Universidad de las Ciencias Informáticas
Facultad 3

Trabajo de Diploma

Implementación de la capa de Acceso a Datos de los módulos de Administración y Configuración del Proyecto Convenio Integral de Cooperación Cuba Venezuela

Autor: José Manuel Borroto Cintra
Tutor: Ing. Daynier Ruiz Rodriguez

La Habana, Cuba
Julio, 2009

RESUMEN

En este trabajo se expone el desarrollo de las capas de acceso a datos de los módulos de Administración y Configuración del Proyecto Convenio Integral de Cooperación Cuba Venezuela. El cual se desarrolló principalmente utilizando como lenguaje para su programación la plataforma *Java 2 Enterprise Edition* con la JDK 6.0, el *framework* de integración Spring y el *framework* de persistencia Hibernate 3.2. Todas estas herramientas integradas en el IDE de desarrollo Eclipse 3.3

Estos módulos dentro de sus objetivos principales se encuentran la seguridad de la aplicación el manejo de usuarios y el control de todas las configuraciones y principales nomencladores de la aplicación. Para que estos funcionen de una manera adecuada es muy importante un manejo eficiente de los datos y este es el tema principal de este trabajo.

PALABRAS CLAVES

- CCV.
- Implementación.
- Hibernate.
- DAO.
- Java.

TABLA DE CONTENIDO

RESUMEN I
Introducción 1
Problema científico. 2
Objeto de Estudio. 2
Campo de acción. 2
Objetivo. 3
Hipótesis. 3
Tareas de la Investigación. 3
Métodos Científicos. 4
CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA 5
Introducción al capítulo. 5
Metodología usada para el desarrollo. 5
Metodologías pesadas (RUP). 6
Bases de programación. 7
Paradigmas de programación. 8
Java como leguaje orientado a objetos. 11
Motores de Persistencia. 18
Framework de persistencia Hibernate. 20
Integración a Spring de Hibernate. 28
Estándares de codificación. 32
Conclusiones del Capítulo. 35

CaPíTULO 2: Implementación 36
Introducción al capítulo........ 36
Estructura distribución y componentes de la capa de persistencia. 36
Configuraciones. 40
Modelo de diseño y Modelo de datos. 45
Implementación. 69
Conclusiones del Capitulo. 84
Capítulo 3: Pruebas de Unidad 85
Introducción al capítulo........ 85
Pruebas de Unidad con JUnit. 85
Casos de Prueba........ 89
Ejemplos de pruebas........ 100
Conclusiones del capítulo........ 104
Conclusiones 105
Bibliografía........ 106
Glosario de Términos........ 108

INTRODUCCIÓN

En la situación de la economía internacional actual en la cual nuestros países de Latinoamérica deben enfrentar el mercado agresivo e interesado del Capitalismo, ha surgido una posición que desafía todo esto y es la propuesta de nuestro país y nuestra vecina y aliada nación Venezuela, el ALBA. Basados en un espíritu de intercambio y amistad nuestros gobiernos llevan a cabo un grupo de acuerdos y proyectos de cooperación que benefician a ambas partes desarrollando las esferas socio económico, cultural e investigativa de ambos países.

La mayoría de estos acuerdo se ejecutan bajo un marco que se ha venido a llamar “Mixtas”; reuniones anuales en las que básicamente se determinan y aprueban los proyectos que se llevarán a cabo durante ese período. Con el éxito de esta forma de relaciones se ha venido un crecimiento exponencial del número de proyectos que se aprueban cada año derivando en un nuevo tipo de problema para la administración de ambos países el cómo mantener bajo un marco controlado y centralizado los mismos.

Para enfrentar el problema Cuba necesita poner en funcionamiento una solución informática (Convenio Integral de Cooperación Cuba Venezuela), la cual consiste en una aplicación de gestión empresarial la cual automatiza, centraliza y facilita a las secretarías de ambos países el control sobre los proyectos. Por su premura se desplegó una primera fase o “Solución rápida” que se encuentra actualmente en funcionamiento y que tenía como objetivos principales, recoger la información existente hasta el momento de los proyectos en ejecución entre ambos países y la información de los organismos implicados en los mismos.

Además, esta buscaba ganar tiempo para el desarrollo y puesta en funcionamiento de una segunda fase, que tiene como principales objetivos cumplir con muchas de las necesidades que se tenían desde un principio pero que no se alcanzaron a solucionar en la versión anterior por su demora, aumentar la posibilidades de control y seguimiento de los proyectos además de posibilitar la presentación de los

nuevos proyectos a través de esta y adicionar módulos que permitan en cierto modo controlar también los contratos y mantener un seguimiento y control de las actividades y cronogramas de los proyectos.
Debido a la cantidad de nuevos requisitos funcionales que debe cumplir la nueva aplicación se ha producido un aumento de la cantidad y sensibilidad de datos a manejar y además por el crecimiento de la cantidad de usuarios del sistema lo que también influye en la cantidad de datos que se deben gestionar de forma simultánea y sumando a todo esto que en una aplicación de gestión empresarial los datos juegan uno de los papeles más importantes se hace necesario el desarrollo de una capa de acceso a datos rápida robusta y estable que garantices la integridad y el manejo adecuados de la información.

Siendo la aplicación de un tamaño considerable se hizo necesario la creación de dos módulos independientes que garanticen la seguridad y la correcta configuración de la aplicación, al igual que el resto de la aplicación estos necesitan la integridad rapidez y robustez del acceso a datos y los datos que los mismos manejan debido a que de estos depende el correcto manejo de los usuarios con todo lo que esto conlleva además de la correcta configuración de todos los nomencladores y información general que necesita la aplicación para su correcto funcionamiento.

Problema científico.

¿Cómo implementar la capa de acceso a datos de los módulos de Administración y Configuración de la solución informática del Proyecto Convenio Integral de Cooperación Cuba Venezuela?

Objeto de Estudio.

Capa de persistencia de datos.

Campo de acción.

Flujo de implementación de la capa de acceso a datos de los módulos de Administración y Configuración.

Objetivo.

Implementar la capa de acceso a datos de los módulos de Administración y Configuración del Proyecto Convenio Integral de Cooperación Cuba Venezuela cumpliendo con los estándares de Java, utilizando correctamente los patrones de implementación y los *frameworks* de desarrollo especializados Hibernate y Spring.

Hipótesis.

Si se implementa la capa de acceso a datos de los módulos de Administración y Configuración del Proyecto Convenio Integral de Cooperación Cuba Venezuela, cumpliendo con los estándares de Java, utilizando correctamente los patrones de implementación y los *frameworks* de desarrollo especializados (Hibernate y Spring) se logrará un manejo eficiente de los datos, una gran estabilidad e integración y se acortarán los tiempos de implementación.

