VNIVERSIDAD NACIONAL AVTNOMA DE MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

"DESARROLLO DE UN SISTEMA DE ADMINISTRACIÓN DE PROCESOS EN PLONE"

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS (COMPUTACIÓN)

P R E S E N T A:

EDUARDO ESPINOSA AVILA

DIRECTOR DE TESIS: SERGIO RAJSBAUM GORODEZKY

MÉXICO, D.F. **2009.**

Agradecimientos.

Índice general

Resumen.

A fin de llevar la delantera, muchas de las compañías más grandes y mejor organizadas del mundo han adoptado implementar sistemas automatizados para administrar sus procesos principales, esta tesis analiza de forma general la manera de desarrollar este tipo de sistemas. Se estructura fundamentalmente de tres partes:

La primera de ellas consiste en una revisión de la problemática detectada en el Instituto de Matemáticas de la Universidad Nacional Autónoma de México, así como una propuesta de solución, dando una revisión al marco de trabajo seleccionado para implementar dicha solución.

En la segunda parte se presenta una investigación cuyo objetivo es analizar el estado del arte de los sistemas manejadores de *workflows*, se realizó una revisión profunda de diferentes propuestas desde diversos puntos de vista, tanto teóricos como prácticos; su relación con procesos de negocio y cómo facilitan la implementación de sistemas de tipo administrativo.

Por último se muestra la forma en la que se dio solución al problema planteado, pasando diversas etapas, desde el diseño, hasta la del producto desarrollado, haciendo hincapié en la utilización de la tecnología seleccionada para este fin; finalmente se exponen las conclusiones obtenidas y el trabajo futuro que puede realizarse.

1. Introducción.

1.1 Antecedentes.

El Instituto de Matemáticas (IMATE) de la Universidad Nacional Autónoma de México (UNAM) es una institución reconocida a nivel internacional por su alto nivel académico; es responsabilidad de la misma mantenerse a la vanguardia académica y tecnológicamente, por lo que constantemente busca mejorar la automatización de sus sistemas de información.

El IMATE inició sus actividades el 30 de junio de 1942 y con ello también comenzó el desarrollo moderno de las matemáticas en nuestro país, que en la actualidad se distingue por haber consolidado la investigación en el área. Además de sus instalaciones en CU, el IMATE cuenta con una sede en Morelia, y otra en Cuernavaca, así como una representación en Oaxaca.

La Secretaría Académica es el auxiliar de la Dirección para lograr el funcionamiento adecuado del Instituto. Sus funciones y atribuciones, entre otras, son [1]:

- ✓ Colaborar con el Director para establecer los planes de trabajo del Instituto y los programas adecuados para llevarlos a cabo
- ✓ Auxiliar al Director en la elaboración del anteproyecto de presupuesto.
- ✓ Auxiliar al Director en la elaboración del Informe Anual de Actividades del Instituto.

- ✓ Auxiliar al Director en la supervisión de las labores académicas y administrativas del Instituto.
- ✓ Suplir al Director en ausencia de éste.
- ✓ Coordinar los servicios de apoyo con que cuenta el Instituto.
- ✓ Coordinar las comisiones auxiliares del Consejo Interno.

Al mismo tiempo, en la secretaría académica del IMATE se controlan y manejan diversos procesos administrativos que involucran al personal académico de dicho instituto. El departamento de cómputo, a cargo de la secretaría académica tiene, entre otras, las responsabilidades siguientes [2]:

- ✓ Mantener una red robusta, eficiente y segura en el instituto de matemáticas.
- ✓ Dar servicio a todas las máquinas que estén integradas a la red, es decir, aquellas que cuenten con el mismo sistema operativo que el que se encuentra en los servidores del instituto.
- ✓ Dar servicio al servidor del departamento de publicaciones del Instituto.
- ✓ Instalar y mantener actualizados los programas que los investigadores usan en red para el desarrollo de su investigación, esto sujeto a que haya el presupuesto necesario.
- ✓ Ofrecer una página de web del instituto que sea eficiente y que siempre esté actualizada.

Para este último punto, se inició el proyecto InfoMatem. InfoMatem [1] es un sitio accesible desde el web donde se almacena la información académica del IMATE de la UNAM, de manera que sea fácilmente accesible y fácil de mantener actualizada.

El proyecto inicia a mediados del 2006 con Sergio Rajsbaum, Secretario Académico como responsable, Mónica Leñero como coordinadora, Gildardo Bautista como administrador del sitio y programador de su estructura básica, Adriana Ramírez, Marco López, Ernesto Badillo y Alexander Zapata apoyando en programación de herramientas.

InfoMatem está programado sobre Plone, que es un sistema de administración de contenidos, robusto y maduro sobre el cual muchos otros sitios tanto académicos como comerciales se han construido. Es un sistema que sigue estándares internacionales de internet y web, de *software* libre y abierto.

1.2 Planteamiento del problema.

En la actualidad, los procesos administrativos del IMATE se realizan en papel y de forma presencial, por ejemplo, para realizar una solicitud un académico debe descargar un formato de la página de InfoMatem, imprimirlo, llenarlo, reunir una serie de documentos necesarios y después entregarlo en la Secretaría Académica, dónde se le sella una copia como acuse de recibo; posteriormente, la solicitud es revisada por un representante de la Comisión Especial quién después de analizarla, añade comentarios y la turna al Consejo Interno con una recomendación de aprobar o volver a revisar; finalmente en una reunión, el Consejo Interno

revisa la solicitud y, en su caso aprueba o no la solicitud, informando al solicitante la resolución que tomaron respecto a su solicitud.

Esta forma de trabajo presenta algunos problemas, entre los que destacan:

- ✓ Es necesario imprimir y sacar copias de todas las solicitudes realizadas, provocando desperdicio de recursos.
- ✓ Es necesario que el usuario físicamente acuda a entregar su solicitud.
- ✓ Una vez entregada la solicitud, el usuario no puede tener seguimiento de la misma.
- ✓ Al tenerse movimiento físico de papelería, en algún punto del proceso, podría perderse o traspapelarse la solicitud o alguno de los documentos que lo acompañan.
- ✓ Es difícil y tedioso mantener un historial actualizado de las solicitudes realizadas.
- ✓ Como resultado del punto anterior, resulta también complicado realizar búsquedas de solicitudes, y llevar la cuenta del dinero gastado por cada académico.
- ✓ El tiempo que toma dar respuesta a algunas solicitudes puede resultar muy grande.

1.3 Propuesta de solución.

Se propone realizar un sistema sobre Plone, que es considerado uno de los mejores *Sistemas manejadores de contenido* (*CMS*) disponibles hoy en día [3].

La razón por la cual se eligió Plone y no otra tecnología, es porque Plone permite el desarrollo de aplicaciones de forma muy rápida debido a que existen muchos productos disponibles, dado que está programado sobre Python [13]. Además, Plone ya viene listo para usarse, sólo hay que personalizar algunos detalles; es decir, el manejo de usuarios, control de acceso, permisos y otras características subyacentes ya vienen implementadas y de forma eficiente, sólo habría que enfocarse en lo relacionado con el almacenamiento y recuperación de información [4].

Este sistema debe permitir, mediante interfaces amigables, que los académicos del IMATE realicen solicitudes desde cualquier equipo que disponga de un navegador Web y una conexión a Internet; además tendrán acceso la Comisión Especial y el Consejo Interno.

Para su fácil utilización, se tomarán como base los formatos actualmente utilizados, dado que la mayoría de los académicos están familiarizados con ellos, además, se mostrarán en pantalla el menor número de campos para no hacer tan tedioso el llenado del formato.

Existen diversos tipos de usuarios que actualmente utilizan InfoMatem y que realizarán sus solicitudes desde el mismo sistema, esto será tomado en cuenta para implementar correctamente el control al acceso de este tipo de contenido, dependiendo del estado del proceso en el que se encuentre, el sistema sólo permitirá que dicho contenido sea accedido (vista, edición ó borrado) sólo por aquellos usuarios que tengan ese permiso específico.

1.4 Objetivo General.

Implementar un sistema de administración de procesos sobre Plone, que controle tres procesos similares del Instituto de Matemáticas de la Universidad Nacional Autónoma de México; que pueda ser tomado como base para implementar otros procesos administrativos, no sólo del IMATE, sino también para otras dependencias universitarias que están mostrando interés en desarrollar algunas actividades con este CMS.

1.5 Objetivos Específicos.

Al desarrollar el sistema de administración de procesos, se desea:

- ✓ Evitar impresiones y copiados excesivos, a fin de reducir el desperdicio de recursos.
- ✓ Evitar que el usuario físicamente acuda a entregar su solicitud, pudiendo realizarla vía la página de InfoMatem.
- ✓ Que el usuario pueda dar seguimiento a su solicitud en todo lugar donde disponga de una computadora conectada a Internet.
- ✓ Evitar el movimiento físico de papelería, con esto también se evade la posibilidad de perder o traspapelar la solicitud o alguno de los documentos que lo acompañan.
- ✓ Mantener un historial actualizado de las solicitudes realizadas.
- ✓ Realizar búsquedas de solicitudes, y llevar la cuenta del dinero gastado por cada académico de forma fácil y eficiente.
- ✓ Reducir al mínimo el tiempo de respuesta a las solicitudes.
- ✓ Que el producto obtenido al final sea fácilmente reutilizable para modelar e implementar otros procesos administrativos, tanto del mismo IMATE como de otras dependencias de la UNAM.

1.6 Estructura de la tesis.

Los capítulos que conforman la tesis muestran las diferentes etapas del desarrollo del sistema desde el planteamiento del problema hasta la implantación del mismo, dando énfasis al estado del arte de *Workflows* y su utilización en Plone.

Capítulo 2. Marco de Referencia.

Este capítulo explica qué es un CMS, así como las herramientas sobre las cuáles funciona Plone, es decir el servidor Web Zope y la base de datos ZOBD, que a su vez están desarrolladas con el leguaje de programación Python.

Capítulo 3. Estado del Arte de *Workflows*.

En este capítulo se muestra el estado actual de los sistemas manejadores de *workflows*, desde sus conceptos básicos, su modelado y su representación, hasta algunas arquitecturas propuestas y una descripción de sistemas de *workflows* que utilizan agentes.

Capítulo 4. *Workflows* en Plone 3.

Este capítulo describe los *workflows* que vienen ya integrados a la versión 3 del CMS Plone, además, muestra la forma de crear *workflows* personalizados.

Capítulo 5. Análisis y diseño del producto *Solicitudes*.

En este capítulo se muestran los requerimientos solicitados, los diagramas de casos de uso, para tener una idea clara de lo que realizará el producto final y la arquitectura utilizada.

Capítulo 6. Implementación del producto *Solicitudes*.

Este capítulo muestra de forma más detallada la forma en la que se desarrolló el producto, presentando herramientas y productos auxiliares para el mismo.

Capítulo 7. Conclusiones.

En este capítulo se evalúan las conclusiones obtenidas y se explica el trabajo relacionado que puede realizarse.

2. Marco de Referencia.

2.1 Sistema manejador de contenido (CMS).

En los primeros años de la WWW los sitios Web consistían en gran parte de páginas con texto estático, ligas y un número limitado de imágenes. Todo esto era hecho principalmente con código escrito en HTML. Las páginas del sitio eran editadas directamente en el servidor de producción o enviadas por ftp a este servidor. Los sistemas para el manejo del contenido Web consistían básicamente de herramientas que permitían la producción de código HTML [5].

Con el surgimiento de los sistemas manejadores de contenido, la responsabilidad de la creación de contenido Web pasó a manos del creador del contenido. Con esto ya no es necesaria la separación en dos servidores, uno productivo y otro para desarrollo, sino que el contenido puede ser creado directamente en el servidor productivo con un proceso definido de publicación [4].

Para poder definir correctamente un CMS, es necesario distinguir algunos conceptos, Bob Boiko [6], proporciona las siguientes definiciones:

- ✓ **Datos**. Unidad mínima de información que representa de forma simbólica (numérica, alfabética, etc.) un atributo o característica de una entidad, y no tiene valor semántico por sí mismo.
- ✓ **Información**. Es en lo que los humanos convierten su conocimiento cuando lo quieren comunicar a otras personas. Este conocimiento puede ser visible, audible o escrito.
- ✓ **Contenido**. Información contenida dentro de un conjunto de metadatos clasificados.

El diccionario de la Real Academia Española define lo siguiente [7]:

- ✓ **Dato**. Antecedente necesario para llegar al conocimiento exacto de algo o para deducir las consecuencias legítimas de un hecho. *Inform.* Información dispuesta de manera adecuada para su tratamiento por un ordenador.
- ✓ **Información**. Comunicación o adquisición de conocimientos que permiten ampliar o precisar los que se poseen sobre una materia determinada. Conocimientos así comunicados o adquiridos.
- ✓ **Contenido**. Cosa que se contiene dentro de otra.

Además, Stephen R. G. Fraser define contenido como "*cualquier cosa*" que se encuentra en un sitio Web [8]. El contenido es información envuelta con datos que permiten definir y manipular la información. A estos datos se les conoce como metadatos, y prácticamente describen la información y le dan un contexto explícito de tal forma que la computadora la pueda manejar [4].

En [6] se definen los CMS como sistemas que permiten la creación, manejo, distribución y búsqueda de contenido. Cubren todo el ciclo de vida de las páginas de un sitio Web desde la creación y publicación del contenido hasta que es quitado y almacenado. Deben permitir que

este contenido sea de una gran variedad de formatos y tipos, sin limitarse sólo a HTML, XML, videos, audio, imágenes y documentos.

Un CMS puede tener tanto páginas estáticas, escritas en HTML, como dinámicas, utilizando alguna tecnología como JSP, ASP o PHP. Pero sin importar si el contenido es una página estática o es mostrada utilizando páginas dinámicas, para el sistema sigue siendo contenido con diferencias menores, como la forma en que se va a mostrar. De hecho, la mayoría de los sitios Web grandes utilizan tanto páginas estáticas como dinámicas.

En [9] se indica que la creación del contenido debe estar estrictamente separada de la presentación que se le va a dar. Así el creador del contenido no tiene que tener conocimiento técnico para diseñar o crear una página, y el diseñador no tiene que estar creando cada página que se quiere publicar. De esta forma se reduce la complejidad de las tareas realizadas por los actores envueltos en el proceso de publicación de contenido, además de la fricción entre éstos.

La creación del contenido incluye la agregación de metadatos a la información. Como se mencionó anteriormente, el contenido es información y metadatos. Estos metadatos permiten que el CMS pueda manejar el contenido. Ejemplos de metadatos pueden ser: creador del contenido, fecha de última modificación y tipo de contenido; además pueden definir la forma en la que se visualizará el contenido o su comportamiento.

Usualmente el almacenamiento del contenido es realizado por medio de un repositorio centralizado y soportado por varias herramientas que permiten su manejo y manipulación.

Estas herramientas pueden incluir: control de versiones, respaldos, recuperación, *workflow*, seguridad en la integridad del contenido, reportes, etc.

Los CMS incluyen una forma de definir la lógica de negocios llamada *workflow*, que es responsable de coordinar, programar y hacer cumplir ciertas tareas. El *workflow* se define en [9] como una serie de pasos que ocurren periódicamente en un CMS con el fin de coordinar las partes del sistema que se encuentran en movimiento, es decir, aquellas partes que se encuentran en estado cambiante. Por ejemplo, un proceso de publicación de contenido puede implicar varias etapas: la creación del contenido, revisión, aprobación y publicación. Cada una de estas etapas es un estado y el contenido en movimiento es aquel que cambia de un estado a otro.

Cada paso del *workflow* puede ser activado de diferentes formas, ya sea manualmente por el usuario (puede ser con un botón), al crear o borrar contenido, en cierta hora y fecha, al ocurrir un evento, etc. Además, cada paso puede tener tareas que permiten cambiar el estado de algún objeto. Siguiendo el ejemplo anterior, en el estado de revisión de un contenido se puede requerir que nadie más pueda acceder a éste sino sólo la persona que lo va a revisar y en el de publicación, que todos puedan acceder a éste.

Los CMS permiten la creación, edición, manejo y publicación de contenido a equipos variados de técnicos y no técnicos de forma tanto centralizada como descentralizada. Estas funciones son dirigidas mediante un conjunto de reglas, procesos y *workflows* centralizados, que aseguran un sitio Web válido y coherente [10].

Los Sistemas Manejadores de Contenido Web (WCMS), así como los CMS son parte del gran concepto de *Manejo de Contenido*. Un CMS es considerado WCMS si tienen su interfaz de control sobre Web o si la forma en que muestran y envían la información es sobre Web por medio de Internet o una Intranet. Pero frecuentemente se encuentra en la literatura que se

describen ambos como si fueran lo mismo, como en [6], [8], [11] y [12]. Además la mayor parte de la literatura revisada se enfoca principalmente a WCMS. Dado esto y por simplicidad, en lo sucesivo se utilizará el concepto de CMS sin importar que se trate de un WCMS.

2.2 Python.

Python [13] es un lenguaje de programación de propósito general aplicado a menudo en roles de *scripting*. Comúnmente se define como un *lenguaje de scripts orientado a objetos*, una definición que combina soporte para POO con una orientación general hacia funciones de *script*.

Las principales características citadas por usuarios de Python son las siguientes:

- ✓ *Calidad del software*. Para muchos, Python se centra en la legibilidad, la coherencia y la calidad del *software* en general se distingue de otras herramientas de scripting en el mundo. El código de Python está diseñado para ser legible, y por lo tanto, reutilizable y mantenible, mucho más que el de lenguajes de script tradicionales.

- ✓ *Productividad del desarrollador*. Python aumenta la productividad muchas veces más allá que lenguajes compilados o estáticamente escritos como C, C++ y Java. El código de Python es habitualmente entre un tercio y un quinto del tamaño de su equivalente en código C++ ó Java. Esto significa que hay menos que escribir, depurar, y menos para mantener después de hecho.

- ✓ *Portabilidad de programas*. La mayoría de los programas de Python se ejecutan sin cambios en las principales plataformas de computación.

- ✓ *Soporte de bibliotecas*. Python viene con una gran colección de funcionalidad pre-compilada y portátil, conocida como la biblioteca estándar. Esta biblioteca es compatible toda una gama de tareas a nivel de programación de aplicación. Además, Python se puede ampliar con ambas opciones, las bibliotecas y una amplia colección de aplicaciones de soporte de terceros.

- ✓ *Integración de componentes*. Los scripts de Python pueden comunicarse fácilmente con otras partes de una aplicación, utilizando distintos mecanismos de integración. Estas integraciones permiten que Python sea usado como un producto de personalización y herramienta de extensión.

De estos factores, los dos primeros (calidad y productividad) son probablemente los mayores beneficios para la mayoría de los usuarios de Python.

En general, Python, goza de una gran base de usuarios, y una muy activa comunidad de desarrolladores. Dado que Python ha existido por más de 15 años y ha sido ampliamente utilizado, también es muy estable y robusto. Además de ser utilizado por usuarios individuales, Python también se aplica a productos reales de ingreso-generación de empresas reales. Por ejemplo:

- ✓ **Google** hace un amplio uso de Python en su sistema de búsqueda web, y emplea al creador de Python.

- ✓ El servicio de video **YouTube**, es en gran medida escrito en Python.
- ✓ El popular *peer-to-peer* **BitTorrent** para compartir archivos es un programa Python.
- ✓ **Intel**, **Cisco**, **Hewlett-Packard**, **Seagate**, **Qualcomm** e **IBM** utilizan Python para las pruebas de hardware.
- ✓ **Industrial Light & Magic**, **Pixar**, y otros utilizan Python en la producción de cine de animación.
- ✓ **JPMorgan Chase**, **UBS**, **Getco**, y **Citadel** aplican Python para los mercados financieros de previsión.
- ✓ La **NASA**, Los **Alamos**, **Fermilab**, **JPL**, y otros utilizan Python para tareas de programación científica.
- ✓ **iRobot** utiliza Python para desarrollar aspiradoras robóticas comerciales.
- ✓ **ESRI** utiliza Python como una herramienta de personalización de usuario final para sus populares productos de cartografía GIS.
- ✓ La **NSA** utiliza Python para criptografía y análisis de inteligencia.
- ✓ El servidor de correo electrónico de **IronPort** utiliza más de 1 millón de líneas de código Python para hacer su trabajo.
- ✓ El proyecto ***One Laptop Per Child*** (OLPC) basa su interfaz de usuario y el modelo de actividad en Python.

Y la lista continua, para más detalles sobre empresas que utilizan hoy Python, véase el sitio web de Python para mayor referencia *http://www.python.org/*.

Greg Stein, ingeniero administrador del grupo *Open Source* de Google, dijo en su presentación en la reciente *SDForum Python Meeting: En Google*, Python es uno de 3 "*lenguajes oficiales*", junto con C++ y Java. Oficial significa que los empleados de Google pueden usar estos lenguajes en proyectos de producción. Internamente, la gente de Google puede usar muchas otras tecnologías, incluyendo PHP, C#, Ruby y Perl.

Python está bien adecuado a los procesos de ingeniería en Google. El típico proyecto en Google tiene un equipo pequeño (de 3 personas) y una corta duración (de 3 meses). Después de que el proyecto se ha terminado, los desarrolladores pueden irse a otros proyectos. Los proyectos más grandes pueden subdividirse en otros más pequeños presentables en 3 meses, y los equipos pueden elegir su propio lenguaje para el proyecto.

2.3 Servidor web Zope.

Un servidor de aplicaciones Web es un ambiente de trabajo (*framework*) que permite crear aplicaciones Web. La mayoría de estos servidores permiten adaptar la presentación de forma dinámica, proveer características de búsqueda, facilidades de integración de bases de datos, manejo del contenido y presentación, construir aplicaciones de comercio electrónico,

proveer mecanismo de control de acceso, integración con diversos sistemas y construir sistemas manejadores de contenido [14].

Zope es un *framework* para la construcción de aplicaciones Web. Una aplicación Web es un programa al que los usuarios pueden acceder a través de Internet por medio de su navegador (*browser*). Se puede pensar en una aplicación Web como un sitio dinámico que proporciona no sólo información estática a los usuarios sino que además les permite utilizar herramientas dinámicas para trabajar con una aplicación.

Las aplicaciones Web se encuentran en todas partes, y los usuarios que navegan por la red trabajan con ellas continuamente. Ejemplos típicos son aquellos lugares que permiten buscar en la web, como yahoo o google, colaborar con proyecto, como *sourceforge*, o comunicarse con otras personas a través del correo electrónico como hotmail. Todas estas aplicaciones pueden ser desarrolladas con Zope.

Zope consiste en varios compontes diferentes que trabajan de manera conjunta para ayudar a construir aplicaciones Web; Zope viene con:

- ✓ **Un Servidor Web.** Zope dispone de un servidor que se encarga de proporcionar los contenidos. Zope puede trabajar también con otros servidores web, como Apache o Microsoft IIS.

- ✓ **Una Interfaz basada en Web.** Cuando se construyen aplicaciones Web con Zope, se puede utilizar el navegador para interactuar con la interfaz de administración de Zope (ZMI). Esta interfaz es un entorno de desarrollo que permite hacer cosas como crear páginas web, añadir imágenes y documentos, interactuar con bases de datos relacionales externas y escribir scripts en diferentes lenguajes. En la figura 2.1 se muestra la pantalla principal del ZMI.

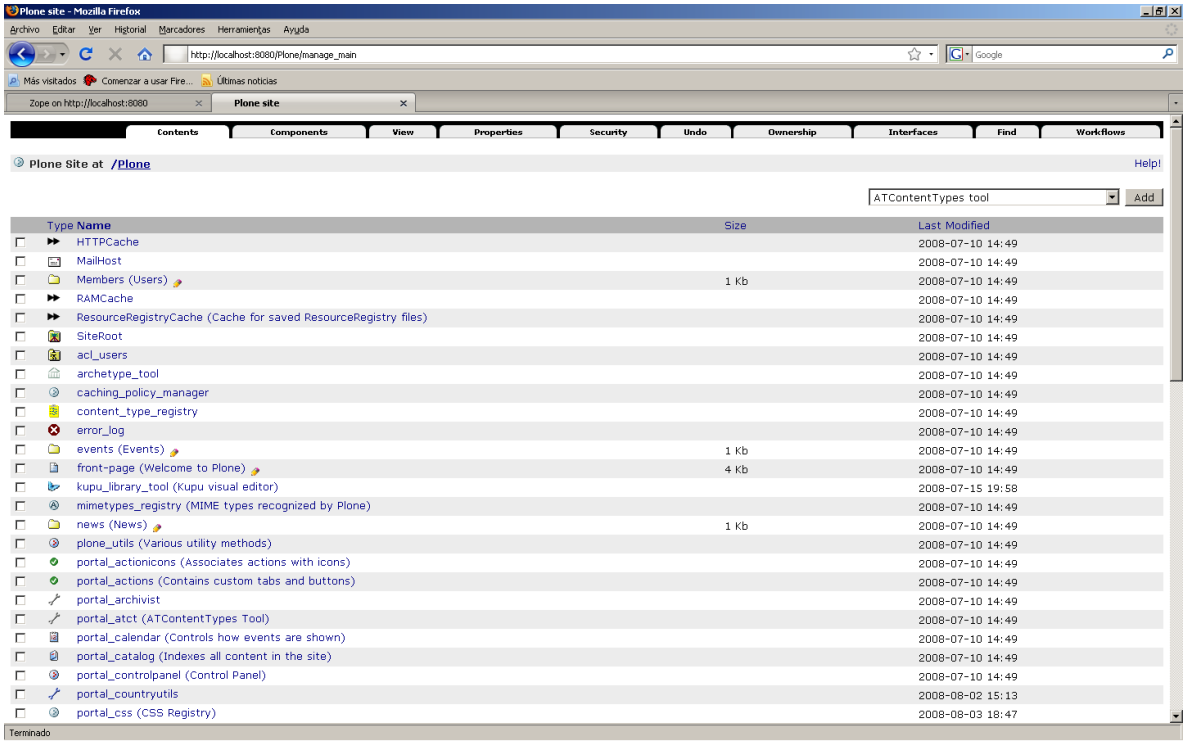

Figura 2.1: Zope Management Interface (ZMI).

- ✓ **Una base de datos de objetos.** Cuando se trabaja con Zope, la mayoría de las veces se trabaja con objetos almacenados en la base de datos de Zope. El interfaz de gestión de Zope proporciona una manera simple y familiar de administrar objetos que se asemeja bastante a la forma de trabajar con manejadores de archivos tradicionales.

- ✓ **Integración Relacional.** No es obligatorio almacenar la información en la base de datos de objetos de Zope, ya que Zope puede trabajar con otras bases de datos relacionales como Oracle, Postgres o Sybase entre otras.

- ✓ **Soporte de lenguajes basados en *scripts*.** Zope permite escribir aplicaciones en varios lenguajes diferentes como Python , Perl o el del propio de Zope, *Document Template Markup Language* (DTML) y *Template Attribute Language* (TAL).

Estas son algunas de las características que han hecho de Zope tan popular para el desarrollo de aplicaciones Web. Quizás la mejor de todas sus características sea su licencia *open source*. Esto significa que la descarga de Zope no es sólo gratuita, sino que también se es libre de usar Zope en los productos y aplicaciones propias sin pagar derechos de autor o derechos de uso. La licencia open source de Zope significa además que está disponible todo su código fuente para verlo, o ampliarlo.

Desde una perspectiva empresarial, existes tres ideas claves para comprender lo que Zope puede hacer: una poderosa colaboración, una gestión de contenidos simple y componentes web.

Diversos servidores de aplicaciones permiten realizar algunas de las tareas siguientes:

- ✓ **Presentar contenidos dinámicos.** Los servidores de aplicaciones permiten crear contenido dinámico. Regularmente un servidor de aplicaciones cuenta con facilidades para la personalización, integración con bases de datos y búsquedas de contenido.

- ✓ **Administrar el sitio web.** Un sitio web pequeño es fácil de administrar, pero un sitio que sirve miles de documentos, imágenes y archivos necesita potentes herramientas de administración. Resulta muy útil contar con herramientas simples pero, a la vez que poderosas para manejar gran cantidad de contenidos web. Se puede gestionar la lógica, presentación o datos, todo desde el navegador.

- ✓ **Construir un CMS.** Un CMS permite a editores sin conocimientos técnicos crear y administrar contenidos para el sitio Web; un servidor de aplicaciones provee las herramientas necesarias para construirlo.

- ✓ **Crear una aplicación de *E-Commerce*.** El servidor de aplicaciones proporciona un *framework* sobre el cual se pueden crear aplicaciones sofisticadas de comercio electrónico.

- ✓ **Hacer que el sitio Web sea seguro.** Cuando se atiende a muchos usuarios, la seguridad se vuelve algo muy importante. Resulta crucial organizar a los usuarios y ser capaz de delegarles tareas de forma segura. Con el servidor de

aplicaciones se puede controlar fácilmente las políticas de seguridad y delegar tranquilamente el control a otros.

- ✓ **Proporcionar servicios de red.** Actualmente la mayoría de los sitios web dan servicio a usuarios, pero muy pronto será necesario que los sitios web proporcionen sus servicios a programas y otros sitios web. El manejo integrado de red de Zope hace que todo sitio sea un servicio de red. La lógica de negocio y los datos puedan ser consultados sobre la web, vía HTTP y XML-RPC.

- ✓ **Integrar contenido diverso.** El contenido actual puede estar almacenado en muchos lugares, bases de datos relacionales, archivos, sitios web separados, etc. Los servidores de aplicaciones típicamente permiten presentar una vista unificada del contenido existente.

- ✓ **Proporcionar escalabilidad.** Los servidores de aplicaciones permiten que las aplicaciones web sean tan escalables como sea necesario para manejar las demandas de los sitios web.

El servidor de aplicaciones de Zope permite realizar todas las tareas arriba mencionadas.

2.4 Base de datos de Zope (ZOBD).

La ZODB es una base de datos para Python, orientada a objetos, que permite un alto grado de transparencia. Las aplicaciones en Python se ven beneficiadas de esta base de datos al no tener que hacer cambios o siendo éstos mínimos en la lógica de la aplicación.

La ZODB es utilizada para almacenar todas las páginas, archivos y objetos creados. Casi no requiere modificar la configuración o darle mantenimiento. Almacena los objetos sobre múltiples registros, donde cada objeto almacenado tiene su propio registro, y cuando éste es modificado, sólo su registro es afectado. Todos los objetos tienen un identificador que los distingue de forma única dentro de la base de datos [15].

Cada petición realizada a la ZODB por medio de la Web es manejada como una transacción separada y sólo es confirmada si no ocurre ningún error. Pero además la ZODB provee un mecanismo para deshacer múltiples transacciones, aunque ya hayan sido confirmadas anteriormente. Esto puede provocar que la base de datos crezca mucho al guardar diferentes copias de un mismo objeto, pero se puede indicar que las copias anteriores a ciertos días se borren. Cabe mencionar que esto no implica ninguna carga al desarrollador de aplicaciones, pues es transparente.

La base de datos tiene un objeto raíz por diseño. Una aplicación típicamente tiene un objeto raíz y todos los demás objetos son accedidos a través de éste por medio de sus atributos o métodos, es decir, regularmente una aplicación forma una estructura de árbol y para alcanzar cualquier hoja hay que iniciar el recorrido a partir de la raíz.

La ZODB no impone ninguna restricción a la aplicación sobre la organización de los objetos, tampoco es necesaria ninguna noción sobre bases de datos relacionales, sino que las

aplicaciones son libres de imponer la organización dentro de la base de datos, hasta se puede implementar una base de datos relacional sobre ZODB.

La principal actividad que debe realizar el administrador con respecto a la ZODB es la de purgarla, ya que la base de datos guarda varias copias de un mismo objeto cada que éste es modificado. Lo cual permite que uno pueda regresar a una copia anterior de un objeto, pero provoca que la base de datos crezca bastante. La tarea del administrador es de indicarle a Plone que borre copias de un objeto anteriores a cierta fecha; por ejemplo, para guardar sólo modificaciones realizadas en la última semana o último mes.

2.5 CMS Plone.

Plone es un CMS de código abierto, gratuito y distribuido bajo la licencia *GNU GPL*. Permite construir sitios ricos en contenido rápidamente. Por sus muchas características se puede decir que es comparable o mejor, a sistemas manejadores de contenido que cuestan cientos de miles de dólares [12]. Además, es considerado como uno de los mejores CMS disponibles hoy en día.

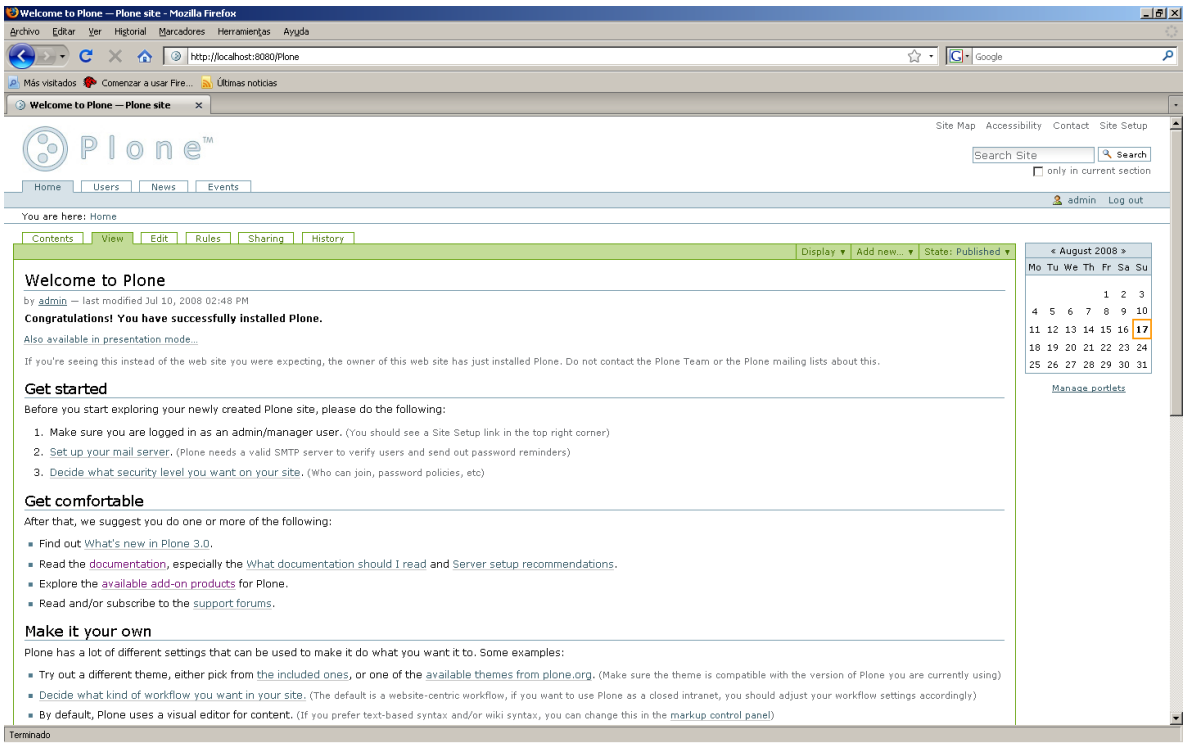

Figura 2.2: Página de Bienvenida de Plone.

La filosofía de que sea código abierto tiene la finalidad de que muchos usuarios, desarrolladores, expertos en usabilidad, escritores técnicos, traductores y diseñadores gráficos trabajen en el desarrollo de Plone. Actualmente se tiene una comunidad de cientos de desarrolladores y compañías que le dan soporte a Plone alrededor del mundo.

Plone está construido sobre Zope, por lo que la mayoría de las características antes mencionadas también aplican para Plone.

A diferencia de Zope, Plone está listo para usarse, basta con configurar algunas opciones, como la forma en que se verá el sitio, usuarios y permisos, para poder empezar a utilizarlo.

Una de las herramientas que se le pueden agregar a Zope es el CMF *(Content Management Framework)*, que provee herramientas que permiten crear CMS. Plone es una capa sobre CMF que se encuentra corriendo sobre Zope, por lo que la mayor parte de tareas de administración del CMF requieren utilizar la interfaz de administración de Zope, llamada (*ZMI*).

