El metamodelado de un *framework*: Spring Web Flow

A. M. Reina Quintero
Dpto. Lenguajes y Sistemas Informáticos
ETS Ingeniería Informática
Univ. de Sevilla
41012 Sevilla
reinaqu@lsi.us.es

J. Torres Valderrama
Dpto. Lenguajes y Sistemas Informáticos
ETS Ingeniería Informática
Univ. de Sevilla
41012 Sevilla
jtorres@lsi.us.es

M. Toro Bonilla
Dpto. Lenguajes y Sistemas Informáticos
ETS Ingeniería Informática
Univ. de Sevilla
41012 Sevilla
mtoro@lsi.us.es

Resumen

Para sobrevivir en el mercado, las compañías han de adaptarse de forma rápida a los cambios de tecnología. Esta rápida evolución ha provocado que los periodos de lanzamiento de nuevas versiones de productos se acorten cada vez más. Esta circunstancia es más destacable si cabe, en el caso del software libre, ya que se cuenta con la inestimable y más frecuente ayuda del usuario, gracias a un contacto más directo con el mismo. Este artículo se centra en una de las etapas definidas en [15], una propuesta basada en MDA para mejorar la adaptación de un producto a las distintas versiones del software utilizado en su desarrollo. Así, el artículo describe la etapa de metamodelado del *framework*, tomando como caso de estudio, Spring Web Flow (SWF), un *framework* que se utiliza para definir los flujos de interfaces de usuario de aplicaciones web. Además del metamodelo de SWF, en el artículo también se propone una notación gráfica para definir flujos basados en esta herramienta. Así mismo, se proporciona un ejemplo de empleo de la notación.

1. Introducción

La rápida evolución de la tecnología hace que las compañías, para mantenerse en el mercado, acorten cada vez más los periodos de tiempo necesarios para lanzar nuevas versiones de sus productos. Muchas veces, esta versiones ofrecen características nuevas y mejoradas, pero también provocan la incompatibilidad con las versiones anteriores. Estos plazos se recortan aún más en el caso de las aplicaciones de software libre, ya que éstas recogen mucha más información por parte de los usuarios acerca de los errores y posibles mejoras debido, puesto que tienen un contacto más directo con los mismos, gracias a foros y otros mecanismos de comunicación.

El lanzamiento constante de nuevas versiones de los *frameworks* provoca una reacción en cadena, de tal forma que las incompatibilidades que se producen con versiones anteriores afectan a todo el software desarrollado con ese *framework*. Este proceso de lanzamiento de nuevas versiones de un producto puede considerarse como un proceso de evolución del software.

Hay diversas técnicas para tratar con la evolución del software que van de métodos formales a soluciones *ad-hoc*. Entre estas técnicas nos podemos encontrar con: reingeniería [20], técnicas de análisis de impacto [18, 21], técnicas relacionadas con la teoría de categorías [1, 19], o soporte a la evolución automatizada[10].

MDA [11] también se presenta como una filosofía prometedora para tratar la problemática de la evolución del software, no solo porque permite la separación de los conceptos del dominio de los conceptos tecnológicos, sino también porque la información de diseño y las

dependencias están explícitas en los modelos y las transformaciones, respectivamente.

En [15] se presenta una propuesta basada en MDA para mejorar la adaptación del software a distintas versiones de la misma plataforma. Como caso de estudio se escoge Spring Web Flow (SWF) [3], un *framework* que permite realizar la definición de flujos de interfaces de usuario en las aplicaciones web. En [15] se muestra como evoluciona este *framework* de la versión SWF 1.0 RC1 a la versión, más estable, SWF 1.0. Dos versiones que fueron lanzadas con apenas seis meses de diferencia. La propuesta presentada en este artículo se puede considerar una fase en el marco de esta propuesta general, el metamodelado del *framework*.

Se ha escogido Spring Web Flow porque es un *framework* que no maneja muchos conceptos, y que además, permite la definición de los flujos tanto en formato de clases Java, como utilizando un archivo de configuración XML, lo que facilita la tarea de la definición de transformaciones de modelo a texto, y la abstracción de los conceptos fundamentales que se manejan.

