El sistema de plantillas para navegador Yeast

2 authors:

Francisco J. García-Izquierdo
Universidad de La Rioja (Spain)
33 PUBLICATIONS **150** CITATIONS
SEE PROFILE

Víctor Dorado Asensio
1 PUBLICATION **1** CITATION
SEE PROFILE

CONTRIBUCIONES CIENTÍFICAS
EN HONOR DE MIRIAN ANDRÉS GÓMEZ
(Laureano Lambán, Ana Romero y Julio Rubio, editores),
Servicio de Publicaciones, Universidad de La Rioja,
Logroño, Spain, 2010.

EL SISTEMA DE PLANTILLAS PARA NAVEGADOR YEAST

FRANCISCO JOSÉ GARCÍA IZQUIERDO Y VÍCTOR DORADO ASENSIO

Dedicado a la memoria de Mirian Andrés Gómez, de quien siempre admiré su capacidad para sonreír ante cualquier dificultad

Resumen. El presente artículo es un compendio de los logros obtenidos a lo largo de los últimos años por el grupo de investigación PSYCOTRIP de la Universidad de La Rioja en el campo del desarrollo de aplicaciones Web. Más específicamente hablaremos del sistema de plantillas *Yeast*. El artículo analiza las dificultades que habitualmente se encuentran durante el desarrollo de este tipo de aplicaciones, evidenciando sus síntomas y proporcionando unos criterios que deben guiar el desarrollo. Analizamos el problema de la separación entre presentación (o vista en terminología MVC) y contenido (o modelo), y proponemos una solución arquitectónica basada en el empleo de un doble modelo. Esta propuesta separa de forma efectiva ambos aspectos de la aplicación. Para terminar, presentamos Yeast, un sistema de plantillas para navegador, que cumple los mencionados criterios de desarrollo y es conforme al doble modelo. Estudiamos varios aspectos de Yeast: funcionalidad, rendimiento, accesibilidad, seguridad y su relación con AJAX.

Abstract. This article is a compilation of the results obtained by the PSYCOTRIP research group of the University of La Rioja during the last years in the Web application development field. More specifically we will deal with the *Yeast* Web template system. The article analyzes the typical difficulties that appear along the development of this kind of applications, showing their symptoms and providing a set of criteria that must guide the development. The problem of separation between presentation (or view in MVC terminology) and content (or model) is analyzed, and an architectural approach based on the use of a double model is proposed. This approach effectively separates both parts of the application. Finally, Yeast is presented as a browser-side templating system that meets the development criteria and is conform to the double model approach. Several Yeast aspects are analyzed: its functionality, performance, accessibility, security and its relation to AJAX.

1. Introducción

El presente artículo es un compendio de los trabajos realizados y logros obtenidos a lo largo de los últimos años por el grupo de investigación PSYCOTRIP de la Universidad de La Rioja en el campo del desarrollo de aplicaciones Web.

2000 *Mathematics Subject Classification.* 68M11, 68U35.

Key words and phrases. Template systems, Web applications, software architectures, MVC, browser-side templating.

La redacción de este artículo ha sido parcialmente subvencionada por la Comunidad Autónoma de La Rioja, proyecto FOMENTA-2008/01.

Más concretamente nos referiremos al sistema de plantillas web *Yeast*. Algunos de los resultados presentados en este artículo ya han sido publicados [38, 30] y otros están o estarán disponibles en breve de forma on-line, en el sitio web de Yeast `www.yeasttemplate.org`.

La contribución de Mirian al sistema de plantillas Yeast tuvo lugar en 2003, por medio de una comunicación en la International Conference WWW-Internet 2003 [38]. Por aquel entonces yo era su director de tesis y ni ella ni yo teníamos claro cómo íbamos a enfocar la carrera investigadora de Mirian. Estábamos buscando un tema de investigación en el que se pudiera desarrollar una tesis. Con excelente voluntad y cargada de valor, decidió afrontar el reto de participar en algo que, dada su formación como matemática, le era desconocido.

Desde entonces Yeast ha cambiado mucho; ha cambiado incluso de nombre, ya que la denominación original de nuestro sistema de plantillas era JST, acrónimo de *JavaScript Templates*. A lo largo de estos años Yeast ha ido tomando forma y cargándose de funcionalidad convirtiéndose en una herramienta útil que ha sido utilizada en la construcción de varias herramientas de gestión en el Departamento de Matemáticas y Computación de la Universidad de La Rioja, así como en varios proyectos fin de carrera de titulaciones adscritas al mismo. El actual, 2009, es el año de su lanzamiento público, y es nuestra esperanza que Yeast sea aceptado por la comunidad de desarrolladores Web.

1.1. Los orígenes. Podríamos decir que la concepción de Yeast comenzó allá por 2001, cuando trabajaba para una compañía privada de desarrollo de software. Entre otras aplicaciones, desarrollábamos portales Web empleando XML/XSLT como tecnología base. Es decir el contenido era generado por la aplicación en formato neutro XML (eXtensible Markup Language) y transformado a HTML mediante hojas de estilo XSLT (eXtensible Stylesheet Language Transformations [79]). El HTML así obtenido era enviado al navegador del cliente.

La idea era buena, en la línea de la deseada separación de intereses (*separation of concerns* [54]). Hay que considerar que el proceso de desarrollo de una aplicación Web involucra dos tareas importantes. Por una parte es necesario desarrollar la lógica de aplicación, la que lleva a cabo los procesos de negocio que le dan sentido y personalidad. Esta lógica incluirá muy habitualmente la gestión de datos desde algún repositorio persistente y producirá resultados, *contenido*, que deberá ser mostrado al usuario. Por otra parte, para que la aplicación tenga éxito, y aunque pese a muchos programadores, es necesaria una buena *presentación*: un interface atractivo y bien diseñado, que facilite la interacción del usuario. Ambas tareas requieren de habilidades completamente diferentes y es lógico que sean desarrolladas por equipos diferentes, incluso pertenecientes a distintas compañías. Sería muy deseable que ambos equipos pudieran trabajar de forma independiente. Es por esto muy recomendable el empleo de tecnologías y patrones de diseño como MVC [62, 42] que propicien la separación de contenido y presentación.

En nuestro caso, el uso de XSLT nos prometía un completo desacoplo entre estilo de presentación y contenido. Contábamos con equipos diferentes, un sub-equipo de programadores y otro de diseñadores Web especializados, encontrándose estos

últimos subcontratados. Cada sub-equipo trabajaba en una parte distinta de la aplicación, sobre distintos ficheros fuente, de forma que el trabajo de unos no era importunado por el de los otros. Sin embargo la experiencia no acababa de confirmar los resultados teóricos esperados. El uso de XML generaba otra serie de fuentes de acoplamiento entre los sub-equipos que eran incluso más perjudiciales que los que se pretendían evitar. El proceso de desarrollo se complicaba grandemente y se hacía, permítaseme el término, in-usable.

La recuperación de la usabilidad en el proceso de desarrollo fue precisamente el objetivo que promovió la creación inicial del sistema de plantillas Yeast (entonces JST). Partiendo de un sistema de desarrollo basado en plantillas, como lo era el empleo de XML/XSLT, analizamos los problemas de acoplamiento encontrados y sus causas, y enunciamos unas recomendaciones que debían ser cumplidas por cualquier sistema de plantillas "usable".

1.2. Estructura del artículo. Por tanto, antes de describir Yeast, debemos comentar las ideas que fundamentan su filosofía. A lo largo de las primeras secciones del artículo analizaremos el hecho paradójico de que no todas las tecnologías que prometen separación de presentación y contenido, lo consiguen realmente, ni en la separación de equipos, ni tan siquiera en los aspectos técnicos. La sección 2 se centrará en el análisis de los comentados problemas de usabilidad y, dejaremos para la sección 3 el análisis profundo de la separación entre contenido y presentación. Tras ello, en la sección 4, describiremos la propuesta de doble modelo y su arquitectura asociada. Eso nos conducirá a Yeast, un sistema de plantillas Web que facilita la separación efectiva entre el contenido de la presentación de una aplicación Web y que propicia un proceso de desarrollo usable. Para terminar las secciones de trabajo relacionado, trabajo futuro y conclusiones.

2. La usabilidad en el proceso de desarrollo de aplicaciones Web

Si buscamos la definición de "usabilidad" en Wikipedia, en castellano encontramos "capacidad de uso, es decir, la característica que distingue a los objetos diseñados para su utilización de los que no". La acepción inglesa es más amplia y se refiere a "la facilidad o nivel de uso, es decir, al grado en el que el diseño de un objeto facilita o dificulta su manejo". Al hablar de usabilidad de una aplicación informática [52] casi siempre se piensa en la mejora de la experiencia del usuario al utilizar su interface de usuario. Sin embargo, nosotros nos apoyaremos más en la acepción inglesa del término al enfocar nuestros esfuerzos en la mejora de la usabilidad del propio proceso de desarrollo de aplicaciones Web.

2.1. Problemas detectados en el desarrollo de aplicaciones Web. Tal como hemos comentado en la introducción, aunque se emplee una tecnología o patrón de diseño que permita la separación entre presentación y contenido, ya sea XML/XSLT, MVC u otros sistemas de plantillas, en muchas ocasiones se presentan problemas e inter-dependencias entre los sub-equipos de desarrollo que complican artificialmente el proceso de desarrollo. Entre estos problemas podemos citar los siguientes.

2.1.1. Duplicación del esfuerzo. Para desarrollar una plantilla que sirva como base para una página de la presentación de una aplicación, ya sea mediante XSLT u otra tecnología (JSP, Freemarker...), el diseñador gráfico de HTML suele trabajar dos veces. Es muy complicado concebir y desarrollar el aspecto gráfico de la página y al mismo tiempo tener presentes las peculiaridades del sistema de plantillas elegido. Por ello, habitualmente se prefiere diseñar un prototipo HTML de la página, no funcional, con ayuda de la herramienta de diseño WYSIWYG de su elección y cuidando todos los detalles de estilo. Este prototipo es posteriormente transformado a la versión final de plantilla empleando el lenguaje adecuado (XSLT, Java en el caso de JSP, FTL en el de Freemarker...)

Hacer cambios a posterior en el HTML es complicado. De nuevo se tiene que hacer el trabajo en las dos versiones (prototipo HTML y plantilla) o bien actualizar la plantilla directamente y dejar de tener sincronizadas ambas versiones.

2.1.2. El trabajo del diseñador gráfico se complica. Los diseñadores gráficos están acostumbrados a una herramienta de edición que les permite centrarse en la distribución de cada página (que es el valor que deben aportar a la aplicación). Desde ella componen y tienen un feedback inmediato del resultado. Es la forma más eficaz y productiva de hacer su trabajo. Al usar un sistema de plantillas la plantilla producida precisa del motor de plantillas para ser visualizada (un procesador XSLT, una máquina Java, un motor Freemarker...). Su entorno de desarrollo se complica con herramientas que les son ajenas. Aunque quizás esto no sea percibido como un problema para los programadores (dado que están acostumbrados a ello) no es algo que entiendan los diseñadores (o lo entienden pero no ven la necesidad de añadir más etapas a su trabajo para conseguir el mismo resultado visual).

Además los diseñadores no podrán probar la plantilla si no tienen un modelo de datos que la "alimente" (un documento XML para la XSLT, un conjunto de objetos Java para JSP o Freemarker...), por lo que, o bien se espera a que el programador le pase uno, o bien incluso los diseñadores tienen que estar creando datos de prueba usando lenguajes y tecnologías que no dominan.

2.1.3. Problemas de formación para los diseñadores. En las plantillas se mezcla HTML con código que aporta el dinamismo característico de estos documentos. Los diseñadores gráficos no suelen tener conocer estos lenguajes, ni siquiera tienen conocimientos de programación necesarios. Deben ser formados en tecnologías complejas en muchos casos orientadas a objetos. Los diseñadores no entienden el porque de esta necesidad, más aún cuando ellos ya han hecho su trabajo de diseño gráfico usando HTML.

2.1.4. ¿Es suficiente la separación entre contenido y presentación? Muchos sistemas de plantillas prometen conseguir esta separación pero muy pocos la consiguen a un nivel aceptable. Este es un aspecto central en nuestra discusión, al que dedicaremos una sección completa más adelante (véase la sección 3).

2.1.5. Mayor dependencia entre programador y diseñador. Ya hemos dicho que, para ser probadas, las plantillas necesitan de un modelo de datos. De hecho las plantillas dependen de la estructura de ese modelo de datos, ya sea XML o un

modelo de clases. Este modelo de datos es creado por los programadores que deben explicárselo a los diseñadores gráficos. Es posible que el modelo de datos definitivo no esté definido cuando los diseñadores gráficos comienzan su trabajo, Pero estos necesitan uno, lo que obliga a los programadores a realizar decisiones de diseño tempranas e inmaduras, que dan lugar a diseños que van a cambiar. Un cambio en el modelo de datos es fatal, tanto para los programadores que tienen que volver a explicar el nuevo modelo a los diseñadores, como para los propios diseñadores que tiene que volver a modificar las plantillas que ya habían creado.

Tampoco hay independencia del programador respecto de la presentación. Es habitual que el diseñador la pida al programador que modifique el modelo de datos o incluya en él ciertos valores que simplifiquen el desarrollo de la plantillas.

2.1.6. ¿Por qué no hacen el trabajo los programadores? En realidad esto es lo que sucede en muchas ocasiones. Los diseñadores se limitan a proporcionar el prototipo inicial HTML de la presentación y son los programadores los que lo transforman en plantillas. En nuestra opinión esto es un grave error. Se está empleando mano de obra cualificada en tareas para las que no suelen estar formados, y que habitualmente rechazas por tediosas, con el consecuente riesgo de estropear el diseño gráfico.

2.2. Requisitos para un desarrollo usable de aplicaciones Web. A la vista de los inconvenientes encontrados, y antes de evaluar otros métodos, se propuso plantear los requisitos indispensables que deberá cumplir un proceso de desarrollo para así tener criterios de decisión.

Requisito 1. La información debe ir del diseñador al programador. Como hemos visto lo habitual es lo contrario. El programador explica al diseñador el modelo de datos de dónde tiene que sacar los datos. El flujo de trabajo debería ser el inverso: el diseñador debería decir qué datos necesita para su página y el programador proporcionárselos. Al diseñador no le importa cómo se han modelado los datos y por tanto no le afectarán los cambios de éstos. Sin embargo al programador, que es el que conoce el modelo, le resultará inmediato saber de dónde obtener los datos a visualizar.

Requisito 2. Minimizar la comunicación entre diseñadores y programadores. De hecho el requisito 1 ya avanza en esta línea. Sin embargo, la independencia del programador también debe ser considerada. Ningún cambio en la presentación debe suponer cambios en el modelo de datos de la aplicación, salvo en contadas ocasiones, p.ej. la adición de nuevas características. Así, la necesidad de contacto entre diseñadores y programadores se reducirá hasta casi desaparecer. Si, a pesar de este grado de aislamiento, la comunicación es todavía necesaria, debe ser lo más simple posible.

Requisito 3. El diseñador dispondrá de un entorno de desarrollo sin imposiciones por la tecnología. La tarea del diseñador es crear la estética del portal. El proceso será usable para él si le permite concentrarse en esa tarea, sin

necesidad de usar herramientas adicionales. Solo las habituales herramientas de edición de gráficos y de HTML. Su trabajo debe poder ser visualizado directamente en un navegador Web sin emplear ningún preprocesador de plantillas. Este matiz es importante: el procesador de plantillas solamente se debería usar en tiempo de producción, cuando la aplicación esté en marcha. Por otro lado, el entorno de desarrollo debe proporcionar mecanismos a los diseñadores para diseñar, introducir y eliminar los datos de prueba de la parte dinámica de la plantilla con el mínimo esfuerzo.

