UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

# TESINA DE LICENCIATURA

**Título:** Software Libre en Gobierno Digital. Soluciones para la gestión municipal.
**Autores:** Laferrara, Víctor Gabriel. Lastra, Fabricio Carlos. Maceira, Roberto Ezequiel.
**Director:** Prof. Lic. Francisco Javier Díaz.
**Codirector:**
**Asesor profesional:**
**Carrera:** Licenciatura en Informática.

## Resumen

*A través de este trabajo, se propone un recorrido que va desde la presentación de una serie de conceptos básicos relacionados con el software libre y los fundamentos de la necesidad de su implementación en el ámbito gubernamental, pasando por una revisión del Estado del Arte a nivel mundial del fenómeno del software libre y de código abierto (FOSS) en torno a las administraciones públicas y finalizando con la presentación de algunos detalles que tienen que ver con el desarrollo de la solución FOSS propuesta.*

*Dentro de las descripciones que tienen que ver con el desarrollo, el cual abarca el proceso completo de relevamiento, análisis, diseño (tanto del diagrama de datos como del modelo de objetos) e implementación, se incluye además, la especificación de las herramientas de desarrollo utilizadas, la tecnología y estándares involucrados, la definición de la arquitectura, como así también detalles de implementación donde se muestra cómo han sido implementadas las diferentes capas y el uso de dichas tecnologías y estándares.*

## Palabras Claves

*Software Libre. Gobierno Digital. FOSS. Open Source. Gestión Municipal. Estado. Administración Pública. E-Goverment. Gobierno Electrónico. Dependencia Tecnológica. Estándares abiertos.*

## Trabajos Realizados

*Investigación sobre el estado del arte de las soluciones de software libre para Gobierno Digital a nivel mundial y desarrollo del prototipo de la solución FOSS que presentamos, lo cual abarca el relevamiento, análisis, diseño e implementación de nuestro aporte: dos soluciones para la gestión municipal, Sistema de Ingresos Públicos y Sistema de Liquidación de Haberes.*

## Conclusiones

*Creemos que es necesaria la implementación de soluciones FOSS en el ámbito de la administración pública, puesto que cada una de las características de FOSS se adapta perfectamente al ambiente gubernamental debido a que la información es pública y propiedad de los ciudadanos. Por su parte, con el desarrollo de nuestra solución, hemos demostrado que se pueden construir soluciones robustas sin necesidad de obligar al estado a una dependencia tecnológica y económica de empresas privadas.*

## Trabajos Futuros

*Debido a que los organismos públicos cuentan con estructuras y problemáticas similares, es infinita la cantidad de proyectos que podrían y deberían implementarse con software libre y de código abierto (FOSS) en forma conjunta, compartiendo la experiencia y su financiamiento. Como puntapié inicial, se podría continuar el desarrollo del resto de los módulos analizados, para construir así una opción más que interesante para la gestión de municipios de diferente envergadura.*

**Fecha de la presentación:** Noviembre 2012

# Contenido

Nuestro trabajo se encuentra organizado de la siguiente manera:

- **Introducción**: presenta una serie de conceptos básicos relacionados con el *software* libre y los fundamentos de la necesidad de su implementación en el ámbito gubernamental.
- **Estado del arte**: describe la situación actual a través del análisis detallado de diferentes proyectos relacionados con este tema.
- **Solución presentada:** detalla el análisis, diseño e implementación de nuestro aporte: dos soluciones para la gestión municipal (Sistema de Ingresos Públicos y Liquidación de Haberes).
- **Conclusión y trabajos futuros:** destaca los aspectos relevantes del trabajo realizado y propone futuras ampliaciones a nuestras soluciones y nuevos desarrollos.

# Introducción

En estas líneas, pretendemos hacer mención de las razones por las cuales los gobiernos deberían, a nuestro criterio, utilizar y promover el uso de *software* libre. Por lo tanto, antes que nada, presentaremos una clasificación que divide el *software* en dos grandes grupos o tipos predominantes: el *software* **libre** y el **privativo** o "bajo licencia". *Software* libre es aquel respecto del cual el usuario tiene amplios derechos de uso, distribución y modificación. Por su parte, el *software* privativo es aquel que restringe los derechos del usuario al mero uso de sus funciones, bajo ciertas condiciones establecidas por el dueño de los derechos de autor. La principal diferencia entre el *software* libre y el privativo consiste, entonces, en las libertades que otorga el primero, que permite a los usuarios no solo ejecutar las aplicaciones en tantas computadoras como deseen, sino también copiarlo, inspeccionarlo, auditarlo, modificarlo, mejorarlo, adaptarlo a sus necesidades, corregir errores y distribuirlo.

Veamos algunas cuestiones para entender mejor la importancia del uso de *software* libre en el Estado.

**Seguridad**: para cumplir con sus funciones de administrador público, el Estado debe almacenar y administrar información relativa a los ciudadanos. La integridad de estos datos debe ser resguardada de riesgos específicos. En una publicación de abril de 2001, Federico Heinz [1], presidente de la Fundación Vía Libre y ex miembro del consejo de la FSFLA (Fundación *Software* Libre América Latina), hace mención de tres riesgos específicos en cuanto al *software* y la seguridad nacional, a saber:

- Solo las personas e instituciones autorizadas deben tener acceso a los datos confidenciales.
- Los datos deben estar almacenados de manera tal que su acceso esté garantizado durante toda la vida útil de la información.
- La modificación de los datos debe estar restringida solo a las personas e instituciones autorizadas.

Según lo expresado por Heinz en dicho documento, faltar a cualquiera de estos tres principios puede traer consecuencias graves, y, cuando los datos son procesados electrónicamente, el *software* que se emplea para tal fin puede suscitar vulnerabilidades a estos riesgos.

**Dependencia tecnológica**: una vez que una tarea se informatiza, las aplicaciones utilizadas para cumplirla se vuelven imprescindibles: la tarea pasa a depender de su disponibilidad. Si la institución no tiene libertad de contratación para ampliar o corregir los sistemas, se produce una dependencia tecnológica en la que el proveedor está en condiciones de dictar unilateralmente los términos, plazos y precios. Un ejemplo de dependencia tecnológica puede verse en la misma Argentina. Desde hace un tiempo, la Administración Federal de Ingresos Públicos (AFIP) exige a los contribuyentes la presentación de sus declaraciones juradas en formato digital, lo cual es razonable; lo que no lo es, sin embargo, es la forma en la cual se implementó: las declaraciones podrán generarse exclusivamente a través de un *software* provisto por la AFIP. Estos programas son gratuitos, pero solo corren en plataforma Windows; por lo tanto, el Estado está obligando a sus ciudadanos a comprar licencias de Windows a fin de poder cumplir con sus obligaciones impositivas. Algo similar ocurre con la Agencia de Recaudación de la Provincia de Buenos Aires (ARBA), ente recaudador estatal, que implementó el sistema informático para la declaración jurada del impuesto a los Ingresos Brutos. Dicha aplicación se distribuye de forma gratuita (se la puede bajar desde la página *web* oficial del organismo, arba.gov.ar), pero se integra únicamente con la plataforma SIAP, que corre solo bajo un entorno Windows.

Un caso adicional de dependencia tecnológica se da en la exigencia del gobierno provincial a los municipios de implementar la aplicación RAFAM, que corre en una plataforma Windows y tiene como soporte de base de datos a Oracle. De nuevo, dichas aplicaciones son gratuitas, pero se obliga a las administraciones municipales a adquirir licencias de Windows y Oracle para poder cumplir con los mandatos provinciales.

**Informatización de la Administración Pública**: El Estado administra información que es, a la vez, sobre los ciudadanos y de su pertenencia. El ignorar cómo opera el *software* privativo implica exponer estos datos a un riesgo injustificable.

Según Heinz[1], desde el punto de vista social y estratégico, el uso de *software* libre es imperativo. Esta es la única manera de garantizar no sólo la democratización del acceso a la información y los sistemas del Estado, sino también la competitividad de la industria local de *software*, que constituye una fuente potencial de trabajo de altísimo valor agregado.

Del mismo modo, en su publicación, sostiene que la dependencia tecnológica es inaceptable para el Estado, ya que los contribuyentes se ven forzados a adquirir *software* de determinada marca y modelo a fin de cumplir con las obligaciones tributarias.

El Estado debería beneficiarse en todos sus niveles con las ventajas del *software* libre. Por ejemplo, las provincias con estructuras y problemáticas similares deberían unirse para financiar el desarrollo de una solución libre y compartirla entre todas.

El desarrollo de *software* libre y de código abierto está siendo impulsado por los gobiernos de diversos puntos del globo y, en especial los de la Unión Europea, que admiten que es una tendencia a nivel mundial con grandes ventajas. Esta idea ha sido puesta de manifiesto en diferentes oportunidades([2], [3], [4], [5], [6]) por los diferentes poderes europeos, en las que han instado a su respectivo Poder Ejecutivo a fomentar el desarrollo de aplicaciones libres y de código abierto, fundamentalmente en los sistemas que serán usados por los ciudadanos, o a llevar a los escritorios de trabajo de sus asambleístas sistemas operativos basados en Linux y *software* libre para las herramientas de escritorio, bajo la consideración de que el *software* libre conlleva una serie de libertades y capacidades que están muy por encima de las ofrecidas por el *software* bajo licencia (como afirma Bernard Carayon, asambleísta francés, en su informe).

Estas decisiones están muy bien fundamentadas, ya que el uso de *software* libre y el desarrollo de aplicaciones bajo el paradigma *open source* suscitan innumerables ventajas y brinda un atractivo número de oportunidades para sus gobiernos. Entre las ventajas que hemos visto enumeradas a través de los documentos que hemos relevado y citamos al pie, podemos mencionar:

- El *software* libre no requiere pagos anuales en concepto de licencias, lo que trae como consecuencia un considerable ahorro.[6]
- El reducido peso virtual de las soluciones *open source* y su actualización automática permite una mayor vida útil de los equipos informáticos.[6]
- La idoneidad, seguridad e interoperabilidad tecnológica alcanzadas con el *software* libre no se comparan con las tecnologías bajo licencia o privativas.[4]
- La modernización de los estados a partir de la implementación de soluciones *open source* posibilita la libertad de elección, la protección de la inversión, una mejor relación costo/beneficio y la garantía de comunicación e interoperabilidad.[6]
- La migración a *software* libre supone una **independencia tecnológica** inigualable ya que evita una relación exclusiva con su fabricante, lo que defiende el interés general de los ciudadanos y ofrece una mayor seguridad.[6]
- Al tener la libertad de inspeccionar el mecanismo de funcionamiento del *software* y la manera en que este almacena los datos, junto con la posibilidad de modificar estos aspectos, queda en manos del Estado, y no en manos privadas, la llave del acceso a la información.[7]
- El *software* libre, al ser público, está sometido a la inspección por parte de muchas personas, que pueden buscar problemas, solucionarlos, y compartir la solución con los demás. Debido al denominado "Principio de Linus" ("dada la suficiente cantidad de ojos, cualquier error del *software* es evidente"), los programas libres gozan de un excelente nivel de confiabilidad y estabilidad, requerido para las aplicaciones críticas del Estado.[7]

Este gran número de ventajas, junto con algunas otras que no hemos mencionado aquí, ha llevado a los gobiernos a pasar del campo de la teoría al de la práctica: en países como Francia, Brasil, Dinamarca, Argentina, Perú, Italia, España, Australia, Bélgica, Colombia, Costa Rica, Estados Unidos, Portugal y Ucrania se han presentado proyectos de ley para el uso de *software* libre dentro del Estado; mientras que en Alemania, Brasil, Chile, China, Corea, Estados Unidos, Eslovenia, España, Filipinas, Francia, Holanda, India, Italia, México, Pakistán, Polonia, Sudáfrica, Suiza, Tailandia y Venezuela, se han comenzado a implementar políticas concretas para la migración a *software* libre en las dependencias estatales. Al mismo tiempo, la Unión Europea está empleando políticas de desarrollo tecnológico a través de programas específicos para el impulso del *software* libre.

Por estos motivos, creemos que la Administración Pública Nacional debería apostar a la aplicación e impulso de las tecnologías, licencias y estándares libres.

Teniendo en cuenta esta situación y en pos de contribuir con la afirmación anterior, la intención final de la presente tesis es el desarrollo de una solución de *software* libre que resuelva procesos de negocio de importancia en el ámbito municipal. Para llegar a este objetivo final, se realizó un estudio del estado del arte a nivel mundial, analizando distintas soluciones que abarcan las siguientes áreas dentro de la Administración Pública: ofimática, procesos de negocio internos y comunicación entre el Estado y sus ciudadanos. Como resultado de dicho estudio, se desprende la importancia que se le está dando a este tipo de soluciones en otros países y la necesidad de implementar este fenómeno mundial a nivel local.

# Bibliografía

1. http://www.vialibre.org.ar/2002/09/29/razones_por_las_que_el_estado_debe_usar_software_libre/
2. http://www.microsoft.com/argentina/prensa/2007/feb/translator
3. http://www.gobiernodigital.org.ar/texto.asp?are=18&idf=101
4. http://www.gobiernodigital.org.ar/texto.asp?are=18&idf=169
5. http://www.hispalinux.es/node/596
6. http://www.plonegov.org.
7. "Requisitos del *software* para su uso en el Estado". http://docs.hipatia.net/dsl/requisitos.html

# Estado del arte

A pesar de que el fenómeno *open source* ha estado creciendo en importancia a nivel mundial desde hace ya algunos años, en el ámbito gubernamental, este movimiento ha tomado impulso recién en este último tiempo. Al abordar esta investigación, encontramos proyectos *open source* que han sido o están siendo implementados con relevancia mundial en la administración pública. Entre los proyectos estudiados, creemos que vale la pena destacar los siguientes:

- OpenDocument Format
- PloneGov
- Open Cities

## OpenDocument Format

Cuando comenzamos a investigar sobre este tema, encontramos un sitio *web* dedicado a la promoción, uso y desarrollo del formato OpenDocument perteneciente a una organización voluntaria con miembros de todas partes del mundo denominada *OpenDocument Fellowship*.

En este sitio, se plantea que si uno tiene problemas para leer un documento que le enviaron o compra una determinada versión de un producto (por ejemplo, Microsoft Office) y solo puede leer documentos de dicha versión, está en presencia de lo que se denomina dependencia tecnológica (*vendor lock-in*)[1], en la cual el proveedor escribe (codifica) los documentos a través de un formato secreto que solamente él conoce y puede leer (decodificar). Como podemos leer dichos documentos únicamente con su *software*, esto inhabilita a sus competidores para leer y escribir tales archivos, por lo que, en definitiva, el cliente tiene solamente una posibilidad de elección. La dependencia tecnológica es la enemiga de la competencia.

El sitio presenta OpenDocument Format (ODF) para salir de esta dependencia en lo que respecta a documentos de oficina.[1]

El formato ODF es un estándar abierto[2] basado en XML, definido para documentos de texto, hojas de cálculo, presentaciones y dibujos, entre otros, disponible libremente para que cualquier proveedor pueda utilizarlo. Fue desarrollado por la organización OASIS (*Organization for the Advancement of Structured Information Standards*), un consorcio internacional sin fines de lucro dedicado a la creación y difusión de estándares.

OpenDocument fue aprobado como estándar OASIS en mayo de 2005. Un año más tarde, la Organización Internacional para la Estandarización (ISO) y la Comisión Electrotécnica Internacional (IEC) aprobaron la propuesta de estándar internacional

1 *Vendor lock-in*: término utilizado en economía para describir la situación que se presenta cuando un cliente depende de un único proveedor de productos o servicios y no puede cambiar de proveedor sin que esto signifique un costo significativo.

2 Estándar abierto: especificación disponible públicamente para lograr una tarea específica.

ISO/IEC 26300, denominada "*Open Document Format for Office Applications* (OpenDocument) v1.0".

A partir del formato ODF, se definen los siguientes tipos de archivos:[5]

| Tipo de archivo | Extensión | Tipo MIME |
|---|---|---|
| Texto | .odt | application/vnd.oasis.opendocument.text |
| Hoja de cálculo | .ods | application/vnd.oasis.opendocument.spreadsheet |
| Presentación | .odp | application/vnd.oasis.opendocument.presentation |
| Dibujo | .odg | application/vnd.oasis.opendocument.graphics |
| Gráfica | .odc | application/vnd.oasis.opendocument.chart |
| Fórmula matemática | .odf | application/vnd.oasis.opendocument.formula |
| Base de datos | .odb | application/vnd.oasis.opendocument.database |
| Imagen | .odi | application/vnd.oasis.opendocument.image |
| Documento maestro | .odm | application/vnd.oasis.opendocument.text-master |

Tabla 1. Documentos

| Tipo de archivo | Extensión | Tipo MIME |
|---|---|---|
| Texto | .ott | application/vnd.oasis.opendocument.text-template |
| Hoja de cálculo | .ots | application/vnd.oasis.opendocument.spreadsheet-template |
| Presentación | .otp | application/vnd.oasis.opendocument.presentation-template |
| Dibujo | .otg | application/vnd.oasis.opendocument.graphics-template |

Tabla 2. Plantillas

Las consecuencias de este nuevo estándar oficial, como se detallan en una nota publicada en el sitio Software Libre Chile[3] serán muy importantes.

La principal consecuencia enunciada es que cualquier ciudadano podrá enviar documentación oficial a su administración pública en el nuevo formato. Luego, esas administraciones podrán leer ese formato para responder adecuadamente a la comunicación oficial de su administrado. El hecho de que este formato sea oficial, internacional y de derecho implica que, en entornos públicos, ya no podrá obligarse a los ciudadanos a remitir su documentación en formatos cerrados y propietarios.

Por otra parte, no se le podrá exigir al ciudadano que adquiera un producto de una determinada empresa a la hora de leer un documento emitido por una administración pública. Por el contrario, podrá exigirse a esa administración que emita dicho documento en formato ISO 19005 (PDF/A) o en formato ISO 26300 (OASIS OpenDocument). Algunas de las implementaciones de este formato que se encuentran actualmente en producción son el StarOffice de Sun Microsystems y el Workplace Managed Client de IBM, ambas comerciales, así como por el paquete OpenOffice y el LibreOffice, ambos *software* libre y

gratuito. Además, múltiples desarrolladores lo están incorporando también o han anunciado que lo harán próximamente, y es de esperar que, ante la aprobación de ISO/IEC, se acelere el proceso de su incorporación a cada vez más aplicaciones.

En la nota citada, se estima que, ante esta exigencia por parte del ciudadano para con las administraciones públicas, se deberán instalar aplicaciones informáticas que soporten los dos estándares anteriormente mencionados. En el caso de ISO 19005, las administraciones lo tienen relativamente solucionado, pues la aplicación más extendida, Acrobat Reader, es capaz de leer el PDF/A. También lo hacen aplicaciones alternativas como Xpdf y Evince, entre otras. Pero en el caso del nuevo ISO 26300 se presentan dificultades, pues el fabricante de la aplicación ofimática más difundida, Microsoft con su Office, posee su propio formato denominado Office Open XML. Aunque este formato se encuentra en proceso de estandarización, hasta el momento no ha logrado alcanzar el 75% de votos a favor que requiere la ISO/IEC para aprobarlo como estándar internacional.[2]

Sin embargo, a mediados de 2006, Microsoft dio a conocer su apoyo al proyecto de código abierto Open XML Translator, orientado a permitir la conversión de documentos en formato OOXML al formato ODF y viceversa. La versión 1.0, publicada en febrero de 2007, soporta los actuales formatos de documentos con estándares de la industria tanto de Open XML como de ODF. Asimismo, ha sido evaluada en Microsoft Office 2007, Office 2003 y Office XP. Novell anunció que Translator se implementará de forma nativa en su próxima versión de OpenOffice.

La segunda etapa del proyecto Translator, incluyendo los traductores Spreadsheet (Microsoft Office Excel) y Presentation (Microsoft Office PowerPoint) comenzó en febrero de 2007 y las primeras versiones ya están disponibles en SourceForge.[4]

El proyecto continuará siendo de código abierto y su desarrollo seguirá a cargo de SourceForge, además de que estará disponible sin costo alguno para todos los clientes, ya sea para su desarrollo o uso.

Sin lugar a dudas, este complemento facilitará la adopción del nuevo estándar por aquellas administraciones públicas que ya han decidido hacerlo como norma. También facilitará enormemente las migraciones a las aplicaciones que sí siguen la normativa estándar, puesto que permite exportar (traducir) los documentos de los formatos de extensión .doc, .xls y .ppt de la popular suite de Microsoft al formato OpenDocument oficial.

Por lo mencionado anteriormente y teniendo en cuenta las facilidades con las que se cuenta para su adopción, creemos que es una buena solución que los gobiernos distribuyan la información en este formato. De esta manera, podrán garantizar el derecho que poseen los ciudadanos a la plena accesibilidad presente y futura de los documentos públicos y así eliminar cualquier tipo de dependencia tecnológica, que beneficia de manera arbitraria y monopólica a un determinado proveedor, mientras que obliga al

ciudadano a adquirir la licencia del *software* y depender de su continuidad para el acceso futuro a dichos datos.

Según datos publicados por OASIS, algunas de las organizaciones gubernamentales que están adoptando OpenDocument son:

- el Ministerio de Defensa de Singapur;
- los Ministerios de Hacienda, Economía e Industria de Francia;
- el Mini sterio de Salud de Brasil;
- la ciudad de Múnich en Alemania;
- el Concejo de la Ciudad de Bristol, del Reino Unido, y
- la ciudad de Viena, en Austria.

# PloneGov

## Introducción

El proyecto PloneGov es una iniciativa de *software* libre basada en Zope y Plone, que tiene como objetivo directo proveer soluciones IT para las e-administraciones de los gobiernos locales y regionales de todo el mundo. La idea central del proyecto es el desarrollo colaborativo de soluciones y, puesto que la mayoría de los municipios no posee la infraestructura necesaria para desarrollar y financiar proyectos de e-administración aunque sus necesidades sean muy similares, creemos que son especiales candidatos para beneficiarse del proyecto. De esta manera, las administraciones locales lograrían acceder a aplicaciones consistentes en un tiempo relativamente corto y a muy bajo costo para su propio uso y el de sus ciudadanos, algo que de otro modo sería imposible, además de ganar en independencia de proveedores de servicios de IT y demás ventajas asociadas al uso del *software* libre.[6]

## PloneGov: Plone aplicado a entornos gubernamentales

El proyecto PloneGov es una fusión de iniciativas europeas que ya estaban en marcha para producir herramientas para e-gobierno sobre Plone, y tiene como objetivo compartir experiencias y desarrollar una plataforma de *software* libre adaptada a las necesidades de las administraciones locales.

Plone es un administrador de contenido (CMS) *open source*, licenciado bajo la *General Public License* (GPL), una licencia común de código abierto que permite el uso de los archivos fuentes libremente. Está implementado en el lenguaje de programación Python y corre sobre el servidor de aplicaciones Zope. [7]

Plone puede ser empleado como una aplicación prefabricada, lo cual permite un óptimo nivel de reusabilidad, facilitando el desarrollo de aplicaciones complejas en muy poco tiempo. Es utilizado para construir portales, sitios *web* corporativos, sitio de noticias, servidores de *extranet* o *intranet*, sistemas de publicación y repositorio de documentos, herramientas *groupware*, e-comercio, etc.

A continuación, describiremos las principales características de un sitio Plone básico, para demostrar qué aspectos de la problemática de las administraciones públicas podrían ser resueltos a través de su utilización.

## Elementos de Plone y características técnicas[7], [8]

De base, un sitio Plone posee los siguientes elementos:

- diferentes tipos de contenido;
- herramientas para administración de usuarios y roles;
- *workflows*;
- *layouts* y plantillas predefinidos y personalizables;
- una interfaz de administración;
- hojas de estilo (*style sheets*);
- buscador en tiempo real;
- soporte multilenguaje, y
- políticas de seguridad.

Por defecto, la página principal de un sitio construido con Plone posee un encabezado, un pie de página y un área de contenido dividida en tres columnas: izquierda, centro y derecha.

El encabezado posee una imagen con un logotipo, enlaces a páginas con información acerca de estándares, un mapa del sitio generado automáticamente, un formulario de contacto y una herramienta de búsqueda dentro del sitio.

La columna del medio puede contener distintos tipos de información: aquí es donde se concentra la mayor interacción con el usuario, donde se dispone de la mayor cantidad de funciones para agregar, editar y visualizar contenido específico.

El pie de página, por lo general, dispone de información sobre Plone, autores, estándares cumplidos, etc.

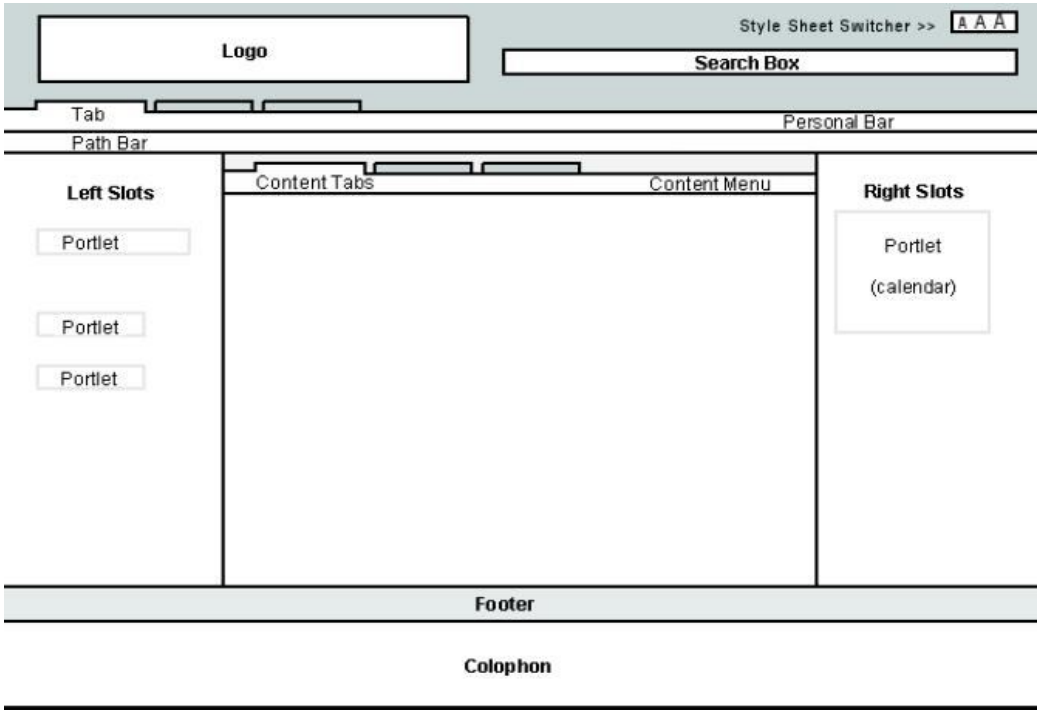



Figura 1. Estructura de un sitio Plone

## Tipos de contenido

Los tipos de contenido básicos provistos por Plone son los siguientes:

- **Documento**: presenta información estática al usuario. Es el tipo de contenido más común y es similar a las típicas páginas *web*.
- **Ítem de noticia**: es un documento que muestra campos especiales (como Fecha) y se despliega automáticamente dentro de la pestaña de noticias del sitio.
- **Enlace**: es un vínculo a una URL, que posee atributos como título, descripción y URL, y puede ser interno o externo.
- **Imagen**: es un objeto que contiene imágenes en formato digital. Los formatos de dichos archivos pueden ser los comunes, por ejemplo, GIF o JPG.
- **Evento**: contiene información sobre eventos a realizarse tales como reuniones, charlas, exposiciones y conferencias.
- **Carpeta**: es similar a las carpetas del Filesystem, es decir, puede almacenar otros tipos de contenido y provee mecanismos para organizarlos.
- **Archivo**: permite almacenar contenidos como películas, sonido, documentos de texto, hojas de cálculo, archivos comprimidos o cualquier otro tipo de archivo que se pretenda subir a un sitio Plone.

- **Carpeta inteligente** (*smart folder*): es similar a la carpeta pero se diferencia de esta en cuanto a que, en lugar de guardar contenido en su interior, muestra el resultado de una búsqueda cuyo criterio está previamente definido.

## *Portlets*

Son *frames* o cajas que pueden mostrar información dinámicamente o bien tener una función muy específica. Dentro de la estructura de un sitio Plone, se encuentran en las columnas de la izquierda y de la derecha.

Los *portlets* básicos predefinidos que contiene un sitio Plone son los siguientes:

- **Calendario**: muestra el almanaque del mes en curso con el día actual resaltado o seleccionado de manera especial. Además, se permite la navegación por meses hacia delante o atrás. También brinda la posibilidad de marcar aquellas fechas para las cuales figuran eventos programados.
- **Eventos**: en este *portlet* o *frame* se encuentran los eventos programados. Cuando un usuario del sitio crea un nuevo evento, aparece en el *portlet* de eventos, que puede ser configurado para mostrar una cantidad cualquiera de eventos próximos, o bien los eventos programados en un intervalo de tiempo.
- **Identificación**: este *frame* aparece si el usuario que visita el sitio no se ha indentificado. Contiene dos campos para permitir ingresar el nombre de usuario y la contraseña y su correspondiente botón de *submit* o aceptación. Además, posee enlaces a otros formularios para crear una nueva cuenta de usuario o bien acceder a la función correspondiente al olvido de la contraseña.
- **Navegación**: muestra un árbol del estilo del explorador para la estructura de directorio del sistema de archivos que contiene las carpetas con las secciones dentro del sitio. Cuando se selecciona una carpeta, se muestra su contenido.
- **Noticias**: es muy similar al *portlet* de eventos: muestra las noticias que cumplen con un cierto criterio predefinido.

Estos *portlets* básicos pueden ser personalizados e incluso se brinda la posibilidad de crear nuevos.

Figura 2. Ejemplos de *portlets*

# Contenido del sitio

El acceso a los contenidos de un sitio Plone, en conjunto con la creación y modificación de contenidos propios dentro de él, se lleva a cabo a través de la combinación del rol del usuario más el estado del contenido, donde este último será uno de los estados posibles previamente definidos en el *workflow* del sitio. De este modo, un usuario solo tendrá acceso a los contenidos que se encuentren permitidos para su rol y en un estado determinado, por ejemplo, publicado.

## *Roles de usuario*

En Plone, existen dos tipos de roles, **local** y **global**. Un rol **global** es válido en todo el sitio y permite al usuario tener acceso a cualquier contenido dentro de él; por el contrario, un rol local solo tiene acceso a ciertos contenidos. Cada usuario puede tener más de un rol, y debe tener asignado al menos uno. Por lo tanto, cada usuario posee, por defecto, el rol estándar de miembro.

### Rol *manager*

Es un rol global y el más alto que puede obtener un usuario. Este rol cuenta con privilegios absolutos dentro de Plone y, por lo tanto, dispone de los niveles de acceso suficientes para realizar tareas como las siguientes:

- **Administración de usuarios**: puede agregar y modificar usuarios.
- **Administración de roles de usuario**: puede agregar o quitar roles locales y asignarlos o quitarlos a los usuarios.
- **Administración de contenidos**: tiene privilegios suficientes para agregar, modificar, publicar o rechazar cualquier tipo de contenido dentro del sitio.
- **Administración de plantillas y otros elementos**: puede modificar plantillas y otros elementos de la instalación de Plone.

### Rol miembro

Es el rol básico, el que se le asigna por defecto a todo usuario que se une a un sitio Plone. Dado que estos sitios están orientados al concepto de comunidad, es muy fácil suscribirse, y cuando un nuevo usuario se une, adquiere el rol miembro, a través del cual podrá tener su espacio dentro del sitio además de crear, modificar y administrar sus contenidos propios. Luego, un usuario con rol *manager* podrá otorgarle roles adicionales a estos usuarios con rol miembro simple, para que puedan acceder a otros contenidos.

## *Workflow*

El *workflow* es la herramienta que nos permite administrar el contenido dentro del sitio Plone. Inicialmente se dispone de dos *workflows*: uno propio para el manejo de las carpetas

y el *workflow* por defecto para la administración del resto del contenido. Este último establece los **estados** y las **transiciones** entre estados para cada tipo de contenido definido en el sitio.

El estado es la información acerca de un contenido en particular en un momento dado. Ejemplos de estados son: público, privado, pendiente, rechazado y publicado, entre otros. La transición es la acción mediante la cual un contenido cambia de un estado a otro.

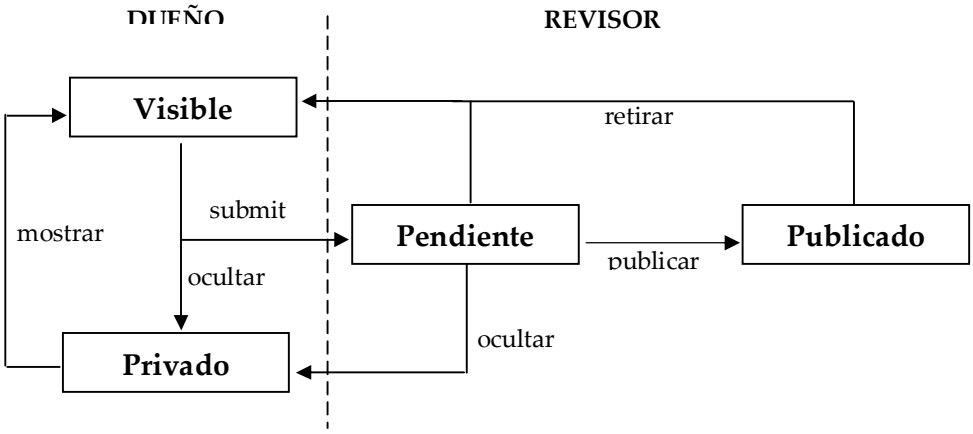



Figura 3. *Workflow* por defecto

## *Estados y sus transiciones*

- Cuando un usuario agrega contenido en su espacio de trabajo, ese usuario es el **dueño** del contenido.
- Al tomar la acción de *submit*, pasa el contenido al estado **pendiente**, a la espera de que un miembro/usuario con rol de revisor/supervisor lo publique.
- Cuando el usuario supervisor toma la acción de **publicar** el contenido, lo hace visible al resto de los usuarios, y dicha acción pasa el contenido al estado **publicado**.
- Si, por el contrario, un supervisor decide que el contenido no debe ser publicado, toma la acción de **retirar** el contenido en cuestión y este pasa al estado **visible** solo para su dueño.
- Si se decide **ocultar** un contenido, queda en estado **privado** y solo su dueño puede acceder a él. Dicho contenido es removido del índice de búsqueda para que no aparezca como parte del resultado en las sucesivas consultas.

# Proyectos PloneGov

Hasta aquí hemos visto de qué se trata Plone, es hora entonces de mostrar ejemplos que den testimonio de lo que es posible hacer con esta potente herramienta. En esta parte del documento, describiremos algunos de los proyectos que forman parte de la iniciativa PloneGov y han sido o están siendo desarrollados alrededor del mundo por la comunidad

Plone, así como también implementaciones exitosas de sitios construidos con Plone en el ámbito gubernamental.

## *PuRe (Public Requeriments) [9]*

PuRe es un sistema que, como indica su nombre, permite crear y administrar licitaciones públicas.