El resto del artículo está estructurado como sigue: En primer lugar se da una visión general de Spring Web Flow. Luego, se define el metamodelo del mismo. Más tarde, en la sección 4, se describe la notación visual propuesta para la definición de flujos. Posteriormente, se aplica esta notación a un ejemplo concreto, para finalmente, analizar los trabajos relacionados y concluir el artículo apuntando las líneas futuras de investigación.

2. Spring Web Flow (SWF)

Spring Web Flow (SWF) [3] es un *framework*, situado por encima de *Spring Framework* [9], que permite la definición y representación de flujos de interfaces de usuario en las aplicaciones web de una forma clara y simple. Un *flujo* define un diálogo con el usuario, de tal forma que las respuestas a los eventos producidos por el usuario durante el diálogo van guiando la ejecución del código de la aplicación.

En SWF los flujos se pueden definir de forma declarativa gracias a un lenguaje específico de dominio (LED). SWF permite trabajar con este lenguaje tanto con archivos XML como a través de un conjunto de clases Java. Un flujo está formado por un conjunto de uno o más estados. Cada *estado* define un paso dentro del flujo del ejecución del diálogo, de tal forma que al entrar en cualquier estado se ejecuta un comportamiento asociado al mismo. Este comportamiento dependerá tanto de la configuración como del tipo de estado en el que se entra. Un *evento* causará una transición de estados, así, por ejemplo, cuando el usuario pulsa un botón `Enviar`, causará el lanzamiento de un evento de tipo *submit*. Este evento tendrá una correspondencia con una transición que provocará un cambio de estado, por ejemplo, al estado `InformacionEnviada`. Así, en SWF se definen cinco tipos de estados:

ViewState Es el estado que permite al usuario (o a un cliente externo) participar en el flujo. Cuando el flujo entra en un estado de este tipo, se detiene, y se devuelve el control al sistema que lo invocó, y que será el encargado de mostrar una interfaz al usuario. Después, el usuario, mediante la interacción con la interfaz, producirá un evento que provocará la reanudación de la ejecución del flujo.

ActionState Este tipo de estado es el utilizado para ejecutar código con la lógica de negocio de la aplicación. El resultado de la ejecución de este código decidirá cuál será el siguiente estado en el que entrará el flujo. La *acción* es la abstracción que se utiliza para encapsular la ejecución de una unidad lógica del código de la aplicación. Cuando el flujo entra en un estado de tipo `Acción`, se van ejecutando en orden un conjunto de acciones. Si el resultado de la salida de la primera acción ejecutada concuerda con una transición, entonces se ejecuta la transición. En caso contrario, se ejecutará la siguiente acción de la lista.

DecisionState Este estado permite cambiar la ruta del flujo dependiendo de una decisión. Cuando el flujo entra en un estado

de tipo `Decisión`, se evalúan un conjunto de expresiones booleanas. El resultado de la evaluación decidirá cuál será el siguiente estado en el que entrar.

SubflowState Con este tipo de estado se permite la reutilización de un flujo ya existente, de tal forma que, cuando se entra en un estado de esta clase, se ejecuta un nuevo flujo como subflujo del anterior. Al subflujo se le pueden pasar parámetros de entrada y puede devolver una serie de valores de retorno.

EndState Es el estado que indica la terminación del flujo. Así que, cuando un flujo entra en un estado de tipo `Final`, termina su ejecución. Si el flujo estaba actuando como subflujo, la sesión termina y continúa la ejecución del flujo padre.

3. Un metamodelo para SWF

En esta sección se describe el metamodelo MOF [14] para definir los flujos de Spring Web Flow a nivel específico de plataforma (Figura 5 del apéndice). Aunque la implementación final del metamodelo se ha realizado utilizando EMF [2, 6], en el artículo se presenta la versión MOF, que es una notación más extendida y con la que posiblemente esté más familiarizado el lector. De cualquier manera, se puede definir una equivalencia entre ambos lenguajes de metamodelado([13, 7, 5]), de tal forma, que es posible obtener un metamodelo EMF a partir de uno MOF, y viceversa.