Requisito 4. Evitar nuevos lenguajes de script propietarios. El diseñador habitualmente sabe HTML, JavaScript y posiblemente Flash. Hay que intentar que no tenga que aprender nuevos lenguajes. De otra forma habría que dedicar tiempo a su formación y requeriría usar herramientas fuera de su ámbito.

3. Sistemas de plantillas para la separación de contenido y presentación en aplicaciones Web

Uno de los hitos en el desarrollo de aplicaciones Web ha sido la aparición de frameworks que facilitan las tareas de programación propias de este tipo de sistemas. Véanse, por ejemplo Struts [4], Spring MVC [67] o Ruby on Rails [1]. Muchos de ellos emplean plantillas en el marco de una arquitectura *Modelo Vista Controlador* (MVC) Web [48], o de tipo *Model 2* en terminología de Sun Microsystems [64]. En una aplicación que sigue este patrón es la parte "V" de la triada, la vista, la que alberga las plantillas, que se alimentan de datos provenientes de la "M", el modelo. Las interacciones entre la V y la M son reguladas por la "C", el controlador.

De esta manera, los sistemas de plantillas se han constituido en un estándar de facto para el desarrollo de aplicaciones Web que devuelven contenido dinámico [26]. En lugar de desarrollar para cada servicio un programa específico que confeccione y envíe directamente hacia el navegador el código HTML, los desarrolladores utilizan sistemas de plantillas para conseguir que sus diseños, inherentemente estáticos, dispongan de mecanismos para procesarse dinámicamente. La comunidad de desarrolladores ha creado multitud de sistemas de plantillas. De hecho, mientras escribimos este artículo, una simple búsqueda del término "template engine" en `sourceforge.net` arroja 304 resultados.

El empleo de plantillas simplifica sobremanera la codificación de largos y tediosos programas ad-hoc y, a priori, se promueve la deseada separación entre presentación y contenido, o, respectivamente, vista y modelo en terminología MVC. Las pretendidas bondades de los sistemas de plantillas se derivan directamente de las ventajas de esta separación [56], ventajas que incluyen el confinamiento del aspecto visual de la aplicación en las plantillas, la división del trabajo entre diseñadores gráficos y programadores, reutilización, facilidad de mantenimiento y evolución de la presentación y la posibilidad de adaptar la presentación al tipo de usuario simplemente cambiando de juego de plantillas empleadas en tiempo de ejecución.

En el contexto de las aplicaciones Web, podríamos definir una plantilla como un documento HTML en el que se han insertado marcas especiales que albergarán datos. En tiempo de ejecución, el motor de procesamiento reemplaza las marcas por el contenido dinámico real que toma de la aplicación. La definición anterior muestra los dos elementos que integran un sistema de plantillas. Por un lado un *lenguaje* para el desarrollo de los documentos plantilla, y, por otro, un *motor de procesamiento*. La eterna promesa de los sistemas de plantillas es que los diseñadores gráficos responsables de la creación de las plantillas y los programadores de la aplicación pueden trabajar de forma completamente separada. Ambas partes se encuentran sólo al final del desarrollo, siendo la integración de ambos extremos limpia y sin coste. Pero en la práctica, como ya hemos discutido en la sección 1, esta afirmación no es cierta para muchos de los sistemas.

La forma en la que son insertadas las marcas para datos en las plantillas y cómo éstas son procesadas ha ido evolucionando. De hecho, los primeros sistemas de plantillas que podemos citar (JSP [70], ASP, PHP [58]) permiten la inclusión directa de código en la plantilla (*Model-1*, en terminología de Sun [64]). Este código está programado en el mismo lenguaje que el resto de la aplicación, de modo que puede acceder sin limitaciones a la misma en busca de contenido dinámico. El resultado es un único documento en el que se mezcla código HTML con código nativo, siendo necesaria una gran interacción entre los equipos de diseñadores y de programadores. Estos problemas de acoplamiento, que han sido tratados por varios autores, desde la clásica referencia de Hunter sobre JSP [37] al brillante trabajo de Parr [56], pueden aliviarse con una buena disciplina de programación que aleje la lógica de negocio de la plantilla. Sin embargo, esta medida no suele ser suficiente, porque la posibilidad de inclusión de código en la plantilla es una puerta trasera que los programadores suelen usar cuando los plazos de entrega suelen estar cerca. Estos sistemas facilitan desarrollos rápidos, pero difíciles de mantener (cualquiera que haya una aplicación de tamaño medio o grande en la que se usen JSP de esta manera estará de acuerdo). Esta forma de proceder contraviene todos los principios de la separación entre vista y modelo.

Un segundo paso en la evolución de los sistemas de plantillas nos lleva a aquellos que no permiten la inclusión de código nativo. Este factor elimina la posibilidad de acceso directo a la lógica de negocio desde la presentación, reduciendo el acoplamiento, pero no eliminándolo del todo. Dependiendo del sistema, se pueden conseguir varios grados de separación. Existen sistemas como StringTemplate [55, 56] que proporcionan una separación radical, pero con el coste de limitar la potencia del lenguaje; otros, como Velocity [6], que presumen de separar, pero a la postre sólo aconsejan disciplina contra el acoplamiento; y otros para los que la separación es sólo una cuestión de marketing. Sistemas como XSLT [79], FreeMarker [27], WebMacro o Velocity [82, 6] o Smarty [65] obligan a los diseñadores a aprender pseudo-lenguajes específicos de programación, así como a usar herramientas propias para el desarrollo de las plantillas. Otros, como Tapestry [5] o Tea [81], obligan al desarrollo de clases especiales o adaptadores. Se trata de sistemas todos ellos que no cumplen los requisitos de usabilidad descritos en la sección 2.2, y que complican artificialmente el proceso de desarrollo para los diseñadores gráficos Tal

como hemos comentado, los diseñadores deberían usar solo HTML. CSS, JavaScript (y quizás ActionScript o lenguajes propios de aplicaciones RIA como Flex [2]). Es un hecho que los diseñadores siguen prefiriendo maquetar sus aplicaciones con estas tradicionales herramientas. Por ello resultan interesantes propuestas de lenguajes de plantillas basados en atributos tales como Zope Page Templates [66], Tapestry [5], o el propio Yeast que presentamos aquí [29]. En ellos las directivas de procesamiento para el motor son insertadas en el documento HTML en forma de atributos de las propias etiquetas HTML. Esta característica conlleva una de las ayudas más valorables que puede esperar un diseñador gráfico: la "revisuabilidad" (*previewability*), o capacidad de los editores de HTML de mostrar cómo queda el diseño de forma visual sin necesidad de recurrir a herramientas externas.

Finalmente, en los últimos años, se han desarrollado sistemas de plantillas para navegador (*browser-side template systems*), dentro de los cuales puede situarse Yeast. En ellos el motor de procesamiento es desplazado hasta el propio navegador Web, normalmente codificado en JavaScript. Su aparición guarda relación con la popularización de la tecnología AJAX [31], de tal modo que algunos lo catalogan como un patrón AJAX [46]. El coste de desarrollo de aplicaciones AJAX suele ser alto ya que hay que recurrir al empleo de la criptica API DOM (Document Object Model, [77]) proporcionada por JavaScript. Los sistemas browser-side templating tratan de facilitar las manipulaciones AJAX, proporcionando un lenguaje de definición de plantillas para la generación dinámica de HTML. Se podría decir que Xforms [57, 15] es el primer precursor de plantilla browser-side, aunque su funcionalidad es muy reducida y su empleo está especialmente destinado al envío de formularios. Otros ejemplos de estos sistemas son JavaScript Templates [76] (no confundir con nuestro JST[1]), Authenteo [86], Jemplate [43], prototype.js [60] o QueryTemplates [61]. Sin embargo Yeast no surgió en relación con AJAX. En 2001, fecha de arranque del proyecto JST, precursor de Yeast, no se oía hablar de esa tecnología. Incluso, y hasta donde nosotros sabemos, Yeast puede considerarse el primer ejemplo de este tipo de sistemas.

3.1. Análisis de la separación entre vista y modelo. La realidad es que, a pesar de los esfuerzos, el acoplamiento entre presentación y lógica de negocio, o vista y modelo en terminología MVC, aún persiste. Pocos trabajos en la literatura se han dedicado a analizar las causas y el sentido de este acoplamiento para atacarlo de raíz. Una buena referencia introductoria puede ser [24], en la que se estudian varios sistemas de plantillas.

La primera referencia conocida que realiza un estudio formal sobre el tema es [56], a cargo de T. Parr, trabajo en el que se definen los términos de plantilla,

[1]Como ya hemos comentado, el desarrollo de Yeast comenzó allá por el año 2001, datando de 2003 las primeras publicaciones en congresos internacionales [38]. Entonces llamábamos a nuestro sistema de plantillas JST, JavaScript Templates. En 2007, durante la redacción de otra publicación relacionada [30], tuvimos conocimiento de otro sistema de plantillas para navegador, que resultó llamarse de la misma manera. Las primeras referencias sobre este segundo JST datan de 2005, de modo que podemos afirmar que nos copiaron el nombre. En 2009 hemos decidido hacer públicamente accesible nuestro software, y hemos buscado un nombre más "comercial" y distintivo. De ahí el cambio de nombre a Yeast Templates.

separación estricta entre vista y modelo, y se proporciona una clasificación de diferentes tipos de plantillas. Así mismo se formulan las cinco reglas que cualquier sistema de plantillas debe cumplir para forzar la separación estricta entre vista y modelo, y se da un *índice de "enmarañamiento"* que puede utilizarse al comparar sistemas de plantillas. Para resumir el trabajo de Parr podemos enunciar sus reglas: (1) "la vista no puede modificar el modelo"; (2) "la vista no puede realizar cómputos con datos cuyo valor es variable"; (3) "la vista no puede comparar datos variables"; (4) "la vista no puede hacer suposiciones sobre los tipos de los datos"; y (5) "los datos provenientes del modelo no pueden contener información relativa a su presentación o estructura visual". El índice de enmarañamiento de un sistema de plantillas se define como el número de reglas de separación que puede violar el sistema, siendo 1 el mínimo valor, ya que es imposible evitar que se viole la quinta regla. Parr previene contra los sistemas de plantillas que alientan la separación, y dejan la responsabilidad de preservarla en manos de bienintencionados desarrolladores. La esencia de su trabajo consiste en dar las pautas para forzar, en vez de alentar, la separación, prohibiendo, por ejemplo, que las vistas realicen cálculos de ningún tipo con los datos del modelo. Así, una vista no debería poder calcular el precio de venta de un libro como "`$price*0.9`" (regla 2), o comparar el precio con un cierto límite como en "`$price<25`" (regla 3), o acceder a una componente de un array de nombres con el valor de otra variable, que se supone de tipo entero, como en "`$names[$id]`" (regla 4).

A pesar de que las reglas de Parr están bien enunciadas, nuestra opinión es que son demasiado restrictivas y su completa aplicación penalizaría la funcionalidad de los sistemas que lograran cumplirlas, dificultando el trabajo de los diseñadores gráficos. La renuncia a esta funcionalidad podría ser un bajo precio a pagar si la consecución de la separación entre vista y modelo fuese posible, pero otro factor hace que el objetivo pretendido por las reglas enunciada por Parr sea imposible de alcanzar. En nuestra opinión algunas de estas reglas no pueden imponerse en sistemas de plantillas que devuelven HTML como salida. Parr no tiene en cuenta la posibilidad de usar JavaScript en los documentos HTML. Es imposible evitar que, usando el lenguaje de la plantilla, se inserten referencias a valores en medio de una expresión JavaScript, con lo que es imposible evitar la realización de cálculos o comparaciones. Así por ejemplo:

```
<html>
 <body>
  The final book price is <script>document.write($precio$*0.9)
  </script> &euro;
 </body>
</html>
```

Este código es una pequeña muestra de una plantilla de tipo StringTemplate [55], que una vez procesada, y suponiendo que el valor de precio sea 30, tiene como resultado el siguiente bloque HTML:

```
<html>
 <body>
  The final book price is <script>document.write(30*0.9)
  </script> &euro;
 </body>
</html>
```

Este hecho causa la violación de las reglas 2 y 3 de separación, y hace que, en la práctica, el mínimo valor del índice de enmarañamiento sea 3. Ello nos lleva a afirmar que es imposible forzar la separación estricta entre vista y modelo en sistemas de plantillas para la producción de HTML. La próxima sección la dedicaremos a estudiar nuestra propuesta de análisis y tratamiento del problema de la separación, con el que conseguiremos un índice de acoplamiento de 3 como máximo.

Otro trabajo que debemos incluir en esta sección es [41], donde se clasifican los tipos de acoplamiento en *intra-crosscuting* e *inter-crosscutting*, siendo ambos ortogonales. El primero se refiere al código que aparece mezclado con la presentación, al estilo de los JSP, que se materializa en un fuerte acoplamiento de la lógica y la presentación de la aplicación, lo que redunda en una fuerte dependencia entre los grupos de programadores y diseñadores. Éste es el tipo de acoplamiento analizado también por Parr en [56]. El segundo tipo es más sutil y provoca el desperdigamiento de la estructura de recursos de la aplicación a lo largo del código y la presentación, que se vuelven dependientes de esa estructura. Afecta a la mayoría de las páginas, tanto dinámicas como estáticas. Ejemplos de este tipo de acoplamiento pueden ser las URLs que redirigen entre los servicios de la aplicación, los nombres de los campos de una tabla en la BD, los nombres de los parámetros de un formulario, etc. Piénsese, por ejemplo, en que un simple cambio de nombre en un campo de un formulario no queda confinado en la plantilla del formulario sino que afecta al código de servidor que ha de procesarlo y es posible que al servicio que debe procesar la plantilla que muestra el formulario. Además de hacer esta clasificación, el trabajo propone formas de aliviar ambos tipos de acoplamiento, mediante el uso de MVC, para reducir (no eliminar) el intra-crosscutting, y una técnica denominada *puntos de extensión*, para reducir el inter-crosscutting. En ambos casos las soluciones son parciales y poco efectivas.

Como conclusión podemos afirmar que las soluciones aportadas en la literatura al problema del acoplamiento de vista y modelo son, o muy restrictivas [56], o parciales [41]. Esto nos llevó a proponer en [30] otra forma de abordar la separación entre vista y modelo, una solución arquitectónica que se basa en una modificación del patrón MVC: la propuesta de doble modelo. Nuestra solución garantiza un índice de acoplamiento de 3 cómo máximo (reducción del intra-crosscutting) y puede utilizarse como soporte para reducir el inter-crosscutting.

4. La propuesta de doble modelo

Tal como hemos comentado, la mayoría de los frameworks de desarrollo de aplicaciones Web utilizan algún sistema de plantillas como componente de una

arquitectura más general basada en el patrón MVC. De esta manera el patrón MVC se ha convertido en el estándar de facto para el desarrollo de aplicaciones Web.

Originalmente MVC se concibió como un patrón para la construcción de aplicaciones de escritorio completas en Smalltalk [62, 42], habilitando un diseño modular y permitiendo la realización de pruebas por separado del interface de usuario. Surgió en un entorno tecnológico en el que no se habían producido grandes avances en esta materia (la mayoría de los interfaces eran textuales). Hoy, y especialmente en relación con la Web, se utiliza sobre todo en la capa de presentación. El patrón define claramente una triada de elementos y sus relaciones, pero también deja, quizás deliberadamente, libertad cuando se trata de implementarlo[2]. El hecho es que hay numerosas variantes del patrón: Model View Presenter (MVP), del cual hay tres versiones: la de Taligent [59], la Dolphin [12] y la de Microsoft [22]; Presentation Model [25] (o M-V-VM para Microsoft [33]); MVC-Renderer [56],... Otra referencia que analiza diferentes implementaciones del patrón MVC en la Web es [48].