Actualmente, soporta licitaciones de servicios y está específicamente dirigido a aquellas relacionadas con el desarrollo o adquisición de productos de *software*, aunque puede ser utilizada para cualquier otro tipo de licitaciones.

La herramienta permite generar un documento ODF, DOC o PDF con el pliego de la licitación gracias al uso de la librería POD (*Python Open Document*).

Actualmente, la herramienta está disponible solamente en francés y el código fuente está escrito en inglés.

PuRe considera una adquisición pública como un conjunto de requisitos. Estos requisitos son agrupados por pedidos. De hecho, cuando una administración publica las bases para una adquisición pública, describe el conjunto de resultados que se esperan en el formulario de pedidos y por cada uno de estos expresa claramente el conjunto de requisitos. Un pedido puede ser un documento, código fuente, pruebas o cosas menos tangibles como, por ejemplo, entrenamiento sobre una tarea específica o una cierta cantidad de horas de soporte.

## *CPComarquage – (Etiquetado de documentos compartidos) [10]*

CPComarquage es una herramienta para el etiquetado de documentos con contenido RSS. Permite agregar etiquetas a los documentos con datos generados por servidores. Lo interesante de etiquetar un documento de esta manera es el hecho de que los datos etiquetados serán actualizados externamente por otros usuarios. De esta manera, el sitio se actualizará "auto-mágicamente". Este producto podría ser adaptado para analizar (*parse*) datos RSS y mostrarlos en algún otro documento.

## *POD (Python Open Document) [11]*

POD permite generar dinámicamente y desde cualquier programa Python o sitio Plone o Zope documentos de extensión .odt, .pdf o .rtf a través de OpenOffice.org más el agregado de algunas expresiones Python.

La primera versión de POD es un *beta release* lanzado en junio de 2007.

POD es una librería que permite generar fácilmente documentos de contenido dinámico. El principio es simple: se crea un documento de texto ODF (*Open Document Format*) a través de alguna herramienta como el OpenOffice Writer 2.0 o superior, se agrega algo de código Python en ciertos lugares del documento, y desde algún programa escrito en Python se puede llamar a POD con el archivo OpenDocument y un grupo de

objetos Python como entrada. POD genera otro documento de texto ODF (.odt) que contiene el resultado deseado. Si se prefiere tomar el resultado en otro formato, POD puede llamar a OpenOffice en modo servidor para generar el resultado con extensión .odt, .pdf o .rtf.

## *Plone como* extranet

### CommunesPlone College [12]

Este es un producto Plone desarrollado por el proyecto CommunesPlone para el manejo de actas. Básicamente, el producto permite las siguientes funciones:

- Generar y organizar la orden del día de las asambleas del colegio.
- Gestionar las resoluciones de los temas que se tratan en las asambleas.
- Generar un archivo de formato PDF con el acta de la asamblea.

#### Breve descripción del funcionamiento

- Un agente comunal crea un apunte que contiene los temas a discutir en la asamblea. Luego, hace que ese apunte pase del estado **privado** al estado **pendiente**. Ahora el apunte puede ser visto por la secretaria comunal, que eventualmente puede completar el apunte dejando una descripción con indicaciones.
- La secretaria comunal crea una orden del día con la fecha de la asamblea del colegio comunal. También puede atender aquellos apuntes que se encuentren aún en estado pendiente.
- Se puede generar e imprimir un PDF con la orden del día, que servirá a los miembros de la asamblea durante su desarrollo.
- La orden del día, cuya acta se genera automáticamente, puede ser modificada durante la asamblea. Por esta razón, la aplicación ofrece la posibilidad de incluir los puntos complementarios de la asamblea en su acta.
- La secretaria puede editar y completar el acta (los puntos correspondientes se modifican automáticamente con el contenido del acta).
- La secretaria finalmente cierra el acta luego de su aprobación, lo que genera un archivo PDF que la contiene.

#### Roles predefinidos

En la descripción anterior, se mostraron diferentes tipos o perfiles de uso de las funciones detalladas. Dichos perfiles se conocen con el nombre de **roles**. Un rol representa un conjunto de acciones que un usuario puede llevar a cabo (tiene asociados ciertos permisos), por ejemplo, solo el rol **secretaria** puede crear una orden del día.

Los siguientes roles están presentes en la aplicación:

- **Agente comunal**, que puede crear un tema o punto a tratar.
- **Secretaria comunal**, que genera y administra los puntos o temas, órdenes del día y actas.
- **Mandatario**, que puede crear un punto o tema y visualizar las órdenes del día y las actas.

Es posible adaptar los permisos asociados a cada rol a través de la modificación del *workflow* del producto.

### Utilización del producto

Las etapas del uso estándar del producto College son:

- Creación de un tema para tratar en la orden del día.
- Propuesta del tema.
- Creación de la orden del día.
- Tratamiento de los temas en estado pendiente.
- Listado y análisis de los puntos propuestos y aún no incorporados a la orden del día.
- Descripción de un tema para el acta (la secretaria puede editar un tema y agregar comentarios sobre él).
- Cierre de la orden del día y creación de la correspondiente acta.
- Agregado de los temas discutidos en la asamblea que no figuraban dentro de los temas a tratar en la orden del día al acta correspondiente.
- Decisión sobre los temas (la secretaria puede indicar las decisiones que se tomaron sobre cada uno de los temas tratados en la asamblea).
- Cierre final del acta una vez aprobada por el colegio comunal.
- Generación del archivo PDF que contiene el acta de la asamblea.

## POI para manejo de expedientes

### Issue Tracker

POI es un administrador de **tareas** o **asuntos** (*issues*) para Plone. Dentro de un equipo de desarrollo de *software*, un *issue tracker* permite a sus miembros hacer seguimiento de tareas, pedidos, mejoras o cualquier cosa que tenga que ver con el circuito de un proyecto de *software*. La herramienta permite consultar y listar *issues*, organizarlos por áreas, ver sus estados (**abierto**, **resuelto**, **cancelado**), agregarlos y modificarlos.

La página de POI permite navegar los *issues* por estado o área, entre otros, así como también realizar búsquedas de *issues*.

Los atributos de un *issue* son los siguientes:

- título;
- descripción;
- área a la que va dirigido, y
- responsable.

Opcionalmente, se pueden adjuntar archivos.

Una vez agregado un *issue*, se lo puede aprobar o rechazar, y se pueden documentar cambios, agregar nuevos archivos y cambiar al responsable asignado. Los *issues* resueltos se pasan al estado **cerrado**.

La herramienta permite configurar la notificación vía correo electrónico, para reportar los cambios en los *issues* por ese medio. Esta utilidad es muy importante, ya que permite que los miembros de un equipo de trabajo estén notificados de lo que pasa en el proyecto.

Para adaptar POI al manejo de expedientes en un entorno gubernamental, alcanzaría con definir las áreas que conforman la organización. Luego, cada expediente sería un *issue* que se pueda seguir como si se tratara de una tarea dentro de un equipo de trabajo.

## Open eGov, una iniciativa de *software* colaborativa unida a PloneGov[13]

Open eGov es un proyecto lanzado por la ciudad de Newport News, Virginia, Estados Unidos. La idea es crear un ecosistema de *software* colaborativo, donde las organizaciones gubernamentales, las organizaciones sin fines de lucro y las del sector privado trabajen juntas para compartir los costos en pos de aumentar las capacidades de desarrollo.

En agosto de 2007, Open eGov se unió a otras 55 organizaciones gubernamentales de Europa, África y América del Sur y se fusionó con el proyecto PloneGov. La iniciativa tiene como objetivo ofrecer todo el *software* y la documentación relacionada a través de los canales de distribución de PloneGov.

### Características del proyecto

- Este proyecto intenta proporcionar una barrera baja para el acceso de las organizaciones a soluciones informáticas. Los propulsores del proyecto consideran que debería ser simple y poco costoso para las organizaciones acceder al nivel que se ajuste a sus necesidades y que la misma eficiencia debería aplicarse a lo largo del ciclo de vida del producto: período de prueba, puesta en producción, mantenimiento, soporte y mejoras.
- Está disponible para su uso sobre plataformas Linux, Windows y Mac.
- Empaqueta un conjunto de "productos Plone" bajo la licencia *open source* GPL (*General Public License*). Open eGov ha seleccionado e integrado más del 20% de los productos disponibles para Plone, además de otras herramientas y utilitarios *open source* complementarios.

## Open Cities

El proyecto Open Cities define e implementa una plataforma de interoperabilidad administrativa para soportar la gestión de expedientes y el control del flujo de actividades dentro de una administración pública, interrelacionando al ciudadano, al empleado público y todos los sistemas y aplicaciones que conforman el *back office* de la entidad. Hace un uso intensivo de tecnologías *open source* e integra en la plataforma de interoperabilidad diferentes componentes vinculados a tecnologías de *workflow* (Bonita), gestión documental (Fedora), XForms (Orbeon) y orquestación de servicios (Intalio BPMS).

El sitio open-cities.org ofrece una reseña acerca del surgimiento de este proyecto. En ella, se afirma que fue una iniciativa conjunta del Ministerio de Industria, Turismo y Comercio, la Universidad Politécnica de Madrid y la empresa Ándago Ingeniería, con el compromiso del uso intensivo de estándares internacionales como los siguientes:

- XPDL (*XML Process Definition Language*): lenguaje para la definición de flujos de trabajo.
- BPEL (*Business Process Execution Language*): lenguaje basado en XML, diseñado para el control centralizado de la invocación de diferentes servicios *web*, con cierta lógica de negocio añadida que ayuda a la programación en gran escala.
- XADES (*XML Advanced Electronic Signatures*): formato que permite incorporar información a la firma básica definida por el *World Wide Web Consortium* (W3C): *time stamps*, certificados de atributos, indicaciones de responsabilidades asumidas al firmar, etc.
- XML (*eXtensible Markup Language*): metalenguaje extensible de etiquetas desarrollado por el W3C.

En este proyecto, se define como objetivo general dotar a las administraciones públicas de un conjunto de herramientas basadas en *software* de fuente abierta, que les permitan establecer relaciones con los ciudadanos de forma óptima, e identificar y solucionar las carencias en la prestación de los servicio públicos.

Guillermo Pastor, gerente de soluciones de Ándago Ingeniería, afirma que este proceso de liberación de aplicaciones en el ámbito de la administración pública es uno de los principales objetivos de este proyecto, que intenta cubrir la necesidad de la apertura del mercado de la administración electrónica, fundamentando su propuesta en el desarrollo de una alternativa basada en el empleo masivo de componentes *open source*, siempre demostrando la viabilidad y validez de las tecnologías y su estricto cumplimiento de los estándares y normativas nacionales y comunitarios.

La liberación de aplicaciones nos pareció una característica muy importante, ya que vemos que, actualmente, en el ámbito gubernamental de nuestro país, se utilizan herramientas *open source* para el desarrollo de sistemas relativos a sus procesos internos,

pero no tenemos conocimiento de que se hayan liberado sistemas en el ámbito de la administración pública.

Esta liberación sería de mucha utilidad por el hecho de que muchos de los procesos internos en los diferentes municipios son muy similares entre sí, y cada mejora o nuevo proceso se podría compartir, lo que resultaría en una disminución significativa de costos de tiempo y dinero, además de mejorar la calidad de los procesos.

Al implementar esta metodología de trabajo, los municipios que anualmente gastan grandes sumas de dinero en licencias de *software* podrían independizarse tecnológicamente de estos proveedores cuando otro municipio liberara un *software* de las mismas características. Además, los municipios que no cuentan con presupuesto o infraestructura suficiente como para alcanzar determinados programas podrían utilizarlos a través de la liberación.

## Descripción técnica del proyecto

Esta descripción está basada íntegramente en un *paper* publicado por el propio Guillermo Pastor[14], que detalla las siguientes características del proyecto:

El alcance de este proyecto se centra en la definición de una plataforma tecnológica que proporcione un entorno base para una arquitectura de administración electrónica, orientada a servicios y centrada en los procesos y documentos, capaz de soportar la demanda de manera satisfactoria y ordenada, y adecuando su operativa a la normativa legal vigente.

La plataforma se centra en la creación de los elementos de soporte a tres grupos de actividades que deberán ser integradas y unificadas para poder obtener toda la potencialidad de la solución:

- certificación digital y firma electrónica;
- gestión de la seguridad y control de acceso, y
- gestión ordenada de los procesos y flujos de trabajo.

## *Framework* de desarrollo y *middleware* de integración

Open Cities es una plataforma tecnológica que proporciona la infraestructura básica y esencial para la definición, formalización y automatización de los procesos administrativos, dándoles soporte informático vía *web* e integrándolos con los sistemas de *back office* habituales de la corporación (padrón, catastro, ERP, GIS, etc.).

Tal y como aparece en la ilustración, existen distintos niveles y capas:

- La plataforma tecnológica ó *middleware* que proporciona los servicios esenciales. Los servicios básicos inicialmente definidos son: firma y certificación electrónica, identificación, mecanismo *single-sign-on*, gestión de procesos y custodia documental.

- El nivel de servicio, constituido por los diferentes subsistemas y procesos de negocio definidos por cada organización, por ejemplo la presentación de una solicitud o documentación, etc.

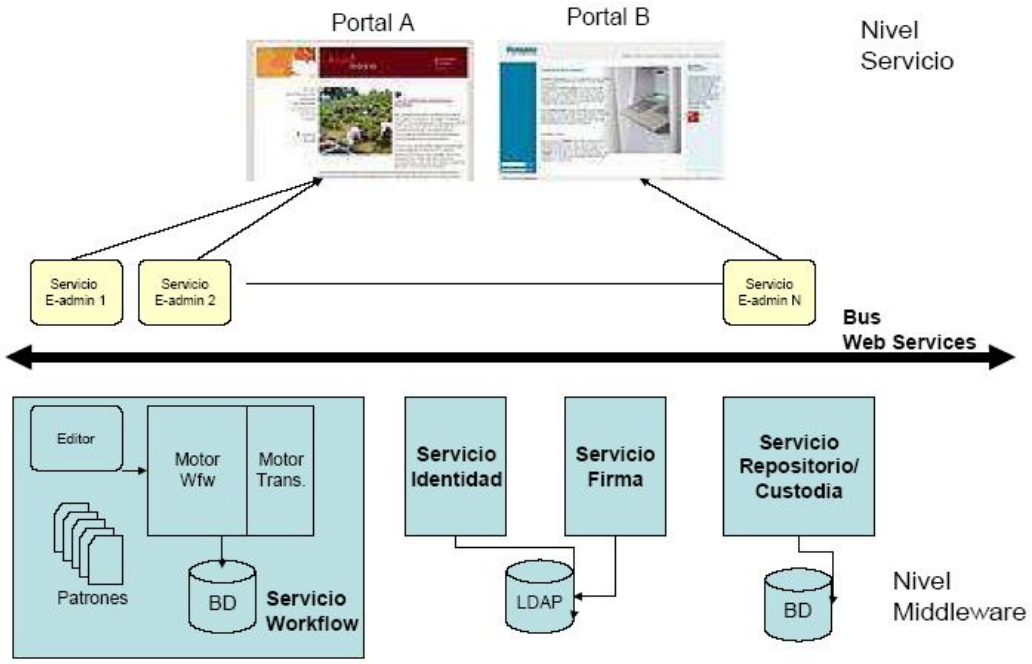



Figura 4. Plataforma Open Cities – Visión global

Esta arquitectura funcional implica niveles de escalabilidad elevados, ya que cualquier otro módulo funcional adicional podrá ser implantado en el futuro sobre la arquitectura definida.

Open Cities es un mediador y por lo tanto, facilita la interconexión del *back office* de la organización con los servicios de propósito general que ofrece al ciudadano, empleado público, etc.

Propone una infraestructura de administración electrónica basada en:

- automatización de los procesos administrativos;
- generalización del uso de certificados y firma digital;
- servicios de custodia de los expedientes;
- integración de tecnologías bajo una única plataforma, y
- seguimiento de los estándares disponibles.

## Tecnología de la solución

Open Cities es una plataforma tecnológica de interoperabilidad sustentada en una SOA (*Service Oriented Architecture*). Los servicios que se incorporen a la plataforma, a través de mecanismos de orquestación, podrán ser empleados en diferentes procesos

administrativos propios o de otras organizaciones, y, además, podrán ser intercambiables, es decir que cada uno de los componentes que inicialmente integran Open Cities pueden ser sustituidos por otros equivalentes, que, por ejemplo, ya existan en el organismo, siempre y cuando cumplan determinados requisitos funcionales.

Desde el punto de vista arquitectónico y tecnológico, Open Cities posee las siguientes características:

- es multiplataforma, multiusuario, y multiproceso;
- fue desarrollado en Java y es compatible con especificaciones J2EE;
- posee una Arquitectura Orientada a Servicios (SOA);
- incluye un *bus* de servicios *web* (SOAP);
- permite el intercambio de datos XML;
- permite la gestión de flujos de trabajo transaccional;

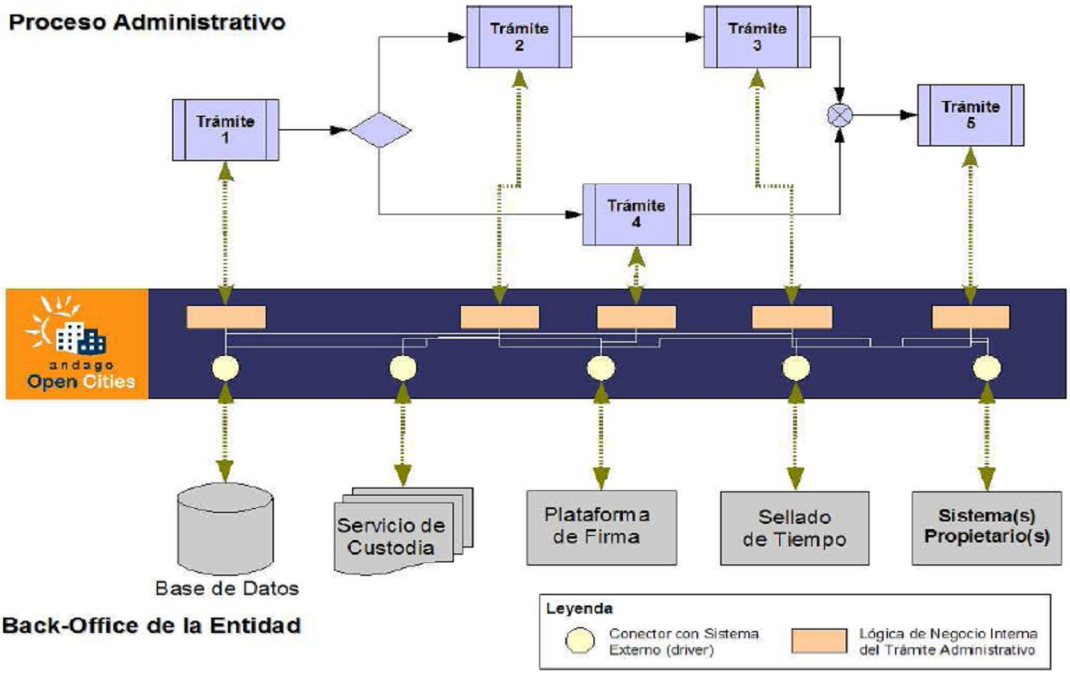



Figura 5. Integración de la gestión de expedientes

- emplea XADES como estándar de firma digital;
- opcionalmente, puede incluir una entidad de certificación (CA), y
- emplea componentes de "*software* libre" probados y fiables.

## Plataforma de administración electrónica

La administración electrónica debe soportar y sustentar procesos administrativos que manejan documentos y expedientes. Sin embargo, existe una barrera tecnológica entre los sistemas *back office* de una corporación y los procesos soportados por el e-gobierno.

En cada implementación, se deben definir los procesos soportados así como los conectores con los sistemas de *back office* específicos. De esta forma, Open Cities parte de unos servicios base (firma, custodia, sellado de tiempo, base de datos, mensajería) y, a través de una arquitectura abierta, basada en servicios *web*, es independiente de las aplicaciones disponibles y, por lo tanto, permite la personalización de nuevos servicios e incluso el intercambio de los ya existentes, gracias a la definición de conectores específicos.

## Arquitectura de tres capas

Open Cities está basado en una arquitectura de *software* de tres capas, como se muestra en la figura 6.

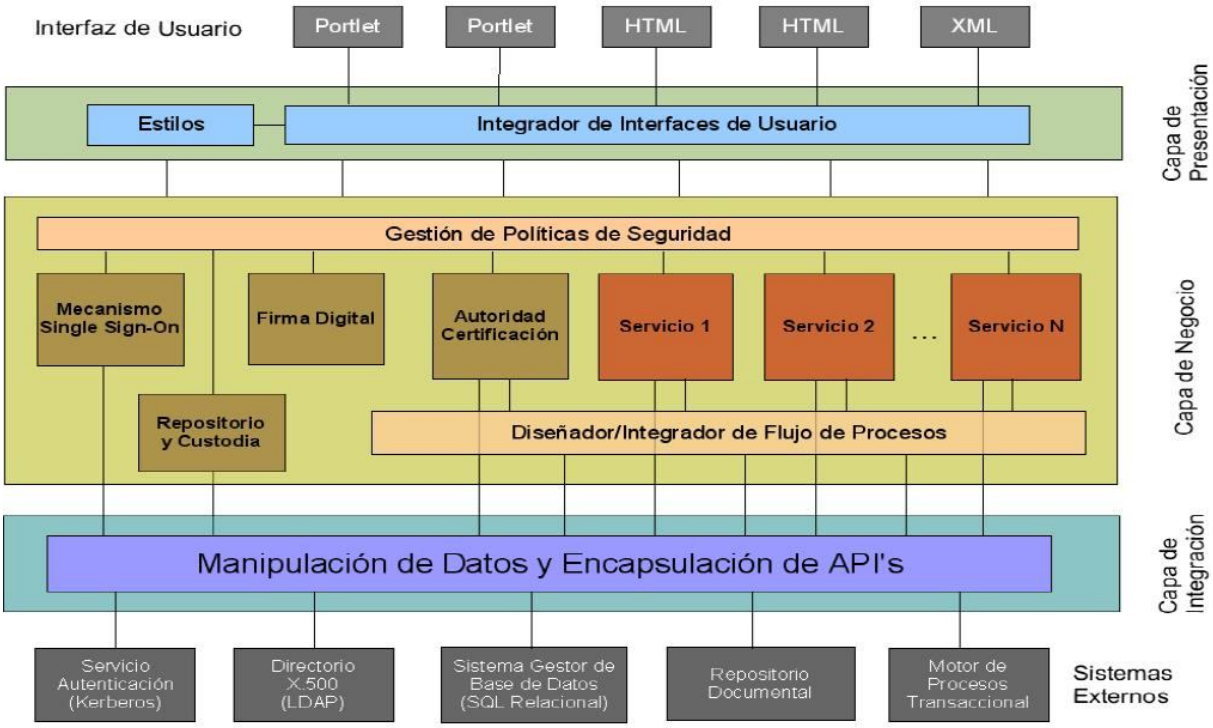



Figura 6. Arquitectura de capas del sistema Open Cities

Por un lado, estas tres capas arquitectónicas posibilitan la interacción con sistemas y actores ya existentes ajenos al proyecto y con los que se integrará y, por otro lado, encapsulan las reglas de negocio propias a los servicios de Administración Electrónica de la entidad que incorpore esta tecnología.

- **Capa de presentación**: encargada de unificar los criterios de visualización de los servicios ofrecidos al usuario. Está construida utilizando un sistema de vista/controlador.
- **Capa de negocio**: esta capa constituye la parte esencial de la plataforma de administración electrónica, dado que encapsula las reglas de negocio, descripción de procesos y políticas propias a la operativa tanto de la plataforma como de los servicios que ésta proporciona al usuario final. Este nivel engloba no solo los componentes que forman parte del bloque funcional de la plataforma tecnológica, sino también la parte de servicios ofrecidos.
- **Capa de integración**: en esencia se trata de un conjunto de componentes que permiten la manipulación y acceso a través de APIs de los elementos externos tales como la base de datos, el sistema de directorio organizacional, el servidor de aplicaciones, etc. Esta capa es la encargada de garantizar la independencia de dichos elementos de tal modo que sean intercambiables en un futuro con un coste mínimo y un impacto nulo sobre el resto del sistema.

## Componentes *open source* empleados

A continuación, se presentan los diferentes componentes que se integran bajo la arquitectura de Open Cities, agrupados en diferentes categorías. Dichos componentes han sido elegidos teniendo en cuenta tres aspectos principales:

- se trata de soluciones maduras y asentadas con referencias reales;
- son compatibles tecnológicamente con el entorno base del proyecto, y
- cumplen los estándares y normas adoptadas dentro del proyecto.

### *Motor de* workflow *BONITA*

Open Cities consta de un componente de *workflow* que permite la implantación de flujos de trabajo asociados a procedimientos administrativos. El motor debe soportar el estándar XPDL del WfMC como lenguaje de definición de *workflow*, así como la definición de reglas en función de expresiones y de reglas de negocio. Asimismo, se apoya en el conjunto de servicios *web* de la Open Cities, ya sean servicios simples o composición de servicios dirigidos por el orquestador.

Con estas premisas, en Open Cities se integra el sistema de código abierto Bonita para suministrar la funcionalidad de la plataforma de *workflow*. El proyecto Bonita está enmarcado en ObjectWeb, la principal organización de *middleware* de código abierto en Europa.

### *Repositorio documental FEDORA*

El principal componente Open Source para la funcionalidad del servicio de archivo y custodia es Fedora. Es un desarrollo conjunto de las Universidades de Cornell y Virginia,

con el apoyo de las fundaciones Andrew W. Mellon Foundation y la National Science Foundation.

Fedora ofrece a las organizaciones la posibilidad de una arquitectura orientada a servicios para gestionar su contenido digital. Su núcleo posee un poderoso modelo de objetos digitales que soporta múltiples vistas de cada uno de ellos y sus interrelaciones. Los objetos digitales pueden encapsular contenidos gestionados localmente o hacer referencias a contenido remoto.

## *Motor de orquestación PXE*

El empleo de una plataforma de orquestación de servicios *web* basada en BPEL posibilita la independencia tecnológica de los sistemas y aplicaciones ya existentes, y ofrece un interfaz estándar para que otras aplicaciones, canales y servicios puedan interactuar. Se trata del esquema funcional del entorno propuesto como solución al desarrollo de los portales del ciudadano y del empleado, habituales en la administración local.

Se ha seleccionado el motor de orquestación PXE de Intalio BPMS, que a su vez forma parte del proyecto Apache ODE. Dicho proyecto fue liberado como consecuencia de la adquisición por parte de Intalio de la compañía FiveSight, y de su movimiento estratégico hacia el modelo *open source*.

El producto tiene una madurez de más de 3 años con cientos de implementaciones. Algunos análisis realizados por consultoras en BPM como "*Business Process Management Essentials – Illustrated Using Open Source Solutions*" de Glintech lo califican como el *software* BPM más usado y el proyecto BPM *open source* más ambicioso y prometedor.

Entre las características de Intalio, se encuentran las siguientes:

- Soporte a OASIS WS-BPEL 2.0 y al lenguaje propietario BPEL4WS 1.1.
- Compilación BPEL que permite análisis y validación mediante línea de comandos, programas o entornos de compilación Ant.
- Arquitectura modular, motor de ejecución tipo *microkernel* para facilitar su extensión y su inclusión en otros *containers*, como servidores de aplicación J2EE.
- Soporte JMX para la gestión de tareas administrativas sobre las instancias de proceso.
- API de depuración y gestión de instancias.
- Licencias *open source*: CPL 1.0 y MIT.

## *Generación de XFORMS - Orbeon*

El empleo de tecnología *web* como canal de acceso implica que el mantenimiento de formularios asociados a los diferentes expedientes administrativos debe realizarse a través de esta tecnología. Hoy en día, cobran fuerza las soluciones que emplean XForms como

mecanismo de definición de los formularios *web*, debido a que simplifican y estandarizan la realización de la parte de interacción con el usuario y facilitan su adaptación a cambios legislativos y de procedimiento por personal no técnico.

Open Cities integra la solución Orbeon Presentation Server como tecnología base para la implementación del servicio de definición y soporte a XForms. Dicha solución está basada en el proyecto Objectweb Orbeon Presentation Server y se trata de una solución *open source* bajo licencia LGPL de un motor basado en los estándares XForms que utiliza técnicas AJAX para la presentación y facilita el uso y creación de formularios *web*.

Asimismo, usa documentos XML y XForms, lo que, junto al uso de la arquitectura AJAX, permite que se puedan llevar a cabo tareas de captura, procesamiento y presentación de datos XML (en particular formularios de datos) sin necesidad de escribir programas Java o *scripts* para implementar una presentación dinámica de la aplicación *web*.

## Casos reales de aplicación

En España, hay 6 proyectos en marcha y en distintos puntos de su ciclo de desarrollo que utilizan Open Cities como plataforma base para los procesos de tramitación electrónica:

- Ayuntamiento de Getafe (Madrid): soporte a procedimientos administrativos del área de participación ciudadana y procedimientos internos del empleado municipal. Integración con sistemas de información geográfica y otros sistemas corporativos.
- Excma. Diputación de Toledo: estudio de viabilidad en el soporte de procedimientos internos de la Diputación.
- Biblioteca Nacional Española: soporte a procedimientos internos del empleado público de la Biblioteca, dentro de la *intranet* del empleado.
- Ayuntamiento de Onda (Castellón): implantación de veinte procedimientos administrativos del servicio de atención y tramitación del Ayuntamiento. Integración con los sistemas de padrón, registro general, tesorería, contratación pública electrónica, firma electrónica, registro telemático.
- Concello A Estrada (La Coruña): implantación de las carpetas del ciudadano y del empleado público, para soporte a 10 procedimientos administrativos. Integración con plataformas de pago seguro de RED.ES y registro general.
- Consejería de Medio Ambiente (JCCM): gestión de expedientes de evaluación ambiental, con integración de mecanismos de firma electrónica, sistemas de información geográfica.

# Bibliografía

1. http://www.opendocumentfellowship.org
2. http://www.techtear.com/2007/09/05/microsoft-peligra-office-open-xml
3. http://www.softwarelibre.cl/drupal//?q=node/977
4. http://sourceforge.net/projects/odf-converter
5. http://www.stewart.es/software/que-es-opendocument.jsp
6. http://www.plonegov.org
7. Definitive Guide to Plone, diciembre de 2006
8. Plone en entornos gubernamentales, Roberto Allende, 2007
9. http://www.plonegov.org/products/pure-public-requirements
10. http://www.plonegov.org/products/cpcomarquage
11. http://www.plonegov.org/software/products/pod-python-open-document
12. CommunesPlone, Products/College
13. http://nngov.com/egov/open-egov
14. http://badajoz07.opensourceworldconference.com/virtual/comunicaciones/open-cities.pdf

# Solución presentada

## Relevamiento

El relevamiento de las necesidades y funcionalidades a cubrir se llevó a cabo en La Municipalidad de Pila. Para llevar a cabo esta tarea, se realizaron viajes a la ciudad de Pila, donde se mantuvieron reuniones con el contador municipal Martín Ezequiel Funes, el secretario de Gobierno y Hacienda Fernando David Genzone, los Directores y Jefes de las distintas áreas (Compras, Rentas, Ingresos Públicos, Contaduría, Tesorería y Oficina de Personal) y los empleados de los sectores involucrados en los procesos.

En este punto, vale la pena hacer mención especial a la colaboración y excelente predisposición de parte del Intendente Municipal Gustavo Alfredo Walker y de todos los actores intervinientes, lo cual, obviamente, nos facilitó esta tarea. A todos ellos, nuestro sincero agradecimiento.

## Análisis y diseño

De acuerdo a lo relevado y a partir del análisis efectuado sobre la información obtenida, se determinó que la solución debería estar compuesta por 5 (cinco) módulos: Ingresos; Compras y Contrataciones; Contabilidad y Presupuesto; Tesorería, y Liquidación de Haberes. Elegimos profundizar en el desarrollo de los módulos de Ingresos y Liquidación de Haberes, puesto que además de ser dos puntos muy importantes dentro de la administración municipal, es allí donde identificamos una mayor necesidad, al menos en el municipio relevado y en los de la zona (Chascomús, General Belgrano, Castelli y Dolores), con los cuales nos contactamos a través del Departamento de Informática. Otro de los aspectos que se tuvo en cuenta fue el hecho de que estos dos módulos son los únicos del sistema RAFAM que no han sido totalmente desarrollados y en los cuales hay mayor divergencia entre los diferentes municipios. Esto se da fundamentalmente debido a la diferencia significativa en la naturaleza de las comunas. Por ejemplo, la política de recaudación del Partido de la Costa estará centrada en la tasa por Habilitación e Inspección de Seguridad e Higiene de comercios, mientras que un municipio como Pila, típicamente rural por su pequeña población y la gran extensión de tierras, se centrará en la recaudación a través de la tasa por servicios rurales, de la cual proviene el 90% de sus ingresos.

## Módulo Ingresos Públicos

En este sector de la Administración municipal, se tiene como elemento principal al contribuyente, que constituye la base sobre la cual se sustentan los ingresos a nivel local, pues es él quien recibe los servicios brindados por el municipio (y paga por ellos) y quien tiene a su cargo las tasas municipales mediante las cuales se percibe el ingreso mayoritario de los recursos. En este punto, debemos mencionar que hemos planteado la división de los

Ingresos Públicos en tres grandes grupos, **Ingresos por Tasas Liquidadas**, **Ingresos por Servicios** e **Ingresos de Terceros**.

**Ingresos por Tasas Liquidadas** comprende aquellos ingresos provenientes de pagos en concepto de cancelación de cuotas generadas por liquidaciones periódicas (quincenal, mensual, bimestral, anual, etc.). Dichas liquidaciones generan un devengamiento en las cuentas corrientes de los contribuyentes, en función del objeto imponible (campo, lote, comercio, etc.) y el tributo o tasa que pese sobre éste. Algunos ejemplos de tasas que se liquidan son: tasa por servicios rurales, tasa por alumbrado, barrido y limpieza (ABL), tasa por inspección de seguridad e higiene, tasa por publicidad y propaganda, y patente automotor.

**Ingresos por Servicios** abarca tanto los ingresos percibidos por servicios prestados por la comuna (servicio de movimiento de tierra, servicio atmosférico, publicidades en FM Municipal, etc.), como así también aquellos ingresos considerados derechos (derechos de oficina, derechos de construcción, derecho a espectáculos públicos, etc.). Lo que diferencia a este tipo de ingresos de los anteriores es que no se liquidan, es decir, no se tributan periódicamente sino de forma eventual.

**Ingresos de Terceros** está conformado por los recursos que ingresan al municipio a través de otros organismos provinciales o nacionales. Estos recursos pueden ser subsidios provinciales, fondos provinciales con destino a obras y servicios públicos, o coparticipaciones provenientes de la contaduría general de la provincia (como, por ejemplo, la coparticipación de juegos de azar), entre otros.

Por todo esto, el módulo Ingresos Públicos es una herramienta que permite la gestión y administración de los ingresos municipales a través de la liquidación de las tasas municipales y de los servicios prestados por la comuna, así como también del registro de los Ingresos de Terceros y coparticipaciones provinciales.

