MUNICIPALIDAD DE ROSARIO

Sistema Integrado de Administración Tributaria SIAT

Arquitectura SIAT

Noviembre 2007

Contenido

Características del documento

Versiones

Fecha	Version	Descripcion	Autor
01/05/07	1.0	Inicial	tecso:
08/11/07	1.1	Incorporación de cambios de Diseño	tecso:

Objetivos del documento

El objetivo del documento es documentar las características de la arquitectura para el desarrollo y despliegue de aplicaciones J2EE.

Introducción

Entendemos por Arquitectura del Software a la organización fundamental de un sistema en términos de: sus componentes, las relaciones entre ellos y el contexto en el que se implantarán, y los principios que orientan su diseño y evolución. (IEEE Std 1471-2000)

De este modo, la Arquitectura del Software establece una visión de alto nivel sobre una serie de aspectos:

- Módulos principales
- Responsabilidades que tendrá cada uno de estos módulos
- Interacción que existirá entre dichos módulos
- Control y flujo de datos
- Secuenciación de la información
- Protocolos de interacción y comunicación
- Ubicación en el hardware

El objetivo de la visión de alto nivel es aportar elementos que ayuden a la toma de decisiones y, al mismo tiempo, proporcionar conceptos y un lenguaje común que permitan la comunicación entre los equipos que participen en un proyecto y faciliten la comprensión del software a nuevos participantes.

Existen varias formas de documentar estos "aspectos relevantes". En este documento presentamos una visión general y luego una serie de vistas particulares que presentan distintos enfoques para su comprensión.

Vista general

En esta sección se describen las características generales de la arquitectura. Inicialmente se nombran las premisas que fueron tomadas en cuenta para el diseño. Luego las características generales y finalmente, aspectos técnicos generales.

Premisas

Como enunciamos anteriormente, la arquitectura responde a una serie de premisas tomadas como base para establecer los lineamientos generales en cuanto a componentes y diseño en general.

Dichas premisas, pueden ser agrupadas de acuerdo a los siguientes aspectos:

- Relativos a las características generales de las aplicaciones
- Relativos a las características del ciclo de vida de los proyectos
- Relativos a los requerimientos No-Funcionales, asociados a las aplicaciones a construir.

A continuación se describen brevemente cada uno de ellos:

Características de las aplicaciones a construir

Las aplicaciones a desarrollar con esta arquitectura son:

- Puramente Transaccionales.
- La interacción de los usuarios es a través de lnterfaces Web.
- De gran porte.
- Participan un gran número de analistas, desarrolladores y testeadores.
- Se conciben como un conjunto de módulos independientes que interactúan con una única base de datos.

Características del ciclo de vida de los proyectos

Los ciclos de vida asociados con las aplicaciones a construir, están basados en el Proceso Unificado.

De este modo, existen cuatro fases: Incepción, Elaboración, Construcción y Despliegue. Durante cada una de ellas, se desarrollan, al mismo tiempo, varias disciplinas.

Inicialmente, se busca acotar los requerimientos del cliente respecto de las características de la aplicación. Para ello, se construyen prototipos navegables.

Sobre esta base, se considera al prototipo como cliente de Servicios entregados por la capa de negocios y se define la Interface de los Servicios y las estructuras de datos de entrada/salida.

Luego, se construye el modelo de negocios que debe soportar la funcionalidad exigida por los servicios e interactuar con la base de datos.

Requerimientos No-funcionales

Control de acceso

- Los usuarios que ingresen al sistema deben ser identificados y autenticados, y las funciones del sistema deben habilitarse según su sector y perfil en cada una de las funciones que así lo requieran.
- Los usuarios que realizaron operatorias sobre el sistema deben poder ser identificados posteriormente.
- Deben quedar registradas todas las excepciones al tratamiento normal de las operaciones que provee el sistema (transigencia de controles).
- Debe permitir realizar definiciones por grupos y asignar los usuarios a los mismos.
- Deberá disponer de un administrador de tareas de seguridad
- Poseer un nivel de seguridad sobre grupos de datos que impida su alteración por parte de usuarios no autorizados.
- Debe realizar el cierre automático de una sesión de trabajo por inactividad en un lapso de tiempo configurable.

Usabilidad

- Debe disponerse de ayuda en línea sensible al contexto.
- El sistema deberá estar íntegramente en español, tanto los literales de las pantallas/ventanas y reportes como los mensajes específicos y ayudas en línea.
- Las interfaces de usuario deben ajustarse a criterios de aceptación del diseño de interfaces.
- El usuario debe poder recordar como operar el sistema entre usos del mismo.
- El sistema debe anticipar y prevenir los errores de usuario más comunes.
- El sistema debe ayudar al usuario a recuperarse ante los errores, y sugerir posibles acciones a seguir.
- El usuario debe percibir que el sistema le facilita su trabajo.

Performance

- Durante la interacción en la carga de datos y consultas simples el tiempo máximo de respuesta del sistema no deberá superar los 10 segundos.
- El sistema deberá asegurar una performance estable, sin variaciones aleatorias o sin causa imputable al hardware o al sistema operativo

Mantenibilidad

- Los cambios potenciales al sistema deben involucrar una cantidad mínima de componentes distintos.
- Deberá estar modularizado y diseñado teniendo en cuenta la reusabilidad de componentes.

Características generales

En función de los fundamentos enunciados, se define como característica esencial de la arquitectura la necesidad de contar con una adecuada separación de capas que permita definir claramente las responsabilidades de cada componente de la aplicación y, transitivamente, de los grupos de desarrollo que participen del proyecto.

A continuación se describe como se dividirá la aplicación en Capas y luego, en como se distribuirán los distintos módulos funcionales en cada una de ellas.

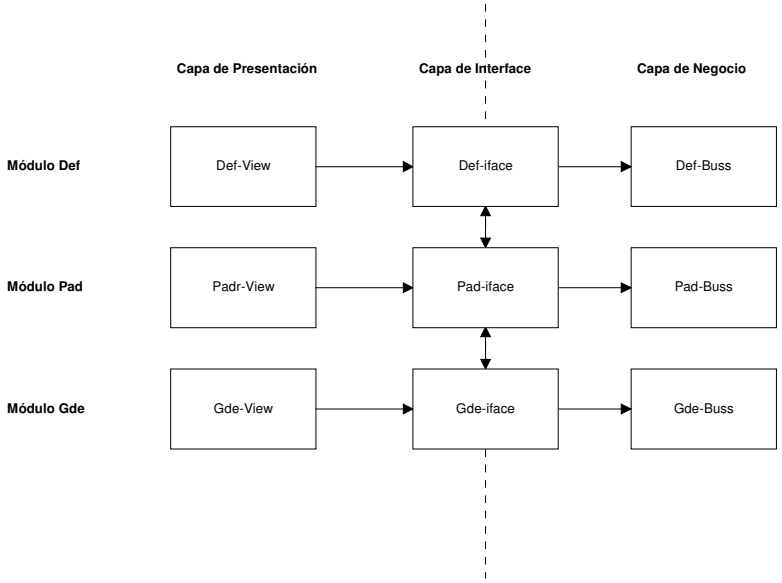
Separación de Capas

La arquitectura está separada en tres capas: Vista, Interface y Negocio.

La capa de Vista actúa como cliente de los servicios descriptos en la capa de Interface e implementados en la capa de Negocio.

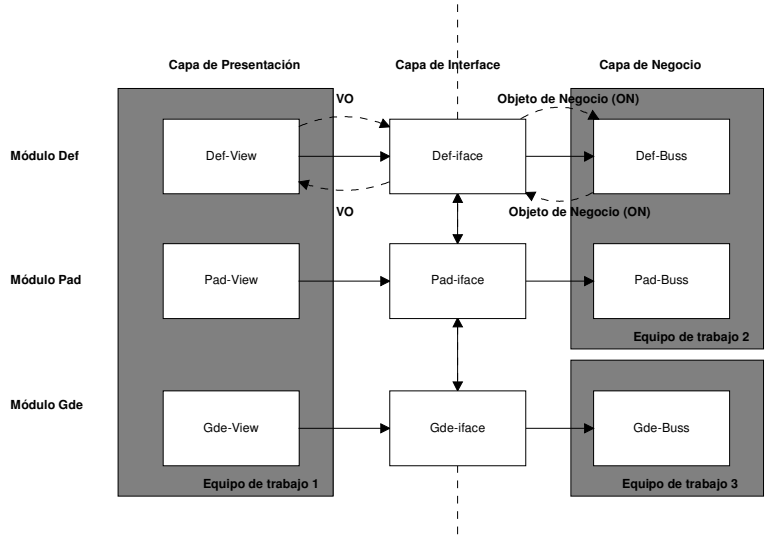
De este modo, la capa de interface actúa como un contrato entre la Vista y el negocio.

La capa de Negocio, implementa los servicios.



Como resultado de la separación de capas (y de módulos), obtenemos una separación de responsabilidades entre la Vista y el Negocio y de este modo, es posible distribuir convenientemente los equipos de trabajo.

A continuación, se muestra una posible separación de equipos:



En todos los casos, la capa de Interface actúa como un contrato que permite desacoplar las capas.

División funcional en Módulos y Submódulos

Desde el punto de vista funcional, el modelo de análisis se divide en Módulos. Cada módulo se divide en Submódulos y en cada submódulo se encuentran los casos de uso.

Por ejemplo los CU del *Mantenedor de Recurso*, y *Mantenedor de Tipo Objeto Imponible* se encuentran dentro del *Modulo Definición*, en los S*ubmódulos Gravamen y Objeto Imponible* respectivamente.

Implementación del patrón: Value-Object

Dentro de la capa de negocio, existirán una serie de objetos (algunos persistentes y otros no) que son quienes poseen el conocimiento y el comportamiento requerido para cumplir con los requerimientos planteados al sistema.

Dichos objetos de negocio, "viven" o "existen" dentro de la capa de negocios y se espera que no "salgan" al exterior.

Si lo hicieran, significaría que la capa de Vista tiene pleno conocimiento sobre los objetos del negocio y, en ese caso, no existiría la separación de responsabilidades enunciada anteriormente.

Entonces, existirán objetos de negocio en la capa de negocios y Value-Objects en la capa de Interface.

Los Servicios son descriptos en términos de Value-Objects y los **Objetos de Negocio tienen el "conocimiento" para transformarse "automáticamente" en Value-Objects.**

Características Técnicas

A continuación se describen las características generales de la arquitectura desde el punto de vista de las herramientas a utilizar , la modularización y de las responsabilidades de cada módulo.

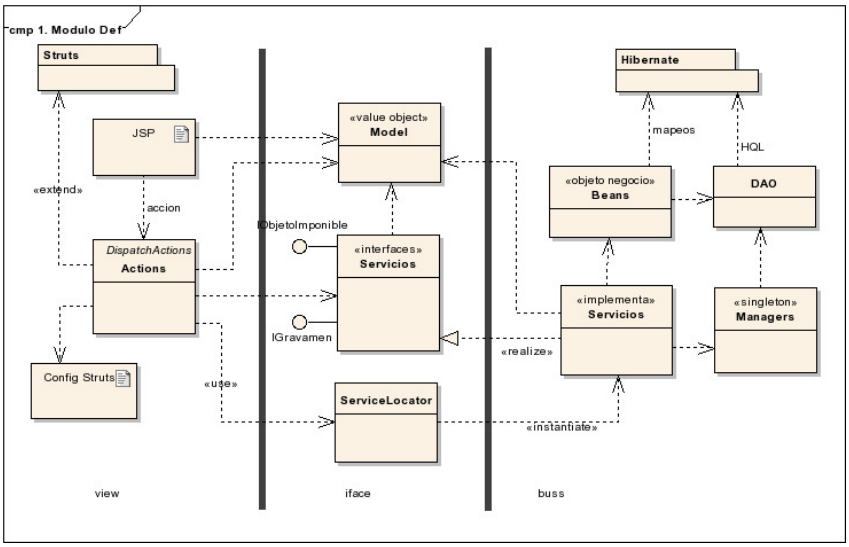
Herramientas

Paquete de Software	Funcionalidad General
Utiliza los frameworks	> Struts 1.2.9 > Hibernate 3.2.5 > Itext 1.2.4 > Apache FOP > Log4j 1.2.13
Posee una serie de servicios tipo Fachada que encapsulan el acceso a:	> Seguridad Web 2.0.1.3 > Servicio de Georeferenciación > mcr-framework
Utiliza:	> Compilador: Java JDK 5.0 > HTTP: HTML 4.01 Strict y CSS2 (compatibilidad total con el standard W3C). > Tecnologías: JSP, XML
Corre sobre:	> Servidor de Aplicaciones: Tomcat 5.5 > Base de Datos: Informix Versión 10
Se utiliza como herramienta de desarrollo:	> Eclipse con los plug-ins: . Ant . Junit . Administración de Tomcat

Modularización de la aplicación

Desde el punto de vista de la arquitectura, cada módulo de SIAT estará implementado como un paquete que a su vez se divide en tres subpaquetes *view, iface, buss*. Ademas cada paquete de módulo podrá tener un submoudlo extra *ws* correspondiente a las funcionalidades expuestas como web services del SIAT.

El siguiente diagrama ilustra las clases y componentes más importantes que forman parte de un módulo. Este mismo diagrama se repite para cada módulo del SIAT.



El paquete *view* de un módulo, es quien implementa las funciones de la vista, posee las clases Action de Struts, definiciones de constantes de la Vista, los archivos JSP y los archivos de configuración de Struts. En cada caso, son las partes necesarias para implementar específicamente el módulo en cuestión. Las clases Action del *view* reciben las acciones ejecutadas por los usuarios en los JSP e invocan a los servicios de SIAT utilizando el paquete *iface*.

El paquete *iface* de un módulo, expone un modelo de objetos de tipo "Value Object" utilizados para comunicar la Vista con el Negocio. Además *iface,* presenta la definición de las interfaces de negocio del modulo, así como los mecanismos de entregar las implementaciones que requiera la vista. En *iface*, también se encuentra definiciones de constantes utilizadas en ambas capas, *i.e.* Constantes de Error y Mensajes, y de Seguridad.
Por lo general, el paquete *view* toma los datos ingresados por el usuario, los transfiere al modelo de objetos de *iface*, y llama a la capa *buss* enviando estos objetos e información del usuario logueado.

El paquete *buss*, es quien realiza la lógica de negocio del SIAT. Recibe como entrada los "Value Object" del *iface* (VO) enviados por la vista, datos de contexto del usuario y la acción realizada en la vista. Básicamente el servicio instancia los Objetos Negocio a partir de los VO, y delega la lógica en el modelo de Objetos de Negocio (ON). Los ON realizan la funcionalidad delegando en otros ON, y en la capa DAO. Existe un componente más en *buss*, que son las clases Singleton Manager, dichas clases absorben las funcionalidades que no puede absorber ningún ON. Más adelante en la Vista Lógica se explican en detalle cada componente del Negocio.

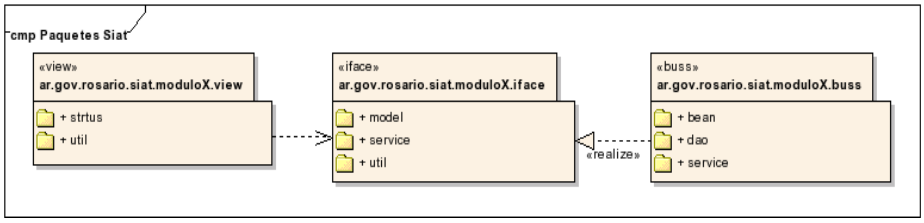
Ocasionalmente un módulo podrá tener un paquete extra correspondiente a los web services de SIAT. Los web services se implementan como simples componentes adaptadores de la interfaz de servicio del módulo. De esta manera el paquete *ws* y el *view* tiene similares comportamientos respecto de sus responsabilidades la diferencia es que para el caso *view* las acciones las dispara un usuario con su navegador, y para el caso *ws*, las acciones las dispara el protocolo SOAP.

Vista lógica

En esta sección se describen los componentes desde el punto de vista lógico. Inicialmente se describen los componentes a un nivel general, luego se describen los componentes de Demoda y finalmente los componentes de las tres capas de SIAT.

Componentes

Las aplicaciones a desarrollar con esta arquitectura, poseen tres capas bien diferenciadas: Vista, Interface y Negocio.