El uso de MVC no es garantía para el desacoplo de vista y modelo. De hecho las descripciones del patrón indican todo lo contrario ya que la vista debe conocer el interface del modelo y acceder a él para dibujarse. Una de las variantes, MVP (quizás la versión de Microsoft [22] es la más clara en este sentido) evita esta interrelación haciendo que el Presenter se encargue de actualizar el modelo y las vistas. En relación con las plantillas Web, algunos llaman a este esquema MVC modelo *push* (el controlador "empuja" los datos sobre la plantilla) en contraposición al MVC modelo *pull* (las plantillas "tiran" de los datos del modelo) [85]. Los modelos push y pull también han sido estudiados en [56], decantándose el autor por el modelo push, si bien advierte de que no es suficiente para garantizar la separación.

Para formular nuestra propuesta, nos centraremos en las aplicaciones Web, y analizamos el sentido pragmático de la separación entre vista y modelo. Este no es otro que proteger ambos lados de la aplicación de cambios en el otro. Si no se garantiza el aislamiento, los cambios en una de las partes conllevarán modificaciones en la otra. Si se consigue la separación cada parte puede evolucionar aisladamente, incluso ser desarrollada independientemente.

Una vez que ya hemos analizado que no es posible alcanzar la separación estricta según fue propuesta por [56] en sistemas de plantillas HTML, analizaremos otra solución para obtener un grado de aislamiento menor, pero efectivo. Si no podemos aplicar el adjetivo estricto, nos confirmaremos con efectivo. Nuestra propuesta estará asociada a un modo de desarrollo que cumplirá los requisitos enunciados en la sección 2.2.

Denominamos a nuestra propuesta *doble modelo*. Su fundamento es que cada parte de la aplicación tiene su propio modelo. Por un lado, la vista, utiliza su propio y privado modelo que mantiene los datos necesarios requeridos para implementar la funcionalidad de la página. La vista está, por lo tanto, completamente vinculada

[2]Consúltese, por ejemplo, el debate sobre el sentido y responsabilidades del controlador (`http://c2.com/cgi/wiki?WhatsaControllerAnyway`).

a su modelo. De hecho, la vista sólo puede usar su modelo. No se permite ninguna referencia a la lógica de la aplicación, ni siquiera accesos de sólo lectura. Este modelo se forma de un conjunto de pares nombre-valor, pudiendo ser estos últimos simples o estructurados, y está poblado con una serie de *valores de prueba* que permitirán comprobar el funcionamiento de la vista y afinar su aspecto visual. Llamamos a este modelo, *modelo del diseñador*. Es creado por el diseñador gráfico, quien se basa en los requisitos funcionales de la vista. Una vez que todos los requisitos han sido atendidos, el modelo puede considerarse estable y no necesita ser cambiado. Por ejemplo: si la vista muestra una lista de libros de una supuesta cesta de la compra, el modelo definirá estructuras de datos que mantengan los datos de los libros comprados, así como, quizás, el nombre del comprador o la fecha de la compra.

Es importante hacer notar que el modelo del diseñador no debe ser modificado necesariamente cuando la apariencia visual de la vista cambie, a menos que algún requisito funcional sea añadido y necesite de nuevos datos para implementarlo. Decimos que no tiene porque ser cambiado, aunque sí puede ser cambiado (nadie puede evitar que alguien se complique su trabajo). El matiz es que la decisión de si cambiar el modelo o no hacerlo recae en una única persona, la misma que lo creó: el diseñador gráfico. No es un cambio motivado por otro cambio en la aplicación.

El modelo del diseñador constituye el *interface de datos* de la vista. Son los datos que la vista necesita para mostrarse y constituyen un contrato que la vista impone al resto de la aplicación.

Las vistas son desarrolladas de forma separada al otro modelo, al que llamaremos *modelo del programador*, debido a que es desarrollado por los programadores. Este es el clásico modelo de la aplicación, la M en el acrónimo MVC, habitualmente implementado como una serie de clases, que pueden o no ser hechas persistentes en una base de datos. El modelo del programador puede evolucionar como necesite, seguramente sujeto a continuos reajustes durante el desarrollo. Pero esta evolución no afectará al anterior modelo del diseñador, que sigue aislado en la vista.

Para que las vistas puedan mostrar información real en tiempo de ejecución ambos modelos deben ser integrados. Esta tarea es realizada por el programador, que es el único que conoce el verdadero modelo de la aplicación y sabe como extraer datos de él. Esta integración consiste en suministrar a la vista datos procedentes del modelo del programador pero *re-estructurados* de la forma que impone el modelo del diseñador. De este modo se hace cumplir el contrato del interface de datos de la vista. Es importante recalcar que la vista debe ser pasiva respecto de este proceso de alimentación de datos, es decir algo en la aplicación debe enviar los datos a la vista (el modelo push es un requisito de nuestra propuesta). Este "algo" es el controlador. En la figura 1 se muestra un esquema de los componentes e interrelaciones del doble modelo en comparación con el clásico MVC.

A estas alturas, el lector versado podría pensar que la mayoría de los sistemas de plantillas son conformes a la estrategia de doble modelo. Considérese por ejemplo Freemarker [27] (con su DataModel), WebMacro [82] or Velocity [6], Smarty [65] (para PHP), StringTemplates [55] (para Java, Python y C#) o HTML::Template [75] (para Perl o Lisp). Todos ellos son sistemas de plantillas con atributos que son

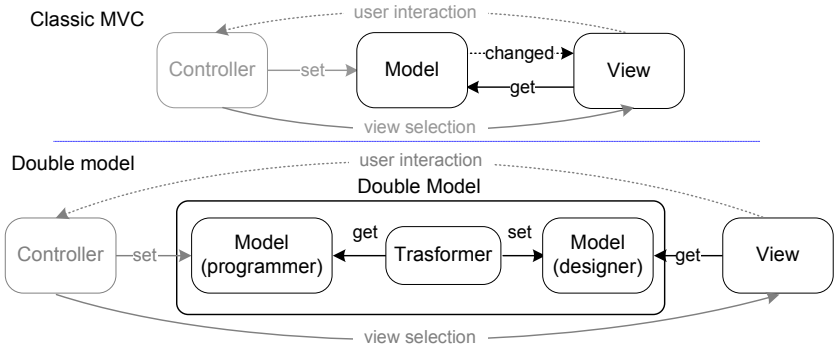


FIGURA 1. Comparación de MVC y la propuesta de doble modelo.

cambiados con valores reales usando un modelo push desde el modelo de la aplicación. La estrategia push no es suficiente. Por tanto, ¿cuál es el factor clave que nos hace afirmar que estamos o no ante un doble modelo? La respuesta a esta pregunta se encuentra en el párrafo anterior: "*... Esta integración consiste en suministrar a la vista datos procedentes del modelo del programador pero re-estructurados de la forma que impone el modelo del diseñador...*" Esta re-estructuración o transformación entre modelos en obligatoria y no estaremos ante un sistema de doble modelo si este paso puede ser obviado de una forma u otra.

Ninguno de los sistemas de plantillas referidos sigue la propuesta de doble modelo ya que permiten que el hipotético modelo del diseñador adopte estructuras de datos pertenecientes al modelo del programador. Esto es, permiten que el programador sugiera al diseñador cual es la mejor estructura de su modelo para que el proceso de integración de modelos no requiera re-estructuración. Esto permitiría al programador enviar estructuras de datos de su modelo directamente a la vista, la cual debería conocer el interface de los datos del modelo de la aplicación, resultando en una vista dependiente del modelo de la aplicación. Evidentemente estos sistemas de plantillas aconsejan a los desarrolladores que usen modelos completamente diferentes, pero la puerta trasera está abierta, y la presión puede hacer que se usen este tipo de atajos para cumplir los compromisos de entrega. Evidentemente las estructuras de datos en ambos modelos pueden coincidir o ser muy similares (un libro tiene un título y un autor en cualquier lugar), pero estas coincidencias no pueden ser usadas para evitar la transformación, que siempre es obligatoria. Y esto debe entenderse no como una complicación artificial, sino como una vacuna contra el acoplamiento.

La idea del interface de datos de la vista es importante ya que remarca el grado de aislamiento conseguido. Las vistas desarrolladas siguiendo la propuesta de doble modelo podrían ser utilizadas en diferentes aplicaciones con la única condición de que pertenezcan al mismo dominio (las aplicaciones respetan el interface de datos de la vista). Del mismo modo, una determinada aplicación puede cambiar de aspecto visual sin ninguna modificación, siendo el único requisito que las nuevas vistas impongan el mismo interface de datos. El aislamiento conseguido con el

doble modelo es tan grande que la vista puede ser visualizada sin ni siquiera estar conectada al modelo de la aplicación, utilizando en este caso los valores de prueba del modelo del diseñador. Este hecho puede ser utilizado para el prototipado del interface de usuario.

4.1. Requisitos de usabilidad y doble modelo. Es el momento de relacionar nuestra propuesta de doble modelo con los requisitos del proceso de desarrollo usable enunciados en la sección 2.2. Al usar la propuesta de doble modelo, el desarrollo es dirigido por el diseñador, que impone su modelo de datos al programador. Es este último el que debe realizar la transformación de modelos. Es decir se ayuda a cumplir el requisito 1 ("la información debe ir del diseñador al programador").

El segundo requisito ("minimizar la comunicación entre diseñadores y programadores") también se cumple, ya que, los modelos de diseñador y programador son desarrollados por separado, sin contacto entre ambos sub-equipos. El modelo de la vista debe ser lo suficientemente auto-descriptivo como para ser entendido por los programadores sin requerir más explicaciones. Es una buena práctica recomendada que los diseñadores utilicen nombres descriptivos para los datos de su modelo y acompañen aquellos que puedan ser confusos de los necesarios comentarios que clarifiquen su significado.

Los restantes requisitos son específicos para el caso del diseño Web basado en HTML, por lo que su cumplimiento dependerá de que el sistema de plantillas que adopte el doble modelo como fundamento use ese lenguaje. Yeast, el sistema propuesto en este artículo, así lo hace.

4.2. Propuesta de arquitectura de aplicación Web para el doble modelo (MmVCT). La transformación de los datos tomados del modelo del programador para adaptarlos al modelo del diseñador es realizada por un nuevo componente arquitectónico de la aplicación, que se sitúa muy cerca del controlador (la C en MVC). Llamaremos a este nuevo elemento transformador.

La presencia del transformador modifica el flujo típico de MVC (o, de su adaptación a la Web, Model-2 [64]). Nos referiremos a esta nueva arquitectura con el acrónimo MVC+mT. El nuevo flujo de control se muestra en la figura 2. Las peticiones (1) son recibidas por el controlador (C) que modifica el modelo (M) de la aplicación (2). El transformador (T) es activado (3), realizando la carga de la vista (V) a utilizar (4), tomando los datos necesarios del modelo de la aplicación (5), y desencadenando la transformación (6) hacia el modelo del diseñador (m). Entonces, el transformador pone los nuevos datos en la vista (7), que es devuelta al usuario.

La idea es que con el doble modelo la vista dialoga con su modelo (m) y el controlador con el de la aplicación (M). El único que se relaciona con ambos modelos es el transformador. Se cumple por tanto la ley de Demeter [45], o principio del mínimo conocimiento, que redunda en sistemas poco acoplados.

4.3. Requisitos del doble modelo. Como recapitulación, pasamos a enumerar las condiciones a exigir para que un determinado sistema de plantillas sea conforme al doble modelo.

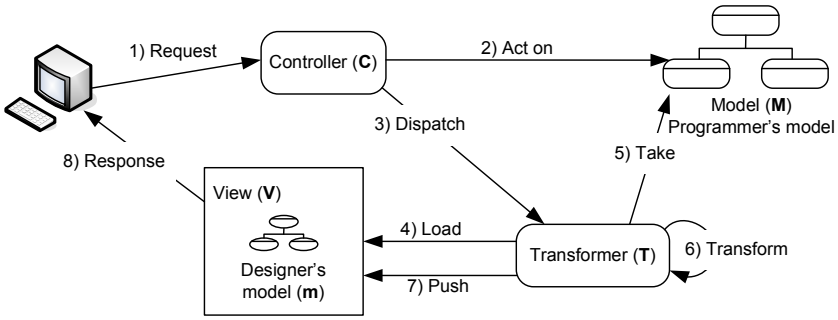


FIGURA 2. Flujo de aplicación en la arquitectura MVC+mT.

1. Ambas partes de la aplicación, modelo y vista, tienen sus respectivos y propios modelos (del diseñador y del programador, respectivamente), que dan soporte a sus respectivos requisitos funcionales.
2. Ambos modelos están expresados en distintos lenguajes, incluso usando distintos paradigmas.
3. La vista no está autorizada a accede al modelo del programador en ninguna circunstancia.
4. El modelo del diseñador está inicializado con valores de prueba que permiten comprobar el funcionamiento de la vista sin necesidad de estar conectada a la aplicación.
5. La aplicación usa la estrategia push para alimentar a la vista con los datos reales.
6. Los datos tomados del modelo del programador no pueden ser enviados a la vista directamente. Deben ser transformados para adaptarlos a la forma impuesta por el modelo del diseñador.

4.4. Doble modelo e índice de enmarañamiento. Es posible relacionar el trabajo de Parr [56] con la propuesta de doble modelo. De hecho podemos afirmar que un sistema de plantillas que sigue la aproximación de doble modelo tiene un índice de enmarañamiento de valor 3 como máximo. Esto es equivalente a decir que cualquier sistema de plantillas conforme al doble modelo cumple las reglas 1 y 4 de Parr. El cumplimiento de la regla 1 se deduce inmediatamente del requisito 3 enunciado en la sección anterior. El cumplimiento de la regla 4 es garantizado por los requisitos 2 y 6, que garantizan que ambas partes de la aplicación (las clásicas vista y modelo) usan diferentes tipos de datos que deben ser adaptados.

4.5. Otras implementaciones conocidas de doble modelo. En este artículo describiremos un sistema de plantillas Web basado en doble modelo que se denomina Yeast [29]. Tal como mostraremos en la siguiente sección Yeast implementa el modelo del diseñador en un bloque JavaScript que contiene la definición de todas las estructuras de datos y atributos de la plantilla.

Pero Yeast no es el único sistema de doble modelo conocido. Las Islas XML de Internet Explorer [49] son casi conformes a la propuesta, salvo en el requisito 2 (ambos modelo se escriben en XML). Las islas XML aparecieron en la versión 5.0 del navegador y sólo pueden usarse en Internet Explorer. Solo conocemos un intento de replicar la funcionalidad en navegadores Mozilla (véase [36]). Las islas XML usan la etiqueta `xml`, no estándar HTML, donde se coloca un elemento XML que contiene el modelo del diseñador. Estos datos expresados en XML y contenidos en esas islas XML puede ser vinculado a elementos HTML de la página.

Algo parecido se puede decir de Flex [2]. Flex es un lenguaje XML para generar declarativamente aplicaciones ActionScript. En los documentos Flex se puede declarar una zona de datos (`XMLList`) que sería el lugar donde colocar el modelo del diseñador. Los componentes utilizados en la página pueden tomar datos de esa zona de datos (usando el atributo `dataProvider`). La zona de datos se puede cargar de forma dinámica mediante un componente de comunicaciones `mx:HTTPService`, que puede llamar a un servidor que devuelva datos extraídos del modelo del programador.