Tareas de la Investigación.

- Realización de un estudio de la bibliografía especializada de programación en la plataforma Java y utilización e integración de los *frameworks* Hibernate y Spring.
- Implementación utilizando las ventajas de los *frameworks* de desarrollo especializados en la implementación de las capas de acceso a datos de los módulos de Administración y Configuración del Proyecto Convenio Integral de Cooperación Cuba Venezuela
- Validación realizando pruebas de unidad con el *framework* JUnit a las que se pueden sumar las que se realizarán posteriormente por el grupo de calidad de *software* de la facultad y la universidad.

Métodos Científicos.

Teóricos:

analítico – sintético: permitió detectar la necesidad del módulo de comenzar con el flujo de implementación de la capa de acceso a datos utilizando *frameworks* especializados que mejorarán el rendimiento de la misma y optimizarán los tiempos de desarrollo.

inductivo – deductivo: ayudo al módulo a detectar la necesidad de realizar la implementación de la capa de acceso a datos utilizando *frameworks* especializados a partir de la experiencia con los problemas que se habían producido al programar la misma sin el uso de estas.

histórico – lógico: permitió definir los principales problemas y errores que se cometían al implementar la capa de acceso a datos sin la utilización de *frameworks* especializados y de esta forma identificar las principales ventajas de las que se podía tomar provecho al utilizar las mismas.

hipotético – deductivo: permitió identificar las posibles soluciones que se le podrían aplicar en el proceso de desarrollo de la capa de acceso a datos a partir de la utilización de las *frameworks* especializados que eliminen los errores que se cometen sin la utilización de estas.

Empíricos:

observación: permitió ir desarrollando conocimientos y tener una concepción más realista del flujo de desarrollo que se fueron acumulando a largo de la experiencia en el desarrollo de aplicaciones.

CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA

Introducción al capítulo.

En este capítulo se realizará un análisis de las principales materias en los que se soporta esta investigación. Se irán discutiendo en este orden la metodología de desarrollo que se uso, los paradigmas de programación que se aplicaron junto con el lenguaje que permiten la aplicación del mismo. Por último se describirán los motores de persistencia, porque la necesidad de su uso e importancia y entraremos a analizar el *framework* de trabajo Hibernate específicamente su integración con Spring.

Metodología usada para el desarrollo.

Las metodologías ingenieriles han estado presentes durante mucho tiempo. No se han distinguido precisamente por ser muy exitosas. Aún menos por su popularidad. La crítica más frecuente a estas metodologías es que son burocráticas. Hay tanto que hacer para seguir la metodología que el ritmo entero del desarrollo se retarda.

Hoy en día existen numerosas propuestas metodológicas que inciden en distintas dimensiones del proceso de desarrollo. Un ejemplo de ellas son las propuestas tradicionales centradas específicamente en el control del proceso. Estas han demostrado ser efectivas y necesarias en un gran número de proyectos, sobre todo aquellos proyectos de gran tamaño (respecto a tiempo y recursos). (1)

Metodologías pesadas (RUP).

El proceso unificado de desarrollo (RUP) es una metodología para la ingeniería de *software* que va más allá del mero análisis y diseño orientado a objetos para proporcionar una familia de técnicas que soportan el ciclo completo de desarrollo de *software*. El resultado es un proceso basado en componentes, dirigido por los casos de uso, centrado en la arquitectura, iterativo e incremental.

Características principales de RUP.

"Guiado por los Casos de Uso: Los Casos de Uso son el instrumento para validar la arquitectura del *software* y extraer los casos de prueba."
"Centrado en la arquitectura: Los modelos son proyecciones del análisis y el diseño constituye la arquitectura del producto a desarrollar."
"Iterativo e incremental: Durante todo el proceso de desarrollo se producen versiones incrementales (que se acercan al producto terminado) del producto en desarrollo."

Beneficios que aporta RUP.

Permite desarrollar aplicaciones sacando el máximo provecho de las nuevas tecnologías, mejorando la calidad, el rendimiento, la reutilización, la seguridad y el mantenimiento del *software* mediante una gestión sistemática de los riesgos.
Permite la producción de *software* que cumpla con las necesidades de los usuarios, a través de la especificación de los requisitos, con una agenda y costo predecible.
Enriquece la productividad en equipo y proporciona prácticas óptimas de *software* a todos sus miembros.
Permite llevar a cabo el proceso de desarrollo práctico, brindando amplias guías, plantillas y ejemplos para todas las actividades críticas.
Proporciona guías explicitas para áreas tales como modelado de negocios, arquitectura Web, pruebas y calidad.

Se integra estrechamente con herramientas de modelación, permitiendo a los equipos de desarrollo aprovechar todas las ventajas de las características de las mismas, el Lenguaje de Modelado Unificado (UML) y otras prácticas óptimas de la industria.

Unifica todo el equipo de desarrollo de *software* y mejora la comunicación al brindar a cada miembro del mismo una base de conocimientos, un lenguaje de modelado y un punto de vista de cómo desarrollar *software*.

Optimiza la productividad de cada miembro del equipo al poner al alcance la experiencia derivada de miles de proyectos y muchos líderes de la industria.

No solo garantiza que los proyectos abordados serán ejecutados íntegramente sino que además evita desviaciones importantes respecto a los plazos.

Permite una definición acertada del sistema en un inicio para hacer innecesarias las reconstrucciones parciales posteriores. (2)

Bases de programación.

Junto con la evolución de las computadoras han evolucionado los sistemas, herramientas, programas y leguajes. La complejidad de los mismos ha ido en un constante aumento lo cual ha ido dejando obsoletos a los antiguos modelos y dando lugar a nuevos paradigmas que se adapten a las nuevas exigencias. Junto a la creación de estos paradigmas que son solo la teoría o el enfoque surgen numerosos leguajes que se encargan de dar forma e implementar estos conceptos en la práctica. Luego surgen herramientas que dan fortalezas a estos lenguajes y multiplican sus posibilidades.

Paradigmas de programación.

Los paradigmas son un conjunto de conocimientos y creencias que forman una visión del mundo (cosmovisión), en torno a una teoría hegemónica en determinado periodo histórico. Cada paradigma se instaura tras una revolución científica, que aporta respuestas a los enigmas que no podían resolverse en el paradigma anterior. Una de las características fundamentales, su inconmensurabilidad: ya que ninguno puede considerarse mejor o peor que el otro. Además, cuentan con el consenso total de la comunidad científica que los representa.

Los paradigmas cumplen una doble función, por un lado, la positiva que consiste en determinar las direcciones en las que ha de desarrollarse la ciencia normal, por medio de la propuesta de enigmas a resolver dentro del contexto de las teorías aceptadas. Por otro lado la función negativa del paradigma, es la de establecer los límites de lo que ha de considerarse ciencia durante el tiempo de su hegemonía.