Algunas características de Plone son [10]:

- ✓ Tiene instaladores para Windows, Linux y Mac. En la instalación se incluyen productos y *add-ons* de terceros.
- ✓ Su interfaz está traducida a más de 35 idiomas y es fácil agregar traducciones para los productos propios utilizando el estándar i18n.
- ✓ Ofrece altos niveles de usabilidad y accesibilidad ya que es compatible con el estándar gubernamental WAI-AAA y la sección 508 de los Estados Unidos. Además, esto permite obtener mejores búsquedas, al estilo Google.
- ✓ Separa el contenido de las plantillas utilizadas para presentarlo (*skin*). Estas plantillas son escritas principalmente en HTML, utilizando ZPT (*Zope Page Templates* - Plantillas de Páginas Zope) y hojas de estilo (CSS).
- ✓ Tiene un completo sistema de registro de usuarios, cada usuario se registra con su propio nombre de usuario (*login*) y contraseña (*password*), y cualquier otra información que se requiera, con la cual se puede personalizar la interfaz de cada usuario.
- ✓ Utiliza *workflows* para controlar la lógica para procesar los contenidos dentro del sitio. Está lógica puede ser configurada con herramientas gráficas por medio de Web. Los administradores pueden hacer los sitios tan complejos o tan sencillos como lo deseen; por ejemplo, se pueden enviar notificaciones enviándolas por *e-mail*.
- ✓ Para cada pieza de contenido dentro del sitio, se puede configurar el control de acceso e interacción: quien tiene permiso de editarlo, verlo o comentarlo. Esto también puede configurarse vía Web.
- ✓ Dado que es *open source*, puede ser modificado. Se puede cambiar y configurar casi cualquier aspecto de Plone para adaptarse a las necesidades. Innumerables paquetes y herramientas proveen una amplia gama de opciones para sitios pequeños y para empresas muy grandes.
- ✓ Utiliza por defecto una base de datos orientada a objetos, propia de Zope, pero brinda la posibilidad de utilizar otras, incluso relacionales.
- ✓ El proyecto Plone mantiene documentación actualizada, incluyendo libros y tutoriales (*how-tos*) con diferentes niveles, desde creadores o editores de

contenido, hasta desarrolladores. Esta documentación puede revisarse en *http://plone.org/documentation*.

- ✓ Existen muchos sitios que utilizan plone, tanto gubernamentales como privados, el listado se puede consultar en *http://plone.net/sites*.

A pesar de sus beneficios, Plone tiene algunas limitantes [9]:

- ✓ Se requiere mucho poder de CPU para ensamblar las páginas y tratar con la memoria conforme los objetos son cargados desde la base de datos. Por lo que Plone nunca será tan rápido en generar páginas Web como lo puede hacer un servidor Web puro, que es algo que los usuarios finales pueden notar. El tiempo que tarda en desplegar las páginas se debe a que realiza varias tareas importantes como comprobación de permisos para controlar el acceso al contenido, cargar los objetos que se van a utilizar y mostrarlos de acuerdo al perfil del usuario.

- ✓ Aunque se puede utilizar un servidor Apache frente a Zope para dar mayor flexibilidad al servir a otras aplicaciones o páginas estáticas, y proveer seguridad, esto implica que se disminuya un poco el tiempo de respuesta.

- ✓ Depende de varios componentes principales actualizados constantemente por separado, como lo son: Python, Zope, Apache y el mismo Plone. Esto puede causar problemas de compatibilidad entre los componentes agregados y los productos previamente instalados, por lo cual se debe evitar realizar actualizaciones lo más posible para prevenir inestabilidad y fallas en el sistema.

En los últimos años, Plone ha pasado de ser "Solamente otro CMS de código abierto", a una plataforma. Grandes y pequeñas empresas, organizaciones no gubernamentales, gobiernos y particulares, construyen sus sitios web, intranets, y aplicaciones especializadas sobre Plone con gran éxito.

Alentados por los avances en Python y Zope, junto con el aumento de la credibilidad, Plone ha mejorado constantemente durante los últimos años. Plone 2.1 (liberado en 2005) y Plone 2.5 (liberado en 2006) se centraron sobre todo en mejoras incrementales y en sentar las bases para futuros saltos en la funcionalidad. Ahora, con la versión 3.x, Plone una vez más entrega innovadoras y emocionantes nuevas características. [16]

Actualmente InfoMatem se encuentra funcionando sobre Plone 2.1 y el equipo de desarrollo está migrándolo a 3.x, para aprovechar todas las nuevas funcionalidades que ofrece esta nueva versión.

2.6 Productos para Plone.

Los productos son una de las claves principales del CMF y Plone que permiten extender la funcionalidad de Zope. Con los productos se pueden definir nuevos tipos de contenido, herramientas y funcionalidad al sitio. Existen aproximadamente 800 productos disponibles al público en general dentro de la página de Plone [17] con una gran variedad de funcionalidades.

Una vez que un producto ha sido instalado, éste puede hacer uso de toda la funcionalidad del sitio, aún de la provista por otros productos. De esta forma es posible que algunos productos sean dependientes de otros, es decir, un producto utiliza la funcionalidad de otro y sin ese no puede ser instalado.

Si no se encuentra disponible algún producto que cumpla con las necesidades requeridas se puede crear uno, y para esto se han creado diferentes herramientas que facilitan esta tarea [4].

Un producto es un componente de *software* el cual encapsula alguna funcionalidad expuesta a través de interfaces. Dado que en Plone a los componentes también se les llama productos, en lo sucesivo se hará referencia indistintamente a componente o producto.

El apéndice A describe la estrategia general para desarrollar productos de contenido para Plone.

3. Estado del Arte de *Workflows*.

El presente capítulo basa buena parte de su contenido en [18].

3.1 Introducción.

En los últimos años ha habido muchos cambios en la forma en que las empresas hacen negocios. Las estructuras jerárquicas tradicionales están desapareciendo y siendo sustituidas por modernas organizaciones-proceso u organizaciones-equipo. A finales de la década de 1990 la mayoría de las empresas consideraba a la información y los sistemas de información como sus más valiosos activos. Debido a la enorme cantidad de información accesible hoy esto ha cambiado. La gente está abrumada por la información de que dispone y, a menudo, no pueden encontrar exactamente lo que es relevante. Hoy en día, en lugar de centrarse en la información, el principal campo de interés es el conocimiento (saber qué es lo que la organización sabe realmente). En una época de fusiones y empresas conjuntas, la combinación de los procesos de negocio a un proceso de organización transversal es de máxima importancia.

Los conceptos tradicionales de *software* no pueden cumplir estos requisitos; las arquitecturas modernas deben ser capaces de comunicarse a través de las fronteras de la organización, reunir información de diversas fuentes, soportar la colaboración entre personas o *software* tan automáticamente como sea posible (no hay tiempo para supervisar cada paso).

Durante años los investigadores han estado trabajando en conceptos de *software* que se asemejan a las capacidades humanas. El resultado es conocido como teoría de agentes. Lo que comenzó como moda, lentamente se convierte en objeto de consideración racional, dónde y cómo modelar agentes-sistema. No todo el *software* debe construirse sobre agentes, pero existen diversas aplicaciones para las que presenta las capacidades adecuadas.

Con los conceptos de realización casi resueltos, la pregunta sigue siendo ¿Cómo modelar *workflows* (flujos de trabajo, WFs) y cómo describir procesos que no fueron hechos para ser descritos? Extrañamente, la vieja (en términos de TI) técnica llamada redes de Petri [19] ha mostrado ser apropiada para modelar WFs, mientras que nuevos conceptos como semánticas basadas en UML o XML son desarrolladas para trabajar automáticamente las definiciones de procesos.

Con nuevos proyectos emergiendo constantemente existe una clara necesidad de presentar un panorama general de las técnicas disponibles, los problemas aún no resueltos y otros aspectos importantes a considerar antes del desarrollo de nuevos sistemas de *software*. Este capítulo trata de presentar el estado actual de la tecnología de agente, las técnicas de elaboración de modelos de WFs y las arquitecturas alternativas para sistemas de WFs basados en agentes.

3.2 Modelado de workflows.

Los WFs son mapas de procesos de negocio. Se componen de tareas y sus relaciones, criterios para indicar inicio y término de los procesos, y la información sobre la tarea individual. No todo proceso de negocio es un proceso de WF. Un proceso de WF se caracteriza por tres atributos:

- ✓ El proceso es dirigido por caso.
- ✓ El proceso en sí es fundamental.
- ✓ El proceso puede ser definido explícitamente.

Sin embargo un flujo de trabajo (WF) es un conjunto de tareas definidas con dependencias entre ellas. La *Workflow Management Coalition* (WfMC) utiliza actividad como sinónimo de tarea y la define de la siguiente manera: "La descripción de una pieza de trabajo que forma un paso lógico dentro de un proceso. Una actividad puede ser manual, que no admite automatización computacional, o de WF (automatizada). Una actividad de WF requiere recursos humanos y/o de máquinas para apoyar la ejecución del proceso, donde se requieren recursos humanos la actividad se asigna a un participante del WF". En este trabajo la definición difiere de la WfMC en que una actividad, además, puede ser una agregación de actividades atómicas. De ahora en adelante una actividad se denominará tarea para hacer hincapié en esto. Una agregación de tareas está sometida a varias reglas, las tres más importantes son:

- ✓ **Membresía** (una tarea agregada puede verse como un conjunto de actividades de un proceso base).
- ✓ **Atomicidad** (una tarea agregada es una unidad atómica de procesamiento).
- ✓ **Preservación de orden** (una tarea agregada preserva el orden original de las relaciones del proceso base).

Las diferentes tareas son realizadas por uno o más actores. Cada actor (también llamado "agente") desempeña un papel que se define como una descripción generalizada de la responsabilidad del agente en una tarea. El modelo de referencia general para workflows con sus componentes e interfaces principales se puede encontrar en las especificaciones oficiales publicadas por el WfMC [20].

Comercialmente se ofrecen tres tipos de WFs:

- ✓ Un WF ***ad-hoc*** típicamente muestra procesos que involucran coordinación, colaboración, o codecisión humana. Esto lleva a que el problema de orden y coordinación no está automatizado y tiene que ser gestionado por personas. Las tareas de ordenamiento y coordinación se logran mientras se lleva a cabo el WF.
- ✓ El segundo tipo, el WF **administrativo**, puede caracterizarse como una colección de procesos repetitivos, previsibles con reglas de coordinación sencillas. La ventaja de este tipo de WF es que el ordenamiento y la coordinación pueden ser automatizados.
- ✓ Los WFs de **producción** consisten, al igual que el administrativo, de procesos repetitivos y previsibles. A diferencia de los administrativos, los WFs de producción incluyen un complejo proceso de información incluyendo acceso a múltiples sistemas de información. De forma similar a los administrativos, los WFs de producción pueden ser automatizados.

En este capítulo se utiliza el término *workflow* de dos maneras, como se dijo anteriormente, un WF es una agregación de tareas; por otro lado un WF puede ser también

una agregación de WFs, el segundo significado se explicará posteriormente, en el concepto jerárquico de los WFs insertados en un *Metaworkflow*.

Un WF necesita diversas directivas con el fin de cumplir con su trabajo. Esto significa que las reglas de negocio necesarias son mapeadas a reglas de WF. Una regla de negocio puede describirse mejor con los siguientes tres suposiciones: Una regla de negocio es:

- ✓ **Información** sobre la manera en que el negocio se hace,
- ✓ Una **ley o costumbre** que guía el comportamiento de un actor o acciones vinculadas a la organización,
- ✓ Una **declaración de política** o condición que deba cumplirse.

Un WF debe entregar los datos correctos a la persona con la herramienta adecuada en el momento adecuado, la gestión apoya esto. Su propósito general consiste en asegurarse de que las actividades apropiadas son ejecutadas por la persona adecuada en el momento adecuado. Estos requisitos conducen a la elaboración de un sistema de administración de WFs que define, crea y gestiona su ejecución mediante el uso de *software*, ejecutándose en uno o más motores de WF, que son capaces de interpretar el proceso de definición, interactuar con los participantes y, en su caso, invocar el uso de herramientas y aplicaciones de TI externas. Independientemente del número de motores, es indispensable que el sistema de administración de WFs tenga la capacidad de admitir múltiples instancias de WFs simultáneamente. Estas instancias pueden verse coexistiendo con los procesos de negocio reales [21].

3.3 Conceptos de workflows.

Antes de describir los conceptos de flujo de trabajo, es necesario listar en detalle los requisitos generales para los WFs. Con el fin de comprender los problemas y las condiciones para la elaboración de modelos de WFs, la complejidad de las tareas debe ser entendida. Como consecuencia de ello, aparecen varios requisitos para el manejo de tareas. Su realización requiere de la creatividad, la iniciativa y la cooperación entre los participantes. Aunque no es un problema para los agentes humanos atender estos criterios, un sistema de *software* tendría dificultades para hacer esto sin una descripción detallada del contexto. Hasta ahora no existe un método satisfactorio para describir el contexto o incluso la tarea en sí, lo suficientemente bueno para que el *software* pueda entenderlo. Debido a esto lo único que puede hacerse es construir sistemas que ayuden al usuario a definir sus objetivos y sus estrategias.

Los sistemas de administración de WFs rara vez proporcionan la flexibilidad necesaria. Los principales requisitos pueden clasificarse de la siguiente manera:

- ✓ **Apoyo a la cooperación**: La cooperación dinámica entre los participantes debe ser apoyada.
- ✓ **Administración del flujo de la tarea**: A menudo, los patrones de tarea no pueden determinarse por adelantado y requieren adaptaciones en tiempo de ejecución.
- ✓ **Organización temporal**: Las tareas tienen alta duración y necesitan organización y planeación temporal.

Basados en estos requerimientos, este tipo de tareas se pueden combinar en el grupo de WFs *ad-hoc*. En la literatura se han desarrollado varios conceptos para que los sistemas de administración puedan manejar WFs *ad-hoc*.

El **concepto de caja negra** es una de las posibilidades para resolver el problema dinámico. Partes complejas del proceso se asignan a secciones de caja negra, que puede definirse durante su activación y que son desconocidos para el WF global. El *metaworkflow* tiene que "confiar" que esas cajas negras hagan su trabajo, cada vez que se inicializan. Obviamente una estructura jerárquica de al menos dos niveles está integrada a este concepto.

Un enfoque más específico es el **concepto de *Activity Template*** (plantilla de actividad). Se basa en la idea de que varios grupos de plantillas de actividad fueron creados para organizar diferentes actividades en una instancia de WF. Cada plantilla consta de tres atributos: los **atributos de entrada**, que listan el material necesario, la **asignación de atributos**, que especifican las responsabilidades humanas y los **atributos de tiempo**, que definen la duración mínima y máxima, la primera y última hora de comienzo y el tiempo de preparación de la tarea real. Esto último desempeña un papel especial en situaciones críticas de tiempo. Un actor no debe ser sorprendido por la siguiente actividad, por el contrario, debería saber sobre la próxima actividad en la lista de trabajo dándole tiempo para prepararse a sí mismo y a sus instrumentos para el próximo trabajo (tiempo de preparación). Una completa planificación de una situación crítica no es viable. Por esta razón, es de suma importancia permitir el establecimiento y la actualización dinámica de los atributos de las actividades. Además, la inserción y supresión dinámica de una actividad, así como la gestión dinámica de listas de trabajo es de vital importancia.

Un tercer concepto se refiere a un aspecto más técnico. El **concepto de *Triggering*** (disparo) alude al factor más débil del WF: el actor humano. El sistema de administración de WFs no puede obligar a un actor a ejecutar una tarea o responder una vez que ha terminado. Este concepto intenta contrarrestar este problema, evidentemente existe una diferencia entre el sistema que permite una tarea y la solicitud a un usuario para ejecutarlo. Después de habilitar una tarea, el actor puede ejecutarla, pero también puede verse impedido. En un intento para que el sistema de WF pueda controlar el proceso se introduce el concepto de *triggering*. Un *trigger* es una condición externa que lleva a la ejecución de una tarea habilitada. En los sistemas de WF controlados por *triggers* cada usuario tiene una "bandeja de entrada" en la que tiene todas las tareas que un usuario puede estar trabajando en este momento. Cuando el usuario selecciona una de estas tareas el correspondiente *trigger* es disparado, avisando al sistema que se ha (re)-iniciado el trabajo. Cabe señalar, que no todo es tarea de un actor (usuario). Hay otros tres tipos de tareas/*triggers*:

- ✓ **Automática**: En el momento en que una tarea se ha habilitado será activada.
- ✓ **Mensaje**: Una tarea será activada por un evento externo
- ✓ **Tiempo**: Una tarea será ejecutada por un reloj.

3.3.1 *Workflows* de conocimiento intensivo, débilmente estructurados.

En la introducción del capítulo, se señalaron diferencias entre varios tipos de WF. En esta sección se examinará la agilidad de los WFs *ad-hoc* con más detalle. Su principal característica es que no existe un modelo de flujo de trabajo predefinido de los procesos de

negocio, por tanto, se requiere mayor flexibilidad, una mejor perspectiva con cambios dinámicos en las definiciones de reglas de negocio, aplicaciones, servicios y lugares. Esto significa que los WFs *ad-hoc* no se crean "*a priori*", sino cuando se encuentran instanciados y en ejecución.

Para desarrollar conceptos eficientes de WFs *ad-hoc*, es indispensable comprender las razones de este comportamiento ad-hoc. Implícito en el título de esta sección, las dos características principales son conocimiento intensivo y débilmente estructurado. Se refieren a los atributos de los procesos de negocio que mapea este tipo de WF. El conocimiento intensivo describe un proceso complejo con muchas actividades de documentos centralizados y pocas decisiones de actores humanos. Estas decisiones requieren amplio conocimiento sobre el problema real, casos similares pasados, etc. Los actores necesitan conocimientos específicos sobre la actividad y el contexto para lograr el fin. Por otro lado, débilmente estructurado significa que en este tipo de proceso muchos actores de diferentes departamentos con funciones diferentes en distintas áreas interactúan unos con otros. Es imposible describir el WF completo con tantas variables desconocidas. Por ello, este WF es una construcción inestable, que cambia frecuentemente. Los procesos de negocio complejos e informales pueden asignarse a varias cajas negras, y/o el proceso puede cambiar durante su activación.

3.3.2 *Workflows* jerárquicos inter-organizacionales.

¿Dónde está la coherencia de los WFs jerárquicos inter-organizacionales? Inter-organizacional se refiere principalmente a la interoperabilidad entre dos o más organizaciones/actores autónomos con instancias privadas y su cooperación en un WF público. Jerárquico, por otro lado, describe una subordinación de instancias en un *metaworkflow*. Comparando el *metaworkflow* con el componente público y el *sub-workflow* con la parte privada de los WFs inter-organizacionales, se puede encontrar una concordancia. Pero es superficial pensar que no hay diferencias. La instancia pública no es necesariamente una *metaworkflow* en un sentido jerárquico. También es posible que un WF particular más o menos dicte las reglas del escenario. Para los WFs jerárquicos el *metaworkflow* determina el procedimiento general. Además, el concepto jerárquico no es compatible con aspectos privados mientras que los inter-organizacionales sí, por lo tanto, un WF jerárquico inter-organizacional es una combinación de ambos. Contiene la jerarquía con un *metaworkflow* definiendo la ruta principal, pero que consta de muchos *sub-workflows* que hacen su trabajo independiente e interactuando con el *metaworkflow* de forma inter-organizacional.

En la literatura se conocen varias formas de interoperabilidad:

- ✓ **Compartición de la capacidad**: Cumplir con la tarea bajo control de un WF administrador. (Esto corresponde al concepto jerárquico, donde el *metaworkflow* sería ejecutado por un motor central).
- ✓ **Ejecución encadenada**: Dividir la instancia en un número de *sub-workflows* disjuntos que son ejecutados en orden secuencial por diferentes socios (*bussiness partners*). (Este enfoque no pertenece a un concepto jerárquico, debido a que el control del WF es compartido).

- ✓ **Subcontratación**: En esta forma de inter-operar un actor tiene el control del WF. Él es el actor principal y distribuye los *sub-workflows* a otros actores subordinados. Así se genera un árbol jerárquico.

- ✓ **Transferencia de casos**: Cada actor tiene una copia de la descripción del proceso de WF. Los casos pueden ser transferidos para equilibrar la carga de trabajo, ya que las tareas no se pueden implementar en todos los lugares. Es importante considerar que en cualquier momento, cada caso existe exactamente en un lugar. Si en esta forma se utiliza un gestor central de equilibrio, se crea una jerarquía.

- ✓ Casi el mismo criterio es el **caso extendido**. Se diferencia de transferencia de casos porque aquí es posible diversificar las definiciones del proceso con tareas adicionales en un lugar específico.

- ✓ La última forma que se llama **débilmente acoplado**. Este enfoque es casi equivalente al enfoque de caja negra. Un proceso se divide en piezas disjuntas que pueden activarse en paralelo. Todos los sub procesos son locales y conocidos sólo por el actor que los ejecuta. El único componente público es el protocolo que se utiliza para comunicarse. Este enfoque no es propiamente un concepto jerárquico, porque no hay una instancia administradora central.

Independientemente de la forma de inter-operación, se han especificado algunas políticas de transporte para llevar los casos de un actor a otro. Se basan en las siguientes restricciones:

- ✓ **Reducir al mínimo el número de transferencias**, ya que esto lleva tiempo y ocupa la red.

- ✓ **Equilibrar la carga de los actores**. Especialmente en situaciones críticas, es importante no sobrecargar de solicitudes a un actor.

- ✓ **Minimizar el tiempo de salida**. El tiempo total de flujo de los casos debe ser lo más corto posible, ya que más tráfico causa más carga en la red.

- ✓ **Minimizar los costos**, significa que las tareas deben ser ejecutadas en el lugar donde los costos de procesamiento sean mínimos. En este contexto "costo" tiene un doble significado. Por una parte "costo" en el **sentido financiero**, por otra parte, "costo" en **sentido de seguridad** (riesgo mínimo para los actores).

Antes de presentar las *políticas de transferencia*, es necesario explicar la clasificación de los *estados* posibles. Los cuatro estados en orden jerárquico son ***activo*** (una tarea que está en ejecución), ***listo*** (por lo menos una tarea está activada y no espera de un *trigger* externo), ***espera*** (por lo menos una tarea está activada, pero todas las tareas están esperando un *trigger* externo) y ***bloqueado*** (no hay tareas activadas).

Las políticas de transferencia mencionadas arriba se clasifican en las siguientes seis formas:

- ✓ La **política de transferencia persistente** establece que un caso se transferirá, si y sólo si se encuentra en un estado de espera ó bloqueado y una transferencia conduce a un estado listo en otro lugar. Esta política escala la jerarquía.

- ✓ En la **política de transferencia fuertemente persistente**, el caso se transferirá, si y sólo si, está bloqueado y una transferencia conduce a un estado de espera ó listo o en otro lugar. Esta política también escala la jerarquía.

- ✓ En una **política de transferencia de tiempo fuera**, un caso se transferirá, si y sólo si, no ha estado activo durante un tiempo determinado y una transferencia conduce a un estado de listo ó espera en otro lugar. Si el caso fue bloqueado, esta política escala la jerarquía de lo contrario se mueve a lo largo de la jerarquía (o incluso desciende).

- ✓ La **política de transferencia de balanceo de carga**, establece que los casos que están listos, esperando, o bloqueados son trasladados a otro lugar si no son bloqueados en la nueva ubicación y la nueva carga de trabajo es menor. Esta política no pertenece a una transferencia jerárquica. El desempeño se vuelve más importante.

- ✓ En una **política de transferencia dirigida por el costo**, los casos se transfieren a la ubicación en donde los costos de procesamiento para la siguiente tarea son mínimos. Esta política no pertenece transferencia jerárquica. Aquí el costo es el punto importante.

- ✓ Una **política de transferencia inteligente** utiliza una mezcla de las estrategias anteriores y también anticipa desarrollos futuros en sus cálculos (por ejemplo, la carga de trabajo en el próximo período).

Los WFs inter-organizacionales en colaboración con procesos de conocimiento intensivo, débilmente estructurados, causan varios problemas que pueden ser resueltos mediante jerarquías. Los cuatro principales desafíos son:

- ✓ El **problema de cambio dinámico** ocurre si los cambios se realizaron mientras el WF está "corriendo". Los cambios individuales sólo afectan a un solo caso o un pequeño número de casos, mientras que los cambios estructurales influyen en todos los casos nuevos. Existen tres formas diferentes para manejar este problema. La primera solución es reiniciar. Esto afecta a todos los casos en ejecución. Tienen que deshacerse y reiniciarse al principio de un nuevo proceso. El segundo método consiste en transferir el caso a un nuevo proyecto. El último enfoque es de proceder. Los casos en ejecución no se ven afectados por el cambio, múltiples versiones del proceso se producen por un (corto) período.

- ✓ El **problema de gestión de la información** pertenece al último enfoque del problema de cambio dinámico. Múltiples variantes de un proceso deben ser manejadas. La solución es una especie de jerarquía, llamada *herencia*. Este concepto exige WFs mínimos de modo que cada variante utilizada puede ser mapeada como una subclase o superclase de este WF.

- ✓ El **problema de coordinación** se refiere a un arreglo de problemas de dos (o más) actores que están involucrados en un proceso. Cada uno de estos actores es responsable de una parte del WF. De esta forma sólo tareas con relevancia local puede añadirse sin informar a los otros actores. Esto puede causar *deadlocks*, *livelocks* y otras anomalías. La solución es la estrategia de WF privado/público. Son necesarios cuatro pasos para crear un concepto funcional: En primer lugar, se debe desarrollar un WF, en segundo lugar éste debe ser

blindado y distribuido a los agentes, tercero, crear una instancia privada para cada uno de los participantes y, por último, la modificación de todos los WFs de cada actor debe ser heredada y preservada.

- ✓ El último problema es cómo **medir la diferencia entre los procesos**. (Automáticamente) Determinar las diferencias entre dos procesos es necesario para minimizar los costos de un proceso, sin embargo esto causa dos problemas. Antes de fusionar ambos procesos es importante notar dónde ambos procesos se cumplen. El Máximo Común Divisor (GCD) es una posibilidad para lograr este objetivo. Sólo las tareas que se encuentran en ambos procesos son absorbidos en el GCD. Cuanto más cerca se está de la definición de GCD, menor es el número de WFs que resultan, y viceversa. Por tanto, la dificultad es la correcta elección para obtener un buen resultado.

La cuestión sigue siendo por qué se debe usar jerarquías. Cabe señalar que la jerarquía se puede ver desde dos perspectivas. En primer lugar se aborda el concepto de herencia como se conoce en el desarrollo de *software* orientado a objetos. Un WF que está por debajo de un segundo en la jerarquía tiene las mismas funciones básicas y algunas partes nuevas o modificadas. La segunda forma de la jerarquía se refiere al concepto organizacional. Un WF consiste en un conjunto de tareas. Una tarea en sí puede a su vez ser un objeto complejo (es decir, un WF en sí mismo). El WF completo puede ser visto como una jerarquía, en contraste con la otra forma mencionada anteriormente no hay herencia y pueden no existir similitudes entre el WF y *sub-workflows*.

Básicamente este concepto tiene dos ventajas. En primer lugar, facilita la elaboración de modelos, la edición y la ejecución de WFs. Partes del WF pueden verse como una caja negra por el diseñador y por el motor, simplemente se pasan a la instancia responsable. Además este concepto de caja negra soporta privacidad que es especialmente útil en los WFs inter-organizacionales. La segunda ventaja es la herencia que permite la reutilización de componentes y edición central.

3.3.3 Auditoría de datos (Historial de eventos).

Auditar los datos es un mecanismo esencial para incrementar el rendimiento del proceso y minimizar los costos al reducir el tiempo de salida y optimizar el WF. Es evidente la necesidad de análisis de los procesos. El procedimiento de análisis del proceso es para identificar los cuellos de botella en el mismo y determinar cómo rediseñar tantas tareas como sea necesario para lograr el objetivo. De este modo tres tipos básicos de análisis se han destacado. El primer análisis es la **validación**. Esta forma de evaluación, por ejemplo revisa si el WF se comporta como se esperaba. Esto se hace frecuentemente por una simulación interactiva, en donde se alimenta al sistema con casos ficticios. El segundo análisis es la **verificación**. Aquí por ejemplo, la corrección del WF es establecida. El análisis de **desempeño** trata de encontrar los cuellos de botella y pasarlos. Los dos últimos análisis requieren técnicas avanzadas para cumplir con sus requisitos. Varias de estas han sido implementadas mediante redes de Petri [22].

Dada la necesidad del análisis de datos, los eventos de los WFs deben ser almacenados. El sistema prototipo EvE implementa ejecuciones de instancias dirigidas por eventos utilizando un historial de eventos de acuerdo al siguiente concepto. EvE registra varios

atributos, como el tipo de evento, el WF donde se produce, la hora en la que el servidor detecta el evento, etc., para el análisis post-mortem.

3.3.4 Recuperación del *workflow*.

Recuperar el WF incluye, al menos, dos características. En primer lugar, deshacer tareas incompletas, o incluso tareas completadas que deben ser canceladas. En segundo lugar, rehacer un WF deshecho. Si un proceso se cancela antes de terminar, hay dos reacciones posibles: Por un lado, la instancia puede completarse, y otro WF puede cancelar el proceso, por otra parte, la instancia puede desertar y las tareas ejecutadas pueden deshacerse. La primera opción es inviable, debido a cuatro razones [23]:

- ✓ Un WF que será deshecho después de su término requiere **tiempo de cómputo adicional**.
- ✓ Los **recursos humanos** se ocupan inmediatamente después de terminar un desarrollo.
- ✓ Un **segundo WF** para destruir al primer proceso malgasta recursos.
- ✓ Los **recursos instalados** no pueden utilizarse en otros WFs.

Evidentemente, dados estos motivos, la segunda opción es la mejor elección. Por lo tanto, los sistemas de administración de WFs deben implementar una función para cancelar las instancias y tareas durante su ejecución para ahorrar recursos y garantizar la consistencia del WF.

3.3.5 Integración de recursos.

Un recurso (también conocido como actor, ejecutante o participante del proceso [24]) es una entidad asignada a una actividad del WF a la que se solicita realizar un trabajo con el fin de completar el objetivo de la actividad.

La integración es considerada como uno de los principales objetivos durante el diseño de sistemas de información. Integrar, significa [7]: hacer que alguien o algo pase a formar parte de un todo, en el contexto de sistemas de información, integrar quiere decir coordinar o fusionar recursos en un todo unificado funcional a fin de terminar con la segregación existente. Dos tipos distintos de integración se pueden distinguir [25]:

- ✓ *Integración mediante conexión* ocurre si un nuevo sistema se produce a través de la creación de vínculos entre entidades o sub-sistemas separadas, pero lógicamente conectadas. Típicamente se trata de una integración de sistemas existentes.
- ✓ *Integración mediante combinación* ocurre si elementos similares del sistema se combinan, lo que lleva a una disminución en el número de elementos y relaciones dentro del sistema (en sentido abstracto). Normalmente esta forma de integración ocurre durante la fase de diseño conceptual de un sistema de información.

Rosemann señala como objetivos principales de los esfuerzos de integración: la reducción de redundancia, el aumento de la consistencia e integridad del sistema, y mejor apoyo a la toma de decisiones [26]. El diseño de un sistema de WF crea necesidades de integración, que pueden ser distinguidos como integración interna y externa. Los **requisitos de integración interna** se refieren a los sistemas a los que un sistema de WF necesita conectar a fin de garantizar la funcionalidad del núcleo del mismo. Los **requisitos de integración externa** existen como interfaz para sistemas que o bien invocan al sistema de WF desde el exterior o sistemas que son invocados por éste.

3.3.5.1 Requisitos de integración interna.

Un sistema de WF coordina actividades, recursos, datos y aplicaciones. En consecuencia, todos estos elementos deben integrarse para garantizar la funcionalidad del sistema.

- ✓ La **integración de recursos** es necesaria por el sistema de WF para llevar un registro de los participantes disponibles para la asignación de actividad. Dado que muchas empresas mantienen la información sobre los recursos en directorios organizacionales o aplicaciones similares, un sistema de WF totalmente integrado debería utilizar esta información en lugar de replicar los datos internamente.

- ✓ La **integración de datos** se requiere para hacer accesibles los datos relevantes para el sistema de WF. Esto puede lograrse mediante la conexión del sistema a las bases de datos utilizadas por sistemas externos. Si el sistema de WF actúa como centro de integración de aplicaciones, la conversión de tipos de datos y valores de campos puede ser necesaria. Desde la perspectiva de gestión de recursos, la integración de datos es necesaria si los valores de campos de objetos de negocio serán utilizados durante la selección de los ejecutantes calificados del WF.

- ✓ La **integración de aplicación** describe la capacidad del sistema de WF para invocar aplicaciones externas durante la ejecución de un proceso. Para procesos organizacionales, las aplicaciones son a menudo llamadas en su totalidad, mientras que los procesos de *software* la granularidad de la invocación de aplicación es a nivel de método o función.

- ✓ Además de estos tres requisitos de integración, la utilización de la infraestructura de seguridad existente es otra característica importante de los sistemas de WF. La **integración de seguridad** se refiere a la utilización de los mecanismos de autenticación y autorización a través del sistema de WF, por ejemplo, control de acceso basado en roles.

3.3.5.2. Requisitos de integración externa.

La integración externa de un sistema de WF se refiere al hecho de que el sistema de WF es una aplicación del sistema en sí mismo. Aplicaciones externas pueden llamar servicios del motor de WF, invocar y consultar el estado de las instancias, o manipular las asignaciones de recursos mediante mecanismos de planificación externa. Además, puede ser necesario que el sistema de WF presente trabajos a terceros, que no son usuarios directos de la aplicación.

- ✓ La **invocación externa** del motor de WF se utiliza durante el proceso de integración B2B (*business to business*, protocolo de comunicación de comercio electrónico), por

ejemplo. La puesta en marcha del servicio de WF puede exponer sus capacidades como servicios a terceros, permitiéndoles invocar un proceso y pasar datos para la instancia. Ejemplos de invocación externa son e-mail (el demonio de correo pone en marcha el WF), la web (un servidor web pone en marcha el WF) u otras aplicaciones, que integran el sistema de WF.

- ✓ La **presentación de información a terceros** es necesaria, si el sistema de WF debe notificar a los participantes externos sobre el estado de "*su*" instancia o si la carga de información del sistema se transmite a un sistema externo de gestión. Además, el uso de información de auditoría por aplicaciones externas entra en esta categoría.

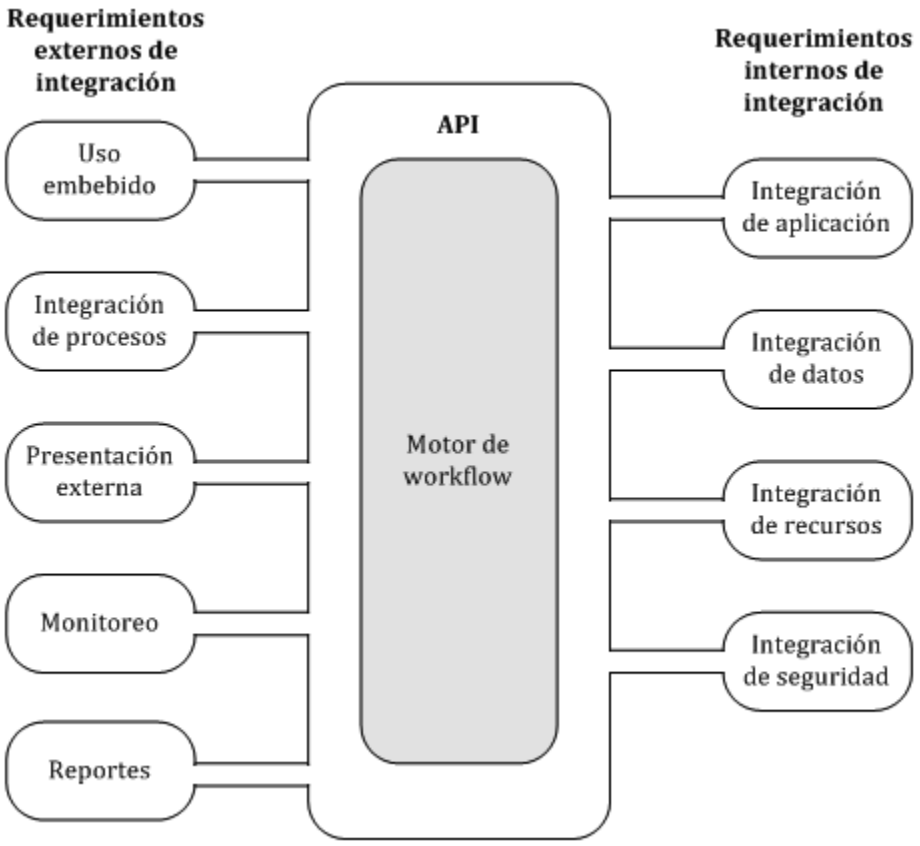

Figura 3.1: Requerimientos de integración [25].

Además de la ejecución normal, el sistema de WF debe manejar notificaciones de los ejecutantes y administradores del proceso, así como la gestión de situaciones inesperadas (manejo de excepciones [27]).

3.3.6 *Workflows* temporales.

Cualquier proceso de negocio requiere de una ejecución coordinada de actividades para alcanzar una meta común. Un WF describe formalmente estas actividades, incluyendo criterios para asignar actividades simples a las unidades de ejecución. El modelo de proceso o esquema (*schema*) define la estructura del WF y la forma en la que las actividades atómicas son coordinadas y realizadas secuencialmente. El diseñador del WF especifica el modelo de proceso, incluyendo los criterios para asignar tareas a las unidades de ejecución (agentes).

Distintas instancias de un WF pueden ser ejecutadas, cada una teniendo sus datos específicos. Los sistemas de administración de WF (WfMS), son sistemas de *software* que soportan la ejecución de instancias de WFs. La mayoría de los WfMSs utilizan un sistema administrador de base de datos (DBMS) basado en el modelo relacional.

