PHPWebBuilder

Framework integral para el desarrollo de aplicaciones Web sobre PHP

Alejandro Siri Mariano Montone

Resumen

El diseño y desarrollo de aplicaciones web plantea muchos problemas reiterativos. Existen librerías, frameworks y herramientas que facilitan el desarrollo de cada uno de los aspectos que componen una aplicación de este tipo. Sin embargo, la responsabilidad de seleccionar aquellas que son apropiadas e integrarlas queda en general a cargo del programador. En este artículo presentamos *PHPWebBuilder*, un Framework Open Source Orientado a Objetos escrito en PHP para el desarrollo de aplicaciones Web que integra soluciones a cada uno de estos problemas. Estas fueron seleccionadas, desarrolladas y concretadas en el desarrollo de este framework en base a la experiencia que obtuvimos en el desarrollo de este tipo de aplicaciones.

Keywords: Templates declarativos, desarrollo por componentes, mapeo objeto-relacional, persistencia por alcance, AJAX, Comet, compilación, macros, DSLs, Mixins.

1. Introducción

El diseño y desarrollo de aplicaciones plantea muchos problemas reiterativos. La interacción con el usuario, la presentación de la información, el almacenamiento y la recuperación de la misma son algunos de los problemas más frecuentes y que más tiempo consumen. Por esto mismo han recibido mucha atención habiendo obtenido a lo largo de los años múltiples soluciones. Seleccionar e integrar cada una de éstas queda a cargo del programador o grupo de trabajo de un proyecto específico.

PHPWebBuilder[1] es un framework de desarrollo que integra múltiples soluciones que según lo experimentado por nosotros, mejoran el tiempo de desarrollo y la calidad del producto de software sin sacrificar flexibilidad en el diseño.

El framework está diseñado bajo una arquitectura MVC (Model-View-Controller)[2]. Esto quiere decir que una aplicación se compone de 3 capas:

- El *Modelo*. Es la representación de la información de dominio específico de la aplicación.
- El *Controlador*. Está basada en la programación de componentes. Responden a la acción del usuario y definen los aspectos navegacionales de la aplicación (más sobre ésto en la sección 3).
- La *Vista*. La forma en la que se presentan los datos y botones que el usuario ve, se define a través de templates declarativos, HTML y CSS. Desarrollamos los aspectos de presentación en la sección 4.

Existe además otra parte que integra al framework. Esto es la "Programación en lo pequeño" (*Programming in the Small*), la programación de los módulos y funciones del programa, que sirven para unir estas capas. Los frameworks actuales ayudan a simplificar el desarrollo bajo MVC; el soporte para *Programming in the Small* generalmente es dependiente del lenguaje/plataforma. *PHPWebBuilder* se apoya en PHP para parte de esta tarea, y además provee al desarrollador herramientas para solucionar o simplificar las tareas en las otras áreas:

- Para el modelo, presenta un mapeo automático de base de datos (sección 2.2), un lenguaje de consultas complejas (sección 2.4) que tiene en cuenta la herencia del modelo de clases y autogeneración del esquema de base de datos (sección 2.5). Esto permite que la tarea del desarrollador se limite a enfocarse a resolver la problemática que plantea el diseño del modelo.
- Para el controlador, utiliza un sistema de componentes (sección 3.1) que permiten reutilizar "partes de aplicación" , ya sea en diferentes partes de una misma aplicación o en diferentes proyectos. Además, los widgets (sección 3.2) simplifican y encapsulan la interacción con el usuario, y con el context y múltiple dispatching (sección 3.4) los componentes se pueden adaptar a muchos contextos.

- Para la vista, el sistema de templates (sección 4.1) basados en XML permiten una adaptación directa del trabajo de un diseñador gráfico (sección 4.1.1). El sistema se encarga de presentar la información utilizando AJAX de manera transparente o mediante otra forma de rendereo (sección 4.2).
- Por último, para *Programming in the Small*, *PHPWebBuilder* presenta características disponibles en otros lenguajes y plataformas pero no existentes en PHP. Algunas de ellas son la implementación de Eventos (sección 5.2) Weak References (sección 5.3), creación de DSLs (sección 5.4), Mixins (sección 5.6). y Macros (sección 5.5).

2. Características del Modelo

Para ejemplificar, veremos la creación de un blog (figura 1).

Figura 1: Nuestro blog

Las tareas a realizar son:

1. Crear el modelo de clases.
2. Decidir cuáles son las formas de interacción de los usuarios con el sistema.
3. Presentar esas formas de interacción de una manera atractiva y entendible para el usuario.

2.1. Diseño de la aplicación

El Modelo de la aplicación, es la representación de la información de dominio específico de la aplicación. Para el blog, tendremos las siguientes clases: *Post*, *Tag*, *User* (figura 2).

2.2. Persistencia y metadata

Los datos del modelo necesitan en general ser persistidos para que éstos puedan ser accedidos en otra ejecución de la aplicación, ya sea simultánea o posterior.