La pieza central del metamodelo es la metaclase `Flow`, que representa un flujo de navegación entre páginas. Un flujo estará compuesto por una serie de estados, que en el diagrama MOF se modelan mediante la metaclase `State`. Un flujo tiene solamente un estado inicial (en el que comienza la ejecución del mismo), pero puede tener más de un estado final.

Los estados se pueden clasificar en estados a partir de los cuales pueden partir transiciones (`TransitionableState`), y estados sin transiciones salientes. En este caso, los estados finales (de tipo `EndState`) son los únicos de los que no parte ninguna transición.

Los estados con transiciones salientes pueden ser o bien estados de acción (`ActionState`), o bien estados de tipo vista (`ViewState`), o subflujos (`SubflowState`), o estados de decisión (`DecisionState`). Cada una de estas metaclases representa a uno de los tipos de estado disponible en SWF. La semántica de cada tipo de estado se comentó en la sección 2, dedicada a introducir al *framework* para la definición de flujos.

Una transición (`Transition`), asociada a un evento, hace que un flujo cambie de un estado a otro. Cualquier estado de tipo traspasable, es decir, del que pueden partir transiciones, define un conjunto de una o más transiciones que indican un camino hacia otro estado.

Una transición puede ser una transición global, condicional, de un estado, dinámica, o de excepción. Una transición global (`GlobalTransition`) está relacionada con el flujo, y se puede activar en cualquier estado del mismo. Si una transición se puede producir en todos los estados traspasables del flujo, entonces se define como una transición global. Se puede decir que con este tipo de transición se proporciona un mecanismo para no tener que repetir por cada estado la misma información.

Una transición condicional (`ConditionalTransition`) solamente puede tener como origen a un estado de desición (`DesicionState`), y su activación dependerá del resultado de evaluar una condición booleana. Este tipo de transición es el único que puede tener definidos dos estados destino. El encaminar el flujo hacia un estado u otro dependerá del resultado de la evaluación de la condición mencionada anteriormente.

Las transiciones modeladas con `StateTransition` se disparan cuando se produce el evento asociado a la transición. Es el tipo de transición que se define en cualquier máquina de estados. Sin embargo, una transición dinámica (`DynamicTransition`) representa una transición cuyo estado destino se calcula en tiempo de ejecución. Por último, una transición de excepción es una transición que tiene la característica de ser disparada cuando se produce una excepción. Es decir, el evento que provoca su lanzamiento es del tipo *"se ha producido tal*

excepción".

Una acción (`Action`) representa a una funcionalidad a ejecutar en un punto determinado del flujo. Los puntos en los que se pueden ejecutar acciones son: al iniciar el flujo; cada vez que se entra en un estado; justo antes de que se produzca el cambio de estado debido a que se dispara una transición; antes de dibujar un formulario o vista; cuando se sale de un estado, y cuando se termina el flujo. Esta semántica se ha modelado mediante las relaciones de composición entre `Action` y `Flow`, `Action` y `ViewState`, `Action` y `State`, y `Action` y `TransitionableState`, respectivamente.

Además de estos puntos, las acciones también se pueden asociar a los estados de tipo acción (`ActionState`), ya que es la forma de definir la lógica de negocio que se ejecuta en los mismos. Las acciones que se realizan antes de dibujar un formulario o vista, solamente se pueden ejecutar en estados de tipo `ViewState`, y serán acciones relacionadas con el *renderizado*, como por ejemplo, la configuración de los datos de un formulario.

Además de agrupar las acciones atendiendo a los puntos concretos donde se pueden lanzar, éstas se pueden clasificar en los siguientes tipos: `SimpleAction`, `EvaluationAction`, `BeanAction` y `SetAction`. La acción simple se utiliza para encapsular acciones ligeras. Una acción de tipo `EvaluationAction` se utiliza cuando se quiere o bien invocar a un *bean* en un flujo, o bien evaluar alguna expresión de un flujo. Una acción de tipo *bean* se utiliza para invocar a un método de un objeto de los conocidos como tradicionales (lo que en la bibliografía se conoce como *Plain Old Java Object* o POJO). Por último, las acciones de tipo `SetAction` se utilizan para asignar atributos en el flujo.