La capa de Vista, conoce las interfaces de servicios y los Value Objects (model) expuestos por la capa iface. Cuando requiere una implementación de un servicio, se la solicita a un ServiceLocator, también comprendido en la capa iface.

La capa de negocios brinda una (o varias) implementación de la interfaz de Servicios que es utilizada por la capa de Vista. Dichos servicios pueden entenderse como un conjunto de funciones que pueden invocarse, y no mantienen estado. En este caso, dichas implementaciones están basadas en el framework Hibernate.

Para obtener una total independencia, entre la capa de Vista y la de negocio, se implementa el patrón Value Object (model). La capa de vista no puede hacer directamente ninguna modificación ni consulta sobre la persistencia; cualquier operación debe ser solicitada vía servicios a la capa de negocio, enviando eventualmente como parámetros los Value Objects (model) correspondiente a los objetos persistentes involucrados.

La capa de negocio sólo mantiene estado en sus objetos persistentes; cualquier estado transitorio (p.ej. estado de una operación en proceso) se mantiene en la vista, y es enviado a negocio en cada invocación a servicio.

Cada invocación a servicio es, atómica y aislada. Existe una correspondencia entre un Request lanzado contra la capa de vista y toda la transacción ejecutada. (incluso a nivel de actualización de datos)

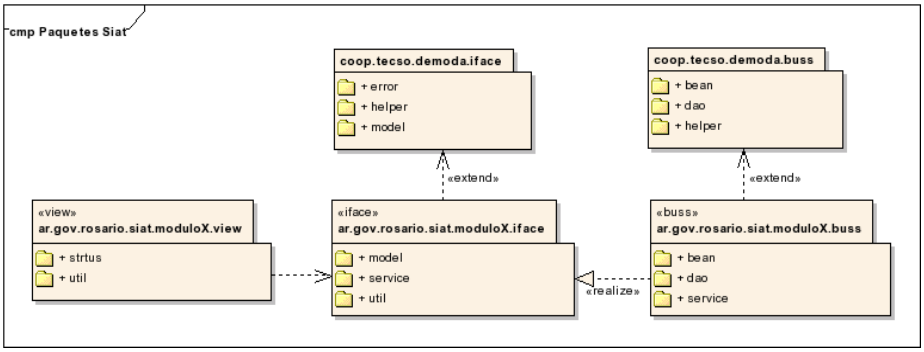
La funcionalidad y lógica de la aplicación está en los Objetos del Negocio (Profesional, Turno, Centro de Salud, etc.) Los servicios son sólo fachadas a esa funcionalidad, que exponen lo que el negocio decide dejar disponible para la vista. Cada objeto de Negocio posee su Value Objects. Además, hay otros Value Object que empaquetan objetos de negocio (como el Page)

Se define, para cada módulo una clase (tipo Singleton) que agrupa la funcionalidad que no puede ser absorbida por ningún objeto en particular. De este modo, se asegura que no se implemente lógica de negocio en el servicio.

Arquitectura Demoda

Existe un pequeño framework, que es simplemente un agrupamiento de clases según sus responsabilidades, que denominamos: Demoda. Básicamente definen Clases Base abstractas que luego deberán ser implementadas para cada aplicación

El esquema anterior, con el agregado de Demoda, quedaría:



Demoda presenta dos paquete una para la capa *iface* y otro para la capa *buss*.

Dentro del paquete *iface* las clases se agrupa en los paquetes:

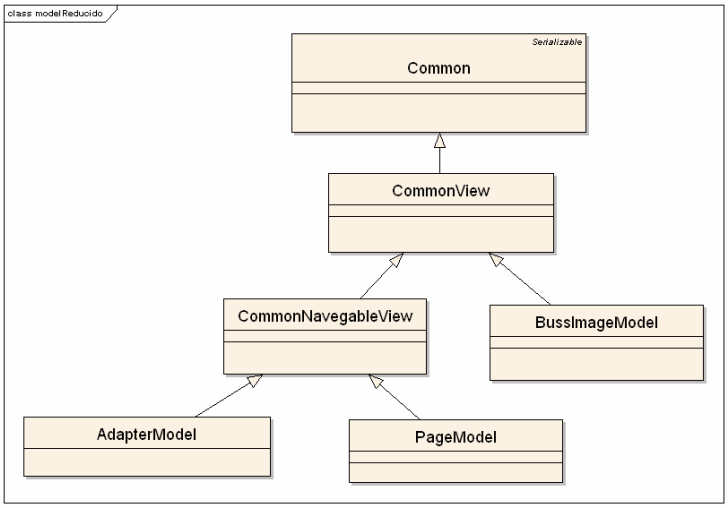
- ***Model:***
  Posee las clases base de la capa *iface* del framework. La capa *iface* de SIAT y cualquier aplicación que utilice demoda deberá extender de las clases de estos paquetes, para obtener los beneficios de demoda a la hora de utilizar el patrón DTO.
- ***error***
  Pequeña clase abstracta expone métodos para trabajar con mensajes de error y genéricos.
- ***helper***
  Una serie de clases de ayuda disponibles en la interfaz para ser usadas tanto desde la vista como en el negocio. Mayormente son clases que ayudan al formateo y conversión de tipos.

Dentro del paquete *buss* las clases se agrupa en los paquetes:

- ***bean***
  Posee las clases base de la capa *buss* del framework. La capa *buss* de SIAT y cualquier aplicación que utilice demoda deberá extender de las clases de estos paquetes, para obtener los beneficios de demoda a la hora de realizar la persistencia y usar el patro DTO.
- ***dao***
  Clases para simplificar la implementación de la capa DAO de las aplicaciones que usen demoda.
- ***helper***
  Clases de ayuda para la implementación de la capa de negocio de las aplicaciones que usen demoda.

Modelo de Clases de Demoda

Demoda cuenta con una serie de clases que brindan un marco de trabajo para cada capa. A continuación vemos las clases fundamentales de la capa iface:



La responsabilidad fundamental de cada una de estas clases se describe a continuación:

- La clase **Common** es la encargada de definir propiedades y métodos comunes (*e.g.*: la propiedad "id" que es utilizada como identificador único para los Value Objects). De ella extienden todas las clases del iface así como la clase BaseBo de la capa buss.

- La clase **CommonView** contiene una definición básica de propiedades que serán utilizadas para decidir la habilitación o visibilidad de un elemento de la vista (comúnmente llamadas banderas de seguridad o simplemente banderas).

- La clase **BussImageModel** que es de la cual extienden los Value Object y que contiene propiedades que serán utilizadas en estos (*e.g.*:"usuario", "fechaUltMod" usadas para auditoria y "estado" usada para contener el estado de las entidades).

- En las aplicaciones las clases que extienden de BussImageModel se las llama Clases Value Object o VO, y llevan el nombre con sufijo VO. Por ejemplo: FacturaVO, FacturaItemVO, etc.

- La clase **CommonNavegableView** que contiene las propiedades necesarias para almacenar la información de navegación.

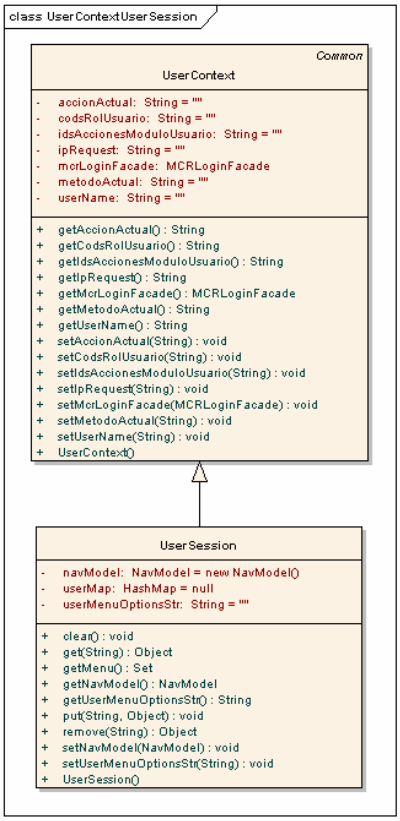
- La clase **PageModel** que contiene propiedades para manejar una lista de resultados y su paginación. De esta extenderán todas las clases que precisen un formato que incluya filtros y una lista de resultados y contendrán los datos a representar en la página.

En las aplicaciones, las clases que extienden de PageModel se las llama clases SearchPage.

- La clase **AdapterModel** que contiene propiedades de manejo de seguridad. De esta extenderán todas las clases que precisen un formato libre y contendrán al igual que el SearchPage los datos a representar en la página (comúnmente denominadas clases Adapter).

Las clases Adapter se utilizan en las aplicaciones para componer clases BussImageModel. Por ejemplo, una clase de FacturaAdapter, tendrá un atributo de tipo FacturaVO.

La capa *iface* también posee dos clases usadas para el manejo de la sesión:

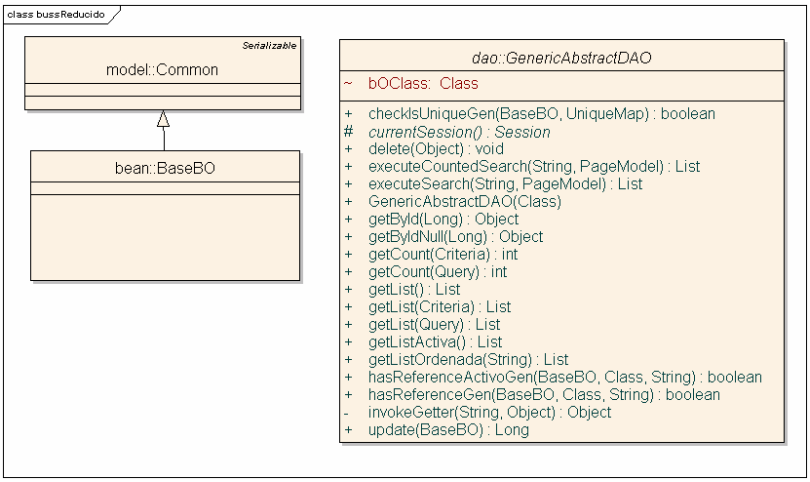


La responsabilidad de cada una de estas clases se define a continuación:

- La clase **UserContext** contiene la información de sesión que necesita la capa *buss* (*e.g.*: la propiedad "userName" contiene el nombre del usuario logueado, que es utilizado por los DAO al momento de realizar una actualización para grabar el usuario que realizo el cambio.

- La clase **UserSession** extiende de la anterior, añade a esta las propiedades y funcionalidad que solo es requerida por la capa *view*.

A continuación vemos las clases fundamentales de la capa *buss*:



La responsabilidad fundamental de cada una de estas clases se describe a continuación:

- La clase **BaseBO** es la encargada de definir propiedades y métodos comunes a todos los objetos de negocio (*e.g.*: la propiedad "id" que es utilizada como identificador único para los objetos de negocio). De ella extienden todas las clases que definen objetos de negocio. Esta clase contiene el método que se encarga de hacer la conversión de un objeto de negocio a un Value Object, el método "toVO()"

- La clase **GenericAbstractDAO** contiene una definición básica de los métodos (por ejemplo el método update) que serán utilizados por los objetos de acceso a datos (comúnmente llamados **DAO**). Por lo que de esta clase extenderán todas las clases DAO.

Demoda también cuenta con una serie de helpers y clases auxiliares que ayudan a realizar diferentes tareas:

- La clase **NavModel** contiene información utilizada para la navegación entre páginas.
- La clase **ListUtil** contiene métodos utilizados para el tratamiento de listas (*e.g.*: el método "toVO", que recibe como parámetro una lista de objetos de negocio y se encarga de pasarla a una lista de Value Object).
- La clase **DateUtil** contiene métodos utilizados para el tratamiento de fechas (*e.g.*: el método "isValidDate"que recibe como parámetro un String y validada si el mismo es una fecha con formato valido o no).
- La clase **StringUtil** contiene métodos para el tratamiento Strings.
- La clase **ModelUtil** contiene métodos para realizar tratamientos y validaciones sobre los Value Object (*e.g.*: el método "isNullOrEmpty" que recibe como parámetro una instancia de cualquier clase que extienda de BussImageModel y determina si es diferente de nulo y tiene un "id" valido).
- La clase **DemodaUtil** contiene una serie de métodos que son utilizados por el framework (por ejemplo el método "populateVO" que lee el request y copia los valores que correspondan a un Value Object, también valida errores de formato en los datos ingresados).
- La clase **AbstractAuditLog** provee la funcionalidad necesaria para loguear datos de auditoria. Brinda la flexibilidad para que estos datos puedan ser guardados de diferentes formas.
- La clase **UniqueMap** contiene un mapa de propiedades para verificar su unicidad en la Base de Datos. Esta clase posee un Mapa interno donde se van

cargando propiedades para verificar su unicidad invocando al método "checkIsUnique" de la clase GenericAbastractDAO.

Servicios de Demoda para la administración de la Seguridad

Las banderas son la estructura básica que plantea Demoda para la seguridad a nivel de componentes de la vista.

La clase **CommonView** provee las siguientes banderas:

```
private Boolean agregarBussEnabled = true;
private Boolean modificarBussEnabled = true;
private Boolean eliminarBussEnabled = true;
private Boolean modificarEstadoBussEnabled = true;
private Boolean bajaBussEnabled = true;
private Boolean altaBussEnabled = true;
private Boolean verBussEnabled = true;
```

Estas banderas serán utilizadas para determinar por ejemplo si un botón o un link esta habilitado o visible. La página jsp lee esta información desde alguna de las banderas previamente enunciadas o desde alguna bandera que halla sido definida especialmente.

Ademas, del sistema de habilitacion/deshabilitacion de icones, antes de ejecutar cada acción la aquitectura vuelve a verifica las credenciales necesarias para llevar a cabo la tarea.

Servicios de Demoda a la capa View (vista)

Sesión de usuario

La clase **UserSession** es la encargada de contener la sesión de usuario. La misma es instanciada cada vez que se realiza exitosamente un logueo de usuario. Y a partir de ese momento mantendrá su estado hasta que la sesión sea cerrada. Contiene un mapa que es utilizado para almacenar todos los valores que deban guardarse en sesión, de esta forma se centraliza el manejo de la sesión facilitando su administración.Tambien con

Esta extiende de la clase **UserContext** la cual será utilizada para enviar como parámetro a la capa buss cuando sea necesario, debido a que esta solo contiene la parte de información de sesión que interesa a dicha capa.

Copia de datos mediante el populateVO

El método "**populateVO**" de la clase **DemodaUtil** tiene como principal objetivo leer los valores del request y pasarlos a alguna instancia de clases **SearchPage** o **Adapter**. Este es utilizado en las páginas de entrada de datos para pasar los datos ingresados por el usuario a una de las clases antes mencionadas, que luego serán enviados al buss para realizar la operación que corresponda.

A continuación se muestra un ejemplo de su utilización:

DemodaUtil.populateVO(atributoSearchPageVO, request);

Como se observa recibe el objeto (de clase searchPage o Adapter) que se quiere popular y el request con los valores submitidos y realiza la copiar de los valores a las propiedades correspondientes.

Para todos los valores de request se realiza una operación trim, para los tipos Date, chequea existencia de alguna annotation java que contenga el formato de fecha deseado, si existe intenta la instanciacion con la mascara correspondiente, para los tipos Long, Integer, Double y Float si el valor submitido es "" se asignara un valor nulo y en caso de ser una enumeración se seteará el valor de la enumeración seleccionado.

Si es un propiedad terminada en View, se setea la misma y se intenta instanciar la correspondiente sin View según el tipo de datos devuelto por el método get

Si hay algún error de formato al intentar setear alguna de las propiedades, se carga un error recuperable al VO pasado como primer parámetro.

Navegación entre páginas

El modelo de navegación se basa en dos clases:

- La clase **NavModel** que maneja los parámetros de navegación que deben ser pasados a través de las diferentes páginas por las que se mueva el usuario. Una instancia de esta clase es creada como composicion dentro de la instancia la clase UserSession que se crea cuando se logue el usuario
- La clase **CommonNavegableView** que tiene la siguiente información:
- El lugar adonde tiene que **volver** la pagina.
- El lugar adonde debe **ir** la página cuando se encuentre en modo selección y se **seleccione un elemento** (esto solo para objetos de las clases <Bean>SearchPage).
- El **"id"** seleccionado cuando se ingreso a la página.
- El **"act"** (este parámetro es usado para pasar de un action a otro) del NavNodel cuando se entro a la página.