Dentro de las utilidades provistas en el módulo, podemos mencionar las básicas, tales como alta, baja y modificación (ABM) de contribuyentes, ABM de propiedades o sujetos imponibles sobre los que se aplicarán las tasas (lotes, campos, comercios, vehículos, etc.), manejo de las cuentas corrientes de dichos contribuyentes en las correspondientes tasas, descuentos, moratorias, planes de pago y todas las funciones necesarias para agilizar la gestión de liquidación y cobro de los diferentes ingresos.

La idea es que la herramienta esté dotada de las funciones necesarias para posibilitar la integración de la administración central (por lo general ubicada en el Palacio Municipal) con las distintas delegaciones (en algunos casos, serán delegaciones rurales, y, en otros, serán oficinas de recaudación ubicadas en diferentes lugares: incluso pueden considerarse oficinas de recaudación los hospitales municipales, las bibliotecas o cualquier otra entidad municipal donde la comuna preste un servicio y perciba por el mismo algún tipo de arancel). Se propone llevar a cabo la integración mediante la implementación de servicios

*web*, a través de los cuales las aplicaciones distribuidas en las distintas oficinas permitan, por ejemplo, la consulta del estado de deuda de los contribuyentes o el pago de una tasa o servicio determinado, además de los servicios correspondientes para llevar a cabo la migración de la recaudación diaria a la oficina central.

Además, con el objetivo de que esta no sea una solución aislada de la realidad actual y teniendo en cuenta la implementación del sistema RAFAM en la gran mayoría de los municipios de la Provincia, incluso en la municipalidad de Pila, se han desarrollado herramientas tendientes a dotar a la aplicación Alcalde (fundamentalmente a los módulos de Ingresos y Caja) de las funcionalidades correspondientes para trabajar con el nuevo sistema contable, provisto por la provincia de Buenos Aires. Para cumplir con esta premisa, se diseñó e implementó un nuevo submodelo de datos en la base de datos Alcalde existente, para brindar soporte a la nueva estructura de cuentas del modelo RAFAM. Esto, en lo que respecta al modelo de datos.

Por su parte, en cuanto a la aplicación, se llevó a cabo el correspondiente diseño, desarrollo e implementación de las nuevas herramientas para cumplir con la interfaz hacia el sistema RAFAM. Este nuevo conjunto de funciones conlleva la migración de los ingresos diarios del sistema Alcalde al sistema RAFAM, particularmente al módulo Tesorería, que se realiza vía archivos .txt generados por el módulo de Ingresos con la estructura especificada por el sistema RAFAM y almacenados en la red para su posterior importación a través de funciones provistas por ese sistema. Como complemento a estas nuevas funciones, se generaron reportes y listados que sirven de soporte a la información generada y exportada.

## Principales entidades

### *Contribuyentes*

Por su naturaleza, son personas físicas o jurídicas que, como dijimos al comienzo, constituyen el elemento fundamental en este sector de la administración, ya que son los contribuyentes quienes tienen a su cargo las tasas municipales con las que se sostiene la administración municipal, además de ser quienes utilizan los servicios prestados por la comuna y reciben, de una u otra manera, los beneficios (ya sea a través de servicios sociales u obras públicas) gestionados y administrados por el municipio local.

### *Tasas municipales*

Son recursos generados por la prestación efectiva de un servicio público individualizado donde el sujeto pasivo es el contribuyente (persona física o jurídica). En nuestro diseño, como mencionamos anteriormente, decidimos diferenciar las tasas correspondientes a los servicios o derechos que se liquidan o devengan en forma periódica de aquellos que se perciben en forma eventual; a los del primer grupo los denominamos

simplemente tasas y a los del segundo grupo, servicios al contribuyente (más allá de que puedan ser tanto ingresos por prestación de servicios como tasas por derechos).

## *Servicios al contribuyente*

Parte de su definición se encuentra en la de tasas municipales, ya que, como dijimos antes, en nuestro diseño, definimos como servicios al contribuyente a todos aquellos ingresos por contraprestación de servicios o por cobro de derechos y obligaciones de terceros para con el municipio, que tengan lugar de manera **eventual**. En el momento de dar de alta en el sistema un ingreso de este tipo, se generará un recibo que contendrá como datos principales al contribuyente, el servicio en cuestión (por ejemplo, servicio atmosférico fuera de planta urbana) y la variable (en este ejemplo, los kilómetros recorridos por el camión atmosférico) que, junto a la tarifa a aplicar, conformarán la base imponible para ese servicio o derecho. Vale la pena aclarar que los servicios configurados en la aplicación tendrán configurado un tipo de servicio, que contendrá información tal como, por ejemplo, la cuenta de imputación a la que ingresarán los recursos.

## *Ingresos de terceros*

Del mismo modo que ocurrió con tasas y servicios, ya hemos mencionado parte de la definición de ingresos de terceros, por lo tanto, sólo resta decir que, dentro de nuestra solución, el alta de un ingreso de este tipo se realiza mediante la función provista para tal fin en el menú de ingresos de terceros, en la cual se encontrará el ABM correspondiente para dar de alta un recibo, cuyos principales datos serán: el contribuyente, una cuenta de terceros (en la cual se realizará el asiento de ingreso del recurso), el monto y la fecha de ingreso.

## *Propiedades*

Las propiedades son los objetos imponibles, es decir, los objetos en función de los cuales, a partir de las alícuotas que correspondan en cada caso, se calcula el impuesto determinado. A modo de ejemplo, en el caso de la tasa por servicios rurales, el monto del impuesto estará determinado por el resultado de multiplicar la cantidad de hectáreas del campo por el monto por hectárea definido en la Ordenanza Tributaria Municipal. Entre las entidades que pueden ser consideradas propiedades, podemos mencionar: campos, lotes, comercios, viviendas y vehículos, entre otros. Básicamente, puede ser considerada como propiedad cualquier entidad que cuadre como objeto imponible dentro de nuestro modelo de liquidación de tasas.

## *Cuentas corrientes de contribuyentes*

Una cuenta involucra tres entidades: **contribuyente**, **propiedad** y **tasa**. Funciona como una cuenta corriente y sobre dichas cuentas se realizan los débitos correspondientes a las liquidaciones de la tasa, además de la imputación de los pagos percibidos.

## *Ordenanza Tributaria Municipal*

Conocida también como Tarifaria Municipal o Código Tributario, es la ordenanza municipal mediante la cual se establece y regula todo lo referido a los tributos municipales para un período o ejercicio dado. Entre otras cosas, dicha ordenanza puede definir o establecer:

- el hecho imponible;
- el contribuyente y, en su caso, el responsable del pago del tributo;
- la base imponible, la alícuota o el monto del tributo;
- exenciones o beneficios;
- infracciones y sanciones, y
- procedimientos para la determinación y fiscalización de las obligaciones tributarias.

## *Liquidación o devengamiento de tasas*

Se denomina de este modo a la tarea de generar movimientos deudores en las cuentas corrientes de los contribuyentes en una tasa dada, en función del objeto imponible, para un período específico. La liquidación tiene lugar en las tasas municipales, de acuerdo a lo especificado por la Ordenanza Tributaria para la tasa en cuestión. Por ejemplo, en el caso de la tasa por servicios rurales, el tributo se divide en seis cuotas anuales y bimestrales, motivo por el cual en cada ejercicio existirán seis liquidaciones o devengamientos para dicha tasa.

## *Multas y contravenciones*

En cuanto a las multas y contravenciones, su dominio y su alcance también están establecidos en la Ordenanza Tributaria Municipal. Allí, se establece qué situaciones se entienden como infracción o contravención y qué multas se prevén para cada caso.

## *Recibos pendientes*

Son aquellos recibos generados a demanda o por una liquidación o devengamiento de una tasa, ya sea por tasas, servicios o ingresos de terceros, pendientes de pago. Dichos recibos tienen una fecha de vencimiento que, una vez superada, lleva a su eliminación a través de una tarea que se ejecuta diariamente, para volver a ser generados con los movimientos de interés correspondientes.

## *Recibos cobrados*

Son aquellos recibos que han sido cancelados por los contribuyentes. La cancelación de los recibos genera diferentes movimientos dependiendo de su tipo: por ejemplo, en el caso de los llamados recibos por tasas liquidadas, se genera, además del movimiento de ingreso en la cuenta de imputación correspondiente, el registro del movimiento de pago en la cuenta corriente asociada al contribuyente, la tasa y el objeto imponible.

## *Cuentas de imputación*

También denominadas cuentas de recursos, en ellas se clasifican e ingresan los recursos con los que cuenta el municipio. Se clasifican y agrupan en forma de árbol, y se sumarizan en todos los niveles, aunque reciben asiento sólo en las cuentas-hoja. Un ejemplo de plan de cuentas de imputación puede ser el siguiente:

| Codificación | Descripción | Hoja (recibe asiento) |
|---|---|---|
| 1 | Ingresos corrientes | |
| 1.1 | De jurisdicción municipal | |
| 1.1.1 | Tributos municipales | |
| 1.1.1.2 | Tasa por servicios especiales de limpieza e higiene | ✔ |
| 1.1.1.3 | Tasa por habilitación de comercio e industria | ✔ |
| 1.1.1.4 | Derechos de publicidad | ✔ |
| … | … | |
| 1.1.2 | Otros ingresos | |
| 1.1.2.3 | Multas por contravenciones | ✔ |
| 1.1.2.4 | Infracción a las obligaciones y deberes fiscales | ✔ |
| … | … | |
| 1.2 | De otras jurisdicciones | |
| 1.2.1 | Régimen de coparticipación | |
| 1.2.1.1 | Régimen de coparticipación vial | |
| 1.2.1.1.1 | Ley 8071 | ✔ |
| 1.2.1.2 | Coparticipación impositiva | |
| 1.2.1.2.1 | Coparticipación provincial de impuesto Ley 10559 | ✔ |
| 1.2.1.2.2 | Participación en juegos de azar | ✔ |
| … | … | |
| 2 | Ingresos de capital | |
| … | … | |

Tabla 3. Plan de cuentas de imputación

En nuestra solución, existe un Plan de Cuentas de Recursos que tiene su correspondencia en el Plan de Cuentas de Recursos del sistema RAFAM, donde las cuentas de uno y otro modelo están relacionadas entre sí, para facilitar la migración de los ingresos en las cuentas de nuestro modelo hacia el modelo RAFAM.

## *Usuarios del sector Ingresos*

Los usuarios del sector Ingresos son aquellos empleados municipales del sector Ingresos Públicos (o Rentas, como se denominaba antiguamente al sector) que tienen asignadas funciones administrativas relativas a los contribuyentes y las tasas y servicios, para lo cual disponen del módulo en cuestión como herramienta de soporte.

### *Usuarios del sector Tesorería*

Los usuarios del sector Ingresos son aquellos empleados administrativos del sector Tesorería, más precisamente del sector Cajas, que utilizan una parte del módulo Tesorería de la actual solución, fundamentalmente las utilidades disponibles para el cobro de los recibos y la correspondiente migración diaria de dichos ingresos al sistema RAFAM.

## Principales funciones

Aquí detallaremos las principales funciones del módulo, para comprender mejor qué aspectos de cada sector resuelve.

En relación al ABM de contribuyentes, contiene una herramienta para listar las cuentas corrientes que posee cada uno, lo que facilita la visualización del estado de deuda de un contribuyente determinado en sus diferentes cuentas.

En cuanto al ABM de propiedades, se ocupa de campos, lotes, comercios, viviendas, motos y automotores.

En relación al ABM de cuentas corrientes, facilita, entre otras, las siguientes tareas: gestión de movimientos (visualización de fechas de vencimiento de movimientos, reimpresión de boletas de pago, etc.); generación de recibos de movimientos vencidos (adeudados); generación de planes de pago/moratorias de movimientos adeudados; categorización de contribuyentes (por ejemplo, para la tasa por servicios rurales), y ABM de domicilios postales (asociados a la cuenta corriente y no al contribuyente).

En cuanto a la gestión de liquidaciones de tasas, se ocupa, entre otras cosas, del ABM de liquidaciones, de la manera que se describe a continuación. Una vez seleccionada una tasa, permite el alta de una nueva liquidación que contendrá la siguiente información:

- La cuota a la que corresponde la liquidación (se presentan las opciones de acuerdo a la cantidad de cuotas configuradas para la tasa seleccionada).
- El año de la cuota (que puede diferir del año correspondiente a la fecha en que fue liquidada o el año correspondiente a su fecha de vencimiento: por ejemplo, la cuota 6 del ejercicio 2009 correspondiente a la tasa por seguridad e higiene cuya fecha de vencimiento está establecida como el 31 de enero de 2010, probablemente se liquide en enero de 2010).
- El período de liquidación (mes par o impar, relacionado principalmente con el padrón de cuentas corrientes de contribuyentes de la tasa por servicios rurales, dividido en dos grandes grupos para garantizar el ingreso por dicha tasa todos los meses, puesto que su frecuencia de liquidación es bimestral).
- La fecha de liquidación.
- La fecha de vencimiento.

En este punto, se ocupa de generar el devengamiento o la liquidación de una deuda: a partir de la selección de la tasa, el año, la cuota y el período, se habilita la herramienta mediante la cual se permite el devengamiento de deuda para la totalidad del padrón de cuentas corrientes, o bien para una cuenta corriente especificada a través de un número de cuenta válido.

El módulo permite, luego, la impresión de los recibos de una liquidación seleccionada y la visualización de totales devengados, totales de descuento y otros relativos a la liquidación seleccionada.

En cuanto a la gestión de servicios prestados al contribuyente, este módulo se ocupa del ABM de tasas por servicios, el ABM de tipos de servicios prestados y el ABM de servicios prestados al contribuyente. Entre los datos importantes del primero, se encuentra la cuenta de imputación a la que se imputarán los ingresos. El ABM de tipos de servicios prestados representa la lista o tipificación de los servicios que se prestan en el municipio y contiene, entre otros datos, la tasa por servicio (entidad que los agrupa) y el monto o tarifa a aplicar por dicho servicio.

En relación a la gestión de ingresos de terceros, el módulo gestiona el ABM de cuentas de terceros y el ABM de ingresos de terceros. Contiene, además, una herramienta que permite el cobro y la impresión masiva de los recibos correspondientes a las tasas de aquellos contribuyentes que son agentes municipales, y a los cuales se les retiene el importe por dicha tasa a través de ítems de descuento en sus recibos de haberes. Una vez realizada la liquidación de haberes de cada mes, se genera en el módulo de Personal la información necesaria para que los usuarios del departamento Ingresos puedan efectuar la imputación de pagos de los recibos en las cuentas corrientes de los contribuyentes que están asociadas a un número de legajo de un agente municipal, por los importes retenidos en la liquidación de haberes.

La herramienta para el cobro de los recibos gestiona los recibos por tasas liquidadas, por servicios prestados al contribuyente y por ingresos de terceros, tanto cobrados como pendientes de cobro.

La herramienta para gestión de cuentas bancarias se encarga del ABM de Bancos, el ABM de cuentas bancarias, el ABM de movimientos, el libro-banco y el listado de movimientos de cuentas bancarias por tipo de movimiento y número de documento (por ejemplo, por número de interdepósito) para facilitar la conciliación bancaria.

Como ya mencionamos en la descripción general del presente módulo, la herramienta se compone, básicamente, de la funcionalidad necesaria para la migración de los ingresos diarios del sistema Alcalde al sistema RAFAM, particularmente al módulo Tesorería. Esta tarea se realiza exportando la información de los ingresos diarios, agrupados por cuenta de imputación, vía archivos .txt generados por el módulo de Ingresos, con la estructura

especificada por el sistema RAFAM, y almacenados en la red para su posterior importación a través de funciones provistas por dicho sistema.

Este módulo permite la impresión de los siguientes listados e informes:

- Planilla de ingresos por tasas liquidadas (por rango de fechas).
- Planilla de ingresos por servicios (por rango de fechas).
- Planilla de ingresos por ingresos de terceros (por rango de fechas).
- Listado de deuda de tasas (permite seleccionar una tasa determinada y una fecha hasta la cual se considera la deuda).
- Listado de padrones (de contribuyentes, vehículos, campos, etc.)

## Módulo Liquidación de Haberes

Este módulo se encarga de la administración del personal, lo que involucra el registro de movimientos de personal como incorporaciones, bajas y modificaciones en el escalafón, así como también todo lo directamente relacionado con la liquidación de haberes propiamente dicha. Respecto de esto último, podemos destacar el diseño de un conjunto de utilidades que le otorgan a la herramienta una gran flexibilidad y una rápida adaptación a los constantes cambios en materia de leyes y reglamentaciones laborales. Cabe destacar, entonces, que estos cambios, en estructuras rígidas, impactarían de manera costosa en la forma de modificaciones en conceptos o fórmulas de cálculo, lo que generaría rediseños o cambios importantes en la estructura de los programas, mientras que en nuestro modelo son algo totalmente configurable.

### Principales entidades

#### *Legajo*

Se denomina así al conjunto de datos acerca del empleado acerca de su situación personal y no su situación laboral. Dentro de la información contenida en el legajo, podemos mencionar la siguiente:

- número de legajo;
- apellido y nombre;
- tipo y número de documento;
- CUIL;
- sexo;
- estado civil;
- fecha de nacimiento;
- fecha de ingreso a la Municipalidad;
- domicilio;
- antecedentes laborales;
- familiares con sus datos personales, grados de parentesco, escolaridad, etc;

- obras sociales, y
- estudios (títulos obtenidos).

## *Agente*

Así como el legajo agrupa información acerca de los datos personales del empleado, el agente contiene la información de todo lo que concierne a lo estrictamente laboral, es decir, el cargo que ocupa el empleado dentro del municipio. Así, un agente es la asociación de un legajo a un escalafón específico, en un intervalo de tiempo. Vale la pena destacar que un legajo puede ir pasando a través del tiempo por diferentes puestos de trabajo o cargos dentro del municipio, por lo que, si bien será siempre el mismo legajo, los agentes serán distintos, siempre teniendo en cuenta que un legajo no puede estar al mismo tiempo en dos cargos diferentes, por lo que los intervalos fecha desde y fecha hasta no se podrán superponer. Son datos propios del agente, como ya dijimos, el legajo y el escalafón, fecha desde, fecha hasta y tipo de relación laboral, entre otros.

## *Escalafón*

Agrupa información del cargo o puesto de trabajo que ocupa el agente dentro del municipio. Dicha información está compuesta, básicamente, por la jurisdicción a la que pertenece la oficina donde desempeña la tarea, el agrupamiento del cargo (tipo de personal), categoría de revista, cargo que desempeña y finalidad y programa al que está destinado el cargo. A continuación, mostramos un conjunto de escalafones de diferentes programas y finalidades con cargos representativos, para ejemplificar mejor:

| Jurisdicción | Finalidad | Programa | Agrupamiento | Cargo | Categoría |
|---|---|---|---|---|---|
| Departamento ejecutivo | Adm. gral. | Cultura y educación | Personal jerárquico | Director | 15 |
| Departamento ejecutivo | Adm. gral. | Cultura y educación | Personal técnico | Ayudante Bibliotecario | 6 |
| Departamento ejecutivo | Adm. gral. | Cultura y educación | Personal administrativo | Auxiliar Administrativo | 5 |
| Departamento ejecutivo | Adm. gral. | Cultura y educación | Personal de servicio | Ordenanza | 2 |
| Departamento ejecutivo | Adm. gral. | Adm. gral. sin discriminar | Personal superior | Intendente | 1 |
| Departamento ejecutivo | Adm. gral. | Adm. gral. sin discriminar | Personal superior | Secretario General | 1 |
| Departamento ejecutivo | Adm. gral. | Adm. gral. sin discriminar | Personal jerárquico | Contador Municipal | 1 |

| | | | | | |
|---|---|---|---|---|---|
| Departamento ejecutivo | Adm. gral. | Adm. gral. sin discriminar | Personal jerárquico | Director | 15 |
| Departamento ejecutivo | Adm. gral. | Adm. gral. sin discriminar | Personal jerárquico | Jefe Departamental | 14 |
| Departamento ejecutivo | Adm. gral. | Adm. gral. sin discriminar | Personal administrativo | Oficial Superior | 10 |
| Departamento ejecutivo | Adm. gral. | Adm. gral. sin discriminar | Personal administrativo | Oficial Contable | 7 |
| Departamento ejecutivo | Adm. gral. | Adm. gral. sin discriminar | Personal administrativo | Oficial Administrativo | 7 |
| Departamento ejecutivo | Adm. gral. | Adm. gral. sin discriminar | Personal de servicio | Chofer | 6 |
| Departamento ejecutivo | Salud pública | Hospital municipal | Personal superior | Secretario de Salud | 1 |
| Departamento ejecutivo | Salud pública | Hospital municipal | Personal profesional | Profesional | 12 |
| Departamento ejecutivo | Salud pública | Hospital municipal | Personal técnico | Enfermero Profesional | 10 |
| Departamento ejecutivo | Salud pública | Hospital municipal | Personal obrero | Chofer Ambulancia | 6 |
| Departamento ejecutivo | Salud pública | Hospital municipal | Personal de servicio | Mucama | 2 |
| Departamento ejecutivo | Salud pública | Hospital municipal | Personal jornalizado | Guardia Médico Clínico | 1 |
| H.C.D. | Adm. gral. | H.C.D. | Personal superior | Concejal | 1 |
| H.C.D. | Adm. gral. | H.C.D. | Personal administrativo | Secretario H.C.D. | 1 |

Tabla 4. Ejemplo de escalafones y cargos representativos

## *Jurisdicción*

Es el ámbito en el que se agrupan las finalidades. Por lo general, está asociado a los poderes, que, en el caso de los municipios, se encuentran divididos en Poder Ejecutivo y Poder Deliberativo, lo que da lugar a dos ámbitos jurisdiccionales: el del Poder Ejecutivo y el del Honorable Concejo Deliberante.

## *Agrupamiento*

Es la forma de agrupar a los agentes municipales de acuerdo al **tipo de personal**. Una categorización de los agrupamientos puede ser la siguiente:

- 1 - Personal superior
- 2 - Personal jerárquico
- 3 - Personal profesional
- 4 - Personal técnico
- 5 - Personal administrativo
- 6 - Personal obrero
- 7 - Personal de servicio
- 8 - Personal docente
- 9 - Personal de carrera hospitalaria
- 10 - Personal de cómputos
- 11 - Personal mensualizado
- 12 - Personal jornalizado
- 13 - Personal destajista
- 14 - Retribuciones a personas

## *Categoría*

También conocida como categoría de revista, es uno de los parámetros, junto con el agrupamiento, que sirven como base de cálculo para el concepto sueldo básico en la conformación de la remuneración de los agentes. Para esto, existe una tabla que especifica para cada categoría y agrupamiento, la cantidad de **módulos** que corresponden a la remuneración salarial. Luego, con este dato y en función al **valor del módulo**, se obtiene el importe del sueldo básico que será percibido por todos aquellos agentes que pertenezcan a igual categoría y agrupamiento.

## *Cargo*

Refiere al cargo que desempeña el agente dentro del municipio. Sirve para diferenciar el puesto de trabajo específico dentro de una misma finalidad, programa, agrupamiento y categoría, ya que, si bien dos agentes pueden desempeñar sus funciones en la misma oficina o dependencia municipal y con el mismo agrupamiento y la misma categoría de revista, la tarea que desempeñan puede ser muy diferente y, por ese motivo, puede ser que se le deban liquidar cierto tipo de conceptos específicos que tengan que ver con dicha tarea. Por ejemplo, se puede dar que en el programa 3 – Administración General sin Discriminar de la finalidad 1 – Administración General, agrupamiento 5 – Personal Administrativo y categoría de revista 10, existan dos agentes, uno, Cajero, y el otro, Oficial

de Tesorería. Los dos agentes pertenecerían incluso al mismo departamento (Tesorería), pero al cajero se le debería liquidar el concepto **Fallo de Caja** (por manejar dinero público) y al otro no.

Además de dar una descripción más real de la función que desempeñan los agentes, el escalafón especifica y establece los conceptos que se liquidan para todos los agentes del escalafón.

## *Finalidad*

Así como existen las jurisdicciones que agrupan las finalidades, estas últimas sirven para agrupar los diferentes programas en los que se distribuye el presupuesto. Hablar de una finalidad es similar a hablar de un sector o secretaría a la que se destinarán las partidas presupuestarias con las cuales se financiarán las actividades en un ejercicio económico. A continuación, presentamos un ejemplo de las finalidades que podemos encontrar en un municipio:

| Jurisdicción | Número de finalidad | Descripción de la finalidad |
|---|---|---|
| Departamento ejecutivo | 1 | Administración central |
| Departamento ejecutivo | 2 | Secretaría de salud |
| Departamento ejecutivo | 3 | Servicios especiales urbanos |
| Departamento ejecutivo | 4 | Vialidad municipal |
| Departamento ejecutivo | 5 | Promoción social |
| H.C.D. | 1 | H.C.D. |

Tabla 5. Ejemplo de finalidades

## *Programa*

Es el conjunto de partidas presupuestarias destinadas a un mismo sector económico y social, dentro de una misma finalidad. Esta entidad nos permite agrupar las partidas presupuestarias para la ejecución del gasto público, para lograr un mayor control de él y facilitar así la coordinación entre los planes de desarrollo y el presupuesto. Si, por ejemplo, consideramos que el monto del presupuesto destinado a gastos en combustible (partida 1.1.2.2, Combustibles y Lubricantes) de la finalidad 1 – Administración Central será de $1000, tendremos un control mucho mayor si especificamos que se destinarán $500 para Deportes y Recreación, $300 para Cultura y Educación y $200 para la movilidad del Intendente. Para este tipo de situaciones es que existen los programas dentro de las finalidades. Como ejemplo de programas en una finalidad, podemos ver los correspondientes a la finalidad 1 – Administración Central:

| Finalidad | Número de programa | Descripción del programa |
|---|---|---|
| Administración central | 1 | Cultura y educación |
| Administración central | 2 | Deportes y recreación |
| Administración central | 3 | Adm. gral. sin discriminar |
| Administración central | 4 | Servicios especiales |
| Administración central | 5 | Producción y empleo |
| Administración central | 6 | FM municipal |

Tabla 6. Ejemplo de programas de una finalidad

## *Partida presupuestaria*

Son los recursos que el municipio estima o prevé aplicar en un período determinado para el cumplimiento de un objetivo específico. Las partidas presupuestarias permiten clasificar los egresos (gasto público) previstos en el presupuesto. Como ya anticipamos, el gasto público se agrupa en programas y finalidades, pero además, en pos de proporcionar una mayor información para el estudio de la economía y de las políticas económicas que aplicará el municipio, se pueden organizar y clasificar los egresos de diferentes formas, entre las cuales se encuentran las siguientes:

- **Clasificación institucional**: permite ordenar los gastos públicos de las dependencias e instituciones a las cuales se asigna el crédito presupuestario para el cumplimiento de sus objetivos.
- **Clasificación por la naturaleza del gasto**: permite la identificación y el seguimiento de los bienes y servicios que se adquieren con las asignaciones previstas en el presupuesto. Facilita, además, el control de las variaciones de los activos y pasivos del sector público.
- **Clasificación económica**: permite ordenar el gasto público de acuerdo a la estructura básica del sistema de cuentas nacional. Los principales rubros en los que se los puede clasificar son los siguientes:
  - Gastos corrientes: son los gastos de consumo o producción, la renta de la propiedad y las transacciones otorgadas a los otros componentes del sistema económico para financiar gastos de esas características.
  - Gastos de capital: son los gastos destinados a la inversión real y las transferencias de capital que se efectúan con ese propósito a los exponentes del sistema económico.

| Codificación | Partida | Recibe asiento | Presupuestado |
|---|---|---|---|
| 1 | Erogaciones corrientes | | $360.000 |
| 1.1 | Funcionamiento | | $360.000 |
| 1.1.1 | Gastos en personal | | $200.000 |
| 1.1.1.1 | Sueldos individuales | | $200.000 |
| 1.1.1.1.2 | Personal superior | ✔ | $85.000 |
| 1.1.1.1.3 | Personal jerárquico | ✔ | $100.000 |
| 1.1.1.1.4 | Personal técnico | ✔ | $15.000 |
| … | … | … | … |
| 1.1.2 | Bienes y servicios | | $160.000 |
| 1.1.2.1 | Alquileres y arrendamientos | ✔ | $120.000 |
| 1.1.2.2 | Combustibles y lubricantes | ✔ | $40.000 |
| … | … | … | … |
| 2 | Erogaciones de capital | | $150.000 |
| 2.5 | Inversión física | | $150.000 |
| 2.5.1 | Bienes | | $150.000 |
| 2.5.1.4 | Vehículos varios y embarcaciones | ✔ | $50.000 |
| 2.5.1.5 | Inmuebles | ✔ | $100.000 |
| … | … | … | … |

Tabla 7. Ejemplo de plan de cuentas de partidas presupuestarias

**(*)** Las partidas se agrupan de acuerdo a su codificación, conformando así los totales de las **partidas totalizadoras**.

## *Presupuesto*

Es una estimación o previsión de lo que se gastará (egresos o gastos públicos) durante un período de ejecución, para el cumplimiento de los objetivos. Por lo general es un ejercicio económico (que coincide con un año calendario). El presupuesto está compuesto por todas las partidas presupuestarias de todos los programas de todas las finalidades.

## *Concepto*

Los conceptos de liquidación son, básicamente, los ítems que componen una liquidación de haberes. Entre otros, pueden ser de haberes, de descuento y para cálculo de otros conceptos, como veremos más adelante. Dependiendo de su tipo, los conceptos pueden sumar o restar el importe correspondiente a su ítem dentro de la liquidación del agente y que puede sumar o restar puede ser un valor fijo o bien el resultado de una consulta. Otra característica de los conceptos es que pueden ser aplicados o estar asociados a nivel de escalafón (se liquidan para todos los agentes que pertenecen al mismo escalafón) o bien a nivel agente específicamente (conceptos que estarán incluidos sólo para el agente). Ejemplos de conceptos pueden ser los siguientes:

| Código | Nombre | Tipo de concepto |
|---|---|---|
| 1 | Sueldo básico | **Haberes** |
| 22 | Bonificación título | **Haberes** |
| 23 | Fallo de caja | **Haberes** |
| … | … | … |
| 601 | Aporte a IPS 14 % | **Descuentos** |
| 609 | Aporte a IOMA 4,8 % | **Descuentos** |
| 629 | Bono solidario de UPCN | **Descuentos** |
| … | … | … |
| 1000 | Valor módulo | **Para cálculo** |
| 1001 | Cantidad de módulos | **Para cálculo** |
| 1002 | Cantidad de hijos no discapacitados | **Para cálculo** |
| … | … | … |
| 801 | Aporte patronal a IPS | **Aportes patronales** |
| 809 | Aporte patronal a IOMA | **Aportes patronales** |
| … | … | … |

Tabla 8. Ejemplo de conceptos

## *Tipo de concepto*

En el punto anterior, dimos la definición de concepto y hablamos de que hay diferentes tipos de concepto: incluso en los ejemplos que expusimos hemos tomado la precaución de incluir conceptos de diferentes tipos para graficar mejor la cuestión. El tipo de concepto es importante porque define el rol del concepto en la liquidación. A continuación, detallaremos algunos de los tipos de concepto.

### Haberes

Son conceptos cuyos importes operan positivamente en la liquidación de haberes cuando son incluidos como un ítem en ella. Son conceptos asignados a los agentes como parte de la remuneración por la tarea desarrollada, o bien bonificaciones de distinta índole.

### Descuentos

Como contrapartida de los conceptos de haberes, se encuentran los conceptos de descuento, que, incluidos como ítems de una liquidación, operan en forma negativa. El importe de estos conceptos puede ser fijo o calculado en función de otros conceptos, como, por ejemplo, el concepto 609 – Descuento de IOMA, cuyo importe resulta de la ejecución de una **consulta** que es básicamente el cálculo de un porcentaje (4,8) sobre el concepto 1008 – Suma con Aportes, que se obtiene también a partir de la ejecución de otra consulta que devuelve la sumatoria de todos los conceptos de haberes que llevan aportes.

### Para cálculo

Dentro de los tipos de conceptos existentes, se encuentran aquellos cuya función es servir como parámetro para el cálculo de otros conceptos. Este tipo de conceptos no se incluye directamente como un ítem de la liquidación de haberes, sino que forman parte del cálculo de otros conceptos. Como ejemplo de este tipo de conceptos, podemos mencionar el concepto que citamos en el ejemplo anterior, en el que la consulta utilizada para obtener el importe de descuento del concepto 609 – Descuento de IOMA tenía asociada una consulta que utilizaba como base de cálculo al concepto 1008 – Suma con aportes, un concepto **para cálculo**.

### Aportes patronales

Son conceptos que, si bien se incluyen en la liquidación de haberes, no se ven reflejados en los recibos de haberes de los agentes. Esto es así porque son aportes que realiza el municipio por cada agente a diferentes organismos, en cumplimiento de sus obligaciones de carácter previsional o social. Son ejemplos de este tipo los conceptos 801 – Aporte patronal a IPS y 809 – Aporte patronal a IOMA.

### Aportes personales (totales)

Este tipo de conceptos se utiliza para el cálculo de totales a pagar en la generación de órdenes de pago por el total de los importes retenidos a los agentes por diferentes motivos. Por citar algunos de los motivos que pueden justificar una retención a un agente, podemos mencionar una sentencia judicial de familia (por ejemplo, por embargos por juicio de alimentos de hijos de padres divorciados), una sentencia judicial por embargo ordenado por incumplimiento con entidades bancarias, las retenciones partidarias de UCR, PJ, etc. (practicadas a los agentes con cargo jerárquico dentro del gobierno municipal), entre otros.

### Genérico

Son conceptos utilizados exclusivamente en las consultas y obtenidos al comienzo de cada liquidación, puesto que se utilizarán seguramente en más de un cálculo. Dentro de los conceptos genéricos, se encuentran los siguientes:

- Legajo: el número de legajo que se está liquidando.
- Agente: el código de agente que se está liquidando.
- Fecha inicio: es la fecha en la que inicia (fecha desde) el período liquidado
- Fecha fin: es la fecha en que finaliza (fecha hasta) el período liquidado.
- Liquidación: es el código de la liquidación de haberes en cuestión.

## *Tipo de asignación*

El tipo de asignación es una característica que ayuda a determinar, junto con el tipo de concepto, el rol de los conceptos en la liquidación de haberes, fundamentalmente a la hora de ser tenidos en cuenta o no para la conformación de importes de otros conceptos. Para comprender mejor la idea, abordemos un ejemplo. Si bien los conceptos de tipo haberes juegan a favor, es decir, suman en la liquidación de haberes, para el cálculo del concepto de descuento 609 - Descuento IOMA, según lo especificado en su consulta, se deben considerar todos aquellos conceptos de tipo haberes cuyo tipo de asignación sea **sujetos a aportes**. En nuestra aplicación, tendrían los siguientes tipos de asignación:

- 1 – Básico;
- 2 – Remunerativo no bonificable;
- 3 – Bonificación, y
- 4 – Remunerativo.