Además de estas clases, existen otras que se pueden considerar como secundarias, porque no expresan los conceptos centrales que aporta el *framework*, y que se utilizan para definir casos muy específicos en los flujos. La metaclase `Attribute` se utiliza para definir atributos en un flujo, un estado o una transición. `ExceptionHandler` modela un manejador de excepciones asociado bien a un flujo, bien a un estado. `Variable` expresa una variable que se puede crear cuando se crea un flujo, y `Argument` modela un argumento de un método definido en una acción.

Finalmente, la metaclase `AttributeMapper` representa una correspondencia entre los atributos de un flujo y un subflujo. Se utiliza para modelar el hecho de que cuando se lanza un subflujo desde un flujo, a éste se le pueden pasar parámetros de entrada. Igualmente, cuando el subflujo termina, puede devolver información mediante parámetros de salida. Mediante `AttributeMapper` se define la correspondencia entre los atributos del flujo y del subflujo. Como tal, solamente estará relacionada con `SubflowState`.

4. Modelado Visual

Aunque el modelado visual sea un handicap a la hora de la evolución del lenguaje de modelado, y de que los programas se ajusten a las distintas versiones de los *frameworks*, se ha definido un lenguaje visual para representar los flujos de Spring Web Flow. El lenguaje visual está basado en la notación que define UML para los diagramas de estado, con las siguientes peculiaridades o adaptaciones:

Se han creado cinco estereotipos asociados a los estados definidos en los diagramas de estado UML. Cada uno de ellos representa uno de los estados tipo definidos en Spring Web Flow (Figura 1). Cada estado se representa, por tanto, por un rectángulo con las esquinas redondeadas y su correspondiente estereotipo. La única excepción son los estados finales que se representan con dos círculos concéntricos, de los cuales, el más interno aparece relleno. Algunos de estos estados (`ViewState`, `ActionState` y `SubflowState`), se representan con el rectángulo dividido en dos partes. En la parte superior se muestra el nombre del estado y el estereotipo, que representa el tipo de estado, mientras que en la parte inferior se indica, mediante un estereotipo, la propiedad más representativa del mismo. Así, en los estados tipo vista se representa la vista (`<<view>>`), en los de acción, la acción(`<<action>>`), y finalmente, en los subflujos (`<<subflow>>`), el

subflujo al que se accede. El estado final también puede tener asociada una vista, que es la pantalla o interfaz que se muestra al terminar el flujo.

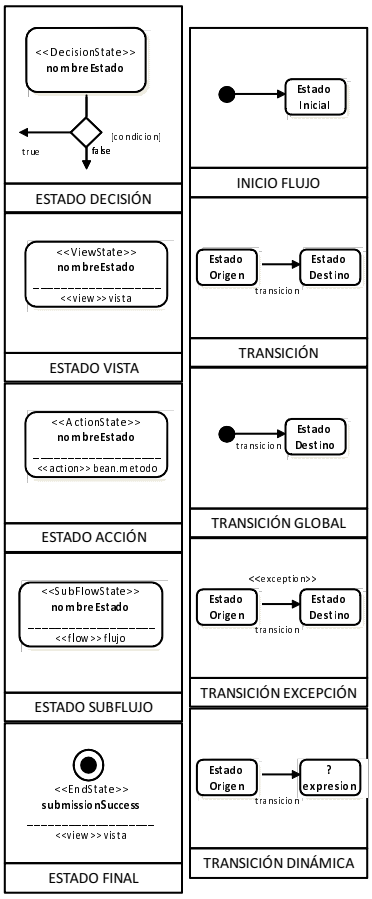

Figura 1: Tipos de estados y de transiciones definidos en Spring Web Flow

En la Figura 1 también se representan los distintos tipos de transiciones que nos podemos encontrar. Así, una transición normal se representa con una flecha que parte del estado origen y se dirige hacia el estado destino, acompañada por una etiqueta que da nombre a la transición. Una transición global se representa por una flecha que parte de la marca de inicio del flujo (un círculo pequeño relleno)