El concepto básico es que cada instancia de las clases SearchPage y Adapter contenga la información de su navegación (e.g.: donde volver, donde ir cuando se selecciona un elemento) y que una instancia de clase NavModel sea la encargada de transportar los datos de navegación.

Arquitectura SIAT

A continuación se describe la arquitectura de SIAT.

Se comienza explicando los componentes de la capa de Interface (iface).

A continuación, se describe la capa de presentación (View). Inicialmente, se abordan las características de los Action, luego los JSP y luego, la interacción que existe entre ellos.

En la siguiente sección describimos la capa de negocio (Buss): Implementación de Servicios, DAO, Bean y Manager, explicando sus responsabilidad y características.

Finalmente abordamos el esquema de Login y Seguridad en SIAT utilizando SWE.

Capa Interface (iface)

Esta compuesta por las siguientes interfaces y clases:

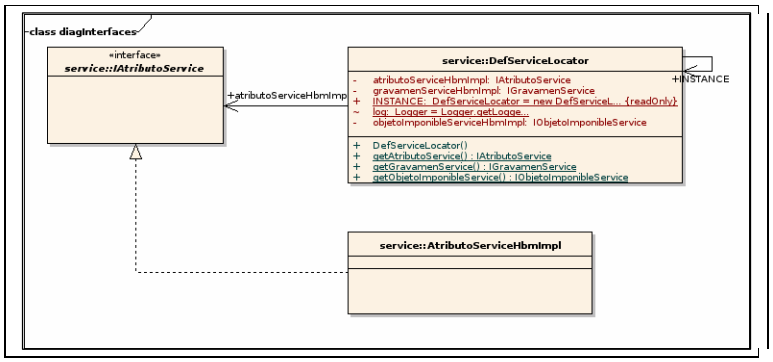
- **Interfaces de Servicio**. representan un contrato de los servicios que SIAT brinda.
- **La clase ServiceLocator** se utiliza para ubicar el servicio que implementa cada interfaz, son singleton. Se construye una por cada módulo de SIAT.
- **La clase SiatAdapterModel** extiende de AdapterModel perteneciente a Demoda.
- **La clase SiatPageModel** extiende de PageModel perteneciente a Demoda.
- **La clase SiatBussImageModel** extiende de BussImageModel perteneciente a Demoda.
- **Clases VO** extienden de SiatBussImageModel. Representan a los Beans de Negocio asociados en la Vista.
- **Clases EntidadXSearchPage** extienden de SiatPageModel. Son utilizadas por la Vista para resolver las búsquedas.
- **Clases EntidadXAdapter** extienden de SiatAdapterModel. Son utilizadas por la Vista para para realizar acciones sobre un objeto de negocio.
- **Las Enumeraciones** implementan la Interfaz IDemodaEmun. Se utilizan para representar las constantes de SIAT.
- **Clases ClaseModuloXError** extiende de DemodaError perteneciente a Demoda. Contiene las definiciones de las constantes estáticas correspondientes a los errores.

A continuación se describen cada uno de ellos:

Interfaces de Servicios

Representan un contrato de los servicios que SIAT brinda, se construye una interfaz por cada submódulo del SIAT, son implementadas por las clases de Servicios del buss. Se encuentran en siguiente ubicación: ar.gov.rosario.siat.<modulo>.iface.service.

A continuación un ejemplo de la interfaz IAtributoService.



ServiceLocator

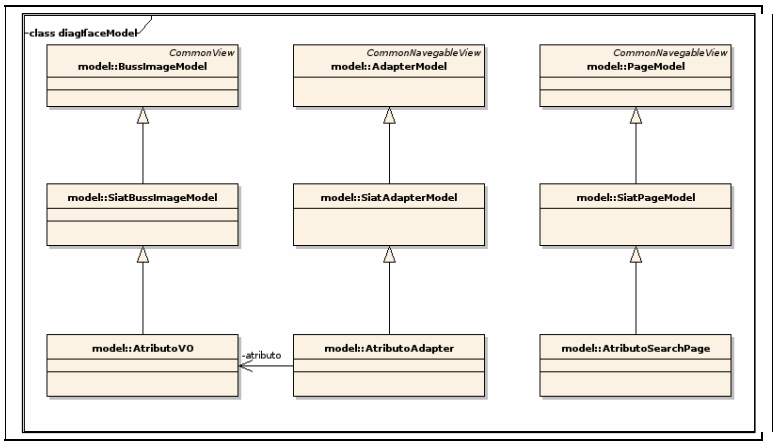
Se utilizan para ubicar el servicio que implementa cada interfaz, son clases singleton, se construye una clase por cada módulo de SIAT.

Contiene los siguientes elementos:

- Propiedades con cada interfaz de servicio.
- Constructor que carga el valor de cada propiedad con la clase del servicio implementado.
- Métodos getters que obtienen el valor de las propiedades, que son las implementaciones de los servicios definidos en el buss.

Model

Estas son clases cuya función principal es la de transportar datos entre la vista y el negocio. En el paquete base se encuentran agrupadas aquellas de las cuales extenderán las de los otros paquetes, las mismas poseen el prefijo Siat.



A continuación se dará una descripción de cada una de ellas.

SiatBussImageModel

Se encuentra en el paquete *siat.base,iface.model*. Extiende de BussImageModel perteneciente a demoda y de esta clase extenderán todos los Value Object de SIAT.

Contiene los siguientes métodos:

- Método *hasEnabledFlag()*: es un helper que verifica si una bandera esta habilitada o no chequeando los permisos según la acción y método que dependen de esa bandera.
  Debe recibir los siguientes parámetros:
    - El valor Booleano que indica si el negocio permite ejecutar la acción-método de SWE.
    - El nombre de la acción de SWE.
    - El nombre del método de SWE.
- Métodos utilizados para determinar las habilitación o no de las banderas de pertenecientes a las acciones *modificar, eliminar, alta, baja* y *ver*. Estos métodos invocan al método estático *hasEnabledFlag()*.

SiatAdapterModel

Extiende de AdapterModel perteneciente a *Demoda*. Es utilizada desde la vista para determinar si el usuario tiene permiso sobre acciones y métodos. Se encuentra ubicada en el siguiente paquete: *siat.base.iface.model*

- Los métodos utilizados para determinar el permiso de acceso del usuario a acciones invocan al método *hasEnabledFlag()* de *SiatBussImageModel*.

SiatPageModel

Extiende de *PageModel* perteneciente a *Demoda,* es utilizada para determinar si el usuario tiene permiso sobre acciones y métodos desde la vista. Se encuentra ubicada en el siguiente paquete: *siat.base.iface.model*.

Contienen métodos para determinar el permiso de acceso del usuario a acciones. Invocan al método *hasEnabledFlag()* de *SiatBussImageModel*.

Clases Value Object (VO)

Representan a los Beans de Negocios asociados en la Vista, extiende de *SiatBussImageModel* y el nombre se forma según: XVO, donde X es el nombre de la entidad.

Contienen los siguientes elementos :

- Propiedad estática **NAME** que identifica a las instancias del VO en la Vista.
- Propiedades **simples**.
- Propiedades que son **ensambles a otros VO**, generalmente instanciados (no se instancian cuando se forman referencias cíclicas).
- Propiedades que son **enumeraciones** pertenecientes a Iface y se corresponden con propiedades de la clase BO asociada.
- Propiedades que son **banderas** utilizadas por la Vista.
- **Constructor** del VO, aquí se setean los valores de las propiedades utilizadas por seguridad para determinar el permiso de acceso a las acciones sobre el objeto de negocio asociado.
- **Getters y setter** de todas las propiedades.
- **Getters** adecuados para la **Vista.** Sus nombres finalizan en "View", son del tipo String y obtienen los valores de propiedades de tipo de datos de fechas y numéricos del VO. En los setters de la propiedades originales se setea el valor formateado de la correspondiente propiedad "View". Para el caso de fechas y horas se utilizan máscaras definidas como anotaciones java. Para utilizar una máscara al momento de validar el formato del valor de una propiedad, se agrega en modo de annotation arriba del setter de la propiedad.

Clases SearchPage

Son clases ubicadas en el paquete *model* del *iface* utilizados por la Vista para resolver las búsquedas, tienen una estructura compuesta por un conjunto de filtros, y una lista de resultados. Extienden de la clase *SiatPageModel* y el nombre se forma según: XSearchPage, donde X es el nombre de la entidad.

Contienen los siguientes elementos:

- Propiedad estática **NAME** que identifica a las instancias del SearchPage en la Vista.
- Propiedades que representan a los **filtros de búsqueda** seleccionados.
- Propiedades que son **listas** (de otros VO y enumeraciones) que cargan los combos de selección de los filtros.
- La propiedad **listResult** de la super clase PageModel para contener el resultado de la búsqueda.

Clases Adapter

Son clases ubicadas en el paquete *model* del *iface* utilizadas por la Vista para crear, ver, modificar, eliminar y realizar otras acciones sobre un objeto de negocio. Extienden de *SiatAdapterModel* y el nombre se forma según: XAdapter, donde X es el nombre de la entidad.

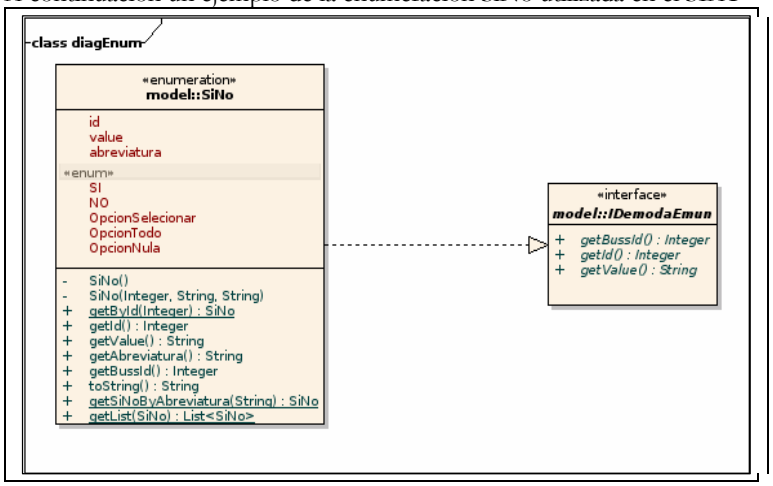
Contiene los siguientes elementos:

- Propiedad estática **NAME** que identifica a las instancias del Adapter en la Vista.
- Una propiedad que es un **ensamble al VO** sobre el cual se realizan las acciones.
- Propiedades que son **listas** (de otros VO y enumeraciones) que cargan los combos de selección.
- **Constructor** del VO, aquí se setean los valores de las propiedades utilizadas por seguridad para determinar el permiso de acceso a las acciones sobre el objeto de negocio asociado.

Enumeraciones

Se utilizan para representar listas de constantes de SIAT, son enumeraciones de java, implementan la Interfaz *IDemodaEnum* de *Demoda*. Se encuentran ubicadas en el siguiente paquete: *coop.tecso.demoda.iface.model*.

A continuación un ejemplo de la enumeración SiNo utilizada en el SIAT



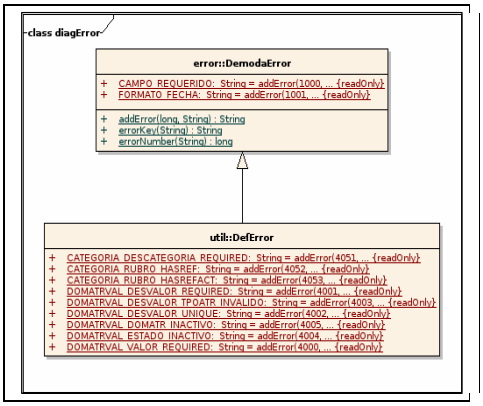
Contienen los siguientes elementos:

- Declaraciones de cada elemento de la enumeración.
- Propiedades de la enumeración (por ej.: id, value, abreviatura).
- Constructores privados.
- Getters de las propiedades.
- Método estático *getById()*, utilizado para obtener una instancia a partir de la clave.
- Métodos estáticos que permiten obtener listas de instancias de la enumeración.

Constantes de Error y Mensajes

Contienen las definiciones de las constantes estáticas correspondientes a los errores de cada módulo. Extienden de *DemodaError* perteneciente a *Demoda*. Se construye una clase por cada módulo del SIAT.

Las constantes se agrupan, dentro de la clase, por submódulo y por objetos asociados. El método *addError()* se utiliza para obtener un String con sus parámetros concatenados. Se en encuentran ubicadas en los siguientes paquetes: *siat.moduloX.iface.util.*



Capa de presentación (view)

En esta sección se describen de los componentes de la vista.

Ya que la Vista no posee lógica de negocio, pudo llevarse a una estructura bien definida que da solución a cuestiones de navegación entre JSP, interacción con SWE para manejar la seguridad, manejos de mensajes, etc. Esta estructura bien definida se crea ajustándose a convenciones de nombres de clases, métodos, y comportamientos.

Es por esto que en esta sección cada vez que se vea algo encerrado entre signos <> uno debe reemplazar esto por la entidad que corresponda. Por ejemplo: Buscar<Bean>DAction, es el nombre genérico que representa todos los Action Buscar de la Vista *e.g.*: BuscarAtributoDAction, BuscarRecursoDAction, etc.

Introducción

La capa de Vista es la encargada de la interacción directa entre el usuario y el sistema. Esta compuesta por las siguientes clases y archivos:

- La clase **BaseDispatchAction** que extiende de la clase **DispatchAction** del framework **Struts** y provee una serie de métodos que son utilizados por las clases que la extienden para realizar una serie de tareas comunes (por ej.: el método baseVolver provee la funcionalidad básica que permite volver de una página a la que la llamo).

- Clases **Buscar<Bean>DAction y Administrar<Bean>DAction** (comúnmente llamadas action) que heredan de la clase BaseDispatchAction y son las que controlan el flujo de la aplicación (controlers del patrón MVC).

- Paginas **jsp** que contienen el código para renderizar el HTML y archivos **js** que contienen código java script utilizado en los jsp. Las páginas jsp se dividen en dos grandes grupos las **<Bean>SearchPage** y **<Bean>Adapter.**

- Archivos tipo **struts-config-<modulo>.xml** que son utilizados por el framework Struts para manejar las llamadas a los action.

- Archivos tipo **tiles-defs-<modulo>.xml** que son utilizados para definir los layouts de los diferentes jsp.

- Clases **<Modulo>Constants** que son utilizados para definir constantes que serán utilizadas principalmente desde los action.

- Archivos tipo **<modulo>.properties** que son utilizados como archivos de recursos para almacenar todas las frases y etiquetas que aparecen en los jsp.

A su vez utiliza las siguientes clases de *Demoda* y de la capa *iface:*

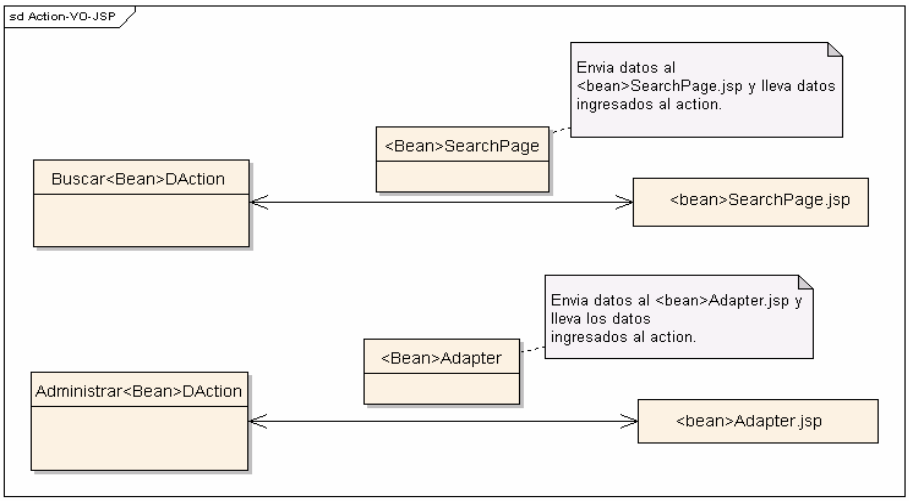
- La clase **NavModel** de Demoda que se utiliza para transportar la información de utilizada para la navegación entre paginas.
- Las clases **<Bean>SearchPage**, **<Bean>Adapter** y **<Bean>VO** que pertenecen a la capa *iface* y cuyas instancias contienen los datos que serán visualizados luego del armado del JSP.
- La clase **UserSession** de *Demoda* que es utiliza para almacenar información en la sesión.