Un paradigma de programación es una colección de modelos conceptuales que juntos modelan el proceso de diseño y determinan, al final, la estructura de un programa.

Esa estructura conceptual de modelos está pensada de forma que esos modelos determinan la forma correcta de los programas y controlan el modo en que pensamos y formulamos soluciones, y al llegar a la solución, ésta se debe de expresar mediante un lenguaje de programación. Para que este proceso sea efectivo las características del lenguaje deben reflejar adecuadamente los modelos conceptuales de ese paradigma. Cuando un lenguaje refleja bien un paradigma particular, se dice que soporta el paradigma, y en la práctica un lenguaje que soporta correctamente un paradigma, es difícil distinguirlo del propio paradigma, por lo que se identifica con él.

Floyd describió tres categorías de paradigmas de programación:

> Los que soportan técnicas de programación de bajo nivel (ej.: copia de ficheros frente estructuras de datos compartidos)

Los que soportan métodos de diseño de algoritmos (ej.: divide y vencerás, programación dinámica, etc.)

Los que soportan soluciones de programación de alto nivel, como los descritos en el punto anterior

Floyd también señala lo diferentes que resultan los lenguajes de programación que soportan cada una de estas categorías de paradigmas. Sólo comentaremos los paradigmas relacionados con la programación de alto nivel. Se agrupan en tres categorías de acuerdo con la solución que aportan para resolver el problema

Solución procedimental u operacional. Describe etapa a etapa el modo de construir la solución. Es decir señala la forma de obtener la solución.

Solución demostrativa. Es una variante de la procedimental. Especifica la solución describiendo ejemplos y permitiendo que el sistema generalice la solución de estos ejemplos para otros casos. Aunque es fundamentalmente procedimental, el hecho de producir resultados muy diferentes a ésta, hace que sea tratada como una categoría separada.

Solución declarativa. Señala las características que debe tener la solución, sin describir cómo procesarla. Es decir señala qué se desea obtener pero no cómo obtenerlo.

Paradigmas procedimentales u operacionales La característica fundamental de estos paradigmas es la secuencia computacional realizada etapa a etapa para resolver el problema. Su mayor dificultad reside en determinar si el valor computado es una solución correcta del problema, por lo que se han desarrollado multitud de técnicas de depuración y verificación para probar la corrección de los problemas desarrollados basándose en este tipo de paradigmas. Pueden ser de dos tipos básicos: Los que actúan modificando repetidamente la representación de sus datos (efecto de lado); y los que actúan creando nuevos datos continuamente (sin efecto de lado). Los paradigmas con efecto de lado utilizan un modelo en el que las variables están estrechamente relacionadas con direcciones de la memoria del ordenador. Cuando se ejecuta el programa, el contenido de estas direcciones se actualiza repetidamente, pues las variables reciben múltiples asignaciones, y al finalizar el trabajo, los valores finales de las variables representan el resultado. Existen dos tipos de paradigmas con efectos de lado:

- el imperativo.
- el orientado a objetos.

La Programación Orientada a Objetos (POO u OOP según siglas en inglés) es un paradigma de programación que define los programas en términos de "clases de objetos", objetos que son entidades que combinan estado (es decir, datos), comportamiento (esto es, procedimientos o métodos) e identidad (propiedad del objeto que lo diferencia del resto). La programación orientada a objetos expresa un programa como un conjunto de estos objetos, que colaboran entre ellos para realizar tareas. Esto permite hacer los programas y módulos más fáciles de escribir, mantener y reutilizar.

De esta forma, un objeto contiene toda la información, (los denominados atributos) que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases (e incluso entre objetos de una misma clase, al poder tener valores bien diferenciados en sus atributos). A su vez, dispone de mecanismos de interacción (los llamados métodos) que favorecen la comunicación entre objetos (de una misma clase o de distintas), y en consecuencia, el cambio de estado en los propios objetos. Esta característica lleva a tratarlos como unidades indivisibles, en las que no se separan (ni deben separarse) información (datos) y procesamiento (métodos).

Dada esta propiedad de conjunto de una clase de objetos, que al contar con una serie de atributos definitorios, requiere de unos métodos para poder tratarlos (lo que hace que ambos conceptos están íntimamente entrelazados), el programador debe pensar indistintamente en ambos términos, ya que no debe nunca separar o dar mayor importancia a los atributos en favor de los métodos, ni viceversa. Hacerlo puede llevar al programador a seguir el hábito erróneo de crear clases contenedoras de información por un lado y clases con métodos que manejen esa información por otro (llegando a una programación estructurada camuflada en un lenguaje de programación orientado a objetos).

Esto difiere de los lenguajes imperativos tradicionales, en los que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos de entrada para obtener otros de salida. La programación estructurada anima al programador a pensar sobre todo en términos de procedimientos o funciones, y en segundo lugar en las estructuras de datos que esos

procedimientos manejan. Los programadores de lenguajes imperativos escriben funciones y después les pasan datos. Los programadores que emplean lenguajes orientados a objetos definen objetos con datos y métodos y después envían mensajes a los objetos diciendo qué realicen esos métodos en sí mismos. (3)

Java como leguaje orientado a objetos.

Existen algunos lenguajes de programación en los que todo debe definirse de acuerdo con la filosofía de la programación orientada a objetos, convirtiéndose en una programación demasiado compleja; también los hay que no soportan esta filosofía, que son los lenguajes estructurados simples.

Java esta en un punto intermedio, su modelo de objetos es sencillo e incluye algunos tipos de datos (como los arreglos) que no únicamente se utilizan en el modelo de programación orientada a objetos, acercándose más al modelo de lenguaje estructurado. Las características de Java son, esencialmente, las de C++, con algunas mejorías en el manejo de los objetos y ayuda al programador, como el hecho de incorporar un recolector automático de basura que impide incurrir en un error por inexperiencia en el manejo directo de memoria. (4)

Dentro de las características que marcan la fortaleza del lenguaje y que muchas no las podemos encontrar en ningún otro por el momento se encuentran la portabilidad, una arquitectura neutral, un sistema distribuido, robusto, seguro, dinámico y multihilos; estas características vienen explicadas a continuación:

Portable

Debido a que Java no tiene especificaciones que dependan de un tipo de procesador en particular, no es necesario crear las versiones para cada uno de los sistemas operativos existentes. En Java, usted crea un programa y este mismo, sin ningún cambio, puede ser ejecutado en Unix, Windows y Macintosh, no necesita crear las versiones para cada uno de ellos.

Para que un programa sea portable no debe depender de una característica que solamente tenga un tipo de procesador o un conjunto reducido de ellos. Si bien es imposible que absolutamente todos los

procesadores interpreten todo, si es muy posible definir ciertos estándares que puedan entender la mayoría de ellos, y eso es lo que hace Java.