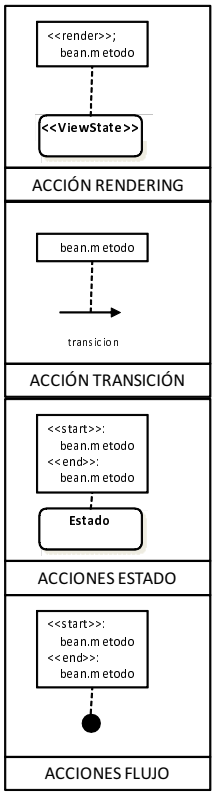

Figura 2: Acciones en distintos puntos del flujo

hacia el estado destino, acompañada también de una etiqueta que indica el nombre de la transición. Una transición de tipo excepción se representa igual que una transición normal estereotipada con `<<exception>>`. Una transición dinámica se representa igual que una transición normal, con la diferencia de que en este caso, el rectángulo que representa el estado destino contiene un signo de interrogación (?) y una expresión que sirve para calcular cuál será el estado destino. Finalmente, y aunque no esté representada en la columna derecha de la Figura 1, una transición condicional se representa por un rombo hueco del que parten dos bifurcaciones, una con la etiqueta `true`, y otra con la etiqueta `false`. El rombo va acompañado de una etiqueta que contiene la

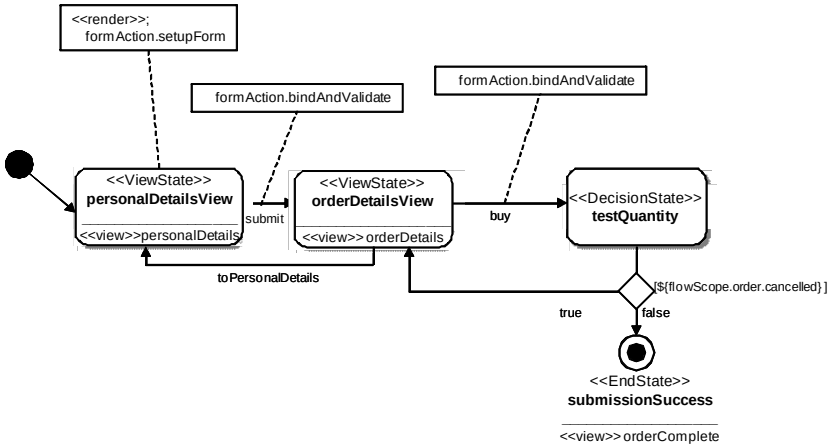

Figura 3: Flujo simple para una aplicación de compra virtual

condición a evaluar y que determinará el estado final. La rama `false` es opcional, es decir, puede no aparecer. Este tipo de transición se puede ver en el estado de decisión dibujado en la columna de la izquierda.

Una acción o conjunto de acciones se representa por un rectángulo acompañado de una línea discontinua por la que queda unida a otro elemento del flujo (Figura 2). Así, se puede asociar a un estado (Figura 2, tercer recuadro), a una transición (Figura 2, segundo recuadro), o a la marca de inicio del flujo (Figura 2, cuarto recuadro). En este último caso, se representa a una acción a nivel del flujo. Dentro de la lista de acciones pueden aparecer varios estereotipos. En las acciones asociadas a los estados de tipo vista (Figura 2, primer recuadro) puede aparecer `<<render>>`, y en las acciones asociadas al flujo y a los estados pueden aparecer bien `<<start>>`, bien `<<end>>`. Para expresar acciones que se han de ejecutar al entrar o al salir del flujo o el estado, respectivamente.

5. Un ejemplo de modelado

En la Figura 3 se representa un flujo para una aplicación de compra de productos sencilla, adaptada de [4]. En él se definen cuatro estados. Los dos primeros estados representan dos vistas correspondientes a los datos que el usuario tiene que rellenar. En primer lugar, se le pide que rellene sus datos personales, para más tarde solicitarle los datos propios del pedido a realizar. Luego se llega a un estado de decisión en el que se evalúa si el usuario ha cancelado la operación a realizar. Si la respuesta es afirmativa, se le vuelven a solicitar los datos del pedido, mientras que si la respuesta es negativa, se llegará al estado final, que hará que el flujo termine y mostrará la información del pedido completado.

Gracias a la implementación del metamodelo de la Figura 5 en EMF (*Eclipse Modeling Framework*) [2], y a la definición de una serie de transformaciones de modelo a texto mediante la herramienta MOFScript [12], se genera el archivo XML mostrado en la Figura 4, que contiene la definición del flujo lista para ser tomada como entrada en cualquier programa que utilice SWF.