Siempre que los aspectos temporales de información necesitan ser administrados, dos escenarios posibles pueden ser identificados: **administración explícita**, es decir, el desarrollador de la aplicación considera el tiempo como una pieza de *datos especiales* junto con otros *datos tradicionales*; y **administración automática**, realizada directamente por el DBMS. En la última situación, para cada contenido el desarrollador de la aplicación puede referirse al *período de validez*, es decir, el tiempo durante el cual el hecho es verdadero en el modelo, que es administrado directamente por el sistema [28].

La administración de algunos aspectos temporales se define en el nivel de modelo conceptual, esto es, al momento de definir el esquema: por ejemplo, los cambios definidos sobre un esquema afectan tanto a casos que están por iniciarse, como casos que ya comenzaron, pero no han sido completados aún. La administración de algunos otros aspectos temporales sólo puede definirse en tiempo de ejecución: por ejemplo el balanceo de carga de trabajo entre agentes puede ser realizado sólo durante su ejecución.

Se pueden obtener ventajas considerables al adoptar una arquitectura temporal para el WfMS. Lamentablemente, debido a la carencia de DBMS temporales de alto rendimiento y confiables, no existe un WfMS que funcione sobre un sistema de este tipo. Además de las ventajas típicas de la administración de datos estándar, existen otras ventajas principales que resultan de la adopción de un DBMS temporal dentro de un WfMS, éstas se relacionan con la ejecución de consultas (*queries*) temporales: la administración de versiones del esquema, el balanceo de carga, elegir al agente ejecutante, manejo de excepciones, y el análisis temporal del depósito de datos; por mencionar algunas. En [28], los autores proponen arquitecturas diferentes para un WfMS temporal, donde son considerados explícitamente aspectos temporales; después se estudian los problemas de implementación relacionadas con una arquitectura factible, que no es sencilla debido a algunas restricciones técnicas inherentes a los componentes adoptados.

3.4 Conceptos de representación.

Los WFs necesitan ser modelados. Aquí se presentan tres técnicas de elaboración de modelos, con sus ventajas y desventajas. En primer lugar la técnica más popular, las redes de Petri, se presentarán varios tipos de redes de Petri (normal, reactiva, de prioridad, de tiempo, coloreada, y jerárquica). Posteriormente un estudio general del lenguaje de modelado de procesos (MPL) y se mostrarán varias técnicas de UML. Finalmente, se discutirán las diferencias de estos tipos de modelos de representación.

3.4.1 Redes de Petri.

Como se mencionó anteriormente, las redes de Petri son capaces de modelar estados, eventos, condiciones, sincronización, paralelismo, selección, iteración y prácticamente los procesos reales independientemente de su complejidad. De esta forma, las redes de Petri son la técnica básica para modelar WFs, por un lado gracias a su profunda base teórica, y otra

parte por la facilidad para visualizar el flujo de trabajo con redes de Petri, dadas cinco ventajas [19]:

- ✓ La "correctez" en la **probabilidad** de una red de Petri
- ✓ Adecuación para representar **modelos dinámicos discretos**
- ✓ La existencia de gran número de **herramientas de modelado y simulación**
- ✓ La implementación de un WF basado en redes de Petri es **conveniente**
- ✓ La **estandarización** de los sistemas de gestión de WFs con redes de Petri

La idea básica de combinar WFs y redes de Petri es asociar tareas con transiciones de redes de Petri, de modo que una tarea de ejecución es equivalente a una transición disparadora. El problema aquí es que los motores de WFs son sistemas reactivos que deben disparar, y el juego semántico de las redes de Petri "pueden" hacerlo. Debido a esto las redes de Petri normales no son tan convenientes como parecen. De ahí que en la literatura se presentan otros tipos de redes de Petri más competentes para este requisito. Algunos de los más importantes se describen a continuación.

Una red de **Petri reactiva** puede ser construida simplemente cambiando la regla "puede disparar" en una regla "debe disparar". De esta manera, el juego semántico se convierte en reactivo. Debido a la proximidad de las redes de Petri este procedimiento no es aconsejable, ya que el ambiente del sistema también es incluido. El problema surge del hecho de que el ambiente es activo y no reactivo. Así, para el ambiente la regla es "puede disparar" y para transiciones internas se utiliza "debe disparar". El comportamiento de una red de Petri reactiva se describe de la siguiente manera. Una red de Petri tiene dos posibles estados: estable e inestable, mientras que un estado estable se alcanza si no hay transiciones internas habilitadas. Si una o más transiciones del ambiente son disparadas, la red se convierte en inestable. En este caso, el sistema debe seguir disparando transiciones internas habilitadas, siempre que no llegue a un estado estable. Esto crea el problema de que si la red contiene un bucle, un estado estable no se alcanzará nunca y el sistema diverge. Por lo tanto, un sistema reactivo se mueve de uno a otro estado estable si está libre de bucles.

Si se asignan a WFs, hay al menos tres características que impiden que las redes de Petri normales sean adecuadas para el motor de WFs:

- ✓ Las transiciones se **disparan** instantáneamente.
- ✓ Las transiciones en un WF generalmente **modelan tareas** a diferencia del motor de WFs que supervisa las tareas, pero no las ejecuta.
- ✓ Los WFs basados en redes de Petri se basan en la **semántica activa**, mientras que un motor de WFs debe ser reactivo.

Esto puede modificarse refinando tareas, distinguiendo entre el motor y el ambiente, y la modificación de la regla de disparo.

Similar a las redes reactivas, se desarrollaron las llamadas **redes de prioridad**. Este tipo de redes de Petri define una prioridad estática para cada transición que las ordene. Estas prioridades obtienen la "viveza" lo que significa que la red no caerá en *deadlock* ni producirá *trheads*/situaciones que no pueden correr.

Basado en el trabajo de varios problemas nuevos como el mapeo de tiempo y datos o modelos grandes se analizaron con las llamadas **redes de Petri de alto nivel**. Tres de las más populares mejoras son la extensión con color para modelar datos, la extensión con tiempo y, por último, la ampliación con jerarquía para mapear modelos grandes.

Las **redes de Petri coloreadas** difieren de una normal en que cada señal (*token*) tiene un color. Esto se refiere a la posibilidad de almacenar valores en los *tokens*. Estos valores pueden alterarse por transacciones.

Las **redes de Petri con tiempo** se utilizan para simular el comportamiento temporal de un sistema. Esto puede lograrse usando *tokens*, lugares, y/o transiciones.

Las **redes de Petri con jerarquía** son creadas por el uso de subredes. Se trata de lugares, transiciones y subsistemas. De esta manera se crea una estructura jerárquica. El nivel superior describe un proceso en la forma más simple posible y cualquier nivel más profundo especifica al anterior. [22]

3.4.2 Lenguaje de Modelado de Procesos (PML).

El PML, tiene la capacidad de describir procesos. Desgraciadamente, no hay un estándar para PMLs, sino más bien numerosas variedades con el mismo nombre. Por tanto, el PML no existe. Debido a este hecho, se proporciona una visión general de PMLs. En primer lugar, un desarrollador de PML tiene que decidir si el lenguaje debe ser formal, semiformal e incluso informal. En segundo se debe tomar una decisión sobre el lenguaje de "distribución":

- ✓ Un pequeño núcleo de PML y varios sub-PMLs para cada sub-dominio de un proceso o un enorme PML, con sub-PMLs como vista.
- ✓ Un PML para todas las fases del meta-proceso PML o un PML para cada meta-fase.

La decisión de un núcleo de PML se recomienda por varias razones:

- ✓ Tecnologías activamente **reutilizables**, de forma que el trabajo conceptual y técnico no tenga que hacerse dos veces.
- ✓ **Estandarización** (reduce los costos de la adaptación).
- ✓ **Interoperabilidad** con otros procesos.
- ✓ Facilidad en la **evolución a nivel de usuario** del modelo de proceso (un lenguaje no causa costo adicional para aprender si otro proceso debe ser descrito) [29].

3.4.3 Lenguaje Unificado de Modelado (UML).

Si bien muchos PMLs se basan en redes de Petri, ahora se presenta un enfoque completamente diferente utilizando UML. Este lenguaje ofrece diversos diagramas de proceso para las diferentes vistas de procesos [30]:

- ✓ **Diagrama de estructura estática** para mapear la estructura del equipo,

- ✓ **Diagrama de secuencia** para mapear instancias de los procesos de negocio,
- ✓ **Diagrama de caso de uso** para mapear relaciones estáticas entre procesos de negocio,
- ✓ **Diagramas de secuencia** para mapear las interacciones entre los procesos de negocio y los actores,
- ✓ **Diagrama de colaboración** para mapear las interacciones y relaciones entre los procesos de negocio y los actores, y
- ✓ **Diagrama de actividad** para mapear órdenes de los procesos de negocio admisibles.

Vale la pena examinar más de cerca esta última, que es una forma especial de diagramas de estado de UML. La primera y más importante diferencia respecto a las redes de Petri normales es que la semántica de los diagramas de actividad es reactiva. Un diagrama de actividad UML describe el comportamiento de un sistema de WFs. Este sistema tiene dos características principales. En primer lugar es evidente que es reactivo y se ejecuta en paralelo con su ambiente; responde o reacciona a los eventos de entrada. En segundo lugar un sistema tiene una función de coordinación que indica que no ejecuta una tarea, sino que coordina la ejecución por el agente [19].

3.4.4 Comparación.

Como se mencionó, las redes de Petri normales son activas no reactivas, no pueden guardar datos ni tiempos y no son adecuadas para procesos grandes. Para este tipo de requerimientos son necesarias diversas extensiones de redes de Petri. En contraste los diagramas de actividad UML incluyen todas estas características. Dado que las redes reactivas no pueden almacenar datos (por lo tanto, se necesitan redes coloreadas) transacciones de decisión conflictivas se activan y la red se comporta no determinista, mientras que los diagramas de actividad son deterministas.

En conclusión tanto las redes de Petri como UML están en posición de manejar todo tipo de requerimientos de proceso. PML puede hacer esto también, pero debido a la falta de estandarización de este lenguaje pueden ocurrir diversos problemas de comunicación en colaboración con otros sistemas. Ignorando este problema, ninguno de los planteamientos presentados es más adecuado que los otros de acuerdo a las potencialidades técnicas y la elección se deja en manos de las preferencias personales de los desarrolladores.

3.5 Agentes, marco de referencia.

Como se mostró anteriormente, los modernos sistemas ágiles de WFs requieren gran flexibilidad, es decir deben ser capaces de reaccionar a los cambios en el WF durante su ejecución. El *software* convencional alcanza aquí su capacidad máxima, requiriendo casi siempre rutas predeterminadas. Si bien este problema podría resolverse utilizando un intérprete para procesar la definición del WF, muchos procesos de negocio se distribuyen en varias estaciones de trabajo, departamentos e incluso organizaciones. En la mayoría de los casos la compatibilidad a través de estos límites no es soportada, lo que hace imposible que un solo sistema de WF cumpla con su tarea. Diversos estudios ven la solución a este problema en

la aplicación de la tecnología de agentes. Los sistemas de agentes son distribuidos por su propia naturaleza, consisten de varias entidades autónomas, cada una de ellas posiblemente desarrollada independientemente, y además gestionan la comunicación a fin de lograr un objetivo común. Las siguientes secciones, presentan los fundamentos de los sistemas de agentes, modelos alternativos de sociedades, lenguajes de comunicación, así como posibles aplicaciones además de sistemas de WFs.

La teoría de agentes ha sido un tema importante de investigación en los últimos años dando como resultado distintos enfoques, definiciones y estándares. En este trabajo se presenta una revisión de la arquitectura general y de las decisiones que deben tomarse en el diseño de un marco de referencia.

Antes de proceder con sociedades es necesario dar una definición de agente simple. Un agente es una unidad autónoma de *software* que actúa de forma independiente.

Si bien en esa definición se afirma expresamente a un agente de *software*, es importante señalar que en determinados ambientes (especialmente sistemas de WFs) un agente a menudo puede referirse tanto a agentes de *software* como agentes humanos.

Dependiendo del uso del término, existen diversos atributos asignados a los agentes. Una clasificación común es agentes fuertes y débiles.

Un *agente débil* se caracteriza por:

- ✓ **Autonomía**: actuar independiente, basado su propio razonamiento.
- ✓ **Habilidad social**: capacidad de interactuar con otros agentes.
- ✓ **Reactividad**: capacidad de percibir el entorno y reaccionar adecuadamente.
- ✓ **Proactividad**: no sólo reaccionar al ambiente sino influir en él activamente.
- ✓ **Orientación a objetivos**: el agente trabaja para una meta establecida por el director (agente ó usuario)

*Agentes fuerte*s tienen atributos adicionales:

- ✓ **Movilidad**: la habilidad de moverse en una red (a través de estaciones de trabajo, redes, etc).
- ✓ **Benevolencia**: el supuesto de que los agentes no tienen metas en conflicto, lo que les permite trabajar sin tener que resolver los conflictos primero.
- ✓ **Racionalidad**: los agentes buscan la mejor manera de lograr sus objetivos.
- ✓ **Adaptabilidad**: capacidad de adaptación a las preferencias de los directores.
- ✓ **Preservación**: la capacidad de preservar el estado interno de desactivación a reactivación.

La lista de atributos puede no ser completa pero contiene los aspectos más importantes para este tipo de clasificación.

Un enfoque diferente es descrito por los agentes BDI (*Beliefs-Desires-Intentions*, creencias-deseos-intenciones), que tratan de especificar con mayor detalle los componentes

sociales y racionales de un agente. Un agente está equipado con un conjunto de creencias que describen su comprensión sobre su entorno (posibles estados y desarrollos). Un subconjunto de todos los posibles estados se considera como deseable (deseos). Pueden ser alcanzados ya sea a través de ciertas acciones tomadas por el agente o por otras circunstancias. A veces el término "objetivo" se utiliza para describir los estados que pueden ser alcanzados, basado en los esfuerzos del propio agente. El término "intenciones" finalmente aborda un conjunto de acciones que el agente intente tomar, con el fin de lograr uno o más de sus objetivos. Una lista secuencial de acciones es a menudo llamada "plan".

3.5.1 Sociedades de agentes.

Un agente simple por sí solo es de poca utilidad para el director. Existe la necesidad de negociar contratos, recuperar información o contactar a otros agentes. Un conjunto de agentes que interactúan entre sí para algún propósito y/o comparten una ubicación pueden ser denominados una sociedad. En años recientes se ha tomado conciencia de las similitudes entre los sociedades humanas y sociedades de agentes. En [31], se afirma que "los agentes (como los humanos) pueden funcionar de manera más eficaz en grupos que se caracterizan mediante cooperación y división del trabajo". Las mismas reglas que se aplican a las sociedades humanas también se aplican a las sociedades de agentes: Son necesarias reglas, servicios y coordinación, dependiendo del tipo de sociedad. Se puede distinguir tres tipos básicos de **arquitecturas de agente**:

- ✓ Arquitectura de **agente simple**: Este escenario consta de un solo agente que sirve a un objetivo predefinido. Aplicaciones comunes son las interfaces de usuario, o agentes de información personal.

- ✓ Sociedad **homogénea**: Este tipo de sociedad se compone de numerosos agentes independientes (posiblemente desarrollados por autores independientes), que se ajustan a una interfaz predefinida. La principal ventaja de esta opción es que a pesar de que cada agente pueda tener objetivos individuales que no sirven a los demás, todos comparten las mismas capacidades. En teoría sería posible que alguno de estos agentes pueda cubrir las funciones que actualmente requiere esta sociedad. La mayoría de las plataformas hacen uso de este tipo de arquitectura al publicar una interfaz y permitir que cualquier agente compatible con esta interfaz pueda unirse a la sociedad. La interfaz incluye manejadores de eventos, mensajes predefinidos, etc. para permitir la comunicación, además, la creación y eliminación de agentes. Debido al desarrollo independiente de los agentes existe la posibilidad de comportamiento no deseado a pesar de que el agente cumpla con los requisitos externos.

- ✓ Sociedad **heterogénea**: Esta forma es la más liberal de la sociedad; los agentes se desarrollan sin restricciones y puede entrar o salir de la sociedad arbitrariamente. Al igual que en las sociedades humanas la única capacidad común es alguna forma de comunicación, actualmente existen varios intentos de estandarizar dicha comunicación. A excepción de esto, los agentes son libres de trabajar para alcanzar sus propias metas, formar equipos para lograr sus objetivos o adoptar cualquier rol para el cual se consideran apropiados. Al igual que con todas las sociedades liberales, este escenario es extremadamente vulnerable al uso indebido de tipos. No hay control sobre los objetivos del

agente, ni su comportamiento social, no hay instancias tomando el control de los miembros de esta sociedad.

Estrictamente hablando, la sociedad heterogénea es la única arquitectura abierta justa. En los últimos años, muchos desarrolladores han diseñado marcos para agentes, encontraron soluciones a insuficiencias conceptuales y para ello han disminuido o incluso destruido toda posibilidad de permitir a nuevos agentes convertirse en una parte de esa sociedad. En otras palabras una sociedad sólo puede estar abierta a nuevos miembros si no hay conocimientos implícitos necesarios para la interacción.

En [32], se describen cuatro tipos básicos de agentes:

- ✓ **Egoísta**: cuando toma un cierto rol, el agente intenta cumplir con este, pero en caso de conflictos con sus propios objetivos, los prioriza.
- ✓ **Social**: cuando un conflicto entre sus propios objetivos con los de la sociedad y el agente sigue tomando su rol.
- ✓ **Máximamente social**: el agente se dedica por completo a los objetivos sociales, dejando de lado los suyos.
- ✓ **Máximamente egoísta**: el agente se dedica por completo a sus propios objetivos dejando de lado los de la sociedad.

Para cualquier tipo de organización existen dos enfoques contrarios, *bottom-up* y *top-down*. Cuando se crea una sociedad *top-down* el diseñador comienza creando instituciones, departamentos, roles necesarios, etc. y predefine la forma de la sociedad. Una vez que el diseño se completa los agentes pueden entrar en la organización y moverse a los puestos disponibles. El enfoque *bottom-up* empieza con un grupo no estructurado de agentes con objetivos más o menos comunes. Haciendo uso de su lenguaje común pueden negociar la estructura más adecuada para su situación actual y, por lo tanto la estructura de la sociedad.

En la siguiente sección se presentan diferentes **formas de organización/coordinación** adoptadas de la teoría organizacional. Una organización se define como un conjunto de entidades autónomas que trabajan conjuntamente para alcanzar metas comunes, regulada por un conjunto de reglas y normas sociales.

La forma tradicional de coordinación es una **jerarquía**. Los agentes se organizan en grupos jerárquicos, cada grupo tiene un líder que a su vez puede ser también parte de un grupo. El líder es responsable de los servicios que su grupo debe ofrecer, mientras que la mayoría de los modelos ven la forma de trabajo dentro del grupo como irrelevante para los miembros fuera de este grupo. Al igual que en las compañías modernas, el líder del grupo es el responsable de los resultados presentados a los superiores.

Lo contrario es una estructura de **mercado**. Aquí, la organización básicamente no tiene estructura. En lugar de que los agentes se reúnan en un mercado virtual para negociar las condiciones del servicio, precios, etc., una vez que la transacción se completa el contrato se concluye y los agentes deben buscar nuevos socios. Si bien esto permite gran flexibilidad también hace transacciones poco confiables (ninguna de las partes puede confiar en alcanzar su objetivo de compra o venta en un punto específico en el tiempo).

El último concepto de coordinación encontrado en la teoría de agentes es una sociedad **basada en red** o en **equipos**. Ligeramente distinta a una estructura basada en el mercado,

donde los agentes simplemente se encuentran, intercambian información o productos y después buscan otras oportunidades. Una red está formada generalmente por un grupo de agentes que descubren metas que no pueden alcanzar o las alcanzan muy ineficientemente por su propia cuenta. Por tanto, se juntan y forman una sociedad en la que cada agente se mueve a un nuevo rol definido. Las transacciones entre los miembros del grupo principalmente deben servir al beneficio del grupo en lugar de los agentes individuales.

Hasta el momento sólo se ha mostrado que los conceptos tradicionales de coordinación pueden aplicarse a sociedades de agentes, pero aún no se ha descrito la forma en que se ocupan de los problemas de comunicación, la forma en que se gestiona la admisión, se exige el cumplimiento de todos los contratos y se llenan los roles vacantes. Dependiendo de su forma de coordinación, una serie de entidades pueden ayudar a resolver estos problemas. Su objetivo principal es:

- ✓ Especificar la **estructura de coordinación**.
- ✓ Describir los **mecanismos de intercambio** (tipo de contratos, precios, etc.).
- ✓ Determinar las **formas de interacción** y comunicación.
- ✓ Transmitir los **objetivos y normas**.
- ✓ Obligar a los agentes a alinearse a los **objetivos de la sociedad**.

Es ampliamente aceptado que las instituciones mejoran la eficiencia de las transacciones y ayudan a crear confianza, aún cuando sean de menos importancia en los sistemas basados en el mercado. Pueden proporcionar un marco de comercio (estandarización de los contratos, facilidades de pago, etc.) o de un conocimiento común que permite una mejor comunicación. Desafortunadamente, al mismo tiempo, reduce significativamente la flexibilidad, existen procedimientos predefinidos que tienen que ser obedecidos y el mecanismo de mercado no puede desarrollarse plenamente.

Los contratos son un medio importante de regulación. Por un lado los contratos individuales (agente a agente) contienen los términos de las transacciones, por otro lado, se encuentran los contratos sociales (agente a sociedad), que contienen las obligaciones y los beneficios del agente al unirse a la sociedad. Como en todos los contratos de este tipo necesita también algún tipo de vigilancia (monitoreo) y de una autoridad legal responsable de penalizar a los agentes que violen estos contratos (por ejemplo, expulsión o prohibición de nuevos contratos).

Cada arquitectura utiliza diferentes títulos para las instituciones. Como referencia se introduce brevemente la arquitectura propuesta por la FIPA (*Foundation for Intelligent Physical Agents*, Fundación para Agentes Físicos Inteligentes). Es una colección de organizaciones con el objetivo de estandarizar agentes inteligentes incluyendo sus atributos, comunicación y organización. Un aspecto central de este estándar es su apertura a los agentes desarrollados de forma independiente y la posibilidad de interactuar basados en un lenguaje común. La FIPA en su estándar, no favorece explícitamente una sociedad jerárquica o una basada en el mercado, pero en cambio intenta crear un marco adecuado capaz de adaptarse a cualquier forma de organización. Como consecuencia, las instituciones descritas son conceptos abstractos [33]:

- ✓ **Directorio Facilitador** (DF): contiene el servicio de *sección amarilla* (servicios ofrecidos por los agentes); Es interesante notar que la arquitectura incluye explícitamente la posibilidad de tener más de un DF en una plataforma de agentes, lo cual requeriría la comunicación entre DFs, a fin de localizar un determinado servicio no inscritos en el DF consultado inicialmente.

- ✓ **Sistema de Administración de Agentes** (AMS): esta es la instancia de administración general, responsable de la creación y destrucción de agentes dentro de la plataforma. Como tal, es también el más adecuado para hacerse cargo de *páginas blancas* (lista de agentes registrados).

- ✓ **Servicio de Transporte de Mensajes** (MTS): responsable del transporte de mensajes dentro de la plataforma (agente a agente) y a través de diferentes plataformas.

Es interesante notar que esta parte de la especificación no incluye ningún detalle sobre confianza, contratos, etc. Existen varias plataformas de agentes disponibles o en desarrollo y a menudo es difícil entender los roles/instituciones necesarios para que los agentes funcionen y los simplemente necesarios para el marco de la aplicación.

3.5.2 Comunicación de agentes.

La comunicación es esencial para cualquier *software* distribuido. Mientras que los desarrolladores pueden confiar en la capacidad de entornos homogéneos, los entornos heterogéneos son completamente desconocidos para él. Esto plantea dos problemas principales a cualquiera que esté dispuesto a desarrollar un agente compatible con esta arquitectura (asumiendo que el transporte de mensajes está disponible):

- ✓ ¿Qué mensajes deben entender los demás agentes y cuáles son sus respuestas?

- ✓ Si ambos agentes saben el mensaje '*x*', ¿Quién puede asegurar que tengan el mismo significado?

Existen diferentes enfoques para estos problemas, desde las extremadamente restrictivas hasta las extremadamente liberales. En la versión más restrictiva la comunicación estándar se establece independiente y de ninguna manera relacionada con la sociedad de agentes, consiste de una lista de términos definidos y mensajes sin ninguna posibilidad de interpretación; cada mensaje posible tiene una sintaxis y significado definidos y prácticamente no hay posibilidad de extender esta comunicación con mensajes personalizados o nuevos sin perder compatibilidad. La versión más liberal prácticamente no depende de mensajes fijos, los agentes envían mensajes arbitrarios a partir de una gramática estandarizada de acuerdo a una ontología, esperando que el otro agente entienda; esto plantea un gran desafío para cualquier desarrollador dado que prácticamente no cuenta con nada. La solución más eficaz parece ser una combinación de ambos enfoques, utilizando un conjunto estandarizado de mensajes (registro, notificación, consulta, etc.) y una ontología cambiante, esta solución se puede adaptar a nuevas situaciones, el agente recibe un mensaje que entiende y adopta la ontología listada en ese mensaje; de este modo "aprende" el nuevo lenguaje y puede comunicarse con el agente sin necesidad de programación adicional o tener congruencia con una interfaz proporcionada por la sociedad.

Al permitir que el agente aprenda nuevas ontologías (lo que lo hace abierto a cualquier tipo de cambio), añadir creencias u otras características comerciales no plantea un gran reto, una vez que pueden describirse por ontologías.

Un problema diferente se puede ver en la siguiente descripción. Si tenemos un número *n* de agentes en una sociedad, y hablan *n* diferentes lenguajes, pero dispuestos a comunicarse, existen dos posibilidades: o cada agente debe disponer de un traductor para los *n*-1 lenguajes restantes (y para cualquier otro idioma posible) o la sociedad debe ponerse de acuerdo sobre un lenguaje común. En este último caso, todos los agentes, sin importar la lengua que usen, sólo deben incluir un traductor de su propio idioma, a la lengua común, lo que los hace mucho más pequeños y más fáciles de desarrollar, sobre todo teniendo en cuenta que en una sociedad abierta hay numerosas lenguas desconocidas y/o dialectos de los que el desarrollador no tiene conocimiento en tiempo de diseño. Sin embargo, el lenguaje común debería ser predefinido de alguna manera [31].

Las ontologías plantean un problema fundamental para cualquier sistema de información/conocimiento. Algunos autores proponen una arquitectura de tres niveles para la óptima representación:

- ✓ Nivel tres (parte inferior) es la **ontología de la información**. Es independiente de los datos actuales almacenados en la ontología y almacena términos primitivos (autor, documento, etc.), estructuras de datos, métodos de acceso, etc. Este nivel puede ser reutilizado en otros dominios, ya que es sólo un diccionario de los términos más usados, con definiciones comunes.

- ✓ Nivel dos es el **dominio de la ontología** que abarca el contenido actual.

- ✓ El nivel más alto es el nivel de **ontología de la empresa**. Proporciona un contexto para la información almacenada en el dominio de la ontología (información sobre la información) para permitir un procesamiento más fácil o más preciso.

En cuanto a sistemas de WFs basados en agentes, algunos autores proponen una ontología separada de la terminología requerida por los agentes para la gestión del WF.

De acuerdo al lenguaje común referido anteriormente se han descrito varios enfoques, dos de los cuales se presentan aquí.

El término utilizado para estos estándares es **ACL** (*Agent Communication Languages*, Lenguajes de Comunicación de Agentes). Los ACLs son lenguajes de alto nivel diseñados especialmente para negociación, contratación, colaboración e intercambio de información, los requisitos básicos de las sociedades de agentes. Existen en una capa lógica por encima de los protocolos de transporte comunes como TCP/IP, HTTP o el IIOP de CORBA. Este último es a menudo utilizado para tecnología de agentes, ya que permite interacción independiente del lenguaje para componentes de *software*, incluidos los servicios, como la movilidad y persistencia. Además de especificar el conjunto de mensajes los ACLs también proporcionan las políticas de conversación, es decir, cuándo enviar el mensaje o la manera de reaccionar [31].

Algunos requisitos básicos para un ACL son:

- ✓ La **sintaxis** y **semántica** deben ser lo suficientemente expresivos para permitir interacción abstracta y solución de problemas.
- ✓ El ACL debe ser **verificable** para asegurar mensajes correctos.
- ✓ La comunicación de agentes debe ser **independiente** del dominio actual.

En un intento de mapear el lenguaje humano para comunicar agentes, muchos ACLs se basan en la teoría de discurso. No obstante los desarrolladores tienden a utilizar los conceptos tradicionales de programación. Esto trae como resultado que el remitente prepara todo el mensaje (admitiendo tiempos de espera) y luego lo envía. Algunos autores señalan que esto es una violación de los modelos de lenguaje humano donde partes de un mensaje pueden enviarse tan pronto como estén listos.

El primer intento de estandarizar la comunicación de agentes es KQML (*Knowledge Query and Manipulation Language*, Lenguaje de Consulta y Manipulación del Conocimiento). Se compone de un conjunto de mensajes de propósito general y una serie de mensajes utilizados por el sistema, por ejemplo al registrarse en una sociedad. Los mensajes disponibles pueden ampliarse fácilmente mediante la creación de nuevos mensajes, éstos no deben cumplir ningún criterio sintáctico, sin embargo esta libertad produce que KQML desarrolle rápidamente muchos dialectos incompatibles con mensajes ambiguos o desconocidos. Aunque la conveniencia de mensajes predefinidos hace que este lenguaje sea fácil de implementar, los problemas de compatibilidad son una de las principales debilidades.

La mencionada FIPA ha propuesto el segundo mayor estándar, llamado FIPA-ACL. La diferencia principal entre KQML y FIPA-ACL es que este último no tiene ningún mensaje de sistema. Comienza con un número mínimo de mensajes y un conjunto de reglas sintácticas. Cualquier mensaje de sistema debe enviarse utilizando una combinación de estos mensajes de propósito general. La extensibilidad se logra permitiendo la combinación de los mensajes existentes bajo determinadas reglas sintácticas; si todos los agentes obedecen las reglas, nuevos mensajes podrán ser entendidos por los demás sin problemas, ya que son fáciles de descomponer en mensajes básicos. Sin embargo, el módulo de lenguaje de un agente FIPA-compatible es mucho más compleja que la de un agente KQML.

Comparando ambos ACLs, una vez más, la elección se reduce a los requisitos específicos de la arquitectura. Una arquitectura abierta que utiliza el agente FIPA debe favorecer el modelo estándar FIPA. Ambientes cerrados pueden utilizar KQML dado que es mucho más fácil de aplicar.

Los proyectos originales de FIPA-ACL y KQML han tenido que luchar contra una falta de interés porque los conceptos parecían estar demasiado lejos de los procesos de negocio reales, ambos se basan en LISP que no es ampliamente aceptado para modelar entornos empresariales. En respuesta a esto, proyectos más recientes se basan en la tecnología XML de amplia aceptación, incluido BRML (*Business Rule Markup Language*, Lenguaje de Marcado para Reglas de Negocio), para formar un dialecto de XML llamado ACML (*Agent Communication Markup Language*, Lenguaje de Marcado para Comunicación de Agentes) [34].

Una próxima mejora en la elaboración de ACLs puede surgir con el desarrollo de la web semántica, permitiendo a los agentes a "comprender" el medio ambiente que se encuentran.

3.5.3 Organización de agentes.

En sociedades de agentes hay dos conceptos centrales que pretenden facilitar su coordinación y modelado. Tal como sucede con la mayoría de los temas de organización, también son tomados de las sociedades humanas.

El primer concepto es el de **roles**. Los roles pueden definirse como modelos de comportamiento; en un sistema social, ellos describen el comportamiento esperado y externamente perceptible de uno o más miembros.

Aunque la teoría de agentes se centra en *software* que funciona uniformemente, en general, uno debe tener en cuenta que los agentes BDI (creencias-deseos-intenciones) u otro tipo de agentes "emocionales" pueden comportarse de manera diferente bajo un mismo rol (por razones de calificación, motivación, etc.) En los casos más graves los agentes pueden incluso estar violando sus funciones (roles), la sociedad debe proporcionar los mecanismos para manejar este tipo de comportamiento.

Una sociedad de agentes ofrece múltiples roles, dependiendo principalmente del dominio en el que está integrado. Varias sociedades de agentes se han desarrollado utilizando su propia nomenclatura. Algunos servicios básicos descritos anteriormente pueden ser aplicados como instancias estáticas de programa o utilizar agentes con roles. La siguiente lista contiene los roles más comunes en sistemas de WFs basados en agentes o sistemas de gestión del conocimiento:

- ✓ *Gatekeeper* / *Administrador de Dominio* (KAoS) (*Knowledgeable Agent-oriented System*, Sistema Orientado a Agentes Bien Informados) / *Coordinador de Área*: Dependiendo de la forma de coordinación o del grado de apertura deseada, la sociedad podría requerir un agente o institución encargada de la admisión, el *Gatekeeper* asume esta responsabilidad al permitir entrar o salir a los agentes y mantener una lista de los agentes actuales del sistema.

- ✓ *Matchmaker* (KAoS) / *Agentes de Sección Amarilla* / *Coordinador de Roles* (WONDER)(*Workflow on Distributed Environment*, *Workflow* en Entornos Distribuidos): Los agentes de este rol se encargan de mantener o acceder a las listas de los servicios ofrecidos por los agentes en una sociedad. Pueden ser consultados para ubicar un servicio específico y son capaces de encontrar agentes que lo ofrecen.

- ✓ *Proxy-Agent* (KAoS) / *Agente de Cooperación de Dominio*: Agentes responsables de la comunicación intra-sociedad (agente a agente). Estas pueden ser necesarias cuando por ejemplo, se necesita traducción o cuando los agentes están distribuidos y no pueden establecer contacto directo. Los agentes-proxy reciben mensajes, posiblemente los procesan y los trasladan al destino original. Además puede haber ocasiones en las que un agente necesita direccionar a muchos otros agentes; en lugar de repetir el mensaje él mismo, este trabajo se puede dejar a una instancia especializada

- ✓ *Agente de Mediación* (KAoS) / *Agente de Interacción* (CAGIS)(*Cooperative Agents In Global Information Space*, Agentes Cooperativos en Espacios de Información Global): Estos agentes son responsables de la comunicación inter-sociedad o *inter-workspace* (CAGIS). La distinción entre comunicación *intra-* e *inter-workspace* es difícil de notar. Se puede definir la comunicación *inter-workspace*

como aquella que tendrá lugar entre sociedades independientes (traducción de ontologías, etc), mientras que la comunicación *intra-workspace* simplemente se refiere a entornos distribuidos de agentes compartiendo una gran sociedad.

- ✓ *Agente de Transporte* (KAoS): Agentes responsables de la movilidad de otros agentes de en entornos distribuidos. Esto puede incluir servicios persistentes, mecanismos de *backup* y serialización, etc.

- ✓ *Propietario* o *solicitante* de información/productos: Estos son los roles más abstractos. Los agentes adoptan este rol al entrar en una sociedad, ya sea para ofrecer o solicitar servicios.

- ✓ *Autoridad cambiaria*, *Notario* (También pueden verse como instituciones): Ambientes comerciales pueden requerir una autoridad que sea fiable para solicitantes y ofertantes a fin de completar fielmente las transacciones.

Además de estos roles comunes toda arquitectura define roles especiales. MOMIS (*Mediator envirOnment for Multiple Information Systems*, Entorno Mediador para Sistemas de Información Múltiple) [35] es una arquitectura que se centra en la integración de fuentes heterogéneas de información. Como tal, el sistema requiere de mecanismos conocidos de los sistemas de bases de datos relacionales como los *schedulers*. Asimismo las consultas deben ser mapeadas al lenguaje de consulta propio del sistema. Estas tareas las realizan los *agentes de mapeo* y el *planificador*. La consulta real es ejecutada por *agentes de consulta*. La fusión de los resultados de diversas fuentes requiere *agentes administrativos*.

Sistemas de WFs basados en agentes (por ejemplo, WONDER [36] o CAGIS [37]) hacen uso de un *agente de tarea* o *agente de trabajo* especial, que se encarga de la realización de una determinada tarea (ya sea por sí mismo o por supervisión de *software* o de usuarios). En WONDER los agentes de tarea (de hecho, el WF completo) son organizados por *coordinadores de casos* y *coordinadores de procesos*.

Una segunda forma importante de estructuración de las sociedades de agentes son las llamadas **agencias**. Una agencia consta de un único agente responsable, un conjunto (posiblemente vacío) de tareas que el agente responsable debe ejecutar y, un conjunto (posiblemente vacío) de sub-agencias representados por un único agente responsable.

Por último cabe señalar que el número de organismos y funciones depende no sólo del dominio, sino también de la forma de coordinación. Una estructura basada en el mercado, por ejemplo, no requiere *gatekeeper* u otros servicios de admisión. Dado que los agentes buscan independencia, no se basan en la sociedad para prestar servicios, sino que la mayoría de ellos los ofrecen por sí mismos. Sin embargo, los agentes desean evitar peligros en las transacciones y pueden, por tanto, buscar por ejemplo una autoridad cambiaria o un notario para sellar los contratos. Sociedades jerárquicas o basadas en redes, en cambio, a menudo están cerradas para agentes foráneos y por ende requieren de *gatekeepers*, *monitores* etc. Como las jerarquías son entornos controlados y supervisados con contratos y relaciones a largo plazo predefinidas, esto aminora las sospechas entre los agentes y reduce el número de autoridades necesarias para proporcionar confianza.