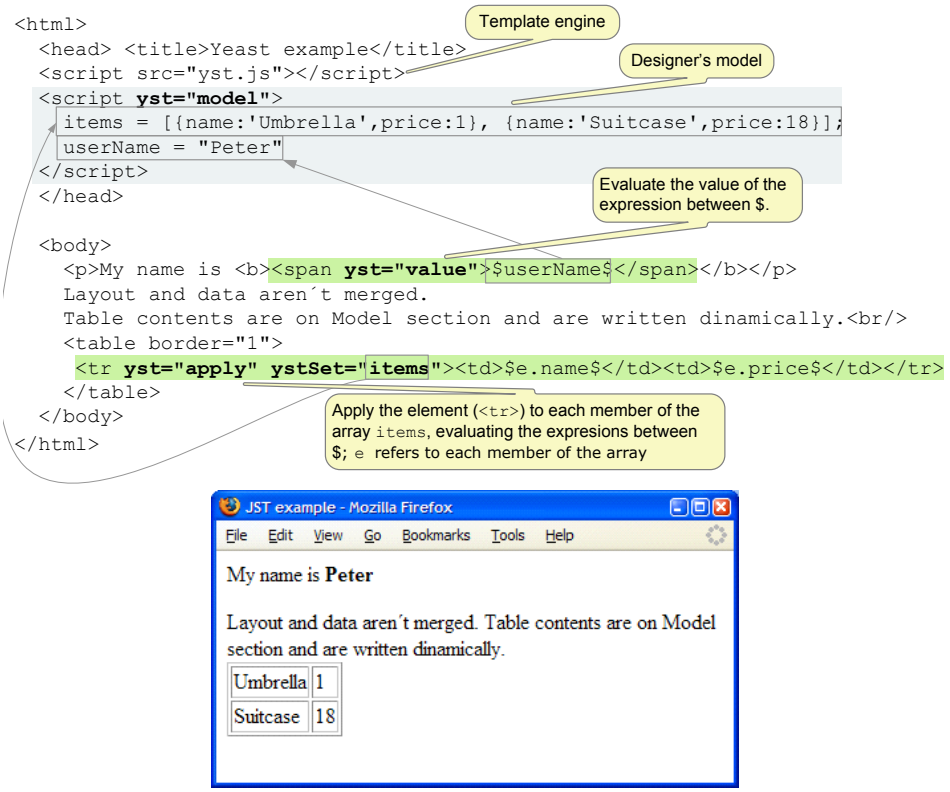
Otro sistema que consideramos conforme al doble modelo es XMLC [21]. En él, el modelo del diseñador está compuesto por una serie de elementos HTML marcados con identificadores (usando el atributo HTML `id`). Estos elementos contienen valores de prueba que ayudan al diseñador a comprobar el aspecto visual de su diseño. Este prototipo de página es compilado para generar una clase Java que permite la manipulación de la estructura DOM de la plantilla, y más precisamente la manipulación de los elementos marcados. El programador puede cambiar el valor de los valores de prueba del documento clonando o cambiando la estructura DOM de los elementos marcados.

No podemos olvidarnos de los sistemas de plantillas para navegador (browser-side templating), que ya hemos mencionado em la sección 3. Lo verdaderamente importante de estos sistemas es que pueden usarse para implementar nuestra propuesta de doble modelo. Si bien no todos los sistemas citados anteriormente son previewables, como lo es Yeast, sí que mantienen una versión de los datos de la aplicación en el navegador y otra en la aplicación.

Por último un par de sistemas más, Mixer (para Java) [83] y FastTemplate (para PHP) [3], pueden considerarse casi conformes al doble modelo, ya que cumplen los requisitos de la propuesta salvo asignar valores de prueba a los atributos de la plantilla. Su modelo de diseñador está compuesto por variables simples, cuyos valores reales son colocados por un módulo de servidor tomando valores del modelo del programador.

5. Yeast

Yeast es un sistema de plantillas HTML basado en doble modelo, licenciado bajo LGPL (`http://www.gnu.org/licenses/lgpl.html`). Yeast pertenece a la categoría de sistemas de plantillas de navegador (browser-side templating), es decir plantillas que son procesadas en el navegador Web, en vez de en el servidor Web, que es lo más habitual.




```
<html>
  <head> <title>Yeast example</title>
  <script src="yst.js"></script>
  <script yst="model">
    items = [{name:'Umbrella',price:1}, {name:'Suitcase',price:18}];
    userName = "Peter"
  </script>
  </head>

  <body>
    <p>My name is <b><span yst="value">$userName$</span></b></p>
    Layout and data aren´t merged.
    Table contents are on Model section and are written dinamically.<br/>
    <table border="1">
     <tr yst="apply" ystSet="items"><td>$e.name$</td><td>$e.price$</td></tr>
    </table>
  </body>
</html>
```

FIGURA 3. Ejemplo de plantilla Yeast y cómo es visualizada en un navegador Web.

Lo realmente importante y novedoso de Yeast es su motor de plantillas para navegador. A pesar de ello, se ha realizado un gran esfuerzo para desarrollar un API que pueda ser usado en aplicaciones de servidor J2EE. En la presente sección resumiremos las características más importantes de ambos componentes de Yeast: el motor de plantillas y el API de servidor (*Yeast-Server*).

5.1. Plantillas Yeast. El motor de plantillas. Comencemos por un ejemplo sencillo. Una plantilla Yeast tiene el aspecto que se muestra en la figura 3.

Como en cualquier otro sistema de plantillas, un documento Yeast es un documento HTML con agujeros en los que posteriormente se pueden incluir valores dinámicamente generados. Como se verá en breve, Yeast ayuda a cumplir los requisitos de usabilidad descritos en la sección 2.2 y es conforme a la propuesta de doble modelo, lo que asegura la separación de la vista y del modelo de la aplicación de una forma efectiva.

Una característica esencial de Yeast es el uso de JavaScript [3]. Es usado para definir el modelo del diseñador y también para procesar la plantilla. El motor Yeast, incluido en cada plantilla, está programado en JavaScript (esta es una característica común de los sistemas de plantillas de navegador). Yeast define una serie de atributos HTML no estándares que son añadidos a los elementos de marcado de un documento HTML. Estos atributos son procesados por el motor Yeast para producir el resultado HTML final.

Que se use solamente HTML y JavaScript es importante para los diseñadores gráficos ya que no necesitan formación extra en otros lenguajes de script ni utilizar herramientas más allá de un editor HTML. Para probar sus diseños sólo necesitan abrir sus plantillas con cualquier navegador Web (la figura 3 lo demuestra). Es decir, pueden trabajar de forma autónoma, desconectados del servidor de aplicaciones, lo que permite hacer cumplir los requisitos 3 y 4 descritos en la sección 2.2. Los requisitos 1 y 2 son así mismo respetados, como consecuencia de que Yeast es conforme a la propuesta de doble modelo (ver sección 4.1).

La mayoría de los editores Web respetan, sin eliminarlos, atributos incluidos en los elementos HTML, aunque no sean estándar. Pero, al mismo tiempo, el editor los ignora al previsualizar la página en sus paneles WYSIWYG. Por tanto, el empleo de atributos HTML permite que el diseño de las plantillas se pueda hacer de forma gráfica.

5.1.1. El proceso de trabajo con Yeast. Desde el punto de vista de un diseñador Web la creación de una plantilla Yeast supone crear la página HTML concentrándose solamente en el en el diseño gráfico. Así mismo los diseñadores deben identificar las partes dinámicas de la página para conformar el modelo del diseñador (p.ej. en la figura 3, el nombre de usuario y una lista de ítems). Para cada dato dinámico se definirá una variable JavaScript, con un nombre y un valor de prueba. Estas definiciones se realizan en un bloque `script` que debe ser marcado con el atributo no HTML `yst` de valor `model`. Este bloque será denominado en terminología Yeast sección de modelo (*model section*). Las variables pueden ser simples, de cualquier tipo JavaScript (`userName` en el ejemplo) o estructuradas, arrays u objetos (`items` en el ejemplo, un array de objetos). Un aspecto importante es que el diseñador gráfico tiene completa libertad para estructurar su modelo como quiera. Si el diseñador tiene conocimientos de orientación a objetos en JavaScript, podrá usarlos, como en el caso de la figura 3, en la que se emplea la notación object literal notation. Pero existen otras posibilidades, usando funciones constructoras o sin usar objetos, por ejemplo dos arrays de tipos simples, uno para los nombres y otro para los precios. El diseñador crea su modelo y él es el único afectado por ese hecho (el modelo del diseñador sólo se usa en la plantilla). En ciertas ocasiones el diseñador deberá especificar estructuras de datos adicionales, como las funciones

[3]El uso de JavaScript en páginas Web es siempre una fuente de polémica, sobre todo cuando se consideran aspectos de accesibilidad y seguridad. Sin embargo, de acuerdo a las últimas estadísticas (`http://www.w3schools.com/browsers/browsers_stats.asp`) el 95 % de los navegadores tenían activado el uso de JavaScript y según [87] el 96 % de las páginas de unos 6700 sitios Web populares incluyen JavaScript. Así mismo en [34] se dice que el 98 % de los sitios Web más visitados en EEUU usan JavaScript en el cliente.

constructuras de los objetos a emplear. Estas declaraciones se realizarán en otro bloque `script` que debe ser marcado con el atributo HTML `yst` de valor `init` (*sección de inicialización*). Es claro el alineamiento de Yeast con la propuesta de doble modelo.

5.1.2. Componentes fundamentales de Yeast. En una plantilla Yeast, además de la sección de modelo y la de inicialización, podemos encontrar *expresiones* y *atributos Yeast*.

Expresiones Yeast. Siempre que el diseñador desee insertar un valor dinámico en la plantilla lo hará usando el nombre de la variable que aloja el valor, encerrado entre un par de símbolos `$$` (p.ej., `$userName$`). Una referencia a una variable de modelo puede sola o como parte de una expresión JavaScript más compleja (como `$price*0.9$`). Los diseñadores tienen a su disposición toda la potencia de JavaScript sin necesidad de aprender un nuevo lenguaje. Las expresiones Yeast pueden encontrarse en elementos o atributos HTML.

Atributos Yeast. Contienen instrucciones de procesamiento destinadas al motor de plantillas. El atributo más importante es `yst`. Llamaremos a los elementos HTML que incluyen el atributo `yst`, *elementos Yeast*. Su valor indica el tipo de procesamiento que debe sufrir el element HTML que lo contiene, y debe ser uno de los siguientes:

- `value`: todas las expresiones Yeast que aparecen en el elemento, y sus elementos hijos, son evaluadas y su valor sustituirá a la expresión.
- `if`: condicionales. El elemento Yeast será procesado sólo si una condición expresada mediante otro atributo llamado `ystTest` es evaluada a `true`. Si la condición evaluase a `false`, el elemento sería eliminado del documento procesado. Todas las expresiones Yeast son evaluadas como en value.
- `apply`: iteraciones. El elemento Yeast debe ser repetido una vez para cada uno de los valores de un array especificado en un atributo auxiliar denominado `ystSet`, o debe aparece tantas veces como se especifique en el atributo `ystUpTo` . Hay tres variables implícitas asociadas al proceso iterativo: `i`, el índice de iteración, `e` el valor iterado, y `values`, el conjunto de valores sobre el que se itera. Es posible usar estas variables en las expresiones Yeast contenidas en el elemento.
- `compApply`: iteraciones condicionadas o complejas. Es como `apply`, pero para un grupo de elementos Yeast relacionados (`apply` se aplica a un único elemento). Se apoya en dos atributos auxiliares ya comentados, `ystSet` (o `ystUpTo`) y `ystTest`. Para cada valor de iteración se evalúan todos los elementos relacionados del `compApply`. Sólo aquellos para los que el valor de `ystTest` sea true son incluidos en el resultado final.
- `declare`: usado para definir sub-plantillas (véase la sección 5.1.4).
- `include`: usado para incluir una subplantillas en el document final (véase la sección 5.1.4).

- `ignore`: permite ocultar del resultado final determinados elementos que el diseñador gráfico haya podido usar como auxiliares para realizar su diseño. Los elementos no deben aparecer en el resultado final, pero tampoco ser eliminados de la plantilla, porque puede ser útiles al usar el editor HTML.
- `ajax`: usado para marcar elementos que serán reprocesados como resultado de una interacción AJAX o como resultado de alguna acción realizada sobre la página. Pueden consultarse más detalles sobre Yeast y AJAX en la sección 5.3.

Evidentemente los elementos Yeast pueden anidarse, dando lugar a estructuras complejas e incluso recursivas (por ejemplo en la declaración de sub-plantillas).

5.1.3. Ejemplos simples. No pretendemos incluir en el artículo un tutorial completo sobre Yeast. El lector puede consultar [29] donde además encontrará multitud de ejemplos. Incluiremos aquí estas breves muestras de código Yeast como ilustración de las instrucciones de procesamiento previamente especificadas.

Empezaremos suponiendo que el diseñador gráfico ha determinado la siguiente sección de modelo para su uso en la plantilla:

```
<script yst="model">
 items = [{name:'Umbrella',price:1\},{name:
          'Suitcase',price:48},{name:'Belt',price:18}];
 userName = "Peter"
 temperature = -2;
</script>
```

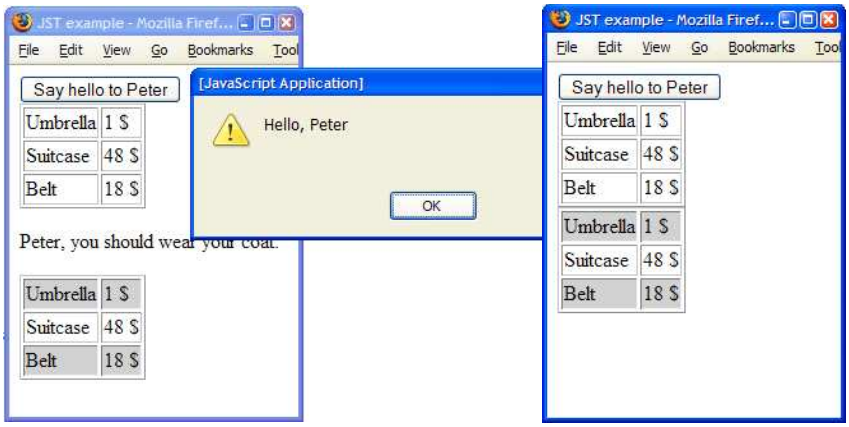
Y a continuación los ejemplos:

```
<button yst="value" onClick="alert('Hello, $userName$')">
  Say hello to $userName$
</button>

<table border="1">
 <tr yst="apply" ystSet="items">
  <td>$e.name$</td><td>$e.price$ \$</td>
 </tr>
</table>

<p yst="if" ystTest="temperature<10">
  $userName$, you should wear your coat.
</p>

<table border="1">
 <tr yst="compApply" ystSet="items" ystTest="i%2==0"
     bgcolor="#CCCCCC">
  <td>$e.name$</td><td>$e.price$ \$</td>
 </tr>
 <tr yst="compApply" ystSet="items" ystTest="i%2!=0">
  <td>$e.name$</td><td>$e.price$ \$</td>
```



a) b)

FIGURA 4. Ejemplos Yeast, a) con valor de temperatura -2; b) con valor de temperatura 25.

```
 </tr>
</table>
```

Una vez procesado el documento resultante contendrá el siguiente HTML (que puede verse representado gráficamente en la figura 4:

```
<button onClick="alert('Hello, Peter')">
  Say hello to Peter
</button>

<table border="1">
 <tr><td>Umbrella</td><td>1 $</td></tr>
 <tr><td>Suitcase</td><td>48 $</td></tr>
 <tr><td>Belt</td><td>18 $</td></tr>
</table>

<p>Peter, you should wear your coat.</p>

<table border="1">
 <tr bgcolor="#CCCCCC"><td>Umbrella</td><td>1 $</td></tr>
 <tr><td>Suitcase</td><td>48 $</td></tr>
 <tr bgcolor="#CCCCCC"><td>Belt</td><td>18 $</td></tr>
</table>
```

5.1.4. Sub-plantillas. Una sub-plantilla es un fragmento de código Yeast (o simplemente HTML) que, una vez declarado, puede ser incluido en la plantilla tantas veces como se necesite.