## *Consulta*

En la descripción de las entidades que anteceden, ya hemos mencionado en un par de oportunidades la existencia de **consultas** que se utilizan para el cálculo de conceptos, cualquiera sea su tipo. Estas consultas son almacenadas en la base de datos, y pueden ser reutilizadas como base para el cálculo de numerosos conceptos, otorgando a la herramienta una gran flexibilidad y versatilidad frente a los constantes cambios. Las consultas se componen básicamente de un algoritmo simple, escrito en código SQL Standard, que incluye parámetros que son, a su vez, también conceptos de **entrada** o de **salida**. Estos parámetros, al tratarse de conceptos, pueden ser el resultado de la ejecución de otras consultas. Siguiendo con el ejemplo del concepto 609 – Descuento IOMA, su importe se obtiene al aplicar la consulta asociada denominada IOMA 4,8, que contiene el siguiente código SQL:

```
select round( ( #1008# ) * 0.048, 2 ) as result from dual
```

En este caso, el parámetro **#1008#** es un parámetro de entrada que será reemplazado con un valor cuando se ejecute la consulta. En este caso, dicho valor se obtendrá a partir del concepto 1008 – Suma con aportes, un concepto para cálculo que también obtiene su valor a partir de la ejecución de una consulta asociada. Esto está especificado en la tabla de parámetros denominada **consultaparametro**, donde se establecen los parámetros de entrada y salida para cada consulta. Para el ejemplo en cuestión, los parámetros especificados para la consulta IOMA 4,8 son los siguientes:

| Consulta | Parámetro | Orden de ejecución | Código de concepto | Tipo de dato | *In/Out* |
|---|---|---|---|---|---|
| IOMA 4,8 | Suma | 1 | 1008 | *Double* | *I* |
| IOMA 4,8 | Ioma | 2 | 609 | *Double* | *O* |

Tabla 9. Parámetros para la consulta IOMA 4,8

La consulta asociada al concepto 1008, que tendrá que ejecutarse previamente para obtener el valor, se denomina Suma con Aportes, y tiene el siguiente código SQL:

```
select SUM(valor) as result from detalleliquidacion
where LiquidacionKey = #90003# and AgenteKey = #90004# and
TipoAsignacionKey in (1,2,3,4)
```

En este caso, al ejecutarse la consulta en cuestión, se reemplazarán los valores **#90003#** y **#90004#,** ya que son parámetros de entrada. La siguiente tabla grafica la especificación de parámetros para dicha consulta:

| Consulta | Parámetro | Orden de ejecución | Código de concepto | Tipo de dato | *In/Out* |
|---|---|---|---|---|---|
| Suma con Aportes | Liquidación | 1 | 90003 | *Integer* | *I* |
| Suma con Aportes | Agente | 2 | 90004 | *Integer* | *I* |
| Suma con Aportes | Suma | 3 | 1008 | *Double* | *O* |

Tabla 10. Parámetros para suma con aportes

## *Novedad*

En nuestra implementación, hemos decidido denominar de esta manera a aquellos conceptos que se asocian a nivel del agente. Así, todos los conceptos que deban ser incluidos en una liquidación de haberes, más allá de que deban ser liquidados en forma **eventual** (solo para la liquidación en curso, especificando la fecha desde y la fecha hasta comprendidas entre las fecha desde y fecha hasta de la liquidación de haberes, son ejemplo de eventuales los conceptos 202 – Viáticos por destino, 204 – Vacaciones no gozadas, 205 – Adelanto de haberes, etc.) o **permanente**, se deben cargar como una novedad al agente.

## *Liquidación de haberes*

La liquidación se trata básicamente y a grandes rasgos, de la asignación de valores a los conceptos asociados a los agentes, ya sea a nivel del agente o a nivel de su escalafón. Esta tarea se lleva a cabo una vez que han sido cargadas al sistema todas las novedades del mes y a partir de la orden de los responsables del sector. El proceso consiste en recorrer la base de datos, agente por agente, e ir tomando los conceptos asignados, tanto a nivel agente como a nivel escalafón, e ir insertando, de acuerdo al orden de cálculo, los ítems con los valores finales en una tabla detalle. Si son valores fijos, se insertan directamente, y para los conceptos que deban ser calculados se aplican las consultas asociadas para obtener sus valores. Una vez realizada la liquidación, los responsables del sector la verifican en base a la impresión de un listado con los detalles liquidados por agente, al que llaman "previo" (en el sistema existen opciones para obtener dicho listado por finalidad y programa o bien

imprimir la liquidación de un agente en particular). En caso de encontrarse errores, se realizan las modificaciones pertinentes y se lleva a cabo una nueva liquidación. Si todo está en orden, se genera la información necesaria para efectuar el pago correspondiente, así como también la información para la entidad bancaria, para que se realicen los depósitos en las cuentas bancarias de los agentes municipales.

## *Tipo de liquidación*

El tipo de liquidación tiene como finalidad dar una idea acerca de la liquidación en curso. Esto es útil para asociar conceptos específicos que deben estar presentes sólo para un tipo de liquidación determinado. Un ejemplo claro de esto es el concepto 249 - SAC, que sólo debe ser incluido cuando se realiza la liquidación del sueldo anual complementario. Justamente para eso, existe un tipo de liquidación llamado sueldo anual complementario, y hay dos liquidaciones adicionales en el año, una en junio y otra en diciembre, con este tipo de liquidación. Existen tres tipos de liquidaciones:

- Original o normal;
- Adicional, y
- Sueldo anual complementario.

## *Familiares del legajo – Asignaciones familiares*

Para cada legajo, existe la posibilidad de tener asociados sus familiares o miembros de su grupo familiar, cualquiera sea su grado de parentesco. Esta función es necesaria para el cálculo de aquellos conceptos que tienen que ver con las asignaciones familiares, ya que este tipo de conceptos se aplican o tienen como base de cálculo a los familiares del legajo. En este sentido, y en pos de permitir el cálculo, los familiares asociados deben detallar información como la que sigue:

- Tipo de familiar: esposo/a, hijo/a, menor a cargo o prenatal.
- Escolaridad: sin escolaridad, pre-escolar, primaria, media, superior.
- Discapacidad: sí/no.
- Fecha de nacimiento: es indispensable, por ejemplo, para determinar si se debe liquidar o no el concepto 151 – Hijo, ya que, para obtener el importe, debe consultarse si los familiares a cargo de tipo hijo tienen una edad inferior a 18 años.
- Apellido y nombre.
- DNI.
- Sexo.

## *Antecedente laboral*

La posibilidad de almacenar los antecedentes laborales del legajo es fundamental, ya que esto posibilita el cálculo de los conceptos relativos a la antigüedad del agente. Vale la pena destacar que, para que un antecedente laboral pueda ser tenido en cuenta para el cálculo de la antigüedad, es necesario que el legajo haya prestado servicios en algún

organismo público. Dentro de los datos que son fundamentales a la hora de realizar el cálculo, podemos consignar los siguientes:

- Cantidad de años trabajados.
- Cantidad de meses trabajados.
- Ámbito: municipal, provincial, o nacional.
- Organismo público: Municipalidad de General Belgrano, otra Municipalidad, Ministerio de Salud, Ministerio de Educación, Dirección de Vialidad Provincial (DVBA).
- Caja previsional: IPS, Anses, Caja Profesional.
- Fecha desde.
- Fecha hasta.

### *Cuenta bancaria del legajo*

Son las cajas de ahorro de tipo sueldo que el municipio mantiene abiertas para cada uno de sus agentes y donde deposita el importe correspondiente a los haberes luego de cada liquidación.

## Principales funciones

Del mismo modo que en el módulo de Ingresos, a continuación, detallaremos las principales funciones y herramientas provistas en el módulo, con el objetivo de mostrar un poco más en detalle el alcance de nuestra herramienta.

En cuanto al ABM de los legajos, se ocupa del ABM de sus familiares (con datos como tipo de familiar, tipo de escolaridad, y discapacidad), el de sus obras sociales, el de sus sindicatos, el de sus estudios (que se encarga del nivel de estudio, establecimiento educativo, título obtenido, fecha de recibido y matrícula, etc.), el de sus previsiones sociales (que, entre otros datos, maneja el encuadre previsional: general, docente, artístico, insalubre, concejal, anticipo jubilatorio, *ad honorem*, etc., la caja previsional, el beneficio previsional: jubilación, pensión, etc. y la caja otorgante), el de sus cuentas bancarias (las cuentas de tipo sueldo o salario a través de las cuales los agentes perciben sus haberes, que incluyen información sobre el número de cuenta, el número de tarjeta, la entidad bancaria, el tipo de cuenta y el tipo de tarjeta), el de sus antecedentes laborales (que se consideran al momento de liquidar el concepto **antigüedad** e incluyen datos como el período comprendido, el organismo público donde el agente prestó funciones y el ámbito, entre otros), y el ABM de sus pólizas (con datos como la compañía de ART, el número de póliza y la suma asegurada).

En relación al ABM de escalafones, dentro de las utilidades provistas, se incluye una que permite administrar los **conceptos** asignados a nivel escalafón, es decir, aquellos conceptos que serán liquidados de la misma manera a todos los agentes pertenecientes a dicho escalafón.

Este módulo también se ocupa del ABM de agentes. Como mencionamos en la descripción de las principales entidades, un agente municipal es la entidad que asocia un legajo a un escalafón en un período de tiempo determinado. Dentro de los datos que constituyen a un agente podemos mencionar su legajo, su escalafón, la fecha desde y hasta la cual ejerce, la función que desempeña dentro del escalafón al que pertenece, su tipo de acceso (ascenso, designación, antigüedad, concurso interno, otros), el tipo de relación laboral (permanente, destajista, jornalizado, temporal, contratado, etc.), su relación laboral (general, docente, insalubre, concejal, consejero escolar, etc.) y su modalidad de revista (personal de planta permanente, temporario, contratado, *ad honorem*, etc.).

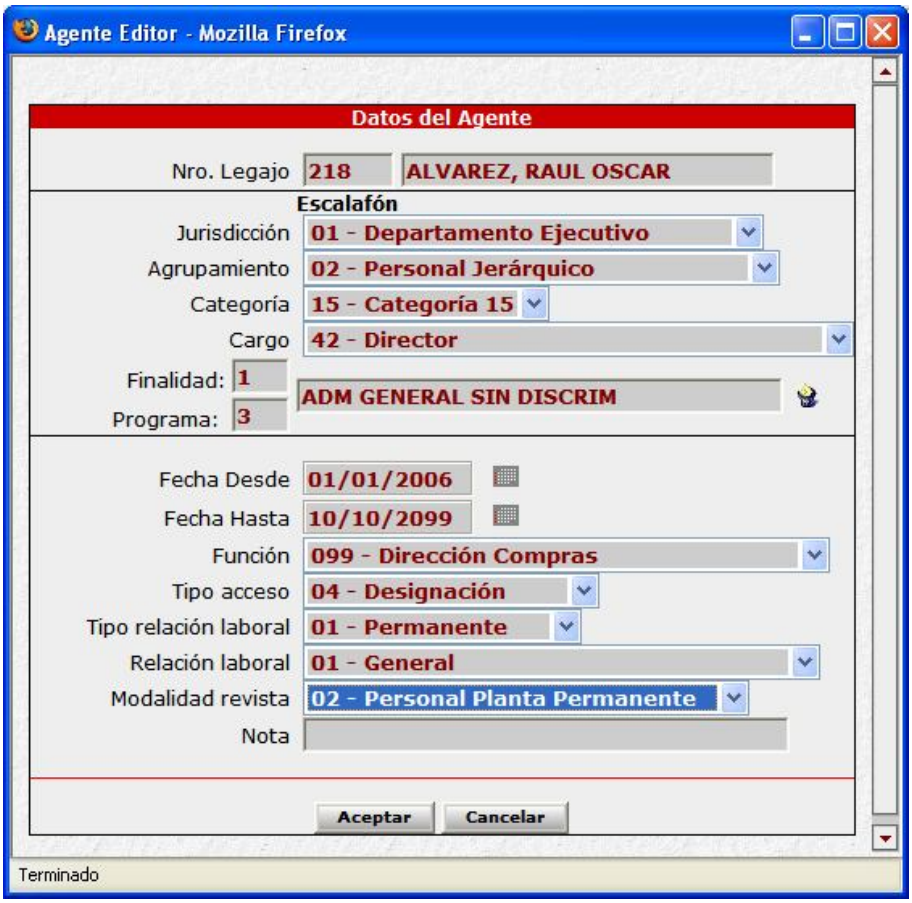



Figura 7. Editor de agentes

En cuanto al ABM de los conceptos utilizados en la liquidación de haberes, la funcionalidad contendrá la siguiente información:

- Código de concepto: un código único.
- Nombre: su descripción.
- Tipo de concepto: de haberes, de descuento, para cálculo, etc.
- Tipo de asignación: básico, descuentos, bonificación, aportes patronales, asignación familiar, bonificación no remunerativa, etc.

- Tipo de liquidación: para especificar si un concepto determinado sólo debe ser liquidado en determinado tipo de liquidaciones. Las opciones son: original, liquidación adicional y SAC.
- Partida presupuestaria: se aplica exclusivamente en conceptos de haberes, y especifica la partida presupuestaria de donde se sacará el dinero para el pago de los haberes. Por ejemplo, en el concepto "Cód. 22 – BONIFICACIÓN TITULO", el dinero se sacará de la partida 1.1.1.3.5.2 – Bonificación por título del programa al que pertenezca el escalafón del agente.
- Valor: en caso de que sea un concepto de valor fijo.
- Consulta: en caso de un concepto calculado en base a otros conceptos parámetro.
- Orden de cálculo: el orden en el que se calcularán los conceptos cuando se realice la liquidación de haberes. Esto se tiene en cuenta, fundamentalmente, cuando hay conceptos resultado del cálculo entre otros conceptos, que necesariamente deben ser obtenidos anteriormente.
- Tipo de dato: doble, entero, texto, etc. Esto también es utilizado en el cálculo de conceptos, fundamentalmente en aquellos que son utilizados como parámetros de entrada o salida de las consultas.

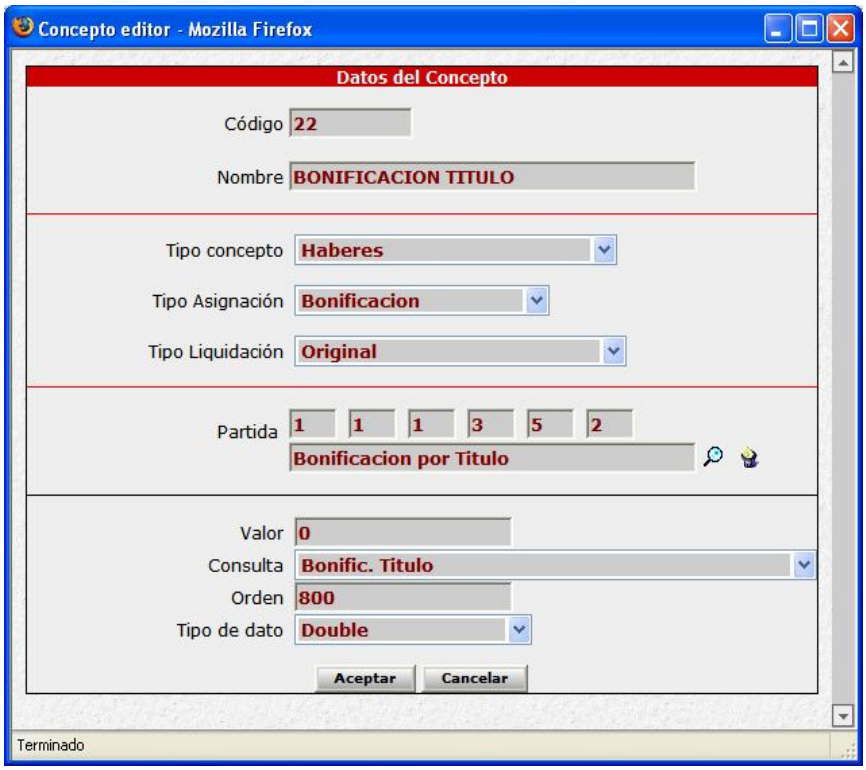



Figura 8. Editor de concepto

Este módulo se encarga del ABM de distintas entidades, la mayoría de las cuales es del tipo código/descripción, tales como agrupamientos, categorías, cargos, funciones, tipos de datos, tipos de relaciones laborales, sindicatos, establecimientos educativos, y profesiones.

En cuanto al ABM de liquidaciones, maneja datos como el tipo de liquidación, el período que comprende (fecha desde y fecha hasta), el estado de la liquidación (pendiente, abierta, liquidada y cerrada) y el ejercicio económico.

La herramienta para la liquidación de haberes permite la liquidación por legajo y por finalidad y programa y la liquidación de la totalidad de los legajos.

En cuanto a la carga de novedades, vale la pena destacar dos herramientas: la herramienta de **novedades del agente** y la herramienta de **carga masiva de novedades**. A continuación, daremos una breve descripción de ambas.

## *Novedades del agente*

Esta herramienta permite la carga o asociación de conceptos a un agente específico. Como mencionamos anteriormente, existen conceptos a nivel escalafón, es decir, comunes a todos los agentes que pertenecen al mismo escalafón, y otros que deben ser liquidados para un agente en particular (por ejemplo, la bonificación por título), o bien son conceptos de cálculo y tienen un valor particular para cada agente. Con esta herramienta, se posibilita la carga en este tipo de situaciones, a las que denominamos **novedades**.

## *Carga masiva de novedades*

Esta herramienta, al igual que la antes descripta, permite la carga de novedades a los agentes, pero con la diferencia de que facilita un tipo de interfaz más ágil para cargar conceptos en forma masiva. Hay situaciones en donde esta funcionalidad es de mucha utilidad, por ejemplo para la carga de horas extra previa a la liquidación de haberes, donde los jefes de departamento acercan las planillas con las horas extra laborales trabajadas en el período para ser liquidadas y la información debe ser incorporada al sistema en poco tiempo, para que se vea reflejada en la liquidación del mes.

En cuanto a la generación de archivos de información por las herramientas para la migración de información, se utilizan archivos de texto con un formato preestablecido por convenios, destinados a los organismos que los requieran, luego de cada liquidación de haberes. A continuación, brindamos una lista de los archivos generados por el sistema.

- Archivos para bancos (contienen, entre otras cosas, el detalle de los importes a depositar en las cuentas de los agentes).
- Archivos para ARTs.
- Archivos para IOMA.
- Archivos para IPS.
- Archivos de revista.

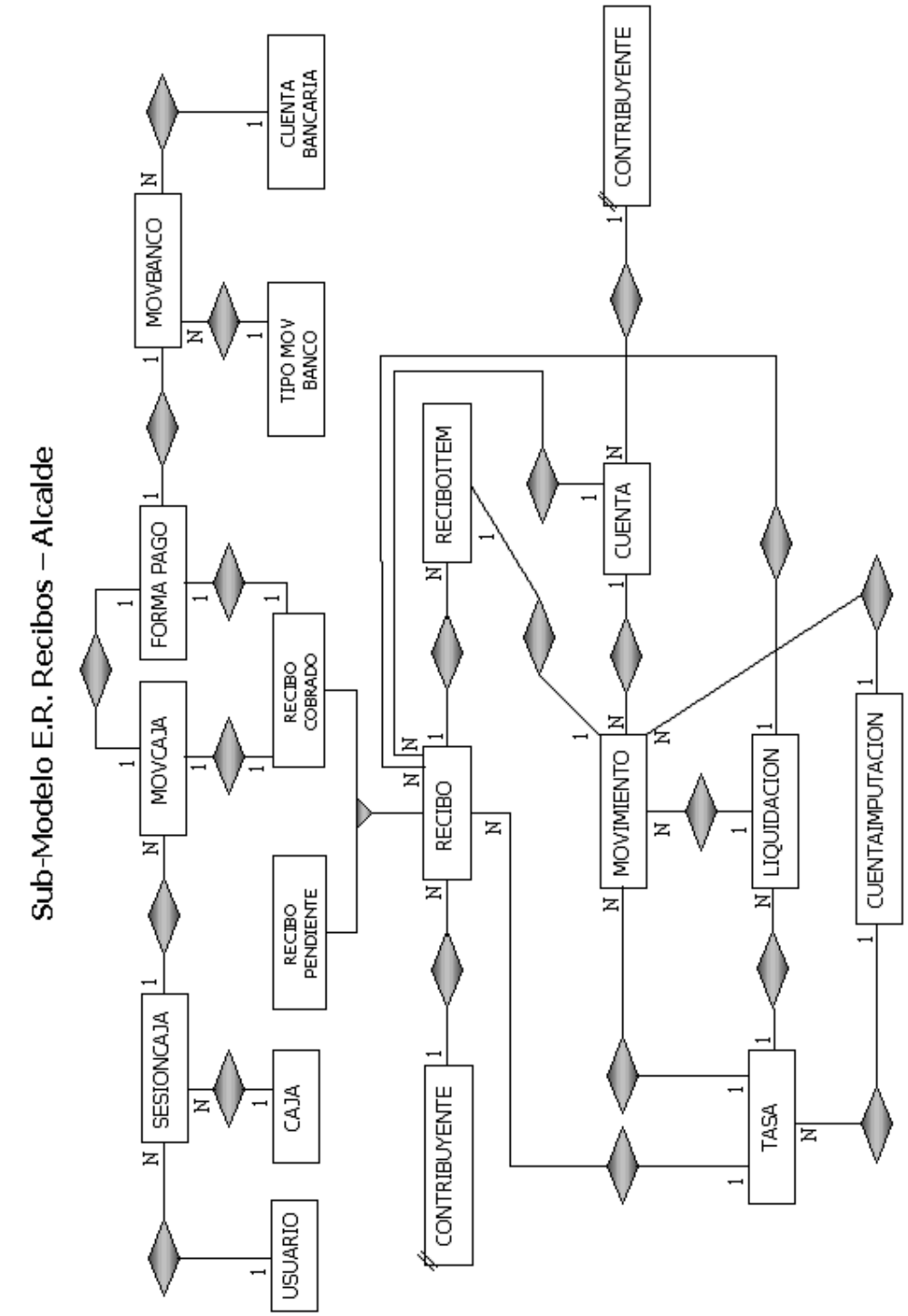
- Archivos de descuentos por tasas (dirigidos al sector Ingresos del municipio, detallan los importes retenidos a los agentes en concepto de tasas municipales).
- Archivos familiares.
- Archivos de afiliados.
- Archivos de salario.
- Archivos de datos del empleado.
- Archivos para RAFAM (un .txt generado con el formato especificado por el sistema RAFAM, para su importación desde este, a partir del cual se realizan los asientos contables correspondientes en concepto de la imputación presupuestaria de gastos por el pago de haberes).

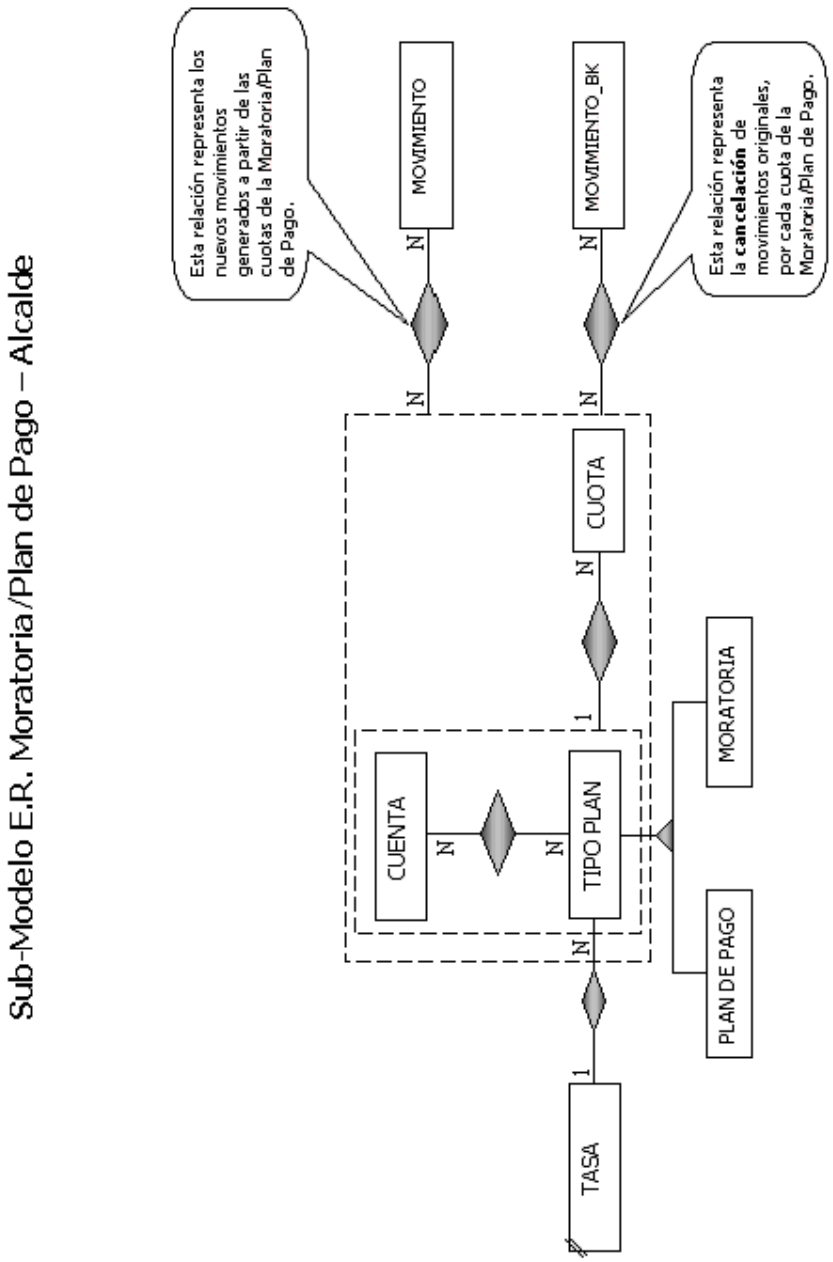
La lista que sigue detalla los listados y reportes generados por el módulo.

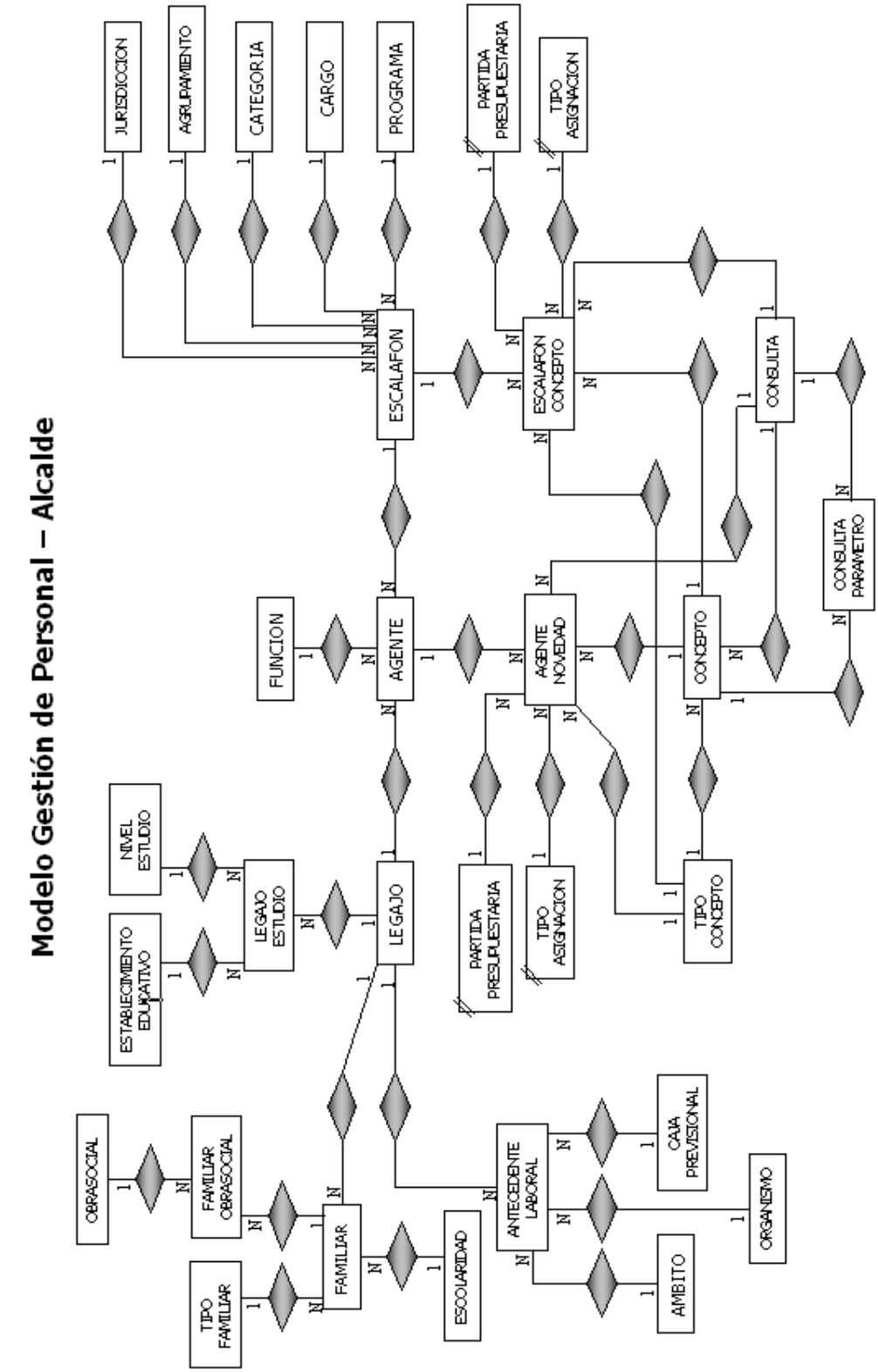
- Liquidación de haberes: lista el detalle de la liquidación de haberes ordenada por legajo para ser verificada por los jefes del sector de Personal, el Contador Municipal y demás funcionarios encargados de dar el visto bueno para el pago de la misma.
- Liquidación de haberes por finalidad y programa: es similar a la anterior, con la diferencia de que lista el detalle de los legajos pertenecientes al programa y la finalidad seleccionados.
- Recibo de haberes del legajo seleccionado.
- Recibos de una finalidad y programa seleccionados.
- Detalle de imputaciones de haberes: listado agrupado por finalidad, programa y partida presupuestaria, que sirve como complemento de la información generada en el archivo .txt para el sistema RAFAM.
- Listado detalle de descuentos de haberes: por concepto, es el soporte de la información enviada al sector Ingresos con el detalle de los importes retenidos a los agentes en la liquidación de haberes en concepto de descuentos por tasas liquidadas.
- Listado para bancos: al igual que el listado de imputación de haberes, este listado acompaña la información enviada al banco para la acreditación de los haberes.
- Listado para IOMA: tiene la misma función que el anterior.
- Listado para IPS: tiene la misma función que el anterior.
- Planilla de haberes IOMA-IPS.

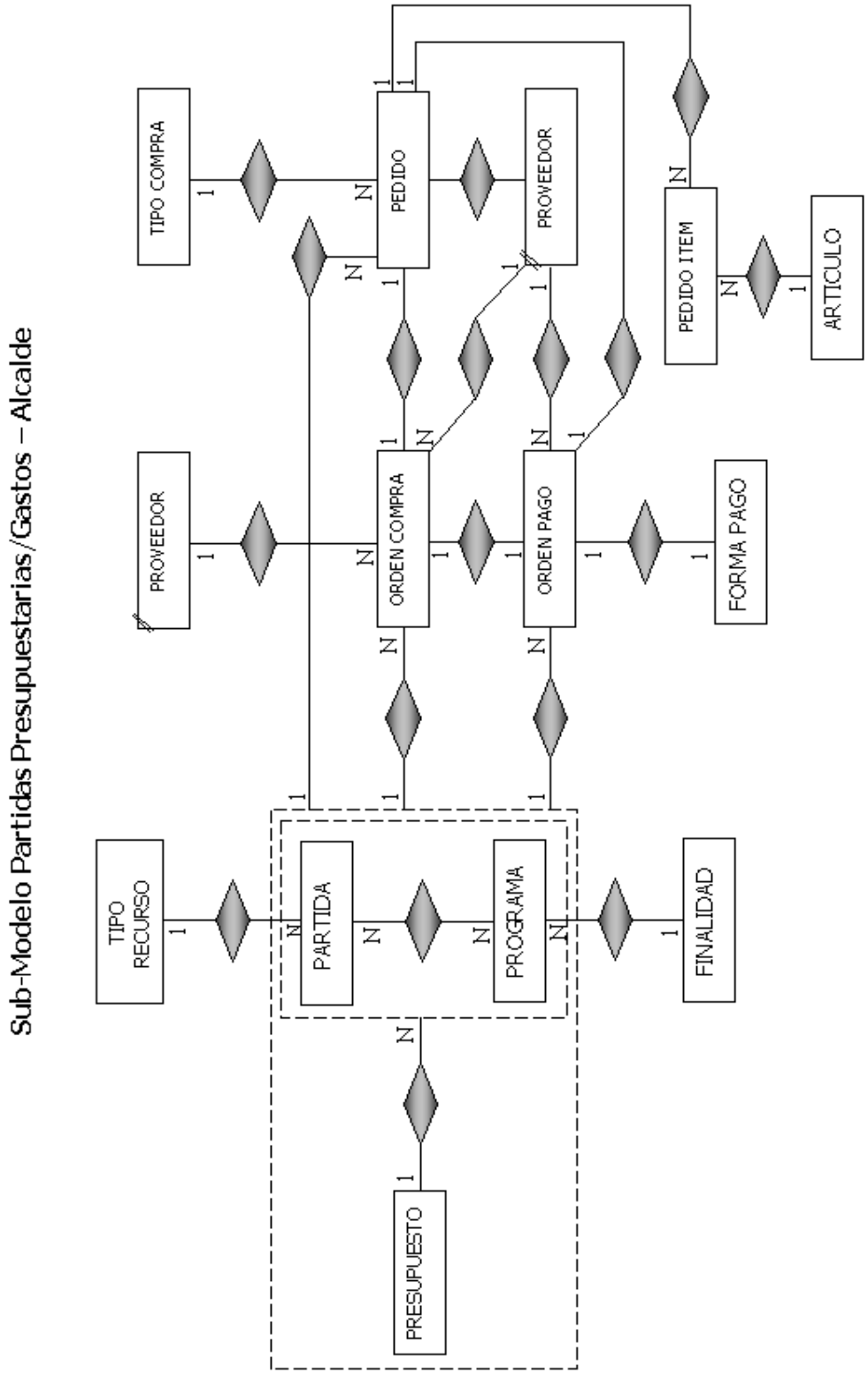
## Diseño del modelo de datos (Diagramas ER)

El modelo de datos se diseñó contemplando los cinco módulos, ya que es prácticamente imposible considerarlos en forma aislada. Los siguientes diagramas representan los sub-modelos más relevantes, es decir, son aquellas entidades (y sus correspondientes relaciones) que adquieren mayor importancia dentro del modelo y que se relacionan con los módulos elegidos para el desarrollo.