3.5.4 Dónde utilizar agentes.

En los sistemas de *software* modernos hay muchos campos de aplicación para agentes. Es importante mencionar que el término agente no siempre se refiere a un agente de *software*. Muchos sistemas señalan cualquier tipo de actor como un agente (agente principal, agente usuario, etc.).

La forma más conocida de agentes de *software* son *Agentes de Recursos*. Se utilizan cuando se necesita acceso a un recurso externo, por ejemplo, un disco duro, datos, Internet o algún usuario. Una aplicación fundamental es el acceso a sistemas obsoletos (*legacy systems*). Agentes especialmente diseñados que tienen interfaces para *software* existente y capacidad para traducir las consultas y los resultados al lenguaje ontología común de los agentes. Este tipo de agentes son a menudo denominados *wrappers*.

En principio el concepto parece muy útil, especialmente en el contexto de sistemas de WFs descritos aquí, que requieren acceso a otros programas. Lamentablemente, el *framework* de agentes puede no ser compatible con la "vieja" técnica. Los agentes son autónomos y, por tanto, de ninguna manera sincronizados. Múltiples agentes pueden necesitar acceso al mismo recurso al mismo tiempo; el *software* externo puede no ser capaz de manejar esta carga o no estar listo para acceso multi-usuario y multi-*threading* (por ejemplo, muchos procesadores de textos no soportan trabajar el mismo documento al mismo tiempo). En estos casos el agente de recursos también debe incluir un componente de planificación, mecanismos de sesiones para simular multi-*threading* y asegurar que él es el único agente de recursos para este *software*.

Otras aplicaciones incluyen:

- ✓ (*Personal*) *Agente de Información*: Agentes que tienen acceso a muchas fuentes de información y son conscientes de sus principales intenciones y preferencias. Su principal objetivo es consultar a sus fuentes y proactivamente proporcionar información que sea de interés para el usuario.
- ✓ *Agente de Dominio de Ontología*: Responsable de administrar las ontologías, traducir, almacenar y estructurar la memoria organizacional (conocimientos).
- ✓ *Agentes de Análisis*: Reciben información y documentos para su procesamiento, clasificación etc.
- ✓ *Agentes Proveedores de Contexto*: Hacen uso de sensores externos, dominio del conocimiento, etc. para proporcionar, por ejemplo agentes de información con su campo de trabajo. Usando información de contexto pueden ayudar a los agentes a hacer un mejor juicio sobre la información que encuentran o los roles que toman.
- ✓ Los *Sistemas de Administración de WFs* ofrecen muchas oportunidades para utilizar la tecnología de agentes (realización de tareas, la gestión del WF, estabilidad, etc.).

Teniendo en cuenta todas las posibles aplicaciones, no hay que olvidar que aún existen algunos problemas graves. Anteriormente, se mencionaron diferentes actitudes, intenciones ocultas o incompatibilidad de lenguajes.

Otro aspecto se refiere a los problemas de sistemas de agente sólo, especialmente en el contexto de sistemas de negocio:

- ✓ **Falta de coordinación**: Los sistemas de agentes son entornos distribuidos y no ofrecen prácticamente ningún control sobre los agentes únicos a menos que se establezca algún tipo de jerarquía o exista una forma de autoridad a la que los agentes son subordinados.

- ✓ **No Optimización**: Los sistemas de agentes son muy flexibles, los sistemas de WFs basados en agentes intentan hacer frente a los WFs débilmente estructurados. Las características previenen que todos los usuarios involucrados puedan hacer el seguimiento de los procesos reales. Por tanto es muy probable que el potencial/necesidad de optimización no sea utilizado.

- ✓ **Difícil de Monitorear**: Los agentes son autónomos y repartidos en diversos lugares. Prácticamente no hay forma de llevar un registro del estado actual de los trabajos a menos que los agentes reporten a cierta instancia de supervisión.

- ✓ **Aspectos de Seguridad**: Cualquier entorno distribuido necesita de comunicación. Sin embargo, en los ambientes cerrados los protocolos pueden ser codificados y permanecen desconocidos para un posible intruso. En las sociedades de agentes abiertas, uno debe tener cuidado de no sobreponer la compatibilidad con la seguridad.

- ✓ **Aspectos Legales**: Los agentes son autónomos y no son objeto de vigilancia constante; esto trae a colación la cuestión de quién será responsable de las acciones del agente. ¿Puede el agente principal ser responsable, aunque no sea consciente de las acciones del agente?

3.6 Arquitecturas de sistemas de workflows.

En los últimos años la arquitectura de sistemas de WFs ha cambiado de manera significativa. Si bien el desarrollo se inició con una arquitectura centralizada, existen varios enfoques basados en agentes disponibles en la actualidad.

Antes de discutir conceptos alternativos, deben señalarse las ventajas de la aplicación de tecnología de agentes a los sistemas de WFs:

- ✓ **Flexibilidad**: La ejecución de acciones/tareas puede basarse en la situación actual del agente, es decir circunstancias locales (recursos disponibles, etc.), contrario a predeterminarlos en tiempo de diseño.

- ✓ **Agilidad**: Pueden añadirse nuevos servicios/tareas al WF sin afectar a otras partes (agentes). No hay necesidad de bloquear, suspender o cancelar la ejecución del WF dado que las modificaciones sólo tienen efectos locales.

- ✓ **Adaptabilidad**: A diferencia de los conceptos tradicionales, los agentes pueden moverse a lo largo de WFs débilmente estructurados. Este concepto permite explícitamente al WF ser cambiado durante o después de la ejecución, por ejemplo, un proceso de aprendizaje. En sistemas de WFs anteriores estas modificaciones necesitaban cambiar las definiciones del WF actual. Sin embargo,

en sistemas basados en agentes, el WF es leído y trasladado dinámicamente a los agentes para permitir una fácil modificación, al mismo tiempo, los agentes son capaces de modificar el WF, basados en retroalimentación (ramas frecuentes, estados inalcanzables, etc.).

Desde un punto de vista más técnico las ventajas son:

- ✓ Extensibilidad **Basada en Componentes** (dependiendo de la arquitectura): Los agentes pueden estar colocados en diferentes lugares, combinados, removidos o intercambiados a voluntad. Ellos no necesitan un acceso a un motor central. La comunicación permite que varios agentes se puedan combinar en la ejecución de una única instancia del WF.

- ✓ **Manipulación de Eventos/Excepciones**: Muchos agentes móviles están escritos en lenguajes de programación que están construidos para manipular eventos y excepciones. Cada vez que ocurre una excepción durante la ejecución del WF o un agente debe ser notificado, se pueden utilizar estos manipuladores. También pueden ser utilizados para hacer frente al déficit en la coordinación entre agentes y para llamar y actualizar el estado de flujo de trabajo.

- ✓ **Instalación Remota**: No hay necesidad de instalar el sistema de WFs, un agente-marco de trabajo presente puede ser desplegado en toda la organización y utilizado por el sistema de WFs como si fuera otro agente de *software*.

- ✓ **Soporte para Dispositivos Móviles**: Los dispositivos móviles se caracterizan por tener poca memoria y poco poder de procesamiento. Los agentes generalmente tienen poco código de base y su carga es pequeña.

Los sistemas de WFs tradicionales se basan en un *motor de WF* que es responsable de la ejecución de una instancia única o varias instancias del WF; durante la ejecución sitúa el *software* necesario para la siguiente tarea, envía los datos requeridos y recibe el resultado después del procesamiento. Este tipo de arquitectura no incluye agentes. Una primera mejora fue colocar a los agentes de *software* al final de cada tarea, es decir, el agente de *software* recibe una carga de trabajo, interactúa con el usuario u otro *software* y reporta los resultados al motor de WF centralizado. Esta arquitectura se conoce como ***Arquitectura RPC***, basada en RPC (*Remote Procedure Call*, llamada a procedimiento remoto), la tecnología utilizada para la comunicación con el motor de WF.

El siguiente paso tecnológico es relajar el motor de WF, delegando su trabajo a los diferentes agentes. El resultado es el ***Modelo DartFlow***. Cuando se necesita una nueva instancia, el motor crea un nuevo agente y lo equipa con todos los datos necesarios, así como el WF completo. El agente ve el siguiente paso en el WF y toma una decisión sobre dónde ejecutarlo. Usando sus componentes de movilidad se migra a un sistema adecuado y ejecuta la tarea (que puede tardar mucho tiempo, en función de la propia tarea, interacción del usuario, etc.). Al terminar, recibe el resultado y lo almacena en su memoria interna. Luego se mueve al siguiente paso, de nuevo decide dónde ejecutarlo y emigra. De ser necesario el agente se duplica/clona a sí mismo y emigra a diferentes lugares. El motor de WF se reduce a la creación de los agentes y monitorear su ejecución y estado (utilizando gestores de eventos invocados por los agentes).

La tercera arquitectura es la llamado ***Modelo de Máxima Secuencia***. Una ruta de secuencia es un conjunto de tareas que se pueden ejecutar secuencialmente, sin divisiones o

uniones. Si la ruta de secuencia no puede incluir otra tarea para ejecutarse en forma secuencial, la ruta se denomina una ruta de máxima secuencia. Basados en esta definición el WF se analiza algorítmicamente y se divide en secuencias máximas. Cada secuencia se asigna a un agente independiente que puede comenzar inmediatamente a trabajar en su secuencia (a menos que debe esperar por alguna entrada). El motor de WF es básicamente el mismo que en el Modelo DartFlow, pero además se encarga de dividir al WF.

Al parecer, el desempeño de las tres arquitecturas es diferente. Si bien la arquitectura RPC tiene que lidiar el problema de cuello de botella, el Modelo DartFlow debe hacer frente a agentes "hinchados" (sobrecargados). Ambas cargan con el WF completo, así como todos los datos requeridos, haciendo de migraciones requeridas un problema importante. Por otra parte, el Modelo de Máxima Secuencia divide un WF en muchas partes que pueden o no ser independientes, para probar y garantizar la corrección hay una gran cantidad de comunicación entre todos los agentes.

Evidentemente esto es sólo teoría, pero existen algunas implementaciones de prueba (por ejemplo WONDER).

Algunos autores presentan un enfoque más integrador. En lugar de (re)desarrollar sistemas de WFs basados en agentes, proponen añadir una capa de agentes a los sistemas existentes, los agentes forman una red de seguridad para manejar condiciones de falla durante la ejecución de la instancia. En determinadas situaciones el sistema de WFs podría requerir reiniciar el proceso completo mientras que la tecnología de agentes podría ayudar en la renegociación y reanudación de la instancia actual. Al mismo tiempo, los agentes podrían usar una interfaz predefinida para modificar la secuencia de tareas fijas, añadiendo o eliminando tareas según sea necesario. Por último los agentes podrían utilizar sus habilidades de comunicación para traducir datos/llamadas de procedimiento del formato interno del sistema de WFs al formato requerido por la aplicación externa o trabajar como balanceadores de carga [38].

Otros roles que podrían ser adoptados por los agentes en un sistema de WFs, incluyen:

- ✓ **Monitoreo de Desvíos**: Un usuario del sistema podría no estar disponible, el agente debería conocer las alternativas y podría desviar el flujo de trabajo a la nueva ubicación, o incluso dejar a un lado las tareas, si es posible.
- ✓ **Agentes de Trabajo Automatizados**: Agentes que realizan su trabajo no por eventos, sino basados en el tiempo. Por ejemplo, recordatorios o trabajos de copias de seguridad.
- ✓ **Asesores**: Agentes que asesoran a agentes de tareas sobre la manera de hacer su trabajo de manera más eficiente, procesan información del contexto (ancho de banda de la red, carga del procesador, estado de usuario, etc.) y pasarlo al agente de tarea que puede basar su decisión sobre dónde para proceder con su trabajo.

En [28] se analizan algunas arquitecturas para un WfMS temporal, destacando la administración de tiempos para el proceso, organización, y modelos de información. La utilización de un DBMS temporal puede proporcionar grandes ventajas incluso para el manejo temporal de excepciones en tiempo de ejecución, y sobre el historial de tareas completadas. Su principal contribución es la demostración de cómo algunos rasgos temporales del nivel conceptual de un WF pueden ser mapeados sobre una arquitectura real.

Pueden existir muchas otras opciones dependiendo de arquitecturas específicas.

3.7 Colaboración y Proactividad.

En varias ocasiones se han señalado similitudes entre las sociedades humanas y de agentes. Al igual que en organizaciones humanas la **colaboración** también es relevante en organizaciones de agentes. Hay trabajos que pueden realizarse por un solo agente y trabajos que se realizan más eficientemente por más de un agente. La colaboración se define como un conjunto de prácticas o de esfuerzo intelectual para lograr un objetivo.

La colaboración se puede ver desde diferentes puntos de vista. Por un lado, permite a las entidades trabajar más eficientemente, ya que autoriza compartir recursos y conocimientos que no serían accesibles de otra forma. Sin embargo, es a menudo un toque negativo asociar la colaboración con la traición o trabajar para el enemigo. Aunque para la teoría de agentes esta perspectiva puede parecer algo extrema, no se puede negar que un agente colaborador puede dar más de lo que recibe. A pesar de que un esfuerzo de equipo en realidad debería ser juzgado por los resultados del equipo, puede haber evaluaciones individuales que no calificarían justamente el rendimiento del agente, ya que pudo haber gastado tiempo ayudando a los demás mejorando sus evaluaciones.

La colaboración puede existir en varios niveles. El más pequeño es un equipo de agentes trabajando, por ejemplo, en completar una determinada tarea en un WF. Formas más grandes abarcan desde agentes que gestionan un WF hasta agentes que negocian a través de las fronteras de la organización para concluir contratos.

La colaboración o cualquier forma de trabajo en equipo requiere un objetivo común. La teoría de agentes utiliza el término plan para hacer frente a una secuencia de acciones que a la larga llevan a un estado deseable o a lograr uno de los objetivos del agente. El concepto de *Plan Compartido* se extiende a un grupo de agentes. El plan es compartido por los miembros del equipo que trabajan diferentes partes de este plan para lograr la meta del equipo.

Cuando se modelan agentes capaces de trabajar en equipo es importante notar que necesitan un componente adicional para detectar conflictos. Los agentes deben ser conscientes de los otros miembros del grupo de trabajo a fin de no hacer mal uso de los recursos necesarios. La detección puede incluir acciones individuales.

En esta parte del documento se describen posibles patrones y dependencias de colaboración. Reflejan conceptos conocidos de muchos lenguajes de programación. Consideremos tres agentes A, B y C, participando en algún tipo de proceso colaborativo.

- ✓ ***Secuence*** (secuencia): El agente A completa una unidad de trabajo. El agente B espera los resultados antes de comenzar su propia tarea.
- ✓ ***Function Definition*** (definición de función) (no hay colaboración por sí misma,): Un agente recibe un patrón de entrada, lo procesa en una tarea y produce un patrón de salida predefinido.
- ✓ ***Function Invocation*** (invocación de función): El agente A proporciona datos en un patrón de entrada de una función. El agente B es responsable de esta función, procesa los datos y almacena los resultados en el patrón de salida. El agente A

fue suspendido durante el tiempo de ejecución de la función y ahora recibe el resultado para seguir trabajando.

- ✓ ***If-Then-Else*** (si-entonces-sino): El agente A completa una unidad de trabajo y almacena los resultados en un patrón de entrada. Dos agentes B y C lo reciben y comparan con un patrón de entrada esperado. Con base en los resultados alguno de los agentes trabaja, mientras que el otro sigue esperando una nueva entrada.
- ✓ ***Case*** (casos): Similar a If-Then-Else con más agentes implicados.
- ✓ ***Loop*** (ciclo): Un agente trabaja continuamente consigo mismo. Los resultados de un ciclo conforman la entrada del ciclo siguiente.
- ✓ ***Parallel Fork*** (procesamiento paralelo): Dos agentes inician trabajo en paralelo basados en el mismo patrón de entrada o un evento.
- ✓ ***Synchronous*** (síncrono): Un agente C fusiona el procesamiento paralelo de los agentes A y B, tan pronto como ambos agentes han entregado los datos requeridos por el patrón de entrada combinado.

Una segunda cuestión importante en sistemas modernos de información/conocimiento es la **proactividad**. La proactividad en contra de la reactividad describe el comportamiento del sistema en el que el propietario de una información no espera a ser requerido, sino que activamente notifica a otros agentes potencialmente interesados. El comportamiento reactivo tradicional, a veces es llamado paradigma maestro-esclavo.

Sistemas proactivos buenos, no sólo anticipan la información potencial de otros, sino que activamente intentan ampliar su área de conocimiento para obtener aún más datos relevantes, esto puede lograrse, por ejemplo, consultando activamente otras fuentes potenciales de información.

Sin embargo, a fin de ser proactivo un agente debe ser consciente de los otros agentes, así como de sus posibles necesidades de información. La información del contexto desempeña un papel importante, información de contexto relevante incluye ubicación, tiempo, intención del agente, otras actividades, recursos y datos obtenidos de sensores (temperatura, etc.).

El problema sigue siendo la forma de determinara los posibles agentes interesados. No se puede hacer un seguimiento de todos los demás agentes, incluyendo el contexto y la posible necesidad de información. Una alternativa para conocer a los demás agentes, es la suscripción. Los agentes interesados en información de un tema específico se registran lo que permite mantener una pequeña lista de agentes para rastrear y notificar.

3.8 Conclusiones.

En este capítulo se presenta el estado del arte de modelado de WFs, sociedades de agentes y sistemas de WFa basados en agentes. Los autores se han centrado en las tecnologías básicas y las decisiones más importantes de arquitectura, así como algunos problemas comunes. Según algunos investigadores, es necesaria investigación especial en los siguientes aspectos:

- ✓ **Proactividad:** Muchos autores mencionan a la proactividad como un requisito básico para sistemas de administración de información/conocimiento, pero actualmente hay muy poca información sobre modelos de proactividad.

- ✓ **Información de Contexto:** La información de contexto es esencial para cualquier entorno proactivo, al mismo tiempo mejora la calidad de trabajo de un agente único permitiéndole decidir sobre lugares potencialmente atractivos, información o cualquier otra ayuda que pudiera necesitar. Sin embargo, hasta el momento no se han logrado formalizar las estrategias humanas de búsqueda, lo que trae como resultado la incapacidad de transferir conceptos humanos a la teoría de agentes. De no hacerlo, los agentes seguirán siendo ineficientes para acceder a información de contexto, podrían perder información relevante sólo por que no están conscientes de otras formas de acceder a la fuente de información.

- ✓ **Comunicación de Agentes:** A lo largo de este capítulo se han presentado diversos aspectos de comunicación y organización de agentes. Sin embargo siguen faltando muchos pasos antes de que la comunicación de agentes plenamente se asemeje a la comunicación humana, sin más estandarización, las posibilidades de agentes plenamente compatibles son pocas.

- ✓ ***Workflows* temporales:** Se mencionó que utilizando DBMS temporales se pueden obtener ventajas considerables al adoptar una arquitectura temporal para el WfMS. Lamentablemente, no existe un WfMS que funcione sobre un sistema de este tipo.

En *http://www.wfmc.org/* y *http://www.e-workflow.org/* puede verse que actualmente existe gran auge por realizar investigación y desarrollo relacionado con WFs, también se muestran diversos casos de éxito, agrupados en los siguientes rubros:

- ✓ Académico.
- ✓ Financiero.
- ✓ Gubernamental.
- ✓ Salud.
- ✓ Industrias.
- ✓ Tecnología.
- ✓ Transporte.
- ✓ Otros.

4. *Workflows* en Plone 3.

http://plone.org/documentation/how-to/new-workflows-in-plone-3.

Plone 3 tiene incluido un conjunto de nuevos WFs. Cada uno es único en el sentido de que permite a los administradores del sitio gestionar el ciclo de vida del contenido, así como el proceso que sus usuarios siguen cuando crean, editan, revisan y ven contenidos.

4.1 Simple publication workflow.

Recién instalado, "*Simple Publication Workflow*" es el utilizado por "*default*". La descripción de la definición del WF dice:

- ✓ Es un WF simple que es útil para sitios web básicos.
- ✓ Los contenidos comienzan como "*private*" (privado) y pueden enviarse a "*review*" (revisar), o directamente a "*published*" (publicado).
- ✓ El creador de algún contenido puede editarlo, aún después de que ha sido publicado.

Cuando un usuario es creado, ese usuario sólo puede editar su "*profile*" y administrar los "*portlets*" que aparecen en su "*dashboard*" (pizarra). Para que el nuevo usuario pueda crear contenido, debe tener los permisos apropiados (figura 4.1).

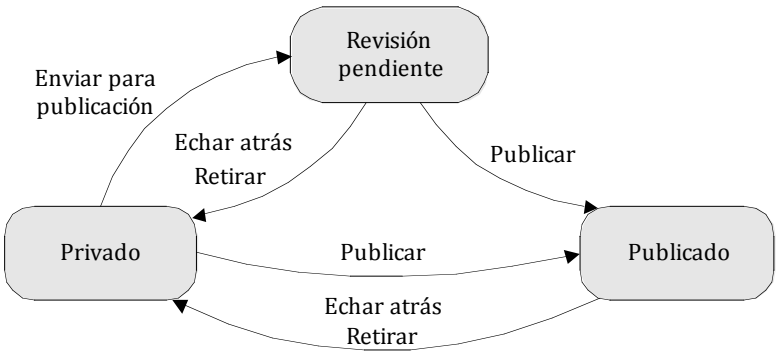

Figura 4.1: Diagrama del *Simple Publication Workflow*

4.2 Intranet/Extranet workflow.

- ✓ En un *intranet workflow* el contenido sólo es accesible si el usuario está autenticado.
- ✓ Los estados básicos son: "*Internal Draft*", "*Pending Review*", "*Internally Published*" y "*Private*".
- ✓ También tiene un estado "*Externally Published*" de tal forma que se pueda seleccionar contenido disponible a personas fuera de la intranet.

Es útil para probar WFs y ver como se ajustan a las necesidades (figura 4.2).

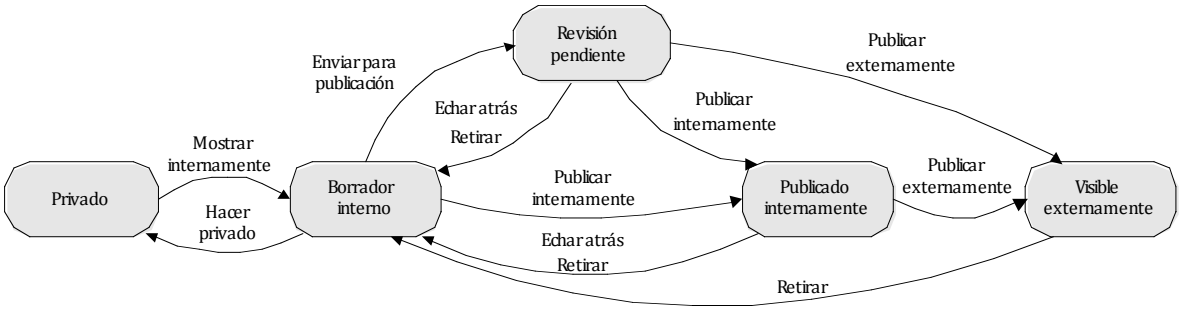

Figura 4.2: Diagrama del *Intranet/Extranet Workflow*

4.3 Community workflow.

- ✓ Los usuarios pueden crear contenido que es publicado y accesible inmediatamente.
- ✓ El contenido puede enviarse para publicación por el creador del contenido o un "*Manager*", que es típicamente hecho para promover eventos y noticias a la página principal.
- ✓ "*Reviewers*" pueden publicar o rechazar el contenido, los propietarios pueden retractar sus envíos.
- ✓ Mientras el contenido está esperando su revisión puede ser leído por todos.
- ✓ Si el contenido es publicado, sólo puede ser retractado por un "*Manager*".

Para sitios dedicados a comunidades abiertas de usuarios y se desea permitir que el público vea el contenido mientras está en proceso de consideración para publicación, el "*community workflow*" puede ajustarse a las necesidades. Este puede ser el caso de un sitio para grupos de usuarios (figura 4.3).

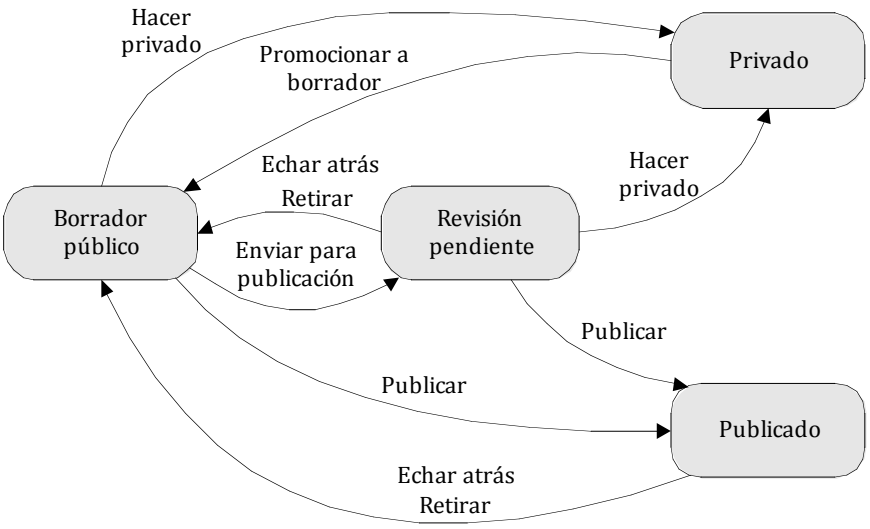

Figura 4.3: Diagrama del *Community Workflow*

4.4 One state workflow.

- ✓ Escencialmente un WF sin transiciones, pero tiene un estado "*Published*".

Es el más sencillo de todos los WFs, útil para una aplicación web personal, tal como un blog. Cuando una sola persona gestiona el contenido, no es útil tener pasos adicionales de marcado de contenido público cuando todo el contenido es creado para ser público (figura 4.4).

Figura 4.4: Diagrama del *One state Workflow*

4.5 Workflows para carpetas.

Además de los anteriores, Plone tiene por default dos WFs para carpetas *Community Workflow for Folders* y otro llamado *Intranet Workflow for Folders*, que funcionan de forma similar, pero se aplican generalmente a contenidos de tipo carpeta.

4.6 Workflows personalizados.

Los WFs son mapas de procesos de negocio. Se componen de tareas y sus relaciones, criterios para indicar inicio y término de los procesos y la información sobre la tarea individual.

Un WF es un conjunto de interacciones que deben realizarse para completar una actividad o tarea. Por ejemplo, solicitudes de admisión en una escuela, o solicitudes de viaje de alguna institución. Estas tareas involucran a varias personas, y son diferentes dependiendo de la institución.

Gráficamente se puede representar como una máquina de estados con transiciones etiquetadas (figura 4.5).

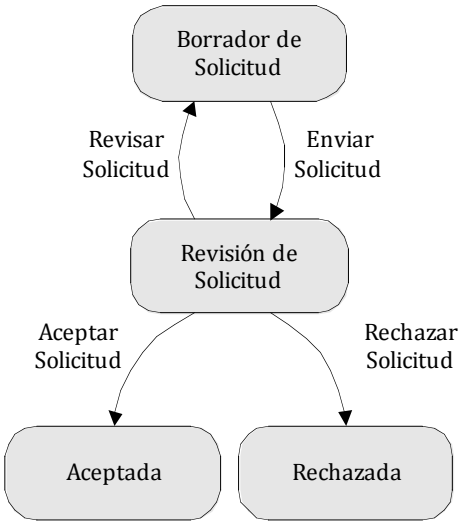

Figura 4.5: Ejemplo de un *workflow* sencillo, representado como máquina de estados.

En general, para diseñar un WF, se realizan tres actividades:

- ✓ Definir los estados del WF.
- ✓ Definir las transiciones.
- ✓ Definir permisos (especificar quién puede realizar las transiciones).

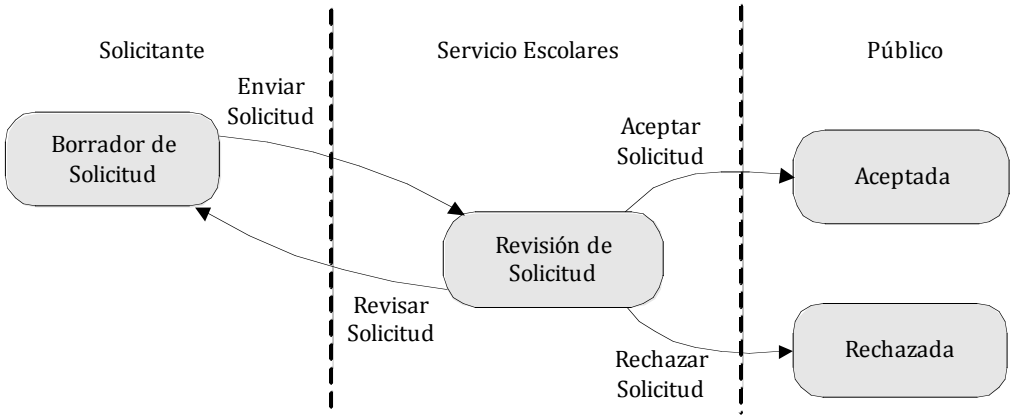

Figura 4.6: El mismo *workflow*, especificando los permisos sobre las transiciones y los estados.

En Plone, además de lo anterior, se pueden definir:

- ✓ *Worklist*, se utilizan para notificar a algún usuario que existe un contenido que se encuentra en un estado específico en espera para completar su tarea.

- ✓ *Scripts*, son una característica muy útil, ya que permiten realizar ciertas tareas cuando un contenido pasa de un estado a otro, por ejemplo, enviar un correo de aviso.

En Plone, una vez que se ha definido y creado el tipo de contenido, los pasos para que tenga un WF asociado, se describen a continuación.

Crear la definición del WF.

- ✓ Estados y estado inicial.
- ✓ Transiciones, asociando los *scripts* (si utiliza alguno).
- ✓ Permisos.
- ✓ *Worklists* (si se requieren).
- ✓ Crear una función que instale los WFs en el servidor y lo asocie a algún contenido.

Si se van a utilizar scripts en alguna transición:

- ✓ Definir lo que va a realizar el script.
- ✓ Crear una función que instale el script en el servidor y lo asocie a algún workflow.

En el archivo de instalación del contenido:

- ✓ Ejecutar la función de instalación de los WFs.
- ✓ Ejecutar la función de instalación de los *scripts*.

Algunos problemas que se pueden presentar en esta parte, son:

- ✓ Si no está completamente bien definido el contenido, se podrían requerir datos adicionales que modifiquen el flujo que seguirá.
- ✓ No definir bien los permisos desde el principio, complicándose cuando se aumentan el número de estados y/o transiciones.
- ✓ Algunos *scripts* podrían requerir permisos adicionales para poder ejecutarse, no permitiendo a algún usuario realizar la transición, aun si tiene el permiso de ejecutarla.

5. Análisis y diseño del producto Solicitudes.

5.1 Introducción.

La metodología a utilizar en el desarrollo del Producto Solicitudes del IMATE, será la basada en componentes[1], propuesta por Pressman [39], ya que se adapta muy bien a la forma de desarrollo de productos para Plone:

- ✓ Establecer los requisitos del sistema.
- ✓ Establecer un diseño arquitectónico.
- ✓ Examinar componentes existentes para integrarlos al sistema.
- ✓ Modificar o eliminar aquellos requisitos que no se puedan implementar con los componentes existentes.
- ✓ De ser necesario, desarrollo de componentes necesarios no existentes.
- ✓ Integración de los componentes con el Sistema InfoMatem.

5.2 Requerimientos.

Los requisitos del producto se dividieron en requerimientos generales y requerimientos por tipo de usuario. Finalmente se muestran casos de uso correspondientes a los requerimientos obtenidos.

5.2.1 Requerimientos generales.

A continuación se enlistan los requerimientos generales del producto Solicitudes:

1. Realizar y administrar las solicitudes desde el sitio InfoMatem [1].
2. Se tienen tres tipos diferentes de solicitudes controladas por la Secretaría Académica del IMATE, y se generará un tipo de contenido para cada una:
   a) Solicitud de Licencia Comisión.
   b) Solicitud de Becario.
   c) Solicitud de Visitante.
3. El nombre (identificador) del contenido dentro del sitio debe tener un formato estándar y generarse automáticamente, sin permitir que el usuario lo modifique.
4. El producto debe permitir adjuntar archivos a la solicitud, para acompañarla de los documentos necesarios.
5. Cada tipo de contenido tendrá un *workflow* diferente al que se tiene por *default* en Plone y será acorde con el proceso que se está modelando.
6. Los contenidos realizados deben estar internacionalizados de acuerdo al estándar i18n, utilizado por Plone para realizar traducciones.

[1] Se utilizará *producto* en lugar de *componente*, dado que es la terminología utilizada en Plone.

7. Una vez que el solicitante envía su solicitud, se le presentará un mensaje de acuse de recibo, que puede imprimir para cualquier aclaración respecto a la misma.

8. En todo momento el propietario puede ver el estado actual de la solicitud, así como la fecha en la cual se realizó la última transición.

9. Las transiciones entre los estados del *workflow* se deberán realizar con botones, para darle mayor facilidad de uso al producto.

10. En cualquier estado del *workflow* puede retirarse la solicitud, excepto si ya ha sido aprobada.

11. El formulario debe ser minimalista, es decir presentar por *default* solo los campos de captura estrictamente necesarios y los no necesarios ocultarlos hasta que se vayan a utilizar.

12. Las cantidades solicitadas y la autorizada pueden contener comas para separar miles (esto no lo hace por default Plone).

13. Se debe llevar la suma de gastos acumulados en el año por cada académico solicitante.

14. Existirán algunos campos no accesibles (para edición) al propietario de la solicitud.

15. Se podrán hacer comentarios al contenido que no sean visibles al propietario de la solicitud.

16. Dado que uno de los objetivos de la presente tesis es obtener un producto reutilizable para modelar otros procesos administrativos, los *workflows* utilizados deberán estar bien documentados, mostrando claramente tanto la forma de modelado como su implementación e instalación en Plone.

5.2.2 Tipos de usuarios.

En Plone se definen varios tipos de usuarios, para el producto solicitudes, se tienen los siguientes tipos:

- ✓ **Solicitante**. Es la persona que realiza la solicitud.
- ✓ **Administrador de la Comisión Especial**. Es un actor intermedio dentro del *workflow*, tiene permisos diferentes al solicitante.
- ✓ **Miembro del Consejo Interno**. Es un actor que pertenece al Consejo Interno del IMATE y se encarga de realizar la última transición del *workflow*.

Nota: Estos tipos de usuarios se mapean con roles predefinidos en Plone, para no afectar el rendimiento del sitio.

Cada usuario tiene diferentes permisos dependiendo del estado actual del contenido, tanto en transiciones como en campos editables, a continuación se muestran los diagramas de estados de los dos *workflows* utilizados en el producto.

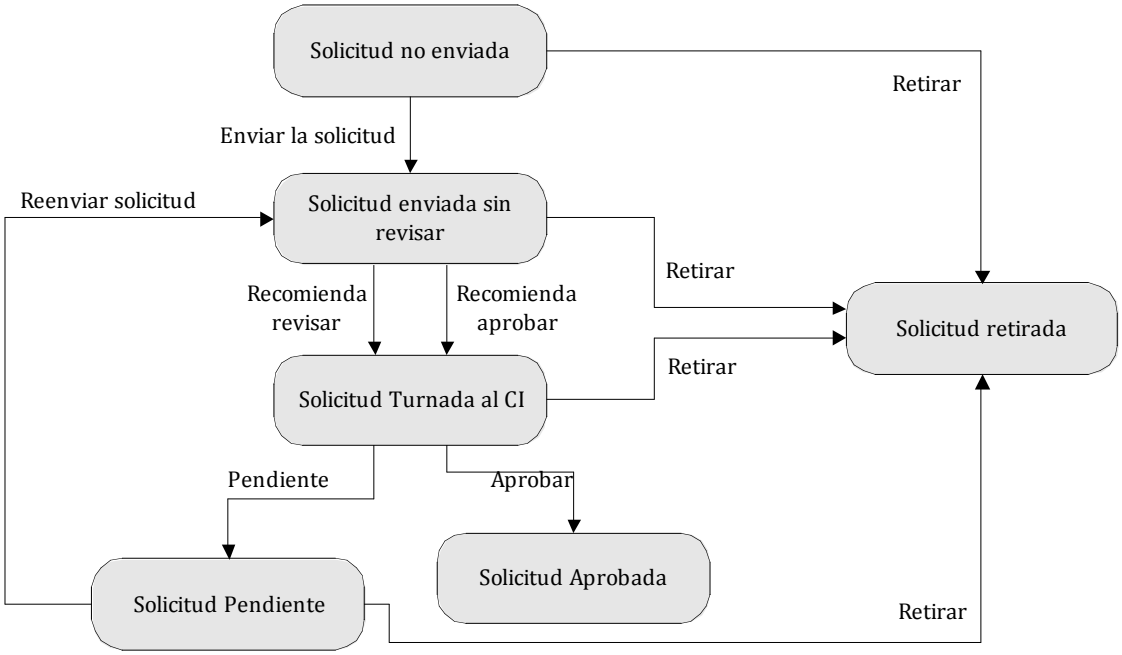

Figura 5.1: *Workflow* de solicitud.