Para la declaración de una sub-plantilla, debemos marcar un elemento contenedor con el atributo `yst=‘‘declare’’` y asignarle un identificador. Cuando la sub-plantilla es incluida, el elemento contenedor no es tenido en cuenta, así que es una buena idea usar un elemento `<div>` que encapsule el resto del código. Como ejemplo consideremos una lista desplegable HTML, para selección del sexo en un formulario:

```
<div yst="declare" id="ld_sexo">
  <select name="sexo">
    <option value="" selected>Elija una opción</option>
    <option value="H">Hombre</option>
    <option value="M">Mujer</option>
  </select>
</div>
```

Para incluir esta sub-plantilla dentro de una plantilla Yeast, tan solo deberemos colocar el siguiente fragmento de código:

```
<div yst="include" ystIdRef="ld_sexo"></div>
```

Las sub-plantillas pueden contener referencias a parámetros formales, que deben ser suministrados en el momento de su inclusión. Así, la siguiente sub-plantilla constituye un menú desplegable genérico:

```
<div yst="declare" id="combo">
 <select yst="value" name="$params.name$">
  <option value="0">- Choose one -</option>
  <option yst="apply" ystSet="params.opts">$e$</option>
 </select>
</div>
```

La sub-plantilla puede incluirse de la siguiente manera, generando el mismo menú que en el ejemplo anterior:

```
<div yst="include" ystIdRef="combo"
     ystParams="{name:'sexo',opts:['Hombre','Mujer']}">
</div>
```

5.1.5. Cuestiones de eficiencia y compatibilidad con navegadores antiguos. Las dos versiones de motor. El motor de plantillas Yeast accede a los elementos a procesar mediante la implementación del API DOM (Document Object Model, [77]) que ofrece el navegador. Como era de esperar, durante el desarrollo del motor aparecieron muchos problemas de compatibilidad entre navegadores (*crossbrowsing*). Se han hecho muchísimas pruebas, incluyendo controles de formularios, elementos estructurales, estilos, manejadores de eventos. Durante las mismas se han probado más de 190 navegadores y versiones de navegadores, en los tres sistemas operativos más usados: Windows, Mac y Linux. Incluso se han encontrado bugs en la implementación de los navegadores, por ejemplo en Firefox (los basados en Gecko),

Opera e Internet Explorer, que han sido notificados a sus fabricantes[4]. El resultado final es que Yeast puede usarse en: Internet Explorer desde la versión 5.5 (y navegadores similares como AOL Explorer o DeepNet), navegadores basados en el motor Gecko (Firefox, Mozilla, SeaMonkey, K-Meleon y Flock han sido probados), el ya abandonado Netscape (desde la version 6), Opera (desde la version 8), Google Chrome, Safari (para Mac y Windows), Konqueror (para Linux), y casi con seguridad en cualquier otro navegador moderno. Esto cubre aproximadamente el 98 % de los navegadores usados en el mundo, de acuerdo a las últimas estadísticas [5]. La tasa de éxito media de las pruebas en los navegadores testados ha sido del 97 % (98 % si nos centramos en navegadores modernos), siendo un 15.5 % de los mismos totalmente compatible (35,5 % en navegadores modernos). Hemos de considerar que los test ejecutados han sido muy detallistas, y que muchos de los fallos encontrados son de importancia minúscula. En todos los casos se dispone de formas alternativas de conseguir el resultado (los fallos y las formas de solventarlos están especificados en [29]).

Sin embargo, el hecho es que hay algunas incompatibilidades. La situación empeora en algunos navegadores antiguos (por ejemplo las versiones 4.x de Netscape) ya que ni siquiera disponen del API DOM. Además, dependiendo del tamaño de la plantilla y, sobre todo, del fabricante del navegador, la velocidad de proceso de Yeast puede considerarse lenta. En palabras de Peter-Paul Koch de quirksmode.org, la manipulación de la estructura DOM de un documento en Internet Explorer 6.0, uno de los más usados, es unas 30 veces más lenta que la manipulación directa del texto del HTML (mediante `innerHTML`). La figura 5 muestra el tiempo de procesamiento de la plantilla patrón Yeast[6] en distintos navegadores. La plantilla patrón contiene multitud de estructuras Yeast, de todos los tipos posibles, siendo su tamaño y estructura complejas[7]. Como puede apreciarse, existe una gran diferencia entre el más lento (Internet Explorer 6.0, y sus derivados, AOL y DeepNet), 4,68 sg., y el más rápido (Safari 4.0.2), 0.13 sg.

El procesamiento hubiese sido mucho más rápido si los valores procesados se hubiesen escrito directamente en el documento, mediante la función `document.write()`, que además está disponible de forma estándar en cualquier navegador, incluso en los más sencillos. Así, en vez de procesar

```
<p>My name is $name$. I work at <A href="$link$"> $link_txt$</A></p>
```

Se procesaría:

```
<p>My name is <script>document.write(name)</script>
I work at <script>document.write('<A href="'+link+'">'+link_txt+
'</A>')</script></p>
```

[4]Uno de los errores informados puede verse en `https://bugzilla.mozilla.org/show_bug.cgi?id=356027`.

[5]Véanse por ejemplo `http://www.w3schools.com/browsers/browsers_stats.asp`, `http://www.Webreference.com/stats/browser.html` o `http://www.thecounter.com/stats/`

[6]La plantilla se encuentra en `http://www.unirioja.es/cu/fgarcia/yst/bench/Patron_DOM.html`

[7]El experimento se ha realizado en una máquina Intel Core Duo T8300 a 2.40 GHz, 2 Gb de memoria RAM y con un S.O. Windows XP S.P. 3.

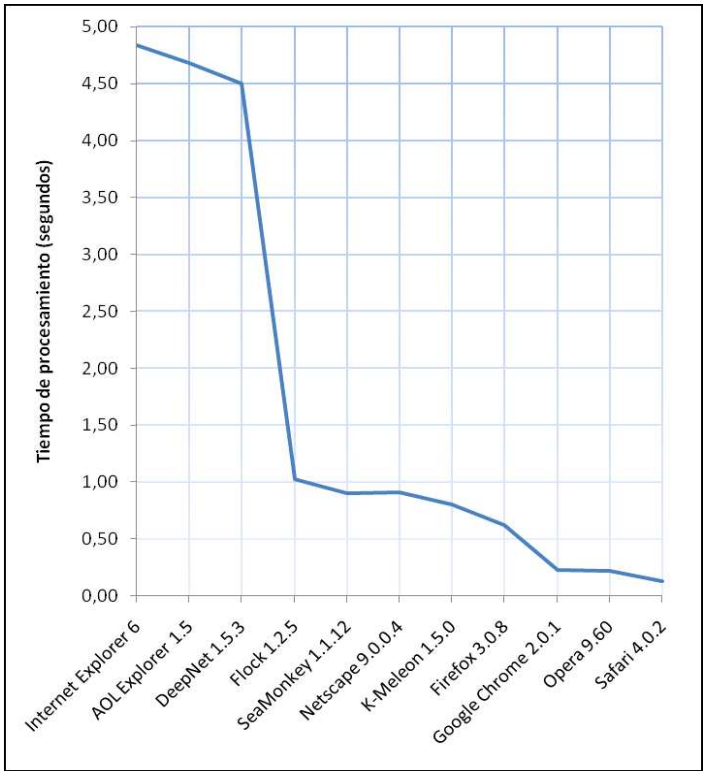


FIGURA 5. Tiempo de procesamiento en modo DOM de la plantilla patrón Yeast en distintos navegadores.

Además, el código anterior sería admitido y procesado sin ningún problema y de forma uniforme en cualquier navegador y versión de navegador, por antigua que fuese. Esa era la forma en la que eran procesadas las antiguas plantillas JST, y presenta evidentes desventajas y problemas de usabilidad en el proceso para los diseñadores gráficos. No son previsualizables en los editores HTML y por su complejidad son proclives a generar errores. Por lo tanto nos encontramos ante un dilema. Por un lado, pensado en los usuarios, una forma más rápida y contemplada por más y más antiguos navegadores, pero que, por el contrario, es menos amigable y provoca más errores desde el punto de vista de los diseñadores gráficos. Afortunadamente Yeast proporciona lo mejor de ambos mundos. Los diseñadores podrán seguir trabajando usando el "*modo DOM*", usando atributos y expresiones Yeast, pero, una vez terminado y probado su trabajo, las plantillas pueden traducirse a otra versión, que llamaremos "*modo textual*".

Yeast proporciona otro motor de plantillas para el modo textual. Está basado en una serie de funciones que emulan el procesamiento de las instrucciones de procesamiento Yeast (`value`, `if`, `apply`, etc), pero manipulando el HTML en forma

de cadenas de caracteres: por ejemplo la función `YST.Txt.value` emula `value`, o `YST.Txt.apply` para `apply`. También contempla la definición e inclusión de subplantillas, así como interacciones AJAX. El resultado final de la ejecución de estas funciones es una cadena de caracteres que contiene el HTML correspondiente al elemento Yeast procesado. Este HTML es escrito en el documento final mediante una llamada a la función JavaScript `document.write`. Esta forma de procesamiento es totalmente funcional en todas las pruebas realizadas, no encontrándose ningún fallo en ninguna de las versiones de navegador analizadas.

En cuanto al rendimiento, el motor textual supera al de la versión DOM, obteniéndose incrementos de rendimiento de hasta el 900 % en el mejor de los casos, tal como se muestra en la figura 6[8].

Yeast proporciona un traductor de modo DOM a modo textual, que puede usarse sólo[9], pero también está integrado con la infraestructura de servidor Yeast-Server. De esta forma ni siquiera es necesario que el diseñador traduzca las plantillas. Yeast-Server lo hará por él automáticamente la primera vez que se use cada plantilla. Un ejemplo de traducción podría ser el paso del siguiente elemento Yeast:

```
<tr yst="apply" ystSet="pfcs"> <td>$e.titulo$</td> </tr>
```

Al siguiente código JavaScript:

```
<script> document.write(YST.Txt.apply(new Array(), 0, new Object(),
'pfcs', ['<tr><td>$e.titulo$</td></tr>'])) </script>
```

Queda fuera del ámbito del articulo la explicación del algoritmo de traducción así como del significado de los parámetros utilizados en las funciones del namespace `YST.Txt`.

5.2. Yeast en el servidor. Como se deduce de nuestra propuesta de doble modelo, los programadores de la aplicación pueden trabajar en el desarrollo de la lógica de negocio en paralelo a los diseñadores, sin necesidad de mantener un contacto continuo con estos para explicarles cómo es el modelo de datos que deben usar en la plantilla. La integración de la capa de presentación puede hacerse al final.

Los programadores sólo deben saber cuál es el modelo de datos que la plantilla necesita. Para ello consultan el modelo del diseñador, el cual, la mayoría de las veces, es suficientemente auto-explicativo, reduciéndose la necesidad de comunicación entre los equipos de desarrollo (en la línea del requisito 2 para el desarrollo usable enunciado en la sección 2.2). Los programadores implementan el código que debe ejecutarse al producirse una petición a la aplicación. Este código finalizará recabando los datos dinámicos necesarios del modelo de datos de la aplicación (propuesta de doble modelo) para usarlos en la composición de la respuesta a la petición. A la plantilla no le importa de dónde salen los datos reales (un grafo de objetos, un SGBD o un documento XML).

[8]Para la realización de este experimento se ha utilizado el mismo entorno de ejecución de la prueba anterior cargando la versión textual de la plantilla patrón, traducida por Yeast. La plantilla traducida está en `http://www.unirioja.es/cu/fgarcia/yst/bench/Patron_txt.html`.

[9]Incluso hemos realizado un servicio de traducción on-line (acúdase a `http://yeastdemo.appspot.com/translate.html`).

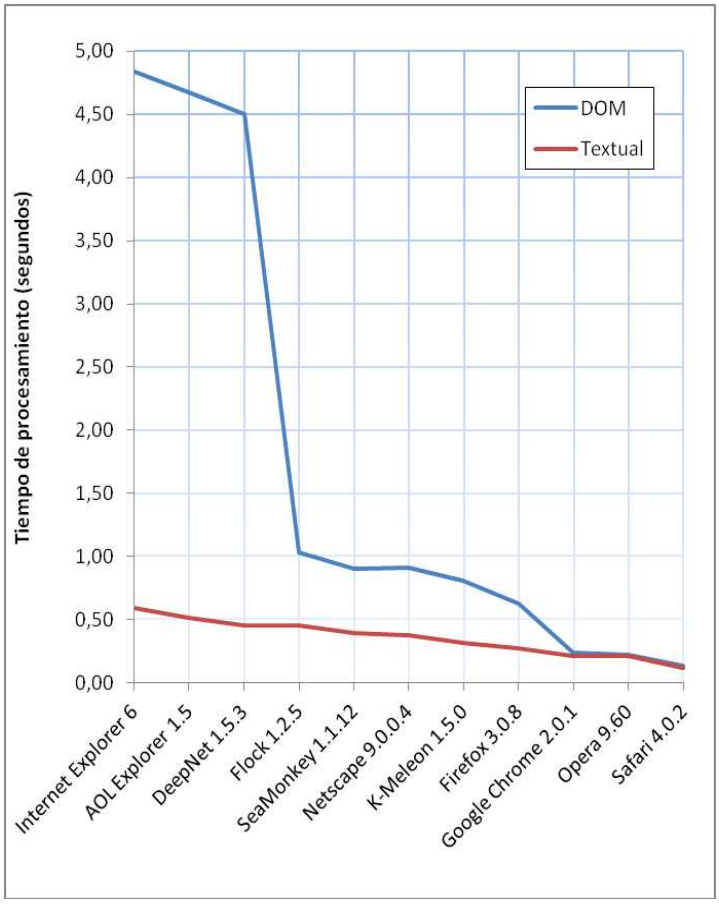


FIGURA 6. Comparativa del tiempo de procesamiento en modo DOM y textual de la plantilla patrón Yeast en distintos navegadores.

Sin embargo los datos brutos obtenidos del modelo de la aplicación no son aptos para la plantilla. Deben ser transformados para que se correspondan con el formato del modelo de la plantilla. Es decir, los datos de la aplicación, vengan de donde vengan serán transformados en una cadena de caracteres que contendrá código JavaScript que reemplazará al modelo del diseñador incluido en la plantilla a modo de prueba.

Por tanto el procesamiento final de una plantilla Yeast en el servidor es extremadamente sencillo; simplemente operaciones sobre cadenas de caracteres: localización de la sección del modelo y su sustitución por el modelo real adaptado del

modelo de la aplicación. Aún así, Yeast proporciona un API propio que puede ser usado en el desarrollo de aplicaciones J2EE[10].