Arquitectura neutral

Para establecer Java como parte integral de la red, el compilador Java compila su código a un fichero objeto de formato independiente de la arquitectura de la máquina en que se ejecutará. Cualquier máquina que tenga el sistema de ejecución "*run-time*" puede ejecutar ese código objeto, sin importar en modo alguno la máquina en que ha sido generado. Actualmente existen sistemas "*run-time*" para Solaris 2.x, SunOs 4.1.x, Windows 95, Windows NT, Linux, Irix, Aix, Mac, Apple y probablemente haya grupos de desarrollo trabajando en potabilizarlo otras plataformas.

El código fuente Java se "compila" a un código de bytes de alto nivel independiente de la máquina. Este código (*byte-codes*) está diseñado para ejecutarse en una máquina hipotética que es implementada por un sistema "run-time", que sí es dependiente de la máquina.

Distribuido

Java tiene capacidades para interactuar con los protocolos ya establecidos del conjunto de protocolos TCP/IP, como http y ftp, lo cual permite el acceso a información que no está necesariamente en la misma zona geográfica. Pero la razón por la que se dice que Java es distribuido se debe a sus características de portabilidad, es decir, la ventaja de que una aplicación desarrollada en Java pueda ser ejecutada en diferentes plataformas implica que la carga de trabajo en un sistema puede distribuirse entre diferentes equipos de computo sin los problemas de incompatibilidad de estructuras de datos. Java puede utilizarse para aprovechar los recursos de sistemas de cómputo geográficamente separados, lo cual es la esencia del concepto de sistemas distribuidos.

Robusto

Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución. La comprobación de tipos en Java ayuda a detectar errores, lo antes posible, en el ciclo de desarrollo. Java obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error. Maneja la memoria para eliminar las preocupaciones por parte del programador de la liberación o corrupción de memoria. También implementa los "*arrays*" auténticos, en vez de listas enlazadas de punteros, con comprobación de límites, para evitar la posibilidad de sobrescribir o corromper memoria resultado de punteros que señalan a zonas equivocadas. Estas características reducen drásticamente el tiempo de desarrollo de aplicaciones en Java.

Además, para asegurar el funcionamiento de la aplicación, realiza una verificación de los "*byte-codes*", que son el resultado de la compilación de un programa Java. Es un código de máquina virtual que es interpretado por el intérprete Java. No es el código máquina directamente entendible por el *hardware*, pero ya ha pasado todas las fases del compilador: análisis de instrucciones, orden de operadores, etc., y ya tiene generada la pila de ejecución de órdenes.
Java proporciona, pues:

Comprobación de punteros
Comprobación de límites de *arrays*
Excepciones
Verificación de *byte-codes*

Seguro

La seguridad en Java tiene dos facetas. En el lenguaje, características como los punteros o el *casting* implícito que hacen los compiladores de C y C++ se eliminan para prevenir el acceso ilegal a la memoria. Cuando se usa Java para crear un navegador, se combinan las características del lenguaje con protecciones de sentido común aplicadas al propio navegador.

El lenguaje C, por ejemplo, tiene lagunas de seguridad importantes, como son los *errores de alineación*. Los programadores de C utilizan punteros en conjunción con operaciones aritméticas. Esto le permite al programador que un puntero referencie a un lugar conocido de la memoria y pueda sumar (o restar) algún valor, para referirse a otro lugar de la memoria. Si otros programadores conocen nuestras estructuras de datos pueden extraer información confidencial de nuestro sistema. Con un lenguaje como C, se pueden tomar números enteros aleatorios y convertirlos en punteros para luego acceder a la memoria. Otra laguna de seguridad u otro tipo de ataque, es el Caballo de Troya. Se presenta un programa como una utilidad, resultando tener una funcionalidad destructiva. Por ejemplo, en UNIX se visualiza el contenido de un directorio con el comando *ls*. Si un programador deja un comando destructivo bajo esta referencia, se puede correr el riesgo de ejecutar código malicioso, aunque el comando siga haciendo la funcionalidad que se le supone, después de lanzar su carga destructiva. Se notará un retardo, pero nada inusual.

El código Java pasa muchas pruebas antes de ejecutarse en una máquina. El código se pasa a través de un verificador de *byte-codes* que comprueba el formato de los fragmentos de código y aplica un probador de teoremas para detectar fragmentos de código ilegal -código que falsea punteros, viola derechos de acceso sobre objetos o intenta cambiar el tipo o clase de un objeto-.

Si los *byte-codes* pasan la verificación sin generar ningún mensaje de error, entonces sabemos que:

El código no produce desbordamiento de operandos en la pila
El tipo de los parámetros de todos los códigos de operación son conocidos y correctos
No ha ocurrido ninguna conversión ilegal de datos, tal como convertir enteros en punteros
El acceso a los campos de un objeto se sabe que es legal: *public*, *private*, *protected*
No hay ningún intento de violar las reglas de acceso y seguridad establecidas

El Cargador de Clases también ayuda a Java a mantener su seguridad, separando el espacio de nombres del sistema de ficheros locales, del de los recursos procedentes de la red. Esto limita cualquier aplicación del tipo Caballo de Troya, ya que las clases se buscan primero entre las locales y luego entre las procedentes del exterior.

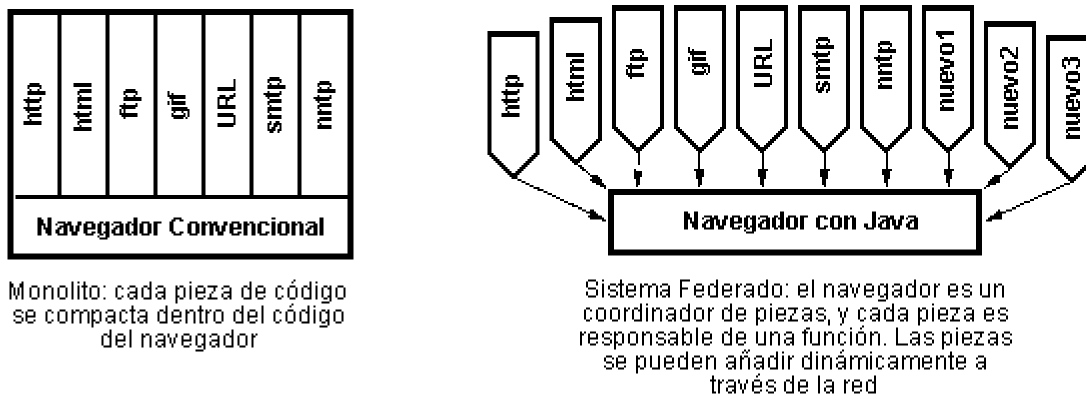
Las clases importadas de la red se almacenan en un espacio de nombres privado, asociado con el origen. Cuando una clase del espacio de nombres privado accede a otra clase, primero se busca en las clases predefinidas (del sistema local) y luego en el espacio de nombres de la clase que hace la referencia. Esto imposibilita que una clase suplante a una predefinida.