Luego por medio de la interacción entre estos componentes se obtiene una salida visible para el usuario (página HTML).

Los actions son los encargados de controlar el flujo de la aplicación, y son accedidos siempre a través del nombre definido en el archivo de configuración struts-config-<modulo>.xml. Son disparados cuando el usuario realiza alguna acción (por ej.: oprimir un botón), una vez que fue disparado un action realizara una serie de tareas (por ej.: validar acceso, validar los datos ingresados en caso de que los hubiere, llamar a un servicio) para finalmente dirigirse a un determinado JSP o a otro action.

Básicamente tendremos dos tipos de actions, estos son: *Buscar<Bean>DAction* y *Administrar<Bean>DAction* que se corresponden los JSP *<bean>SearchPage.jsp* y *<bean>Adapter.jsp***.**

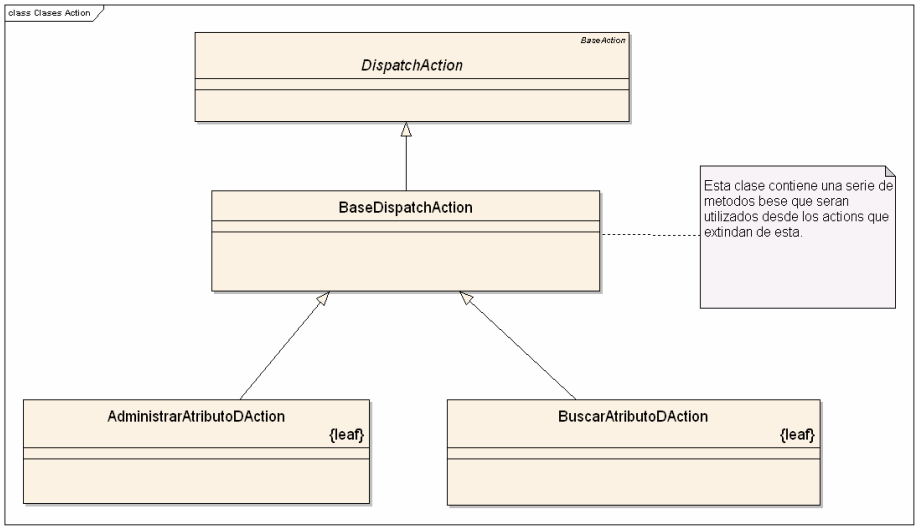
El siguiente diagrama muestra la relacion Action-ValueOject-JSP que existe en la vista para cada entidad.



A continuación se describen las características principales de las clases action y JSP

Clases Action

SIAT tiene dos tipos básicos de action, estos son el *Buscar<Bean>DAction* y el *Administrar<Bean>DAction* que extienden de la clase *BaseDispatchAction***,** como se observa en el siguiente diagrama:



Como se muestra en la figura anterior los action *AdministrarAtributoDAction* y *BuscarAtributoDAction* extienden del *BaseDistpachAction* que es el que provee los métodos básicos que utilizan estos.

Las actividades fundamentales que realizan los métodos de un Action son:

- Chequear si el usuario tiene acceso permitido al Action.
- Recuperar o subir Value Objects del/al objeto *UserSession* para utilizarlos en el jsp o en otros métodos que lo precisen.
- Leer el request y transformarlo en un Value Object (esta tarea la realiza el método *populateVO()* de la clase *DemodaUtil*).

- Verificar errores de sintaxis (esta tarea también es realizada por el método *populateVO()*).
- Llamar al servicio correspondiente.
- Manejar los errores que pueda haber devuelto el servicio.
- Setear los valores de navegación que correspondan.
- Finalmente forwardear a un JSP o a otro action.

A continuación describiremos los métodos más importantes que posee cada tipo de clase action y su funcionamiento estándar:

BaseDistpachAction

- **canAccess:** este método es utilizado para chequear que el usuario tiene acceso al action que se esta ejecutando, también se encarga de setear diferentes valores del request en el objeto UserSession.
- **baseVolver:** este método es utilizado para manejar los volver.
- **baseSeleccionar**: este método es utilizado cuando el searchPage se encuentra en modo selección y se selecciona un elemento.
- **baseForward**: este método se utilaza por otras funciones del mismo BaseDistpachAction para setear los valores de navegación, se explicara mas en detalle en la sección de navegación.
- **saveDemodaErrors** y **saveDemodaMessages**: son utilizados para setear por medio se una función de Struts los errores y mensajes en el request para que puedan ser mostrados al usuario.

Clases Buscar<Bean>DAction

- **inicializar**: este método es utilizado para recuperar y mostrar los campos de filtros en el searchPage.
- **buscar**: este método es utilizado para recuperar y mostrar la lista de resultados, filtrado por los criterios que halla ingresado el usuario en los filtros en caso de haber ingreso alguno.
- **limpiar**: este método es utilizado para volver el searchPage a su estado inicial o sea tal cual como se inicializo la misma.
- **ver**, **modificar**, **eliminar**, **agregar**: estos métodos llaman al action **Administrar<Bean>DAction**, pasándole un parámetro (en la propiedad "act" del una instancia de la clase NavModel) que le indica que tipo de acción selecciono el usuario (ver, modificar, eliminar, agregar, etc.) el usuario para llamar al servicio que corresponda y forwardear luego al adapter adecuado.
- **seleccionar**: este método es llamado cuando la clase se encuentra en modo selección y se encarga de devolver el "id" del objeto seleccionado (este es devuelto en la propiedad "selectedId" del objeto NavModel) a quien lo allá llamado.

Administrar<Bean>DAction

- **inicializar**: este método es el encargado de recuperar los datos necesarios y luego mostrar el adapter correspondiente. Generalmente es llamado desde un Buscar<Bean>DAction, quien le indica (en la propiedad act del objeto NavModel) que servicio es el que debe llamar y que adapter se bebe renderizar.
- **agregar**: llama al servicio que inserte el objeto correspondiente y si no hubo errores vuelve a la pantalla de búsqueda que lo llamo.

- **modificar**: llama al servicio que actualice el objeto correspondiente y si no hubo errores vuelve a la pantalla de búsqueda que lo llamo.
- **eliminar**: llama al servicio que elimine el objeto correspondiente y si no hubo errores vuelve a la pantalla de búsqueda que lo llamo.
- **refill**: este método es el encargado de re dibujar la página cuando sea necesario (por ej.: luego de agregar o modificar un detalle, en el caso que el objeto <Bean>Adapter correspondiente tenga detalles).
- **volver**: estos métodos simplemente vuelven a la página que corresponda, generalmente es la misma que llamo a la página en la que me encuentro.
- **param<X>**: estos método pueden ser utilizados una o varias veces en cualquier action y su función es llenar datos en el SeachPage o Adapter correspondiente luego de realizar una operación que deba re cargar datos en la pagina (por ej.: el llenado de los datos de un combo, que esta relacionado con otro cuando este otro cambia su valor, o el llenado de los datos retornados de un búsqueda).

Páginas JSP

Dentro del paquete *view* se encuentran las paginas JSP encargadas de armar las paginas HTML del SIAT. Las paginas se pueden clasificar según un criterio genérico respecto a diseño visual y funcionalidad que presentan al usuario.

Podemos distinguir los siguiente tipos de páginas que hacen al SIAT:

- Páginas de búsqueda y/o selección:
  Son paginas desde donde el usuario busca sobre un gran conjunto de registros alguno/s en particular para operar con él. Estas páginas presentan una sección de filtro, una sección de resultados, una sección para paginar el resultado, y una sección de acciones a realizar sobre el resultado de la búsqueda. Las acciones pueden ser *e.g.*: Agregar, Modificar, Ver, Seleccionar, etc.

- Páginas Encabezado Detalle:
  Estas páginas se utilizan para mostrar entidades que están compuestas por otras entidades (*e.g.:* Encabezados Detalles). Generalmente poseen una o más secciones con datos campos/valor que hacen a estructura en si y poseen una o más secciones con listas de entidades agrupadas en tablas. Un ejemplo de estas páginas, podría ser la de Gestión de Deuda actual.

- Formularios Campo/Valor de Solo Lectura:
  Son páginas diseñadas para mostrar información, sobre la luego el usuario realizara una operación que afecta al conjunto de los datos, *e.g.*: Mostrar el Recurso antes de Eliminarlo. Presentan pares de Campo/Valor agrupados en una o más secciones. Los valores son presentados para solo lectura, por lo que no se puede modificar explicitamente ningún dato en particular de la entidad.

- Formularios Campo/Valor de Edición:
  Son páginas desde donde se pueden alterar valores que se muestra.
  Presentan pares de Campo/Valor agrupados en una o más secciones. Los valores son presentados para ser alterados, por lo que desde aquí se puede alterar algún dato en particular de la entidad.

Los tipos de páginas de arriba representan una clasificación desde el punto de vista funcional y corresponden tipos de páginas y conceptos de la arquitectura de SIAT.

Desde el punto de vista de SIAT podemos clasificar las páginas en:

- Páginas SearchPage:
  Son las páginas de búsqueda y selección.
  Poseen una sección de Filtros desde donde se pueden ingresar los datos para acotar la búsqueda. Posee el botón Buscar, y Limpiar para efectuar la búsqueda y limpiar los resultados y filtros respectivamente.
  Poseen una sección de Resultados donde se muestra los registros que coinciden con los filtros en forma de tabla. Cada fila de la tabla corresponde a un registro, y muestra los datos y acciones a realizar sobre el registro. Por ejemplo: Ver, Modificar, Eliminar, etc.
  Si el SearchPage esta en *modo seleccionar* la sección de Resultados solo muestra el botón 'Seleccionar'
  Esto permite a los SearchPage una doble utilización, ya sea para disparar acciones típicas a la administración de los registros. Y también para la selección desde otras entidades cuando se necesite asociarlas.

- Páginas Adapter con Encabezado Detalles (o de Entidades Compuestas):
  Son las páginas que deben mostrar datos en encabezado en una o más secciones y luego muestra una o varias listas registros correspondiente a las composiciones de la entidad.

  Hay páginas para **Ver** en las solo permiten acciones para mostrar las entidades y llevan a formularios de tipo solo lectura.

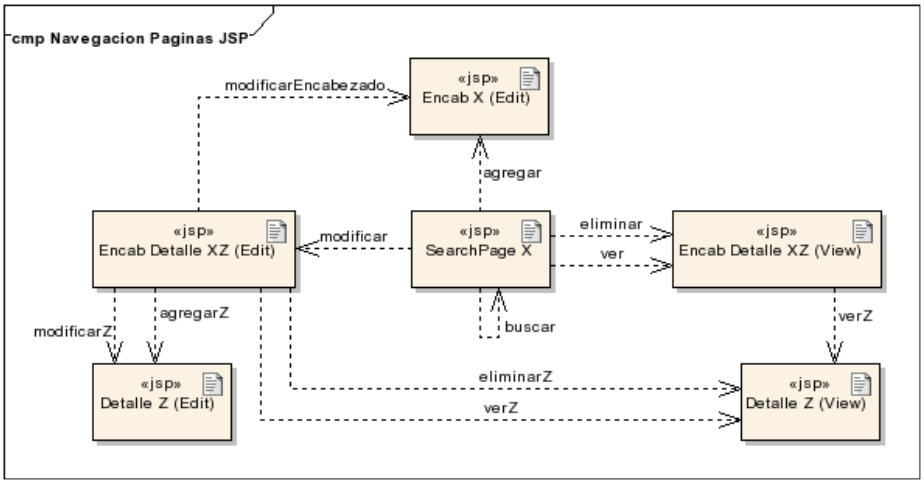
  Hay páginas para **Modificar**, presentan botones extras que permiten administrar tanto el encabezado como cada registro de la composición. Generalmente tiene un botón 'Modificar' para cambiar los datos del encabezado que nos lleva a un Formularios de tipo Edición para alternar los campos del encabezado. Luego posee múltiples iconos varios por cada registro de las composiciones para alterar los datos de cada registro.

- Páginas Adapter Formularios tipo Ver:
  Corresponden a los formularios con campo/valor de solo lectura.
  Generalmente se utilizan como vista previa antes de Eliminar, Activar o Desactivar alguna entidad.
  Presentan todos los datos de la entidad en solo lectura, y a veces permiten ingresar una fecha de inactivación o similar.

- Páginas Adapter Formularios tipo Edición:
  Corresponden a los formularios con campo/valor para modificar.
  Desde aquí se pueden alterar los datos de específicos de una entidad.

Nota: las paginas llevan los nombres SearchPage y Adapter, haciendo referencia a los nombre de los ValuesObject que representan según heredan de PageModel o AdaptaerModel de demoda.

El siguiente diagrama muestra la interacción entre los JSP según las acciones seleccionadas por el usuario. El diagrama muestra la navegación de una entidad compuesta Encabezado Detalle. La entidad la identificamos con la letra X, y esta compuesta con la entidad Z, en este caso por ejemplo, la entidad X posee una lista de Z.



Interacción entre Páginas y Action

Cada vez que el usuario presiona sobre un botón o un icono de una página, se termina por ejecutar el método de un Action de SIAT. Por convención, los request de cada página son enviados al action que produjo su previo response. En la arquitectura se distinguen estos Action: *BuscarXDAction, AdministrarXDAction, AdministrarEncXDAction*. La letra *X* representa el nombre de la entidad y todos finalizan con el sufijo *DAction*, para indicar que son DispatchActions de struts.

Los Action *Buscar* están relacionados con las páginas de búsqueda SearchPage. Controlan el ingreso de filtro y búsquedas sobre las entidades.

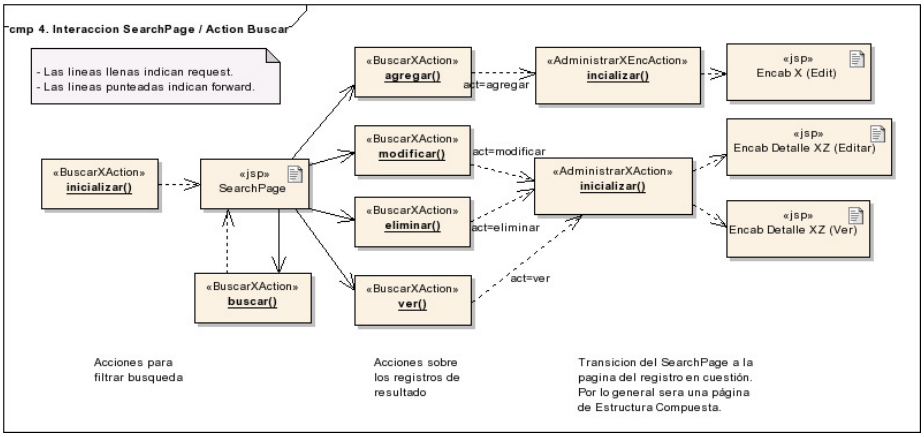
Los *Administrar* y *AdministrarEnc* estan relacionados con las paginas de Formularios y Entidades Compuestas o Maestros Detalles. Se encargan de controlar las acciones relacionadas con la administración de la entidad, por ejemplo, Modificar, Agregar, Ver, Eliminar, etc. En particular los *AdministrarEnc* exponen solo los métodos para administrar el Encabezado para el cazo de las estructura compuestas.

Todos los Actions poseen un método *inicializar()* al que se invoca cuando se quiere, ya sea acceder a una búsqueda o acceder a los formularios para ver o modificar.

En el caso de los Actions *Buscar* inicia los objetos Value Object de la búsqueda y realiza un forward() al JSP SearchPage de búsqueda.

En el caso de los Actions *AdministraEnc* y *Administrar* el método inicializar recibe un dato extra que indica si la acción de Administrar se refiere bien a *modificar, agregar, eliminar*, etc.