5.2.1. Yeast-Server. El propósito de Yeast-Server es ayudar a los programadores de aplicaciones Web que usen plantillas Yeast, gestionando el almacenamiento y configuración de las mismas y facilitando su procesamiento. El uso de las herramientas proporcionadas por Yeast-Server supone una reducción del número de líneas de código. Además, parte de ese código puede ser escrito de forma declarativa, mediante ficheros de configuración, lo que permite hacer cambios en el diseño de las plantillas sin necesidad de recompilar la aplicación.

En cuanto a la arquitectura interna, Yeast-Server sigue la arquitectura MVC+mT descrito en la sección 4.2. De hecho una pieza clave en el API son los transformadores (clases que implementarán el interface `org.ystsrv.Transformer`). El API proporciona una serie de transformadores por defecto para tipos de datos simples, objetos de clases que sean conformes a la especificación JavaBeans [71], así como colecciones de los anteriores.

Yeast-Server hace uso de la tecnología de Servlets Java [68]. Yeast-Server añade funcionalidad a la clase `HttpServlet` de Java, proporcionando la clase `org.ystsrv.servlet.YSTServlet`. La funcionalidad básica de un `YSTServlet` es la siguiente:

1. Recuperar los datos necesarios del modelo de aplicación.
2. Seleccionar la plantilla adecuada para usarla como vista.
3. Inyectar los datos recuperados en la plantilla seleccionada y enviarla al cliente.

Yeast-Server puede ser configurado para devolver plantillas en modo-DOM o en modo textual. Para ello integra el traductor de plantillas descrito en la sección 5.1.5.

5.2.2. Características del servidor Yeast-Server. Sería muy tedioso describir las características de Yeast-Server. Para ello puede consultarse el documento [28]. Nos conformaremos con un listado de las más importantes:

- Gestión de plantillas desde almacenes en directorios o en ficheros JAR o ZIP. Posibilidad de agrupar plantillas en almacenes de plantillas y personalizar la respuesta eligiendo uno u otro almacen.
- Despliegue y modificación "en caliente" de plantillas, sin necesidad de reiniciar la aplicación
- Cacheo de plantillas
- Gestión automática de la codificación de caracteres de la plantilla.
- API de alto y bajo nivel, disponiendo de varios grados de automatización a elección del programador.
- Soporte para interacciones AJAX
- Extracción directa de datos de una base de datos
- Sistema de log integrado y configurable

[10]Está en fase de desarrollo un API para PHP.

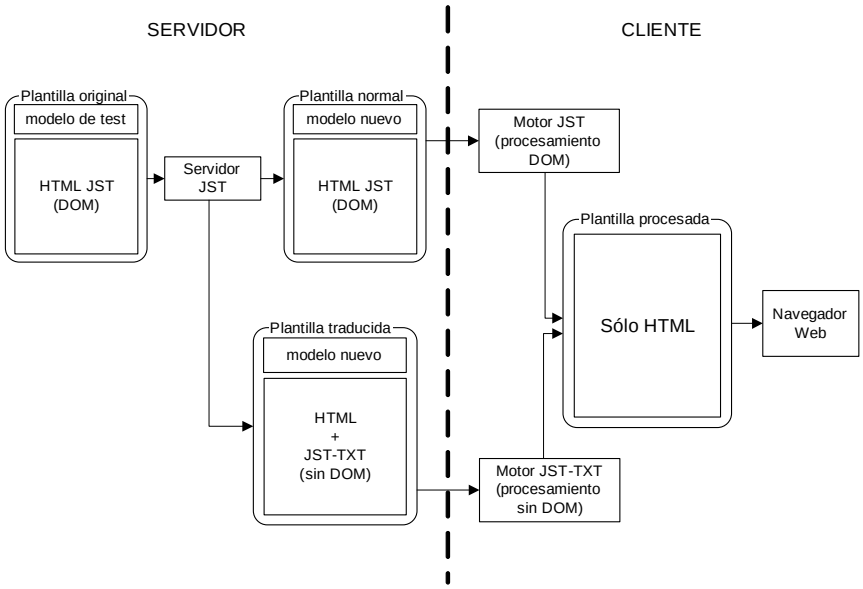


FIGURA 7. Esquema de procesamiento de Yeast-Server.

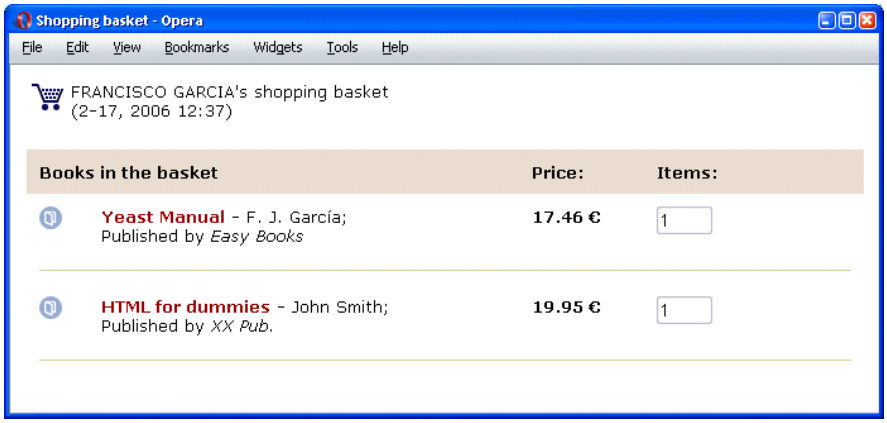


FIGURA 8. Ejemplo de aplicación Yeast.

5.2.3. Un ejemplo. Supongamos que deseamos desarrollar una aplicación on-line de venta de libros. La pantalla de compra podría tener un aspecto parecido al mostrado en la figura 8 siguiente (con el típico carro de la compra).

Supongamos que el diseñador de la plantilla (`Basket.html`) ha creado la misma usando el siguiente modelo de diseñador (con secciones de inicialización y modelo):

```
<script yst="init">
  // Función constructora
```

```
  function Book(id, tit, auth, price, publ) {
    this.id = id;
    this.title = tit;
    this.author = auth;
    this.price = price;
    this.publisher = publ;
  }
</script>
<script yst="model">
  customer="FRANCISCO GARCIA"
  time = '2-17, 2006 12:37';
  books = [new Book('0','Yeast Manual',
              'F. J. García', 17.46,'Easy  Books'),
           new Book('1','HTML for dummies',
              'John Smith', 19.95, 'XX Pub.')];
</script>
```

Se trata de un nombre de Usuario, una fecha y una lista de datos de libros, agrupados en una array de objetos JavaScript (construidos mediante la función `Book`).

Basándose en la estructura del modelo del diseñador y con el conocimiento del modelo de la aplicación que él ha diseñado, el programador es capaz de saber de dónde tiene que extraer los datos dinámicos que la plantilla necesita. Para responder a la petición de compra desarrolla la siguiente clase, extendiendo de `YSTServlet`:

```
1. package ystsrv.demo;
2.
3. import java.util.*;
4. import javax.servlet.http.HttpSession;
5. import org.ystsrv.servlet.YSTContext;
6. import org.ystsrv.servlet.YSTServlet;
7.
8. public class ShowBasket extends YSTServlet {
9.
10.  protected String handle(YSTContext context) {
11.   // Retrieve the customer's name from the session
12.   HttpSession session = context.getRequest().getSession(false);
13.   String customer = (String) session.getAttribute("customer");
14.
15.   // Get input params
16.   String[] ids = context.getRequestParameterValues("ids");
17.
18.   // Persistence method that returns the list of Book objects
19.   List books = processBooksInBasket(ids);
20.
```

```
21.    // Push the data needed for the response onto the template
22.   context.setResponseContentType("text/html");
23.   context.toResponse("books", books, "new Book({id}, {title},
       {author.name} +  ' ' + {author.surname}, {price},
       {publisher})");
24.   context.toResponse("customer", customer);
25.   context.toResponse("time", new Date(),"M-d, yyyy hh:mm");
26.
27.   // Returns the name of the template to be used in the response
28.   return "Basket";
29.  }
30. }
```

La clase recoge los identificadores de los libros comprado mediante los parámetros CGI ids (línea 16). Recupera de la capa de persistencia los datos de los mismos (19) en forma de una lista de objetos de clase `Book` (la coincidencia de nombre entre las clases Java y JavaScript es sólo eso, una coincidencia). Book es la típica clase JavaBean, con métodos set y get para gestionar las propiedades de los objetos (que son `id`, `title`, `price`, `publisher` y `author`, siendo esta última otro bean con propiedades `name` y `surname`).

Estos datos son "incrustados" en la plantilla a través de un objeto de tipo `YSTContext` disponible para el servicio (línea 10). Mediante la llamada al método `toResponse` (línea 23) se sustituyen los valores de prueba de la plantilla por los valores reales. En este momento se produce la transformación entre modelos. Uno de los parámetros que se especifican en la llamada al método `toResponse` es el formato de transformación que debe utilizarse. El programador lo obtiene del mismo formato especificado por el diseñador gráfico. Se sustituyen los valores concretos de prueba por referencias a las propiedades de los objetos que contienen los valores, encerrando estas referencias entre un par de {}. Así, si el código JavaScript usado en el modelo de diseñador de la plantilla para representar un objeto `Book` era:

```
new Book('0','Yeast Manual', 'F. J. García',17.46, 'Easy  Books')
```

El programador especificará una cadena de formato de transformación como la siguiente:

```
"new Book({id}, {title}, {author.name} +  ' ' + {author.surname},
 {price}, {publisher})"
```

Todo es automático. El resto de datos, nombre de usuario y fecha, también son incrustados en la plantilla de forma similar (líneas 24 y 25). El servicio termina devolviendo el nombre de la plantilla a utilizar (línea 28).

5.3. Yeast y AJAX. Utilizando de forma conjunta tecnologías ya existentes, AJAX ha conseguido que los diseñadores gráficos enriquezcan sus diseños con funcionalidades propias de aplicaciones de escritorio, dando lugar a las conocidas como Rich Internet Applications (RIA). Desde la publicación del ya clásico artículo de J.J. Garrett [31], el desarrollo de este tipo de aplicaciones se ha disparado. Tal como se recoge en [34] casi el 50 % de los sitios Web más visitados en EEUU

utiliza este tipo de tecnología. Actualmente existen varios frameworks que permiten el desarrollo de aplicaciones AJAX. Considérense, por ejemplo, los populares Dojo Toolkit [19], jQuey [10] o Prototype [60]. La mayoría de estas herramientas encapsulan la complejidad del establecimiento de conexiones asíncronas con el servidor, dejando para el diseñador la tarea, muchas veces no trivial, de procesar los datos recibidos y de actualizar la página, recurriendo a funciones que facilitan la manipulación DOM de la misma.

Yeast facilita enormemente el procesamiento AJAX, tanto en el establecimiento de la conexión, como en el posterior tratamiento de los datos recibidos. En primer lugar, el modelo de datos de la plantilla ya está diseñado en un formato que facilita la carga dinámica de nuevos datos. De hecho, con Yeast, el navegador y el servidor no intercambian datos XML, como en muchos de los sistemas AJAX actuales; ni siquiera es necesario usar JSON, JavaScript Object Notation [39], como en tantos otros frameworks existentes. Con Yeast, el mismo formato de datos usado en la plantilla, en el modelo del diseñador, es utilizado en subsecuentes interacciones con el servidor para cambiar los valores de los datos del modelo o, incluso, para recabar valores extra que no fueron cargados inicialmente. Una vez cargados, los nuevos datos son evaluados (usando la función JavaScript `eval` [50]), con lo que pasan a estar a disposición de la plantilla. Los elementos que cambian dinámicamente son marcados en la plantilla mediante el atributo `yst`, al que se asigna el valor '`ajax`'. Cuando los nuevos datos son recibidos los elementos afectados son reprocesados, de la misma manera que lo fueron la primera vez que la plantilla se cargó; pero en esta ocasión los valores del modelo son diferentes.

Yeast proporciona utilidades destinadas al desarrollo de aplicaciones AJAX tanto para el lado del cliente, en la propia plantilla, como para el lado del servidor. Por una parte el motor Yeast incluye un API para el establecimiento de conexiones y re-procesamiento de nodos Yeast, cuyo funcionamiento ya ha sido aclarado en el párrafo anterior. Así mismo, Yeast-Server ofrece la posibilidad de desarrollar servicios AJAX mediante la extensión de la clase `org.ystsrv.servlet.AJAXServlet`, que es una especialización de `org.ystsrv.servlet.YSTServlet` (ver sección 5.2.1). El modo de trabajo de un `AJAXServlet` es idéntico al de un `YSTServlet`, es decir, recolección de datos necesarios y transformación de los mismos para adaptarlos al formato de la plantilla. La única diferencia es que el segundo devuelve siempre la plantilla completa, mientras que el primero sólo los datos transformados. De hecho, y de una manera muy sencilla, un mismo `AJAXServlet` puede ser utilizado en interacciones AJAX y no AJAX, devolviendo toda la plantilla o sólo la nueva sección de modelo, evitando, de esta manea, la duplicación de código habitual en otros frameworks, en los que, para atender a un mismo servicio que puede ser accedido de la forma normal o de forma AJAX, es necesario desarrollar dos servicios diferentes.

5.4. Cuestiones de accesibilidad. Últimamente se está poniendo mucho énfasis en el estudio de los problemas que pueden encontrar las personas con algún tipo de discapacidad, para acceder a los contenidos de las páginas en Internet [40]. Existen iniciativas, como la WAI Web Accessibility Initiative (`http://www.w3.org/`

`WAI/`), dedicadas a desarrollar e implantar una serie de pautas relativas al diseño Web, que fundamentalmente se basan en una separación estricta entre el contenido y la presentación del mismo. Estos esfuerzos se han sustanciado en la elaboración de una guía de desarrollo, Web Content Accessibility Guidelines (WCAG), que ya se encuentra en su versión 2.0 [73, 80]. Si bien WCAG 2.0 es más "abierta de miras" que su anterior versión [78] en referencia al empleo de tecnologías como JavaScript, los requisitos impuestos por la versión 1.0 de la guía todavía perdurarán a medio y largo plazo, ya que son la base de la mayoría de las regulaciones administrativas en materia de accesibilidad y estas regulaciones no suelen ni pueden ser cambiadas inmediatamente.

La accesibilidad exige que una aplicación Web debe funcionar independientemente de la tecnología del lado del cliente (WCAG 1.0), ya sea un navegador Web, un dispositivo móvil o una herramieta de ayuda a discapacitados sensoriales. Por ello, en el contexto de las aplicaciones accesibles, el empleo de JavaScript es un problema importante, ya que si el cliente no soporta JavaScript, o lo tiene deshabilitado, la página dejará de funcionar. Además el uso de JavaScript para la modificación dinámica de los contenidos de una página Web hace que sea más difícil explorarlos, bien porque hay zonas que cambian "sin previo aviso" o porque no permiten una correcta interpretación de los mismos a través de, por ejemplo, un lector para personas invidentes, como Windows-Eyes o JAWS. Debido al uso intensivo de JavaScript, las plantillas Yeast no cumplen los estándares de accesibilidad marcados por la WAI.

Para solventar estos problemas y además dar soporte a los navegadores que tienen JavaScript desactivado, aunque sólo sean un 5 % del total, la infraestructura de servidor de Yeast está a punto de incluir un procesador de plantillas, que denominaremos *Yeipee*. Se trata de realizar el procesamiento de la plantilla en el lado del servidor, de manera que lo que se envía al navegador ya no incluya ninguna instrucción JavaScript artificial relativa a Yeast. De esta forma el contenido entregado al navegador será accesible, siempre y cuando lo fuera en origen, antes de ser transformado en plantilla Yeast.