PHPWebBuilder posee un mecanismo para la persistencia que permite, mediante anotaciones especiales en los datos del modelo, el guardado y recuperación automática de los datos en la base de datos.

La persistencia del modelo se consigue mediante la subclasificación de la clase especial `PersistentObject`. Esta clase provee la funcionalidad necesaria para describir la metadata en su inicialización y también para las operaciones de guardado y borrado.

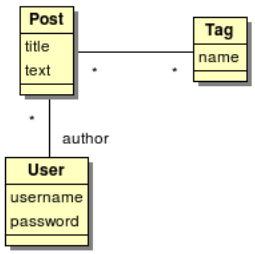

Figura 2: Modelo de objetos del Blog

De esta manera declaramos la clase `Post`:

```
class Post extends PersistentObject {
  function initialize() {
    $this->addField(new TextField(array('fieldName'=>'title')));
    $this->addField(new TextArea(array('fieldName'=>'text')));
    $this->addField(new CollectionField(
      array(
        'fieldName'=>'tags',
        'direct'=>false,
        'JoinType'=>PostTag,
        'targetField'=>'tag',
        'reverseField'=>'post')
      )
    );
    $this->addField(new IndexField(
        array('fieldName' => 'author',
              'type' => 'User')
         )
        );
  }
}
```

Esto define una clase persistente `Post`. En ella definimos un método initialize en el que declaramos los atributos que tiene un `Post`. En este caso son un *título* (`title`), un *texto* (`text`), una colección de *etiquetas* (`tags`) y el usuario *autor* (`author`). La definición de un atributo se hace invocando el método `addField` de `PersistentObject` con el tipo de campo que queremos crear.

La clase `PersistentObject` posee los métodos `save` y `delete`, que permiten insertar, actualizar y borrar objetos de la base de datos.

Para crear y guardar un `Post` de título 'Model persistence' hacemos:

```
$p = new Post;
$p->title->setValue('Model persistence');
$p->save();
```

Esto deja el `Post` persistido en la base de datos.

2.3. Recuperación de objetos

Una operación habitual en el desarrollo de una aplicación es la recuperación de datos y de información calculada en base a estos.

La recuperación de objetos en *PHPWebBuilder* se hace mediante la clase `Report` que permite obtener los objetos de una colección, filtrada por varios criterios y ordenada. Estos reportes tienen en cuenta las subclases de los objetos y utilizan herencia para las variables de instancia.

Para obtener todos los `Posts` cuya etiqueta sea una dada creamos el siguiente objeto Report:

```
new Report(array(
  'class' => 'Post',
  'target' => 'p',
  'exp' => new ExistsExpression(
   new Report(array(
     'collection' => new CollectionPathExpression('p', 'tags',' tag'),
     'exp' => new Condition(array(
         'exp1' => new AttrPathExpression('tag','name'),
         'operation' => '=',
         'exp2' => new ValueExpression("$tag")
       ))
     ))
   )
  ));
```

Esta consulta, genera el siguiente SQL:

```
SELECT
   `p_post`.`id` AS `post_id`,
   `p_post`.`title` AS `post_title`,
   `p_post`.`text` AS `post_text`,
   `p_post`.`PWBversion` AS `post_PWBversion`
 FROM `post` AS `p_post`
 WHERE EXISTS
  (SELECT
      `tag_posttag_tag`.`id` AS `tag_id`
    FROM `posttag` AS ` tag_posttag`, `tag` AS `tag_posttag_tag`
    WHERE `tag_posttag`.`post` = `p_post`.`id`
      AND `tag_posttag`.`tag` = `tag_posttag_tag`.`id`
      AND `tag_posttag_tag`.`name` = $tag
  )
```

Como podemos ver, no se acorta la cantidad de código escrito pero se simplifica. Alcanza con determinar la clase sobre la que hace la consulta (`Post`) e indicar las restricciones (en este caso, que solamente queremos aquellos Post que posean a `$tag` entre los elementos de su colección).

El programador no tiene que preocuparse por seleccionar correctamente los campos del objeto. Los joins de tablas se hacen automáticamente (incluso los de herencia, en este ejemplo no se nota por su simplicidad).

La principal ventaja de generar las consultas es que un cambio en las clases del modelo no implica necesariamente modificar las consultas a mano (por ejemplo, si agregásemos un campo 'fecha de publicación' en el `Post`). Esto mejora notablemente la productividad.

2.4. OQL (Object Query Language)

Dada la complejidad de construir un reporte completo a mano, desarrollamos un lenguaje *OQL* para consulta de los objetos. Usando esto, para obtener todos los `Posts` cuya etiqueta sea una dada, creamos esta consulta:

```
#@select p:Post where exists (p.tags as tag where tag.name=$tag)@#;
```

Vemos que la complejidad de crear el reporte se disminuye en una alta proporción. La extraña sintaxis (los tokens `#@ @#` que encierran la expresión de consulta) corresponde a la utilización de macros, que veremos en la sección 5.5.