El siguiente es una diagrama que muestra la interacción entre un JSP SearchPage y su Action *Buscar*



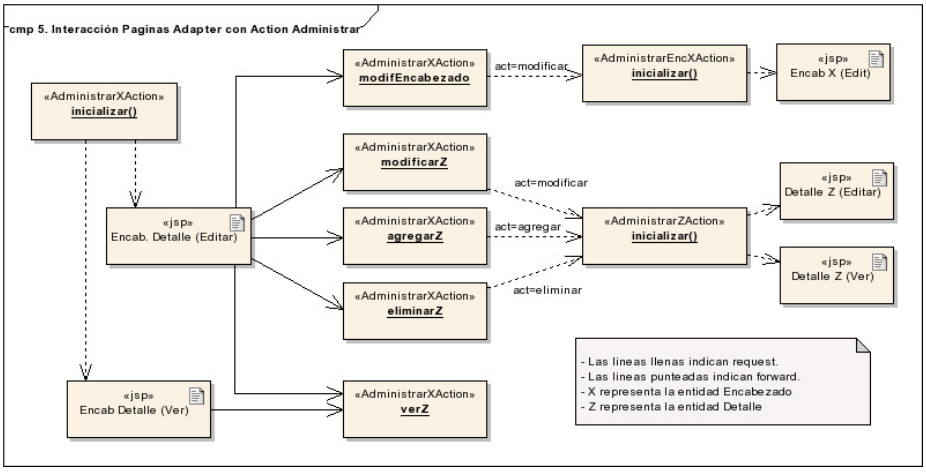
El diagrama muestra como llegamos a la página JSP a través del método *inicializar()* de un Action *Buscar*, luego mientras el usuario realiza la acción de *'buscar'* nos mantenemos en el mismo JSP trayendo el resultado de la búsqueda.

Si el usuario realiza la acción para agregar/crear una nueva entidad X, en ese caso se realiza la acción *agregar* contra el Action Administrar de X. Este procesa la accion y realiza un forward al método *inicializar()* del Action *AdministrarEnc* de X enviando el mensaje *agregar*.

Si el usuario realiza alguna acción sobre los registros de búsqueda (*eliminar, modificar, ver, etc*), las acciones se submiten a distintos métodos del Action *Buscar*. Cada uno de estos métodos redirigen al método *inicializar()* del Action *Administrar* de la entidad pasando información de la acción realizada en el SearchPage. Si las acciones fueron *eliminar* o *ver,* se muestra el JSP de Encabezado Detalle de tipo Ver. Si la acción fue *modificar*, se muestra el JSP de tipo Edición.

A continuación mostramos las acciones y sus interacciones con Action de un Maestro Detalle. Suponemos que la entidad Maestro se llama X y con Z representamos la entidad compuesta dentro de X.

Desde un JSP de Maestro Detalle de tipo Edición, podemos ejecutar bien, las acción para *modificarEncabezado* o las acciones de *modificarZ, eliminarZ, verZ, agregarZ* para cada registro del detalle.



Para los casos de las acciones *modificarZ, eliminarZ, verZ, agregarZ* como se tratan todas ellas de acciones sobre la entidad Z, son se realiza un forward al método *inicializar()* del Action *Administrar* de la entidad Z.

En caso de que el usuario modifique el encabezado de la entidad X, en ese caso se realiza la acción *modifcarEncabezado*, tambien se realiza un forward, pero esta vez al método *inicializar()* del Action *AdministrarEnc* de X enviado el mensaje *modificar*.

Interacción entre Action y Servicios

A grandes rasgos los métodos de los Action realizan los siguientes pasos:

- Obtiene la sesión de usuario y verifica permisos contra SWE
- Obtiene los datos del request y los copia dentro de un ValueObject del modelo de *iface*
- Llam al método adecuado del Servicio
- Verifica errores
- Redirige al JSP

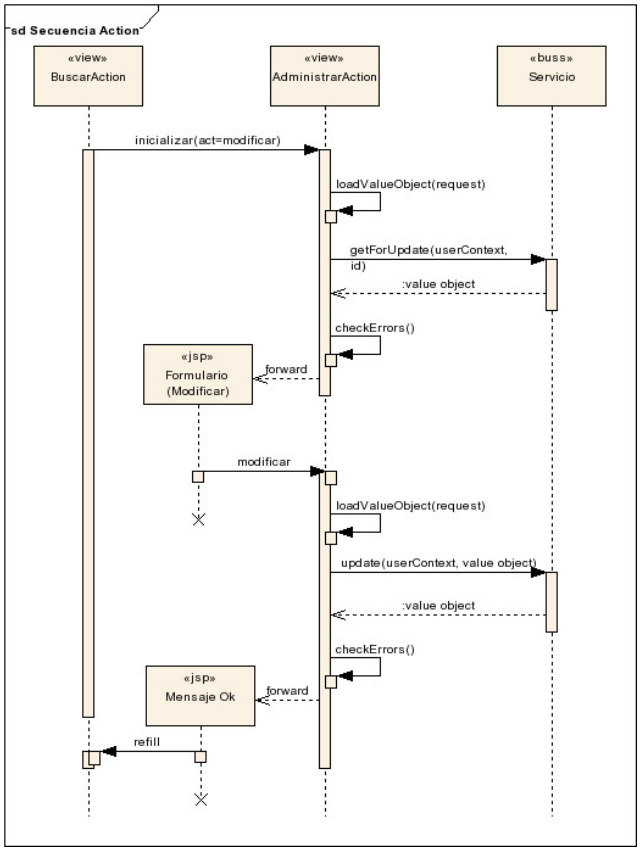
El primer paso lo analizaremos más adelante en la sección Login y Seguridad. Para el segundo paso, los Action se valen de un método helper de demoda que permite transferir los parámetros de un request en una árbol de ValueObjects de demoda.

Para llamar al método del Servicio, el Action utiliza la clase SeviceLocator, para obtener una instancia del mismo. Luego llama al método pasando datos de la sesión de usuario y el ValueObject con los datos cargados.

El servicio realiza la lógica,y el servicio retorna un ValueObject, con los datos procesados. En caso caso de existir errores, se retornan dentro del ValueObject que retorna el método del servicio.

El Action ahora revisa el tipo de error, si se trata de un *error recuperable, e.g.*: errores de validación, el Action los trata adecuadamente para mostrarlos como mensajes de struts y redirige al JSP que muestra los datos retornados. Si se trata de un *error no recuperable, e.g:* excepción no manejada, el Action redirige a un JSP de mensaje de error.

A continuación se muestra el diagrama de secuencia que ilustra la interacción entre los Actions y un servicio (ver próxima página). En el diagrama se muestran nombres genericos de las llamadas con el fin de ilustrar.



La secuencia comienza cuando el Action *Buscar* realiza el forward al método *inicializar()* del *Administrar* pasando el mensaje *modificar*. Durante el *inicializar()* se llama al método *getForUpdate()* del servicio para obtener los datos a mostrar luego se verifican errores, y se termina mostrando el JSP para modificar la entidad.

Luego el usuario altera los datos y submite los cambios con la acción *modificar* al Action *Administrar*. El Action llama al método *update()* del servicio, y enviá los datos del usuario y los datos modificados. Luego se verifica si ocurrieron errores, y se muestra el mensaje de éxito. De allí el mensaje submite con la acción de *refill* (recargar) al Action Buscar desde donde comenzó el ciclo.

Manejo de banderas y seguridad

Las páginas JSP contienen también **banderas** que son seteadas en los objetos de las clases Adapter, SearchPage y Value Objects y leídas de una instancia de la clase UserSession por la jsp, estas banderas definen por ejemplo si un botón o un link esta habilitado o visible. Esta visibilidad o habilitación de un determinado componente de la jsp es determinada de acuerdo a dos factores:

- Las acciones que el usuario tiene permitidas realizar de acuerdo a los roles que tiene asignados.
- Las acciones que el usuario puede realizar determinada por la lógica de negocio.

De este modo, la capa de negocios tiene la responsabilidad de setear las banderas que corresponda mediante la consulta de permisos definidos en SWE y luego, mediante lógica de negocio particular.

Por ejemplo: El perfil: "Procurador" tendrá acceso a las funciones de consulta de deuda, pero cada procurador solo deberá poder consultar la deuda que el posee y no la del resto de los procuradores. En este caso, a nivel SWE se habilita la opción al perfil Procurador, y en la lógica de negocio específica, se deberán filtrar los registros de deuda que corresponda.

Luego, a nivel JSP mediante los tags de struts <*logic:equal*> y <*logic:notEqual*>se consulta al VO a mostrar el valor de una bandera para determinar si un determinado elemento se muestra habilitado o visible dependiendo del caso.

Por ejemplo se puede determinar como se muestra habilitado o no una opción dependiendo del valor de la bandera *getModificarEnabled()* definida en el Value Object *DomAtrValVO*.

Además los action a su vez, al comenzar la ejecución de un método realizan un chequeo de acceso mediante el método *canAccess()*, que verifica si el usuario tiene acceso al método que se comenzó a ejecutar. Este es el chequeo de seguridad que impide que un cliente pueda acceder directamente al action si el usuario logueado no tiene permiso para ejecutar un determinado método del action.

Capa de Negocio (Buss)

Dentro de la capa de negocios podemos enumerar: Implementación de Servicio, Bean, DAO y Managers. Los Servicios exponen la funcionalidades de cada módulo a componentes externos (como la capa de presentación de SIAT o los proceso de ADP). DAO es la capa de acceso a datos, mayormente implementa métodos que utiliza Hibernate y HQL. Con Bean nos referimos al conjunto de Objetos de Negocio que implementan las funcionalidades de SIAT. Los Manager son clases singleton que implementan las funcionalidades que no pueden ser claramente asignadas a los Bean.

A continuación se describen los componentes con un mayor nivel de detalle:

- **Clases de Servicio** que implementan las interfaces de servicio de *iface*, representan los servicios brindados por el sistema a la vista.
- **Clases Manager** una por cada módulo, son singleton que contienen los métodos que corresponden al módulo no asignados a las clases Bean del módulo.
- **Clases de Negocio** que extienden de la clase *BaseBO* de *Demoda*. Son beans con comportamiento que resuelven los servicios de acuerdo a las reglas de negocio.
- La clase **GenericDAO** que extiende de *GenericAbstractDAO* (de *Demoda*). Obtiene la sesión actual de Hibernate.
- **Clases DAO** que extienden de *GenericDAO* utilizadas para resolver el acceso de los Objetos de Negocio a los datos de la base de datos relacional utilizando Hibernate.
- **La clase DAOFactory** son singletons con la responsabilidad de devolver instancias de las clases DAO.
- **La clase SiatHibernateUtil** es una clase estática que permite obtener, abrir y cerrar la sesión actual de Hibernate.
- El archivo de **configuración de Hibernate**.

Clases de Servicio

Los servicios son el punto de entrada al paquete de negocio del SIAT. Implementan las interfaces definidas en el paquete *iface*. La instanciación adecuada de los Servicios es provista por una clase ServiceLocator de cada módulo.

Todos los métodos de los servicios tienen en su primer parámetro la clase UserContext, los demás parámetros son ValueObjects necesarios para realizar su funcionalidad. En la clase UserContext se encuentran los datos del usuario autenticado y el nombre de Acción y Método de SWE solicitados.

Es responsabilidad de los métodos de servicio:

- Hacer disponible UserContext al hilo de ejecución.
- Iniciar las sesión de Hibernate del SIAT.
- Extraer los datos del ValueObject de entrada.
- Delegar la funcionalidad en los Bean (Objetos de Negocio) y/o Manager.
- Verificar la existencia de errores durante la llamada a los Bean.
- Solicitar a los Bean que se copien a ValueObjects.
- Si es necesario retornar un Adapter, o un SearchPage deberá componerlo con los ValueObject que corresponda.
- Realizar commit/rollback de la sesión de Hibernate según corresponda.
- Cerrar la sesión de Hibernate

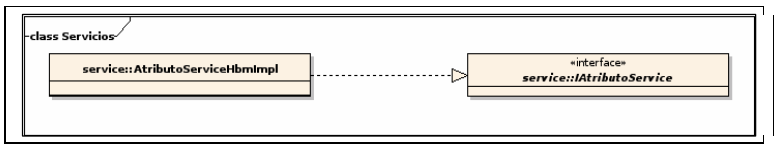
Representan los servicios de negocio brindados por el sistema SIAT, implementan las interfaces de servicio de *iface*. Se construye una por submódulo de SIAT.

Los métodos reciben comúnmente como parámetros:

- *UserContext (*con información del usuario logueado), este sera el primer parámetro en todo los métodos de los servicios.
- *CommonKey (*con el id del objeto sobre el cual realizar acciones).
- Clases de iface.model: *VO, Adapters, SearchPage.*

En general los métodos realizan las siguientes tareas:

- Abren y cierran la sesión de Hibernate.
- Creaciones, actualizaciones y borrados de datos:
    - Abren y cierran transacciones dentro de la sesión de Hibernate.
    - Si no existen errores realizan un commit.
    - Si existen errores realizan un rollback.
- Invocan a métodos de los *Managers* y de otros objetos de negocio, nunca invocan a métodos del mismo u otro servicio.
- Realizan la invocación de métodos necesarios para pasar de Bean a Value Object.
- En caso de atrapar una Exception:
    - Si está abierta una transacción: realiza un rollback sobre la misma.
    - Disparan una *DemodaServiceException.*

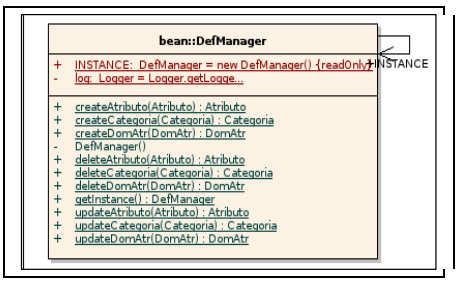


Clases Manager

Los Managers absorben las funcionalidades de sistema que no pueden delegadas a los Bean. Por ejemplo la operación *crearRecurso* dentro del SIAT, o la operación *crearAplicacion* dentro de SWE son operaciones que delegamos en los Manager. De esta manera las operación fundamentales que hacen al Sistema, quedan concentradas en estas clases.

En el SIAT las clase Manager se implementan como Singleton una por submódulo del Modelo de Análisis.

Se crea una clase por cada submódulo del SIAT. Son singleton que contienen los métodos que corresponden al submódulo no asignados a las clase de negocio.



Clases de Negocio

Poseen la lógica de negocio del SIAT y para llevar a cabo las funcionalidades requeridas delegan responsabilidades entre ellos.

Los Bean por lo general poseen algunos atributos mapeados al modelo relacional a través de Hibernate, incluso cuando se considera adecuado se utilizan los mapeos *OneToMany* y *ManyToOnde* de Hibernate para resolver las composiciones y agregaciones. Así y todo, muchas veces resulta imposible utilizar los mapeos, en esos casos se delega el acceso a datos a las clases DAO de cada Bean. Las clases DAO tambien se utilizan para las operaciones comunes de obtener, actualizar, crear, eliminar los Bean de la Base de Datos.

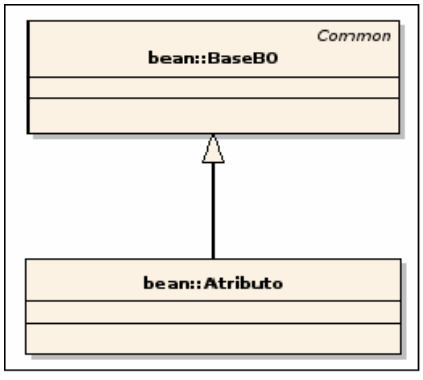
Determinar la responsabilidad de negocio de cada Bean resulta a veces no trivial, como regla general para la asignacion de responsabilidades nos guiamos por los patrones GRASP:

- Experto en Información:
  Esto es asignar la responsabilidad la clase que tiene la información necesaria para realizar la responsabilidad.

- Creador:
  Cuando B debe crear una instancia de una clase A?
  - B agrega objetos de A
  - B contiene objetos de A
  - B registra instancias de objetos A
  - B utiliza más estrechamente objetos A
  - B tiene datos de inicialización que se pasarán a un objeto de A cuando sea creado

- Controlador:
  Existen funcionalidades a nivel sistema que no pueden asociarse a instancias de Bean. Estas funcionalidades deben ser absorbidas por las clases controladoras, para nuestro caso son las clases Manager de cada Submódulo del SIAT.

Extienden de la clase *BaseBO* perteneciente a *Demoda.* Encapsulan la responsabilidad de acuerdo al patrón GRASP de asignación de responsabilidades.



Contienen los siguientes elementos:

- **Propiedades simples**.
- **Propiedades** que son **ensambles** a otros objetos de negocio.
- **Constructores**: uno al menos sin parámetros necesario para Hibernate.
- Métodos **Getters y Setters** de sus propiedades.
- **Anotaciones de Hibernate** para realizar el mapeo O-R de las clases y sus propiedades a las tablas y campos de la base de datos.