En resumen, las aplicaciones de Java resultan extremadamente seguras, ya que no acceden a zonas delicadas de memoria o del sistema, con lo cual evitan la interacción de ciertos virus. Java no posee una semántica específica para modificar la pila de programa, la memoria libre o utilizar objetos y métodos de un programa sin los privilegios del "*kernel*" del sistema operativo. Además, para evitar modificaciones por parte de los *crackers* de la red, implementa un método ultra seguro de autentificación por clave pública. El Cargador de Clases puede verificar una firma digital antes de realizar una instancia de un objeto. Por tanto, ningún objeto se crea y almacena en memoria, sin que se validen los privilegios de acceso. Es decir, la seguridad se integra en el momento de compilación, con el nivel de detalle y de privilegio que sea necesario.

Dada, pues la concepción del lenguaje y si todos los elementos se mantienen dentro del estándar marcado por Sun, no hay peligro. Java imposibilita, también, abrir ningún fichero de la máquina local (siempre que se realizan operaciones con archivos, éstas trabajan sobre el disco duro de la máquina de donde partió el *applet*), no permite ejecutar ninguna aplicación nativa de una plataforma e impide que se utilicen otros ordenadores como puente, es decir, nadie puede utilizar nuestra máquina para hacer peticiones o realizar operaciones con otra. Además, los intérpretes que incorporan los navegadores de la Web son aún más restrictivos. Bajo estas condiciones se puede considerar que Java es un lenguaje bastante seguro.
Respecto a la seguridad del código fuente, no ya del lenguaje, JDK proporciona un desensamblador de *byte-code*, que permite que cualquier programa pueda ser convertido a código fuente, lo que para el programador significa una vulnerabilidad total a su código. Utilizando java no se obtiene el código fuente original, pero sí desmonta el programa mostrando el algoritmo que se utiliza, que es lo realmente interesante. La protección de los programadores ante esto es utilizar llamadas a programas nativos, externos (incluso en C o C++) de forma que no sea descompilable todo el código; aunque así se pierda portabilidad. Esta es otra de las cuestiones que Java tiene pendientes.

Dinámico

Java se beneficia todo lo posible de la tecnología orientada a objetos. Java no intenta conectar todos los módulos que comprenden una aplicación hasta el tiempo de ejecución. Las librerías nuevas o actualizadas no paralizarán las aplicaciones actuales (siempre que mantengan el API anterior).



Java también simplifica el uso de protocolos nuevos o actualizados. Si su sistema ejecuta una aplicación Java sobre la red y encuentra una pieza de la aplicación que no sabe manejar, Java es capaz de traer automáticamente cualquiera de esas piezas que el sistema necesita para funcionar.

Java, para evitar que los módulos de "*byte-codes*" o los objetos o nuevas clases, haya que estar trayéndolos de la red cada vez que se necesiten, implementa las opciones de persistencia, para que no se eliminen cuando de limpie la caché de la máquina.

Interpretado

El intérprete Java (sistema run-time) puede ejecutar directamente el código objeto. Enlazar un programa, normalmente, consume menos recursos que compilarlo, por lo que los desarrolladores con Java pasarán más tiempo desarrollando y menos esperando por el ordenador. No obstante, el compilador actual del JDK es bastante lento. Por ahora, que todavía no hay compiladores específicos de Java para las diversas

plataformas, Java es más lento que otros lenguajes de programación, como C++, ya que debe ser interpretado y no ejecutado como sucede en cualquier programa tradicional.

Se dice que Java es de 10 a 30 veces más lento que C, y que tampoco existen en Java proyectos de gran envergadura como en otros lenguajes. La verdad es que ya hay comparaciones ventajosas entre Java y el resto de los lenguajes de programación, y una numerosa cantidad de folletos electrónicos que provocan fanatismo en favor y en contra de los distintos lenguajes contendientes con Java.

La verdad es que Java para conseguir ser un lenguaje independiente del sistema operativo y del procesador que incorpore la máquina utilizada, es tanto interpretado como compilado. Y esto no es ningún contrasentido, me explico, el código fuente escrito con cualquier editor se compila generando el *byte-code*. Este código intermedio es de muy bajo nivel, pero sin alcanzar las instrucciones máquina, propias de cada plataforma y no tiene nada que ver con el *p-code* de Visual Basic. El *byte-code* corresponde al 80% de las instrucciones de la aplicación. Ese mismo código es el que se puede ejecutar sobre cualquier plataforma. Para ello hace falta el run-time, que sí es completamente dependiente de la máquina y del sistema operativo, que interpreta dinámicamente el *byte-code* y añade el 20% de instrucciones que faltaban para su ejecución. Con este sistema es fácil crear aplicaciones multiplataforma, pero para ejecutarlas es necesario que exista el run-time correspondiente al sistema operativo utilizado.

Multihilos

Al ser "*multithreaded*" (multihilos) Java permite muchas actividades simultáneas en un programa. Los "*threads*" (a veces llamados, procesos ligeros o hilos de ejecución), son básicamente pequeños procesos o piezas independientes de un gran proceso. Al estar los "*threads*" construidos en el lenguaje, son más fáciles de usar y más robustos que sus homólogos en C o C++.

El beneficio de ser "multithreaded" consiste en un mejor rendimiento interactivo y mejor comportamiento en tiempo real. Aunque el comportamiento en tiempo real está limitado a las capacidades del sistema operativo subyacente (Unix, Windows, etc.), aún supera a los entornos de flujo único de programa (*single-threaded*) tanto en facilidad de desarrollo como en rendimiento.

Cualquiera que haya utilizado la tecnología de navegación concurrente, sabe lo frustrante que puede ser esperar por una gran imagen que se está trayendo. En Java, las imágenes se pueden ir trayendo en un "*thread*" independiente, permitiendo que el usuario pueda acceder a la información en la página sin tener que esperar por el navegador. (5)

Motores de Persistencia.