Sub-Modelo E.R. Recibos – Alcalde
CUENTA BANCARIA
MOVBANCO
TIPO MOV BANCO
FORMA PAGO
MOVCAJA
RECIBO COBRADO
RECIBO PENDIENTE
SESIONCAJA
CAJA
USUARIO
CONTRIBUYENTE
RECIBOITEM
RECIBO
CONTRIBUYENTE
CUENTA
MOVIMIENTO
LIQUIDACION
CUENTAIMPUTACION
TASA


Sub-Modelo E.R. Moratoria/Plan de Pago – Alcalde
Esta relación representa los nuevos movimientos generados a partir de las cuotas de la Moratoria/Plan de Pago.
MOVIMIENTO
MOVIMIENTO_BK
Esta relación representa la **cancelación** de movimientos originales, por cada cuota de la Moratoria/Plan de Pago.
CUOTA
CUENTA
TIPO PLAN
MORATORIA
PLAN DE PAGO
TASA
N
1


Modelo Gestión de Personal – Alcalde
JURISDICCION
AGRUPAMIENTO
CATEGORIA
CARGO
PROGRAMA
PARTIDA PRESUPUESTARIA
TIPO ASIGNACION
ESCALAFON
ESCALAFON CONCEPTO
CONSULTA
CONSULTA PARAMETRO
FUNCION
AGENTE
AGENTE NOVEDAD
CONCEPTO
TIPO CONCEPTO
NIVEL ESTUDIO
ESTABLECIMIENTO EDUCATIVO
LEGAJO ESTUDIO
LEGAJO
PARTIDA PRESUPUESTARIA
TIPO ASIGNACION
OBRASOCIAL
FAMILIAR OBRASOCIAL
TIPO FAMILIAR
FAMILIAR
ESCOLARIDAD
ANTECEDENTE LABORAL
CAJA PREVISIONAL
AMBITO
ORGANISMO


Sub-Modelo Partidas Presupuestarias/Gastos – Alcalde
TIPO COMPRA
PEDIDO
PROWEEDOR
PEDIDO ITEM
ARTICULO
PROWEEDOR
ORDEN COMPRA
ORDEN PAGO
FORMA PAGO
TIPO RECURSO
PARTIDA
PROGRAMA
FINALIDAD
PRESUPUESTO

## Diseño del modelo de objetos

De manera similar a lo que ocurrió con el diagrama de datos, se confeccionó el diagrama de objetos de todo el dominio y no sólo de los módulos elegidos, debido a que resulta muy difícil separar las entidades participantes en dichos módulos del resto de la aplicación.

A continuación, mostramos algunos casos dentro del modelo en los que fue necesario tomar ciertas decisiones de diseño para resolver los problemas que se nos fueron presentando.

### Utilización del patrón Singleton

Respecto de la necesidad de **automatizar** y **centralizar** la carga y configuración del contexto, así como también el manejo de las conexiones a la base de datos, decidimos crear una clase `alcalde.struts.plugins.Conector` que implementa la interfaz `org.apache.struts.action.PlugIn`. Este *plugin* se ejecuta al arrancar la aplicación y realiza la carga de la configuración del entorno de acuerdo a lo definido en el archivo de configuración. Dicha clase, además, define un Singleton para la clase `self.Self`, que es la encargada de manejar las conexiones a la base de datos valiéndose de las utilidades del *framework* (el paquete de utilidades del cual hablaremos más adelante, en otra sección). De esta manera, desde cualquier DAO de la aplicación, se puede obtener una conexión a la base de datos con sólo pedirle al *plugin* conector el objeto `Self`, y, una vez recuperado este objeto, solicitarle una conexión. Para entender mejor cómo funciona esto, daremos primero una definición del patrón Singleton y luego mostraremos nuestra implementación a través de la clase que mencionamos.

### *Patrón Singleton*

También conocido como Patrón de Instancia Única, es uno de los patrones de diseño más utilizados. Básicamente, es una clase que, implementada como lo especifica este sencillo patrón, posibilita que exista una única instancia de sí misma, ofreciendo un punto de acceso común a ella. Este patrón nos es útil en casos como el nuestro, cuando pretendemos acceder a un recurso específico en forma centralizada, asegurándonos de que exista únicamente una instancia de una determinada clase, y que, desde cualquier punto de la aplicación en el que queramos utilizar estos recursos, podamos garantizar que siempre se acceda a la misma instancia:

```
package singleton;

public class Singleton {
    private static Singleton aSingleton;

    private Singleton() {
    }
```

```
    static Singleton getInstance() {
        synchronized (Singleton.class) {
            if (Singleton.aSingleton == null) {
                  Singleton.aSingleton = new Singleton();
            }
        }
        return Singleton.aSingleton;
    }
}
```

En nuestra solución, este patrón está implementado, por ejemplo, en la clase `Conector`, la cual contiene un Singleton de la clase `Self` para que, desde cualquier DAO de la aplicación, pueda obtenerse la instancia de `Self` que contiene las configuraciones para acceder a los diferentes Datasources con las conexiones disponibles para los distintos roles.

```
public class Conector implements PlugIn {
...
private static Self self;
...
public static Self getSelf() {
    if (self == null) {
      self = new Self();
      self.setJndiJdbcDefault(Constantes.JDBC_Alcalde_SISTEMA);
      self.setSelfMonitor(monitor);
    }
    return self;
}//getSelf
...
}
```

Otra utilidad que hemos encontrado para este patrón es la implementación del mismo para manejar en forma centralizada el acceso a aquellos datos que son utilizados en toda la aplicación y a su vez es muy raro que cambien, como, por ejemplo, la colección de tipos de documento o la colección de estados civiles. Para esto, contamos con una clase que denominamos `CollectionProvider`, que implementa el patrón Singleton y provee a la aplicación de colecciones de datos como los que mencionamos. Esto lo podemos apreciar en las líneas de código que exponemos a continuación:

```
public class CollectionProvider {

  private Collection tiposDocumento;
  private Collection profesiones;
  private Collection paises;
  private Collection estadosCiviles;
  private Collection xx;
...
  private static CollectionProvider instancia = null;

  private CollectionProvider() {
  }

  public static CollectionProvider getIntance(){
    if (instancia == null) instancia = new CollectionProvider();
```

```
    return instancia;
  }
...
  /**
   * Recupera todos los tipos de documento.
   * @return
   */
  public Collection getTiposDocumento(SelfModule module, Self self)
      throws Exception {
    if (tiposDocumento == null)
      setTiposDocumento(ColeccionesManager.getTiposDocumento(module,
self));
    return tiposDocumento;
  }//getTiposDocumento
...
}
//Fin clase CollectionProvider
```

## Utilización de Reflection

Reflection es una herramienta o característica presente en el lenguaje de programación Java que nos permite examinar "introspectivamente" o modificar el comportamiento en tiempo de ejecución de objetos de aplicaciones que corren en una máquina virtual Java (JVM). Esta es una técnica poderosa que no está disponible en otros lenguajes de programación y permite, por ejemplo, retornar todos los métodos de una determinada clase Java, o bien instanciar una cierta clase, a partir de un literal que contiene su paquete y su nombre. Por lo tanto, y teniendo en cuenta que un buen diseño de objetos es una de las cuestiones más importantes cuando construimos una aplicación, hay veces en que se nos presentan determinados problemas que requieren soluciones como las que puede otorgar esta característica. Tal es el caso de nuestra solución, donde se nos presentó la problemática que describimos a continuación.

Las **cuentas corrientes** de los contribuyentes tienen como atributos principales un **contribuyente**, una **propiedad** y una **tasa**. Con el contribuyente no se presentan mayores dificultades, pero con la propiedad y la tasa nos encontramos con un problema que requiere solución, ya que son clases abstractas, por lo que necesariamente debemos conocer, al momento de instanciarlas, a qué subclase pertenecen. En la base de datos, la entidad en la que persisten dichos datos es la tabla cuenta, donde se guardan las referencias a una propiedad de algún tipo. Lo mismo ocurre con la tasa, de la cual se tiene una referencia pero no se dispone de otra información. En el momento de recuperar un objeto `Cuenta`, debemos conocer de qué propiedad (campo, lote, comercio, etc.) se trata y qué tipo de tasa tiene asociada (ABL, Red Vial, SeH, etc.), para poder crear las instancias correspondientes. Para responder a esto, en cada tabla donde se almacenan las distintas propiedades, se agregó un campo denominado `NombreClase`, donde se guarda el nombre completo, incluido el paquete que hace referencia a la clase en cuestión, (por ejemplo, la tabla campo) para todos sus registros, almacena por defecto el literal

`alcalde.ingresos.Campo`. Este se corresponde con la información del objeto del dominio `Campo`, que es una subclase del objeto del dominio `Propiedad`. Por lo tanto, en el momento de recuperar una cuenta, se basará en dicha información para crear la instancia en cuestión. Con este propósito, en la superclase `Propiedad` existe un método de clase que retorna una nueva instancia en función de un literal que llega como parámetro, el método `newPropiedad()`:

```
public abstract class Propiedad {
...
  public static Object newPropiedad(String nombreClase) throws Exception
{
    Class clase = Class.forName(nombreClase);
    return clase.newInstance();
  }//newPropiedad
...
}//Clase Propiedad
```

Algo similar ocurre con las tasas, donde en la tabla tasa de la base de datos también se definió un campo `NombreClase` que almacena la ruta completa del paquete que hace referencia al objeto del dominio específico para cada tasa. Así, el registro de la tabla que guarda información, por ejemplo, sobre la tasa ABL, contendrá en ese campo el valor `alcalde.ingresos.ABL`, que se corresponde con el objeto del dominio ABL, que pertenece a la jerarquía de clase `Tasa`. Del mismo modo que en la superclase `Propiedad`, la superclase `Tasa` tiene definido un método de clase llamado `newTasa()`, con la misma funcionalidad que el anteriormente descripto:

```
public abstract class Tasa {
...
  public static Object newTasa(String nombreClase) throws Exception {
    Class clase = Class.forName(nombreClase);
    return clase.newInstance();
  }//newTasa
...
}//Clase Tasa
```

Por consiguiente, el método encargado de recuperar una cuenta de un contribuyente incluye la siguiente porción de código, donde se puede apreciar la manera en la que se generaliza la instanciación de los objetos propiedad y tasa:

```
//***Load de la clase Cuenta***
public void load(SelfModule module, Self self) throws SQLException {
...
    Propiedad propiedad = null;
    Tasa tasa = null;

    try {
      query = self.getQuery(module, "PROPIEDADKEY_TO_NOMBRECLASE.sql");
     String nombreClase = IngresosManager.nombreClasePropiedad(module,
self, query, rs.getInt("PropiedadKey"));
      propiedad = (Propiedad)Propiedad.newPropiedad(nombreClase);
     propiedad.setPropiedadKey(rs.getInt("PropiedadKey"));
```

```
      propiedad.load(module, self);
      this.setPropiedad(propiedad);

      query = self.getQuery(module, "TASAKEY_TO_NOMBRECLASE.sql");
      nombreClase = IngresosManager.nombreClaseTasa(module, self, query,
rs.getInt("TasaKey"));
      tasa = (Tasa)Tasa.newTasa(nombreClase);
      tasa.setTasaKey(rs.getInt("TasaKey"));
      tasa.load(module, self);
      this.setTasa(tasa);
    }
    catch (Exception ex) {
      System.out.println("Error: Cuenta.load() - "+ex.getMessage());
      throw new SQLException();
    }
...
}
//***FIN Load de la clase Cuenta***
```

## Utilización de DAOs

La sigla DAO significa *Data Access Object*, que, como su nombre lo indica, son objetos cuya misión es permitir el acceso a los datos persistentes independientemente de la forma en que se encuentran almacenados. De ese modo, los DAOs permiten abstraer y encapsular los accesos a la fuente de datos, haciendo transparente para el resto de los objetos de la aplicación la manera en que se manejan las conexiones para almacenar y recuperar los datos.

La siguiente figura muestra el diagrama de clases que representa las relaciones para el patrón DAO:

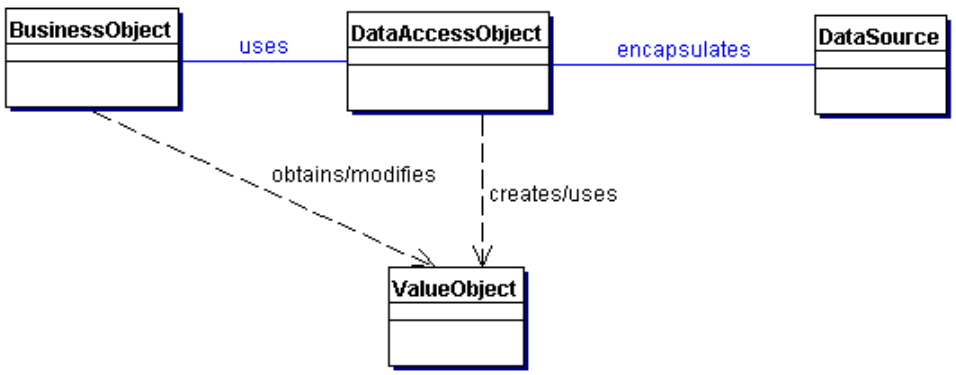



Figura 13. Diagrama de clases del patrón DAO

La figura que sigue muestra el diagrama de secuencia de la interacción entre los distintos participantes en este patrón:

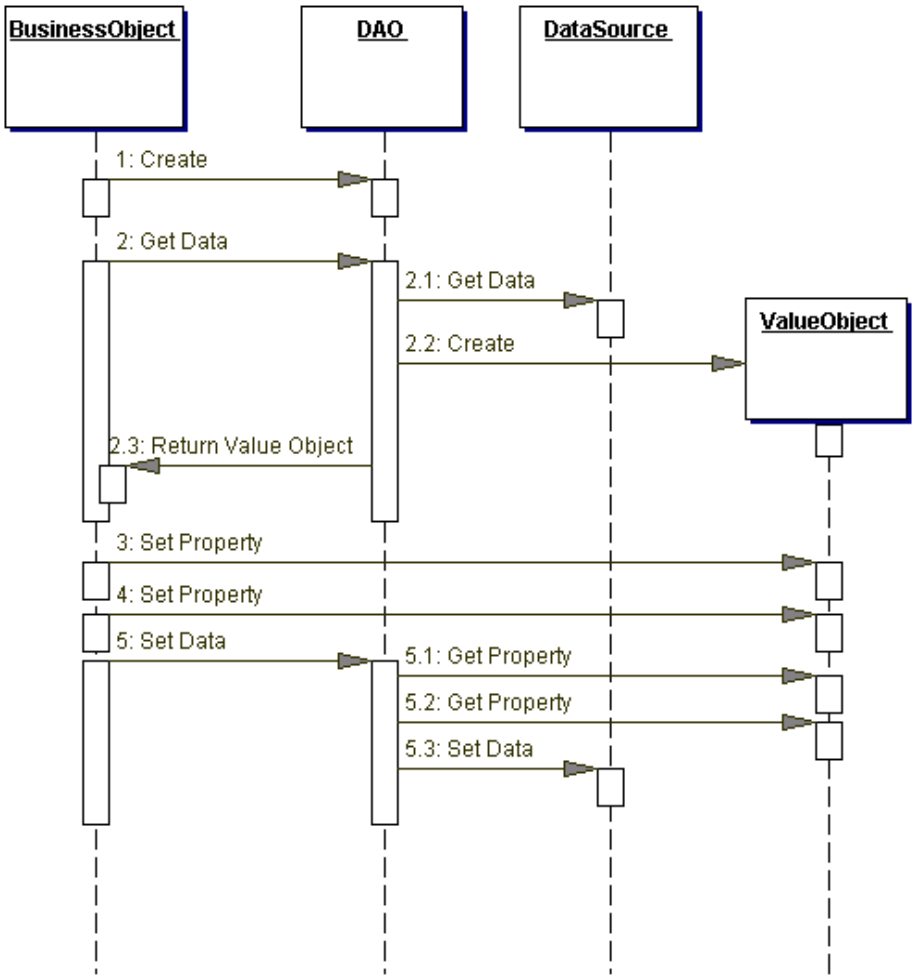



Figura 14. Secuencia de interacción para el patrón DAO

## *Ventajas de la utilización de DAOs*

- Permite la transparencia: los objetos de negocio pueden utilizar la fuente de datos sin conocer los detalles de implementación.
- Facilita la migración de datos: disponer de una capa de DAOs facilita a las aplicaciones la migración a una implementación de base de datos diferente.
- Reduce la complejidad de los objetos de negocio: todo el código relacionado con la implementación del acceso a datos (como, por ejemplo, las sentencias SQL) está dentro de los DAOs y no en los objetos de negocio.
- Centraliza todos los accesos a datos en una capa independiente: como todas las operaciones de acceso a los datos se han delegado a los DAOs, se puede ver como una capa que aísla el resto de la aplicación de la implementación de acceso a los datos.

## *Desventajas de la utilización de DAOs*

- Añade una capa extra: los DAOs crean un capa de objetos adicional entre el cliente y la fuente de datos que necesitamos diseñar e implementar para obtener los beneficios de este patrón. Pero, para obtener estos beneficios, debemos pagarlos con un esfuerzo adicional.
- No es útil para persistencia manejada por el contenedor: como el contenedor EJB maneja los *beans* de entidad con persistencia manejada por el contenedor (CMP), sirve automáticamente todo el acceso al almacenamiento persistente. Las aplicación que utilizan este tipo de *beans* no necesitan la capa DAO.

En nuestra solución, si bien no hemos respetado al pie de la letra lo que especifica el patrón, sí hemos decidido encapsular y delegar el acceso a los datos a los objetos del dominio, que conocen cómo guardarse, actualizarse y recuperarse. Para esto, todos los objetos del dominio implementan los métodos `save()`, `update()` y `load()` a través de los cuales respectivamente se guardan, actualizan y recuperan. De esta manera, desde los objetos de negocio, en nuestro caso los distintos objetos `Managers` como, por ejemplo, `IngresosManager` o `TasasManager`, cada vez que se necesita persistir los cambios de un objeto del dominio o recuperar sus datos, delegamos en él esta tarea. Como ejemplo de esto, podemos ver la implementación del método `load()` del objeto de dominio `Comercio`.

```
public class Comercio extends Propiedad {
…
  public void load(SelfModule module, Self self) throws SQLException {

    String query = self.getQuery(module, "PROPIEDADKEY_TO_COMERCIO.sql");
    Connection conn = self.getConnection();
    PreparedStatement sent = conn.prepareStatement(query);
    sent.setInt(1, this.getPropiedadKey());

    ResultSet rs = sent.executeQuery();

    //Se invoca al load() de la superclase para cargar los atributos
comunes
    super.load(module, self, rs);

    rs.beforeFirst();
    while (rs.next()) {
      this.setRazonSocial(rs.getString("RazonSocial"));
      this.setCantidadEmpleados(rs.getInt("CantidadEmpleados"));
      this.setDatosLocal(rs.getString("DatosLocal"));
      this.setDomicilioComercial(rs.getString("DomicilioComercial"));
      this.setDomicilioLegal(rs.getString("DomicilioLegal"));
      this.setFechaInicioActividad(rs.getDate("FechaInicioActividad"));

     this.setPorcentajeOcupacionLote(rs.getDouble("PorcentajeOcupacionLo
     te"));
      this.setSuperficieLocal(rs.getDouble("SuperficieLocal"));
```

```
        //Cargar la actividad principal
        this.setActividadPrincipal(new Actividad());
        this.getActividadPrincipal().setActividadKey(
             rs.getInt("ActividadPrincipalKey"));
        this.getActividadPrincipal().load(module, self);

        //Cargar el lote
        this.setLote(new Lote());
        this.getLote().setPropiedadKey(rs.getInt("LoteKey"));
        this.getLote().load(module, self);

        //Cargar el estado
        this.setEstadoComercio(new EstadoComercio());
       this.getEstadoComercio().setEstadoComercioKey(
                                    rs.getInt("EstadoComercioKey"));
        this.getEstadoComercio().load(module, self);

        this.setFechaBaja(rs.getDate("FechaBaja"));
      }//while

      rs.close();
      sent.close();
    }//load()

    public void save(SelfModule module, Self self) throws SQLException{
        …
    }//save()

    public void update(SelfModule module, Self self) throws SQLException{
        …
    }//update()
…
}//FIN clase Comercio
```

## Utilización de DTOs

*Data Transfer Objects* (DTO), también conocido como *Value Objects* (VO), es un patrón de diseño utilizado como solución al problema de intercambio de datos entre las diferentes capas de una aplicación *enterprise*.

En una aplicación *enterprise*, los clientes suelen solicitar valores que son algo más que un atributo simple, pero, a su vez, es raro que necesiten toda la información y comportamiento disponibles en los objetos del dominio, y además, las llamadas recurrentes que los clientes realizan impactan en el tráfico de la red, degradándola. Los DTOs ofrecen una buena solución a esta problemática, ya que como usualmente los clientes pueden necesitar más de un valor para un determinado objeto del dominio, se pueden construir tantos DTOs como se necesiten, en función de la porción de datos que se desee transportar en cada caso, adecuándolos así a las necesidades de los clientes.

La siguiente figura muestra el diagrama de clases que representa el patrón *Data Transfer Object* en su forma más simple:

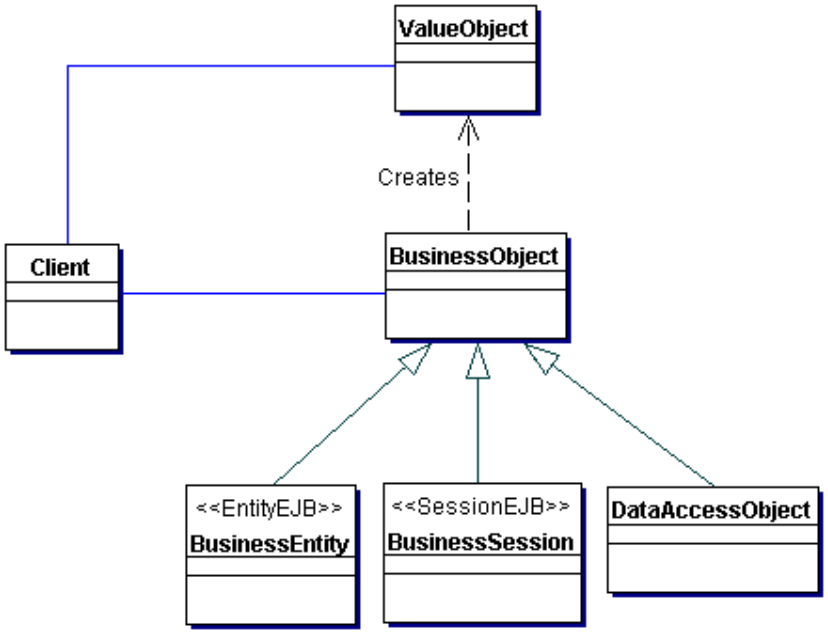



Figura 15. Diagrama de clases del patrón DTO

La figura que sigue contiene el diagrama de secuencia que muestra las interacciones del patrón *Data Transfer Object*:

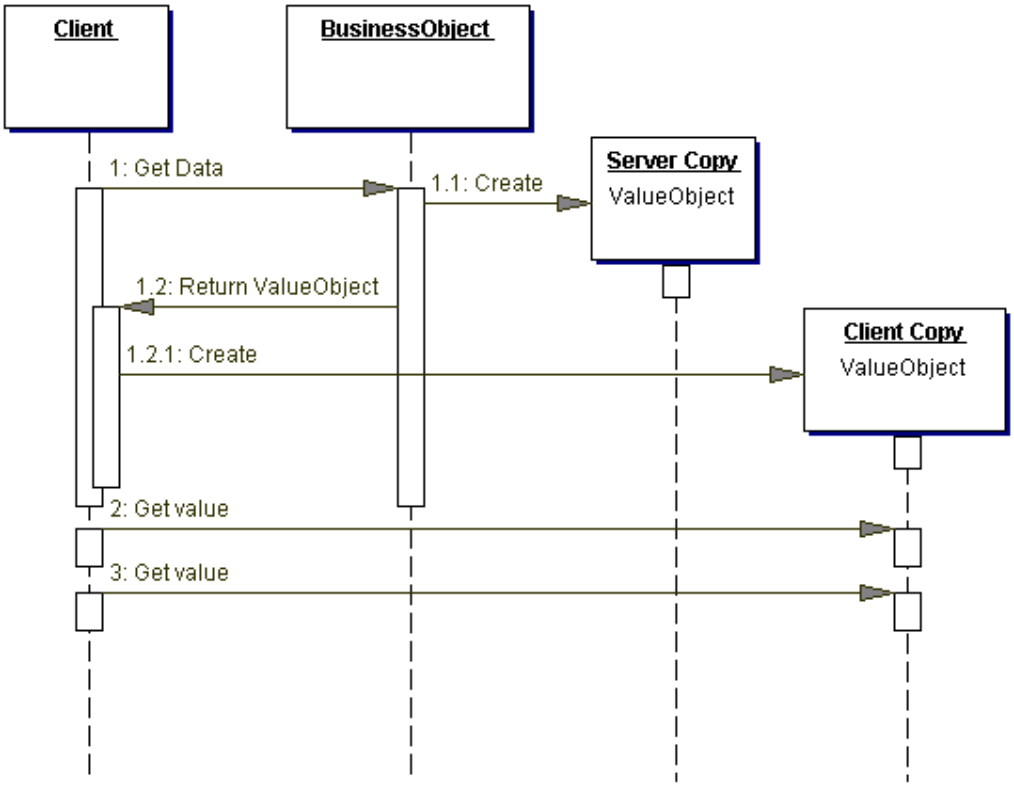



Figura 14. Secuencia de interacción para el patrón DTO

En nuestra solución, los DTOs han sido de mucha utilidad. Un claro ejemplo de esto es su utilización para el transporte de los datos en la impresión de reportes. Este es un caso ideal ya que, por lo general, para la impresión de un listado o un reporte de cualquier tipo, se necesitan los datos formateados de cierta manera, que raramente coincidirá exactamente con la información contenida en un objeto del dominio. Además, como hemos elegido la librería provista por Jasper Report para la generación de reportes, la arquitectura propicia aún más la utilización de DTOs, ya que uno de los mecanismos mediante el cual se le envían los datos para la impresión es a través de una colección de objetos, y qué otra cosa se adapta mejor que un DTO para transportar esos datos a mostrar.

Como ejemplo de utilización de DTOs para la impresión de reportes a través de Jasper Report, podemos mostrar la forma en la que se obtienen y transportan los recibos para poder imprimirlos:

```
<!-- Del lado del cliente. En colMovs se retorna la colección con los
DTOs que se trasladan para la impresión del recibo.
    …
     ArrayList colMovs = (java.util.ArrayList)IngresosManager.getRecibo(
          self.getModule(Constantes.MODULO_INGRESOS), self,
          tipoRecibo, idRecibo);
```

```
      JasperPrint jp = JasperFillManager.fillReport(jr, parameters,
          new JRBeanCollectionDataSource(colMovs));
    …
 -->
<!-- Del lado de la capa de Negocio. Métodos que retornan la colección de
       ítems del recibo.
  public static Collection getRecibo(SelfModule module, Self self,
          int tipoRecibo, int idRecibo) throws SQLException {
    Collection col = null;

    switch (tipoRecibo) {
        case IngresosManager.TIPO_RECIBO_TASA: col =
                getReciboTasa(module, self, idRecibo);
        break;
        case IngresosManager.TIPO_RECIBO_SERVICIOCONTRIBUYENTE: col =
                getReciboServicioContribuyente(module, self, idRecibo);
        break;
        case IngresosManager.TIPO_RECIBO_INGRESOTERCERO: col =
                getReciboIngresoTercero(module, self, idRecibo);
        break;
      default:
        break;
    }//switch

      return col;
  }//getRecibo
  public static Collection getReciboTasa(SelfModule module, Self self,
                        int reciboKey) throws SQLException {
    ArrayList col = new ArrayList();
    ReciboGenericoPrintDTO reciboDTO = null;
    ...
    while (rs.next()) {
      reciboDTO = new ReciboGenericoPrintDTO();

      reciboDTO.setNumeroRecibo(rs.getInt("Numero"));
      reciboDTO.setCodigoBarras(rs.getString("codigoBarras"));
      reciboDTO.setContribuyente(rs.getString("contribuyente"));
     reciboDTO.setDomicilioContribuyente(rs.getString("domicilioContribu
     yente"));
      reciboDTO.setLocalidad(rs.getString("localidad"));
      reciboDTO.setXX(...);
     ...
     col.add(reciboDTO);
    }//while

    ...
    return col;
  }//getReciboTasa
-->
<!-- / Definición del DTO ReciboGenericoPrintDTO.
package alcalde.reportes;

public class ReciboGenericoPrintDTO {
```

```
  private String codigoBarras;
  private int nroCuenta;
  private String contribuyente;
  private String domicilioContribuyente;
  private String codigoPostal;
  private double montoItem;
  private String descripcionItem;
  private double importeTotal;
  private String importeTotalLetras;
...
}
-->
```

## Utilización de jerarquía de clases

Cuando diseñamos utilizando el paradigma orientado a objetos, el elemento fundamental es, precisamente, el objeto. Una breve definición podría ser que un objeto es un conjunto complejo de datos y programas que poseen estructura y forman parte de una organización. Esta definición especifica varias propiedades importantes de los objetos. En primer lugar, un objeto no es un dato simple, sino que contiene en su interior cierto número de componentes bien estructurados. En segundo lugar, cada objeto no es un ente aislado, sino que forma parte de una organización jerárquica, en el sentido de que ciertos objetos son superiores a otros objetos de alguna forma. De este modo, así como el objeto es el elemento fundamental, la jerarquía de clases es la base a partir de la cual se construye la Programación Orientada a Objetos (POO).

Existen varios tipos de jerarquías, las hay simples y complejas, y serán simples cuando puedan ser representadas en forma de árbol. En cualquier caso, sea la estructura simple o compleja, podrán distinguirse en ella tres niveles básicos: la raíz de la jerarquía, los objetos o clases intermedias y los objetos o clases terminales. Este concepto de jerarquía es sumamente importante, ya que nos demuestra la importancia de dividir los problemas en una jerarquía de ideas. El tipo más relevante de jerarquía es la de generalización o especialización, basada en que las propiedades de una categoría se transmiten a todas las categorías que se especializan o subcategorías. Si a esta idea de especializar o jerarquizar entidades le sumamos otros conceptos que tienen lugar gracias a ella, tales como el de **herencia** y el de **polimorfismo**, estamos frente a una herramienta muy útil a la hora de modelar nuestras aplicaciones. En referencia a esto último, debemos mencionar las características de ambos conceptos para comprender de qué se trata:

## *Herencia*

Es una característica esencial, puesto que permite heredar a las clases atributos y comportamiento de una o varias clases denominadas **base** o **superclase**. Las clases que heredan de clases base se denominan **derivadas** y extienden su funcionalidad, lo que les permite ser clases bases o superclases para otras clases derivadas. La herencia ofrece una ventaja importante, ya que permite la reutilización de código.

## *Polimorfismo*

Se denomina polimorfismo a la capacidad que tienen los objetos de una clase de responder al mismo mensaje o evento en función de los parámetros utilizados durante su invocación. Un objeto polimórfico es una entidad que puede contener valores de diferentes tipos durante la ejecución del programa. En la práctica, esto quiere decir que un puntero a un tipo de objeto puede contener varios tipos diferentes, no solo el creado. De esta forma podemos tener un puntero a un objeto de la clase `Propiedad`, pero este puntero puede estar apuntando a un objeto subclase de la anterior como podría ser `Comercio`, `Lote` o `Campo` (todas ellas son subclase de `Propiedad`).

La jerarquía de clases sirve para resolver problemas de diversa índole, que van desde soluciones complejas a situaciones tan simples (que se simplifican a partir de estos conceptos) como, por ejemplo, la recuperación de la información de la descripción de una propiedad en la impresión de un recibo. Retomemos el ejemplo de la impresión de los recibos. Habíamos mostrado que, en la capa de negocios, cuando se recuperaba la información del recibo a imprimir, los datos se cargaban en un DTO de la clase `ReciboGenericoPrintDTO`. En el método que realiza esta tarea, se encuentra la porción de código que se encarga de configurar en el DTO el valor de la descripción de la propiedad en cuestión, independientemente del tipo de propiedad de la que se trate. Esto es posible gracias a la jerarquía de clases y a los conceptos de herencia y polimorfismo:

```
<!– Método que recupera los datos del recibo y los retorna en un DTO
      …
      try {
        Propiedad propiedad =
            (Propiedad)Propiedad.newPropiedad(rs.getString("NombreClase"))
      ;
         propiedad.setPropiedadKey(rs.getInt("PropiedadKey"));
         propiedad.load(module, self);

         reciboDTO.setPropiedad(propiedad.getDescripcion());
       }//try
       catch (Exception ex) {
         reciboDTO.setPropiedad("");
       }//catch
      …
  –->
```

El método `getDescripcion()` se encuentra definido en la superclase `Propiedad` y reimplementado o redefinido en cada una de las subclases o clases derivadas. De este modo, se delega en cada una de ellas (`Campo`, `Lote`, `Comercio`, etc.) la manera en que retornan su descripción, ya que saben cómo hacerlo en función de los atributos que las componen, algo que nos ahorra trabajo y simplifica el código escrito.

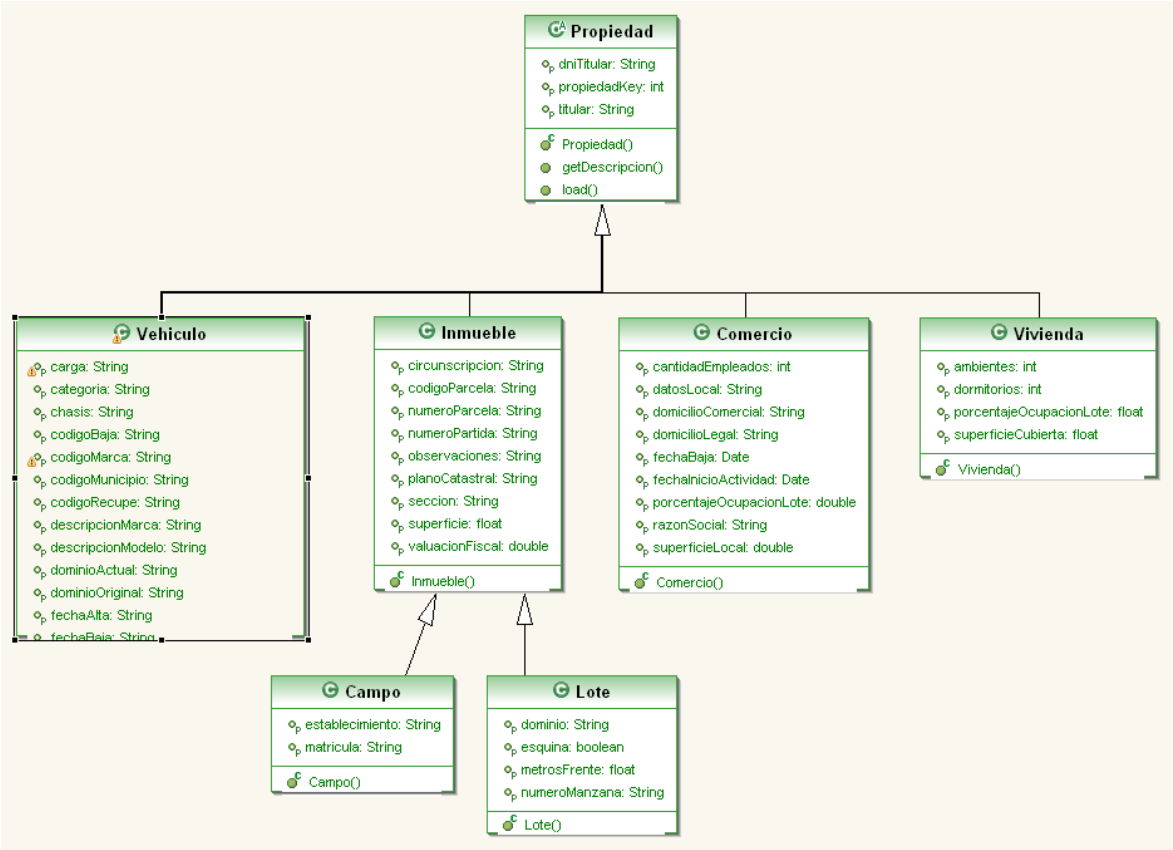



Figura 15. Jerarquía de clases para la superclase `Propiedad`

## Utilización de interfaces

El concepto de interfaz lleva un paso más adelante la idea de las clases abstractas. En Java, una interfaz es una clase abstracta pura, es decir, una clase cuyos métodos son abstractos (ninguno se implementa). La interfaz permite al diseñador establecer el molde o estructura que, como mínimo, deberá contener las clases que la implementen (nombres de métodos que se deben implementar, listas de argumentos que deberán contener dichos métodos, tipos de retorno, etc.). Una interfaz sirve para establecer un "protocolo" entre clases.