Una buena propiedad del lenguaje es que está íntimamente relacionado con el código PHP, ya que puede utilizar las variables en el scope (es decir, el OQL está embebido en el PHP).

Para buscar los post que tengan el tag `$tag`, agregamos en la clase `Post` el siguiente mensaje de clase:

```
class Post extends Component {
  ...
  function Tagged($tag){
    return #@select p:Post
                  where exists
                        (p.tags as tag where tag.name=$tag)
           @#;
  }
  ...
}
```

2.5. Adaptación de base de datos

Para poder persistir los datos del modelo, se necesita un repositorio. *PHPWebBuilder* utiliza una base de datos relacional para ello y realiza un mapeo Objeto-Relacional de los datos. Para poder realizar este mapeo, es necesario que la base de datos mantenga una estructura específica.

La estructura necesaria para la persistencia del modelo se puede inferir de la definición de los objetos, donde establecemos los tipos de las variables de cada clase y las relaciones de subclasificación.

PHPWebBuilder genera el esquema automáticamente librando al programador de esta carga, que es generalmente hasta más larga (y muchísimo más tediosa) que la programación del modelo (figura 3).

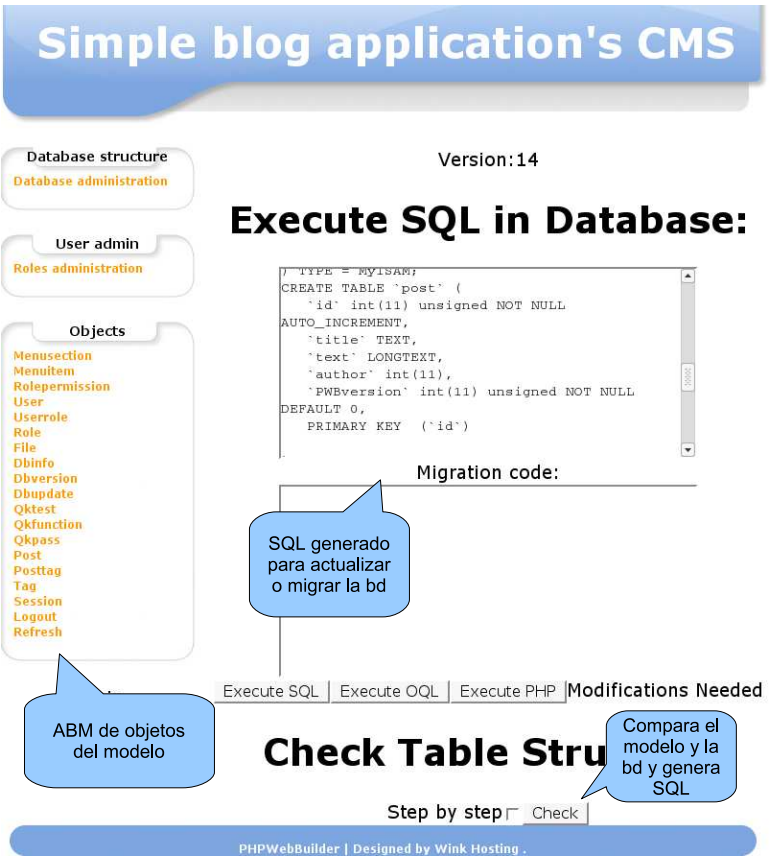

Figura 3: Adaptación de base de datos

Además, podemos verificar que el esquema de base de datos sea el correcto, y mostrar y ejecutar las correcciones necesarias. Esto también es muy útil cuando se hace una modificación en el modelo de una aplicación y se necesita adaptar la base de datos a los nuevos cambios.

Por último, *PHPWebBuilder* incluye una aplicación para la administración de los objetos del modelo como se ve en las figuras 4 y 5.

Figura 4: ABM Autogenerado de un Post

Figura 5: Aplicación de administración de Posts generada

3. Características del Controller

Teniendo ya el modelo programado y persistido, es posible implementar la interacción con el usuario. Veremos cómo listar los Posts del blog y cómo permitirle al usuario ver solamente los posts que tengan un tag.

3.1. Componentes

Las aplicaciones en *PHPWebBuilder* se construyen mediante el ensamblado de componentes. Estos componentes son reutilizables ya que pueden ser parametrizados, pueden disparar eventos y pueden contener otros componentes.

El componente principal de la aplicación del blog consiste simplemente de una lista de posts. Llamemos a este componente `BlogComponent`.

```
class BlogComponent extends Component {
  function initialize(){
      $this->addComponent(new PostList(#@select Post@#));
  }
}
```

Un componente agrega sub-componentes definiendo un método `initialize` e invocando a `addComponent`, dejándolo como *hijo* suyo en el árbol de componentes de la aplicación. Lo interesante de esta forma de diseñar la aplicación es que podemos obtener un componente a partir de otros componentes ya implementados y probados.