El procesador Yeipee utiliza para procesar una plantilla la versión traducida a modo textual de la misma. Yeippe encapsula un procesador JavaScript en el lado del servidor que ha sido desarrollado usando la librería Rhino [51], que será el encargado de ejecutar las funciones `YST.Txt.xxx` incluidas en la plantilla traducida. El conjunto es sensiblemente más lento que el servidor Yeast puro, pero como contrapartida el resultado es accesible. Actualmente estamos planeando el desarrollo de un procesador íntegramente desarrollado en Java que evitará el empleo de Rhino, esperándose una mejora apreciable en el rendimiento.

5.5. La importancia de la seguridad. El cuidado de la seguridad es un aspecto crítico de cualquier aplicación, y más aún en el caso de las aplicaciones Web [32, 87]. La complejidad de la tarea viene incrementada por el hecho de que en una aplicación de este tipo, además del leguaje principal de desarrollo, suelen utilizarse cuatro lenguajes diferentes, si no más: HTML, CSS, JavaScript y SQL. Cada uno

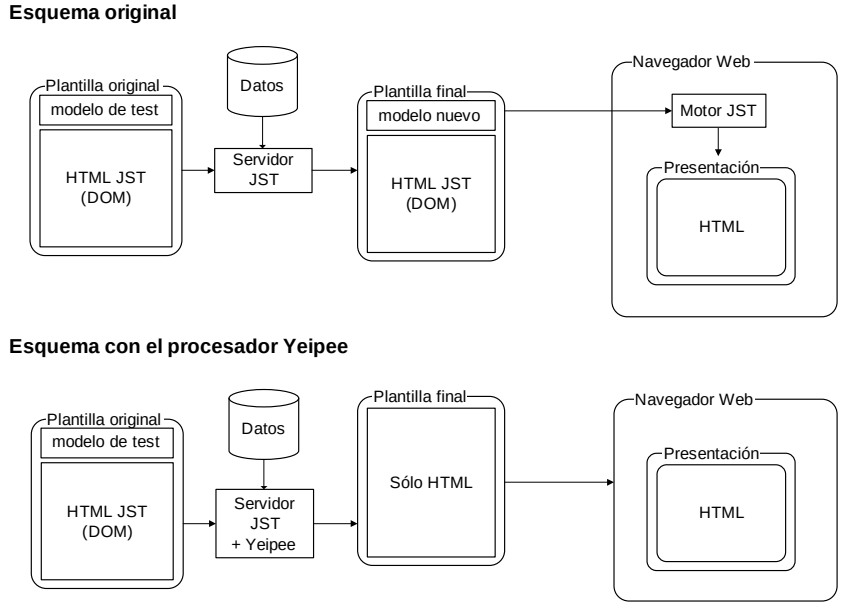


FIGURA 9. Esquema de funcionamiento del procesador Yeipee.

de ellos debe ser tratado de forma diferente, por ejemplo, para evitar inyecciones de código, una de las formas de ataque más comunes.

Al desarrollar Yeast se ha cuidado la seguridad. En primer lugar debemos decir que de los anteriores lenguajes nos hemos despreocupado de dos: SQL, ya que Yeast no se ocupa de la lógica del acceso a datos; y CSS, ya que, por diseño, no se puede utilizar Yeast para alterar el código CSS salvo la determinación dinámica de la clase CSS de un elemento. Las técnicas de protección contra los ataques relacionados con SQL y CSS se deberán seguir aplicando de forma ortogonal a Yeast. Yeast colabora con el cuidado de la seguridad en lo relativo a código HTML y a JavaScript.

La principal amenaza en Yeast viene dada por el uso de la función `eval` [87], empleada para la evaluación de las expresiones Yeast en las plantillas. Para mitigar los riesgos se ha trabajado tanto en la infraestructura de servidor Yeast-Server como en el motor de plantillas. En primer lugar, las actuaciones realizadas sobre Yeast-Server deben centrarse exclusivamente en los valores de la sección de modelo que va a incrustar en la plantilla procesada, procurando que entre esos valores solo haya valores, es decir no aparezca nada que pueda interpretarse como código susceptible de ser ejecutado maliciosamente en el navegador del cliente. Si así fuese un atacante podría utilizar esta vulnerabilidad para completar un ataque de tipo XSS (Cross Site Scripting, [84]) o de otro tipo [32]. Son tres los tipos de valores básicos que podemos encontrar en la sección de modelo, numérico, fecha (`new Date(...)`) y textual, de los cuales no son peligrosos a este respecto los dos

primeros. Ninguno de ellos puede acarrear código. El peligro se centra en los valores textuales. Por ejemplo si un atacante lograra introducir como valor de una variable de modelo algo como `someData = for(;;) alert('You are trying to hang me');`, ese código al ser insertado en el documento provocaría una molestia para el usuario (y se trata sólo de un inocente ejemplo de lo que podría pasar). La situación anterior no puede darse con Yeast-Server, ya que todos los valores textuales son devueltos como cadenas de caracteres, escapando los caracteres potencialmente peligrosos como ¨, ´, `&`, `<`, \ o `/` (que entre otras cosas impide que la cadena `</script>` pueda ser escrita directamente, evitando la finalización anómala de la sección de modelo). Así el ejemplo sería insertado en la sección de modelo como `someData = 'for(;;) alert(Ýou are trying to hang me)';`. Ya no se trata de un código ejecutable, sino de una cadenas de caracteres "un poco rara".

En el lado del motor Yeast, por diseño, no está permitido el empleo de atributos Yeast en elementos `<script>`, lo que impide la utilización de scripts de contenido dinámico, evitándose la inclusión de códigos como los del ejemplo anterior en este tipo de elementos, que los ejecutarían sin miramientos. El mecanismo fundamental de cuidado de la seguridad viene de la mano del uso del API DOM para procesar las plantillas, lo que garantiza que todo el contenido variable procesado es tratado como texto y no cómo código. Por ejemplo, si en la sección de modelo el valor hubiese sido insertado como `someData = '\<script\>for(;;)... <\/script\>';`, para ser usado en `<p yst="value">$someData$</p>`, el contenido insertado finalmente en la plantilla por el motor sería `<p yst="value">&lt;script&gt; for... &lt;/script&gt;</p>`. El propio API DOM se encarga de sustituir los caracteres `<` o `>` por sus correspondientes entidades (`&lt;` y `&gt;`), de modo que el contenido no es interpretado como marcado (ejecutable en este caso), sino como simple texto.

Si en vez de Yeast en modo DOM se utiliza Yeast en modo textual, se pueden encontrar similares mecanismos de protección. En este caso la situación es potencialmente más insegura, ya que no disponemos de la protección del API DOM al insertar el texto en la plantilla. Todo se hace con escrituras directas `document.write()`, que en principio, podrían incrustar cualquier cosa en la plantilla. Para evitar esos peligros, en primer lugar el traductor no procesa elementos script, lo que impide la utilización de scripts de contenido dinámico. Y, en segundo lugar, los valores textuales son escapados por el API `YST.Txt` de modo similar a como se hace en DOM.

Podemos afirmar, por tanto, que Yeast ayuda a evitar ataques de tipo XSS, y que no añade nuevos agujeros de seguridad. En futuras versiones Yeast mejorará el cuidado de la seguridad incluyendo mecanismos opcionales de evitación de JavaScript Hijacking [17].

6. Otros trabajos relacionados

Además de los trabajos referenciados a lo largo del artículo, en esta sección recogemos una serie de publicaciones que guardan relación y pueden ayudar al lector a ampliar sus conocimientos en la materia.

En Internet hay multitud de documentación sobre sistemas de plantillas. Si se buscan trabajos de ámbito académico, la primera referencia que encontramos es [44], que presenta TML, un lenguaje de plantillas basado en etiquetas XML añadidas a documentos HTML. Se basa en *TRiX*, un framework para el procesamiento de plantillas TML, siendo su extensibilidad su principal ventaja. *Mixer* es material de estudio en [83], un sencillo sistema de plantillas de plantillas para aplicaciones basadas en servlets que analiza muy someramente el problema de la separación. Otra referencia es [3], donde se describe *PageGen*, un sistema de plantillas para Microsoft ASP que demuestra limitaciones achacables al fuerte acoplamiento entre su base de datos de control y de datos. [35] estudia la conveniencia de usar "plantillas estándar", es decir, plantillas en las que no se permite utilizar etiquetas no-HTML. *STRUDEL* [23] es un framework para aplicaciones Web que manejan gran volumen de datos (*intensive data based Web application*) que incluye su propio lenguaje de plantillas, el cual puede ser criticado porque permite la inclusión directa de código Java en las plantillas. En [53] se presenta *WISBuilder*, que emplea una arquitectura MVC salpicada de ideas tomadas de la programación generativa. Menciona la separación de tareas entre diseñadores y programadores pero no analiza sus ventajas. Para la generación de la presentación utiliza documentos XSLT. También nos gustaría citar a [16], que propone un framework de desarrollo independiente de aspectos de localización (dónde ejecutar el código y cuándo cargarlo). En él se analiza la posibilidad de generar páginas dinámicas en el navegador, utilizando un script basado en DOM y unos datos en bruto. Si bien no llega a proponer la utilización de plantillas browser-side, la idea guarda relación con ellas y, por tanto, con Yeast.

Otro trabajo interesante es [14], donde se presenta el lenguaje <*bigwig*> para el desarrollo de aplicaciones Web. Incluye el sublenguaje *DynDoc* para la especificación de plantillas HTML, y la estructura de datos *DynDocDag* [13], que facilita la representación de la plantilla en un formato que permite la reconstrucción del documento en el propio navegador mediante un procesador JavaScript. DynDocDag fue diseñado para mejorar las posibilidades de cacheo de documentos DynDoc, pero no presta atención a temas relacionados con la separación analizada en el presente artículo.

El sucesor de <bigwig> es *JWIG* [74, 18, 47]. Se trata de un sistema basado en MVC que emplea un lenguaje de plantillas en las que el diseño gráfico es troceado en una serie de sub-plantillas HTML desarrollados por los diseñadores. El conjunto es ensamblado con un programa desarrollado por el programador. El conjunto es tan simple y el lenguaje tan limitado que cumple con todas las reglas de [56]. Sin embargo, en nuestra opinión adolece de algunos problemas, ya que el diseñador pierde la noción de diseño global al tener que enfocar su trabajo al diseño de pequeños trocitos de HTML. Además el programador complica su trabajo al tener que desarrollar un programa de ensamblado. El entorno introduce un montón de herramientas adicionales para diseño, prueba y depuración que complican el proceso. Siguiendo en la serie de trabajos relacionados con JWIG, en [11] también se hace un estudio del proceso de desarrollo de aplicaciones Web, relacionado con el nuestro de la sección 2. Tras analizar varias tecnologías como JSTL [69] y XSLT,

los autores introducen la noción de contrato entre programador y diseñador gráfico que permite trabajar a ambos de forma independiente y con mínima comunicación. La diferencia con nuestra propuesta consiste en que con JWIG la iniciativa para la construcción de la presentación parte del programador, no del diseñador como nosotros mantenemos. La separación entre sub-equipos de desarrollo en aplicaciones Web también es materia de estudio en [63].

En la línea de trabajos académicos que tratan de sistemas de plantillas para navegador encontramos [72], donde se presentan las *FlyingTemplates*. A diferencia de nuestra propuesta con Yeast, en la que el tanto el motor como el lenguaje son nuevos y enfocados al navegador, con FlyingTemplates se trabaja con sistemas de plantillas ya existentes (por ejemplo, en el artículo con plantillas Smarty [65] para PHP). Los autores han replicado un motor de Smarty para navegador. Ante una petición el servidor envía al navegador un esqueleto de código que incluye el procesador Smarty replicado, una instrucción de carga de la plantillas Smarty y los datos necesarios para el procesamiento en formato JSON. El trabajo analiza las posibilidad de cacheo de este sistema.

Otros sistemas de desarrollo de aplicaciones Web se basan en el empleo de componentes [8, 20]. Estos sistemas, si bien pueden acelerar el desarrollo de aplicaciones en las que determinados elementos aparecen repetidas veces, son, en nuestra opinión, contrarios al desarrollo de interfaces guiado por el diseñador Web. En este tipo de desarrollos el diseñador gráfico no puede abordar el desarrollo de una página completa, sino que debe centrarse en trocitos de HTML que luego son ensamblados por el programador.

En relación con la filosofía del doble modelo, podemos relacionar nuestro trabajo con otros como [88], donde se propone un proceso de adaptación similar al del doble modelo, en este caso entre el front-end y el back-end de una arquitectura MVC de dos niveles que es la base de la *metodología MODFM* para el desarrollo de aplicaciones Web. O [9], que presenta la propuesta del *mvc-dual*, en el cual el controlador de la aplicación MVC es dividido entre los lados del cliente y del servidor. La propuesta reduce las interacciones entre servidor y cliente en aplicaciones Web. Para ello en determinadas peticiones se devuelven al navegador más datos de los estrictamente necesarios, datos que pueden ser útiles en respuesta a determinadas interacciones del usuario con el interface. La página en el navegador tiene su propio controlador que interroga el modelo local en lugar de desencadenar una nueva petición al servidor. El trabajo no tiene nada que ver con sistemas de plantillas, pero reconoce la posibilidad de disponer de dos modelos en la aplicación. El último trabajo que citamos es [7], que utiliza plantillas para la generación de código seguro. En este trabajo el lenguaje de plantillas empleado no permite hacer cómputos con los datos del modelo, por lo que, cuando es necesario un valor derivado, se propone un esquema de transformación entre el modelo de la aplicación y la plantilla que recuerda el cometido de los transformadores en la arquitectura MVC+mT.

7. Trabajo futuro

El proyecto de desarrollo de Yeast no está cerrado, siendo numerosas las anotaciones en nuestra lista de "to-do".

Actualmente las sub-plantillas (ver sección 5.1.4) definidas en una plantilla son locales a la misma, es decir, sólo puede ser usadas en esa plantilla. Son locales a la misma. Tenemos casi desarrollado una modificación del motor que permitirá la carga de sub-plantillas externas desde cualquier URL. Esto dotará Yeast de una herramienta poderosa, que entre otras cosas propiciará la reutilización de bibliotecas de sub-plantillas, con el consiguiente ahorro de tiempo y esfuerzo de desarrollo.

Como ya hemos comentado en la sección 5.4, planeamos desarrollar un procesador accesible de plantillas en el lado del servidor que elimine en gran medida el empleo de la librería Rhino, lo que redundará en un incremento de rendimiento apreciable.

Otra de las mejoras que esperamos incluir en un futuro no muy lejano será la posibilidad de cachear las plantillas en el navegador, siguiendo aproximaciones similares a [13, 72], pero en nuestro caso de forma mucho más sencilla, ya que, a diferencia de los trabajos citados, no se requiere de ningún pre-procesamiento ni elaboración posterior en el navegador. Se trata de aprovechar el hecho de que la única parte de una plantilla Yeast que cambia de una interacción a otra es el contenido de la sección de modelo. El cuerpo de la plantilla siempre permanece igual. Yeast-Server puede aprovechar esta circunstancia, permitiendo que el cuerpo de la plantilla se almacene en el navegador. La idea es crear dinámicamente para cada plantilla un fichero JavaScript que contenga su cuerpo, por ejemplo llamado tmplXBody.js. En este fichero se programará la escritura del contenido del cuerpo de la plantilla, algo como

```
document.write(... el contenido del body ...)
```

En la plantilla, el cuerpo será sustituido por la instrucción `<body><script src="tmplXBody.js"></script></body>`, reduciéndose de forma drástica el tamaño de la misma. La primera vez que la plantilla sea cargada en un navegador se cargará el fichero tmplXBody.js, que quedará cacheado. Esta estrategia mejora el tiempo de carga de las plantillas y reduce el tráfico de red.