6. Trabajos relacionados

Los trabajos relacionados se pueden englobar en dos categorías. En primer lugar, aquellos que están relacionados con el *framework* Spring Web Flow. En segundo lugar, y situados en un marco más general, aquellos rela-

```
01 <?xml version="1.0" encoding="UTF-8"?>
02    <flow xmlns="http://www.springframework.org/schema/webflow"
03          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04          xsi:schemaLocation="
05              http://www.springframework.org/schema/webflow
06              http://www.springframework.org/schema/webflow/spring-webflow-1.0.xsd">

07      <start-state idref="personalDetailsView"/>

08      <view-state id="personalDetailsView" view="personalDetails">
09              <render-actions>
10                      <action bean="formAction" method="setupForm"/>
11              </render-actions>

12              <transition on="submit" to="orderDetailsView">
13                      <action bean="formAction" method="bindAndValidate">
14                              <attribute name="validatorMethod" value="validatePersonalDetails"/>
15                      </action>
16              </transition>
17      </view-state>

18      <view-state id="orderDetailsView" view="orderDetails">
19              <transition on="buy" to="testQuantity">
20                      <action bean="formAction" method="bindAndValidate">
21                              <attribute name="validatorMethod" value="validateOrderDetails"/>
22                      </action>
23              </transition>
24              <transition on="toPersonalDetails" to="personalDetailsView"/>
25      </view-state>

26      <decision-state id="testQuantity">
27              <if test="${flowScope.order.cancelled}"
28                      then="orderDetailsView"
29                      else="submissionSuccess"/>
30      </decision-state>

31      <end-state id="submissionSuccess" view="orderComplete"/>
32 </flow>
```

Figura 4: Definición de flujo XML correspondiente al diagrama de estados de la Figura 3

cionados con la propuesta general en la que está enmarcado el trabajo de este artículo.

En lo concerniente a un lenguaje visual de modelado para Spring Web Flow, debemos mencionar la implementación que se está realizando dentro del marco del proyecto Spring IDE [16]. La principal diferencia de este lenguaje con nuestra propuesta es que éste no se ajusta a ningún tipo de notación estándar, mientras que en este artículo se propone una notación derivada de UML, que será más familiar a muchos diseñadores relacionados con este lenguaje de modelado.

Si nos centramos en el marco general de la propuesta, considerando lo presentado en este artículo como una etapa a la hora de controlar la evolución de un *framework*, podemos citar el trabajo de Ivkovic y Kontogiannis [8], en el que se propone un *framework* para sincronizar artefactos software utilizando transformaciones de modelos. Los artefactos software se han de expresar mediante modelos y metamodelos MOF, y la propuesta se centra en la trazabilidad de los cambios de los modelos como una forma de realizar la sincronización entre los mismos.

En [17], se da un proceso sistemático y basado en MDA para el mantenimiento de *middlewares*. Además se hace una clasificación de los distintos tipos de *middlewares*. SWF podría ser considerado como parte de un *middleware* corporativo, atendiendo a esta clasificación.

7. Conclusiones y trabajos futuros

En este artículo se ha detallado uno de los pasos definidos en [15]. En concreto, el metamodelado de un *framework*. Se ha escogido Spring Web Flow porque define un Lenguaje Específico de Dominio (LED) con un conjunto de conceptos no demasiado grande. Además, proporciona la opción de definir los flujos tan-

to en formato de clases Java como en un archivo XML, lo que simplifica mucho la definición de las transformaciones modelo-texto.

Además de definir un metamodelo para SWF, en el artículo se propone también una notación para el modelado visual de los flujos basada en UML. Para dejar más clara la notación, se utiliza un ejemplo de un flujo.

El metamodelo se ha implementado en EMF, y se han definido una serie de transformaciones modelo-texto utilizando MOFScript para generar archivos XML que puedan ser utilizados en cualquier aplicación que use SWF.

Uno de los trabajos futuros pendientes es la realización de un repositorio de metamodelos, disponibles en la web. De tal forma que se pueda contar con los metamodelos de las distintas versiones de los *frameworks* para que puedan servir como ayuda a la hora de adaptar el software de una versión a otra.