Ahora definamos la lista de `Posts`. Sólo basta subclasificar de `CollectionNavigator`, un componente general para navegación y mostrado de listados de objetos:

```
class PostList extends CollectionNavigator {
  function addLine($post){
    return new PostItem($post);
  }
}
```

Un componente `CollectionNavigator` presenta links para la navegación de los elementos (mostrándolos paginados). Para determinar cómo se van a manejar cada uno de los elementos se retorna un componente en el método `addLine`, en este caso un `PostItem`.

Una de las ventajas que tiene esta implementación del controller es que podemos ensamblar componentes libremente, casi sin restricciones y sin preocuparnos por formularios o pasaje de parámetros por URL, conceptos que surgen al desarrollar aplicaciones web en casi cualquier otro framework.

3.2. Widgets

Necesitamos crear ahora el componente `PostItem`, que muestra cada `Post`. Este debe mostrar el título, el texto y las etiquetas.

La interacción del usuario con la aplicación se logra por medio de `Widgets`[3]. Estos son componentes especiales entre los que se encuentran el componente `Input`, que permite recibir un string del usuario, el `Text`, que le presenta un texto y el `CheckBox`, que presenta un checkbox. Utilizaremos estos componentes para mostrar y lograr la interacción del usuario con un `Post`:

```
class PostItem extends Component {
  function PostItem($post){
    $this->post = $post;
    parent::Component();
  }
  function initialize(){
    $this->addComponent(new Text($this->post->title), 'title');
    $this->addComponent(new Text($this->post->text), 'text');
    // Mostramos las etiquetas del post
    $this->addComponent(new Component, 'tags');
    foreach($this->post->tags->collection->elements() as $tag){
      $this->tags->addComponent(new Text($tag->name));
    }
  }
}
```

3.3. Control de flujo modal

Queremos ahora poder ver la lista de `Posts` filtrada por los que tienen una determinada etiqueta al hacer click sobre ésta. Reveamos entonces nuestras clases `PostItem` y `PostList` para establecer la interacción necesaria.

Modificamos primero la clase `PostItem` para convertir las etiquetas mostradas a links. Para esto agregamos widgets de clase `CommandLink` en lugar de los de clase `Text` anteriores. Queremos actuar cuando una etiqueta es seleccionada, por lo tanto enviamos al padre del `PostItem` que invoque el método `showTag`.

```
class PostItem extends Component {
  function PostItem($post){
    $this->post = $post;
    parent::Component();
  }
  function initialize(){
    $this->addComponent(new Text($this->post->titulo), 'title');
    $this->addComponent(new Text($this->post->texto), 'text');
    // Mostramos las etiquetas del post
    $this->addComponent(new Component, 'tags');
    foreach($this->post->tags->collection->elements() as $tag){
      $this->tags->addComponent(new CommandLink(array(
        'text'=>$tag->name,
        'prodeedFunction'=>
          new FunctionObject(
            $this->getParent(),
            'showTag',
            array('tag'=>$tag)
          )
        ))
      );
    }
  }
}
```

El `CommandLink` es un `Widget` especial que permite al usuario ejecutar una acción en la aplicación.

Necesitamos que la lista de `Posts` se entere de que una etiqueta fue seleccionada y actuar en consecuencia. Modificamos la clase `PostList` para que, al recibir el mensaje `showTag` que le manda el `PostItem`, delegue su comportamiento a otra lista cuyos `Posts` son sólo aquellos que poseen la etiqueta seleccionada. La delegación de control se logra mediante la invocación del método `call`. Desde el punto de vista de la vista, la delegación de control implica que el componente llamante reemplazará su vista por la del componente llamado.

```
class PostList extends CollectionNavigator {
  function addLine($post){
    return new PostItem($post);
  }
  function showTag($params){
    $this->call(new PostList(Post::Tagged($params['tag'])));
  }
}
```

Sería bueno que una vez mostrada la lista de posts con una determinada etiqueta podamos volver a la lista de posts de la que partimos (figura 6). Para esto utilizamos un botón `volver` cuyo comportamiento será hacer `callback` sobre la nueva lista como mostramos a continuación:

```
class PostList extends CollectionNavigator {
  ...
  function addBackButton() {
    $this->addComponent(new CommandLink(
        array('text' => 'back',
              'proceedFunction' => new FunctionObject($this, 'callback')
        )
    ));
  }

  function showTag($params){
    $list = new PostList(Post::Tagged($params['tag']));
    $this->call($list);
    $list->addBackButton();
  }
  ...
}
```

Como se puede ver, se pueden agregar componentes fuera del método initialize, en este caso en el `addBackButton`. Cuando un componente es modificado de manera dinámica, solamente sus cambios se actualizan en la vista (por ejemplo, mediante AJAX - sección 4.2).