La última idea que nos gustaría comentar es la ampliación de la "filosofía" Yeast. Por ahora la parte dinámica de una plantilla se limita a los datos. El diseñador incluye una sección de modelo con datos de prueba que son sustituidos en tiempo de ejecución por datos reales. La propuesta sería incluir comportamiento dinámico. Un ejemplo fácil de entender es el del código de validación asociado a un campo de un formulario. El diseñador puede incluir un código de validación, una función JavaScript, "de pega", que solo sirva para comprobar que en efecto se invoca la función de validación sobre ese campo. En tiempo de ejecución ese código de validación es sustituido por el real, que podría estar adaptado incluso a la petición concreta que se está atendiendo.

8. Conclusiones

A lo largo de los últimos años, especialmente desde 2003 cuando contamos con la colaboración de Mirian, hemos venido trabajando sobre el sistema de plantillas Web Yeast, elaborando materiales más académicos unas veces, y más aplicados en otras ocasiones. El trabajo ha estado plagado de dificultades pero ha sido satisfactorio. Hoy en día podemos ofrecer a la comunidad de desarrolladores nuestro producto, y a la comunidad de investigadores Web nuestras opiniones y aportaciones sobre el problema de la separación entre vista y modelo.

El presente trabajo ha repasado los resultados obtenidos en torno al sistema de plantillas Yeast y sus fundamentos. Los requisitos para un desarrollo usable de aplicaciones Web y la propuesta de doble modelo unida a la arquitectura MVC+mT pueden ser utilizados para producir aplicaciones de manera más eficiente y libres de los problemas de acoplamiento. El sistema de plantillas Yeast es una opción más en el catálogo de soluciones para la realización de aplicaciones Web, que, en nuestra opinión, permite acelerar el desarrollo. Su caracterización como sistema de plantillas para navegador lo hace idóneo para el desarrollo no sólo de aplicaciones AJAX, sino como una opción para realizar páginas Web con contenido HTML dinámico o autogenerado. Al trasladar el procesador de plantillas al navegador Yeast permite hacer un aprovechamiento más eficiente de la potencia de proceso en cliente y servidor.

Referencias

[1] *Ruby on Rails*. `http://rubyonrails.org/`.

[2] Adobe. *Adobe Flex 3*. `http://www.adobe.com/es/products/flex/`.

[3] N. Al-Darwish. PageGen: an effective scheme for dynamic generation of web pages. *Information and Software Technology* **45**(10), 651–662, 2003.

[4] Apache Software Foundation. *Struts*. `http://struts.apache.org/`.

[5] Apache Software Foundation. *Tapestry*. `http://tapestry.apache.org/`.

[6] Apache Software Foundation. *The Apache Velocity Project*. `http://velocity.apache.org/`.

[7] J. Arnoldus, J. Bijpost, M. van den Brand. Repleo: a syntax-safe template engine. En *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pp. 25–32. ACM, New York, 2007.

[8] R. Barrett, S. J. Delany. OpenMVC: a non-proprietry component-based framework for web applications. En *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pp. 464–465. ACM, New York, 2004.

[9] K. Betz, A. Leff, J. T. Rayfield. Developing highly-responsive user interfaces with DHTML and servlets. En *IPCCC-2000: 19th IEEE International Performance, Computing, and Communications Conference*, pp. 437–443. IEEE Computer Society, 2000.

[10] B. Bibeault, Y. Katz. *jQuery in Action*. Manning Publications Co., 2008.

[11] H. Böttger, A. Møller, M. I. Schwartzbach. Contracts for cooperation between web service programmers and HTML designers. *Journal of Web Engineering* **5**, 2003.

[12] A. Bower, B. McGlashan. Twisting the triad: The evolution of the Dolphin Smalltalk MVP application framework. En *European Smalltalk User Group Conference, 2000*. 2000.

[13] C. Brabrand, A. Moller, S. Olesen, M. I. Schwartzbach. Language-based caching of dynamically generated HTML. *World Wide Web* **5**(4), 305–323, 2002.

[14] C. Brabrand, A. Moller, M. I. Schwartzbach. The <bigwig> project. *ACM Trans. Internet Technol.* **2**(2), 79–114, 2002.

[15] R. Cardone, D. Soroker, A. Tiwari. Using XForms to simplify Web programming. En *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pp. 215–224. ACM, New York, 2005.
[16] P. H. Chang, W. Kim, G. Agha. An adaptive programming framework for web applications. *Applications and the Internet, IEEE/IPSJ International Symposium on* **0**, pp. 152+, 2004.
[17] B. Chess, Y. T. O'Neil, J. West. *JavaScript hijacking*. 2007. `http://www.fortify.com/servlet/download/public/JavaScript_Hijacking.pdf`.
[18] A. S. Christensen, A. Moller, M. I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Trans. Program. Lang. Syst.* **25**(6), 814–875, 2003.
[19] Dojo. *The Dojo JavaScript Toolkit*. `http://www.dojotoolkit.org/`.
[20] S. Ducasse, A. Lienhard, L. Renggli. Seaside: A flexible environment for building dynamic web applications. *Software, IEEE* **24**(5), 56–63, 2007.
[21] Enhydra. *About XMLC*. `http://xmlc.objectweb.org/about.html`.
[22] D. Esposito. *Evergreen but still topical: MVC vs. MVP*. 2009. `http://dotnetslackers.com/articles/designpatterns/Evergreen-but-still-topical-MVC-vs-MVP.aspx`.
[23] M. Fernández, D. Florescu, A. Levy, D. Suciu. Declarative specification of Web sites with Strudel. *The VLDB Journal* **9**(1), 38–55, 2000.
[24] N. Ford. *Art of Java Web Development*. Manning Publications Co, 2004.
[25] M. Fowler. *Presentation model*. 2004. `http://martinfowler.com/eaaDev/PresentationModel.html`.
[26] P. Fraternali. Tools and approaches for developing data-intensive web applications: a survey. *ACM Comput. Surv.* **31**(3), 227–263, 1999.
[27] FreeMarker. *Freemarker*. `http://freemarker.sourceforge.net/`.
[28] F. García-Izquierdo. *Yeast Server - User guide*. 2009. `http://www.yeasttemplates.org/server/Yeast-Server_Manual.pdf`.
[29] F. García-Izquierdo, R. Izquierdo Castanedo. *Yeast templates tutorial*. 2009. `http://www.yeasttemplates.org/Yeast_Tutorial.pdf`.
[30] F. García-Izquierdo, R. Izquierdo Castanedo, A. Juan Fuente. A double-model approach to achieve effective model-view separation in template based web applications. En *Web Engineering*, volume 4607 of *Lecture Notes in Computer Science*, pp. 442–456. Springer Verlag, 2007.
[31] J. J. Garrett. *Ajax: A new approach to web applications*. `http://adaptivepath.com/ideas/essays/archives/000385.php`, 2005.
[32] D. Gollmann. Security in distributed applications. En *Advances in Software Engineering, LNCS 5316/2008*, pp. 254–276. Springer Verlag, 2008.
[33] J. Gossman. *Introduction to Model/View/ViewModel pattern for building WPF apps*. 2005. `http://blogs.msdn.com/johngossman/archive/2005/10/08/478683.aspx`.
[34] A. Guha, S. Krishnamurthi, T. Jim. Using static analysis for ajax intrusion detection. En *18th International World Wide Web Conference*, pp. 561–561, 2009.
[35] S. J. Halasz. An improved method for creating dynamic web forms using APL. *SIGAPL APL Quote Quad* **30**(4), 104–111, 2000.
[36] T. Hoffman. *Using XML Data Islands in Mozilla*. 2005. `http://developer.mozilla.org/en/docs/Using_XML_Data_Islands_in_Mozilla`.
[37] J. Hunter. *The Problems with JSP*. 2000. `http://www.servlets.com/soapbox/problems-jsp.html`.
[38] R. Izquierdo, F. J. García-Izquierdo, M. Andrés, A. A. Juan Fuente, P. Manrubia. JST: Towards a usable web site development method. En *ICWI/2003: Proceedings of the IADIS International Conference WWW/Internet 2003*, pp. 515–522. IADIS Press, 2003.
[39] JSON. *Introducing JSON*. `http://www.json.org/`.
[40] W. Kern. Web 2.0 - End of accessibility? – Analysis of most common problems with Web 2.0 based applications regarding Web accessibility. *International Journal of Public Information Systems 2008*(2), 131–154, 2008.

[41] S. Kojarski, D. H. Lorenz. Domain driven web development with WebJinn. En *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 53–65. ACM, New York, 2003.

[42] G. E. Krasner, S. T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Programming* **1**(3), 26–49, 1988.

[43] R. Krimen. *Jemplate*. `http://search.cpan.org/dist/Jemplate/`.

[44] A. Kristensen. Template resolution in XML/HTML. *Computer Networks and ISDN Systems* **30**(1-7), 239–249, 1998.

[45] K. Lieberherr, I. Holland, A. Riel. Object-oriented programming: an objective sense of style. En *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications*, pp. 323–334. ACM, New York, 1988.

[46] M. Mahemoff. *Ajax Design Patterns*. O'Reilly Media, 2006.

[47] A. Møller, M. Schwarz. JWIG: Yet another framework for maintainable and secure web applications. En *Proceedings of the 5th International Conference on Web Information Systems and Technologies*. INSTICC Press, 2009.

[48] R. Morales-Chaparro, M. Linaje, J. C. Preciado, F. Sánchez-Figueroa. MVC web design patterns and rich internet applications. En *Proceedings of the Jornadas de Ingeniería del Software y Bases de Datos*, 2007.

[49] M. Morris. XML Integration Features in IE5.x. *XML Journal* **2**(10), 2001.

[50] Mozilla-MDC. *Eval at Core JavaScript 1.5 Reference*. `http://developer.mozilla.org/en/CoreJavaScript1.5Reference/GlobalFunctions/eval`.

[51] mozilla.org. *Rhino: JavaScript for Java*. `http://www.mozilla.org/rhino/`.

[52] J. Nielsen. *Designing Web Usability*. New Riders Publishing, Indianapolis, USA, 2000.

[53] A. I. Ortiz-Cornejo, H. Cuayahuitl, C. Perez-Corona. Wisbuilder: A framework for facilitating development of web-based information systems. En *Electronics, Communications and Computers, 2006. CONIELECOMP 2006. 16th International Conference on*, p. 46. 2006.

[54] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM* **15**(12), 1053–1058, 1972.

[55] T. Parr. *String template*. `http://www.stringtemplate.org/`.

[56] T. J. Parr. Enforcing strict model-view separation in template engines. En *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pp. 224–233. ACM, New York, 2004.

[57] S. Pemberton. *XForms for HTML Authors*. 2003. `http://www.w3.org/MarkUp/Forms/2003/xforms-for-html-authors.html`.

[58] PHP. *Php*. `http://www.php.net`.

[59] M. Potel. *Mvp: Model-view-presenter, the taligent programming model for C++ and Java*. 1996. `http://www.wildcrest.com/Potel/Portfolio/mvp.pdf`.

[60] Prototype. *Prototype JavaScript framework*. `http://www.prototypejs.org/`.

[61] QueryTemplates. *Querytemplates*. `http://code.google.com/p/querytemplates/`.

[62] T. M. H. Reenskaug. *Models - Views - Controllers*. Technical report, Xerox PARC, Palo Alto, CA, 1979.

[63] C. Ruppel, J. Konecny. The role of is personnel in web-based systems development: the case of a health care organization. En *SIGCPR '00: Proceedings of the 2000 ACM SIGCPR conference on Computer personnel research*, pp. 130–135. ACM, New York, 2000.

[64] G. Seshadri. *Understanding JavaServer Pages Model 2 architecture*. 1999. `http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html`.

[65] Smarty. *Smarty template engine*. `http://smarty.php.net`.

[66] K. Smith. *Zope Page Templates - ZPT*. `http://zpt.sourceforge.net/`.

[67] Spring Source. *Spring MVC*. `http://www.springsource.org/`.

[68] Sun Microsystems. *Java Servlet Technology*. `http://java.sun.com/products/servlet/`.

[69] Sun Microsystems. *JavaServer Pages Standard Tag Library*. `http://java.sun.com/products/jsp/jstl/`.

[70] Sun Microsystems. *JavaServer Pages Technology*. http://java.sun.com/products/jsp.
[71] Sun Microsystems. *JavaBeans 1.01 specification*. 1997. http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html.
[72] M. Tatsubori, T. Suzumura. HTML templates that fly: a template engine approach to automated offloading from server to client. En *WWW '09: Proceedings of the 18th international conference on World wide web*. ACM, New York, 2009.
[73] M. Termens, M. Ribera, M. Porras, M. Boldú, A. Sulé, P. Paris. Web content accessibility guidelines: from 1.0 to 2.0. En *WWW '09: Proceedings of the 18th international conference on World wide web*, pp. 1171–1172. ACM, New York, 2009.
[74] P. Thiemann. An embedded domain-specific language for type-safe server-side web scripting. *ACM Trans. Internet Technol.* **5**(1), 1–46, 2005.
[75] S. Tregar. *HTML-Template*. http://search.cpan.org/dist/HTML-Template/.
[76] Trimpath. *JavaScript templates*. http://code.google.com/p/trimpath/wiki/JavaScriptTemplates.
[77] W3C. *Document object model*. http://www.w3.org/DOM/.
[78] W3C. *Web Content Accessibility Guidelines (WCAG) 1.0*. 1999. http://www.w3.org/TR/WAIWEBCONTENT/.
[79] W3C. *Transformations (XSLT) version 2.0*. 2007. http://www.w3.org/TR/xslt20/.
[80] W3C. *Web Content Accessibility Guidelines (WCAG) 2.0*. 2008. http://www.w3.org/TR/WCAG20/.
[81] Walt Disney Internet Group. *Tea template language*. 2001. http://teatrove.sourceforge.net/docs/TeaTemplateLanguage.pdf.
[82] WebMacro. *Webmacro*. http://sourceforge.net/projects/webmacro/.
[83] P. Wijkman, S. Dissanaike, M. Wijkman. Mixer, supporting the Model-View-Controller design pattern in servlets. En *International Conference on Software Engineering SE 2004*, pp. 658–661. IASTED, 2004.
[84] Wikipedia. *Cross-site scripting*. http://es.wikipedia.org/wiki/Cross-site_scripting.
[85] Wikipedia. *Web application framework. Push-based vs. Pull-based*. http://en.wikipedia.org/wiki/Web_application_framework#Push-based_vs._Pull-based.
[86] Xucia. *Authenteo*. http://www.authenteo.com.
[87] C. Yue, H. Wang. Characterizing insecure JavaScript practices on the web. En *18th International World Wide Web Conference*, pp. 961–961, 2009.
[88] J. Zhang, J. Y. Chung. Mockup-driven fast-prototyping methodology for web application development. *Software: Practice and Experience* **33**(13), 1251–1272, 2003.

Departamento de Matemáticas y Computación, Universidad de La Rioja, Spain
Correo electrónico: francisco.garcia@unirioja.es

Departamento de Matemáticas y Computación, Universidad de La Rioja, Spain
Correo electrónico: doradodos@hotmail.com