- Método estático *getById()* para recuperar un instancia de la clase dado su Id.
- Métodos estáticos para recuperar **listas de instancias** de la clase.
- **Métodos de validación** para las acciones básicas agregar, modificar y eliminar los cuales realizan las siguientes tareas:
    - Realizan validaciones de datos requeridos y rangos.
    - Realizan validaciones de negocio (FK, UK, Reglas de Negocio, etc.).
    - En caso de no cumplir las validaciones, se carga la lista de errores en el objeto y devuelve false.
- Otros métodos **de negocio** que resuelven los casos de uso.

GenericDAO

Extiende de *GenericAbstractDAO* perteneciente a *Demoda*. Todos los DAO del SIAT deben extender esta para brindar los métodos básicos para el acceso a datos.

Su constructor asocia la clase de Negocio que recibe como parámetro al DAO. Además define los siguientes métodos que serán utilizados en SIAT:

- El método *hasReference()* se utiliza para determinar si una determinada entidad tiene referencias a otra entidad. Por ejemplo, se puede utilizar este método antes de eliminar una entidad para verficar que no existan referencias a ella.

- El método *checkIsUnique()* determina si el objeto pasado por parámetro es único (no es duplicado) en la base de datos de acuerdo a la propiedades y valores cargados en la instancia de la clase UniqueMap que recibe como parámetro. Requiere que existan los mapeos de Hibernate adecuados.

- El método *hasReference()* se utiliza para determinar si una determinada entidad en estado activo tiene referencias a otra entidad. Por ejemplo, se puede utilizar este método antes de eliminar una entidad para verficar que no existan referencias a ella.

Clases DAO

A fin de separar los aspectos de diseño dentro del SIAT, se introducen las clases DAO. Cada Bean posee una clase DAO, que implementa la persistencia en la Base de Datos.

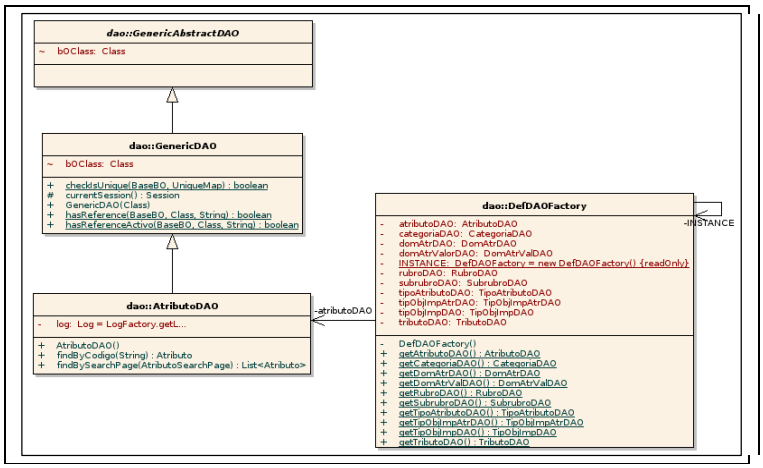
Todos los DAO poseen los métodos comunes, que permiten obtener, actualizar, crear y eliminar un objeto de la Base de Datos. Ademas, cuando las funcionalidades lo requieren, poseen métodos extras que realizan consultas HQL vía Hibernate.

Como regla de arquitectura y diseño se utiliza los DAO para separar aspecto técnico de recuperar los datos utilizando Hibernate del verdaderamente funcional implementado en las clases Bean. De esta manera las clases DAO actúan como helpers de cada Bean.

Extienden de *GenericDAO*. Se utilizan para resolver el acceso de los objetos de Negocio a los datos de la base de datos relacional utilizando Hibernate. En estos se escribirán los métodos específicos utilizados por los objetos de negocio y los manager de SIAT.

DAOFactory

Son singletons con la responsabilidad de devolver instancias de las clases DAO. Existe un *DAOFactory* por cada modulo del modelo de análisis.



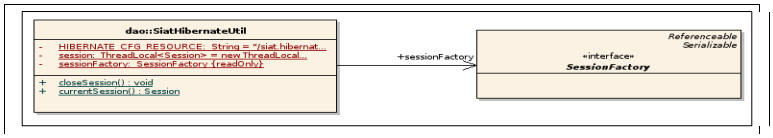
SiatHibernateUtil

Es una clase estática que permite obtener, abrir y cerrar la sesión actual de Hibernate. Contiene las siguientes propiedades y métodos:

- La propiedad **sessionFactory** que es un SessionFactory de Hibernate.
- La propiedad **session** de tipo ThreadLocal usada para contener la sesión de Hibernate.
- La propiedad *HIBERNATE_CFG_RESOURCE* constante String que es el nombre del archivo de configuración de Hibernate: *e.g.:* "siat.hibernate.cfg.xml".
- **Constructor estático** que crea un SessionFactory a partir del configurador de anotaciones de Hibernate.

El método *currentSession()*: obtiene la sesión de Hibernate actual. Si la sesión es nula la crea y la habré y la coloca en TLS del hilo actual, caso contrario devuelve la sesión que se encuentra abierta en en TLS.

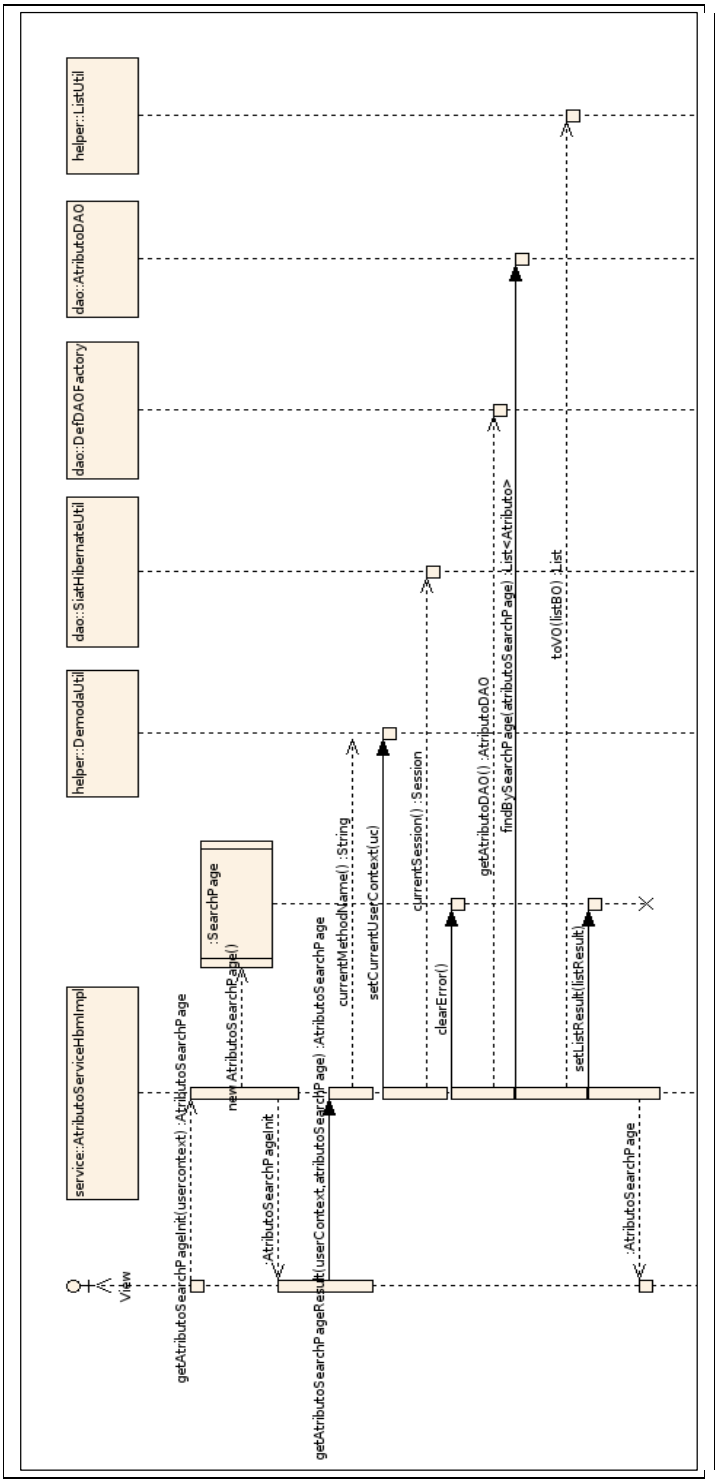
El método *closeSession()* cierra la sesión de Hibernate.



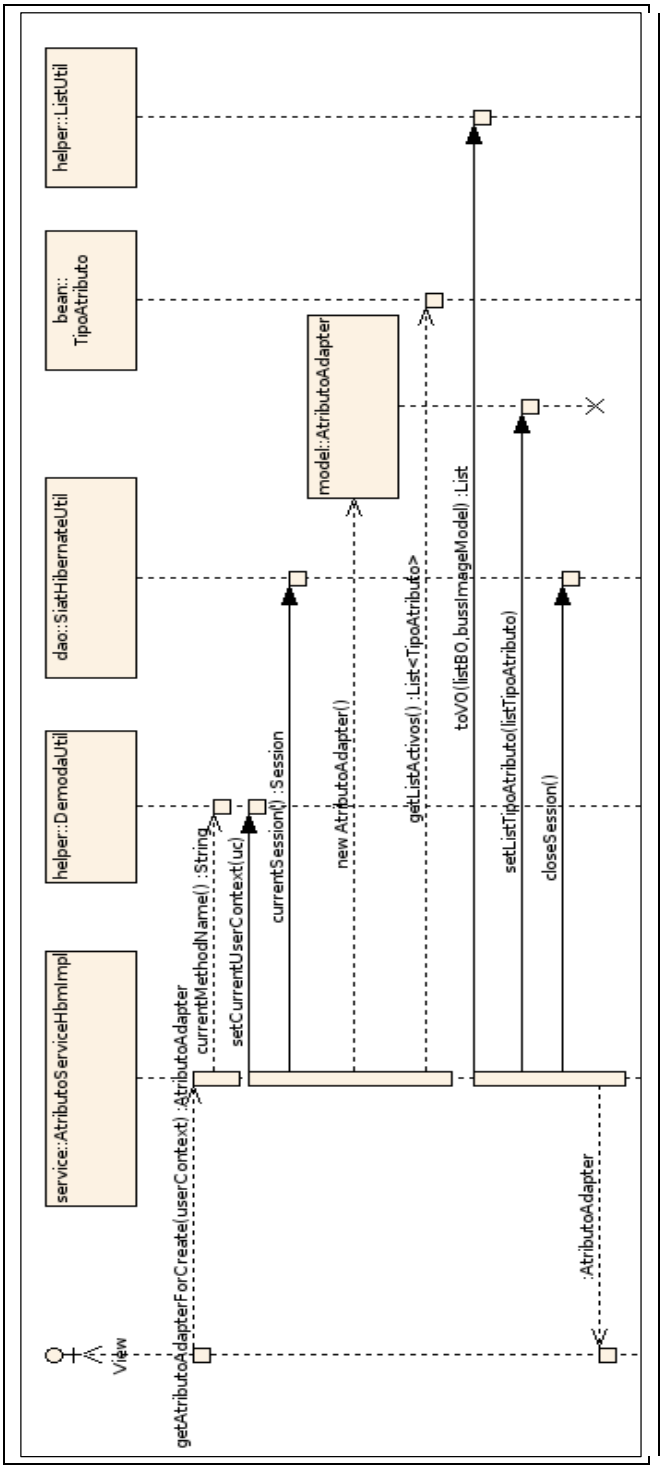
Interacción de clases del paquete buss

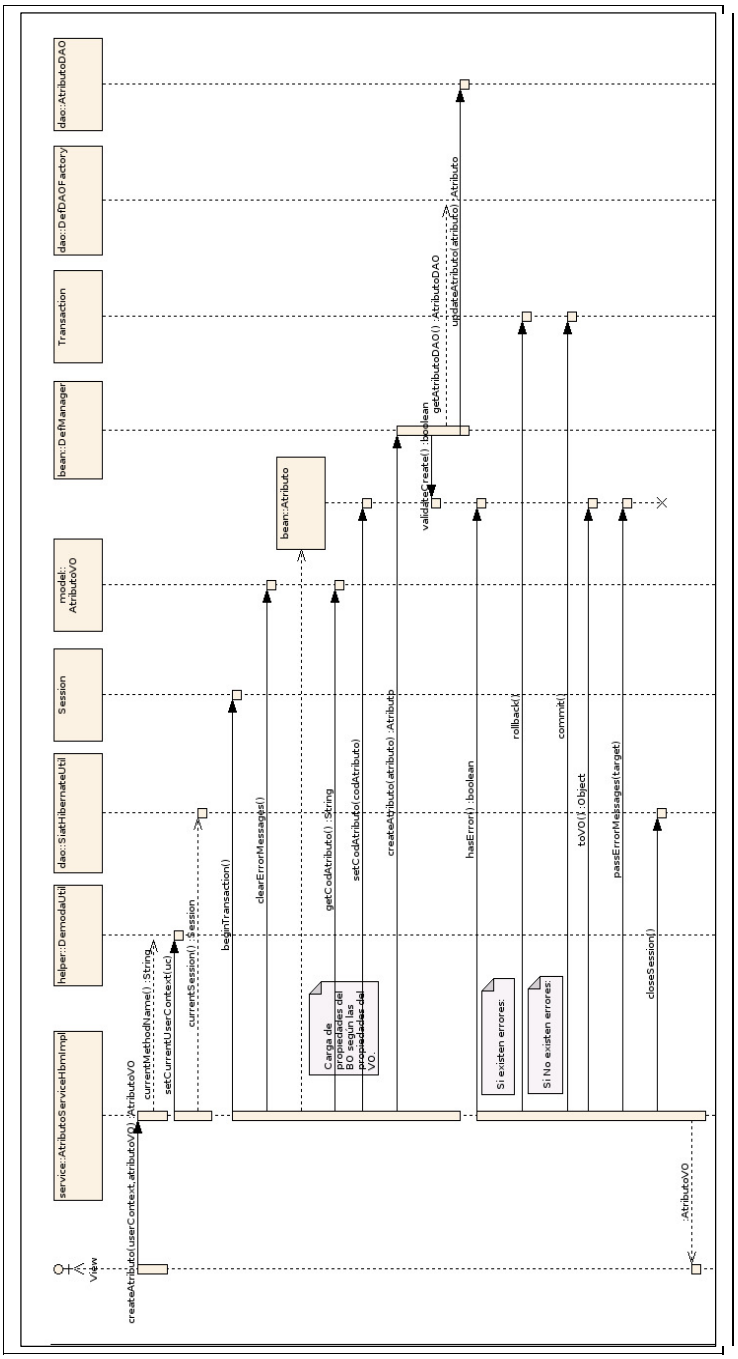
A continuación se mostraran a modo de ejemplo los diagramas de secuencia de una búsqueda, de la obtención de un adapter para una creación y de una creación. Aquí se puede ver detalladamente el funcionamiento de la capa de negocio en estas tres operaciones básicas.

El siguiente diagrama muestra la secuencia de llamadas que ocurren en el servicio cuando un usuario ingresa a una página de búsqueda de Atributos. Ver en el diagrama la llamada método *getAtributoSearchPage()*. En el mismo diagrama se muestra lo que ocurre cuando el usuario ejecuta 'buscar' luego de haber ingresado los filtros, ver la llamada *getAtributoSearchPageResult()*



Los siguientes dos diagramas muestran el caso que el usuario una vez dentro de la búsqueda decide dar de alta un Atributo. El primer diagrama ilustra como el método *getAtributoForCreate()* obtiene un Atributo con valores por defecto para mostrar y que el usuario ingrese los datos adecuados. El segundo diagrama ilustra la secuencia que ocurre cuando el usuario termina de ingresar los datos y acepta darlos de alta, método *createAtributo()*.




View
service::AtributoServiceIbnImpl
helper::DemodaUtil
dao::SiatHibernateUtil
Session
model::AtributoVO
bean::DefManager
Transaction
dao::DefDAOFactory
dao::AtributoDAO
bean::Atributo
createAtributo(userContext:atributoVO) :AtributoVO
currentMethodName() :String
setCurrentUserContext(uc)
currentSession() :Session
beginTransaction()
clearErrorMessages()
Carga de propiedades del BO segun las propiedades del VO
getCodAtributo() :String
setCodAtributo(codAtributo)
createAtributo(atributo) :Atributo
validateCreate() :boolean
getAtributoDAO() :AtributoDAO
updateAtributo(atributo) :Atributo
hasErrors() :boolean
Si existen errores:
rollback()
Si No existen errores:
commit()
toVO() :Object
passErrorMessages(target)
closeSession()
:AtributoVO

Login y Seguridad

Tanto para autenticar a usuarios como para verificar credenciales para la ejecución de acciones, SIAT utiliza los servicios de SWE y SegWeb.