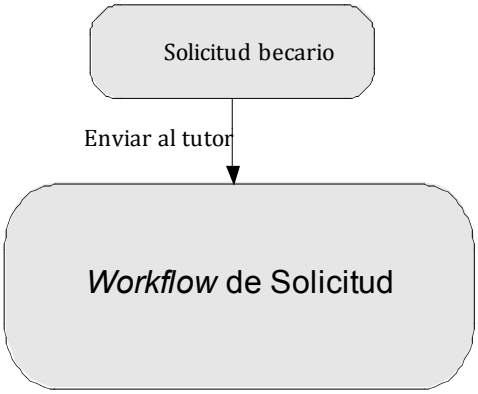

Figura 5.2: *Workflow* de solicitud de becario.
(invocando a una instancia de *workflow* de solicitud)

5.2.3 Requerimientos por tipo de usuario.

Solicitante.

- ✓ Puede crear la solicitud, guardarla y editarla, mientras no la envíe a revisión.
- ✓ Puede retirar la solicitud en los estados *Solicitud no enviada* y *Solicitud pendiente*.
- ✓ Puede revisar el estado de sus solicitudes realizadas.
- ✓ Puede pedir que se retire alguna solicitud que se encuentre en los estados *Solicitud enviada sin revisar* y *Solicitud turnada al CI*, pero debe hacerlo por algún otro medio (por ejemplo e-mail).

Administrador de la Comisión Especial.

- ✓ Tiene acceso al contenido en el estado *Solicitud enviada sin revisar*.
- ✓ Puede realizar las transiciones *Recomienda aprobar* y *Recomienda revisar*, que llevan al estado *Solicitud turnada al CI*.
- ✓ Añade a la solicitud la cantidad que recomienda para asignación, este campo sólo puede ser modificado por este tipo de usuario.
- ✓ Puede retirar una solicitud, previa petición del propietario por algún otro medio (e-mail).

Miembro del Consejo Interno.

- ✓ Tiene acceso al contenido en el estado *Solicitud turnada al CI*.
- ✓ Puede realizar las transiciones *Aprobar*, que lleva al estado *Solicitud aprobada* (estado final) y *Poner pendiente*, que lleva al estado *Solicitud pendiente*.
- ✓ Determina la cantidad aprobada y la fecha en que se aprobó (si es el caso), estos campos sólo pueden modificarse por este tipo de usuario.
- ✓ Determina que comentarios pueden ser visibles por el propietario de la solicitud y cuáles no.
- ✓ Puede retirar una solicitud, previa petición del propietario por algún otro medio (e-mail).

5.2.4 Casos de uso.

Los casos de uso describen la forma en que los actores van a interactuar con el software bajo determinadas situaciones. A continuación se muestran los casos de uso indicando los actores que lo pueden realizar y una descripción.

- ✓ **Crear Solicitud.**

Actor: Solicitante.

Descripción: Un usuario autentificado dentro del sitio crea una nueva solicitud (aplica a los tres tipos de solicitud).

Flujo 1:

Actor	Sistema
En su carpeta personal, da clic en *Agregar nuevo … → solicitud X*.	Presenta el formulario correspondiente al tipo de solicitud seleccionado.
Llena el formulario y da clic en *Guardar*.	Guarda la solicitud, con el nombre estandarizado y los permisos determinados.

Flujo 2:

Actor	Sistema
En su carpeta personal, da clic en *Agregar nuevo … → solicitud X*.	Presenta el formulario correspondiente al tipo de solicitud seleccionado.
No llena el formulario y da clic en *Cancelar*.	Elimina la solicitud para no almacenar contenidos "basura".

- ✓ **Enviar Solicitud al tutor.**

Actor: Solicitante.

Descripción: El propietario de la solicitud (becario) la envía a su tutor para que la revise (aplica a *Solicitud de becario*).

Actor	Sistema
Da clic en *Mis solicitudes*.	Presenta una lista con todas las solicitudes que pertenecen al usuario actual.
Selecciona de la lista la solicitud que desea enviar (que se encuentre en el estado *Solicitud becario*).	Presenta la información de la solicitud seleccionada, con los botones *Enviar al tutor* y *Retirar*.
Da clic en *Enviar al tutor*.	Cambia el estado actual del contenido a *Solicitud no enviada* y envía un aviso al tutor.

- ✓ **Enviar Solicitud.**

Actor: Solicitante.

Descripción: El propietario de la solicitud la envía a revisión por la Comisión Especial (aplica a los tres tipos de solicitud).

Actor	Sistema
Da clic en *Mis solicitudes*.	Presenta una lista con todas las solicitudes que pertenecen al usuario actual.
Selecciona de la lista la solicitud que desea enviar (que se encuentre en el estado *Solicitud no enviada*).	Presenta la información de la solicitud seleccionada, con los botones *Enviar solicitud* y *Retirar*.
Da clic en *Enviar solicitud*.	Cambia el estado actual del contenido a *Solicitud enviada sin revisar*.

- ✓ **Reenviar Solicitud.**

Actor: Solicitante.

Descripción: El propietario de la solicitud la envía a revisión por la Comisión Especial, después de estar pendiente (aplica a los tres tipos de solicitud).

Actor	Sistema
Da clic en *Mis solicitudes*.	Presenta una lista con todas las solicitudes que pertenecen al usuario actual.
Selecciona de la lista la solicitud que desea enviar (que se encuentre en el estado *Solicitud pendiente*).	Presenta la información de la solicitud seleccionada, con los botones *Reenviar solicitud* y *Retirar*.
Da clic en *Reenviar solicitud*.	Cambia el estado actual del contenido a *Solicitud enviada sin revisar*.

- ✓ **Retirar Solicitud** (Solicitante)**.**

Actor: Solicitante.

Descripción: El propietario de la solicitud la retira (aplica a los tres tipos de solicitud).

Actor	Sistema
Da clic en *Mis solicitudes*.	Presenta una lista con todas las solicitudes que pertenecen al usuario actual.
Selecciona de la lista la solicitud que desea enviar (que se encuentre en el estado *Solicitud no enviada* ó *Solicitud pendiente*).	Presenta la información de la solicitud seleccionada, con los botones *Enviar solicitud* ó *Reenviar solicitud* y *Retirar*.
Da clic en *Retirar*.	Cambia el estado actual del contenido a *Solicitud retirada*.

- ✓ **Recomendar aprobación.**

Actor: Administrador de la Comisión Especial.

Descripción: El administrador de la comisión especial revisa la solicitud, emite una recomendación de aprobar y la turna al Consejo Interno (aplica a los tres tipos de solicitud).

Actor	Sistema
Da clic en *Solicitudes pendientes de revisar*.	Presenta una lista con las solicitudes que se encuentren en el estado *Solicitud enviada sin revisar*.
Da clic en alguna de las solicitudes de la lista.	Presenta la información de la solicitud seleccionada, con los botones *Recomienda aprobar*, *Recomienda revisar* y *Retirar*.
Especifica la cantidad recomendada y da clic en el botón *Recomienda aprobar*.	Cambia el estado actual del contenido a *Solicitud turnada al CI*.

- ✓ **Recomendar revisión.**

Actor: Administrador de la Comisión Especial.

Descripción: El administrador de la comisión especial revisa la solicitud, emite una recomendación de revisión y la turna al Consejo Interno (aplica a los tres tipos de solicitud).

Actor	Sistema
Da clic en *Solicitudes pendientes de revisar*.	Presenta una lista con las solicitudes que se encuentren en el estado *Solicitud enviada sin revisar*.
Da clic en alguna de las solicitudes de la lista.	Presenta la información de la solicitud seleccionada, con los botones *Recomienda aprobar*, *Recomienda revisar* y *Retirar*.
Da clic en el botón *Recomienda revisar*.	Cambia el estado actual del contenido a *Solicitud turnada al CI*.

- ✓ **Retirar Solicitud** (Administrador de la Comisión Especial)**.**

Actor: Administrador de la Comisión Especial.

Descripción: El administrador de la comisión especial retira la solicitud previa petición del propietario (aplica a los tres tipos de solicitud).

Actor	Sistema
Recibe la petición de retiro de la solicitud por parte del propietario (vía e-mail) y da clic en *Solicitudes pendientes de revisar*.	Presenta una lista con las solicitudes que se encuentren en el estado *Solicitud enviada sin revisar*.
Da clic en la solicitud que se desea retirar.	Presenta la información de la solicitud seleccionada, con los botones *Recomienda aprobar*, *Recomienda revisar* y *Retirar*.
Da clic en *Retirar*.	Cambia el estado actual del contenido a *Solicitud retirada* y envía un aviso al propietario.

- ✓ **Aprobar.**

Actor: Miembro del Consejo Interno.

Descripción: El consejo interno revisa la solicitud, determina la cantidad y aprueba la solicitud (aplica a los tres tipos de solicitud).

Actor	Sistema
Da clic en *Solicitudes turnadas al CI*.	Presenta una lista con las solicitudes que se encuentren en el estado *Solicitud turnada al CI*.
Da clic en alguna de las solicitudes de la lista.	Presenta la información de la solicitud seleccionada, con los botones *Aprobar, Pendiente* y *Retirar*.
Especifica la cantidad y fecha de aprobación y da clic en el botón *Aprobar*.	Cambia el estado actual del contenido a *Solicitud aprobada*.

- ✓ **Poner pendiente.**

Actor: Miembro del Consejo Interno.

Descripción: El consejo interno revisa la solicitud, determina algún problema y pone pendiente la solicitud (aplica a los tres tipos de solicitud).

Actor	Sistema
Da clic en *Solicitudes turnadas al CI*.	Presenta una lista con las solicitudes que se encuentren en el estado *Solicitud turnada al CI*.
Da clic en alguna de las solicitudes de la lista.	Presenta la información de la solicitud seleccionada, con los botones *Aprobar*, *Pendiente y Retirar*.
Da clic en el botón *Poner pendiente*.	Cambia el estado actual del contenido a *Solicitud pendiente* y avisa al propietario de la solicitud.

- ✓ **Retirar Solicitud** (Miembro del Consejo Interno)**.**

Actor: Miembro del Consejo Interno.

Descripción: El miembro del consejo interno retira la solicitud previa petición del propietario (aplica a los tres tipos de solicitud).

Actor	Sistema
Recibe la petición de retiro de la solicitud por parte del propietario (vía e-mail) y da clic en *Solicitudes turnadas al CI*.	Presenta una lista con las solicitudes que se encuentren en el estado *Solicitud turnada al CI*.
Da clic en la solicitud que se desea retirar.	Presenta la información de la solicitud seleccionada, con los botones *Aprobar*, *Pendiente* y *Retirar*.
Da clic en *Retirar*.	Cambia el estado actual del contenido a *Solicitud retirada* y envía un aviso al propietario.

5.3 Arquitectura utilizada.

5.3.1 Arquitectura física.

En la figura 5.3, se muestra la arquitectura física que utiliza InfoMatem [1] sobre el cuál trabajará el producto Solicitudes.

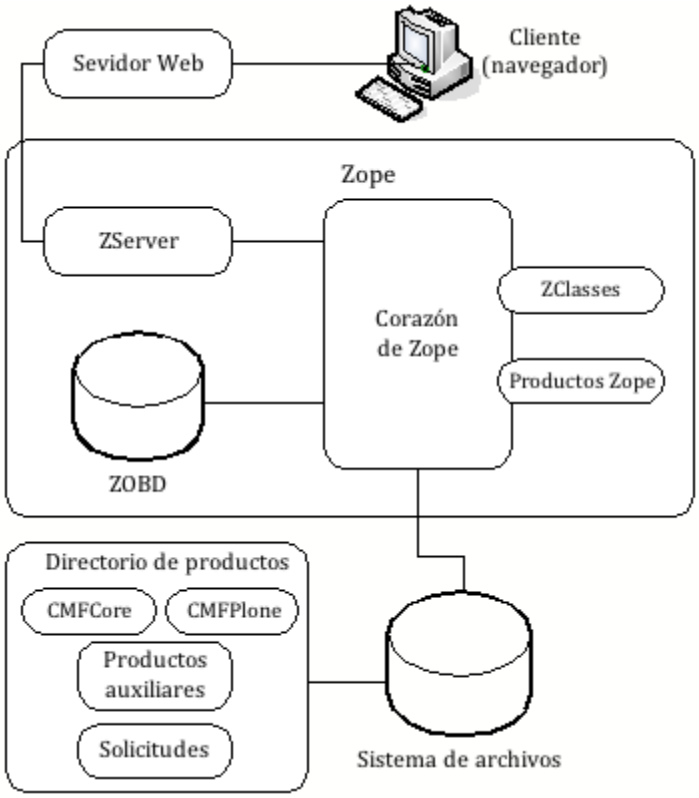

Figura 5.3: Arquitectura física de InfoMatem.

Plone trabaja sobre Zope y CMF (*Content Management Framework*, Marco de trabajo para administración de contenidos). Tanto CMFPlone (o simplemente Plone) como CMFCore (CMF) son productos de Zope, pero Plone depende de CMF para poder instalarse. Además, se tienen muchos otros productos auxiliares para Plone.

5.3.2 Arquitectura de capas.

Plone permite desarrollar productos por capas, de esta forma se logra separar la presentación de la lógica del negocio y la gestión de los datos, una gran ventaja del desarrollo por capas, es que se puede modificar alguna de las capas sin afectar a las otras. La figura 5.4 muestra las capas utilizadas por el producto Solicitudes.

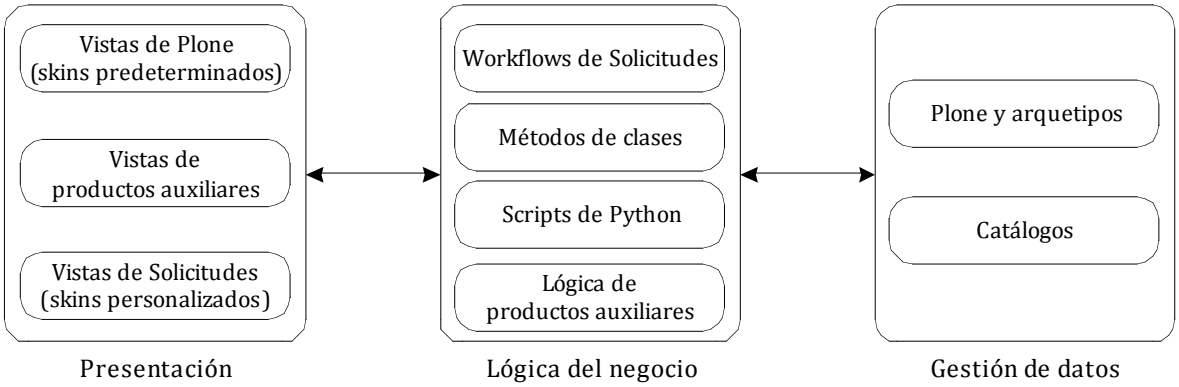

Figura 5.4: Arquitectura de capas del producto Solicitudes.

5.3.2.1 Capa de presentación.

Es la parte encargada de mostrar la aplicación a los usuarios (*front end*). Esta capa incluye código y otros recursos que permiten definir la interface con el usuario (como texto HTML e imágenes). Esta capa está conformada por vistas predefinidas por Plone; además se implementaron otras vistas utilizando plantillas de Zope (ZPT).

Dentro de Plone, las vistas no se encuentran ligadas a ningún objeto en particular, pero dentro de cada definición de contenido, se deben configuran las vistas a utilizar por cada acción, por ejemplo, se debe definir la vista a utilizar para ver y para editar un contenido.

A través de las vistas se pueden enviar mensajes a los objetos del sitio, siempre y cuando los mensajes enviados sean conocidos por los objetos que los reciben. Es decir, los objetos deben tener implementados los métodos invocados por las vistas para que pueda haber comunicación entre ambos.

5.3.2.1 Capa de lógica del negocio.

Dentro de esta capa se encuentra sólo código relacionado con la lógica del negocio. Idealmente esta capa debería ignorar el código utilizado por la capa de presentación. La capa de presentación le envía mensajes a los objetos del sitio para que realicen ciertas acciones.

La lógica del negocio del producto Solicitudes, se encuentra conformada de la siguiente forma:

- ✓ *Workflows* de solicitudes. Determinan la forma en la que se comportarán las solicitudes dentro del sitio.
- ✓ Métodos de clases. Cada objeto dentro del sitio está implementado mediante una o más clases, dentro de las que se pueden tener varios métodos.
- ✓ Scripts de Python. Para el producto Solicitudes se crearon varios scripts de Python que permiten definir parte de la lógica del negocio.
- ✓ Lógica de otros productos. Se aprovecharon algunos scripts y métodos implementados dentro de otros productos.

5.3.2.1 Capa de gestión de datos.

Esta capa, se encarga de interactuar con el manejador de base de datos para guardar o recuperar datos. Permite encapsular las especificaciones de la base de datos (ZODB, MySQL, Oracle, archivos de texto, etc) a la siguiente capa.

Plone y los arquetipos se encargan de administrar esta capa, haciéndolo transparente para el programador. Cada que se agrega o se modifica un contenido creado con arquetipos, de forma transparente se avisa a Plone para que guarde el objeto tal cual en la ZOBD, como si fuera una fotografía, dentro de la base de datos. Cuando se quiere acceder al objeto, Plone se encarga de recuperarlo de la base de datos y cargarlo. Para buscar objetos dentro de la base de datos y recuperarlos de manera eficiente se utilizan catálogos que indexan los contenidos del sitio.

5.4 Productos existentes.

Existen cientos de productos disponibles de forma gratuita en el sitio de Plone [17]. El producto Solicitudes, hace uso de algunos de estos productos debido a que proporcionan buena funcionalidad y facilitan el desarrollo de productos personalizados. A continuación se enlistan los productos utilizados:

- ✓ ***CMFCore***. Este producto define y proporciona muchas características muy útiles en Plone y que a su vez pueden utilizarse en otros productos desarrollados. Entre las funciones que se tomaron para el desarrollo, están: localizar objetos de forma única dentro del sitio, revisar los permisos que tiene el usuario actual del sitio, de acuerdo a su(s) rol(es), que es una característica muy útil cuando se están definiendo y utilizando *workflows*.

- ✓ ***FacultyStaffDirectory***. Es un producto desarrollado en la *Pennsylvania State University*, como parte del proyecto *WebLion*, entre otras cosas, provee las siguientes funcionalidades: un directorio del personal, espacios de trabajo compartidos para los comités, una forma de hacer el seguimiento de las áreas de investigación. Se integra con la infraestructura de usuarios y grupos de Plone y soporta un marco de trabajo de extensibilidad para requerimientos personalizados.

- ✓ ***Archetypes***. Este producto permite definir nuevos contenidos, editarlos, visualizarlos, indexarlos en uno o varios catálogos, navegar a través de ellos y proporciona varias opciones para manejarlos.

- ✓ ***ATCountryWidget, MasterSelectWidget***. Permiten utilizar campos y *widgets* dentro de los esquemas de contenidos, estos productos permiten visualizar y editar las solicitudes de forma más amigable.

- ✓ ***SimpleAttachment***. Permite adjuntar archivos a contenidos de forma sencilla, los archivos añadidos heredan los permisos de sus contenedores.

- ✓ ***qPloneComments***. Es un producto desarrollado para mejorar el mecanismo de comentarios estándar de Plone, permite administración de comentarios (en cuanto a la visibilidad de los mismos).

- ✓ ***collective.workflowed***. Este producto desarrollado por Carlos de la Guardia [40], es un editor de *workflows* gráfico que permite: arrastrar y soltar los elementos visuales; modificar estados, transiciones y propiedades sin tener que cambiar de pestañas (como se hace en el ZMI) y el resultado es visible inmediatamente en el diagrama.

- ✓ ***plone.portlet.collection***. Es un producto que sirve para crear de forma sencilla *portlets* que contengan objetos de tipo *collection* de Plone, un *collection*, es un tipo de contenido que sirve para realizar búsquedas en base a algunos parámetros fácilmente configurables.

- ✓ ***plone.portlet.static***. Sirve para crear de forma sencilla *portlets* que contengan texto fijo, que puede incluir etiquetas de HTML, como ligas y, gracias al editor enriquecido incluido en Plone (*Kupu*), permite encontrar sencillamente contenidos dentro del sitio.

5.5 Producto Solicitudes.

Dado que los productos existentes no satisfacen todos los requerimientos definidos, se desarrolló el producto Solicitudes.

- ✓ **Solicitudes**. Este producto se encarga de la administración de las Solicitudes dentro del sitio InfoMatem: la instalación del mismo producto; los formularios utilizados para capturarlas y editarlas, las vistas adecuadas a cada tipo de solicitud (*skins*); los *workflows* con sus *scripts* y los permisos que se requieren para acceder al contenido en cada estado; y los archivos utilizados para las traducciones, acordes con el estándar i18n. Además, está integrado con el producto *FacultyStaffDirectory*, permitiendo añadir las solicitudes dentro de las carpetas creadas a partir del mismo.

Este producto servirá de base para implementar otros procesos administrativos, en primera instancia dentro del mismo IMATE, y posteriormente se buscará que pueda ser replicado también en otras dependencias de la UNAM.

6. Implementación del producto Solicitudes.

6.1 Definición de schemas.

Un *schema* es una *interface* extendida que define campos, se puede validar que los atributos de un objeto son acordes a los campos definidos en el *schema*. Con *interfaces* "simples" sólo se puede validar que los métodos son acordes con su *interface* de especificación. Así, *interfaces* y *schemas* refieren a aspectos diferentes de un objeto, su código y su estado, respectivamente.

Un *schema* es un modelo de datos en una notación formal "*legible por máquinas*", además, en Plone, dentro del *schema* se puede definir el *widget* que se encargará de presentar la información del campo. *Archetypes* es un marco de trabajo que facilita la construcción de aplicaciones para Plone y CMF; su propósito principal es proveer un método común para crear tipos de contenidos, basados en *schemas*.

A continuación se enlistan los tipos de campos definidos en *Archetypes*, entre paréntesis se indican los *widgets* que puede utilizar cada tipo:

- ✓ ***BooleanField***. Campo que almacena valores de tipo *Verdadero* o *Falso* (*LabelWidget*, *BooleanWidget*).
- ✓ ***ComputedField***. Campo de sólo lectura cuyo contenido no puede ser editado por los usuarios, en cambio, es calculado por una expresión de Python; este tipo de campo generalmente no se guarda en la base de datos, porque su contenido se calcula al momento de mostrar el objeto (*LabelWidget*, *ComputedWidget*).
- ✓ ***MFObjectField***. Campo usado para almacenar valores dentro de un objeto CMF que puede tener algún workflow; sólo puede ser utilizado por contenidos basados en BaseFolder (*LabelWidget*, *FileWidget*).
- ✓ ***DateTimeField***. Campo usado para almacenar fecha y hora (*LabelWidget*, *CalendarWidget*).
- ✓ ***FileField***. Campo que almacena grandes cantidades de datos, tales como archivos de texto, documentos, etc. (*LabelWidget*, *FileWidget*).
- ✓ ***FixedPointField***. Campo para almacenar números de punto fijo (*LabelWidget*, *DecimalWidget*).
- ✓ ***FloatField***. Campo que se utiliza para almacenar datos numéricos de punto flotante (*LabelWidget*, *DecimalWidget*).
- ✓ ***ImageField***. Campo que almacena una imagen y permite modificar su tamaño dinámicamente (*LabelWidget*, *ImageWidget*).
- ✓ ***IntegerField***. Campo usado para almacenar números enteros (*LabelWidget*, *IntegerWidget*).

- ✓ ***LinesField***. Campo que permite almacenar texto como una lista (*LabelWidget*, *KeywordWidget, LinesWidget, MultiselectionWidget, PicklistWidget, InAndOutWidget*).
- ✓ ***ReferenceField***. Campo utilizado para guardar referencias a otros objetos de *Archetypes* (*LabelWidget*, *ReferenceWidget, ReferenceBrowserWidget, InAndOutWidget*).
- ✓ ***StringField***. Campo que almacena texto plano, cadenas sin formato (*LabelWidget*, *StringWidget, SelectionWidget*).
- ✓ ***TextField***. Campo usado para guardar textos largos, cadenas multi-líneas; la cadena puede ser transformada a formatos alternativos (*LabelWidget*, *TextAreaWidget, RichWidget*).

En la siguiente lista se describes brevemente los *widgets* redefinidos en *Archetypes*:

- ✓ ***BooleanWidget***. Muestra un *checkbox* HTML, para elegir valores tipo *Verdadero*/*Falso*.
- ✓ ***CalendarWidget***. Muestra una caja de entrada con una ventana auxiliar para elegir fechas.
- ✓ ***ComputedWidget***. Generalmente usado para campos tipo *ComputedField*, muestra el valor calculado. Si el campo tiene un vocabulario, y el campo es una llave del vocabulario, el *widget* buscará la llave en el vocabulario y mostrará el resultado.
- ✓ ***DecimalWidget***. En modo de edición, muestra una caja de texto que acepta valores de punto fijo.
- ✓ ***FileWidget***. Muestra un *browser* HTML que permite subir archivos.
- ✓ ***ImageWidget***. Muestra un *browser* HTML que permite subir, desplegar, borrar y reemplazar imágenes.
- ✓ ***InAndOutWidget***. En modo de edición, permite mover artículos entre dos listas, los artículos son removidos de la lista origen.
- ✓ ***IntegerWidget***. Muestra una caja de texto de HTML simple.
- ✓ ***KeywordWidget***. Este *widget*, permite al usuario seleccionar palabras clave o categorías de una lista.
- ✓ ***LabelWidget***. Usado para mostrar etiquetas en formularios, sin valores o elementos de entrada.
- ✓ ***LinesWidget***. Despliega un área de texto para permitir al usuario ingresar una lista de valores, una por línea.
- ✓ ***MultiSelectionWidget***. Permite selección múltiple, pos default es una lista de selección HTML, pero también puede mostrarse como *checkboxes*.

- ✓ ***PasswordWidget***. Muestra una caja de texto tipo *password* de HTML.
- ✓ ***PicklistWidget***. Similar al *InAndOutWidget*, pero los valores se conservan en la lista origen, después de la selección.
- ✓ ***ReferenceWidget***. Muestra una caja de texto HTML que acepta una lista de valores de referencia.
- ✓ ***ReferenceBrowserWidget***. Un *widget* sofisticado para buscar, agregar y eliminar referencias, usando el navegador.
- ✓ ***RichWidget***. Permite ingresar texto, o subir archivos en múltiples formatos, que son transformados para desplegarse.
- ✓ ***SelectionWidget***. Muestra una lista de selección HTML, que puede ser representada como lista desplegable, ó como un grupo de botones de radio.
- ✓ ***StringWidget***. Muestra una caja de texto que acepta una línea de texto.
- ✓ ***TextAreaWidget***. Muestra un área de texto HTML para escribir algunas líneas.

Además de los *widgets* que provee *Archetypes*, en el desarrollo se utilizan otros *widgets* definidos en productos de terceros:

- ✓ ***AttachmentsManagerWidget***, de un producto llamado *SimpleAttachment*, permite adjuntar archivos a contenidos de tipo *Folder*, de forma sencilla.
- ✓ ***CountryWidget***, de un producto llamado *ATCountryWidget*, proporciona un listado con continentes, países y ciudades, bajo el estándar *ISO-country*.
- ✓ ***MasterSelectWidget***, de un producto del mismo nombre, permite mostrar/ocultar otros campos, dependiendo de la opción seleccionada.

Cómo se dijo anteriormente, en el producto ***Solicitudes***, se tienen tres tipos de solicitudes, y cada una tiene un *schema* definido; a continuación se muestran tablas con los campos de cada una, así como una pequeña descripción de cada campo y el *widget* que utiliza cada uno.

Nombre	Tipo	Descripción	Widget
title	*ComputedField*	Título, identificador del objeto, su valor se calcula y no se permite que el usuario lo modifique.	*ComputedWidget*
description	*ComputedField*	Descripción del objeto, su valor se calcula y no se permite que el usuario lo modifique, se utiliza para mostrar el estado actual de la solicitud sin tener que abrirla.	*StringWidget*

pais	*StringField*	El país correspondiente a la solicitud, para invitados, es el país de origen.	*CountryWidget*
ciudad_pais	*StringField*	La ciudad correspondiente a la solicitud, para invitados, es la ciudad de origen.	*StringWidget*
institucion	*StringField*	La institución correspondiente a la solicitud, para invitados, es la institución de origen.	*StringWidget*
fecha_desde	*DateTimeField*	Fecha de inicio de la solicitud.	*CalendarWidget*
fecha_hasta	*DateTimeField*	Fecha de término de la solicitud.	*CalendarWidget*
objeto_viaje	*StringField*	El objetivo del viaje.	*TextAreaWidget*
investigacionarea	*LinesField*	El área de investigación del solicitante ó invitado.	*PicklistWidget*
pasaje	*StringField*	Permite mostrar/ocultar los campos correspondientes al pasaje.	*MasterSelectWidget*
tipo_pasaje	*StringField*	Tipo de pasaje a utilizar (auto, autobús, avión).	*MultiSelectionWidget*
cantidad_pasaje	*StringField*	Cantidad solicitada para pasajes.	*StringWidget*
viaticos	*StringField*	Permite mostrar/ocultar el campo correspondiente a los viáticos.	*MasterSelectWidget*
cantidad_viaticos	*StringField*	Cantidad solicitada para viáticos.	*StringWidget*
displayAttachments	*BooleanField*	Muestra/oculta los controles correspondientes a los archivos adjuntos.	*AttachmentsManagerWidget*
fecha_solicitud	*StringField*	Fecha en la cual se envía la solicitud, se obtiene de manera automática.	*StringWidget*
fecha_sesionci	*DateTimeField*	Fecha en la que el Consejo Interno revisa la solicitud.	*CalendarWidget*
actaci	*StringField*	Número de acta en la que el Consejo Interno revisa la solicitud.	*StringWidget*
cantidad_autorizada	*StringField*	Cantidad autorizada por el Consejo Interno.	*StringWidget*

Tabla 6.1: Campos comúnes a los tres tipos de solicitud.

Nombre	**Tipo**	**Descripción**	**Widget**
trabajo	*StringField*	Permite mostrar/ocultar el campo correspondiente al trabajo a presentar.	*MasterSelectWidget*
titulo_trabajo	*StringField*	Título del trabajo a presentar.	*TextAreaWidget*

inscripcion	*StringField*	Permite mostrar/ocultar el campo correspondiente a la inscripción.	*MasterSelectWidget*
cantidad_inscripcion	*StringField*	Cantidad solicitada para inscripción.	*StringWidget*

Tabla 6.2: Campos comúnes a solicitud de licencia comisión y solicitud de becario.

Nombre	**Tipo**	**Descripción**	**Widget**
grado	*StringField*	Grado del becario solicitante (Licenciatura, Maestría ó Doctorado).	*SelectionWidget*
asesor	*StringField*	Nombre del director de tesis.	*SelectionWidget*
apoyo_extra	*StringField*	Permite mostrar/ocultar el campo correspondiente a la descripción de apoyo extra.	*MasterSelectWidget*
apoyo_texto	*StringField*	Guarda información correspondiente a algún apoyo extra.	*TextAreaWidget*

Tabla 6.3: Campos específicos de una solicitud de becario.

Nombre	**Tipo**	**Descripción**	**Widget**
responsable	*StringField*	Grado del becario solicitante (Licenciatura, Maestría ó Doctorado).	*StringWidget*
invitado	*StringField*	Nombre del director de tesis.	*StringWidget*

Tabla 6.4: Campos específicos de una solicitud de visitante.

La siguiente figura muestra gráficamente los campos descritos anteriormente, correspondientes a los tres tipos de contenido definidos: Solicitud de licencia comisión, solicitud de becario y solicitud de visitante.

Para realizarla, se utilizó el programa *Freemind*, una herramienta de *software* libre para realizar mapas mentales y que es recomendada por la comunidad de Plone para describir tipos de contenido (*http://freemind.sourceforge.net/wiki/index.php/Main_Page*).

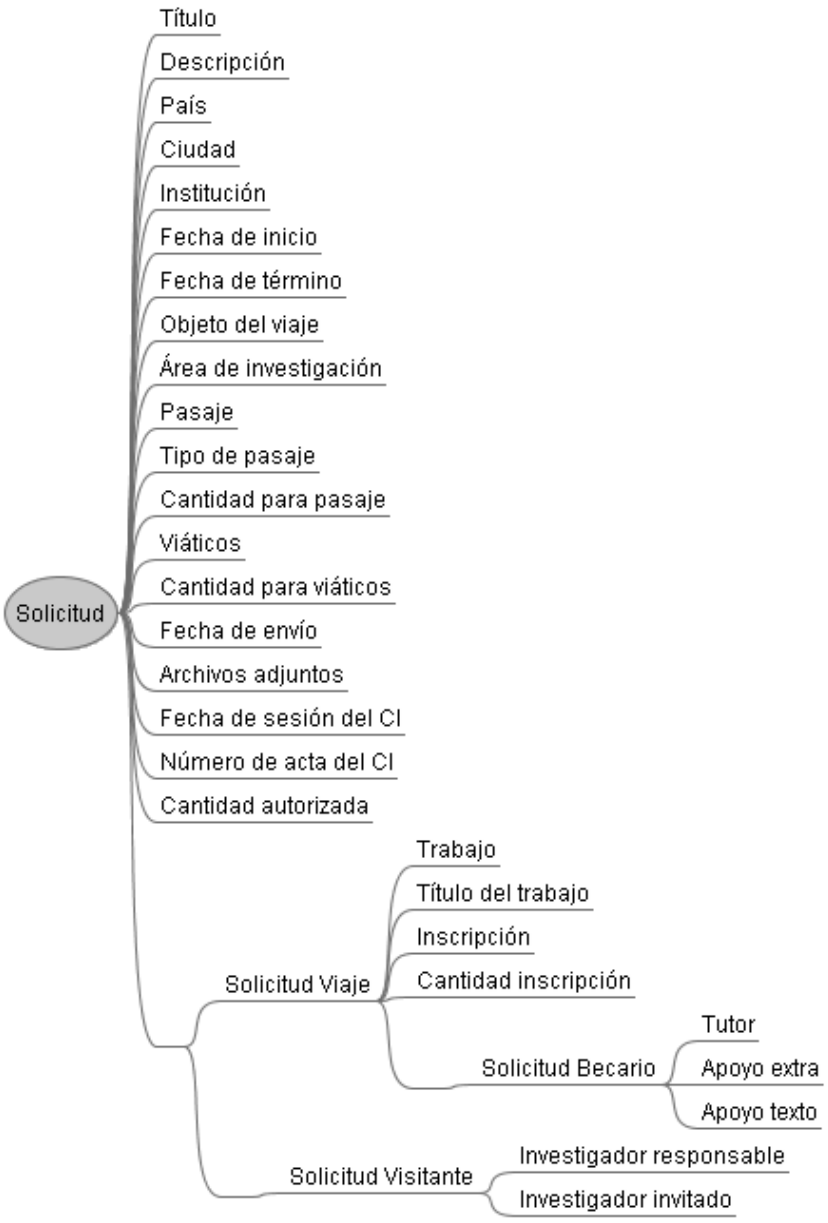

Figura 6.1: Campos de los tipos de contenido definidos.

6.2 Creación de clases para el tipo de contenido.

Una vez definido el *schema* con los campos necesarios para el contenido, se debe declarar una clase por cada tipo de contenido a utilizar, para este producto, se creó una *interface* y tres clases. El objetivo de la *interface*, es establecer un marcador para realizar las búsquedas más fácilmente; las tres clases la implementan y las búsquedas pueden realizarse haciendo referencia a la *interface*.

Dentro de cada clase se definen algunos atributos necesarios; tales como sus clases padre (Python proporciona soporte para herencia múltiple), las interfaces que implementa, el tipo que definirá dentro del sitio y el *schema* del cuál tomará la definición de los campos para ese tipo de contenido, entre otros. Además, se definen todos los métodos asociados a los objetos de ese tipo.

La figura 6.2 muestra la pantalla de captura correspondiente al *schema* definido.

Figura 6.2: Vista del *schema* dentro de Plone.

Una vez que se ha llenado la solicitud, se debe guardar, en este momento se obtiene de forma automática el título de la solicitud, que sirve también como identificador del contenido; además, se muestra un aviso al usuario, indicando que debe enviar la solicitud, utilizando los botones que están en la parte baja de la solicitud, tal como se muestra en la figura 6.3.

