción en español de la clave 'MC_INICIO'
$traduccion['MC_INICIO'] = 'Home'          // traducción en inglés  de la clave 'MC_INICIO'
```

Como característica principal de este proceso es que la generación del fichero con las claves es automática, mediante la consulta a la tabla de idioma, de tal forma que para traducir nuevos conceptos, hay que:

1. añadir una nueva fila a la tabla *pwv_idioma* indicando la nueva clave y las traducciones de cada idioma

2. cambiar en las plantillas o en las clases de la capa de presentación el texto por la nueva clave, entre llaves

Una vez realizado esto, la traducción del concepto está lista para consultarse, sin apenas tocar código de la aplicación.

9.2.8. Implementación del Nivel de Accesibilidad *A* (WCAG 1.0)

Uno de los requisitos imprescindibles para los sistemas web actuales es que ofrezcan un mínimo de accesibilidad a todo tipo de usuarios, es por esto que la WAI[3] proporciona, como guía para el diseño de sitios web accesibles, una especificación conocida como las 'Pautas de Accesibilidad al Contenido Web, 1.0'. Esta especificación está compuesta por 14 pautas (*guidelines*) que describen las normas básicas de diseño Web accesible. Cada una de estas pautas se asocia a uno o más puntos de verificación (*checkpoints*), que detallan el uso que debe hacerse de estas pautas en puntos concretos del diseño Web.

Se establecen un conjunto de puntos de verificación que el diseñador debe cumplir en función del grado de accesibilidad que se desee alcanzar. Nuestro objetivo es la de cumplir el *Nivel de Adecuación A* o *Prioridad 1*. Los puntos de verificación de tenemos que cumplir son los siguientes:

Puntos de verificación de Prioridad 1: Generales

Punto de verificación 1.1 Proporcione equivalentes textuales a todos los elementos no textuales (vía *alt*, *longdesc*, etc). Esto incluye a imágenes, gráficas, representaciones gráficas de texto, mapas de imagenes, animaciones (por ejemplo, GIFs), applets y objetos programáticos, ASCII art, scripts, imágenes usadas como viñetas de párrafos, espaciadores, botones gráficos, sonidos y vídeo.

Punto de verificación 2.1 Asegúrese de que toda información proporcionada mediante colores también se encuentra disponible sin colores.

Punto de verificación 4.1 Identifique claramente cambios en el lenguaje natural de un texto de un documento y en cualquier equivalente de texto.

Punto de verificación 6.1 Organice documentos de forma que sean comprensibles sin hojas de estilo. Por ejemplo, cuando un documento HTML es interpretado sin hojas de estilo, debe ser posible su lectura.

Punto de verificación 6.2 Asegúrese de que las equivalencias para el contenido dinámico se actualizan al cambiar el contenido dinámico.

Punto de verificación 7.1 Hasta que los agentes de usuario permitan controlar los destellos, evite provocar destellos en la pantalla.

Punto de verificación 14.1 Use el lenguaje más claro y sencillo que el contenido de la página permita.

[3]Web Accesibility Initiative

Puntos de verificación de Prioridad 1: Imagenes

Punto de verificación 1.2 Proporciones enlaces de texto redundantes para cada región activa en un mapa de imágenes de servidor.

Punto de verificación 9.1 Proporciones mapas de imágenes de cliente en lugar de servidor siempre que sea posible, excepto cuando las regiones no puedan ser definidas conuna forma geométrica disponible.

Puntos de verificación de Prioridad 1: Tablas

Punto de verificación 5.1 Para las tablas de datos, identifique los encabezamientos de filas y columnas.

Punto de verificación 5.2 Para tablas de datos con dos o más niveles lógicos de encabezamientos de filas o columnas, use marcadores para asociar las celdas de datos con las de encabezamientos.

Puntos de verificación de Prioridad 1: Marcos

Punto de verificación 12.1 Titule cada marco para facilitar la información y navegación entre marcos.

Puntos de verificación de Prioridad 1: Applets y Scripts

Punto de verificación 6.3 Asegúrese de que las páginas son visibles y usables aún cuando scripts, applets u otros objetos programáticos no sean soportados por el usuario. Si esto no fuera posible, provea información equivalente en una página accesible alternativa.

Puntos de verificación de Prioridad 1: Multimedia

Punto de verificación 1.3 Hasta que los agentes de usuario puedan leer automáticamente el texto equivalente de un vídeo, provea una transcripción sonora de la información importante del vídeo o la presentación multimedia.

Punto de verificación 1.4 Para cualquier presentación multimedia basada en tiempos (vídeo o animación), sincronice las alternativas equivalentes (etiquetas o descripciones sonoras del contenido) con la presentación.

Puntos de verificación de Prioridad 1: Si lo anterior falla o no es aplicable

Punto de verificación 11.4 Si, tras todos los esfuerzos aplicados, no puede crear una página accesible, proporcione un enlace a una página Web alternativa que use tecnologías W3C, sea accesible, tenga información (o funcionalidad) equivalente y se actualice a la par que la página original (no accesible).