La mayoría de las aplicaciones de gestión constan de dos componentes principales que en conjunto colaboran para cumplir con la función determinada. Uno de estos componentes lo constituye la base de datos que es el componente encargado de mantener la información necesaria en un medio físico perdurable (discos duros). El otro componente lo constituye el programa que es el encargado de la manipulación y procesamiento de los datos y brindar los resultados deseados por el usuario. Para que estos dos componentes funcionen como una aplicación surge un verdadero problema ya que funcionan bajo modelos y lenguajes completamente diferentes. Por una parte la base de datos utiliza un modelo teórico llamado "relacional", que es el estándar que utilizan la mayoría de los gestores de BD. Este solo se ocupa de la parte estática de la aplicación (los datos) y no de la parte dinámica (los procesos); en el modelo relacional solo interesan las características de los objetos y no las acciones que estos puedan realizar. Estas características son lógicas en un modelo cuyo objetivo es modelar la información de la aplicación. Por otra parte se encuentra el modelo orientado a objetos que es el que utilizan la mayoría de los lenguajes de programación en la actualidad. La idea de este modelo es trasladar los objetos de la realidad a la programación por lo que cada objeto tiene características que lo diferencian de los demás y un grupo de acciones que son propias y específicas del mismo. Por su naturaleza los dos modelos tratan los datos de manera completamente diferente mientras el modelo relacional trata los datos en forma de registros el modelo orientado a objetos los trata en forma de objetos. Esto introduce una dificultad muy grande teniendo en cuenta que tenemos los datos en dos formatos distintos e incompatibles entre sí. En este punto en el nos encontramos es donde surge la necesidad de los motores de persistencia. La función del mismo es servir como intermediario; este consiste básicamente en un componente de *software* encargado de la capa de persistencia.

El programa sólo ve que puede guardar objetos y recuperar objetos, como si estuviera programado para una base de datos orientada a objetos. La base de datos sólo ve que guarda registros y recupera registros, como si el programa estuviera dirigiéndose a ella de forma relacional. Es decir, cada uno de los dos componentes trabaja con el formato de datos (el “idioma”) que le resulta más natural y es el motor de persistencia el que actúa de traductor entre los dos modelos, permitiendo que los dos componentes se comuniquen y trabajen conjuntamente.

Esta solución goza de las mejores ventajas de los dos modelos.

Por una parte, podemos programar con orientación a objetos, aprovechando las ventajas de flexibilidad, mantenimiento y reusabilidad.
Por otra parte, podemos usar una base de datos relacional, aprovechándonos de su madurez y su estandarización así como de las herramientas relacionales que hay para ella.

Una ventaja del motor de persistencia es que es el mismo para todas las aplicaciones. De esta forma sólo debe programarse una vez y puede usarse para todas las aplicaciones que se desarrollen en nuestra empresa. Sin embargo, un motor de persistencia es difícil de programar y de mantener, por lo que necesita un gran esfuerzo en costo y tiempo de desarrollo.

Es por ello que hay dos opciones a la hora de usar un motor de persistencia:

Programarlo dentro de nuestra empresa. Como se ha dicho, esto no es lo más recomendable, por la complejidad y costo que introduce esta opción.
Utilizar un motor que ya esté programado, comprándolo a un vendedor o bien usando un motor gratuito de código abierto.

Se recomienda fuertemente la segunda opción, que es la menos costosa y la menos propensa a fallos. Se debe escoger un motor de persistencia de los que están programados, estudiarlo y aplicarlo a todas las aplicaciones de una misma empresa. (6)

Framework de persistencia Hibernate.

El enfoque de la gestión de la persistencia de datos ha sido una decisión clave en el diseño de cada proyecto de *software*. Dado que la persistencia de datos no es nuevo o inusual requisito para las aplicaciones Java, lo que se esperaría es poder realizar una simple elección entre unas simples y bien establecidas soluciones de persistencia. Piense en *frameworks* de aplicaciones Web (Struts frente WebWork), *frameworks* de componentes GUI (Swing frente SWT), o los motores de plantilla (JSP frente Velocity). Cada una de las soluciones de la competencia tiene varias ventajas y desventajas, pero todos comparten el mismo alcance y enfoque global. Lamentablemente, esto no es aún el caso de las tecnologías de persistencia, donde podemos ver algunas soluciones tremendamente diferentes a los mismos problemas.

Durante varios años, la persistencia ha sido un tema candente de debate en la comunidad de Java. Muchos desarrolladores ni siquiera están de acuerdo sobre el alcance del problema. ¿Es la persistencia un problema que ya está resuelto por la tecnología relacional y extensiones, como procedimientos almacenados, o es más un problema generalizado que debe ser resuelto por los modelos de componente de Java, como EJB (*entity java beans*)? ¿Debemos programar el código manualmente, incluso las formas más primitivas CRUD (*create, read, update, delete*) las operaciones en *Structured Query Language* (SQL) y *Java Database Connectivity*(JDBC), o este trabajo debería ser automático? ¿Cómo lograr la portabilidad si cada sistema de gestión de bases de datos de SQL tiene su propio dialecto? ¿Debemos abandonar completamente el SQL y adoptar una tecnología de bases de datos diferentes, tales como sistemas de bases de datos orientadas a objetos? El debate continúa, pero una solución llamada object / relational mapping (ORM) ahora tiene una amplia aceptación.

Hibernate es un ambicioso proyecto que pretende ser una solución completa al problema de la gestión de persistencia de datos en Java. Este media en la interacción de la aplicación con una base de datos relacional, dejando libre a los desarrolladores para concentrarse en el negocio. Hibernate es una solución no intrusiva. Usted no está obligado a seguir todas las normas específicas de Hibernate y los patrones de diseño al escribir su lógica de negocio ni las clases de persistencia, de modo que Hibernate se integra sin

problemas con la mayoría de las nuevas aplicaciones existentes y no requiere cambios perturbadores para el resto de la aplicación.

Hibernate es un motor de persistencia de código abierto. Permite mapear un modelo de clases a un modelo relacional sin imponer ningún tipo de restricción en ambos diseños. Cuenta con una amplia documentación, tanto a nivel de libros publicados como disponibles gratuitamente en su Web. A nivel comercial está respaldado por JBoss, que proporciona servicios de soporte, consultoría y formación en el mismo.

Actualmente es el rey indiscutible de la persistencia. Desde su versión 1.0, el motor no ha parado de evolucionar, incorporando todas las nuevas ideas que se iban incorporando en este campo.

Hoy en día, desde en su versión 3.2, ya soporta el estándar EJB 3, por lo que ya se puede elegir desarrollar aplicaciones empleando los mismos (que correrán en cualquier contenedor de EJB's que soporte J2EE 5). Esto no solo asegura nuestra inversión de tiempo en Hibernate de cara al futuro, sino que, de ser útil, nos hace totalmente independientes del mismo. (7)

Para la realización de las query en Hibernate se utiliza el HQL (*Hibernate Query Language*) que no es más que un equivalente al SQL tradicional con la particularidad de que el mismo es orientado a objetos. Este no exige mucho para un programador que ya este acostumbrado al SQL ya que su sintaxis mantiene muchas de las características básicas de su predecesor y en algunos casos se hace hasta más sesillo y potente a la vez por el hecho de estar orientado a objetos. El mismo surge por la necesidad de un lenguaje que sirva como pantalla a las disímiles versiones de SQL existente tantas casi como gestores de bases de datos.