Para indicar que una clase debe implementar los métodos de una interfaz determinada, se utiliza la palabra reservada *implements*. Con esto, el compilador se encargará de que la

clase efectivamente defina e implemente todos los métodos de la interfaz. Una clase puede implementar más de una interfaz.

## *Declaración y uso*

```
public interfaz DocumentoIngEgr {

  public int getDocumentoKey() throws RemoteException;
  public void setDocumentoKey(int documentoKey) throws RemoteException;
  public void setNroDocumento(int nroDocumento) throws RemoteException;
  public int getNroDocumento() throws RemoteException;
  public void load(SelfModule module, Self self) throws SQLException;
  public void load(Connection conn, SelfModule module, Self self) throws
      SQLException;
}
```

Luego, las clases que deben implementar esta interfaz se definen de la siguiente manera y, como dijimos, están obligadas a implementar los métodos especificados:

```
public class MovBanco implements DocumentoIngEgr, ItemFormaPago {
  ...
  public int getDocumentoKey() {
    return this.getMovBancoKey();
  }
  ...
}
public abstract class Recibo implements DocumentoIngEgr {
 ...
  public int getDocumentoKey() {
    return this.getReciboKey();
  }
 ...
}
```

A modo de resumen, podemos decir:

- Una interfaz define un protocolo de comunicación entre dos objetos.
- Una definición de una interfaz contiene declaraciones de métodos pero no implementaciones, y podría contener también definiciones de constantes.
- Una clase que implementa una determinada interfaz debe implementar todos los métodos declarados en ella.
- Un nombre de interfaz puede ser usado en cualquier lugar, de manera similar a un tipo.

A continuación, mostramos una serie de diagramas que muestran las diferentes interfaces y su implementación en nuestra solución:

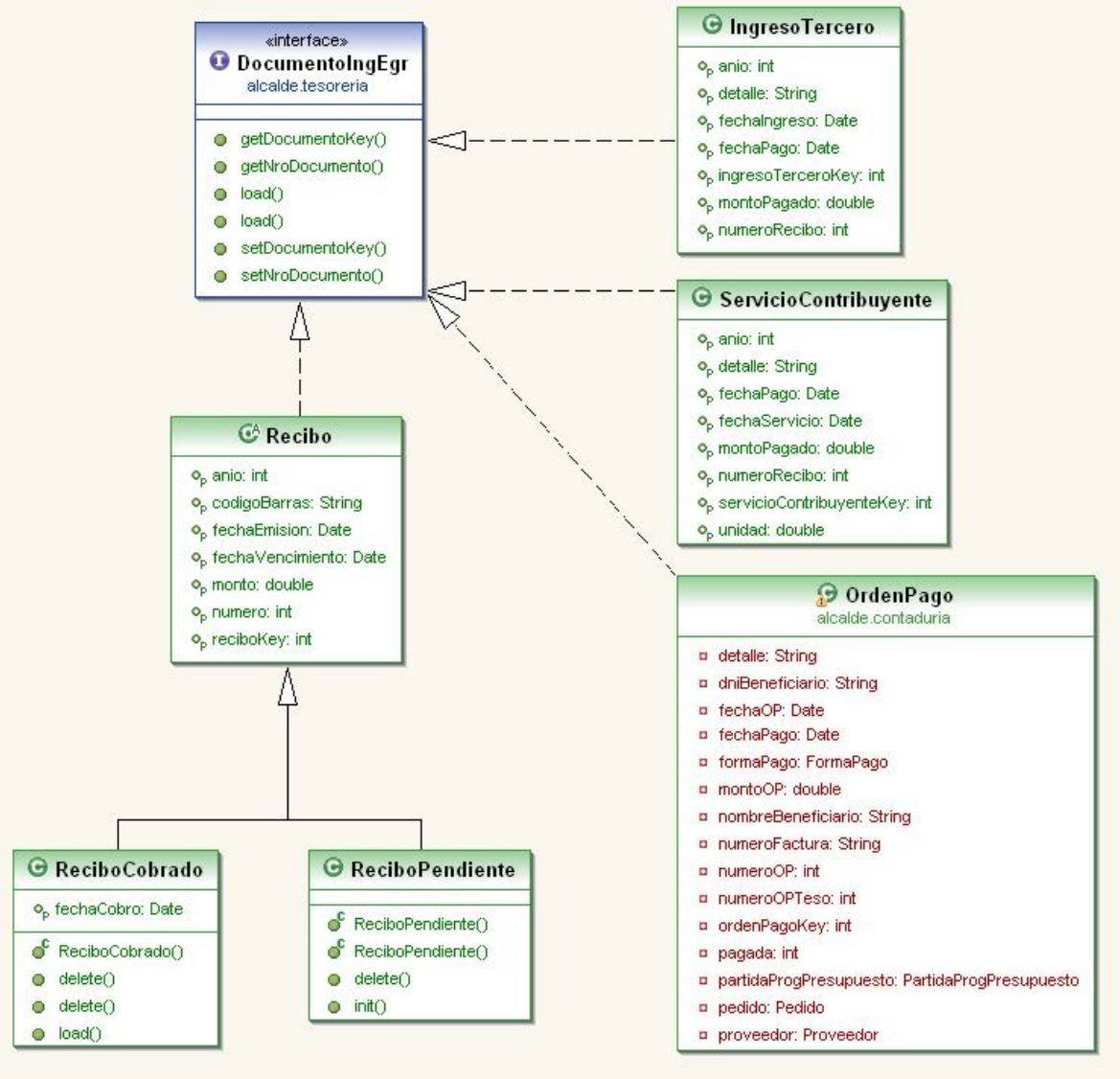



Figura 16. Las entidades de documentos de pago (ingreso y egreso) implementan la interfaz `DocumentoIngEgr`

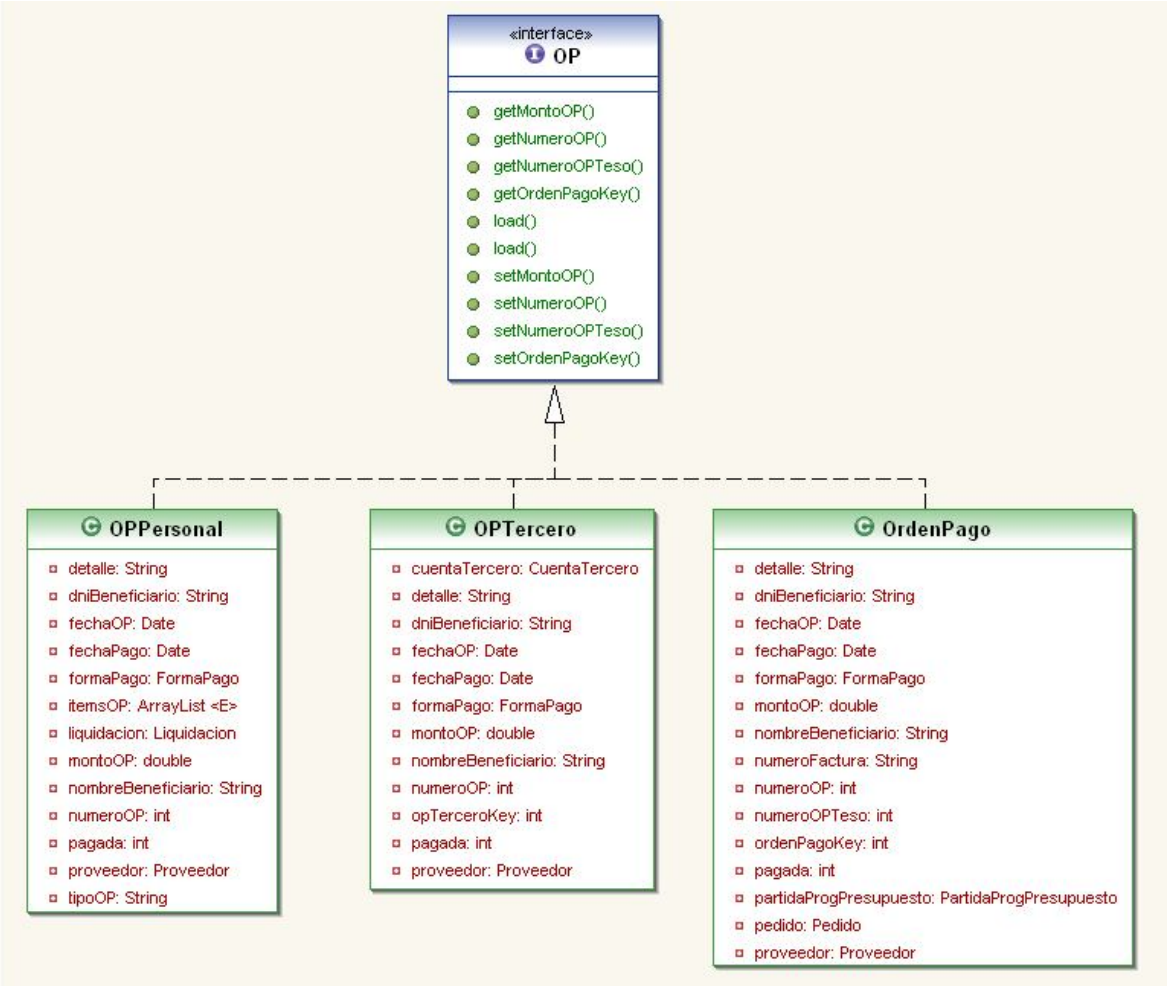



Figura 17. Definición de la implementación de la interfaz OP por medio de las clases de diferentes tipos de órdenes de pago

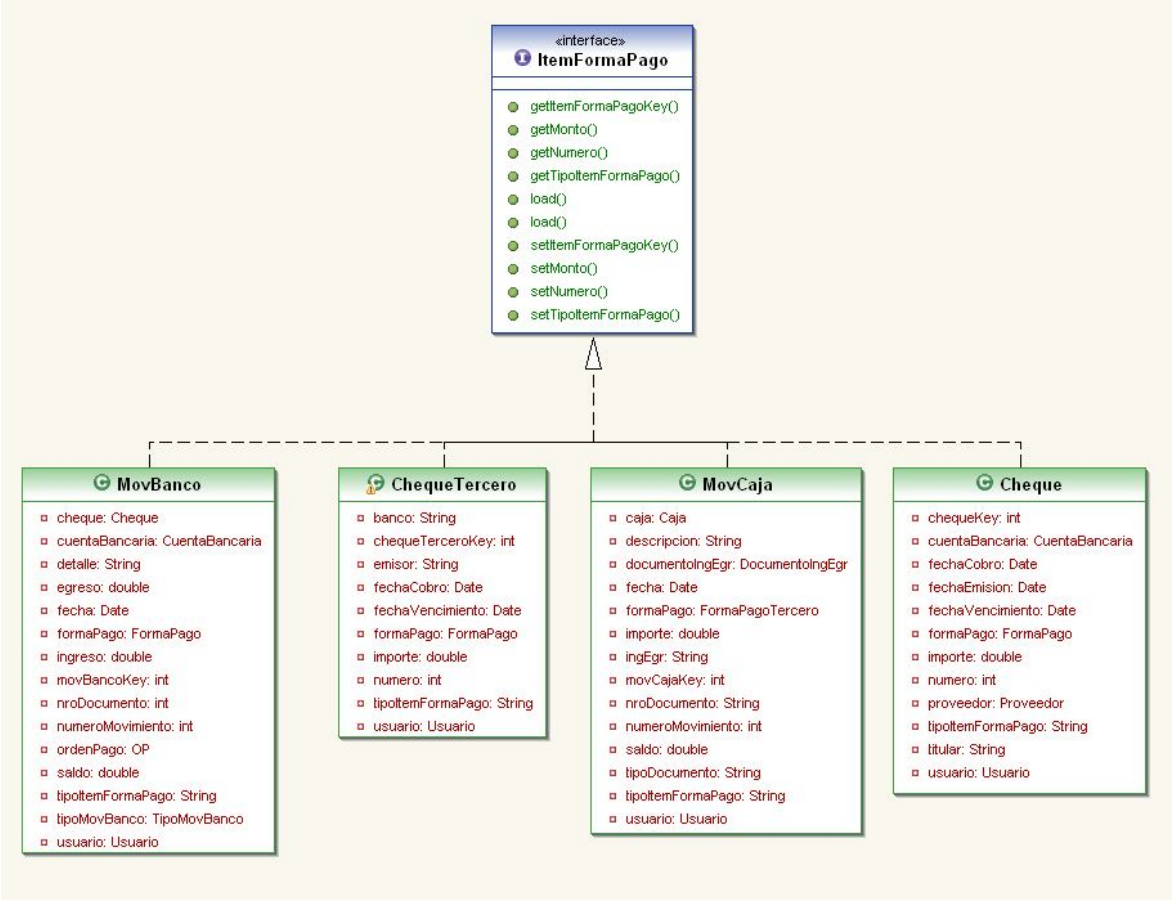



Figura 18. Implementación de la interfaz `ItemFormaPago` por las clases que pueden ser incluidas en una forma de pago

# Definición de la arquitectura

## Características funcionales

Para la implementación de la solución, decidimos optar por un esquema *web* en varias capas, totalmente independiente del sistema operativo en el que corra (es decir, multiplataforma) e independiente también del motor de base de datos que se utilice para almacenar la información. La elección de un esquema de estas características trae a su vez beneficios como conectividad con los clientes extremadamente delgados (clientes *web*, en particular, navegadores de Internet), sin la necesidad de agregados ni conectores especiales adicionales.

La integración con otros sistemas *legacy* puede ser implementada a través de servicios *web*.

El esquema propuesto está graficado en la siguiente figura:

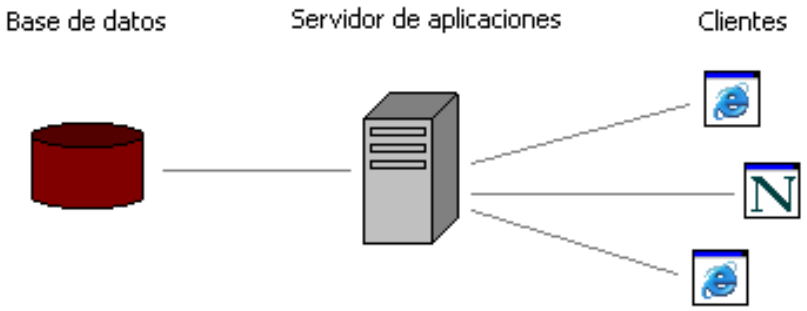



Figura 19. Esquema propuesto

## Servidor de aplicaciones

En cuanto al servidor de aplicaciones, es recomendable utilizar aquellas soluciones que cuenten con la implementación de todos, o al menos la mayoría, de los servicios del paquete de servicios de JEE. Otro requisito que consideramos excluyente es que el producto a utilizar se corresponda con un proyecto de código abierto, al menos en alguna de sus versiones, sostenido y apoyado por una red mundial de colaboradores. Dos proyectos que cumplen perfectamente con estos requisitos son **Apache Tomcat** y **JBoss**, que, por tener la particularidad de estar implementados en Java puro, son multiplataforma y pueden ser instalados en prácticamente cualquier sistema operativo. A propósito de estos servidores, vale la pena hacer una distinción entre ambos, ya que Apache Tomcat funciona sólo como contenedor de *servlets*, mientras que JBoss, además de ser contenedor de *servlets* (implementado a través del mismo Tomcat), cuenta además con la implementación de los servicios necesarios para ser un contenedor de EJBs.

Dicho servidor de aplicaciones realiza toda la lógica de la aplicación, generando páginas que se visualizan en los clientes. Se compone de dos partes: el servidor *web*, que provee el acceso a la información estática, y el contenedor de *servlets*, que genera la información dinámica de la aplicación.

## Base de datos

El sistema está desarrollado sobre bases de datos relacionales. La selección del motor de base de datos a utilizar puede llevarse a cabo en la etapa de implementación de la solución, ya que el desarrollo del *software* es independiente de la base de datos, por lo cual, no se impone ninguna restricción al respecto.

## Clientes

Se utilizan clientes livianos como Internet Explorer 5.5 o superior o Mozilla Firefox, y no se requiere ningún tipo de instalación adicional, como clientes de base de datos o *plugins*. Esto permite la utilización del sistema desde cualquier PC con cualquier sistema operativo, desde cualquier lugar del municipio o incluso desde Internet.

### Arquitectura de desarrollo

El sistema está desarrollado en su totalidad usando el paradigma orientado a objetos con una arquitectura lógica de varias capas, y tecnología Java JEE. Como servidor de aplicaciones, utilizamos Apache Tomcat, debido a que es libre y que, como está desarrollado íntegramente en Java, corre bajo cualquier sistema operativo, por lo que cumple con los requerimientos detallados anteriormente. La decisión responde también a que nuestra aplicación, al no implementar EJBs, no necesita un contenedor de EJBs ni servicios adicionales, en cuyo caso la decisión hubiera estado inclinada hacia JBoss. Además, en la actualidad, Tomcat está siendo utilizado en entornos productivos con importantes niveles de tráfico y alta disponibilidad.

Se utiliza el *framework* Struts como implementación del *pattern* MVC.

En la capa de persistencia, el acceso a la base de datos se realiza por medio del estándar JDBC a través de *drivers* de tipo 4, que se comunican directamente con la base de datos. Las sentencias mediante las cuales se hace transacción con la base de datos, tanto de consulta como de inserción o actualización, se almacenan en el *file system* separadas por módulos (directorios del sistema operativo).

El control de acceso a la aplicación se realiza a través de usuarios almacenados en la base de datos. Estos usuarios tienen asignados **roles**, y las **acciones** disponibles para ellos se otorgan a nivel de **rol**.

Para la presentación, utilizamos páginas JSP que se encargan de mostrar la información dinámica a los clientes. Además, a través de estas páginas, se interactúa con el usuario para capturar el ingreso de datos, realizar validaciones simples y presentar la información.

Los reportes de la aplicación están implementados a través de Jasper Reports e iText para generación de archivos PDF. Cabe aclarar que ninguna herramienta utilizada requiere licencias de desarrollo o de uso.

## Presentación de tecnologías y estándares involucrados

En esta sección, presentaremos todo lo relacionado con las tecnologías utilizadas. Aquí, brindaremos detalles sobre cómo utilizamos ANT para la preparación de los *deployments* según el ambiente (desarrollo, *test*, producción), y la manera en la que implementamos la construcción del menú del usuario en forma dinámica valiéndonos de DOJO Toolkit.

En este punto, vale la pena mencionar que muchos de los servicios utilizados por la aplicación están implementados en un *framework* que hemos denominado FrameworkJEE. Este *framework* consiste en un conjunto de servicios o herramientas que brindan a los usuarios la funcionalidad necesaria para resolver, de forma sencilla e intuitiva, problemas recurrentes que suelen presentarse en las aplicaciones.

Entre estos servicios, podemos mencionar el uso de FTP para operaciones sobre archivos remotos, el envío de *e-mails*, mensajería, administración de archivos, seguridad (identificación y autorización), excepciones, planificación de tareas y acceso a LDAP, entre otros.

El objetivo de esta librería es agrupar servicios y utilidades de uso frecuente y estandarizarlos para lograr que el mantenimiento, las mejoras, el versionado, la distribución y la reutilización sean organizados y acordes a los requerimientos de las aplicaciones a nivel *enterprise*.

## ANT

Es una herramienta para la realización de tareas mecánicas y repetitivas, normalmente durante la fase de compilación y construcción. Desarrollada en Java, tiene la ventaja de no depender de los comandos de cada sistema operativo, sino que se basa en archivos de configuración XML y clases Java para la realización de las distintas tareas. Esto la convierte en una solución multiplataforma. Dispone de un amplio conjunto de utilidades para los más diversos fines, entre los que se pueden destacar: compilación, generación de documentación Java (Javadoc), opción de mover, crear y eliminar archivos y carpetas, y generación de JARS, WARS y EARS para la publicación, entre otros. Para más información acerca de Ant, se recomienda visitar la página del proyecto en la Apache Software Foundation: http://ant.apache.org/.

## Tiles-Defs

A partir de la utilización del *framework* Struts, contamos con un componente que facilita y ordena la capa de presentación: **Struts Tiles**. Básicamente es un sistema de *templates*, que nos permite, entre otras cosas, tener un *look and feel* (interfaz o capa de presentación) común para una aplicación *web*, además de la posibilidad de definir componentes de interfaz reusables. El uso de Tiles involucra ***layouts*** (en nuestro caso, páginas JSP), **componentes** o **piezas** (páginas JSP reutilizables que servirán para la construcción de otras) y **definiciones**. Respecto de estas últimas, hay tres maneras de implementarlas: en la misma página JSP, en un XML centralizado o en un *Action* de Struts. En nuestro caso, hemos decidido hacerlo mediante un archivo XML centralizado. Para más información sobre Struts Tiles, se recomienda visitar la página del proyecto en la Apache Software Foundation: http://struts.apache.org/1.x/struts-tiles/.

# AJAX

AJAX es el acrónimo de *Asynchronous JavaScript And XML* y es un conjunto de técnicas que utilizan el objeto JavaScript **`XMLHttpRequest`**, presente en los navegadores actuales, para comunicar asincrónicamente a los clientes livianos (navegadores *web*) con el servidor de aplicaciones, permitiendo que las aplicaciones *web* sean más rápidas y más interactivas. De esta forma, es posible realizar cambios sobre las páginas sin necesidad de recargarlas, ya que los datos adicionales se solicitan al servidor y se cargan en segundo plano sin interferir con la visualización o el comportamiento de la página.

# DOJO

DOJO Toolkit es un conjunto de herramientas JavaScript *open source* que facilita la construcción de aplicaciones *web* ricas. Estas herramientas conforman un *framework* con APIs y controles que se apoya en la tecnología **AJAX** descripta en el punto anterior, y acorta el tiempo entre la idea y la implementación. Permite resolver asuntos comunes como la navegación y la detección del navegador, el soporte de cambios de URL en la barra de URLs para luego regresar a ellas (*bookmarking*), y la posibilidad de ser ejecutado aún cuando AJAX/JavaScript no es completamente soportado en el cliente. Se lo conoce como "la navaja suiza de las bibliotecas JavaScript (*JavaScript's Swiss Army Knife*)". Proporciona una gama más amplia de opciones en una sola biblioteca JavaScript y es compatible con navegadores antiguos.

## *Complementos DOJO*

Los complementos DOJO son componentes preempaquetados de código JavaScript, HTML y CSS (páginas de estilo), que pueden ser usados para enriquecer aplicaciones *web*. Dentro de los componentes de DOJO, podemos mencionar los siguientes:

- Menús, pestañas y *tooltips*.
- Tablas ordenables, gráficos dinámicos y dibujo de vectores 2D.
- Efectos de animación y la posibilidad de crear animaciones personalizadas.
- *Drag and drop* (arrastrar y soltar).
- Formularios y rutinas de validación para los parámetros.
- Tablas con edición de datos directamente sobre las celdas.
- Calendario, selector de tiempo y reloj.

## *Comunicación asíncrona en DOJO*

La comunicación asíncrona entre el navegador y el servidor de aplicaciones es una de las características más sobresalientes de AJAX. Esto se realiza tradicionalmente a través del comando JavaScript **`XMLHttpRequest`**.

DOJO posee una capa de abstracción *(dojo.io.bind)* para varios navegadores *web*, con la que se pueden usar otros transportes (como IFrames ocultos) y diferentes formatos de datos. De esta forma, podemos obtener los campos que se van a enviar como parámetros del formulario de una manera sencilla.

Además, entre otras funcionalidades que no mencionaremos para no extendernos, DOJO provee **almacenamiento de datos**, tanto del lado del cliente como del lado del servidor. Para más información acerca de DOJO, se recomienda visitar la página del proyecto: http://dojotoolkit.org/.

## JNDI

Java Naming Directory and Interface (JNDI), es una API para servicios de directorio que nos permite encontrar objetos a partir de su nombre.

Encontrar objetos en un sistema de cierta complejidad puede ser un problema. ¿Dónde se ubican? ¿Cuáles son sus características? Necesitamos tener un sistema de nombrado que nos facilite encontrarlos, sea cual sea su ubicación y la aplicación que los utilice. JNDI nos facilita esta tarea, y además, por tratarse de una interfaz de Java, es independiente de la plataforma subyacente, lo que facilita el mantenimiento de aplicaciones y permite que programas de diversas plataformas localicen y recuperen objetos para luego utilizarlos. Es similar a una guía telefónica, sirve para buscar y encontrar un objeto y así recuperarlo, para lo cual hay que partir de nombres que sirvan de referencia a dichos objetos. Algo similar ocurre con el sistema DNS (Domain Name Service) donde se asocian nombres lógicos con direcciones IP. Para el nombrado de los objetos, es conveniente jerarquizar los nombres como las clases de Java, donde se hace referencia a los paquetes donde se encuentran, por ejemplo empresaX.aplicacionX.dominio.nombre_clase.

Cuando decimos que asociamos un **nombre** con un **objeto**, hablamos de ***binding* (unión)** del objeto a ese nombre.

Por otra parte, es necesario ampliar el concepto de servicio de nombres para llegar al de **servicio de directorio**, donde se relacionan nombres de objeto con **nombres de atributos**, al igual que ocurre en el ejemplo de la guía telefónica. De esta forma se enriquece el **sistema de búsqueda**, ya que no sólo buscamos por nombre de objeto (el nombre de una persona en la guía), sino que además podemos buscar por **filtro**. Un filtro de búsqueda es una consulta de objetos que tienen sus **atributos asociados con ciertos valores**. En nuestro ejemplo, sería una búsqueda de personas que cumplan la condición de que su localidad sea "Pila" y su calle “Máxima Planes de Casco”.

La idea de JNDI implica universalidad y abstracción: es una API genérica, por lo tanto, es necesario integrarlo con otros sistemas de nombres. Para ello, la arquitectura JNDI incluye una interfaz de provisión de servicio (Service Provider Interface, SPI), que sirve de intermediario con otros proveedores de servicios de nombres/directorios.

En resumen, la API suministra:

Un mecanismo para asociar (*bind*) un objeto a un nombre.

Una interfaz de búsqueda de directorio que permite consultas generales.

Una interfaz de eventos que permite a los clientes determinar cuándo se han modificado las entradas de directorio.

Extensiones LDAP para soportar las capacidades adicionales de un servicio LDAP.

La porción SPI permite el soporte de prácticamente cualquier tipo de servicio de directorio o nombrado, incluyendo los siguientes:

LDAP;

DNS;

Network Information Service;

RMI;

Servicio de nombres CORBA, y

Sistema de ficheros.

Para mayor información, se recomienda visitar http://java.sun.com/products/jndi/.

## Struts

Es un *framework* que implementa el patrón de arquitectura MVC en Java. MVC (Modelo-Vista-Controlador) es un patrón que define la organización independiente de las capas **Modelo** (objetos de negocio), **Vista** (interfaz con el usuario u otro sistema) y **Controlador** (controlador del *workflow* de la aplicación). En dicho patrón, el flujo de la aplicación está dirigido por un controlador central. El controlador delega solicitudes a un manejador apropiado. Los manejadores están unidos a un modelo, y cada manejador actúa como un adaptador entre la solicitud y el modelo. El modelo representa, o encapsula, un estado o lógica de negocio de la aplicación. Luego, el control se devuelve a la vista apropiada a través del controlador. El reenvío se puede determinar consultando los conjuntos de mapeos, normalmente cargados desde una base de datos o un fichero de configuración. Esto proporciona un acoplamiento cercano entre la vista y el modelo, que puede hacer que las aplicaciones sean significativamente más fáciles de crear y de mantener.

Basándose en el patrón de diseño Modelo-Vista-Controlador, el *framework* **Struts** tiene tres componentes principales: un ***servlet* controlador**, que está proporcionado por el propio **Struts** (que será el **controlador**), las **páginas JSP** (la **vista**), y, por último, la **lógica de negocio** de la aplicación (el **modelo**).

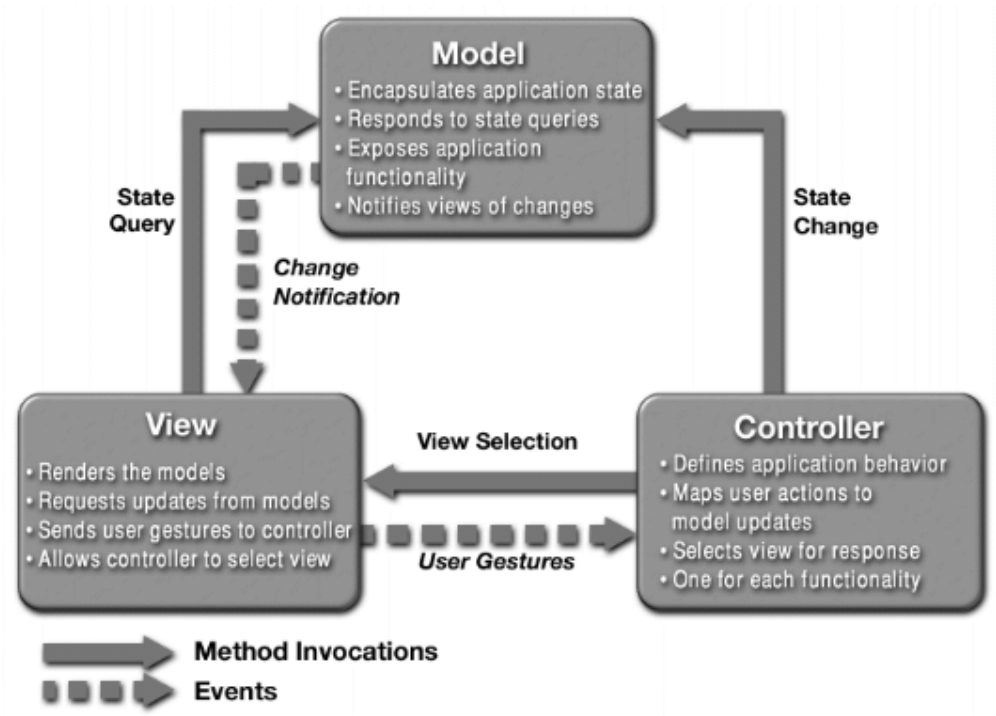



Figura 20. Patrón de diseño MVC

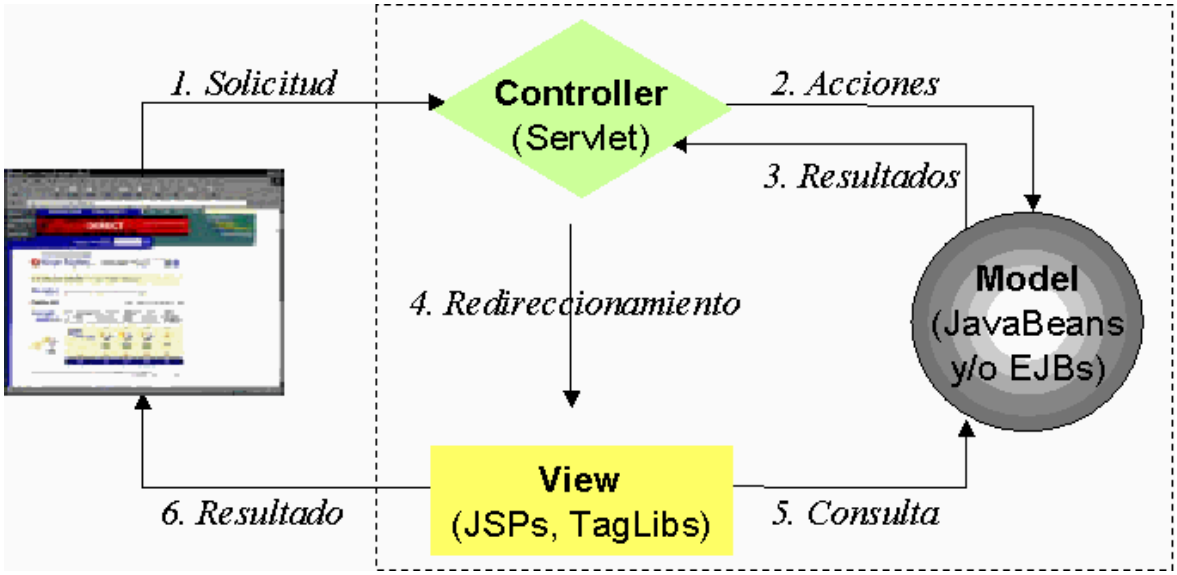



Figura 21. *Framework* Struts

El navegador genera una solicitud que es atendida por el controlador (un Servlet especializado). Éste se encarga de analizarla, seguir la configuración programada en su XML (archivo de configuración) y llamar al *action* correspondiente pasándole los parámetros enviados. El *action* instanciará y/o utilizará los objetos de negocio para concretar la tarea. Según el resultado que retorne el objeto Action, el *controller* derivará la

generación de interfaz a una o más JSPs, que podrán consultar los objetos del *model* para realizar su tarea.

Para mayor información sobre el *framework* Struts, se recomienda consultar http://jakarta.apache.org/struts.

## JSON

JSON es el acrónimo de JavaScript Object Notation (notación de objetos de JavaScript) y es un formato ligero de intercambio de datos. Leerlo y escribirlo es simple para los humanos, mientras que para las máquinas, es simple de interpretar y generar. Está basado en un subconjunto del lenguaje de programación JavaScript Standard, "ECMA-262 3rd Edition - December 1999".

JSON es un formato de texto completamente independiente del lenguaje, pero que utiliza convenciones ampliamente conocidas por los programadores de la familia de lenguajes C, incluyendo C, C++, C#, Java, JavaScript, Perl, Python, y muchos otros. Estas propiedades hacen de JSON un lenguaje ideal para el intercambio de datos.