La autenticación del usuario se realiza durante el Login en SIAT. Para ello existen una página JSP y el Action Login que realizan estas tareas.

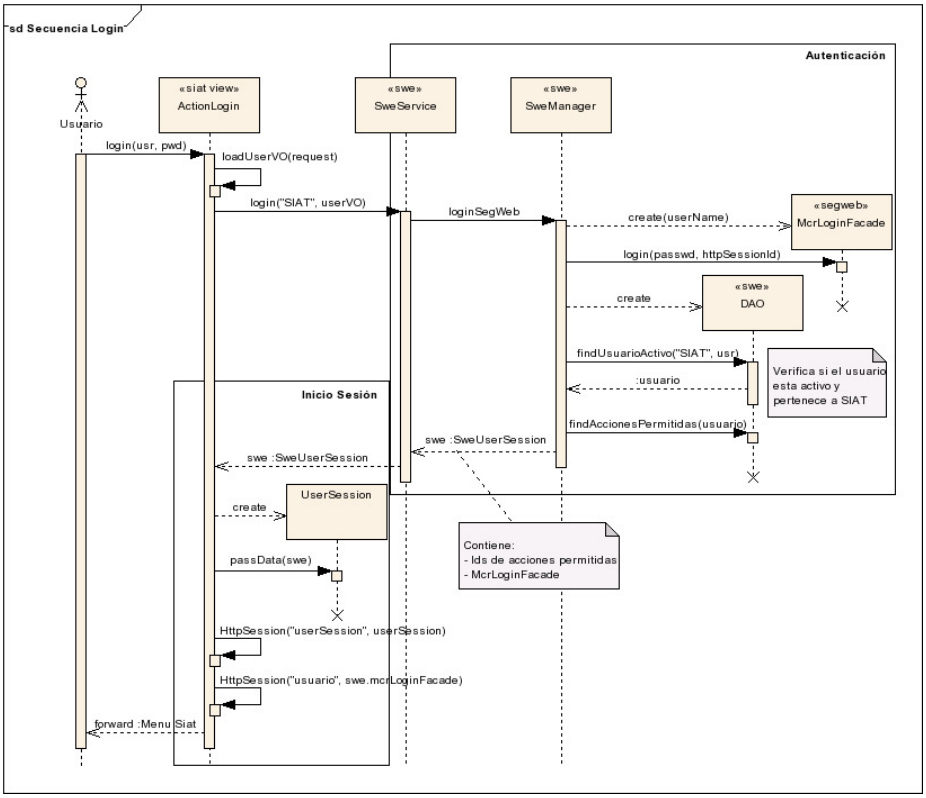
En detalle el Action Login se encarga de:

- Mostrar la página de Login
- Login de Usuario (autenticación, crear sesión de usuario)
- Cerrar la sesión de usuario
- Refrescar los datos de SWE almacenado en cache

El Login de usuario realiza los siguientes pasos:

- Obtiene los datos de Usuario y Password ingresados
- Se invoca al login() de SWE y se recupera SweUserSession
- Se verifica si el login en SWE fue exitoso.
- Se crea el UserSession de SIAT y se transfieren los datos en SweUserSession
- Se almacena en la sesión de http el UserSession de SIAT
- Se almacena en la sesión de http el McrLoginFacade de SegWeb
- Forward al Menu

El siguiente es el diagrama de secuencia del ActionLogin durante el login de un usuario. Se encuentra dividido en dos regiones: Autenticación e Inicio de Sesión

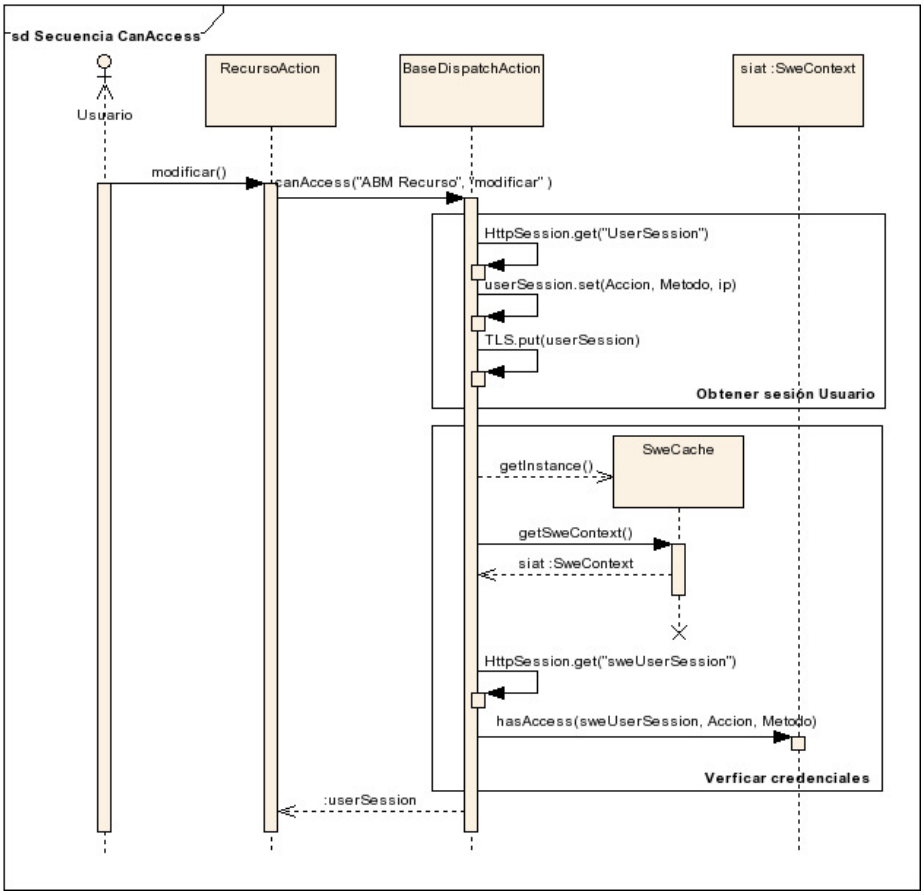


Con respecto a la verificación de credenciales SIAT controla los permisos en la vista. Esto es así por que lo primero que se ejecuta es el código de la vista, verificando permisos lo antes posible se evita la ejecución de que desde ya no están permitidas.

El chequeo de permisos lo realiza la función *canAccess()* en la vista. Los métodos de los Action deben llamar a esta función, informando el nombre de la *Acción* y *Método* de SWE contra el cual se desea verificar permisos.

La función canAccess() recibe estos datos, y obtiene las credenciales del usuario almacenadas en el HttpSession durante el Login. Con estos datos, invoca a SWE para verificar permisos.

El siguiente diagrama de secuencia ilustra los pasos realizados para la verificación de credenciales invocadas por ejemplo durante el Método modificar() Action Recurso.



Vista Desarrollo

Introducción

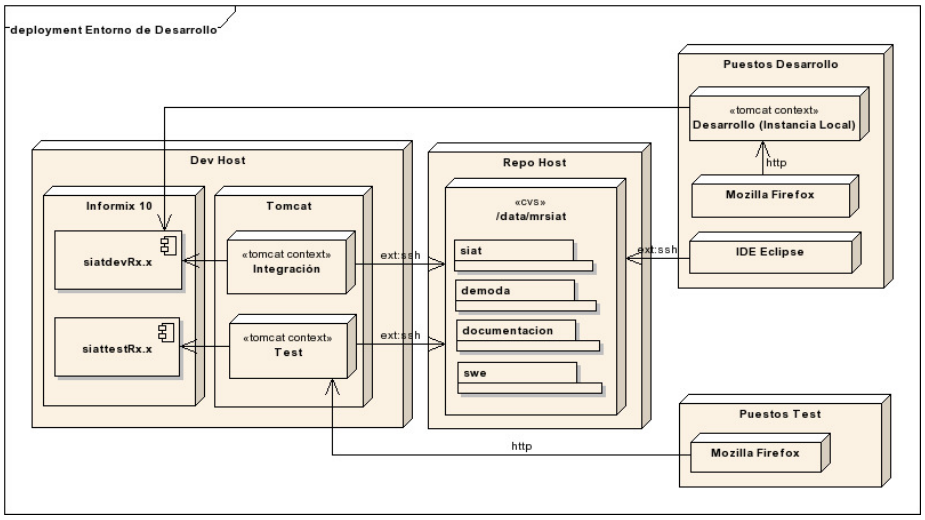
En esta sección se brinda una información más detallada sobre los componentes de cada capa, de este modo se enumeran las clases y sus características de la capa Interfaz, Negocio y finalmente Vista, utilizamos este orden ya que representa el vedadero esquema de dependencias entre si, esto es, la Vista y el Negocio no pueden existir sin la Interfaz. La vista no puede ejecutarse sin el Negocio.

Luego se explican aspectos del proceso de desarrollo con esta arquitectura. Aquí se ven los temas de entornos de trabajo, CVS, estructura de directorios del código fuente y esquema de compilación.

Entornos de Test y Desarrollo

Se utiliza un esquema de múltiples puestos de desarrollo dirigidos por un entorno de Integración y uno de Test accediendo a entornos de informix compartidos. Los puestos de Integración y Desarrollo utilizan la misma base de datos. El entorno de Test se asemeja lo más posible al de producción y utiliza un entorno de base de datos propio.

A continuación un diagrama de deploy de los entornos y aplicaciones.



- **Informix 10:**
  Para el desarrollo y test utilizamos una única instancia de base de datos Informix versión 10. Esta instancia posee varias bases de datos separadas por entorno y por release. Las DB que comienzan con nombre *siatdev* correspondes a las utilizadas para desarrollo e integración. Las DB que comienzan con nombre *siattest* corresponde a las utilizadas para Test, y deberán tener estructuras y datos similares a producción, dicho entorno se utilizaran internamente para la entrega de los releases.

  A medida que se vallan elaborando los releases se ira trabajando en distintas bases de datos para cada entorno distinguidas por la parte final de su nombre. Por ejemplo, durante el Release 1.1 se utilizaran las bases de datos *siatdevR11* y *siattestR11,* para el release 1.2 se usará *siatdevR12* y *siattestR12* y así.

- **Tomcat**
  Ademas de las instancias que existen por cada puesto de desarrollo, existen instancias de Integración y de Test para cada Release, conectadas a *siatdevRxx* y *siattestRxx* respectivamente.

- **Repositorio CVS**
  El repositorio de código fuente y documentación se encuentra en un host separado. El repositorio del SIAT contiene los módulos *siat, swe, demoda, documentacion*,

- **Puestos de Desarrollo**
  En cada puesto se encuentra una instancia de tomcat y el IDE Eclipse para desarrollar y probar localmente. Todas las instancias de tomcat de estos puestos, se conectan a una misma base de datos de informix.

- **Puestos de Test**
  Los puestos se conectan a la instancia de Tomcat del entorno Test. Ocasionalmente podrán utilizar un entorno local para probar las versiones de desarrollo.

Estructura de directorios y Archivos de código fuente

Desde el punto de vista de desarrollo los archivos de clases y recursos de SIAT también se dividen según su capa.
La siguiente es una tabla que describe los directorios más importantes respecto de la ubicación del código fuente en módulo siat de CVS.

Directorios	Archivos	Descripción
`iface/`		Archivos de la capa Interfaz
`iface/src`		Código fuente de la capa Interfaz
`buss/`		Archivos de la capa Negocio
`buss/src`		Código fuente de la capa Negocio
`view`		Archivos de la capa Vista
`view/src`		Código fuente de archivos JSP. Posee un directorio por cada módulo, y un subdirectorio por cada submodulo, dentro están los JSP de los CUS correspondiente al submodulo.
`view/src/WEB-INF`	`struts-config.xml`	Configuración de struts general, este archivo hace referencia a otros .xml en un directorio por modulo, ver próxima fila
	`moduloX/config/struts-config-moduloX.xml`	Configuración de struts específica de un módulo del SIAT. Cada archivo existe dentro de un directorio por módulo
	`tiles-defs-xml`	Configuración tiles de struts general, este archivo hace referencia a otros .xml en un directorio por modulo, ver próxima fila
	`moduloX/config/tiles-defs-moduloX.xml`	Configuración de tiles de struts específica de un módulo del SIAT. Cada archivo existe dentro de un directorio por módulo.

Directorios	Archivos	Descripción
view/src/WEB-INF/resources	*.properties	Propiedades de Mensajes de presentación. Mensajes de etiquetas, advertencias, errores, etc que aparecen en la presentación.
view/src/WEB-INF/src	log4j.properties	Configuración de log4j
	siat.hibernate.cfg	Configuración del SessionFactory de Hibernate para el SIAT
	swe.hibernate.cfg	Configuración del SessioFactory de Hibernate para el SWE

Compilación y archivos generados

El módulo siat del CVS posee los siguientes directorios y archivos utilizados durante la compilación.

Archivo/Directorio	Descripción
/build.xml	Script ant que invoca en secuencia a los script ant de cada capa. Invoca: • /iface/build.xml • /buss/build.xml • /view/build.xml Este script es provisto para simplificar el ciclo de desarrollo y para poder automatizar el esquema de deploy. Otra alternativa es utilizar el script ant de la capa modificada y luego el script de la capa view. *ver demas adenlante...*
iface/build.xml	Script ant de la Interfaz, realiza: • Clean de la corrida anterior • Compilación de iface/src
iface/bin	Directorio temporal utilizado durante la compilación de ant del iface
buss/build.xml	Script ant de la capa Negocio • Clean de la corrida anterior • Compilación de buss/src classpath: iface/bin buss/localJars
buss/localJars	Jars utilizados por buss, que **no se desplegan en WEB-INF/lib.** Se espera que estos jars esten desplegados en el shared/lib o common/lib de tomcat.
buss/bin	Directorio temporal utilizado durante la compilación de ant buss
view/build.xml	Script ant de view • Clean de la corrida anterior • Copia de los jars de iface, buss • Compilacion de WEB-INF/src • Generacion de WARs

Archivo/Directorio	Descripción
`externalJars/lib`	Directorio utilizado como classpath para la compilación de iface, buss, view. **Importante:** Este directorio también se utiliza para realizar el deploy de view, todos los jars que aparezcan aquí se copiaran al WEB-INF/lib de view.
`build/`	Directotio temporal utilizado para la compilación de eclipse. Se provee este directorio para Facilitar el trabajo con el entorno Eclipse. Este directorio no es utilizado por el esquema de compilación y deploy con ant.
`/dist/webapps`	En este directorio se generan los archivos el archivo WAR de SIAT. Ubicación final de *intersiat.war, intrasiat.war*
`/dist/lib`	jars de cada módulo del SIAT generado durante la comilación. Estos módulos son incluidos en los WARs según corresponda.

Pasos para la Compilación

La compilación de la aplicación se realizará mediante scripts de Ant, que podrán ser ejecutados dentro del IDE Eclipse o directamente sobre la línea de comandos.