Figura 6: Posts con etiqueta 'miscellaneous' y el botón para volver

La interacción es modal ya que el componente que llama a otro mediante `call`, le cede el control, que pasa al componente llamado hasta que haga `callback`. Una vez hecho el `callback` el control vuelve al componente llamante para que continúe con su ejecución. A nivel de vista, la vista del componente llamante es reemplazada por la del componente llamado. Esta forma de interacción es similar al uso de diálogos en aplicaciones de tipo desktop. En éstas, abrir un diálogo de forma modal bloquea el acceso a toda la aplicación. Nuestra forma de interacción modal sólo bloquea al componente llamante y por lo tanto sólo a una parte específica de la aplicación. Esa es la principal diferencia con respecto a la utilización de diálogos modales en

aplicaciones de tipo desktop tradicionales.

3.4. Multiple Dispatching y Context Dispatching

Otra característica llamativa, es el múltiple dispatching de funciones. Dado que muchas veces el método a utilizar depende de más de una clase, las funciones de múltiple dispatching nos ayudan a resolver este problema.

Primero se define la función, con los parametros a utilizar tipados, y luego se hace el llamado, que utiliza los tipos de los parámetros para resolver el método a utilizar.

Además, las funciones de múltiple dispatching pueden ser utilizadas pasando el contexto (la rama de componentes dentro de la que se hace el llamado) de aplicación, para de esta manera poder responder de manera diferente a un evento, dependiendo del contexto.

Por ejemplo (saliendo del ejemplo del blog), podemos tener un componente TaskList que muestre las tareas realizada en un día. Para cada tarea muestra la descripción, la hora, y el nombre de quién la completó en un componente TaskList. Si luego quisiéramos ver las tareas realizadas por uno de los usuarios, nos interesaría ver la hora, la descripción, pero no el nombre del usuario, porque ya se sabe por el contexto. Podríamos usar en ese caso un componente UserTaskShow.

Para implementar esa diferencia, una forma común sería tener 2 componentes, TaskList y UserTaskList. Esto duplica código, y además es difícil de implementar cuando el contexto de los componentes es de varios componentes (ya que tendríamos que hacer un nuevo componente para cada elemento en la rama de componentes hasta llegar al que realmente implementa la diferencia).

Entonces, podemos utilizar el Context Dispatching para elegir un componente UserTaskShow, en lugar del otro, el TaskShow, independientemente del nivel de anidamiento del componente que hace el dispatch y el componente que provee el contexto.

4. Características de la Vista

Con la interacción de la aplicación ya implementada, falta presentar la información al usuario de manera entendible.

4.1. Templates

Necesitamos darle formato a los datos a presentarle al usuario. Para ésto utilizaremos templates, que se adaptan a cada uno de los componentes.

Para el `PostItem`, mostramos de manera resaltada el título, y el cuerpo, y más abajo los tags:

```
<templates>
  <template class="PostItem">
    <h1 id="titulo" />
    <p id="texto" />
    Tags: <div id="tags"><container class="CommandLink"/> </div>
    <hr/>
  </template>
</templates>
```

Hacemos un template para los componentes de clase `PostItem`, asociamos al subcomponente de id *titulo* al elemento `h1` del html, el de id *texto* al `p`. Por último, hacemos un `div` para el componente de id *tags*, y decimos que posicione a los `CommandLink` hijos de *tags* con su template default.

Los templates tienen las siguientes características:

- Los templates se "heredan", así que un Componente subclase de otro que tiene template, hereda de este el template (siempre y cuando no tenga uno más específico). También son heredables por `Mixins` (sección 5.6).
- Los templates son declarativos: No incluyen comandos ni iteradores (como tienen otros engines de templates). De esta manera, el control de la aplicación está 100 % en los componentes, y además nos aseguramos que los templates generen un XML bien formado.

- Los templates están basados en XML, con un par de tags extras (`<template>` y `<container>`), así que se puede generar cualquier XML (como XHTML) para mostrar los componentes.

Además, cada componente tiene un template default, por lo que no se necesita crearle uno para tener la aplicación funcionando (en nuestro caso, no lo hicimos para el `CommandLink`, el `BlogComponent` ni el `PostList`).

4.1.1. Adaptación de Diseños existentes

Debido a que los templates son XML, se puede tomar una página en HTML y agregarle los tags `<template>` y `<container>` donde se quiera, y de esta manera conseguir un template a muy bajo costo. Además, como los templates se heredan, utilizando `Mixins` (sección 5.6) y subclases, una aplicación puede tener un diseño completo en sólo 3 o 4 templates.

4.2. Renderings

Una técnica muy utilizada en las aplicaciones web 2.0, es el rendereo de cambios mediante AJAX, que permite al usuario ejecutar varias tareas en simultáneo en la aplicación y refrescar la pantalla sin recargar todo.

El rendereo (generación de la interfaz) de la aplicación se hace 100 % mediante *PHPWebBuilder*, de modo que no hay intervención por el programador.