Está constituido por dos estructuras:

- Una colección de pares de nombre/valor. En varios lenguajes, esto se conoce como objeto, registro, estructura, diccionario, tabla *hash*, lista de claves o arreglo asociativo.
- Una lista ordenada de valores. En la mayoría de los lenguajes, esto se implementa como arreglos, vectores, listas o secuencias.

Estas son estructuras universales; virtualmente todos los lenguajes de programación las soportan de una forma u otra. Es razonable que un formato de intercambio de datos independiente del lenguaje de programación se base en estas estructuras.

En JSON, un objeto es un conjunto desordenado de pares nombre/valor. Comienza con **{** (llave de apertura) y termina con **}** (llave de cierre). Cada nombre es seguido por **:** (dos puntos) y los pares nombre/valor están separados por **,** (coma).

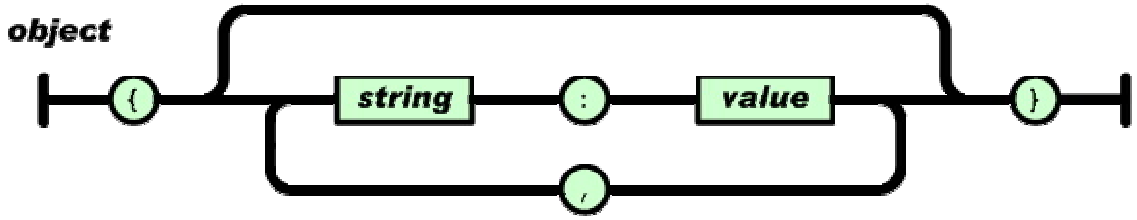



Figura 22. Estructura de un objeto en JSON

Es la simplicidad de JSON lo que ha dado lugar a la generalización de su uso, especialmente como alternativa a XML en AJAX. Una de las supuestas ventajas de JSON sobre XML como formato de intercambio de datos en este contexto es que es mucho más sencillo escribir un analizador semántico con JSON. En JavaScript, JSON puede ser analizado trivialmente usando el procedimiento eval(), lo cual ha sido fundamental para

su aceptación por parte de la comunidad de desarrolladores Ajax, debido a la ubicuidad de JavaScript en casi cualquier navegador *web*.

En la práctica, los argumentos a favor de la facilidad de desarrollo de analizadores y de su rendimiento son poco relevantes, debido a las cuestiones de seguridad que plantea el uso de eval() y el auge del procesamiento nativo de XML, incorporado en los navegadores modernos. Por esa razón, JSON se emplea habitualmente en entornos donde el tamaño del flujo de datos entre cliente y servidor es de vital importancia (de aquí su uso por Yahoo y Google, que atienden a millones de usuarios), cuando la fuente de datos es explícitamente de fiar y no disponer de procesamiento XSLT para manipular los datos en el cliente no es importante.

Si bien es frecuente ver JSON contrapuesto a XML, también es frecuente el uso de ambos en la misma aplicación. Por ejemplo, una aplicación de cliente que integra datos de Google Maps con datos meteorológicos en SOAP hace necesario soportar ambos formatos.

Para más información, se recomienda visitar: http://www.json.org/ y la página en Wikipedia, http://es.wikipedia.org/wiki/JSON.

## Jasper Reports

**JasperReports** es una herramienta *open source* de creación de informes Java que tiene la habilidad de entregar contenido enriquecido en el monitor, la impresora o archivos de tipo PDF, HTML, XLS, CSV y XML. Está escrito completamente en Java y puede ser usado en gran variedad de aplicaciones de Java, incluyendo J2EE o aplicaciones *web*, para generar contenido dinámico. Su propósito principal es ayudar a crear documentos de tipo páginas, preparados para imprimir en una forma simple y flexible, o bien exportar información en distintos tipos de documentos. JasperReports se usa comúnmente con iReports, un *front-end* gráfico *open source* para la edición de informes.

### *Descripción general de la API*

JasperReports organiza los datos recuperados desde un *data source* de acuerdo a un "diseño de reporte" o "*template*" definido en un archivo con extensión JRXML. Para llenar un reporte con los datos recuperados, el template JRXML primero debe ser compilado, lo cual genera un archivo con extensión .jasper.

La compilación del archivo JRXML que representa el *template* o plantilla lo lleva a cabo el método `compileReport()` de la clase `JasperCompileManager`.

Una vez compilado y generado el archivo con extensión .jasper, el template se puede cargar en un objeto *template* (clase `JasperReport`), que luego se serializa y almacena en el disco. Este objeto serializado se usa cuando la aplicación desea llenar el *template* con los datos. De hecho, compilar un *template* implica compilar todas las expresiones Java definidas en el archivo JRXML que representa el *template*.

Para llenar un *template*, se puede utilizar uno de los métodos `fillReportXXX()` expuestos por la clase `JasperFillManager`. Estos métodos reciben como parámetro el objeto `JasperReport` que representa el *template*, o un archivo que represente el objeto *template* especificado, de forma serializada, y también una conexión JDBC a la base de datos o a una colección de objetos, a través de la cual recuperará los datos con los cuales llenará el reporte. El resultado es un objeto que representa un documento listo para imprimir (clase `JasperPrint`), que puede ser almacenado en disco en forma serializada para ser usado posteriormente: puede ser enviado a la impresora o directamente al monitor, o puede ser exportado en documentos de tipo PDF, HTML, XLS, RTF, ODT, CSV, TXT or XML.

Las principales clases usadas cuando se trabaja con JasperReports son, entonces, las siguientes:

```
net.sf.jasperreports.engine.JasperCompileManager
net.sf.jasperreports.engine.JasperFillManager
net.sf.jasperreports.engine.JasperPrintManager
net.sf.jasperreports.engine.JasperExportManager
```

Estas clases representan un *facade* del motor JasperReports. Tienen varios métodos de tipo *static* que simplifican el acceso a la funcionalidad de la API y pueden ser usados para compilar un archivo JRXML con el *template*, llenar un reporte con los datos, imprimirlo o exportarlo en un documento de alguno de los tipos mencionados.

Además de estas clases *facade*, también se puede exportar clases específicas, en el caso en que se desee exportar los reportes a formatos no definidos en `JasperExportManager`, o bien cuando se necesite configurar el proceso de exportación adaptándolo a las necesidades específicas. Estas implementaciones del *exporter* pueden encontrarse en el paquete `net.sf.jasperreports.engine.export` de la librería JasperReports.

Si se necesita mostrar un reporte dentro de una aplicación Swing, se puede usar el componente `JRViewer` que se transporta con la librería y consiste en un componente `javax.swing.Jpanel` embebido y configurable. El `JasperViewer` es una aplicación Swing *stand-alone* que usa el componente `JRViewer` para mostrar el reporte en formato propietario.

Para más información acerca de JasperReports, se recomienda visitar http://jasperforge.org/projects/jasperreports.

## Junit

Es un conjunto de librerías utilizadas para hacer pruebas unitarias de aplicaciones Java. Permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada, se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el

método durante la ejecución, devolverá un fallo en el método correspondiente. Es también un medio para controlar las pruebas de regresión, necesarias cuando una parte del código ha sido modificado y se desea ver que el nuevo código cumple con los requerimientos anteriores y que no se ha alterado su funcionalidad después de la nueva modificación. Incluye formas de ver los resultados (*runners*) que pueden ser en modo texto, gráfico (AWT o Swing) o como tarea en Ant.

Para más información acerca de Junit, se recomienda visitar http://www.junit.org/.

# Desarrollo: detalles de implementación

En este punto, la intención es mostrar la forma en que han sido implementadas las diferentes capas y los detalles sobre el uso de las tecnologías y estándares involucrados presentados en el punto anterior.

## Utilización de ANT

### *Mecanismo de configuración*

Por medio de este mecanismo, y en forma automática, se lleva a cabo la configuración adecuada para cada servicio disponible en el FrameworkJEE, en el momento de la construcción de la aplicación en cuestión.

Consiste básicamente en reemplazar las variables definidas en un archivo de la aplicación, con los valores correspondientes al entorno de ejecución, de acuerdo a lo especificado en el archivo `environment_name.properties` del proyecto **Environment** para dicha aplicación. Este archivo se llama `Configuration.xml`, se encuentra en la carpeta `\\workspace_dir\AppName\ant` y contiene elementos que representan objetos de configuración para cada uno de los servicios.

Cada servicio se valdrá de su clase de configuración, más lo especificado en este archivo, para realizar las operaciones solicitadas, haciendo transparente para el usuario esta problemática y facilitando su uso.

#### Proyecto Environment

Se trata de un proyecto Java donde se definen los valores de configuración de los servicios que utiliza cada aplicación. No cuenta con clases, por lo tanto, no es necesaria su construcción: solamente se utiliza para organizar la configuración de cada servicio por entorno y aplicación.

Es, sin embargo, imprescindible contar con este proyecto en el *workspace* donde se realizará la construcción de la aplicación en cuestión, ya que su archivo de configuración para los diferentes ambientes depende de lo especificado en los archivos de propiedades de este proyecto.

## Estructura del proyecto

Existe un directorio por cada aplicación, dentro de los cuales se definen subdirectorios por cada entorno o ambiente existente (por ejemplo desarrollo, *test* y producción). En cada uno de estos subdirectorios se define un archivo de propiedades (entorno`.properties`) con los valores de las variables de configuración correspondientes a dicho entorno.

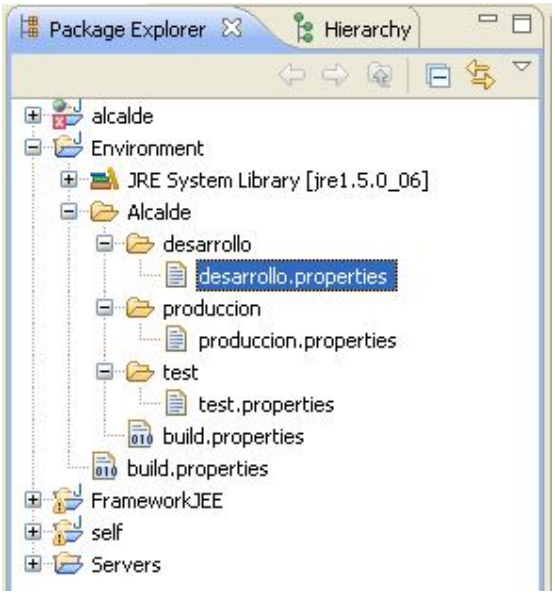



Figura 23. Archivo de propiedades

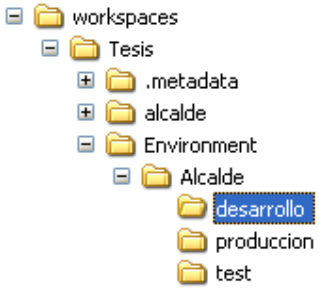



Figura 24. Estructura de directorios

## Archivos de propiedades (`build.properties, entorno.properties`)

El archivo `build.properties` del directorio principal de cada aplicación (aquel que sale de la raíz del proyecto Environment) define el entorno para el cual se realizará la construcción.

Por lo tanto, cuando se quiera construir, por ejemplo, para el ambiente de *test*, deberá asignarse el valor *test* a la variable "entorno" del archivo `build.properties`, y de forma similar para los restantes entornos. Esta asignación implica que las tareas automatizadas de Ant buscarán en el archivo `test.properties` de la aplicación los valores de las

variables a reemplazar (que se encuentran en el archivo `Configuration.xml`), y que, una vez finalizada la construcción, el archivo con los valores correspondientes al entorno seleccionado se encontrará en la carpeta `classes`.

Un ejemplo de archivo `entorno.properties` puede ser:

```
$ftp.server$=192.168.10.33
$ftp.port$=21
$ftp.username$=ftpdes
$ftp.password$=desa_pwd

$ldap.server$=municipio_ldap
$ldap.port$=389
$ldap.username$=ldap
$ldap.password$=654321
$ldap.base$=cn=Users,dc=municipio
$ldap.baseBusqueda$=ou=Alcalde,dc=municipio

$servicelocator.jdbc.containerManaged$=true
$servicelocator.jdbc.initialcontextfactory$=
$servicelocator.jdbc.providerurl$=
$servicelocator.jdbc.lookuproot$=java:comp/env/

$mail.server$=municipio.pila.gov.ar
$mail.port$=25
$mail.protocol$=smtp
$mail.username$=usuariox
$mail.password$=passwordx
```

## Otros conceptos acerca de la configuración

- **Variable de configuración**: llamaremos de esta forma a aquellas variables que serán reemplazadas durante la construcción de un proyecto para un entorno determinado. Estas variables se encuentran ubicadas en el archivo `Configuration.xml` y sus valores, en el archivo `entorno.properties`, en la carpeta correspondiente al entorno y la aplicación en cuestión dentro del proyecto Environmen. Las escribiremos entre signos `$`: por ejemplo, `$ftp.server$`.
- **Clases de configuración de servicios**: Cada servicio perteneciente al *framework* cuenta con una clase de configuración con atributos necesarios para la utilización del servicio. Durante la inicialización de la aplicación, estas clases serán instanciadas y cargadas con la información provista en el archivo de configuración.
- **`Digester`**: Es un componente que utilizaremos para leer archivos xml de configuración para proporcionar la inicialización de los objetos que se utilizarán en una aplicación, en particular en la configuración de servicios.

## El archivo `Configuration.xml`

Como precondición para el funcionamiento del mecanismo de configuración automática de la aplicación, se deberá contar con un archivo denominado

`Configuration.xml` en el directorio `\\workspace_dir\AppName\ant`. En este archivo, estarán definidas las propiedades requeridas por los servicios utilizados por la aplicación, además de diversas propiedades de su interés. Un ejemplo de un archivo `Configuration.xml` para la aplicación Alcalde podría ser el siguiente:

```
<?xml version="1.0"?>
<configuration>

      <ldap default="true">
            <alias>$ldap.alias$</alias>
            <server>$ldap.server$</server>
            <port>$ldap.port$</port>
            <username>$ldap.username$</username>
            <password>$ldap.password$</password>
            <base>$ldap.base$</base>
            <basebusqueda>$ldap.basebusqueda$</basebusqueda>
      </ldap>

      <servicelocator>
            <jdbc
containerManaged="$servicelocator.jdbc.containerManaged$">
                        <initialcontextfactory>
                        $servicelocator.jdbc.initialcontextfactory$
                        </initialcontextfactory>

      <providerurl>$servicelocator.jdbc.providerurl$</providerurl>

      <lookuproot>$servicelocator.jdbc.lookuproot$</lookuproot>
            </jdbc>
      </servicelocator>

        <ftpClient default="true">
            <alias>ftp1</alias>
            <server>$ftp.server$</server>
            <port>$ftp.port$</port>
            <username>$ftp.username$</username>
            <password>$ftp.password$</password>
      </ftpClient>

      <exceptions>
            <path>$exceptions.path$</path>
            <datasourcename>$exceptions.datasource$</datasourcename>
      </exceptions>

      <properties>
            <property name="propiedad1" value="valor1"/>
      </properties>

</configuration>
```

Para cada configuración de servicio se debe especificar un atributo `alias`, que servirá para etiquetar y agrupar los elementos utilizados por él. El resto, serán variables de configuración a ser reemplazadas posteriormente.

Para cada configuración, disponemos de varios elementos que dependen del servicio en cuestión, cuyos valores deben ser los nombres de las variables que luego serán reemplazadas dependiendo del entorno definido a la hora de la construcción.

En el caso en que la aplicación necesite usar más de una configuración para un servicio, basta con definir un nuevo elemento de servicio y especificar los atributos `alias` y `default`. En este caso, el atributo `default` es necesario ya que el mecanismo de configuración lo utilizará a la hora de la inicialización de los objetos, para asignar una configuración por defecto para cada servicio.

## Funcionamiento

Este mecanismo consiste en tres pasos, en primer lugar la **selección del entorno**, luego el **reemplazo de variables** y por último, la **creación de objetos de configuración** de cada servicio.

### Selección del entorno

Para especificar el entorno para el cual queremos construir la aplicación, se debe modificar el archivo `build.properties`, situado en la carpeta principal de la aplicación en cuestión, dentro del proyecto Environment. Por ejemplo, para asignar el entorno deseado a la variable "entorno" en el ambiente de desarrollo, sería de la siguiente forma:

```
entorno=desarrollo
```

### Reemplazo de variables

El reemplazo de las variables de configuración se hace a través de tareas de Ant, las cuales se ejecutan al construir el proyecto para un entorno dado. Como resultado de estas tareas, en la carpeta `classes` del proyecto, el archivo `Configuration.xml` contará con los valores adecuados de cada servicio, según el entorno especificado.

### Creación de objetos de configuración

Existen clases de configuración para cada servicio, donde sus atributos representan propiedades estándar de un servicio, como, por ejemplo, servidor, puerto, nombre de usuario y contraseña. Una clase de tipo `Digester` es la encargada de parsear el archivo `Configuration.xml` e instanciar los objetos necesarios para la configuración de cada servicio. Lo único que se necesita es especificar la ubicación del archivo `Configuration.xml` que va a ser parseado. Existen dos formas de hacer esto:

- En aplicaciones *web*, se deberá realizar en una clase de tipo "`PlugIn`" `(org.apache.struts.action.PlugIn)` en su método `init`. En nuestro caso, la clase encargada de realizar esto es `alcalde.struts.plugins.Conector`. La ubicación para

este caso es la carpeta "WEB-INF". Es necesario destacar que al constructor del DigesterDriver se le pasa un InputStream, y al método parse, un valor null.

```
//Se carga la configuración del entorno
InputStream is = servlet.getServletContext().getResourceAsStream(
"/WEB-INF/classes/Configuration.xml");
DigesterDriver tmpDig = new DigesterDriver(is);
Configuration configFWK = tmpDig.parseFile(null);
```

- En los *test* unitarios, se deberá realizar antes de cualquier otra operación. La ubicación, para este caso, es la carpeta raíz del código fuente. Es necesario destacar que se utiliza el constructor sin parámetros del DigesterDriver y al método parse se le pasa la ubicación y nombre del archivo de configuración.

```
DigesterDriver tmpDig = new DigesterDriver();
tmpDig.parseFile("Configuration.xml");
```

## Requerimientos

Las librerías necesarias para su funcionamiento son las siguientes:

- commons-net-1.4.1.jar;
- commons-digester-1.8.jar, y
- log4j-1.2.8.jar.

## Ejemplo de servicio

Como ejemplo para graficar la utilización de los servicios disponibles en el *framework* y configurados a partir del mecanismo descripto, tomaremos el servicio cliente de ftp y uno de sus métodos para obtener un archivo remoto. Este servicio nos brinda toda la funcionalidad que ofrece el protocolo ftp a nivel cliente a través de la clase ar.com.cinq.jee.ftp.FTPClient.

El método retrieveFile(filename, outputStream) de la clase escribe el archivo especificado por el parámetro filename en el outputStream correspondiente.

En este caso, queremos obtener el archivo "ApplicationResources.properties" del servidor de ftp y que se escriba en el outputStream obtenido a partir del archivo out1.txt.

```
...
OutputStream os = null;
FTPClient ftp = null;
try {
      ftp = new FTPClient();
      os = new FileOutputStream("C:/temp/out1.txt");

      ftp.retrieveFile("ApplicationResources.properties", os);
      os.flush();
}
catch (Exception e) {
      e.printStackTrace();
}
```

```
finally {
      try {
            os.close();
      }
      catch (Exception e) {}
}
...
```

Como se ve, sólo es necesario instanciar la clase `ar.com.cinq.jee.ftp.FTPClient` y luego invocar el método `retrieveFile` con los parámetros adecuados.

En el ejemplo, se instancia la clase a través de su constructor vacío, lo que indica que se tomará la configuración por defecto para dicho servicio, más allá de que exista una o varias de ellas.

Si se desea una configuración en particular, se debe especificar su alias como parámetro en el constructor. Recordemos que la configuración por defecto es aquella que especifica el atributo `default` en "`true`", o la que exista, en el caso que haya una sola.

En términos generales, todos los servicios lanzan una excepción base propia, que se desprende de las ya conocidas. En particular, todos los métodos públicos de la clase `ar.com.cinq.jee.ftp.FTPClient` lanzan la excepción `ar.com.cinq.jee.ftp.FTPClientException`.

## Definición de la estructura de las páginas con `Tiles-Defs`

Como ya mencionamos anteriormente en la definición de Struts Tiles, en nuestra aplicación, hemos decidido organizar la estructura de las páginas JSP a través de la utilización de Tiles, implementándolo mediante la definición en un archivo centralizado llamado `tiles-defs.XML`. Vale la pena aclarar que la definición en forma centralizada permite, entre otras cosas, reducir el código necesario en las páginas.

### *Pre-condiciones*

Para comenzar a trabajar con Tiles, es necesario preparar el ambiente de trabajo, ya que no viene configurado por defecto. Para esto, debemos activar el *framework* mediante la introducción de pequeños fragmentos de código en los distintos archivos de configuración, además de copiar una serie de archivos necesarios que contienen las librerías de *tags*. Estos son algunos de los pre-requisitos a tener en cuenta:

- Utilizar Tags Libraries en nuestra aplicación.
- Modificar el archivo `WEB-INF/ web.xml` para que incluya la declaración del *tag library* `struts-tiles`:

```
...
<taglib>
    <taglib-uri>/WEB-INF/struts-tiles.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-tiles.tld</taglib-location>
</taglib>
...
```

- Copiar el archivo `struts-tiles.tld` al directorio `WEB-INF`.
- Dentro del código fuente de cada página, al inicio, incluir una línea declarativa del *tag library* que se utilizará en esa página. En nuestro caso, nos interesa el *tag* `struts-tiles`.

```
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>
```

- Por último, como hemos decidido implementar la utilización de Tiles de forma centralizada, debemos, por un lado, crear los archivos, y por el otro, indicar en el archivo `struts-config.xml`, cuál o cuáles serán los archivos que contendrán las definiciones para que la clase `org.apache.struts.tiles.TilesPlugin` las pueda leer y cargar. Por ejemplo, suponiendo que el archivo que contiene las definiciones es `WEB-INF/tiles-defs.xml`, la forma de hacerlo es la siguiente:

```
<!-- struts-config.xml

<controllerprocessorClass="org.apache.struts.tiles.TilesRequestProcessor"
bufferSize="4096"/>
 <plug-in className="org.apache.struts.tiles.TilesPlugin" >
   <set-property property="definitions-config" value="/WEB-INF/tiles-
defs.xml" />
   <set-property property="definitions-parser-validate" value="true" />
   <set-property property="moduleAware" value="false" />
 </plug-in>
-->
```

## *Implementación de Tiles en nuestra aplicación*

En nuestra aplicación, valiéndonos de los Tiles, además de componentes reusables tales como las páginas `header.jsp`, `footer.jsp` o `menu.jsp`, creamos una serie de plantillas que definen el formato que tendrán todas las páginas diseñadas. Es decir, todas las páginas definidas en el archivo `tiles-defs.xml` extenderán alguna de estas plantillas. Veamos algunas definiciones a modo de ejemplo:

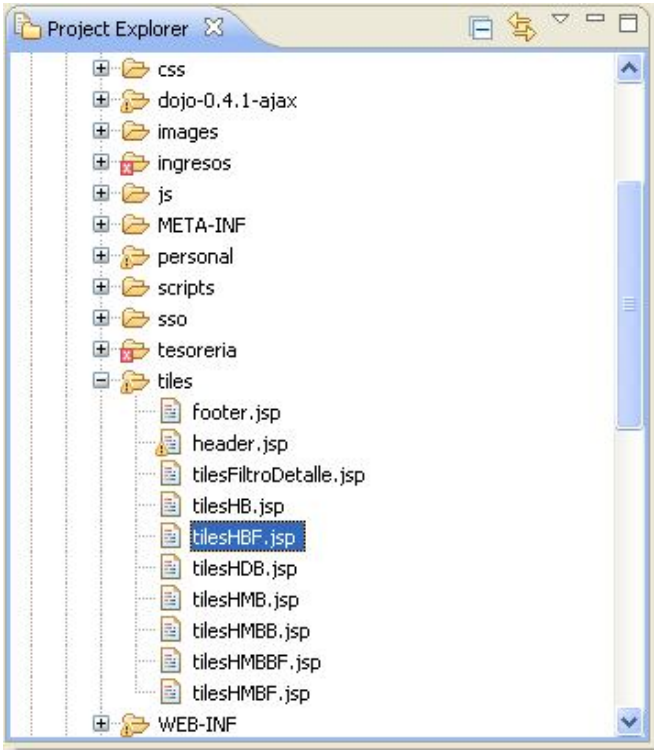



Figura 25. Definiciones

En la figura podemos observar que, dentro de la carpeta **tiles,** se encuentran las páginas `footer.jsp` y `header.jsp` que son **componentes**. Además, hay páginas cuyos nombres están compuestos por la palabra *tiles* y una sigla u otro nombre (por ejemplo, `tilesHBF.jsp`): éstas son las **plantillas** que mencionábamos antes. Los nombres de éstas plantillas siguen una nomenclatura que indica su estructura: en el caso del ejemplo, `tilesHBF.jsp` significa que la página que la implemente estará compuesta por un *header*, un *body* y un *footer*. En este caso, *header* y *footer* serán los componentes predefinidos y *body* hará referencia a una página específica de la aplicación. Eso está definido en el archivo `tiles-defs.xml` de la siguiente manera:

```
<!-- tiles-defs.xml
...
<tiles-definitions>

     <!-- Definiciones de plantillas -->

  <definition name="alcalde.tilesHBF" path="/tiles/tilesHBF.jsp">
     <put name="title" value="ALCALDE"/>
      <put name="header" value="/tiles/header.jsp"/>
     <put name="footer" value="/tiles/footer.jsp"/>
  </definition>
...
-->
```

Luego, en el mismo archivo `tiles-defs.xml`, vamos a tener una parte para la definición de las páginas de la aplicación propiamente dichas, y es ahí donde va a haber definiciones de páginas que implementen esas plantillas, por ejemplo:

```
<!-- tiles-defs.xml
...
<!-- Definiciones de páginas -->

  <definition name="alcalde.seleccionarRol" extends="alcalde.tilesHBF">
     <put name="title" value="ALCALDE - seleccionarRol"/>
     <put name="bodycontent" value="/seleccionarRol.jsp"/>
  </definition>
...
-->
```

En este ejemplo, podemos observar que la página `seleccionarRol` extiende la plantilla `alcalde.tilesHBF` definida antes, por lo que la estructura de la misma estará compuesta por el componente *header* y el componente *footer*, y su *body* será la página `seleccionarRol.jsp`, según lo indicado a través de la definición en el atributo `bodycontent`. Por último, así como la página `seleccionarRol.jsp` contendrá los componentes de interfaz específicos y necesarios de acuerdo a la funcionalidad pretendida, la página `/tiles/tilesHBF.jsp` incluirá, además de los componentes de diseño, los *inserts* de los atributos que se le asignan en las definiciones, como se muestra en el fragmento de código a continuación:

```
<!-- /tiles/tilesHBF.jsp
...
<div id="container">
  <center>
    <tiles:insert attribute="header"/>
    <tiles:insert attribute="bodycontent"/>
  </center>
</div>

<div id="footer">
    <tiles:insert attribute="footer"/>
</div>
...
-->
```

## Detalles de implementación de DOJO en nuestra solución

En esta sección veremos en detalle el uso de DOJO Toolkit, en la implementación de algunas funcionalidades de nuestra aplicación.

Para empezar, veamos la forma en la que implementamos la presentación del menú de ejecución al usuario en función de su **rol** o el **grupo** al que pertenece, justamente valiéndonos de DOJO para su construcción. La arquitectura de esta especie de menú dinámico está compuesta a grandes rasgos por las siguientes tres capas:

- La definición de los roles y las acciones disponibles en la aplicación a través de una estructura de base de datos diseñada para tal fin.
- Un conjunto de utilidades disponibles en el FrameworkJEE (que describimos anteriormente y a través del cual se dispone de funcionalidades recurrentes en las distintas aplicaciones), que brindan la funcionalidad necesaria para la recuperación de las acciones disponibles para el usuario para la posterior construcción del menú.
- La capa de presentación del menú al usuario, la cual se basa en DOJO para construir en las páginas la estructura del menú en forma dinámica.

## *Definición de roles y acciones*

La definición de los roles y la asignación de las acciones disponibles para dichos roles se realiza en una base de datos creada específicamente para este fin. Esta base de datos consiste en unas pocas tablas que alcanzan para definir y almacenar la estructura de acciones de la que hablamos. Las tablas en cuestión son `Aplicacion`, `Accion`, `Rol` y `Rol_Accion`.

**aplicacion**(c_aplicacion, x_aplicacion, x_nom_corto)

**accion**(c_accion, x_accion, x_url, c_padre, c_aplicacion)

**rol**(c_rol, x_rol, c_aplicacion)

**rol_accion**(c_rol, c_accion, c_aplicacion)

La tabla `Aplicacion` contiene las aplicaciones para las cuales se ha configurado el menú dinámico. En la tabla `Accion`, se almacenan todas las acciones disponibles para una determinada aplicación, que equivale a decir todas las opciones de menú disponibles para una aplicación dada. Vale la pena destacar que la estructura de las acciones se realiza en forma de árbol. En la tabla `Rol` se almacenan todos los roles disponibles por aplicación, y por último, en la tabla `Rol_Accion` se establece qué acciones (opciones de menú) están permitidas para un rol determinado dentro de una aplicación dada.

| c_aplicacion | x_aplicacion | x_nom_corto |
|---|---|---|
| 1 | Alcalde | Alcalde |
| 2 | Personal | Personal |

Tabla 11. Tabla `Aplicacion`

| c_accion | x_accion | x_url | c_padre | c_aplicacion |
|---|---|---|---|---|
| 100 | INGRESOS | <null> | 999 | 1 |
| X 101 | Contribuyentes | /alcalde/contribuyentes.do | 100 | 1 |
| X 102 | Campos | /alcalde/campos.do | 100 | 1 |
| X 103 | Lotes | /alcalde/lotes.do | 100 | 1 |

| | | | | | |
|---|---|---|---|---|---|
| X | 104 | Etc | /alcalde/algo.do | 100 | 1 |
| | 200 | TESORERIA | <null> | 999 | 1 |
| | 201 | Caja | <null> | 200 | 1 |
| | 203 | Recibos Pendientes Tasas | /alcalde/recibosPendTasas.do | 201 | 1 |
| | 204 | Recibos Cobrados Tasas | /alcalde/recibosCobradosTasas.do | 201 | 1 |
| | 205 | Recibos Pendientes Servicios | /alcalde/recibosPendServicios.do | 201 | 1 |
| | 206 | Recibos Cobrados Servicios | /alcalde/recibosCobradosServicios.do | 201 | 1 |
| | 300 | COMPRAS | <null> | 999 | 1 |
| | 301 | Pedidos de Suministro | /alcalde/pedidos.do | 300 | 1 |
| | 999 | RAIZ_APLICACION | <null> | 0 | 0 |

Tabla 12. Tabla `Accion`

Como podemos observar, las acciones propiamente dichas (aquellas que tienen datos en el campo **`x_url`**) son las hojas, y por ende es sobre esas acciones que otorgaremos los permisos o no a los diferentes roles.

| c_rol | x_rol | c_aplicacion |
|---|---|---|
| 1 | ADMINISTRADORES | 1 |
| 2 | **ADMINISTRATIVO_INGRESOS** | 1 |
| 3 | ADMINISTRATIVO_TESORERIA | 1 |
| 4 | CONSULTA | 1 |
| 1 | ADMINISTRADORES | 2 |
| 2 | LIQUIDADOR | 2 |
| 3 | ADMINISTRATIVO | 2 |
| 4 | CONSULTA | 2 |

Tabla 13. Tabla `Rol`

| c_rol | c_accion | c_aplicacion |
|---|---|---|
| 2 | 101 | 1 |
| 2 | 102 | 1 |
| 2 | 103 | 1 |
| 2 | 104 | 1 |

Tabla 14. Tabla `Rol_Accion`

En este ejemplo, según lo indicado en la tabla `Rol_Accion`, estamos otorgando los permisos para que a los usuarios pertenecientes al rol `ADMINISTRATIVO_INGRESOS` de la aplicación Alcalde, se les presenten las opciones de menú marcadas con una cruz en la tabla `Accion`.

## *Recuperación de acciones disponibles para el usuario*

En cuanto a la recuperación de las acciones u opciones de menú disponibles para los usuarios, se realiza en función del rol o grupo al que pertenecen. Dicha recuperación, como ya anticipamos, se lleva a cabo a través de una serie de servicios implementados en el *framework* J2EE. Dentro de las funcionalidades provistas por este *framework*, se encuentra la necesaria para gestionar sobre las estructuras de datos y, en función del rol especificado, consultar las acciones disponibles para dicho rol y armar la estructura de menú correspondiente.

Las clases disponibles están organizadas en el paquete `ar.com.cinq.jee.menu.*`. A partir de este paquete de clases disponibles, es posible indicarle a la clase encargada de construir el menú que retorne la estructura correspondiente, por ejemplo, a la aplicación Alcalde, para el rol `ADMINISTRATIVO_INGRESOS`, de una forma muy sencilla:

```
MenuManager.getInstance().getMenu(idAplicacion, idRol,
tipoImplementacion)
MenuManager.getInstance().getMenu(1, 2,
TipoImplementacion.DOJO_IMPLEMENTATION)
```

Recordemos que en el ejemplo donde mostramos las estructuras de las tablas donde se almacenan las acciones, `c_Aplicacion=1` corresponde a la aplicación Alcalde y `c_Rol=2` corresponde al rol `AMINISTRATIVO_INGRESOS`.