El procedimiento para la compilación se resume en los siguientes pasos a realizar:

Por única vez:

- Checkout desde el CVS del modulo *siat*
- En eclipse:
  Menu: Windows -> Show View -> Ant
  Sobre Vista Ant: Boton de la derecha -> Add Build Files
  Seleccionar: siat/build.xml (*leer primer fila de la tabla superior...*)

Durante el ciclo Programación – Prueba:

- Modificar código
- Ejecutar siat/build.xml
- Reiniciar Tomcat
- Realizar Prueba

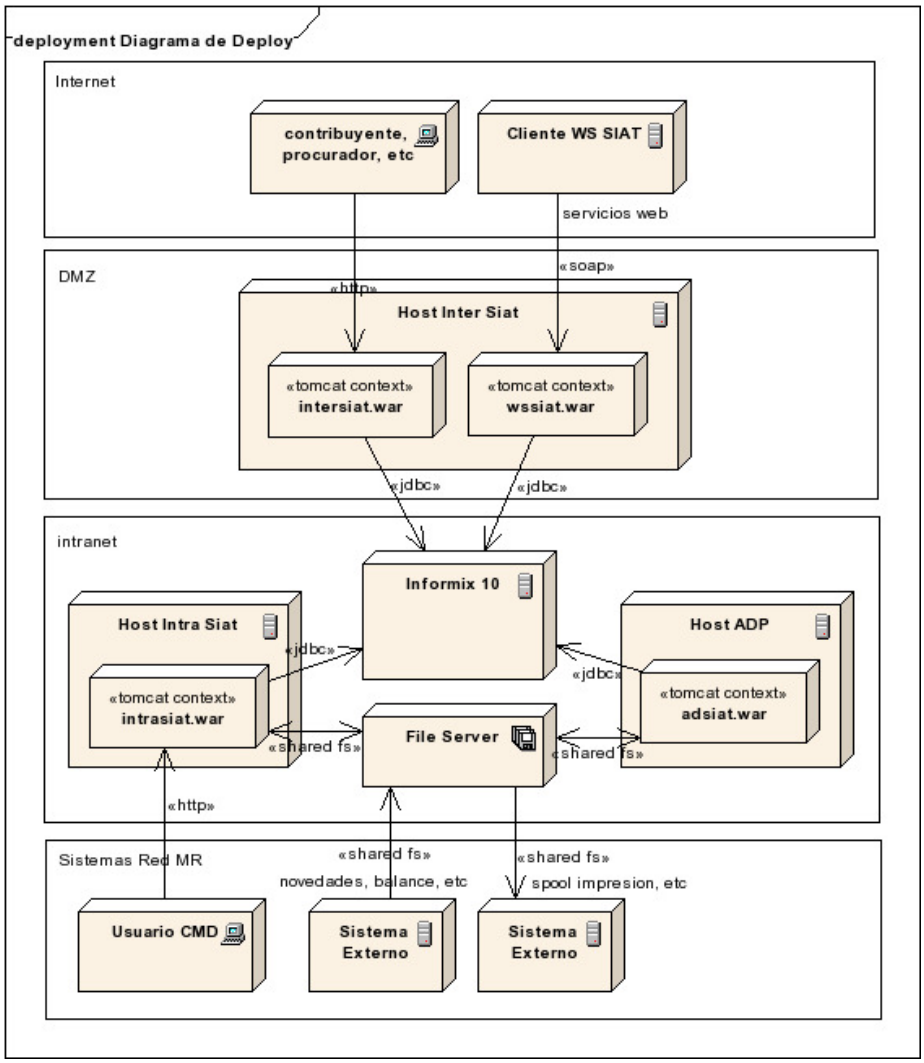
Vista Física

Datos de Despliegue en General

Durante la compilación de SIAT se generan los siguientes componentes que forman parte del despliegue.

- *siat[moduloX].jar*
  Son todos los .jar de cada módulo del SIAT. Estos archivos se generan durante la compilación y existe uno por cada módulo.
- *intrasiat.war*
  Archivo de aplicación a desplegar en host interno de la red de la Municipalidad de Rosario. Los usuarios en la intranet utilizaran esta aplicación. Chequea accesos vía SWE.
- *intersiat.war*
  Archivo de aplicación a desplegar en host expuesto a internet. Los usuarios de esta aplicación serán mayormente los contribuyentes conectados a través de internet con el rol 'Anonimo'. Ademas de chequear acceso vía SWE posee filtros por URL configurados para evitar el acceso a funcionalidades no deseadas.
- *adpsiat.war*
  Aplicación a desplegar en el host destinado a la ejecución de procesos. Posee la funcionalidad core de ADP (Adminsitrador de Procesos), y la lógica de los procesos programados para el SIAT. Chequea accesos vía SWE. *(Ver Anexo ADP)*
- *wssiat.war*
  Aplicación que expone las funcionalidades del SIAT expuestas como Web Services. Chequea accesos utilizando SWE.
  IMPORTANTE: Llegado el momento se evaluará la necesidad de utilizar este WAR ya que tal vez convenga exponer dichos WS en *intersiat.war* para mantener lo más sencillo posible el proceso de deploy.

Todos estos componentes se despliegan según el siguiente diagrama:



Entorno de Implementacion

Para realizar las distintas implementaciones se cuenta con un entorno de implementación en las oficinas de la Municipalidad de Rosario.

Dicho entorno se trata de un sistema operativo GNU/Linux con capacidad para conectarse al CVS de las oficinas de Tecso Rosario usando SSH, ademas tiene instalado las herramientas de compilación y deploy necesarias para SIAT así como una instancia de Apache Tomcat para ejecutar SIAT y sus dependencias (SWE, ADP, etc). Dicha instancia posee definida los Data Sources necesarios para SIAT, SWE y ADP.

La siguiente es una tabla de los data sources requeridos por cada aplicación, y su vinculación con los recursos definidos globalmente en la instancia de Tomcat del SIAT.

SIAT		
Descripción	**Data Source Local**	**Data Source Global**
Base de datos de SIAT	ds/siat	jdbc/siat
Base de datos Seguridad Web con parte Extendida	ds/swe	jdbc/seguridadweb10
Base de datos Sistema de GeoRefernciacion	ds/gisdb	jdbc/gis10
Base de datos de Personas de la Municipalidad de Rosario.	ds/generaldb	jdbc/general10

ADPSIAT		
Descripción	**Data Source Local**	**Data Source Global**
Base de datos de SIAT	ds/siat	jdbc/siat
Base de datos Seguridad Web con parte Extendida	ds/swe	jdbc/seguridadweb10
Base de datos Sistema de GeoRefernciacion	ds/gisdb	jdbc/gis10
Base de datos de Personas de la Municipalidad de Rosario.	ds/generaldb	jdbc/general10

SWE		
Descripción	**Data Source SIAT**	**Data Source Global**
Base de datos Seguridad Web con parte Extendida	ds/swe	jdbc/seguridadweb10

Escenarios de la arquitectura

A continuación, se muestran una serie de secuencia que ilustra la interacción entre las distintas partes de la arquitectura de Siat.

Diagrama de secuencia Caso Modificar DomAtr: Obtiene una página Encabezado Detalle

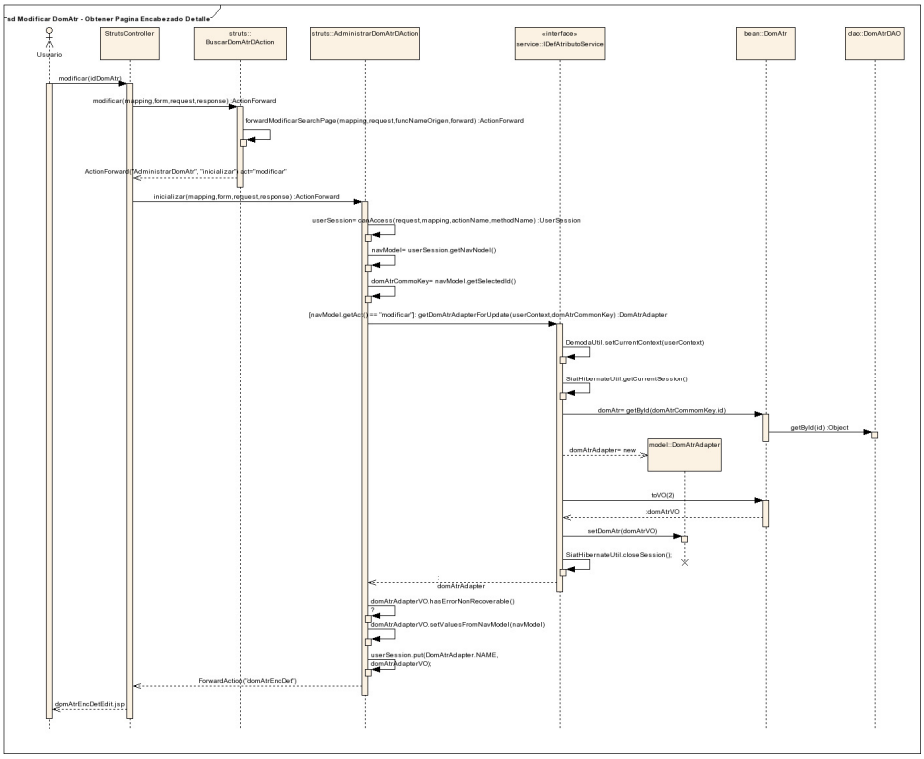


Diagrama de secuencia Caso Modificar Encabezado DomAtr: Obtiene los datos para edición.

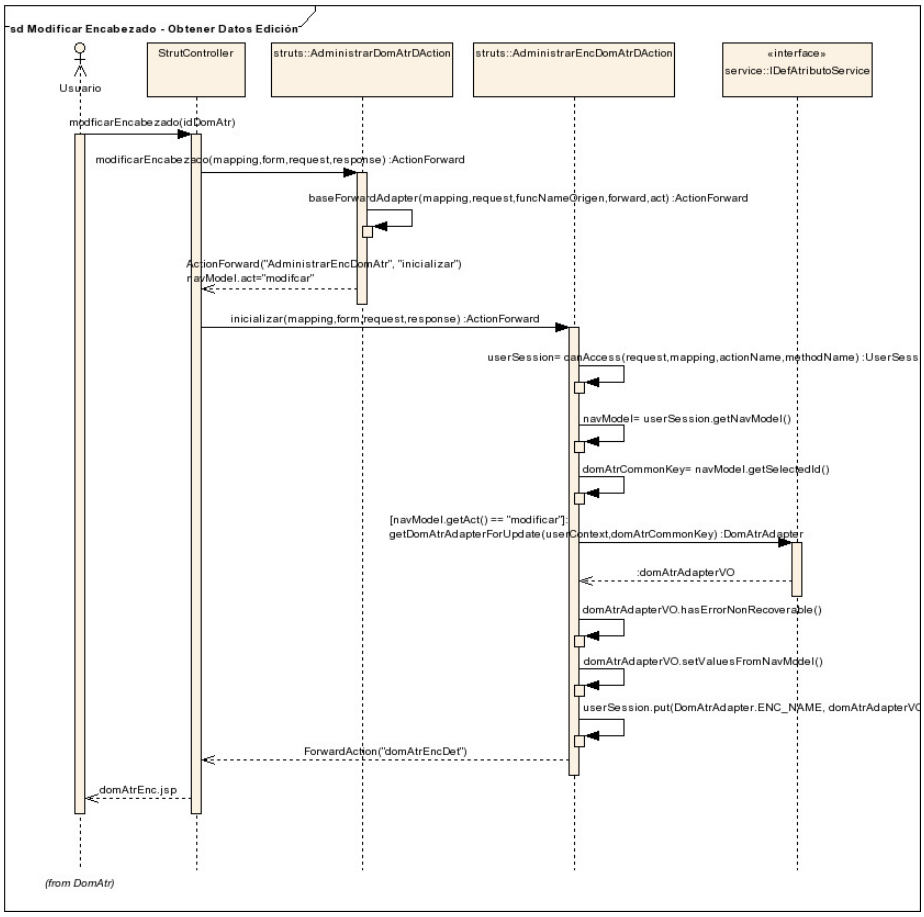
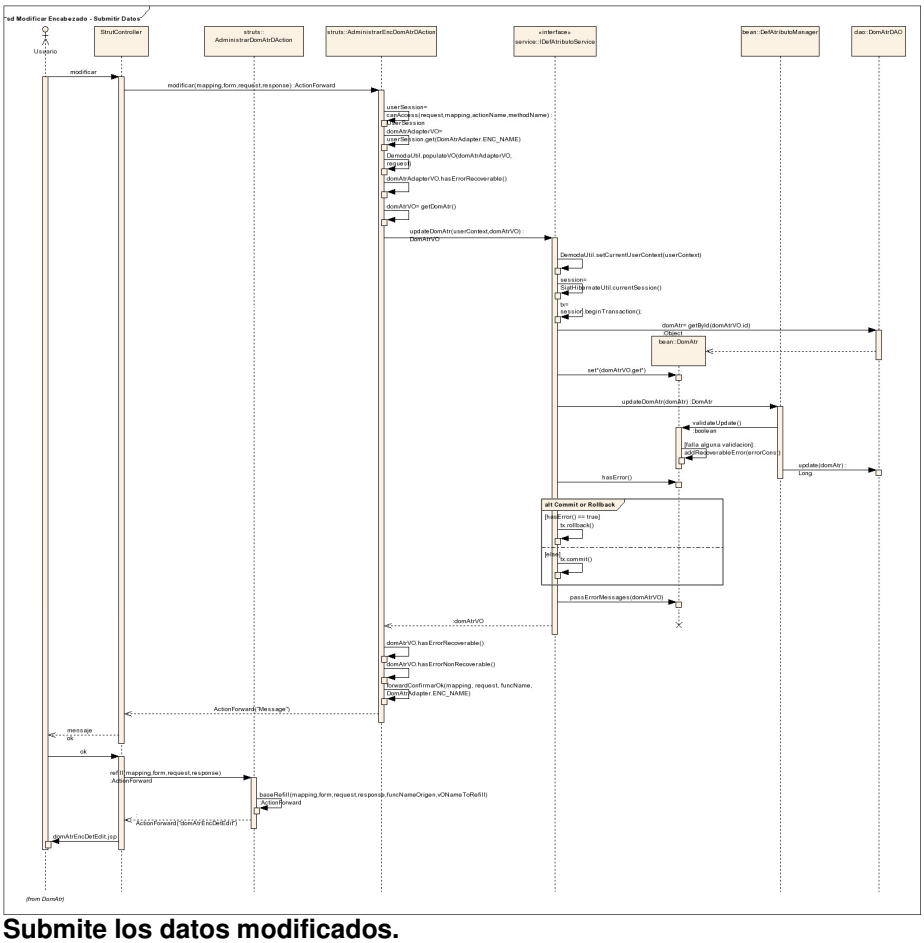


Diagrama de secuencia Caso Modificar Encabezado DomAtr:



Submite los datos modificados.

Diagrama de secuencia Caso Modificar DomAtrVal: Obtiene los datos para edicion de un Detalle

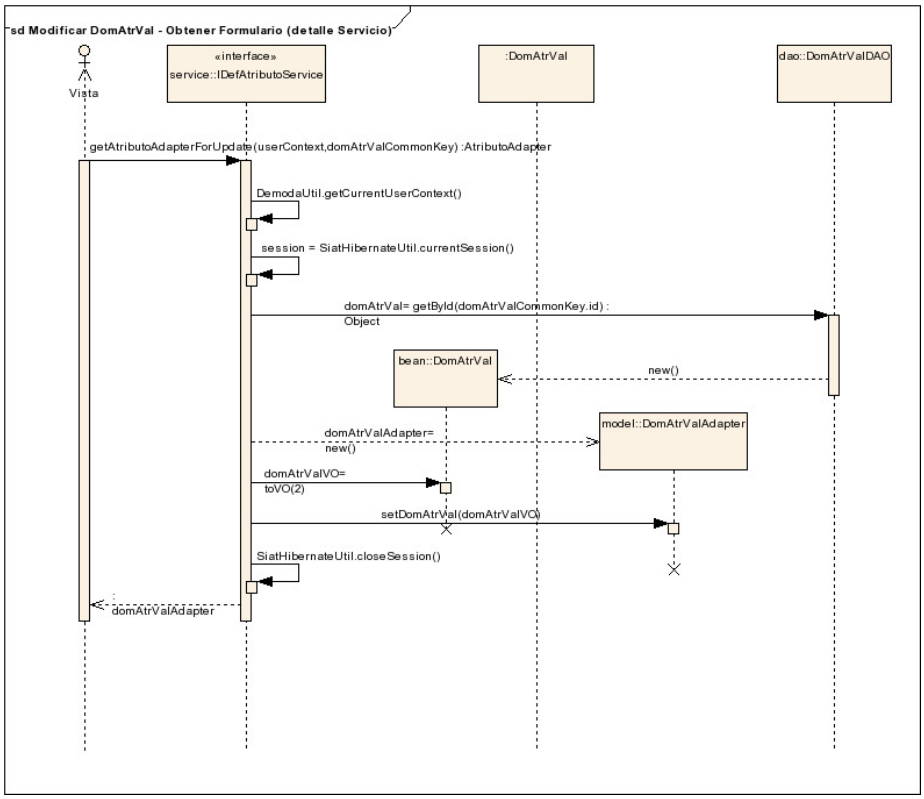


Diagrama de secuencia Caso Modificar DomAtrVal: Submite los datos de un Detalle