La manera de cambiar el engine de rendereo es tan simple como cambiar en el archivo de configuración, donde dice `page_renderer=StandardPageRenderer` por `page_renderer=AjaxPageRenderer`, o incluso por `page_renderer=CometPageRenderer` o `page_renderer=XULPageRenderer`.

4.2.1. AJAX y Comet

Todo lo generado por *PHPWebBuilder* es XML bien formado, de modo que de manera transparente podemos renderear en AJAX.

Comet es una tecnología similar a AJAX que se caracteriza por mantener una conexión abierta entre el browser y el servidor en todo momento. *PHPWebBuilder* aprovecha esto, ya que un importante tiempo de procesamiento de los scripts PHP es la carga del script y los datos de la sesión de usuario, que en este caso quedan vivos en la memoria del servidor.

Mediante pasaje de mensajes de otro script que envía los datos a la aplicación *PHPWebBuilder* ejecutándose, podemos mantener la sesión en memoria, mejorando los tiempos de respuesta e incluso modificando la aplicación del usuario. Esta modificación puede surgir a partir de la generación de un evento por parte del usuario (por ejemplo, un click del mouse) o por sucesos ajenos a ésto, como puede ser la modificación de base de datos.

4.2.2. XUL

El proyecto Mozilla incluye un subproyecto llamado XUL[4]. XUL es un lenguaje de definición de interfaces desktop en XML. Dado que la salida de la aplicación debe ser un XML bien formado y que la interacción usuario-interfaz se hace mediante javascript, *PHPWebBuilder* soporta un XUL Page Renderer que renderea aplicaciones en XUL. La diferencia para el programador se encuentra en el trato de las etiquetas de los templates ya que debe utilizar elementos XUL en lugar de HTML.

Dado que los templates de HTML llevan extensión .xml y los de xul .xul, la misma aplicación, con tener templates de los 2 tipos para cada componente, puede ser rendereada de las 2 maneras.

5. Características de *Programming in the Small*

5.1. BugNotifier

PHPWebBuilder incluye un `BugNotifier`, que automatiza el manejo de errores “no manejados” , permitiendo al usuario que encuentra una condición de error dentro de la aplicación, enviar el reporte de error a un mail configurado a tal fin, y también reiniciar la aplicación para continuar utilizándola.

En caso de que en la aplicación de Blog se encuentre un error, en lugar de presentarle al error, se le presentará con un dialogo.

5.2. Eventos

En nuestra implementación, el `PostItem` tiene un `CommandLink` que le envía a su componente padre el mensaje `showTag` cuando un tag es seleccionado. Esto trae problemas de composicionalidad, ya que no podríamos usarlo como hijo de un componente que no entienda el mensaje `showTag` (ver figura 7).

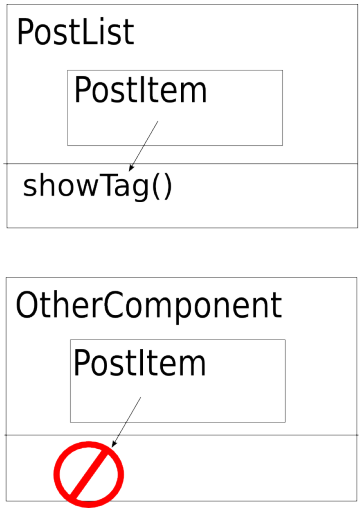

Figura 7: Componentes que no utilizan Eventos

Para realmente mantener la composicionalidad, necesitamos que el `PostItem` no envíe un mensaje al componente padre, pero al mismo tiempo, desde el `PostList` necesitamos saber que se hizo click en el tag. Para esto, utilizamos eventos. El componente `PostItem` dispara un evento cuando se hace click en un tag, y el `PostList`, que escucha cuando este evento se dispara, actualiza la lista de posts.

PHPWebBuilder implementa un mecanismo simple de eventos. Todo objeto `PWBObject` implementa los mensajes `addInterestIn`, que permite a otro objeto escuchar un evento, y `triggerEvent`, que dispara un evento.

Utilizando eventos, implementamos:

```
class PostList extends CollectionNavigator{
  function addLine($post){
    $pi = new PostItem($post);
    $pi->addInterestIn('tag_selected',
        new FunctionObject($this, 'showTag'));
    return $pi;
  }
  ...
}
```

Al crear el `PostItem`, el `PostList` se queda escuchando el evento `tag_selected` de él, y que cuando ocurra, se le envíe el mensaje `showTag`.

```
class PostItem extends Component{
  function initialize(){
    ...
    foreach($this->post->tags->collection->elements() as $tag){
      $this->tags->addComponent(new CommandLink(array(
        'text'=>$tag->nombre,
        'prodeedFunction'=>
                    new FunctionObject($this,
                            'tagSelected',
                            array('tag'=>$tag))
              )));
    }
  }
  function tagSelected($params){
    $this->triggerEvent('tag_selected', $params);
  }
}
```

El `PostItem`, en lugar de mandar un mensaje a su padre, se envía a sí mismo el mensaje `tagSelected`, que dispara un evento `tag_selected`, sin preocuparse de quién esté escuchando.