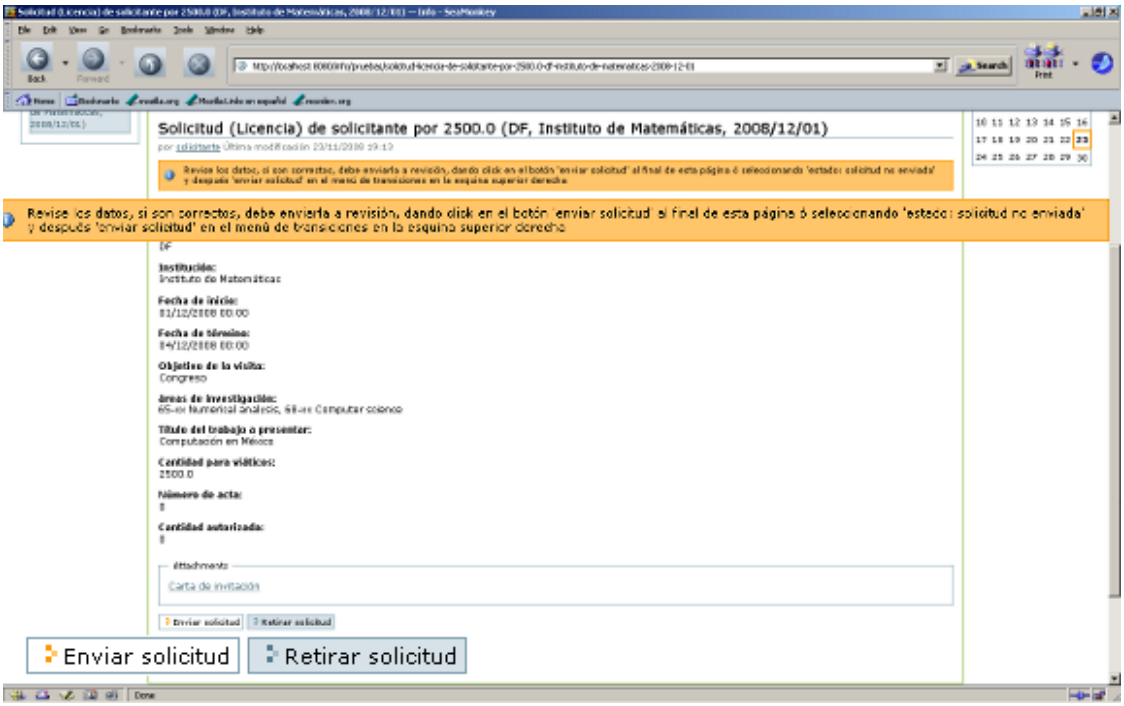

Figura 6.3: Una solicitud recientemente creada.

6.3 Diseño e integración de workflows.

En la sección 4.5, se explica la forma general de diseño de *workflows*, para el producto Solicitudes, se definen dos: *solicitud_workflow* y *solicitud_becario_workflow*, siguiendo los pasos descritos en dicha sección.

6.3.1 Definición de transiciones y estados.

Las figuras 5.1 y 5.2 muestran los estados y transiciones correspondientes a los dos *workflows* utilizados.

6.3.2 Definición de permisos.

Los permisos definidos en las secciones 5.2.2 y 5.2.3 para los estados y las transiciones de los dos *workflows*, se muestran a continuación:

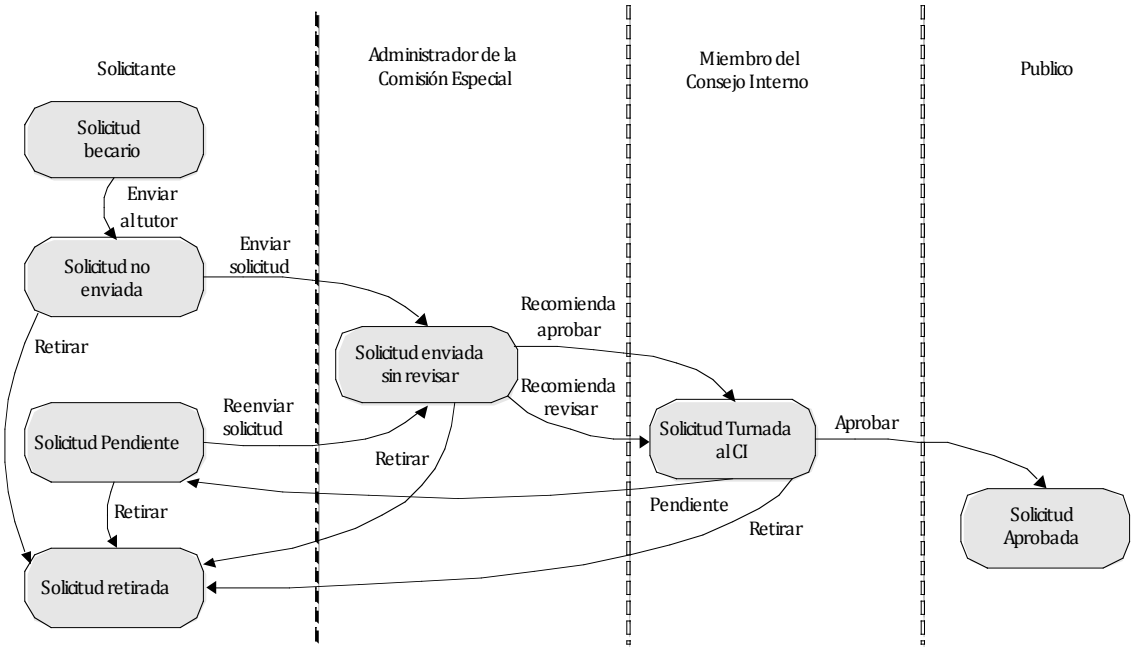

Figura 6.4: *Workflow* de solicitud de becario.

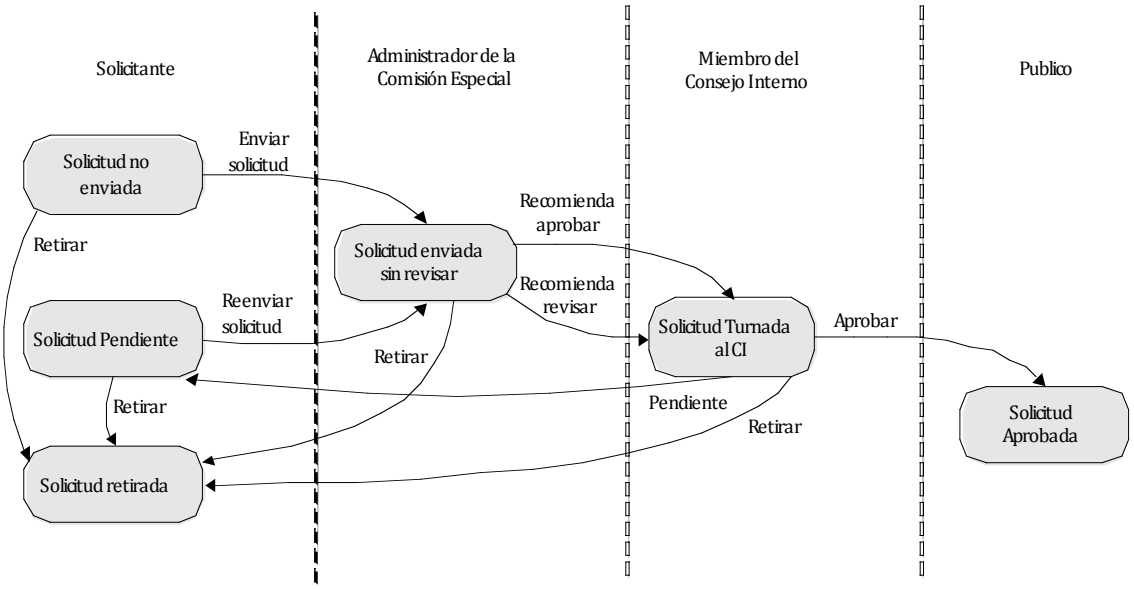

Figura 6.5: *Workflow* de solicitud.

A continuación se muestran las imágenes correspondientes a los mismos *workflows*, utilizando el producto *collective.workflowed*.

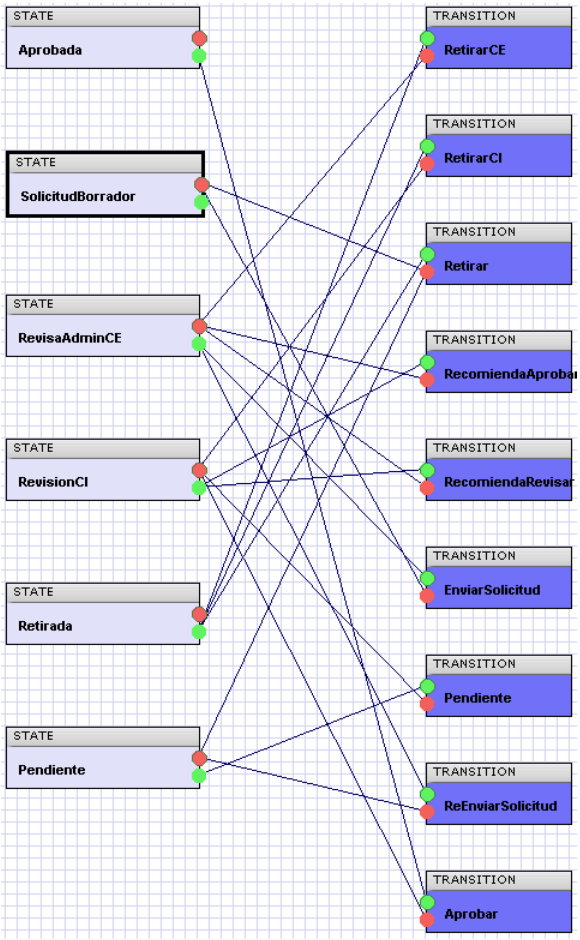

Figura 6.6: *Workflow* de solicitud, utilizando *collective.workflowed*.

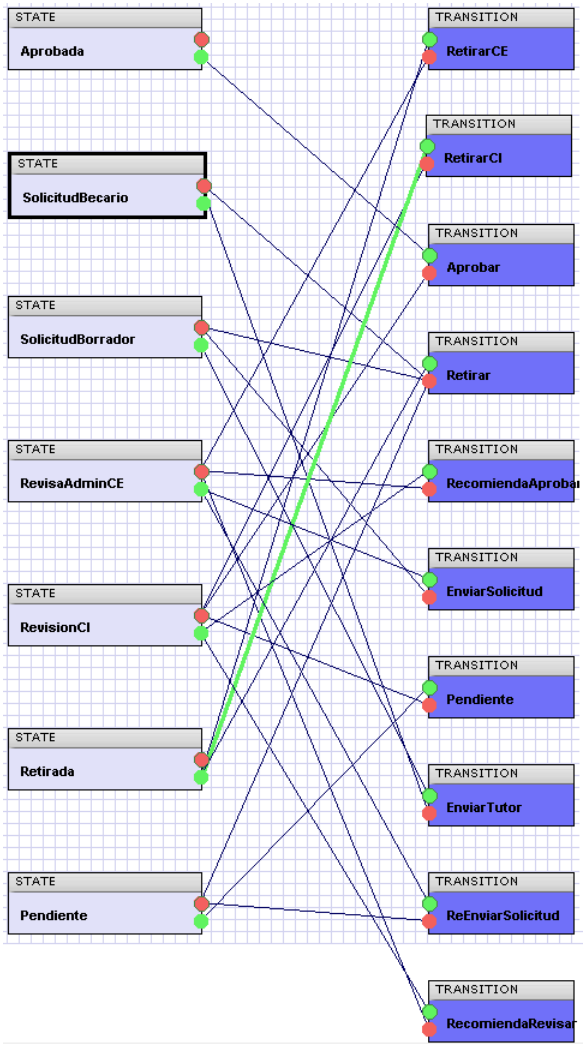

Figura 6.7: *Workflow* de solicitud de becario, utilizando *collective.workflowed*.

6.3.3 Definición de *scripts*.

Para el producto Solicitudes, se definieron los siguientes *scripts*:

- ✓ ***aprobado_script***. Se ejecuta cuando se ejecuta la transición *Aprobar* y envía un e-mail al propietario, avisando que su solicitud fue aprobada.
- ✓ ***pendiente_script***. Se ejecuta cuando se ejecuta la transición *Pendiente* y envía un e-mail al propietario, avisando que su solicitud está pendiente y que puede reenviarla para volver a evaluarla.
- ✓ ***retirarCE_script***. Se ejecuta cuando se ejecuta la transición *Retirar* desde el estado *Solicitud enviada sin revisar*, y envía un e-mail al propietario, avisando que su solicitud fue retirada por el administrador de la comisión especial.

- ✓ ***retirarCI_script***. Se ejecuta cuando se ejecuta la transición *Retirar* desde el estado *Solicitud turnada al CI*, y envía un e-mail al propietario, avisando que su solicitud fue retirada por un miembro del Consejo Interno.

- ✓ ***changeFolder_script***. Se ejecuta cuando se ejecuta la transición *Enviar Solicitud* desde el estado *Solicitud borrador*, y mueve la solicitud de la carpeta del solicitante a la carpeta de *Solicitudes*.

A continuación se muestra una ejecución del *workflow* de solicitud, explicando además algunos detalles. Se utilizan tres navegadores diferentes para presentar la interacción de los roles definidos para el *workflow*.

La solicitud es creada en una carpeta propia del solicitante; una vez guardada, se encuentra en el estado "*Solicitud no enviada*", en este el solicitante puede hacer modificaciones para posteriormente enviarla a revision por la Comisión Especial, una vez enviada se realizan varias actividades; se obtiene de forma automatic la fecha de envío de la solicitud, se muestra una aviso de acuse de recibo y la solicitud es movida a una carpeta concentradora de solicitudes, figura 6.7.

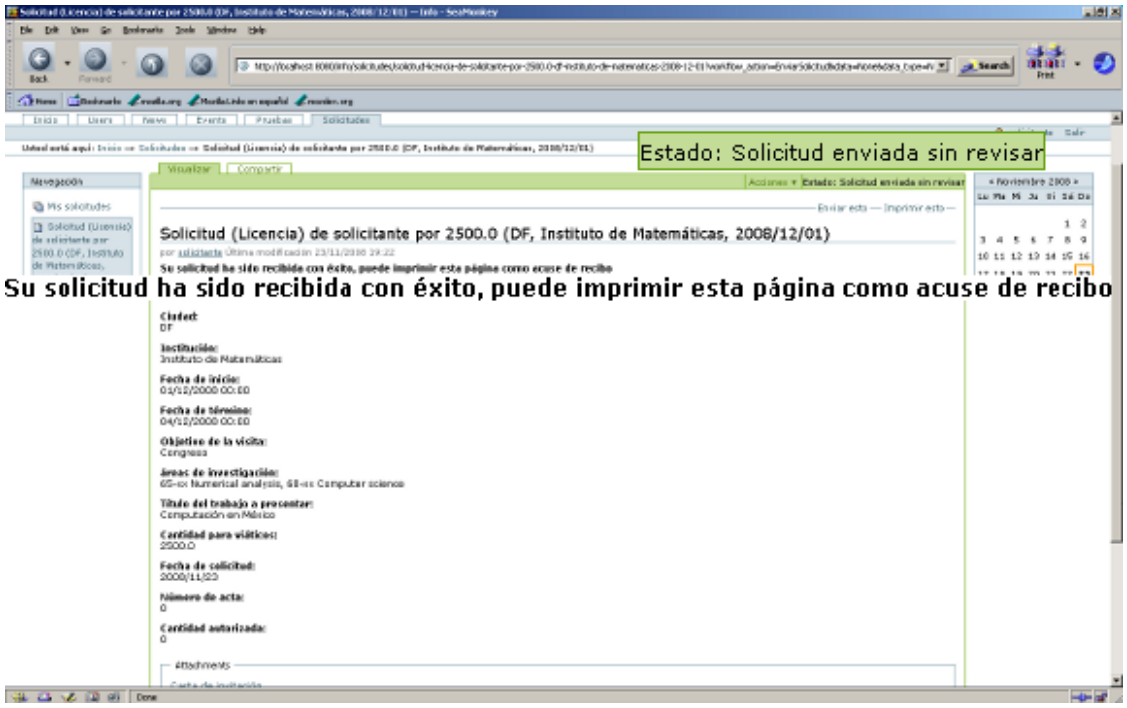

Figura 6.7: Solicitud enviada a revisión.

Una vez enviada, la solicitud llega al estado "*Solicitud enviada sin revisar*", en este es responsabilidad de la Comisión Especial realizar dicha revision y emitir una recomendación, puede ser "*Recomendar aprobación*" ó "*Recomendar revisión*"; es importante indicar que solo algún miembro de la Comisión Especial del IMATE puede realizar alguna de estás transiciones, figura 6.8.

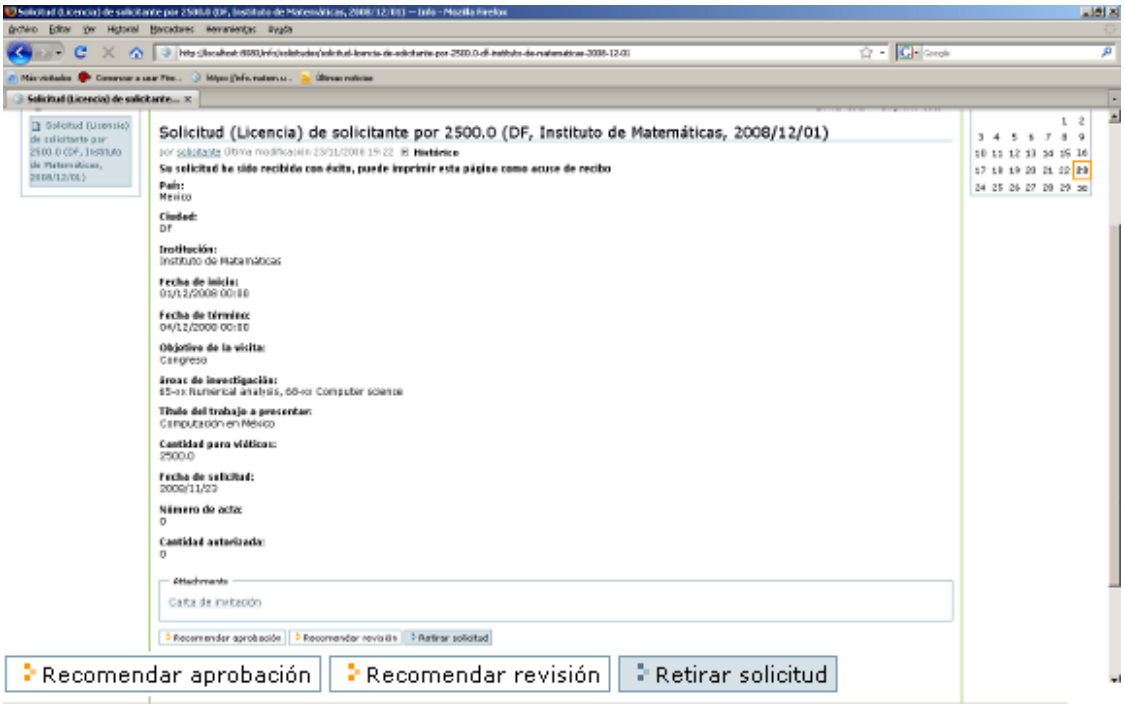

Figura 6.8: Solicitud en revisión por la Comisión Especial.

La figura 6.9 presenta la solicitud después de que un miembro de la Comisión Especial llevó a cabo la transición "*Recomendar aprobación*" y se encuentra en el estado "*Solicitud turnada al CI*".

Figura 6.9: Solicitud turnada al Consejo Interno (CI).

Ya en este estado, solamente los miembros del Consejo Interno del IMATE pueden realizar alguna transición, ya sea "*Aprobar*" ó "*Poner pendiente*", la figura 6.10 muestra una solicitud en revision por el CI.

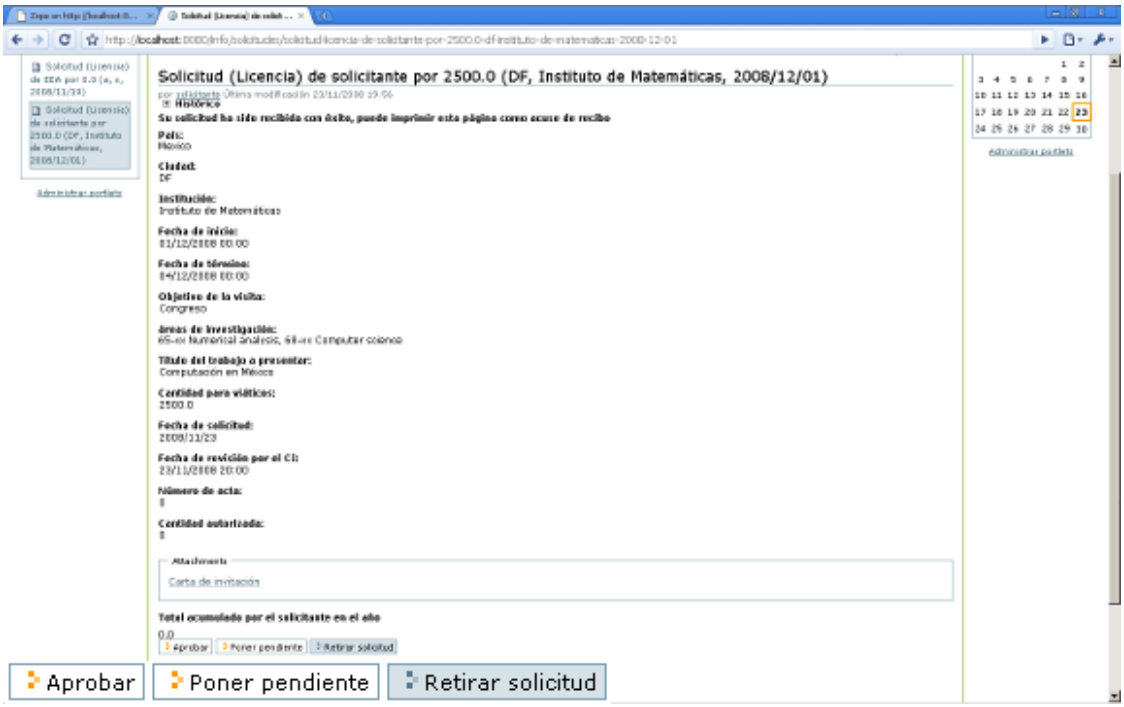

Figura 6.10: Solicitud en revisión por el CI.

Sea cuál sea la resolución tomada, se envía una correo de información al solicitante y se obtiene de forma automática la fecha de revisión, si se decidió aprobar la solicitud, se llega al estado final del *workflow*, si por el contrario, se decide dejar pendiente la solicitud, el solicitante tiene la opción de corregir lo que se le solicite y posteriormente volver a enviar la solicitud para revision por la Comisión Especial y repetir el *workflow*. La figura 6.11 expone una solicitud aprobada.

Figura 6.11: Solicitud aprobada.

En la figura siguiente se puede ver el contenido del correo enviado al realizar esta transición, cabe mencionar que es similar a los correos enviados en las transiciones que también realizan envío de avisos (pendiente y retirar).

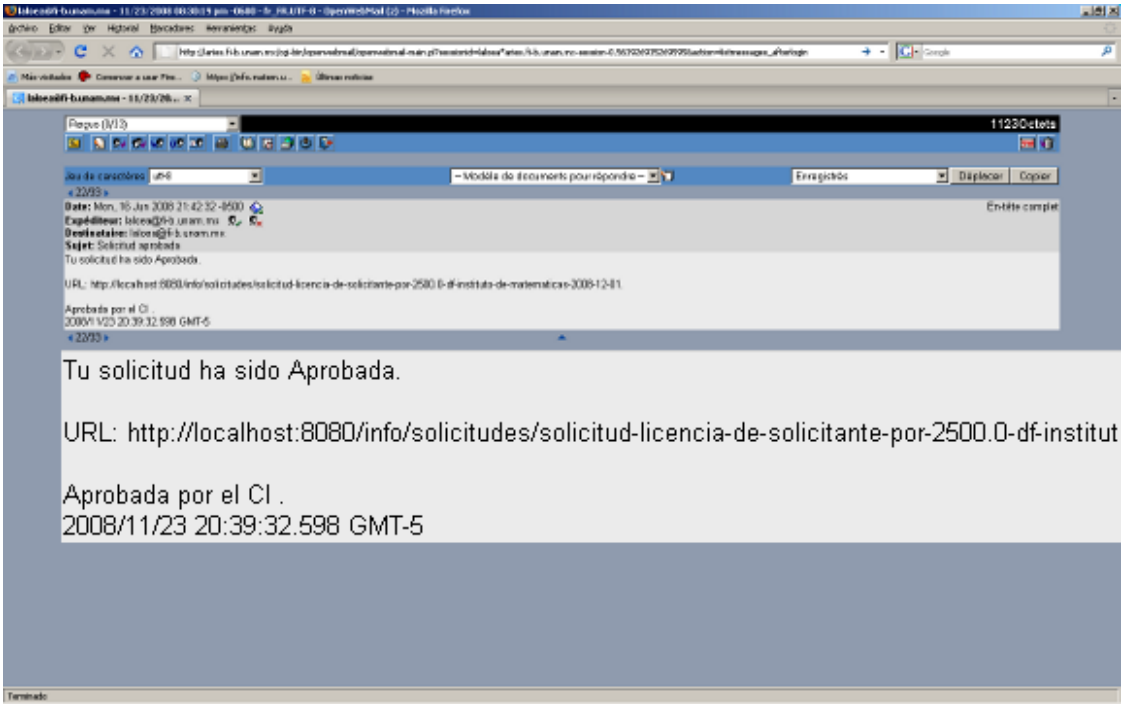

Figura 6.11: Correo enviado en la transición.

Ahora se muestra la solicitud en un navegador configurado en ingles y otro configurado en español, mostrando la facilidad para internacionalizar contenidos con Plone.

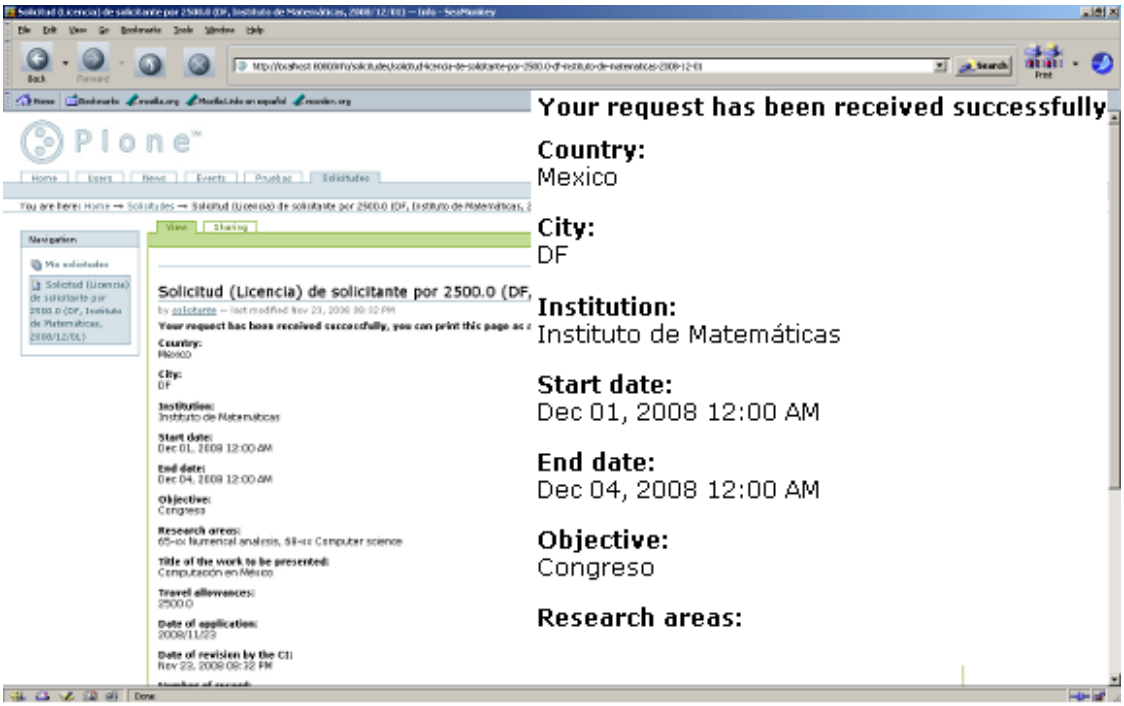

Figura 6.12: Vista del contenido en un navegador configurado para mostrar las páginas en inglés.

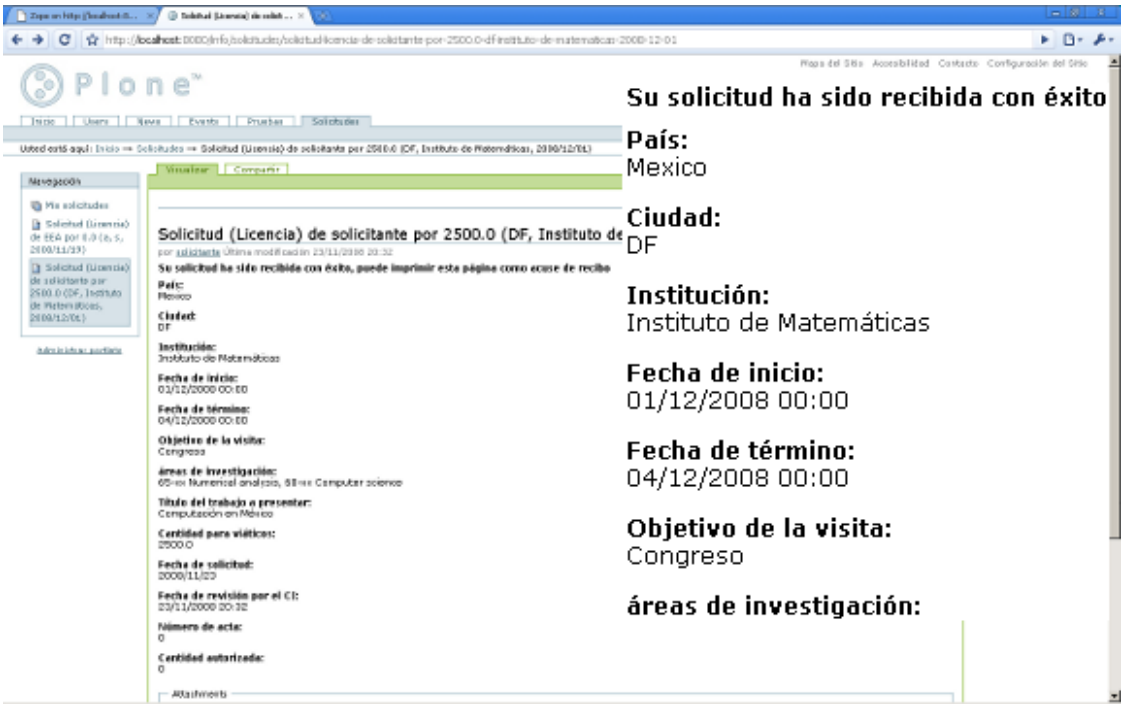

Figura 6.12: Vista del contenido en un navegador configurado para mostrar las páginas en español.

El apéndice A, explica con más detalle la forma de desarrollar productos para Plone (*schemas*, clases, *workflows*, etc.).

6.4 Pruebas.

Las pruebas unitarias no son estrictamente una característica de Plone, pero son una "*mejor práctica*" muy importante. Plone utiliza ampliamente las pruebas unitarias y serán presentadas brevemente aquí [41].

La idea de las pruebas unitarias, es que, conforme la complejidad de una pieza de *software* aumenta, se vuelve más difícil de probar. Es difícil saber si se cubren todos los casos cuando simplemente se prueba el producto en la interfaz de usuario de Plone, y como se sigue desarrollando el producto, se puede estropear algo que antes estaba trabajando.

Las reglas de oro de las pruebas unitarias, son los siguientes:

- ✓ Escribir al menos una prueba para cada funcionalidad.
- ✓ Escribir la interfaz y/o métodos estándar en primer lugar y, a continuación, escribir la prueba, la prueba debe fallar (el código aún no está escrito).
- ✓ Sólo cuando se tiene una prueba fallida, se puede implementar la funcionalidad. El objetivo de cada línea de código escrita debe ser pasar una prueba fallida.
- ✓ Cuando se encuentra un error, no debe arreglarse en lugar de ello, se escribe una prueba fallida, para demostrar que el error está ahí y después se debe arreglar.

Esto puede parecer engorroso todos y contra-intuitivo, pero las pruebas unitarias son:

1. La única manera de convencer incluso a distancia a los clientes de que el código es bueno.
2. La única manera de asegurarse (o al menos tener mayor certeza) de que no se rompen cosas sin darse cuenta de ello.
3. La única manera de asegurarse (o al menos tener mayor certeza) de que no se vuelven a introducir errores que ya se habían arreglado.
4. Por lo general, una manera de ahorrar tiempo a largo plazo, porque se sabe inmediatamente cuando algo deja de funcionar, y se gasta menos tiempo en perseguir errores en el código que se escribió hace mucho tiempo.
5. Una forma útil de escribir y probar código en el mismo entorno (no es necesario cambiar el contexto de un navegador y hacer clics para probar las características más recientes) basta con ejecutar las pruebas.

Las pruebas unitarias funcionan configurando una "*caja de pruebas*" (también llamada instalación fija de prueba), *PloneTestCase* es básicamente una instancia Zope vacía con un solo

sitio Plone, con un único miembro, que contiene una carpeta del miembro por defecto. Todas las pruebas se ejecutan en el mismo entorno (es decir, cada vez que un método de prueba haya finalizado, la transacción Zope se aborta de manera que no importa la forma en que la prueba ha cambiado el estado del portal, la próxima prueba será completamente intacta).

- ✓ Algunas reglas básicas para escribir pruebas unitarias con PloneTestCase son:
- ✓ Escribir las pruebas primero.
- ✓ Escribir una prueba (es decir, un método) para cada cosa que desea probar.
- ✓ Mantener juntas las pruebas relacionadas (es decir, en la misma clase)
- ✓ Ser pragmáticos. No es necesario probar todas las combinaciones de entradas y salidas, es poco probable que se den algunas combinaciones.
- ✓ Mantenerlas simples. No sobre-generalizar, cuando una prueba falla, es necesario determinar fácilmente si se debe a que la prueba está mal, o lo que se está probando tiene un error.
- ✓ Siempre ejecutar todas las pruebas antes de revisar el código (especialmente si no está el autor original) y asegurarse de no romper nada.

Para el producto *Solicitudes* se implementaron las pruebas unitarias con dos archivos:

El contenido del primer archivo *Solicitudes/tests/base.py* es:

```
from Testing import ZopeTestCase
from Products.PloneTestCase import PloneTestCase

ZopeTestCase.installProduct('SimpleAttachment')
ZopeTestCase.installProduct('Solicitudes')
PRODUCTS = ['SimpleAttachment', 'Solicitudes']
PloneTestCase.setupPloneSite(products=PRODUCTS)

class SolicitudesTestCase(PloneTestCase.PloneTestCase):

    class Session(dict):
        def set(self, key, value):
            self[key] = value

    def _setup(self):
        PloneTestCase.PloneTestCase._setup(self)
        self.app.REQUEST['SESSION'] = self.Session()
```

A grandes rasgos, este archivo se encarga de configurar la "*caja de pruebas*" *PloneTestCase*.