La clase `MenuManager` se encarga de realizar las consultas correspondientes a la base de datos a través de la clase `MenuDAO` y, en función del tipo de implementación, armar el menú propiamente dicho. En nuestro caso, sólo hemos realizado la implementación para construir menús valiéndonos de las utilidades de DOJO, pero puede extenderse fácilmente para habilitar la construcción de menús a partir de cualquier otra herramienta, siempre que esté basada en AJAX o aproveche sus bondades. A continuación, transcribimos una porción de código de la clase `MenuManager` donde se muestra la implementación de una parte de la solución, para comprender mejor de qué se trata:

```
/**
 * Método que retorna un string cuyo contenido es la estructura del menú,
 * de acuerdo a la aplicación, el rol y la implementación que llegan como
 * parámetros. Este método se invoca desde la interfaz.
 * @param idAplicacion
 * @param idRol
 * @param codImplementacion
 * @return
 * @throws MenuException
 */
public String getMenu(int idAplicacion, int idRol, int codImplementacion)
throws MenuException {
      try {
            String result = null;
            Menu menu = getMenu(idAplicacion, idRol); (*)
            switch (codImplementacion) {
                  case TipoImplementacion.DOJO_IMPLEMENTATION:
                        result = DojoMenuBuilder.transformar(menu); (**)
                        break;
                  default:
                        result = "NO EXISTE LA IMPLEMENTACION PARA
                        CONSTRUIR EL MENU";
                        break;
            }
            return result;
      }
      catch (MenuException me) {
            throw me;
      }
}//getMenu
/**
 * Método que retorna un objeto Menu, que se corresponde con la
 * estructura de menú de acuerdo a la aplicación y el rol que llegan como
```

```
 * parámetros. Este método es llamado en forma privada desde la clase
 * MenuManager.
 * @param idAplicacion
 * @param idRol
 * @return
 * @throws MenuException
 */
(*)private Menu getMenu(int idAplicacion, int idRol) throws MenuException
{
      Menu menu = (Menu)menues.get(String.valueOf(idRol));
      if (menu == null) {
            Rol rol = RolManager.getInstance().getRol(idAplicacion,
idRol);
            menu = rolMenu(rol); (***)
            menues.put(String.valueOf(rol.getCodigo()), menu);
      }
      return menu;
}//getMenu()
/**
 * Método de la clase DojoMenuBuilder que recibe un objeto Menu
 * instanciado con la estructura correspondiente al menú asociado a un
 * rol dado de una aplicación dada y retorna un string que representa
 * dicho menú con el formato de DOJO.
 * @param root
 * @return
 */
(**)public static String transformar(Menu root) {
      String result = "";
      Cola porExplorar = new EnlazadaCola();
      Menu nodo = null;
      if (root.esRaiz()) {
        result += "<div dojoType='MenuBar2'>\n";

        Iterator hijosRaiz = root.listChildMenus().iterator();
        while (hijosRaiz.hasNext()) {
            Menu hijo = (Menu)hijosRaiz.next();
            porExplorar.insertar(hijo);
            result += "<div dojoType='MenuBarItem2' caption='" +
                  hijo.getDescripcion()+ "' submenuId='" +
            hijo.getCodigo() +
                  "'></div>\n";
        }
        result += "</div>\n";
      }
      while (!porExplorar.esVacio()) {
        try {
             nodo = (Menu)porExplorar.suprimir();
        }
        catch (Exception e) {}

        if (nodo instanceof CompositeMenu) {
            result += "<div dojoType='PopupMenu2' widgetId='" +
                        nodo.getCodigo() + "'>\n";
```

```
            Iterator hijosNodo = nodo.listChildMenus().iterator();
            while (hijosNodo.hasNext()) {
                    Menu hijo = (Menu)hijosNodo.next();
                    porExplorar.insertar(hijo);
                    result += hijo.render();
            }
            result += "</div>\n";
          }
      }
      return result;
  }//transformar()

/**
 * Método que construye la estructura del menú para un objeto Rol, y la
 * retorna en un objeto Menu.
 * @param rol
 * @return
 */
(***)private Menu rolMenu(Rol rol) {
      Menu menu = new CompositeMenu("999", "raiz");
      Accion accion;
      Menu node;
      Iterator it = rol.getAcciones().iterator();
      while (it.hasNext()) {
            accion = (Accion)it.next();
            if (!accion.getUrl().equals("")) {
                  node = new
SimpleMenu(String.valueOf(accion.getCodigo()),
                              accion.getDescripcion().trim(),
                              accion.getUrl().trim(),
                              String.valueOf(accion.getCodigoPadre()));

            }
            else {
                  node = new
CompositeMenu(String.valueOf(accion.getCodigo()),
                              accion.getDescripcion().trim(),
                              String.valueOf(accion.getCodigoPadre()));
                  }
            buildMenu(node, menu);
      }
      return menu;
}//rolMenu()
```

Los métodos que acabamos de mostrar son sólo algunos de los que intervienen en el proceso de construcción del menú: en cada uno de los métodos que se invocan dentro de estos hay mucho más comportamiento. Por ejemplo, cuando en el método `getMenu(int idAplicacion, int idRol)` se invoca la clase `RolManager` para obtener el objeto `rol` (con la instrucción `RolManager.getInstance().getRol(idAplicacion,idRol);`), dentro de ese método `getRol()` se realiza, en caso de que aún no haya sido cargado, una

búsqueda de dicho rol y todas sus acciones en la base de datos a través de una serie de interacciones con los DAOs correspondientes. Sin embargo, nuestra intención en el presente trabajo es simplemente mostrar cómo están implementados los servicios relativos a la construcción del menú dentro del *framework*, y no brindar un detalle pormenorizado de todo el mecanismo.

## *Presentación del menú al usuario*

Por su parte, la presentación del menú disponible al usuario para su ejecución se construye, como dijimos, a partir de las utilidades provistas por DOJO Toolkit. Como vimos en secciones anteriores, cuando hablamos de la definición de las páginas usando `tiles-defs.xml`, cada vez que tengamos intención de que una página de nuestra aplicación contenga un menú para presentarle al usuario, deberá extender una plantilla que incluya el menú, por ejemplo la plantilla `tilesHMB.jsp`. Recordemos cómo estaba definida:

```
<definition name="alcalde.tilesHMB" path="/pages/tiles/tilesHMB.jsp">
  <put name="title" value="ALCALDE"/>
  <put name="header" value="/tiles/header.jsp"/>
  <put name="menuBar" value="/menu.jsp"/>
</definition>
```

Como podemos observar, dicha plantilla especifica que, para el atributo `menuBar`, insertará la página `menu.jsp`. Esta página `menu.jsp` es la encargada de armar y presentar el menú al usuario dependiendo del rol al que pertenece. La estructura de esta página es muy simple, sólo contiene unas pocas referencias a librerías de DOJO, como por ejemplo:

```
<!-- /menu.jsp
...
<script type="text/javascript" src="<%=request.getContextPath()%>/dojo-
0.4.1-ajax/dojo.js"></script>

<script type="text/javascript">
     dojo.require("dojo.widget.Menu2");
     dojo.hostenv.writeIncludes();
</script>

<style type="text/css">
     .dojoMenuBar2 {
          border-top:1px solid #d4d4d4;
     }
</style>

...
-->
```

Y, finalmente, la instrucción mediante la cual se invoca a los servicios del *framework* J2EE para la construcción del menú es como se muestra en las siguientes líneas:

```
<!-- /menu.jsp
```

```
...
<%
      int codRol =
((SelfUser)request.getSession().getAttribute(Constantes.USUARIO)).getIdRo
l();
      int codApp = Constantes.ID_APP_ALCALDE;
%>
<%=MenuManager.getInstance().getMenu(codApp, codRol,

      TipoImplementacion.DOJO_IMPLEMENTATION)%>
...
-->
```

## Funcionalidades implementadas a partir de la utilización de JNDI

Hay un aspecto muy importante de nuestra solución que se apoya en JNDI, ya que es justamente a través de este mecanismo que se localizan y obtienen los *data sources* que contienen las conexiones a la base de datos sobre la cual transige la aplicación. A continuación, detallaremos la forma en que esto está implementado en nuestra solución.

Como dijimos, en nuestra solución, las conexiones se obtienen de *data sources* (o *pools* de conexiones) definidos y administrados por el servidor de aplicaciones, en nuestro caso Apache Tomcat. Esta definición se debe realizar en el archivo **`server.xml`** que se encuentra en el directorio `Tomcat_HOME\conf`, como se muestra en el siguiente ejemplo:

```
<!-- /server.xml
...
 <!-- Global JNDI resources -->
 <GlobalNamingResources>
   <Resource name="jdbc/Alcalde_ADMIN"
     driverClassName="com.mysql.jdbc.Driver" maxActive="3" maxIdle="1"
     maxWait="5000" username="alcalde" password="alcalde"
     type="javax.sql.DataSource"
     url="jdbc:mysql://localhost/alcalde2005"
     validationQuery="select current_date"
   />
...
-->
```

En este ejemplo, la definición corresponde a un *data source* identificado con el nombre **`jdbc/Alcalde_ADMIN`**, y es a través de ese nombre que se lo va a recuperar desde la aplicación. Entre otras cosas, en la definición se configura la URL con el nombre del servidor de base de datos y el nombre de la base de datos propiamente dicha, el driver JDBC que se desea utilizar (relacionado con el motor de bases de datos utilizado, en nuestro caso MySQL), máximo de conexiones del *pool*, etc. Otra definición a tener en cuenta para la configuración y posterior obtención de los *data sources* es el archivo **`context.xml`**, que se encuentra en el directorio `DEPLOY_PATH\APP_PATH\META-INF`. En este archivo debería configurarse algo como lo siguiente:

```
<Context docBase="alcalde" path="/alcalde">
```

```
  <ResourceLink global="jdbc/Alcalde_ADMIN" name="jdbc/Alcalde_ADMIN"
      type="javax.sql.DataSource"/>
  <ResourceLink global="jdbc/Alcalde_SISTEMA" name="jdbc/Alcalde_SISTEMA"
      type="javax.sql.DataSource"/>
  <ResourceLink global="jdbc/JMENU" name="jdbc/JMENU"
      type="javax.sql.DataSource"/>
</Context>
```

Hasta aquí, lo relacionado con la definición de los *data sources.* Veamos ahora la forma de obtener una conexión dentro de la aplicación.

Debemos comenzar mencionando la clase `alcalde.struts.plugins.Conector`, que implementa la interfaz `PlugIn` y se inicializa cuando arranca la aplicación. `PlugIn` es el encargado, entre otras cosas, de levantar la configuración de los servicios del *framework* JEE (tal como lo describimos en secciones anteriores cuando hablamos de ANT y de la manera en que se utilizaba el archivo `configuration.xml` en nuestra aplicación) y de contener un Singleton para la clase `self.Self`, que es el objeto al que le vamos a solicitar una conexión cada vez que la necesitemos. De este modo, en un DAO de nuestra aplicación, cada vez que necesitemos una conexión, haremos `Conector.getSelf().getConnection(usuario.getDescRol())`, o directamente `Conector.getSelf().getConnection()`. La primer instrucción nos retornará una conexión disponible del *data source* asociado al rol del usuario, y el segundo ejemplo nos devolverá una conexión disponible del *data source* por defecto (asociado al rol `SISTEMA`).

Veamos los métodos `getConnection()` de la clase `self.Self` para entender mejor cómo interactúa con el servicio `ServiceLocator` del *framework* JEE para obtener la conexión pedida:

```
  /**
   * Método para obtener una conexión a la Base de datos,
   * para el rol por DEFECTO.
   * @return Un objeto Connection
   */
  public Connection getConnection() throws SQLException{
      Connection conn = null;
      ServiceLocator s = ServiceLocator.getInstance();
      try {
            conn = s.getConnection(this.getJndiJdbcDefault());
      } catch (Exception e) {
      }
      return conn;
  }

  /**
   * Método para obtener una conexión a la Base de datos, del DataSource
   * que se corresponde con el nombre que llega como parámetro, es decir,
   * para un rol específico.
   * @param datasourceName
   * @return
   * @throws SQLException
   */
  public Connection getConnection(String datasourceName) throws SQLException{
      Connection conn = null;
```

```
    ServiceLocator s = ServiceLocator.getInstance();
    try {
          conn = s.getConnection(datasourceName);
    } catch (Exception e) {
    }
  return conn;
}
```

Vale la pena mencionar que `ar.com.cinq.jee.servicelocator.ServiceLocator` es un servicio disponible en el *framework* JEE, que es también un Singleton, que además de localizar los objetos disponibles en un contexto valiéndose de la API JNDI, registra y mantiene las referencias a los objetos localizados. Esto mejora notablemente el rendimiento, ya que una vez localizados los objetos, si se los requiere nuevamente, sólo debe retornar la referencia a éstos sin tener que realizar una nueva búsqueda.

A continuación transcribimos el método disponible en él, donde se puede ver claramente el comportamiento al cual hacemos referencia:

```
/**
 * Retorna una conexión de base de datos a partir del nombre de un datasource.
 * @param datasourceName   proveedor de conexiones de base de datos
 * @return      una conexión de base de datos
 * @throws ServiceLocatorException
 */
public Connection getConnection(String datasourceName) throws
    ServiceLocatorException {
try {
    DataSource ds = datasources.get(datasourceName);
    if (ds == null) {
          JDBCConfig config =
          Configuration.getServiceLocatorConfig().getJdbcConfig();

          Context envCtx = new InitialContext(config.getEnvProps());
          ds = (DataSource)envCtx.lookup(config.getLookupRoot() +
          datasourceName);
          datasources.put(datasourceName, ds);
    }
    Connection conn = ds.getConnection();
    return conn;
}
catch (Exception e) {
    throw new ServiceLocatorException(e,
          "errors.servicelocator.connection.get", datasourceName);
}
}
```

Como vemos en la porción de código que antecede, es ahí mismo donde se utiliza la funcionalidad provista por JNDI para hacer la búsqueda de los objetos (`lookup`). De esta forma, algo que es costoso de administrar por las aplicaciones, se resuelve de manera relativamente sencilla y eficiente.

## Implementación del *framework* Struts

Para apreciar la utilización del *framework* Struts en nuestra solución, estimamos conveniente presentar en detalle una funcionalidad del sistema desde que se invoca hasta que se retorna el resultado, pasando por todas las "capas" del modelo MVC.

Buscamos hacer foco en cómo es el flujo entre las distintas capas del modelo MVC, por lo que hemos decidido seleccionar como ejemplo una funcionalidad sencilla, "listar los agentes", dentro del sistema de liquidación de haberes.

Para comenzar, debemos mostrar cómo está definida la página en el archivo `tiles-defs.xml`:

```
<!-- tiles-defs.xml
...
  <!-- Definiciones de páginas -->

  <definition name="alcalde.agentes"
              extends="alcalde.tilesFiltroDetalle">
     <put name="title" value="Sueldos - Agentes"/>
     <put name="filtroContent" value="/jsp/agentesFilter.jsp"/>
     <put name="detalleContent" value="/jsp/agentesBrowser.jsp"/>
     <put name="bottomBar" value="/jsp/bottomBar.jsp"/>
  </definition>
...
-->
```

Luego, mostramos la parte del archivo `struts-config.xml` que contiene la configuración del mapeo de la etiqueta `agentes` con la clase `AgentesAction`, la cual realiza el procesamiento del requerimiento cuando tiene lugar la invocación:

```
<!-- /struts-config.xml
...
    <action name="agentesFilterForm" path="/agentes" scope="session"
      type="ar.com.cinq.sueldos.struts.actions.AgentesAction">
       <forward name="success" path="alcalde.agentes" redirect="false" />
       <forward name="fallo" path="alcalde.error" redirect="false" />
     </action>
...
-->
```

De este modo, la definición de esta funcionalidad en la capa "vista", es decir en la página JSP, se reduce a lo siguiente:

```
<!-- Página JSP
      ...
      <html:link href="agentes.do" styleClass="link">
        <html:img
      src="${pageContext.request.contextPath}/images/link.gif"
                  styleClass="image" alt="Agentes"/>Agentes
       </html:link>
      ...
-->
```

Por último, mostramos la clase `AgentesAction`, que es la encargada de procesar el requerimiento cuando se lleva a cabo la invocación:

```
public class AgentesAction extends Base_Action {

  public AgentesAction() {}

  public ActionForward execute(ActionMapping mapping, ActionForm form,
      HttpServletRequest request, HttpServletResponse response) throws
      java.lang.Exception {
    DynaActionForm formulario = (DynaActionForm)form;
    Self self = ar.com.cinq.sueldos.struts.plugins.Conector.getSelf();
    utilitarios.Contenedor contenedor = new utilitarios.Contenedor();
    String pageForward ="fallo";
    try {
      java.util.Vector elementos = new java.util.Vector();

      String condicion = (String) formulario.get("condicion");
      String mostrarAgentes = (String) formulario.get("mostrarAgentes");
      String fechaDesdeStr = (String) formulario.get("fechaDesde");
      String fechaHastaStr = (String) formulario.get("fechaHasta");

      if ((mostrarAgentes != null) && (!mostrarAgentes.equals(""))) {
        //Order By
        if (condicion != null) {condicion = condicion +
                                      " order by LegajoKey";}
        else {condicion = " order by LegajoKey";}

        java.util.Date fechaDesde = cinq.Generic.strToDate(
                                      fechaDesdeStr, "dd/MM/yyyy");
        java.util.Date fechaHasta = cinq.Generic.strToDate(
                                      fechaHastaStr, "dd/MM/yyyy");

        elementos = ar.com.cinq.sueldos.negocio.AgenteManager.agentes(
                    self.getModule(Constantes.MODULO_PERSONAL), self,
                    fechaDesde, fechaHasta, condicion);

      }//if ((mostrarAgentes != null) && (!mostrarAgentes.equals("")))

      contenedor.setRango(50);
      contenedor.setElementos(elementos);
      request.getSession().setAttribute("contenedor", contenedor);

      return mapping.findForward("success");
    }//try
    catch (Exception e) {
      System.out.println("Error: AgentesAction - " + e.getMessage());
      pageForward = "fallo";
    }//catch
    return mapping.findForward(pageForward);
  }//execute
}
```

En la porción de código remarcada, se puede observar la forma en que se consulta la capa de negocio para recuperar los agentes a mostrar en la interfaz.

## Uso de JSON en nuestra solución

En estas líneas, pretendemos mostrar la puesta en escena de los conceptos expuestos en la parte de tecnologías JSON. Aquí, exponemos parte del código donde se puede apreciar la utilización de esta tecnología para el intercambio de información.

Para mostrar JSON en acción, debemos ver dos partes elementales. Una es cómo se construyen los objetos pasa ser transmitidos bajo esta estructura, y la otra es la forma en la que se recupera esta información para ser visualizada en la página JSP.

Veamos primero, entonces, cómo se construyen los objetos desde un `Action` de la aplicación, por ejemplo, cuando se necesita mostrar un combo de localidades:

```
public ActionForward getLocalidades(ActionMapping mapping,
      ActionForm form, HttpServletRequest request,
      HttpServletResponse response) throws Exception {
     String toWrite = "";

     try {
      Collection localidades = collectionProvider.getLocalidades(
           self.getModule(alcalde.Constantes.MODULO_INGRESOS),
      self);

      toWrite = JSONResponseBuilder.buildLocalidades(localidades);
     }//try
     catch (ApplicationException ae) {
      toWrite = JSONResponseBuilder.buildExcepcion(ae);
     }//catch
     finally {
      try {
        JSONResponseBuilder.writeHTTPResponse(response, toWrite);
      }
      catch (Throwable t) {}
       return null;
     }//finally
}//getLocalidades()
```

Como vemos en la parte de código remarcada, el `Action` se vale de un objeto que realiza la construcción de la respuesta con la estructura JSON. A continuación, vemos el método `buildLocalidades` de dicha clase para ver en detalle cómo lo lleva a cabo.

```
<!-- JSONResponseBuilder
     ...
     public static String buildLocalidades(Collection items) {
          try {
           Collection localidades = new ArrayList();
           JSONObject dtoJO = null;

           JSONObject jsonObj = new JSONObject();
```

```
            jsonObj.put("cantidad", items.size());
            Iterator<LocalidadDTO> it = items.iterator();
            while (it.hasNext()) {
                  dtoJO = new JSONObject();
                  dtoJO.put("LocalidadKey", it.next().getLocalidadKey());
                  dtoJO.put("Nombre", it.next().getNombre());

                  localidades.add(dtoJO);
            }
             jsonObj.put("localidades", localidades);

             return jsonObj.toString();
           }
           catch (Exception e) {
            return buildExcepcion(e);
           }
      }//buildLocalidades()
      ...
-->
```

Por último, veamos cómo se realizan los requerimientos y cómo se toman desde las páginas JSP.

```
<script language="javascript">
      function getLocalidades(form) {
        clearSelectLocalidades();

        dojo.xhrGet({
            url: "<%=request.getContextPath()%>/localidades.do?method=
                  getLocalidades",
            handleAs: "json",
             load: processLocalidades,
            error: basicErrorHandler
        });
      }

       function processLocalidades(data, ioArgs) {
         if (checkException(data, "tdMensajeError")) {
            return true;
         }
         for (i=0; i<data.cantidad; i++) {
           addObjToSelect(document.forms[0].localidadKey,
                        data.localidades[i].localidadKey,
                        data.localidades[i].Nombre);
         }
         clearException("tdMensajeError");
    }
</script>
...
<table width="100%">
  <tr>
    <td align="left" class="fontTitle" width="40%">
      <html:select property="localidadKey" onblur="getLocalidades(form)">
```

```
            <option value="-1111" class="botonera">Seleccionar...</option>
        </html:select>
      </td>
    </tr>
...
```

## Impresión de reportes con JasperReports

En cuanto a la generación de reportes y listados con información de la aplicación ya sea para visualización, para impresión o para generación de documentos de distintos tipos, utilizamos JasperReports. Como ejemplificación del uso de esta herramienta, veremos la opción del módulo de Liquidación de Haberes a través de la cual se imprime el recibo de haberes de un legajo determinado.

En este punto, debemos mencionar también que, para el desarrollo de los *templates*, utilizamos la herramienta gráfica **iReport** que, además de ser *open source*, es la más difundida. Esto simplifica muchísimo la tarea de diseñar nuestros reportes.

Para comenzar, podemos mencionar que, antes que nada, se debe generar el *template* con el diseño que va a tener el recibo. Esto incluye la definición de los **campos** que van a ser proyectados en la consulta de base de datos, si es que utilizamos esta manera de recuperar la información, o bien los nombres de los atributos de los objetos que llegarán en la colección, si es que decidimos utilizar colecciones de objetos que llegan desde la capa de negocio. En este caso, utilizamos directamente una consulta sobre la base de datos, para lo cual, desde el `Action`, se le pasa la conexión a utilizar, en el método que realiza el "llenado" del reporte, con los datos recuperados de la siguiente manera: `JasperFillManager.fillReport(jasperReport, parameters, conn)`.

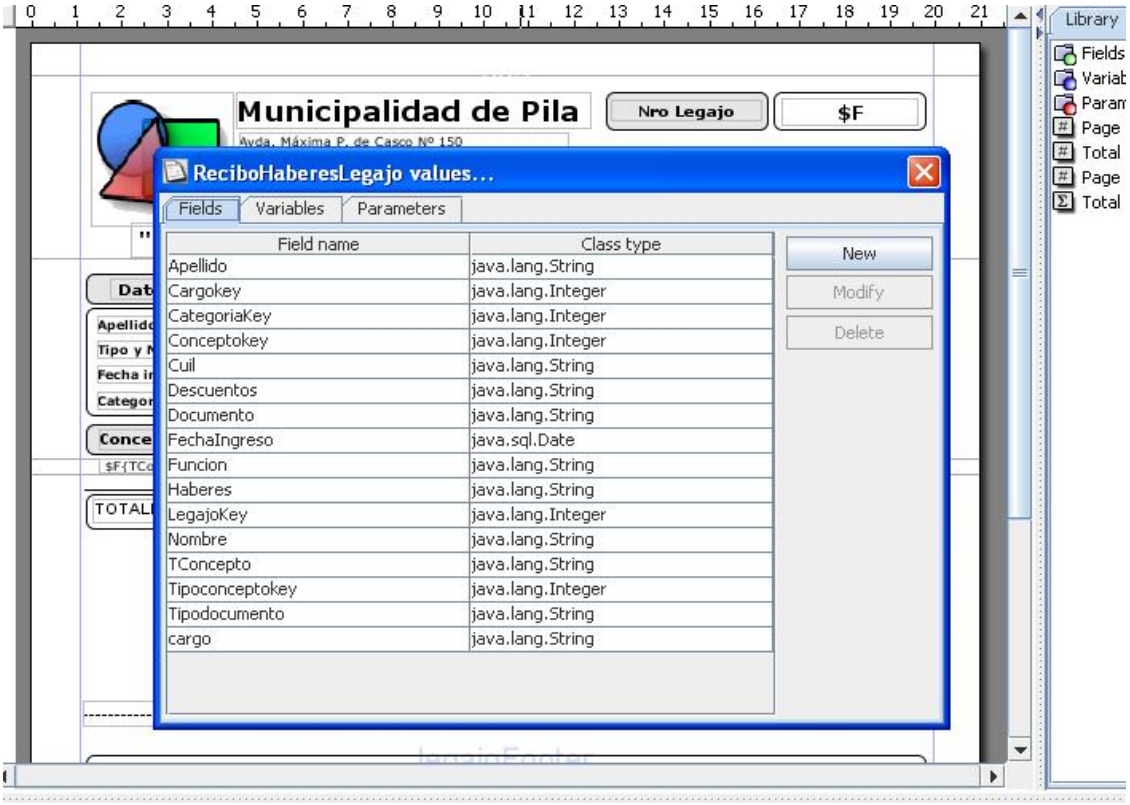


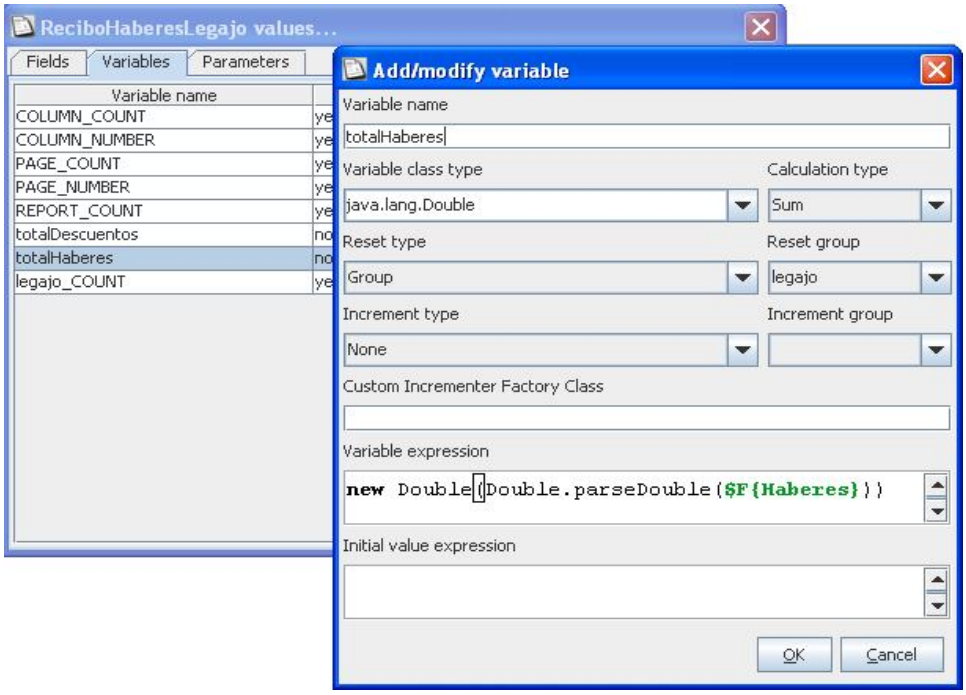

Figura 26. Valores para el recibo

Además de los campos, en este punto se deben definir también las **variables** y los **parámetros**. Las **variables** se definen con el objetivo de, por ejemplo, totalizar datos en función de algún determinado campo, o bien definir un campo calculado para ser impreso bajo determinadas situaciones. Por otro lado los **parámetros** son definidos tanto para ser utilizados dentro de las consultas de base de datos como para ser impresos en el reporte, pero, a diferencia de las variables, son pasados en el método cuando se llena la plantilla en el mismo método en el que se pasa la conexión:

```
<!-- ReciboLegajoReportAction
...
      //Agregar los parámetros
      parameters.put("liquidacionKey", new Integer(liquidacionKey));
      parameters.put("liquidacion", nombreLiquidacion);
      parameters.put("legajoKey", new Integer(legajoKey));
      parameters.put("fechaHastaLiq", liq.getFechaFin());
      parameters.put("fechaHastaNull",
                        Generic.strToDate(Constantes.FECHA_HASTA_NULL,
                        Constantes.FORMATO_FECHA));
       JasperFillManager.fillReport(jasperReport, parameters, conn).
...
-->
```

En cuanto a la definición de las **variables**, como ya dijimos, se realiza directamente en el diseño de la plantilla, y se puede hacer en base a ciertas funciones preestablecidas (por ejemplo, `SUM`, `AVERAGE`, `StandarDeviation` y `Variance`), así como también el tipo de

incremento y el nivel de "reseteo" (en qué momento vuelven a cero), aplicando estos parámetros a campos definidos en el reporte. Esto se puede ver en la imagen siguiente, donde mostramos la definición de la variable `totalHaberes`, sobre la cual se define la




función `SUM` a ser aplicada al campo `Haberes`, con reseteo definido a nivel de grupo:

Figura 27. Definición de la variable totalHaberes

Para concluir, en lo referente a la creación de la plantilla o *template*, se ubican los componentes de diseño de acuerdo a lo que se desee visualizar en el reporte:


Figura 28. Plantilla del recibo de haberes

Por último, mostraremos cómo se realiza la ejecución para la visualización por pantalla del recibo, tarea que se lleva a cabo una vez que la plantilla a sido "llenada" (`fillReport()`) a través del método `exportReportToPdf()` de la clase `JasperExportManager`:

```
<!-- ReciboLegajoReportAction
      ...
      JasperPrint jp = JasperFillManager.fillReport(jr, parameters,
      conn);
      JasperExportManager jem = new JasperExportManager();
       byte buf[] = jem.exportReportToPdf(jp);
       response.setContentType("application/pdf");
       response.setContentLength(buf.length);
       response.setHeader("content-disposition", "inline;
                          filename=report.pdf");
       java.io.OutputStream out = response.getOutputStream();
       out.write(buf, 0, buf.length);
       out.close();
      ...
-->
```

Luego de la ejecución de esto, el reporte se visualiza en la pantalla con el siguiente formato:

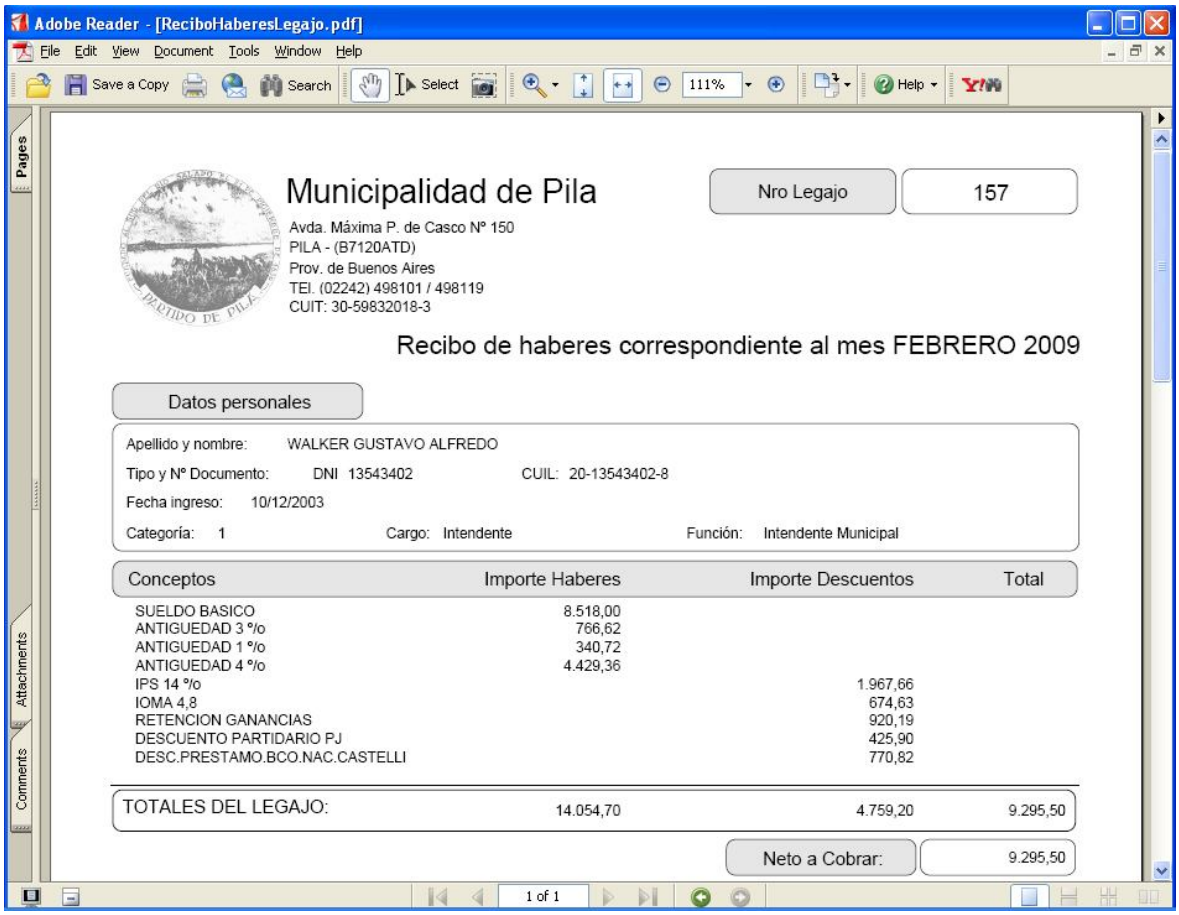



Figura 29. Visualización en pantalla del recibo de haberes

# Conclusión

Para concluir, volveremos sobre los dos ejes principales de nuestra tesis: por un lado, el análisis y la investigación de soluciones FOSS para gobierno digital a nivel mundial, y, por otro, el desarrollo del prototipo de una solución FOSS para la gestión municipal.

En cuanto al primer objetivo, observamos que es sumamente necesaria la implementación de este tipo de soluciones en el ámbito de la administración pública local. Esta conclusión se desprende de nuestra investigación del fenómeno FOSS (software libre y de código abierto) en las administraciones públicas a nivel mundial, y del auge que está teniendo principalmente en Europa y Latinoamérica. Otro aspecto de suma importancia es que cada una de las características de FOSS se adapta perfectamente al ambiente gubernamental y cobra mayor relevancia cuando se las aplica a él, debido al tipo de información que se maneja: es tanto pública como propiedad de los ciudadanos. Por esta razón, consideramos que el Estado **debe** garantizar la democratización del acceso a la información.

Por su parte, con el desarrollo del prototipo de nuestra solución FOSS para la gestión municipal, no hemos hecho más que poner en práctica aquello que investigamos, ya que existen en la actualidad (y las hemos utilizado), una cantidad significativa de herramientas FOSS, tanto para el desarrollo como para el soporte operativo, capaces de dar soluciones de *software* robustas a cualquier organismo público, sin necesidad de obligar a una dependencia tecnológica y económica de empresas privadas. En este punto, es importante resaltar que para llevar a cabo la implementación de nuestra solución no sólo hemos utilizado las herramientas FOSS disponibles, sino que además el trabajo ha sido realizado siguiendo las tendencias internacionales analizadas y descriptas en los capítulos anteriores. Otro aspecto que vale la pena mencionar es que si bien nosotros hablamos de prototipo, por tratarse de la implementación de dos módulos de una aplicación en cuyo análisis se determinó que estaría formada por cinco, cada módulo en cuestión es en sí mismo una aplicación con una complejidad y un volumen de trabajo considerable. De hecho, uno de los módulos implementados es el de "Liquidación de Haberes", el cual se constituye en una aplicación completa capaz de gestionar perfectamente y sin inconvenientes, el registro de los movimientos de personal de un municipio como así también la liquidación de sus haberes, con la complejidad que esto trae aparejado.

## Trabajos futuros

Debido a la naturaleza de los organismos públicos, que, en muchos casos, cuentan con estructuras y problemáticas similares, es infinita la cantidad de proyectos que podrían y deberían implementarse con *software* libre y de código abierto (FOSS), y en forma conjunta, compartiendo tanto la experiencia y el conocimiento de los procesos, como así también su financiamiento.

Como puntapié inicial, se podría continuar avanzando en el desarrollo del resto de los módulos analizados, para construir así una solución integral que se constituya en una opción más que interesante para la gestión de municipios de diferente envergadura.