Existen objetos que disparan algunos eventos por default:

- Los `Widgets`, en cada acción del usuario (cuando hace click, cuando se modifica un elemento).
- Los objetos del modelo (`PersistentObject`), cuando son modificados.

Hay 2 momentos para la ejecución de respuestas a eventos: En el momento de creación del evento, o luego de la ejecución de la rama de programa, al finalizar las otras tareas. En este último caso, es posible ejecutar una sola o una vez por cada disparo del mismo evento. Esto se especifica en el `addInterestIn`.

5.3. Weak References

Cuando un objeto `$x` queda escuchando un evento de un `PWBObject $y`, este último necesita guardar una referencia al primero. En caso de que, por el flujo de la aplicación, `$x` deje de ser necesario, el mecanismo de garbage collection de PHP no puede descartarlo, porque `$y` lo conserva referenciado, aunque no lo necesite realmente.

Por esto implementamos un mecanismo de Weak References, en donde `$y` se queda con una referencia de `$x`, pero de la cual el garbage collector no se entera, permitiendo borrar a `$x` en caso de que sea necesario.

5.4. PHPCC

Para crear el lenguaje OQL, debimos crear un Compiler Compiler para PHP (otros conocidos son Bison[5], yacc[6]). De esta manera ahora también se puede extender el framework con múltiples DSLs (Domain-Specific Languages - Lenguages específicos de Dominio).

5.5. Macros

Un tiempo grande del desarrollo se destina a la corrección de errores. Una modificación de un módulo de programa puede afectar a otros módulos que lo utilicen, provocando un error difícil de detectar, y por ende costoso de solucionar. Una forma de detectar estos errores es mediante el chequeo de tipos y las aserciones, que permiten un control más estricto sobre los datos que se comunican entre módulos.

La ejecución de estas validaciones, si bien son útiles para el desarrollo, consumen un tiempo innecesario cuando el programa es instalado en producción. Una forma de recortar estos tiempos es eliminar del código final todos los chequeos, pero muchos pueden pasar sin ser detectados y pueden llegar a ser útiles ante una modificación posterior del sistema.

Por esto mismo, en *PHPWebBuilder* utilizamos `#@typecheck@#` y `#@check@#` para insertar validaciones desactivables en el código.

Por ejemplo, escribiendo `#@typecheck $post: Post@#` se chequea que la variable `$post` sea un `Post`. En las opciones de configuración de la aplicación, habilitamos o deshabilitamos el typechecking y el chequeo no se incluye.

```
class PostItem extends Component{
  function PostItem($post){
    #@typecheck $post: Post@#
    $this->post=&$post;
    parent::Component();
  }
  ...
}
```

Esto avisará al programador por problemas durante el desarrollo al intentar inicializar erróneamente un `PostItem` con algo que no sea un `Post`.

Habilitando o deshabilitando en el config.php (sección 5.7) `compile=typecheck`, podemos utilizar estos chequeos.

Tanto el OQL (sección 2.4) como `check` y `typecheck` son macros: fragmentos de código que se ejecutan una sola vez, en tiempo de compilación, y reemplazan su texto en el código de la aplicación final.

Un programador puede agregar sus propias macros declarando una función simple de PHP `mi_macro` y luego llamándola con `#@mi_macro` *parámetros* `@#`. Como se ve en lenguajes como C, las macros pueden llegar a tener un rol muy importante en un proyecto ya que son otra forma de modularización.

5.6. Mixins

Otra funcionalidad interesante es una implementación limitada de `Mixins` (mediante `Macros`).

Un `Mixin` es una forma de agrupar funcionalidad y agregársela a múltiples clases de objetos sin que estas clases estén conectadas por subclasificación. Es lo que vemos como el mejor trade-off entre simple y múltiple herencia.

```
#@mixin ValueHolder {
  var $value;
  function getValue(){
    return $this->value;
  }
  function setValue($value){
    $this->value = $value;
  }
}@#

class Contador{
  #@use_mixin ValueHolder@#
  function increment(){
    $this->setValue($this->getValue()+1);
  }
}

class Direccion extends PersistentObject{
  #@use_mixin ValueHolder@#
  function initialize(){
    $this->addField(new TextField(array('fieldName'=>'value')));
  }
}
```

De esta manera podemos implementar una sola vez el comportamiento de `ValueHolder` y utilizarlo en 2 clases distintas (`Direccion` y `Contador`), sin necesidad de relacionarlos por herencia.

Además, agregamos a los mixins en el chequeo de tipos. Si tenemos un
chequeo `#@typecheck $v: ValueHolder@#`, tanto un objeto de clase `Direccion` como uno de clase `Contador` cumplen con la condición.

5.7. Múltiples Configuraciones

Para el desarrollo colaborativo, es útil mantener múltiples configuraciones. O para trabajar en distintos clientes. O para hacer testing y deployment.