El HQL se estructura de una forma muy similar a su contraparte el SQL con las clausuras "*SELECT*", "*FROM*", y "*WHERE*" este incluso tiene soporte para subselecciones, "*inner join*", "*outer join*" y otras muchas funciones como "*avg()","sum()*"."*min()*","*count()*" entre otras incluso operadores como "*distinct*", "*like*", "*group by*" y "*order by*". En el podemos encontrar ventajas como los "*named param*" que no son más que otra herramienta dentro del código que nos permite especificar los nombres directamente de los

parámetros en lugar de marcas en formas de símbolos de interrogación lo que evita confusiones y nos da más libertad a la hora de elegir el orden de los mismos. (8)

A continuación se explicara que significan la mayoría de los elementos que se encuentran configurados en un mapping (mapeo) común de Hibernate.

Tomemos por ejemplo el mapping de NRol

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="cu.uci.ccv.common.vo.NRol" table="nRol" lazy="false">
        <id name="idRol" column="idRol" type="integer">
            <generator class="increment">
            </generator>
        </id>
        <property name="nombre" column="nombre" type="string" length="30"
          not-null="true" lazy="false"/>
        <property name="descripcion" column="descripcion" type="string" length="255"
          not-null="false" lazy="false"/>
        <many-to-one name="dParteidParte" cascade="lock" column="dParteidParte"
          class="cu.uci.ccv.common.vo.DParte" not-null="true" lazy="proxy" access="field">
        </many-to-one>
        <set name="Usuario" table="dUsuario_Rol"   lazy="true" >
            <key column="rol" not-null="true"/>
            <many-to-many column="usuario" class="cu.uci.ccv.common.vo.DUsuario">
            </many-to-many>
        </set>
        <set name="NTipo_NivelidTipoNivel" table="dTipo_Nivel_Rol" lazy="true" >
            <key column="nRolidRol" not-null="true"/>
            <many-to-many column="nTipo_NivelidTipoNivel" class="cu.uci.ccv.common.vo.NTipo_Nivel">
            </many-to-many>
        </set>
        <set name="Funcionalidad" table="dRol_Funcionalidad" lazy="true" >
            <key column="rol" not-null="true"/>
            <many-to-many column="funcionalidad" class="cu.uci.ccv.common.vo.NFuncionalidad">
            </many-to-many>
        </set>
    </class>
</hibernate-mapping>
```

Si se comienza a desglosar en pequeñas partes se pueden identificar primeramente los siguientes tag:

```
<?xml version="1.0" encoding="utf-8" ?>
```

1. Este es un encabezado típico de todos los documentos *Extensible Markup Language* (XML) y que no es objetivo explicarlo en este documento.

```
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

El encabezado DOCTYPE es donde se especifica el DTD "*document type declaration*" lo cual es utilizado por el *xml parser* para validar el documento contra la configuraciones de Hibernate.
Luego encontraremos el cuerpo del mapping el cual se puede identificar por estar dentro de los tag <hibernate-mapping> </hibernate-mapping> y que significa que todo el contenido encerrado dentro de ellos es un mapeo de Hibernate.

```
<class name="cu.uci.ccv.common.vo.NRol" table="nRol" lazy="false">
    <id name="idRol" column="idRol" type="integer">
        <generator class="increment">
        </generator>
    </id>
</class>
```

En este ejemplo podemos identificar varias partes primero los tag <class> </class> los cuales especifican que el contenido y propiedades de la clase deben se escritos dentro de estos. Entre los mismo podemos encontrar varias propiedades algunas de las cuales se pueden ver en el ejemplo y otras se pueden usar en dependencia de lo que se quiera o cuan especifico se necesite ser en las mismas

name: Especifica el nombre de la clase persistente.
table: Especifica el nombre de la tabla equivalente en la base de datos que se desea mapear
shema: Especifica el nombre del esquema de la base de datos donde se encuentra la tabla que se nombro anteriormente
lazy: Nos especifica si esta tabla será tratada por el Hibernate de forma perezosa o no, en otras palabras que si cuando se consulte la misma esta traerá su contenido o no.

La otra parte importante que podemos identificar consiste en los tag <id> </id> entre los que se especifica cuál de los atributos corresponde a la llave de la tabla en la clase, este también contiene propiedades de las cuales se pueden mencionar.

name: Especifica el nombre del atributo en la clase
column: Especifica el nombre de la columna en la tabla
type: Especifica el tipo de datos Java del atributo.
unsaved-value: Le especifican a Hibernate cuando un objeto nunca ha sido guardado en la base de datos.

Luego contiene unos tag (<generator> </generator>) internos que especifican el método que utilizara Hibernate para generar el valor de la llave cada vez que este salve un objeto nuevo en la base de datos el cual contiene la propiedad

class: Especifica el método que será utilizado el cual puede tomar los siguientes valores(*native*, *identity*, *sequence*, *hilo*, *assigned* y increment entre otros más que no interesan por el momento) en dependencia del que se esté utilizando puede conllevar la inclusión de algunas nuevas propiedades.
El método *native* es bastante interesante. Deja que Hibernate escoja entre los métodos *identity*, *sequence* o hilo, en función de las características del gestor de bases de datos con el que trabajemos.
El método *assigned* deja que sea nuestra aplicación la que asigne un identificador al objeto antes de guardarlo en la base de datos.
El método increment genera identificadores de tipo long, short o int que son únicos sólo cuando no hay otro proceso que esté insertando datos en la misma tabla.

```
<property name="nombre" column="nombre" type="string" length="30"
  not-null="true" lazy="false"/>
```

Estos tag <property > </property> contienen la configuración de los atributos normales de la tabla y en ellos podemos identificar comúnmente las propiedades.

name: Especifica el nombre del atributo de la clase
column: Especifica el nombre de la columna en la tabla de la base de datos.

type: Especifica el tipo de datos del atributo.
length: Este se utiliza en el caso de los valores String y sirve para especificar el tamaño del campo en la base de datos.
not-null: Especifica si el campo de la base de datos admite valores nulos.
lazy: Nos especifica si este atributo será tratado por el Hibernate de forma perezosa o no, en otras palabras que si cuando se consulte la tabla que lo contiene en la base de datos esta traerá su contenido o no.