De la misma forma, se pretende seguir avanzando en las etapas definidas en la propuesta más general ([15]).

8. Apéndice

Para no cortar la continuidad del artículo, el metamodelo de SWF se ha incluido como un apéndice al final del mismo.

Referencias

[1] Barr, M., Wells, C. *Category Theory for Computing Science*, Prentice-Hall, 1990.

[2] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T. J. *Eclipse Modelling Framework: A Developer's Guide*. Addison-Wesley. 2003

[3] Donald, K., Vervaet, E. *The Spring Web Flow Home Page*. Avalaible at: `http://opensource.atlassian.com/confluence/spring/display/WEBFLOW/Home`

[4] Devijver, S. *Spring Web Flow Examined*. JavaLobby. Avalaible at: `http://www.javalobby.org/articles/spring-webflow/`.

[5] Duddy, K., Gerber,A., Raymond, K. *Eclipse Modelling Framework (EMF) import/export from MOF / JMI*. Technical Report,Queensland University of Technology. 2003.

[6] Eclipse Project. *Eclipse Modeling Framework*. Avalaible at: `http://eclipse.org/emf`.

[7] Gerber,A., Raymond, K. *MOF to EMF: There And Back Again*. Proc. Eclipse Technology Exchange Workshop, OOPSLA 2003, Anaheim, USA, Oct 2003, pp 6-70.

[8] Ivkovic, I., Kontogiannis, K. *Tracing Evolution Changes of Software Artifacts through Model Synchronization*. Proc. of the 20th IEEE International Conference on Software Maintenance (ICSM'04). 2004.

[9] Johnson, R. et al. *The Spring Framework - Reference Documentation*. Avalaible at: `http://static.springframework.org/spring/docs/2.0.x/spring-reference.pdf`

[10] Mens, T., Tourwé T. *A Declarative Evolution Framework for Object-Oriented Design Patterns*, Proc. International Conference on Software Maintenance, pp. 570-579, Nov, 2001.

[11] *MDA Guide Version 1.0.1*, OMG/03-06-01. www.omg.org/mda.

[12] MOFScript Team. *The MOFScript Home Page*. Avalaible at: `http://www.eclipse.org/gmt/mofscript/`.

[13] Mohamed, M., Romdhani, M., Ghedira, F. K. *MOF-EMF Alignment*. Proceedings of the Third International Conference on Autonomic and Autonomous Systems (ICAS'07). pp. 1. IEEE Computer Society. ISBN:0-7695-2859-5.

[14] OMG. *Meta Object Facility Specification Version 2.0*. 2006. Available at:`http://www.omg.org/docs/formal/06-01-01.pdf`

[15] Reina Quintero, A. M., Torres Valderrama, J., Toro Bonilla, M. *Improving the Adaptation of web applications to different versions of software with MDA*. 7th International Conference on Web Engineering. Workshop Proceedings. Editors: Marco Brambilla, Emilia Mendes. pp. 101-107. ISBN: 978-88-902405-2-2.

[16] Spring IDE Team. *Spring IDE Web Page*. Available at:`http://springide.org`.

[17] Wadsack, J. P., Jahnke, J. H. *Towards Model-Driven Middleware Maintenance* In Proc. Int. Workshop on Generative Techniques in the context of Model-Driven Architecture held with OOSPLA'02. 2002.

[18] Weiser M. *Program Slicing: Formal, Psychological and Practical Investigation of an Automatic Program Abstraction Method. Phd. Thesis*, University of Michigan, 1979.

[19] Wiels, V., Easterbrook, S. *Management of Evolving Specifications Using Category Theory*, Automated Software Engineering, pp.12-21, 1998.

[20] Yang, H., Ward, M. *Successful Evolution of Software Systems*, Artech House Computing Library, 2003. ISBN: 1-58053-349-3

[21] Zhao, J., Yang, H., Xiang, L., Xu, B. *Change Impact Analysis to Support Architectural Evolution*, Journal of Software Maintenance and Evolution, 14(5), pp.317-333, 2002. ISSN:1040-550X.John Wiley & Sons, Inc.