Por todo esto, *PHPWebBuilder* mantiene un archivo de configuraciones múltiples para adaptar cada una a una necesidad específica. En un archivo de configuración global (config.php) se guardan las variables específicas de cada configuración, además de las variables comunes a todas. Luego, en otro archivo (serverconfig) se pone la configuración que se va a utilizar.

5.8. Compilación

El uso de macros y la inclusión de los muchos archivos de *PHPWebBuilder*, hacen que la carga en cada request pueda ser muy lenta. Por eso implementamos una mini-compilación de los archivos PHP (para que las macros ya estén procesadas) y además habilitamos varios outputs de esta compilación del código (todo a un sólo archivo PHP, sólo las clases utilizadas a un archivo PHP, en archivos separados). Esta configuración es seteable desde los archivos de configuración.

6. Trabajos Relacionados

- Muchos frameworks toman el enfoque de programar el controller mediante "páginas"[7, 8]. *PHPWebBuilder* tuvo en un principio esta forma, pero finalmente elegimos componentes [9, 10], dado que consideramos que incrementa la composicionalidad.
- Para la generación de las vistas, encontramos 2 enfoques:
  - *HTML Programático*: Cada componente tiene un método propio para renderearse.
  - *Templates Programáticos*: Son templates que alternan código ejecutable y HTML, y para asociarse a un componente son ejecutados.

  En lugar de los métodos anteriores, decidimos utlizar "Templates Declarativos": Templates que definen lo que se va a mostrar, pero no se ejecutan. Esto permite, por ejemplo, que una colección de elementos sea modificada durante la ejecución, y de esta manera solamente renderear la parte modificada (cuando de la otra forma deberíamos haber redibujado todo, por la necesidad de volver a ejecutar el template completo).
- Para el modelo, utilizamos el mecanismo del pattern ActiveRecord [11], usado también por varios sistemas de persistencia [7, 8, 12, 13]. Dentro de lo que es este pattern, elegimos:
  - mapeo Class-Table [14], donde existe una tabla en la Base de Datos por cada clase en el modelo, y
  - adaptación de la Base de Datos a partir de Clases, en lugar de adaptación de Clases a partir de la Base de Datos [7, 8], porque creemos que simplifica la tarea del programador poder editar su modelo de clases en un lenguaje que tenga integrado el concepto de herencia, y porque no tiene el costo de pasar de un lenguaje a otro para hacer el esquema de datos, y el manejo de los mismos.

  Para la recuperación de objetos, existe el enfoque de utilizar SQL directamente, utilizar objetos que definen una consulta a realizar [7, 8], o tener un lenguage para hacer las consultas [12]. Nosotros tenemos un lenguaje de consulta (OQL) que traduce a Objetos que definen la consulta. Este enfoque nos parece mejor dado que abstrae al usuario del mapeo a SQL (los objetos hacen ese trabajo) y el OQL abstrae de la creación de estos objetos. Nombramos al nuestro OQL por el lenguaje OQL (Object Query Language) [15] de la ODMG.

7. Conclusiones

PHPWebBuilder no presenta en sí conceptos novedosos, sino que reúne las opciones que consideramos mejores y los integra e interrelaciona dentro del mismo framework. Esto libera a los desarrolladores de muchas decisiones reiterativas y les permite enfocarse en los problemas específicos de la aplicación a desarrollar.

Los próximos pasos a realizar son: implementar mejoras en el manejo de colecciones en el modelo e implementar persistencia por alcance [16, 17].

Referencias

[1] Phpwebbuilder - http://phpwebbuilder.sourceforge.net.

[2] Mvc - model-view-controller. *http://en.wikipedia.org/wiki/Model-view-controller*.

[3] Widgets. *http://en.wikipedia.org/wiki/GUI_Widget*.

[4] El proyecto xul. *http://www.mozilla.org/projects/xul*.

[5] Bison. *http://www.gnu.org/software/bison*, 1920.

[6] Yet another compiler-compiler. *http://dinosaur.compilertools.net/yacc/index.html*.

[7] Cakephp. *http://www.cakephp.org/*.

[8] Ruby on rails. *http://www.rubyonrails.org/*.

[9] Seaside - a framework for developing sophisticated web applications in smalltalk. *http://seaside.st/*.

[10] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside — a multiple control flow web application framework. In *Proceedings of ESUG Research Track 2004*, pages 231–257, sep 2004.

[11] Active record pattern. *http://www.martinfowler.com/eaaCatalog/activeRecord.html*.

[12] Hibernate. *www.hibernate.org/*.

[13] Glorp. *http://glorp.org/*.

[14] Class table inheritance. *http://www.martinfowler.com/eaaCatalog/classTableInheritance.html*.

[15] Oql language. *http://www.odmg.org*.

[16] Jpox jdo. *http://www.jpox.org/*.

[17] Jdo. *http://java.sun.com/products/jdo/*.