El segundo archivo *Solicitudes/tests/ testSetup.py* contiene dos clases de prueba y se encargan de probar la instalación de producto, así como la creación y edición del contenido definido:

```
from Products.Solicitudes.tests import base
```

```
class TestInstallation(base.SolicitudesTestCase):
    """Aseguar que el producto se instala apropiadamente"""

    def afterSetUp(self):
        self.css        = self.portal.portal_css
        self.kupu       = self.portal.kupu_library_tool
        self.skins      = self.portal.portal_skins
        self.types      = self.portal.portal_types
        self.factory    = self.portal.portal_factory
        self.workflow   = self.portal.portal_workflow
        self.properties = self.portal.portal_properties

        self.metaTypes =
('Solicitud','SolicitudBecario','SolicitudVisitante',)

    def testSkinLayersInstalled(self):
        self.failUnless('solicitud' in self.skins.objectIds())

    def testTypesInstalled(self):
        for t in self.metaTypes:
            self.failUnless(t in self.types.objectIds())

    def testPortalFactorySetup(self):
        self.failUnless('Solicitud' in
self.factory.getFactoryTypes())

    def testParentMetaTypesNotToQuery(self):
        parentMetaTypesNotToQuery =
self.properties.navtree_properties.getProperty('parentMetaTypesNotTo
Query')
        self.failUnless('Solicitud' in parentMetaTypesNotToQuery)

    def testKupuResources(self):
        linkable =
self.kupu.getPortalTypesForResourceType('linkable')
        self.failUnless('Solicitud' in linkable)

    def testSimpleAttachmentInstalled(self):
        self.failUnless('simpleattachment' in
self.skins.objectIds())

class TestContentCreation(base.SolicitudesTestCase):
    """Asegurar que el contenido puede ser creado y editado"""

    def afterSetUp(self):
        self.folder.invokeFactory('Solicitud', 'sol1')
        self.rd1 = getattr(self.folder, 'sol1')

    def testCreateSolicitud(self):
        self.failUnless('sol1' in self.folder.objectIds())

    def testEditSolicitud(self):
        self.sol1.setTitle('A title')
        self.sol1.setDescription('A description')
        self.sol1.setText('<p>Body text</p>')
        self.sol1.setDisplayImages(True)
```

```
        self.sol1.setDisplayAttachments(False)

        self.assertEqual(self.sol1.Title(), 'A title')
        self.assertEqual(self.sol1.Description(), 'A description')
        self.assertEqual(self.sol1.getText(), '<p>Body text</p>')
        self.assertEqual(self.sol1.getDisplayImages(), True)
        self.assertEqual(self.sol1.getDisplayAttachments(), False)

    def testCreateFileAttachment(self):
        self.sol1.invokeFactory('FileAttachment', 'f1')
        self.failUnless('f1' in self.sol1.objectIds())

    def testEditFileAttachment(self):
        self.sol1.invokeFactory('FileAttachment', 'f1')
        f1 = getattr(self.sol1, 'f1')

        f1.setTitle('File title')
        f1.setDescription('File description')
        f1.setFile('X')

        self.assertEqual(f1.Title(), 'File title')
        self.assertEqual(f1.Description(), 'File description')
        self.failIf(f1.getFile() is None)

    def testCreateFileAttachmentOutsideSolicitudes(self):
        self.assertRaises(ValueError, self.folder.invokeFactory,
'FileAttachment', 'f1')

    def testCreateImageAttachment(self):
        self.sol1.invokeFactory('ImageAttachment', 'i1')
        self.failUnless('i1' in self.sol1.objectIds())

    def testEditImageAttachment(self):
        self.sol1.invokeFactory('ImageAttachment', 'i1')
        i1 = getattr(self.sol1, 'i1')

        i1.setTitle('Image title')
        i1.setDescription('Image description')
        i1.setImage('X')

        self.assertEqual(i1.Title(), 'Image title')
        self.assertEqual(i1.Description(), 'Image description')
        self.failIf(i1.getImage() is None)

    def testCreateImageAttachmentOutsideSolicitudes(self):
        self.assertRaises(ValueError, self.folder.invokeFactory,
'ImageAttachment', 'i1')

def test_suite():
    from unittest import TestSuite, makeSuite
    suite = TestSuite()
    suite.addTest(makeSuite(TestInstallation))
    suite.addTest(makeSuite(TestContentCreation))
    return suite
```

La ejecución de las pruebas unitarias, puede realizarse de las varias formas, la más sencilla es utilizar *zopectl* desde la raíz de la instancia de Zope, con un comando como:

```
./bin/zopectl test --libdir Products/Solicitudes
```

Alternativamente, desde el directorio de pruebas (*Products/Solicitudes/tests*) y ejecutar las pruebas directamente:

```
python testSetup.py
```

Cuando la prueba termina sin errores se obtiene un reporte como el siguiente:

```
...
Ran 18 tests in 6.587s
OK
```

Si hay problemas, se puede obtener un mensaje como:

```
Ran 18 tests in 7.103s
FAILED (failures=1, errors=1)
```

Esto quiere decir que existe un error de código de python y una prueba fallida durante la ejecución. Un error de código de python indica que alguno de los códigos de prueba levantó una excepción. Una prueba fallida indica que la prueba trata de afirmar algo que resulta ser falso (*not true*).

7. Conclusiones.

Para plasmar un panorama global del trabajo, se pueden visualizar de manera puntual algunos de los beneficios obtenidos al finalizar el proyecto, así como las consideraciones de aprendizaje necesarias para el desarrollo en Plone.

7.1 Conclusiones generales.

El trabajo concluido y descrito proporciona al IMATE un mejor manejo de información, es decir, a través de este desarrollo se logra un manejo óptimo de recursos importantes tales como el tiempo y materiales de oficina frecuentemente usados en los procesos administrativos.

Cómo conclusiones generales, se pueden mencionar:

- ✓ Se comprobó que el desarrollo de sitios web dinámicos a través de CMS es una solución viable cuando se presenta el problema de un manejo grande de información que es ajena a la estructura del sitio web pero que es pertenece al contenido vital y función principal de la existencia del sistema.
- ✓ Para el desarrollo en Plone es necesario tomar en cuenta la curva de aprendizaje, una vez que se concluyó esta fase, obtener los resultados deseados en la estructura y contenido de un sitio dinámico es mucho más rápido que iniciar un proyecto desde el diseño básico.
- ✓ La programación de un sitio web resulta más eficaz al usar Plone, entre otras cosas por la gran cantidad de productos ya disponibles que son compatibles con los códigos implementados.
- ✓ La utilización de productos desarrollados por terceros resulta muy útil, tanto para usarlos directamente para resolver algunos requerimientos como ayuda para entender cómo implementar las soluciones de ciertos casos de uso muy específicos.

7.2 Conclusiones particulares.

Además, una vez liberada la nueva versión de InfoMatem funcionando sobre Plone 3, se tienen las siguientes conclusiones particulares:

- ✓ Se reduce el desperdicio de recursos, al evitar impresiones y copiados excesivos e innecesarios.
- ✓ Dado que la solicitud puede llenarse en línea, vía el sitio InfoMatem, se evita que el usuario deba acudir físicamente a entregar su solicitud.
- ✓ El usuario puede dar seguimiento a su solicitud en todo lugar donde disponga de una computadora conectada a Internet, a través del sitio InfoMatem.

- ✓ Al evitar el movimiento físico de papelería, se evade la posibilidad de perder o traspapelar la solicitud o alguno de los documentos que lo acompañan, gracias a la facilidad de adjuntarlos a la solicitud electrónica.
- ✓ Se puede mantener un historial actualizado de las solicitudes realizadas.
- ✓ Se puede realizar búsquedas de solicitudes, y llevar la cuenta del dinero gastado por cada académico de forma fácil y eficiente.
- ✓ Se reduce sustancialmente el tiempo de respuesta a las solicitudes, dado que se puede realizar todo el proceso desde el sitio InfoMatem.
- ✓ El producto obtenido puede ser reutilizable para modelar e implementar otros procesos administrativos, tanto del mismo IMATE como de otras dependencias de la UNAM.
- ✓ El producto *Solicitudes* se entrega internacionalizado de acuerdo al estándar i18n, listo para presentar los contenidos en inglés y en español.
- ✓ Se describió el estado del arte de los sistemas de *workflows* y como caso particular se tomó el motor *DCWorkflow* utilizado en Plone para la implementación del producto *Solicitudes*.

La implementación del producto sobre Plone, fue una de las características más complejas del producto, la cual requirió de un porcentaje muy importante del tiempo total del desarrollo, dada su curva de aprendizaje, sin embargo el resultado es satisfactorio, ya que se cuenta con un producto fácilmente instalable que otorga la funcionalidad requerida.

Finalmente, es importante resaltar que actualmente se está dando gran auge por utilizar Plone como CMS en varias dependencias de la UNAM, encabezadas por el Instituto de Matemáticas: Dirección General de Asuntos del Personal Académico (DGAPA), Facultad de Ciencias (FC), Instituto de Ciencias Nucleares (ICN) y el Centro de Geociencias (Juriquilla).

Asimismo, se creó el Grupo de Usuarios de Plone México en el que participan, además de personal de las dependencias descritas arriba, personas externas a la UNAM que tienen gran reconocimiento en la comunidad internacional de Plone como Carlos de la Guardia y Héctor Velarde.

7.3 Trabajo futuro.

Si bien el producto final cumple con los requerimientos solicitados, se mencionan algunos aspectos que podrían facilitar la creación de productos para administración de procesos que deseen tomar como base el presente trabajo:

- ✓ Aunque el producto puede ser reutilizable para modelar e implementar otros procesos administrativos, queda pendiente la creación de productos de temas (*skins* que modifican la presentación visual del sitio) y productos de apoyo (herramientas administrativas y funcionalidades que extiendan a Plone).

- ✓ No obstante que la herramienta *collective.workflowed* resulta de gran ayuda para manipular los *workflows* de forma gráfica dentro de la *Configuración del Sitio* (*Site Setup*), esta apareció ya que se tenía gran avance en el producto, se sugiere examinar el uso de la nueva versión de *ArchGenXML* que crea la definición de los *workflows*, a partir de modelos descritos con *ArgoUML*, dicha versión apareció ya que se estaba terminando el producto y no hubo tiempo de utilizarla para probar sus bondades.

- ✓ El producto *Solicitudes* se entrega listo para presentar los contenidos en inglés y en español, pero al estar acorde al estándar i18n, existe la posibilidad de traducirlo a otros idiomas de forma sencilla.

- ✓ Revisar con mayor detalle el *framework* para pruebas unitarias *PloneTestCase*, a fin de definir correctamente las pruebas completas para todos los casos de uso.

- ✓ Finalmente, se propone revisar el producto *zope.testrecorder*, una herramienta que graba las acciones realizadas (*clics*) sobre la interfaz de usuario (navegador) y las devuelve como una prueba ejecutable.

Apéndices.

A) Estrategia para desarrollo de productos de contenido.

Un sitio basado en Plone es muy complejo y se compone de una colección de elementos como contenido, configuración y recursos de presentación. La tendencia en Plone 3 es separar lo más posible todas estas áreas, para permitir un desarrollo organizado y estructurado. La base de datos de Zope, ZODB, debe en lo posible almacenar únicamente el contenido generado por los usuarios. Todo el código y configuración del sitio deben estar en el sistema de archivos, de manera que puedan editarse y versionarse con las herramientas comunes de desarrollo y no queden encerrados en la ZODB. Esto también permite una distribución e instalación más sencillas [42].

a. Estructura de directorio del producto.

Cada producto de contenido se define como un paquete, todo paquete sigue convenciones de código ampliamente aceptadas en la comunidad de Plone. Dentro del directorio correspondiente al producto, se encuentran los siguientes archivos y directorios [16]:

__init__.py	Registra una fábrica de mensajes (*message_factory*) para internacionalización de cadenas e invoca la maquinaria de *Archetypes* para registrar tipos de contenido en Zope.
browser/	Contiene una vista de Zope 3 para cada tipo de contenido, consiste de una vista de la clase, y un *template*. Cada tipo de contenido tiene también un icono; otros recursos, como hojas de estilo (css) van también en este directorio.
config.py	Contiene constantes globales, incluyendo el nombre del producto y los nombres de los permisos para la creación del contenido.
configure.zcml	Realiza el registro de componentes. Los directorios *browser*, *content* y *portlets* tienen su propio archivo *configure.zcml*.
content/	Contiene las definiciones de los tipos de contenido.
i18n/	Contiene los archivos de traducción para el producto.
interfaces.py	Contiene las interfaces que describen los tipos de contenido y otros componentes. Estas interfaces son implementadas por las clases en el subdirectorio *content*.
portlets/	Contiene la definición y registro de *portlets*.
profiles/	Contiene el perfil que utiliza *GenericSetup* para instalar el producto.
README.txt	Una breve descripción de producto.
tests/	Contiene pruebas automáticas del producto.
versión.txt	Plone lee este archivo para saber la versión del producto.

Tabla A.1: Estructura de directorio del producto.

b. Definición de interfaces y clases.

Usualmente, las interfaces son el primer paso para un diseño detallado y sirven como una documentación formal de las capacidades de un objeto. Por esto, se defina una *interface* para cada tipo de contenido, todas ellas se encuentran en el archivo *interfaces.py*. Por ejemplo:

```
from zope import schema
from zope.interface import Interface
from zope.app.container.constraints import contains
from zope.app.container.constraints import containers
from Products.Persona import PersonaMessageFactory as _
class IPersona(Interface):
    """Interface marcadora de Persona"""
```

Una buena práctica para hacer el producto traducible, es utilizar una fábrica de mensajes, definida en el archivo *__init__.py* de la siguiente forma:

```
from zope.i18nmessageid import MessageFactory
PersonaMessageFactory = MessageFactory('Persona')
```

Al importar esta fábrica con el nombre especial '_' (guión bajo), las herramientas de internacionalización pueden extraer las cadenas.

Las clases toman la definición del *schema* para los atributos correspondientes a los contenidos que define, pero además sirve para declarar el tipo dentro de Plone, así como otras características particulares, un ejemplo de definición de clase es el siguiente.

```
from zope.interface import implements
from zope.component import adapts,adapter

from Products.Archetypes import atapi
from Products.validation import V_REQUIRED
from Products.CMFCore.utils import getToolByName

from Products.ATContentTypes.content import folder
from Products.ATContentTypes.content import schemata
from Products.ATContentTypes.content.schemata import
finalizeATCTSchema

from Products.Persona.interfaces import IPersona
from Products.Persona.config import PROJECTNAME
from Products.Persona import PersonaMessageFactory as _
...
class Persona(folder.ATFolder):
    """Una persona"""
    implements(IPersona)
    portal_type = "Persona"
    _at_rename_after_creation = True
    schema = PersonaSchema
    title = atapi.ATFieldProperty('title')
    descripcion = atapi.ATFieldProperty('description')
    def listTitles(self):
        """lista"""
        return ('Sr.','Sra.','Srita.',)

atapi.registerType(Persona, PROJECTNAME)
```

Con esto, creamos una clase para encapsular el tipo de contenido, declara que implementa *IPersona*, y especifica un nombre de tipo; el tipo del contenido debe ser único en el sitio. Al hacer *_at_rename_after_creation = True*, causará que *Archetypes* renombre el objeto con una versión normalizada del título del contenido cuando es guardado por primera vez (que es parte del estándar de *Dublin core metadata* [43] y prácticamente requerido para todos los tipos de contenido de Plone); esto hace que los URLs de Plone sean legibles y tengan sentido, a menos que sea necesario maneje explícitamente los IDs.

c. Definición de campos, *widgets, schemata* y vocabularios.

Como se dijo en la sección 6.1, es en el *schema* donde se definen los campos que componen a cada objeto (tipo de contenido) dentro del sitio, así como los tipos y los *widgets* de los mismos, a continuación se muestra el *schema* de definición de un tipo de contenido sencillo.

```
PersonaSchema = folder.ATFolderSchema.copy() + atapi.Schema((

    atapi.StringField(
        name='titulo',
        widget=atapi.SelectionWidget(
            label=u"Title",
        ),
        required=True,
        searchable=True,
        vocabulary='listTitles',
    ),
    atapi.StringField(
        name='nombre',
        widget=atapi.StringWidget(
            label=u"Name",
        ),
        required=True,
        searchable=True
    ),
    atapi.StringField(
        name='apellido_paterno',
        widget=atapi.StringWidget(
            label=u"Last name",
        ),
        required=True,
        searchable=True
    ),
    atapi.StringField(
        name='apellido_materno',
        widget=atapi.StringWidget(
            label=u"Mothers name",
        ),
        required=True,
        searchable=True
    ),
    atapi.DateTimeField(
        name='fecha_nacimiento',
        storage=atapi.AnnotationStorage(),
        widget=atapi.CalendarWidget(label=_(u"Birthday"),
```

```
                                        description=_(u""),
                                        show_hm=False),
            required=True,
            searchable=False,
            ),
        atapi.StringField(
            name='telefono',
            widget=atapi.StringWidget(
                label=u'Phone number',
            ),
            schemata="contacto",
            searchable=True,
        ),
        atapi.StringField(
            name='email',
            widget=atapi.StringWidget(
                label=u'e-mail',
            ),
            schemata="contacto",
            searchable=True,
            validators=('isEmail',)
        ),
        ))

    PersonaSchema['title'].storage = atapi.AnnotationStorage()
    PersonaSchema['title'].widget.label = _(u"Title")
    PersonaSchema['title'].widget.description = _(u"")

    PersonaSchema['description'].storage = atapi.AnnotationStorage()
    PersonaSchema['description'].widget.label = _("Description")
    PersonaSchema['description'].widget.description = _(u"")

    finalizeATCTSchema(PersonaSchema, folderish=True,
    moveDiscussion=False)
```

Primero, copiamos el *schema* del tipo base, en este caso, *ATFolderSchema* de *ATContentTypes*. Después, agregamos nuestro propio *schema*.

Los campos básicos se encuentran en *Products.Archetypes.Field*; se inicializan con cierto número de propiedades y algunos aceptan configuraciones específicas. Los *widgets* básicos, están definidos en *Products.Archetypes.Widget*; cada campo tiene un *widget* por omisión, al igual que los campos, los *widgets* tienen ciertas propiedades. Para mayor referencia, consultar el *Archetypes Reference Manual* (*http://plone.org/products/archetypes*).

Después de definir el *schema*, llamamos a *finalizeATCTSchema()*, este método reordena algunos campos (si es necesario) y asigna los campos a los *schematas* (pestañas) de acuerdo a las convenciones de Plone. Con el esquema finalizado, lo asignamos a la variable *schema* de la clase que define al objeto.

```
class Persona(folder.ATFolder):
...
    schema = PersonaSchema
...
```

De esta forma, *Archetypes* puede generar formularios de edición y vistas a partir de estos campos y *widgets*.

Se puede especificar un ***vocabulario*** para un campo, utilizando la propiedad *vocabulary*. Usualmente, los vocabularios se utilizan con *SelectionWidget*, *MultiSelectionWidget*, o *InAndOutWidget*. El vocabulario más simple es una lista de valores válidos.

Para vocabularios dinámicos, se puede asignar el vocabulario a una cadena que contiene el nombre de un método del objeto. *Archetypes* llama a este método para obtener los valores del vocabulario (una tupla o lista de valores). Por ejemplo:

```
atapi.StringField(
    name='titulo',
    widget=atapi.SelectionWidget(
        label=u"Title",
    ),
    required=True,
    searchable=True,
    vocabulary='listTitles',
),
```

Y luego en la clase se define lo que hará el método:

```
def listTitles(self):
    """lista"""
    return ('Sr.','Sra.','Srita.',)
```

d. Vistas, viewlets y recursos especiales para un tipo de contenido.

Una vez creado el contenido el tipo de contenido y sus *schematas* correspondientes, pasamos a la interfaz de usuario. El código siguiente se encuentra en el directorio *browser*. Usaremos las vistas y recursos definidos en Zope 3; otra opción es registrar una *skin* personalizado.

Definimos un **icono** para cada tipo de contenido en *browser/configure.zcml* haciendo referencia a archivos de imagen dentro de mismo directorio, por ejemplo:

```
<browser:resource
    name="persona_icon.png"
    image="persona_icon.png"
    />
```

Ahora podemos hacer referencia a este icono como *++resource++persona_icon.png*; Por ejemplo, lo utilizamos en el *GenericSetup* al registrar el tipo de contenido.

Las **vistas** (*views*) para el tipo de contenido, se registran de igual forma en *browser/configure.zcml*, con declaraciones como:

```
<browser:page
    for="..interfaces.IPersona"
    name="view"
    class=".persona.Persona"
    permission="zope2.View"
    />
```

Por convención, la vista por omisión se llama @@view.

También se permite al autor elegir entre varias vistas para algunos tipos de contenido, utilizando el menú *display* de Plone; la lista de vistas disponibles se especifica también usando *GenericSetup*, de cualquier forma, se debe proveer al usuario un título y una descripción amigable de la nueva vista registrada:

```
<browser:menuItem
    for="..interfaces.IPersona"
    menu="plone_displayviews"
    title="Persona view"
    action="@@view"
    description="Vista default de persona"
    />
```

El atributo *action*, hace referencia al nombre de la vista y nos aseguramos que el elemento del menú está registrado sólo para el tipo de contenido deseado. Dado el orden de procesamiento, es necesario agregar esta línea al principio del archivo:

```
<include package="plone.app.contentmenu" />
```

La clase de vista contiene métodos relacionados con los campos que se desean mostrar en la vista, por ejemplo:

```
from Acquisition import aq_inner
from AccessControl import getSecurityManager
from Products.Five.browser import BrowserView
from Products.Five.browser.pagetemplatefile import
ViewPageTemplateFile
from Products.CMFCore.utils import getToolByName
from Products.Persona.interfaces import IPersona
from plone.memoize.instance import memoize

class Persona(BrowserView):
    """Persona
    """
    __call__ = ViewPageTemplateFile('persona.pt')
    @memoize
    def nombre_completo(self):
        context = aq_inner(self.context)
        return '%s %s %s %s' % (context.titulo, context.nombre,
context.apellido_paterno, context.apellido_materno)
```

__call__, refiere específicamente a la plantilla (*page template*) correspondiente, dentro de una ruta relativa y que contiene el código de la plantilla.

El uso de *@memoize*, asegura que no importa cuántas veces se llame al método *nombre_completo()* en esta instancia, sólo se ejecutará una vez, el valor de retorno se almacena en caché. Esto se utiliza para mejorar el rendimiento cuando una plantilla necesita llamar múltiples veces al mismo método.

La plantilla correspondiente es *persona.pt*:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
```

```
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      lang="en"
      metal:use-macro="context/main_template/macros/master"
      i18n:domain="persona">
<body>

<metal:main fill-slot="main">
  <tal:main-macro metal:define-macro="main">

    <div tal:replace="structure provider:plone.abovecontenttitle" />

    <h1 class="documentFirstHeading">
      <span metal:use-macro="python:context.widget('title',
       mode='view')" />
    </h1>

    <div tal:replace="structure provider:plone.belowcontenttitle" />

    <div tal:replace="structure provider:plone.abovecontentbody" />

        <div tal:replace="view/nombre_completo" />

    <div tal:replace="structure provider:plone.belowcontentbody" />

  </tal:main-macro>
</metal:main>

</body>
</html>
```

Notamos varios manejadores de *viewlets*, incluidos con declaraciones del tipo:

```
    <div tal:replace="structure provider:plone.abovecontenttitle" />
```

Plone utiliza esto para insertar elementos comunes, tales como acciones o mensajes, y se sugiere incluirlos en las vistas, para conservar la interfaz.

Con esta platilla, obtendremos **edición en línea** del título del contenido, esto es, un usuario con permiso de edición, puede dar clic en el título y podrá editar su valor, sin necesidad de ir a la pestaña de edición (*Edit*). Para habilitar la edición en línea, sólo se necesita usar el modo *view* del *widget* del *Archetype*, por ejemplo:

```
<span metal:use-macro="python:context.widget('title',
 mode='view')" />
```

Si no se desea tener edición en línea, se puede utilizar una sintaxis más simple, accediendo directamente al campo:

```
<span tal:content="context/title" />
```

e. Formularios de creación y edición de contenidos.

Utilizando *Archetypes*, el formulario estándar de edición (*edit form*) se llama *base_edit*; si se utilizan clases basadas en *ATContentTypes*, se utiliza *atct_edit*, pero ambos esencialmente son lo mismo: iteran sobre los campos en el contexto del tipo de contenido y los muestra utilizando su diseño estándar.

El formulario de edición, actúa también como formulario de creación (*add form*) cuando se crean objetos nuevos en el sitio; con *Archetypes*, al igual que con tipos basados en CMF, los objetos se crean primero en la ZOBD antes de ser editadas, esto se debe a que varios vocabularios y validadores dependen de tener un objeto para poder funcionar correctamente.

f. Registro e instalación de tipos.

Una vez creados los tipos de contenido y sus vistas, se debe crear el código de instalación; en Plone 3 hacemos esto utilizando el perfil de extensión (*profile extension*) de *GenericSetup*, en el archivo *configure.zcml* del directorio principal del producto, tenemos:

```
<genericsetup:registerProfile
  name="default"
  title="Persona"
  directory="profiles/default"
  description=""
  provides="Products.GenericSetup.interfaces.EXTENSION"
  />
```

En el archivo *profiles/default/types.xml*, se registran los tipos de contenido:

```
<?xml version="1.0"?>
  <object name="portal_types" meta_type="Plone Types Tool">
    <object name="Persona"
      meta_type="Factory-based Type Information with dynamic views"/>
</object>
```

Esto hará que varios objetos de tipo de información basados en fábrica **FTIs** (*factory-based type information objects*) sean creados en la herramienta *portal_types* (tipos en el portal). Cada FTI se configura más detalladamente en su correspondiente archivo dentro de *profiles/default/types/*. El nombre del archivo debe ser el mismo del nombre del tipo de contenido dentro del portal, por ejemplo, tenemos la configuración en el archivo *profiles/default/types/Persona.xml*.

```
<?xml version="1.0"?>
<object name="Persona"
   meta_type="Factory-based Type Information with dynamic views"
   i18n:domain="formulario"
   xmlns:i18n="http://xml.zope.org/namespaces/i18n">
 <property name="title" i18n:translate="">Persona</property>
 <property name="description"
     i18n:translate="">Persona</property>
 <property name="content_icon">++resource++persona_icon.png
 </property>
```

Estas líneas, dan al tipo de contenido un nombre, descripción y un icono que será mostrado en la interfaz de usuario de Plone.

```
<property name="content_meta_type">Persona</property>
<property name="product">Persona</property>
<property name="factory">addPersona</property>
<property name="immediate_view">view</property>
```

Aquí, establecemos el *meta-tipo*, que usualmente es el mismo que el nombre del tipo; establecemos también una fábrica que se encargará de crear e inicializar objetos nuevos de este tipo, la fábrica es generada y registrada por *Archetypes*. La propiedad *immediate_view*, supuestamente define la vista que se mostrará inmediatamente después de que el objeto es creado, pero aún no se usa en Plone al momento de escribir el presente texto.

```
<property name="global_allow">True</property>
<property name="filter_content_types">False</property>
<property name="allow_discussion">False</property>
```

Estas propiedades controlan las relaciones entre contenedores y sus hijos, folders de tipos genéricos establecen *filter_content_types* a *False*, de esta forma, todos los tipos que tienen *global_allow* establecido a *True*, serán permitidos en este tipo de contenedor. La propiedad *allow_discussion*, determina si, por omisión, está permitido hacer o no comentarios sobre el contenido.

```
<property name="default_view">view</property>
 <property name="view_methods">
  <element value="view"/>
 </property>
```

Estas propiedades se relacionan con el menú de vistas de Plone; especifican que vista es usada por omisión, y la lista completa de vistas disponibles para objetos de este tipo.

```
<alias from="(Default)" to="(dynamic view)"/>
<alias from="edit" to="atct_edit"/>
<alias from="sharing" to="@@sharing"/>
<alias from="view" to="(selected layout)"/>
```

Aquí se especifican *aliases* para nuestro tipo de contenido, por convención, casi todos los tipos de contenido de Plone utilizan estos cuatro aliases.

```
 <action title="View" action_id="view" category="object"
condition_expr=""
    url_expr="string:${object_url}" visible="True">
  <permission value="View"/>
 </action>
 <action title="Edit" action_id="edit" category="object"
condition_expr=""
    url_expr="string:${object_url}/edit" visible="True">
  <permission value="Modify portal content"/>
 </action>
</object>
```

Finalmente, registramos varias acciones específicas del tipo, que por estar en categoría de objeto, se muestran como pestañas en elementos de contenido; nótese como se hace referencia a los aliases definidos anteriormente.

g. Fábricas y permisos de creación.

Cuando se configura el FTI, se hace referencia a una fábrica llamada *addPersona*, la fábrica es generada por *Archetypes*, cuando el producto se inicializa en el archivo *__init__.py* del directorio principal del producto, que también registra los permisos de creación para cada tipo de contenido.

```
from zope.i18nmessageid import MessageFactory
from Products.Archetypes import atapi
from Products.CMFCore import utils
from Products.CMFCore.permissions import setDefaultRoles
from Products.Persona import config

PersonaMessageFactory = MessageFactory('Persona')

def initialize(context):
    from content import persona

    content_types, constructors, ftis = atapi.process_types(
        atapi.listTypes(config.PROJECTNAME),
        config.PROJECTNAME)

    for atype, constructor in zip(content_types, constructors):
        utils.ContentInit('%s: %s' % (config.PROJECTNAME,
            atype.portal_type),
            content_types  = (atype,),
            permission  = config.ADD_PERMISSIONS[atype.portal_type],
            extra_constructors = (constructor,),
            ).initialize(context)
```

Esto hace referencia a dos variables dentro de *config.py*:

```
PROJECTNAME = 'persona'

ADD_PERMISSIONS = {
    "Persona" : "Add Persona",
}
```

Las primeras líneas de *initialize()*, importan algunos módulos necesarios; las siguientes líneas se utilizan para crear la fábrica.

Finalmente se inicializa el tipo de contenido con sus respectivos permisos, permite que la creación de cada tipo de contenido sea controlada por un permiso diferente. También se deben establecer los roles para cada nuevo permiso; esto se define en el archivo *profiles/default/rolemap.xml*:

```
<?xml version="1.0"?>
<rolemap>
  <permissions>
    <permission name="Add Persona" acquire="False">
        <role name="Manager" />
    </permission>
  </permissions>
</rolemap>
```

h. Registrar tipos de contenido con la Factory Tool.

Todos los nuevos tipos de contenido deben usar la herramienta *portal_factory* para mitigar los posibles problemas causados por la creación prematura de objetos. La configuración correspondiente se encuentra en *profiles/default/factorytool.xml*:

```
<?xml version="1.0"?>
<object name="portal_factory" meta_type="Plone Factory Tool">
  <factorytypes>
    <type portal_type="Persona"/>
  </factorytypes>
</object>
```

i. Agregar índices de catálogo columnas de metadatos.

Los índices de catálogo (*catalog index*) y columnas de metadatos (*metadata columns*) sirven para optimizar búsquedas basadas en los valores de esos campos, se configuran en *profiles/default/catalog.xml*:

```
<?xml version="1.0"?>
<object name="portal_catalog" meta_type="Plone Catalog Tool">
    <index name="nombre" meta_type="FieldIndex">
        <indexed_attr value="nombre"/>
    </index>
    <index name="apellido_paterno" meta_type="FieldIndex">
        <indexed_attr value="apellido_paterno"/>
    </index>
    <column value="nombre"/>
    <column value="apellido_paterno"/>
</object>
```

j. Portlets propios del contenido.

Los *portlets* representan una forma útil para mostrar contenidos de interés a los usuarios del sitio. Su definición se encuentra en el subdirectorio *portlets/*, por ejemplo, el ejemplo, tendremos un *portlet* llamado directorio, para el cual tenemos una plantilla llamada *directorio.pt* y un módulo de Python, llamado *directorio.py*.

Veamos primero la plantilla:

```
<dl class="portlet portletDirectory"
    i18n:domain="persona">
    <dt class="portletHeader">
        <span class="portletTopLeft"></span>
        <span tal:content="view/header">Personas</span>
        <span class="portletTopRight"></span>
    </dt>
    <tal:items tal:repeat="persona view/personas">
        <dd class="portletItem"
            tal:define="oddrow repeat/persona/odd;"
            tal:attributes="class python:oddrow and
                            'portletItem even' or 'portletItem odd'">
```

```
            <a href=""
               tal:attributes="href persona/url;
                               title persona/persona;">
                <span class="portletLink"
                      tal:content="persona/persona">
                    Title
                </span>
            </a>
        </dd>
    </tal:items>
    <dd class="portletFooter">
        <span class="portletBottomLeft"></span>
        <span class="portletBottomRight"></span>
    </dd>
</dl>
```

Aquí se emplean los estilos utilizados por los *portlets* estándar de Plone; es buena práctica que toda la lógica que determina que personas mostrar en el *portlet* sea delegada a una vista o a un *renderer* de *portlets*, que se encuentra en *directorio.py*; ahora veamos el código de este archivo; de nueva cuenta, sigue las convenciones y la estructura estándar para *portlets* de Plone; comienza con algunas importaciones necesarias:

```
from zope import schema
from zope.component import getMultiAdapter
from zope.formlib import form
from zope.interface import implements

from plone.app.portlets.portlets import base
from plone.memoize.instance import memoize
from plone.portlets.interfaces import IPortletDataProvider

from DateTime import DateTime
from Acquisition import aq_inner,aq_parent
from Products.Five.browser.pagetemplatefile import
             ViewPageTemplateFile

from Products.CMFCore.utils import getToolByName

from Products.Persona import PersonaMessageFactory as _
```

Nótese el módulo *base* importado de *plone.app.portlets*; este contiene varias clases base que facilitan la creación de *portlets*.

```
class IDirectorioPortlet(IPortletDataProvider):
    """directorio portlet"""

    header = schema.TextLine(title=_(u"Header"),
                description=_(u"Portlet header"),
                required=True)
    searchpath = schema.TextLine(title=_(u"Search Path"),
                description=_(u"Search inside this path"),
                default=u"/")
    maxitems = schema.Int(title=_(u"Max results"),
                description=_(u"Number of results to display"),
                default=0)
```

Esta clase define los aspectos configurables de un tipo de *portlet*;, usamos los campos de *zope.schema* para especificar varios atributos, además de crear formularos de creación y edición para el *portlet*. Esta *interface* es implementada por la siguiente clase:

```
class Assignment(base.Assignment):
    implements(IDirectorioPortlet)

    def __init__(self,header='Personas',searchpath='/',maxitems=0):
        self.header=header
        self.searchpath=searchpath
        self.maxitems=maxitems

    @property
    def title(self):
        return _(u"Directorio")
```

El tipo *portlet assignment*, es un objeto persistente que administra la configuración de una instancia de este *portlet*. La propiedad *title* se encuentra definida en *plone.portlets.interfaces.IPortletAssignment*, declarada en *base.Assignment*; esta propiedad será mostrada en la interfaz de administración de *portlets* de Plone.

La mayor parte de la lógica se encuentra en el *portlet renderer*; esta clase es muy similar a una vista, salvo que sólo muestra una parte de la página:

```
class Renderer(base.Renderer):

    render=ViewPageTemplateFile('directorio.pt')

    @property
    def header(self):
        return self.data.header

    @property
    def available(self):
        return len(self._data())>0

    def personas(self):
        context=aq_inner(self.context)
        for persona in self._data():
            yield dict(persona="%s %s %s" %
                               (persona.apellido_paterno,
                                persona.apellido_materno,
                                persona.nombre),
                       url=persona.getURL())

    @memoize
    def _data(self):
        context=aq_inner(self.context)
        portal_catalog= getToolByName(context, 'portal_catalog')
        results=portal_catalog(portal_type='Persona',
               path=self.data.searchpath,sort_on='apellido_paterno')
        maxitems=self.data.maxitems
        if maxitems==0:
            maxitems=len(results)
        return results[:maxitems]
```

Los *portlet renderers* son un tipo especial de proveedor de contenido; por tanto, tienen métodos *update()* y *render()*; el método *update()* vacío, se define en la clase base, y *render* es asignado a la plantilla mostrada anteriormente; también se define la propiedad *available*, si tiene valor *False*, el *portlet* no se mostrará; si ningún *portlet* de una columna se encuentra disponible, la columna entera se ocultará. El resto de la clase es lógica específicaal *portlet* directorio; busca *personas* en el sitio y devuelve una lista simplificada.

Finalmente, debemos declara los formularios de creación y edición del tipo de *portlet*, que permitirá al usuario añadir y modificar configuraciones del *portlet* directorio.

```
class AddForm(base.AddForm):
    form_fields=form.Fields(IDirectorioPortlet)
    label=_(u"Agregar directorio de personas")
    description=_(u"Este portlet muestra las personas del sitio")

    def create(self,data):
        assignment=Assignment()
        form.applyChanges(assignment, self.form_fields, data)
        return assignment

class EditForm(base.EditForm):
    form_fields=form.Fields(IDirectorioPortlet)
    label=_(u"Editar directorio de personas")
    description=_(u"Este portlet muestra las personas del sitio")
```

Estas clases *widgets* de la *interface* definida anteriormente, además la clase *AddForm*, implementa el método *create()* que es requerido para construir nuevas instancias.

Posteriormente, para configurar el nuevo tipo de *portlet*, agregamos el archivo *portlets/configure.zcml*:

```
<configure
    xmlns="http://namespaces.zope.org/zope"
    xmlns:plone="http://namespaces.plone.org/plone"
    i18n_domain="persona">

    <include package="plone.app.portlets" />

    <plone:portlet
        name=".directorio"
        interface=".directorio.IDirectorioPortlet"
        assignment=".directorio.Assignment"
        renderer=".directorio.Renderer"
        addview=".directorio.AddForm"
        editview=".directorio.EditForm"
        />
</configure>
```

Estas declaraciones registran algunas utilidades y vistas; si no hay algo que editar, se puede evitar el uso del atributo *editview*.

Finalmente, se debe registrar el nuevo tipo de *portlet*, para que pueda añadirse desde la ventana de administración de *portlets*, esto se logra con el archivo *profiles/default/portlets.xml*:

```
<?xml version="1.0"?>
<portlets>
    <portlet
        addview="directorio"
        title="Personas"
        description="Muestra las personas del directorio" />
</portlets>
```

El atributo *addview* definido en este archivo, debe ser el mismo que el del *portlet* definido en *portlets/configure.zcml*.

k. *Workflows* específicos para el tipo de contenido.

La especificación del *workflow* se realiza mediante documentos XML y estos pueden obtenerse de varias formas, a continuación se muestran los documentos XML necesarios para utilizar un *workflow* sencillo y después se describen formas para generar la definición.

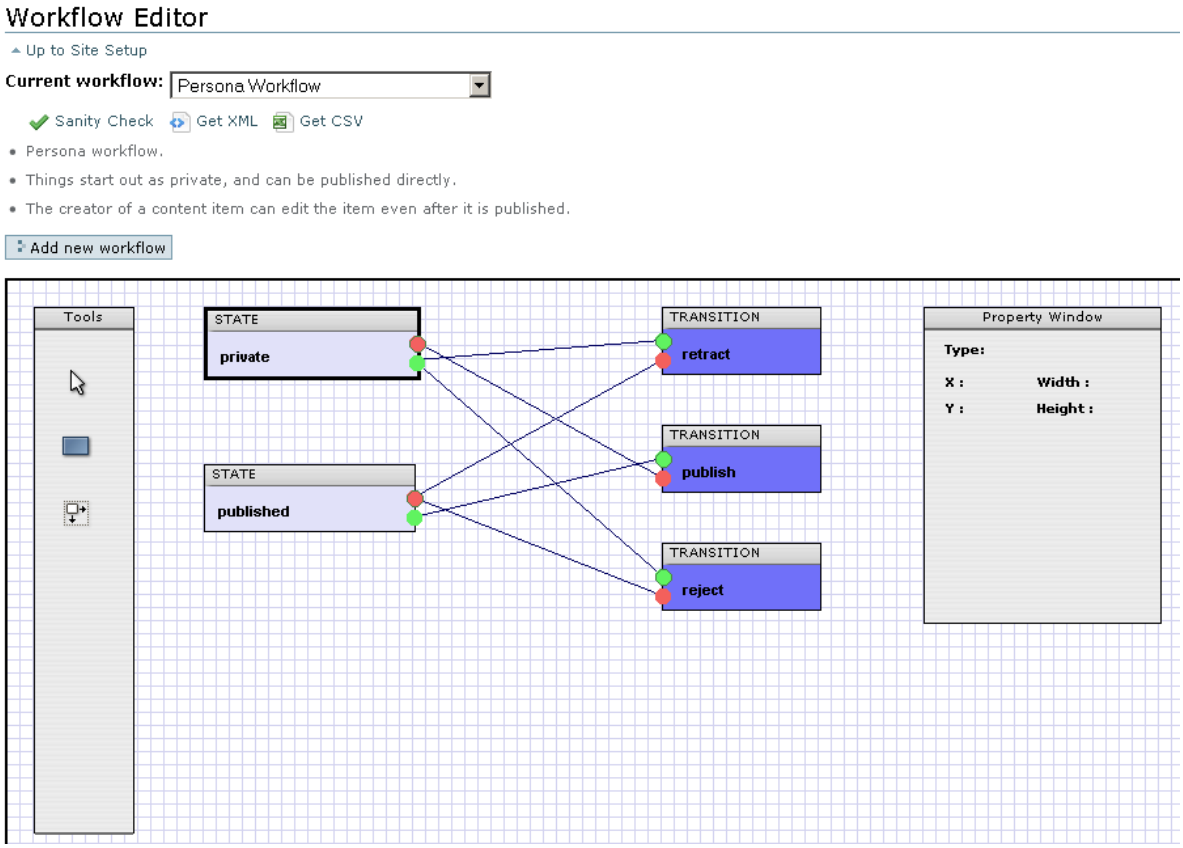

Figura A.1: Estados y transiciones de *persona_workflow*.

Dado que esta especificación es un paso de tipo *GenericSetup*, debe incluirse un archivo llamado *profiles/default/workflows.xml*, el cual se utiliza para definir nuevos *workflows* dentro del sitio:

```
<?xml version="1.0"?>
<object name="portal_workflow" meta_type="Plone Workflow Tool">
 <property
```

```
    name="title">Contains workflow definitions for your
portal</property>
 <object name="persona_workflow" meta_type="Workflow"/>
 <bindings>
  <type type_id="Persona">
   <bound-workflow workflow_id="persona_workflow"/>
  </type>
 </bindings>
</object>
```

Esto crea un nuevo *workflow* dentro del sitio, llamado *persona_workflow*.

Ahora debemos definir las transiciones y estados del nuevo *workflow*; para esto, debemos crear el directorio *profiles/default/workflows/persona_workflow* (nótese que el nombre debe ser exactamente igual que el ID declarado en *workflows.xml*), y dentro de este directorio debe existir un archivo llamado *definition.xml*, a continuación se muestran algunas partes del mismo.

```
<?xml version="1.0"?>
<dc-workflow workflow_id="persona_workflow"
             title="Persona Workflow"
             description=" - Persona workflow. - Things start out as
private, and can be published directly. - The creator of a content
item can edit the item even after it is published."
             state_variable="review_state"
             initial_state="private">
<!—Permisos utilizados por el workflow -->
 <permission>Access contents information</permission>
 <permission>Change portal events</permission>
 <permission>List folder contents</permission>
 <permission>Modify portal content</permission>
 <permission>View</permission>
<!—Los diversos estados con su respectivo mapeo de permisos -->
 <state state_id="private" title="Private">
  <description>Can only be seen and edited by the owner.
</description>
  <exit-transition transition_id="publish"/>
  <exit-transition transition_id="submit"/>
  <permission-map name="Access contents information"
                  acquired="False">
   <permission-role>Manager</permission-role>
   <permission-role>Owner</permission-role>
   <permission-role>Editor</permission-role>
   <permission-role>Reader</permission-role>
   <permission-role>Contributor</permission-role>
  </permission-map>
  <permission-map name="Change portal events"
                  acquired="False">
   <permission-role>Manager</permission-role>
   <permission-role>Owner</permission-role>
   <permission-role>Editor</permission-role>
  </permission-map>
  <permission-map name="List folder contents"
                  acquired="False">
   <permission-role>Manager</permission-role>
   <permission-role>Owner</permission-role>
```

```
   <permission-role>Editor</permission-role>
   <permission-role>Reader</permission-role>
   <permission-role>Contributor</permission-role>
  </permission-map>
  <permission-map name="Modify portal content"
                  acquired="False">
   <permission-role>Manager</permission-role>
   <permission-role>Owner</permission-role>
   <permission-role>Editor</permission-role>
  </permission-map>
  <permission-map name="View" acquired="False">
   <permission-role>Manager</permission-role>
   <permission-role>Owner</permission-role>
   <permission-role>Editor</permission-role>
   <permission-role>Reader</permission-role>
   <permission-role>Contributor</permission-role>
  </permission-map>
 </state>
<state state_id="published" title="Published">
  <description>Visible to everyone, not editable by the owner.
</description>
  <exit-transition transition_id="retract"/>
  <exit-transition transition_id="reject"/>
...
</state>
<!-- Transiciones entre estados, incluyendo condiciones de seguridad
-->
<transition transition_id="publish"
            title="Reviewer publishes content"
            new_state="published" trigger="USER"
            before_script="" after_script="">
  <description>Publishing the item makes it visible to other users.
</description>
  <action
url="%(content_url)s/content_status_modify?workflow_action=publish"
         category="workflow">Publish</action>
  <guard>
   <guard-permission>Review portal content</guard-permission>
  </guard>
 </transition>
<transition transition_id="reject"
            title="Reviewer send content back for re-drafting"
            new_state="private" trigger="USER"
            before_script="" after_script="">
...
</transition>
 <transition transition_id="retract"
            title="Member retracts submission"
            new_state="private" trigger="USER"
            before_script="" after_script="">
...
 </transition>
<!-- Otras definiciones del workflow -->
...
</dc-workflow>
```

Primero, definimos algunas propiedades del *workflow*, ID, nombre del estado inicial y nombre de la variable que almacena el estado actual (debe ser siempre *review_state*). A continuación, enumeramos los permisos que se usarán.

Posteriormente, definimos los diversos estados y transiciones. Para cada estado, establecemos un conjunto de posibles transiciones de salida (*exit transitions*) que estarán disponibles en el menú **Estado**, al igual que el mapeo de permisos en el estado particular.

Las transiciones tienen un ID, un título (*title*), el estado al cual llega (*new_state*) y un tipo de disparo (*trigger*), ya sea levantado por el usuario (*USER*) o automático (*AUTOMATIC*). La etiqueta *<action />*, contiene el nombre que se mostrará en el menú **Estado**, y la URL que se llamará si el usuario elige esta transición. Finalmente, establecemos los permisos "guardia", si el usuario actual no tiene alguno de ellos, la transición no estará disponible.

El código omitido define otros estados, transiciones y mapeos de permisos, además de algunas variables de *workflow* estándar (utilizadas para el historial del mismo).

Ahora, para obtener estos archivos de definición del *workflow*, existen cuatro métodos principales:

A. Crear una copia de algún *workflow* definido dentro del sitio, modificarlo desde el ZMI, una vez que se tiene la funcionalidad deseada, exportar el paso de *workflow tool* de la configuración del sitio.

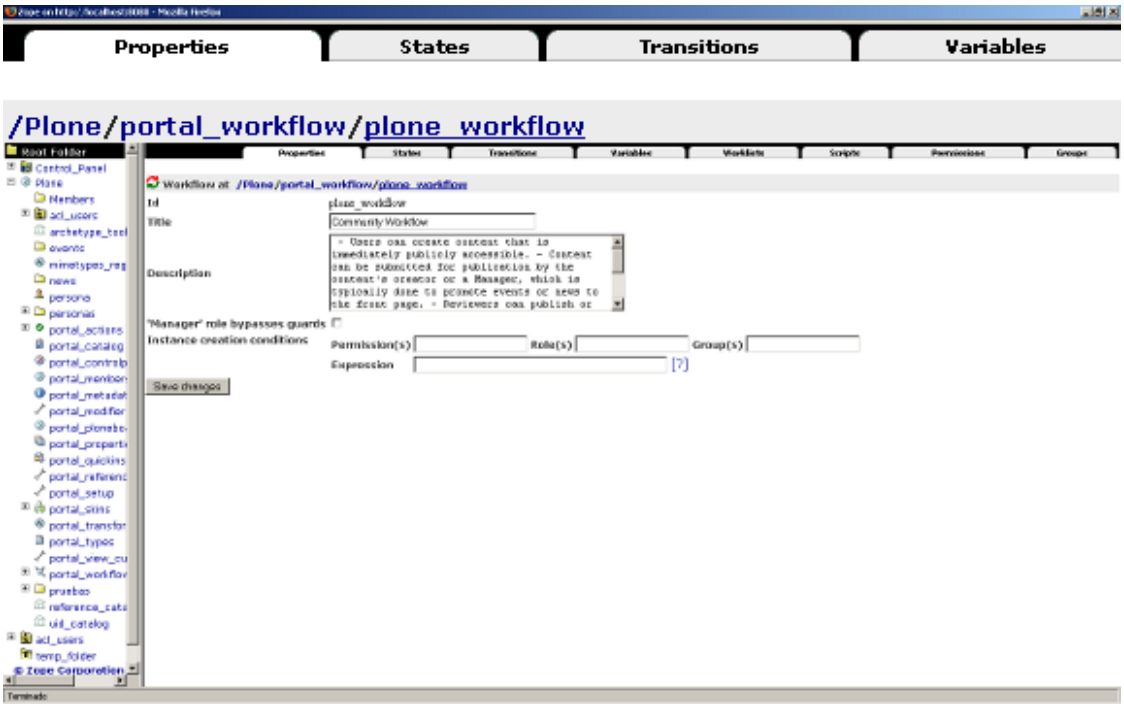

Figura A.2: Interfaz para modificar *workflows* dentro del ZMI.

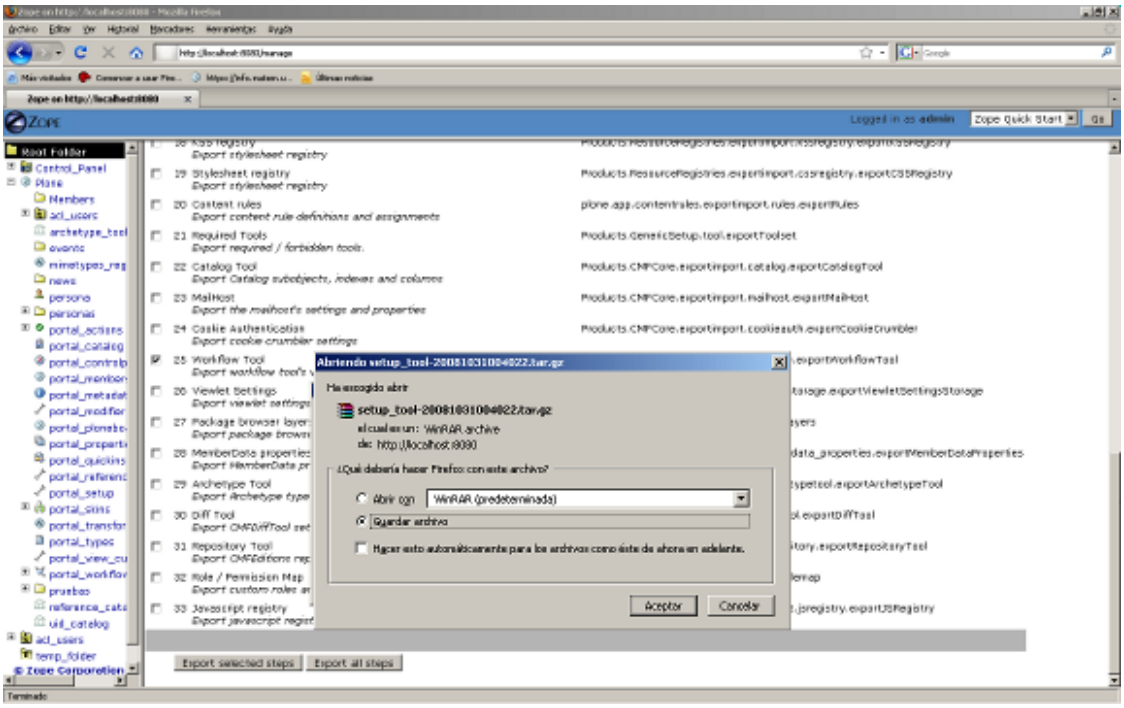

Figura A.3: Exportación del *workflow tool*.

Lo anterior genera una archivo comprimido con la definición de todos los *workflows* del sitio; se debe descomprimir este archivo, editar el archivo *workflows.xml* generado, dejando solamente lo concerniente al *workflow* deseado y eliminar todas las carpetas que contienen la definición de los otros *workflows*, dejando solo el que necesitamos. Finalmente, copiar el archivo *workflows.xml* y la carpeta con la definición al directorio *profiles/defatul/* del contenido que estamos creando.

B. Utilizar el editor gráfico de *workflows* llamado *collective.workflowed* [40] (*http://plone.org/products/collective-workflowed*); este genera las definiciones necesarias para colocarlas dentro de la carpeta correspondiente.

C. Utilizar el producto *ArchGenXML* (*http://plone.org/products/archgenxml*) que es una herramienta que toma un diseño UML (generalmente realizado en con *argoUML*, *Poseidon* u *ObjectDomain*) y a partir de este, genera productos para Zope, basados en el marco de trabajo de *Archetypes*, incluyendo la definición de *workflows* para Plone 3 (anteriormente no tenía esta capacidad).

D. Si se cuenta con un *workflow* definido en alguna versión de Plone 2.x, lo más conveniente, es realizar la exportación de este a un archivo *.zexp* desde el ZMI de esa versión de Plone, después importar ese archivo al sitio existente en Plone 3, y posteriormente exportar el paso de *workflow tool* de la configuración del sitio. Una vez hecho esto, seguir la descripción de la segunda parte del inciso A.

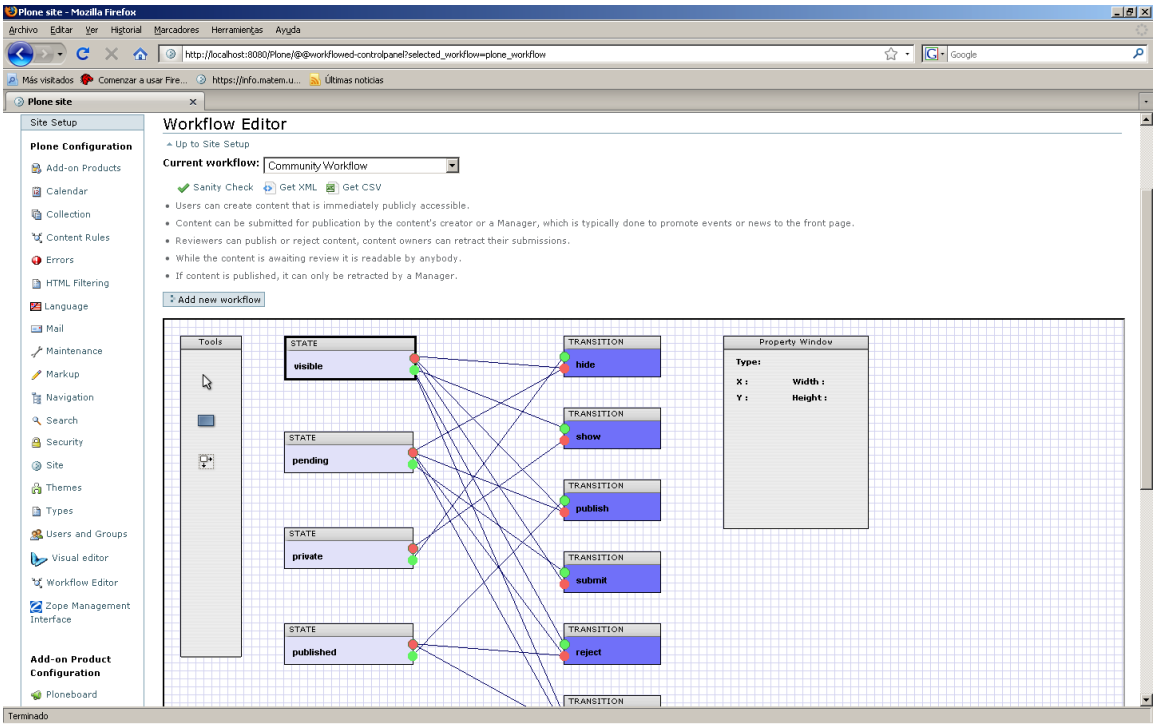

Figura A.4: El editor *collective.workflowed*.

I. Internacionalización del tipo de contenido.

Un sitio web *multilenguaje*, debe ofrecer su contenido en múltiples lenguajes; incluso es posible elegir el lenguaje adecuado de forma automática, dado que el navegador envía un lenguaje preferido como parte de la petición al servidor.

Los productos basados en *Archetypes* se pueden traducir utilizando LinguaPlone (*http://plone.org/products/linguaplone*); usando este producto, no se necesitan realizar muchos cambios al producto que estamos desarrollando. Los autores de contenidos pueden crear versiones traducidas de sus elementos; los usuarios pueden navegar el sitio de forma normal, pero solo verían los contenidos en su idioma preferido.

Es buena práctica desarrollar productos que sigan los estándares de traducción desde el principio, esto facilitará la tarea de traducir toda la aplicación en el futuro. Por ejemplo, en alguna plantilla del sitio, podríamos tener alguna etiqueta como:

```
<a href="http://plone.org" title="The Plone home page"
i18n:attributes="title title_plone_homepage;"
i18n:translate="link_plone_homepage">Visit Plone's home page!</a>
```

El atributo *i18n:translate* especifica que la cadena que se encuentra encerrada por las etiquetas será traducida, buscando el ID de mensaje llamado *link_plone_homepage*; el atributo *i18n:atributes*, enlista los atributos de la etiqueta que serán traducidos (en este caso solo *title*).

Los IDs de mensaje son opcionales en ambos casos, si no se especifican, la cadena completa se utiliza para la traducción. Esto puede resultar más complicado para os traductores y generalmente es útil solo para cadenas pequeñas y que no puedan presentar ambigüedades (de una o dos palabras):

```
<a href="http://plone.org" title="Info"
i18n:attributes="title"
i18n:translate="">Info</a>
```

Existen herramientas de extracción de cadenas de traducción como *i18ndude* (*http://plone.org/products/i18ndude*) que son capaces de buscar estas cadenas dentro de plantillas y código de Python, al igual que archivos XML (de *GenericSetup*), y a partir de ellas, crear un archivo de catálogo de mensajes, con extensión *.pot*, que se coloca dentro del subdirectorio *i18n/* del producto. Este archivo puede editarse para traducir los mensajes en archivos con extensión *.po*.

En general, estos archivos *.po* contienen líneas como las siguientes para cada ID de mensaje que se desee traducir:

```
#. Default: "Visit Plone's home page!"
#: nombre del archivo en el que se encuentra
msgid " link_plone_homepage"
msgstr "¡Visita la página principal de Plone!"

#. Default: "Info"
#: nombre del archivo en el que se encuentra
msgid "Info"
msgstr "Información"
```

Las primeras dos líneas son comentarios, indicando el valor default que se desplegará, es importante señalar que el lenguaje utilizado para las traducciones es inglés, por tanto, los valores default deben ser en este idioma; la segunda línea indica el nombre del archivo en el que se encuentra la frase a traducir.

La tercer línea define el ID del mensaje a traducir, y cuarta línea el texto que será mostrado si el usuario prefiere utilizar la versión en el idioma del archivo actual (en este caso, español). En el segundo ejemplo, el ID de mensaje corresponde con el texto encerrado entre las etiquetas.

Como se dijo anteriormente, el navegador le indicará a Plone que lenguaje es el preferido por el visitante y, de ser posible mostrará las páginas en ese lenguaje, si no existe traducción para ese idioma, muestra el del lenguaje por default.

Si se desea saber más sobre herramientas de traducción, se puede consultar *http://plone.org/development/teams/i18n*, además el documento *http://plone.org/documentation/how-to/i18n-for-developers* puede resultar muy útil para asegurar que las plantillas que utilizamos sean traducibles.

B) Glosario.

ACL: *Agent Communication Languages*. Los Lenguajes de Comunicación de Agentes son lenguajes de alto nivel diseñados especialmente para negociación, contratación, colaboración e intercambio de información, los requisitos básicos de las sociedades de agentes. No confundir con el contexto de seguridad informática, *Access Control List*, Lista de Control de Acceso.

Agencia: Manera en la que se estructuran las sociedades de agentes.

Agente: Actores que realizan diferentes tareas. Cada actor desempeña un papel que se define como una descripción generalizada de la responsabilidad del actor en una tarea.

Arquitectura: Esquema que indican la estructura, funcionamiento e interacción entre las partes del software o hardware.

Arquitectura centralizada: Se basa en la existencia de un equipo servidor que almacena los datos y/o las aplicaciones que los procesan. Los clientes se comportan sólo sirven para introducir datos.

ASP: *Active Server Pages* (ASP) es un framework para aplicaciones web desarrollado y comercializado por Microsoft. Es usado para construir sitios web dinámicos, aplicaciones web y servicios web XML.

Base de datos orientada a objetos: Es una base de datos que soporta el paradigma orientado a objetos. Almacena objetos complejos.

Base de datos relacional: Es una base de datos en donde todos los datos visibles al usuario están organizados estrictamente como tablas de valores, y en donde todas las operaciones de la base de datos operan sobre estas tablas.

Caso de uso: Es una técnica para la captura de requisitos potenciales de un nuevo sistema o una actualización de *software*.

Contenido dinámico: Son páginas Web que son personalizadas o creadas para cada usuario "en tiempo real", basándose en las acciones y/o peticiones del usuario.

CORBA: *Common Object Request Broker Architecture*. Arquitectura común de intermediarios en peticiones a objetos, es un estándar que establece una plataforma de desarrollo de sistemas distribuidos facilitando la invocación de métodos remotos bajo un paradigma orientado a objetos.

Deadlock: Cuando dos procesos están en un estado de ejecución, y requieren intercambiar recursos entre sí para continuar, ambos procesos están esperando por la liberación del recurso requerido, que nunca será realizada

Divergencia del sistema: Indica que el sistema no alcanzará un estado estable.

Estándar I18N: Estandariza la respuesta del sistema de acuerdo con la configuración del idioma del navegador, de tal forma que el usuario podrá escoger el idioma con el que se sienta más cómodo para trabajar.

Framework: Marco de trabajo.

Gestión de contenidos: Es una estrategia empleada en la industria de la tecnología de la información para administrar la captura, almacenamiento, seguridad, control de versiones, recuperación, distribución, conservación y destrucción de documentos y contenido.

Gramática: Es una descripción sincrónica del lenguaje de un sistema.

IIOP: *Internet Inter-ORB Protocol*. Protocolo que permite escribir programas distribuidos en diferentes lenguajes de programación para comunicarse sobre Internet

Instancia de *workflow*: Se refiere a una ejecución del *workflow* definido.

IP: Una dirección IP es un número que identifica de manera lógica y jerárquica a una interfaz de un dispositivo (generalmente una computadora) dentro de una red que utilice el protocolo IP (Internet Protocol), que corresponde al nivel de red o nivel 3 del modelo de referencia OSI.

JSP: Es una tecnología Java que permite generar contenido dinámico para web, en forma de documentos HTML, XML o de otro tipo, es un desarrollo de la compañía Sun Microsystems.

KQML: *Knowledge Query and Manipulation Language*, Lenguaje de Consulta y Manipulación del Conocimiento. Fue el primer intento de estandarizar la comunicación de agentes. Se compone de un conjunto de mensajes de propósito general y una serie de mensajes utilizados por el sistema.

Legacy systems: Son sistemas ó aplicaciones viejas que continúan utilizándose, debido a que el usuario no desea que sea reemplazado o rediseñado.

Lisp: Es el segundo lenguaje de programación, después de Fortran, de alto nivel. Lisp es de tipo declarativo y fue creado en 1958 por John McCarthy y sus colaboradores en el MIT.

Livelock: Ambos procesos están realizando computaciones efectivas, aunque en conjunto no progresan, sin embargo cualquier desviación de la secuencia en el ciclo puede provocar que el lazo se rompa y el sistema progrese de nuevo (cosa que es imposible en un deadlock).

Mensaje: Es el objeto de la comunicación. Está definido como la información que el emisor envía al receptor a través de un canal determinado o medio de comunicación (como el habla, la escritura, etc.).

Metaworkflow: *Framework* que integra motores de *workflow* heterogéneos, herramientas de *software*, datos, recursos de *hardware*, límites organizacionales y/o dominios de búsqueda.

Migración: Se refiere a reescribir o portar un sistema a otro lenguaje de programación, bibliotecas de software, protocolos, o plataforma de hardware.

Motor de *workflow*: Es una herramienta de diseño, modelado, y administración de *workflows* (flujos de trabajo) que permite definir procesos de una manera sencilla y completa, automatizar la conducción de estos a través de su organización y llevar un control adecuado sobre los documentos y actividades relacionadas.

Ontología: En informática hace referencia a la formulación de un exhaustivo y riguroso esquema conceptual dentro de un dominio dado, con la finalidad de facilitar la comunicación y la compartición de la información entre diferentes sistemas.

PHP: PHP es un lenguaje de *scripting* embebido en HTML, significa *PHP: Hypertext Preprocessor* (PHP: Pre-procesador de Hipertexto).

Producto: componente de *software* el cual encapsula alguna funcionalidad expuesta a través de interfaces.

Redes de Petri: Las redes de Petri son una herramienta gráfica y matemática de modelación que se puede aplicar en muchos sistemas. Particularmente son ideales para describir y estudiar sistemas que procesan información y que tienen características concurrentes, asíncronas, distribuidas, paralelas, no determinísticas y/o estocásticas.

Scheduler: Planificador, programador de tareas. Es un componente funcional muy importante de los sistemas multitarea y multiproceso, y es esencial en los sistemas de tiempo real. Su función consiste en repartir el tiempo disponible de un recurso entre todos los procesos que están disponibles para su ejecución

Schema: Es una *interface* extendida que define campos.

Sistema administrador de contenido (CMS): sistemas que permiten la creación, administración, distribución y búsqueda de contenido. Cubren todo el ciclo de vida de las páginas de un sitio Web desde la creación y publicación del contenido hasta que es quitado y almacenado.

Subclase: Concepto utilizado en la programación orientada a objetos. Una subclase desciende de otra de tipo similar heredando ciertas características de la clase padre (e incluso pueden redefinirse o agregarse nuevas características de la clase superior también).

Superclase: En una estructura jerárquica, la clase padre de cualquier clase es conocida como *superclase.* A partir de esta clase se pueden derivar muchas otras.

TCP: El protocolo TCP (*Transmission Control Protocol*, protocolo de control de transmisión) está basado en IP. Es orientado a conexión, ya que es necesario establecer una conexión previa entre las dos máquinas antes de poder transmitir ningún dato. A través de esta conexión los datos llegarán siempre a la aplicación destino de forma ordenada y sin duplicados.

Token: En programación, un elemento individual en un lenguaje de programación. Es un bloque de texto categorizado. Por ejemplo una marca de puntuación, un operador, un identificador, un número, etc.

UML: Lenguaje Unificado de Modelado (*Unified Modeling Language*) es el lenguaje de modelado de sistemas de *software* más conocido y utilizado en la actualidad. Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema. UML ofrece un estándar para describir un "plano" del sistema (modelo), incluyendo aspectos conceptuales tales como procesos de negocio y funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes reutilizables.

Verificable: Que se puede verificar. Que permite comprobar su verdad y examinar el método por el que se ha alcanzado

***Workflow* privado**: Es una instancia de workflow que sólo es usada por un conjunto de agentes (no todos).

***Workflow* público**: Es una instancia de workflow que puede ser usada por todos los agentes.

XML-RPC: Es un protocolo de llamada a procedimiento remoto que usa XML para codificar los datos y HTTP como protocolo de transmisión de mensajes.

Referencias.

[1] *https://info.matem.unam.mx*. Sistema de información del Instituto de Matemáticas.

[2] *http://www.matem.unam.mx*. Página del Instituto de Matemáticas.

[3] G. K. Thiruvathukal and K. Laufer. *Plone and content management*. Computing in Science and Engineering, pp. 88-95, 2004.

[4] M. A López Rabadán. *Sistema sobre Plone para la captura y recolección de información curricular del Instituto de Matemáticas*, Tesis de Maestría, 2007.

[5] S. McKeever. *Understanding web content management systems: evolution, lifecycle and market*. Industrial Management and Data Systems, pp. 686-692, 2003.

[6] B. Boiko. *Content management bible*, Wiley, 2a edición, 2005.

[7] *www.rae.es*. Sitio de la Real Academia Española.

[8] S. R. G. Fraser. *Real-World ASP.NET: Building a content management system*, Apress, 2002.

[9] S. Pastore. *Web content management systems: using Plone open source software to build a website for research institute needs*. icdt, pág. 24, 2006.

[10] A. McKay. *The Definitive Guide to Plone*. Apress, 2004.

[11] J. Robertson. *So, what is a content management system*? KM Column, Step Two Designs, pp. 1-4 ,Junio 2003.

[12] I. Göhlert and T. Höning. *Open source content management systems for small and medium-sized enterprises*. Julio 2004.

[13] Lutz, Mark. Learning Python, 3a edición, 2008.

[14] M. Pelletier and A. Latteier. *The Zope Book*.
http://www.zope.org/Documentation/Books/ZopeBook/2_6Edition/view
N. Díaz A. (traductor)
http://usuarios.lycos.es/zope/Indice.html

[15] J. Fulton. *Introduction to the zope object database*.
http://www.python.org/workshops/2000-01/proceedings/papers/fulton/zodb3.html
In Proceedings of the 8th International Python Conference, 2000.

[16] M. Aspeli: *Professional Plone Development*, Packt Publishing, Birmingham UK, September 2007.

[17] *www.plone.org*. Sitio oficial de Plone.

[18] R. Thimm and T. Will. *AMIRA. A State of the Art Analysis of Workflow Modelling and Agent-based Information-Systems*. University of Trier. Department of Business Information Systems II. 2004.

[19] R. Eshuis and R. Wieringa. *Comparing Petri Net and Activity Diagram Variants for Workflow Modelling - A Quest for Reactive Petri Nets*. In: Petri Net Technology for Communication-Based Systems, Volume 2472 of Lecture Notes in Computer Science / Hartmut Ehrig, Wolfgang Reisig, Grzegorz Rozenberg and Herbert Weber (Eds.). Springer-Verlag, pp. 321-351, (November 2003).

[20] H. Hollingsworth: *Workflow Management Coalition: The Workflow Reference Model*. Future Strategies Inc., Lighthouse Point, FL, USA (June 1996).

[21] *WfMC: Workflow Management Coalition: Terminology & Glossary*. Future Strategies Inc., Lighthouse Point, FL, USA (February 1999).

[22] V. der Aalst, *W.M.P.: The Application of Petri Nets to Workflow Management*. In The Journal of Circuits, Systems and Computers, pp. 21-66, (1998).

[23] D. Georgakopoulos, M. Hornick and A. Sheth. *An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure*. Distributed and Parallel Databases, An International Journal, 3, (1995).

[24] Workflow Management Coalition (WfMC), *Interface 1: Process Definition Interchange Process Model* (Workflow Management Coalition, Winchester, UK, 1999).

[25] M. Zur Muehlen: *Organizational Management in Workflow Applications – Issues and Perspectives*, Kluwer Academic Publishers. The Netherlands, pp.271-291, (2004).

[26] M. Rosemann: *Objectives and Requirements of Integration Management*, in: Integration Management, eds. A. W. Scheer, M. Rosemann and R. Schuette, University of Muenster, Muenster, Germany, pp. 5-18, (1999).

[27] F. Casati: *Models, Semantics, and Formal Methods for the Design of Workflows and Their Exceptions*, University of Milan, Milan, (1998).

[28] C. Combi and G. Pozzi, *Architectures for a temporal workflow management system*, Source Symposium on Applied Computing archive, Proceedings of the 2004 ACM symposium on Applied computing, ACM New York, USA, pp. 659-666, (2004).

[29] R. Conradi and M. L. Jaccheri; Chapter 3, *Process Modelling Languages.* In Software Process: Principles, Methodology and Technology, Springer-Verlag Berlin/Heidelberg, pág. 27, (May 2003).

[30] P. Hruby. *Structuring Specification of Business Systems with UML (with an Emphasis on Workflow Management Systems)*. In OOPSLA 97-98 Published Proceedings p. 77ff, Springer-Verlag, pág. 77, (1998).

[31] B. Chaib-Draa and F. Dignum. *TRENDS IN AGENT COMMUNICATION LANGUAGE*. In International Journal of Computational Intelligence , Vol. 18, Nr. 2 (2002).

[32] M. Dastani, V. Dignum and F. Dignum: *Role-assignment in open agent societies*. In International Conference on Autonomous Agents: Proceedings of the second international joint conference on Autonomous agents and multiagent systems, Melbourne, Australia, Session: Groups and organizations. ACM Press New York, NY, USA, pp. 489-496, (2003).

[33] *FIPA TC B: FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS* FIPA Agent Management Specification. www.fipa.org (2000).

[34] M. B. Blake: *Using Agent Control and Communication in a Distributed Workflow Information System*. In R. CoopIS/DOA/ODBASE 2002, LNCS 2519, Springer-Verlag Berlin/Heidelberg, pp. 163-178, (2002).

[35] D. Beneventano, S. Bergamaschi, G. Gelati, F. Guerra and M. Vincini; *MIKS: an Agent Framework supporting information access and integration*. In Lecture Notes in Computer Science: Intelligent Information Agents: The AgentLink Perspective. Springer-Verlag Berlin/Heidelberg, pp. 22-49, (July 2003)

[36] R. S. S. Filho, J. Wainer and E. R. M. Madeira: *A Fully Distributed Architecture for Large Scale Workflow Enactment*. In International Journal of Cooperative Information Systems, Vol. 12, No. 4, pp. 411-440, (2003).

[37] A. I. Wang: *Using software agents to support evolution of distributed workflow models*. In International ICSC Congress on Intelligent Systems and Applications, Wollongong, Australia (December 2000).

[38] B. R. Odgers, J. W. Shepherdson and S. G. Thompson: *Distributed Workflow Co-ordination by Proactive Software Agents*. Proceedings of Intelligent Workflow and Process Management, IJCAI-99 Workshop (August 1999).

[39] R. S. Pressman. *Ingeniería de Software: Un enfoque práctico*. McGraw-Hill, 2002.

[40] *http://blog.delaguardia.com.mx*. Blog personal de Carlos de la Guardia.

[41] M. Aspeli: *http://plone.org/documentation/tutorial/richdocument*, tutorial de *RichDocument*, agosto 2007.

[42] *http://unam.delaguardia.com.mx*. Sitio del *primer taller de Plone México*, julio 2008.

[43] *http://dublincore.org*. Sitio de *Dublin Core Metadata Initiative*.

Sitios de interés.

http://plone.org/documentation. Documentación oficial de Plone.

http://plone.net/sites. Listado de sitios que utilizan Plone.

http://www.python.org. Sitio web oficial de Python.

http://freemind.sourceforge.net/wiki/index.php/Main_Page. Sitio oficial de *Freemind*, herramienta de software libre para realizar mapas mentales.

http://www.wfmc.org; *http://www.e-workflow.org*. Sitios oficiales de la *Workflow Management Coalition*.

http://plone.org/products/archetypes. Manual de referencia de *Archetypes*.

http://plone.org/products/collective-workflowed. Editor gráfico de *workflows*.

http://www.w3.org/International/techniques/developing-specs. Especificación del estándar de internacionalización con i18n.