En cualquier diseño relacional las tablas se hacen referencias entre ellas a través de relaciones entre estas podemos encontrar: uno a uno, uno a muchos, muchos a uno y muchos a muchos. De estas comenzaremos por los casos uno a uno y muchos a uno que en Hibernate se representan como un atributo de la clase cuyo tipo de datos es la otra clase de objetos con la que se encuentra relacionado. En el otro caso este atributo se convierte en una colección de objetos de la otra clase con la que se encuentra relacionado y se explicara más adelante.

```
<one-to-one name="nivel" class="cu.uci.ccv.common.vo.DNivel" cascade="all" constrained="true"/>
```

<one-to-one> Asociación entre dos clases persistentes, en la cual no es necesaria otra columna extra. Los OID's de las dos clases serán idénticos.

name : Especifica el nombre del atributo de la clase.
class : Especifica la clase persistente del objeto asociado.
cascade ("all|none|save-update|delete"): Permite configurar las operaciones en cascada a partir de la asociación.
constrained ("true"|"false") : Especifica que la clave foránea que es la clave principal de la tabla mapeada tiene la restricción de referenciar la tabla de la clase asociada. Esta opción afecta el orden en el que el “save” y el “delete” realizan la cascada.

```
<many-to-one name="coordinador" cascade="none" column="coordinador"
class="cu.uci.ccv.common.vo.DPersona" not-null="true" lazy="proxy" access="field">
</many-to-one>
```

<many-to-one> La relación muchos a uno necesita en la tabla un identificador de referencia, el ejemplo clásico es la relación entre padre - hijos. Un hijo necesita un identificador en su tabla para indicar cuál es su padre. Pero en objetos en realidad no es un identificador si no el propio objeto padre, por lo tanto el componente muchos a uno es en realidad el propio objeto padre y no simplemente su identificador. (Aunque en la tabla se guarde el identificador).

name : Especifica el nombre del atributo de la clase.
column : Especifica la columna de la tabla donde se guardara el identificador del objeto asociado.
class: Especifica el nombre de la clase asociada. Hay que escribir todo el paquete.
cascade ("all|none|save-update|delete"): Especifica que operaciones se realizaran en cascada desde el objeto padre.
not-null: Especifica si el campo de la base de datos admite valores nulos.
lazy: Nos especifica si este atributo será tratado por el Hibernate de forma perezosa o no, en otras palabras que si cuando se consulte la tabla que lo contiene en la base de datos esta traerá su contenido o no.
access: Especifica la estrategia que seguirá el Hibernate para acceder a la propiedad.

Pasando a los otros dos tipos de relaciones entre las clases persistentes, las uno a muchos y la muchos a muchos. Las colecciones son declaradas utilizando <set>, <list>, <map>, <bag>, <array> y <primitive-array>. Los posibles parámetros y sus valores son.

name: Especifica el nombre de la colección en la clase.
table: Especifica el nombre de la tabla de la colección que el utiliza en el caso del muchos a muchos.
lazy ("true"|"false"): Nos especifica si este atributo será tratado por el Hibernate de forma perezosa o no, en otras palabras que si cuando se consulte la tabla que lo contiene en la base de datos esta traerá su contenido o no.
inverse: Señala esta colección como el fin de una asociación bidireccional. Utilizada en relaciones muchos a muchos sobre todo.
cascade: Permite las operaciones en cascada hacia los entidades hijas.

```xml
<set name="DUsuario" lazy="true" cascade="save-update,lock" inverse="true">
    <key column="persona" not-null="true"/>
    <one-to-many class="cu.uci.ccv.common.vo.DUsuario"/>
</set>
<set name="Usuario" table="dUsuario_Rol"   lazy="true" >
    <key column="rol" not-null="true"/>
    <many-to-many column="usuario" class="cu.uci.ccv.common.vo.DUsuario">
    </many-to-many>
</set>
```

Entre los tag <key> </key> se especifica otras propiedades como

column: Se especifica el nombre de la columna en la base de datos que contiene la llave.

not-null: Especifica si esta propiedad puede contener valores nulos.

Dentro de los tag <one-to-many></one-to-many> y <many-to-many></many-to-many> se encuentran las siguientes propiedades:

class: Especifica la clase de la que se construirá la colección.

column: En esta se especifica el nombre de la columna en la base de datos (de la tabla que contiene la relación muchos a muchos) que contiene la relación con la otra tabla de la que se quiere hacer la colección. Este solo se encuentra en el caso de las relaciones muchos a muchos.

inverse: Especifica cuál de los dos lados de la relación permitiéndole saber a Hibernate cuál de los dos lados no debe actualizar la llave foránea. Este solo se encuentra en el caso de la relación uno a muchos.

A continuación se explicará la forma estándar con la que se generan las clases persistentes y la forma en que vienen representadas las relaciones de cada tipo y la manera en que se representan las columnas de la base de datos.

```
public class NRol implements Serializable{

    private int idRol;

    private String nombre;

    private String descripcion;

    private cu.uci.ccv.common.vo.DParte dParteidParte;

    private java.util.Set usuario;

    private java.util.Set nTipo_NivelidTipoNivel = new java.util.HashSet();

    private java.util.Set funcionalidad = new java.util.HashSet();
```

En esta clase se pueden identificar atributos cuyos tipos son primitivos de Java estos son los casos de la forma en que se representan las columnas de la base de datos cuyo tipo de dato Java es el equivalente al del tipo de SQL (ejemplo varchar-String, integer-int). Además se pueden identificar atributos que son entidades de otras clases, estos representan las relaciones que se explicaron anteriormente en la confección de los *mappings* de tipo muchos a uno y uno a uno. El otro tipo de atributos que podemos encontrar son los que su tipo de datos son colecciones de otra entidad los cuales representan las relaciones del tipo uno a muchos y muchos a muchos explicadas también anteriormente. (9)

Integración a Spring de Hibernate.

Spring

Spring es un conjunto de librerías de entre las que podemos escoger aquellas que faciliten el desarrollo de nuestra aplicación. Entre sus posibilidades más potentes está su contenedor de Inversión de Control (Inversión de Control, también llamado Inyección de Dependencias, es una técnica alternativa a las clásicas búsquedas de recursos vía *Java Naming and Directory Interface* (JNDI). Permite configurar las clases en un archivo XML y definir en él las dependencias. De esta forma la aplicación se vuelve muy modular y a la vez no adquiere dependencias con Spring), la introducción de aspectos, plantillas de utilidades para Hibernate etc.

Es uno de los proyectos más sorprendentes en el panorama actual en Java en el grado en que ayuda a que los diferentes componentes que forman una aplicación trabajen entre sí, pero no establece apenas dependencias consigo mismo. Esta es la primera característica de este *framework*. Sería posible retirarlo sin prácticamente cambiar líneas de código. Lo único que sería necesario es, lógicamente, añadir la funcionalidad que provee, ya sea con otro *framework* similar o mediante nuestro código.

A nivel de soporte de la comunidad, Spring es uno de los proyectos con más actividad, con desarrollos dentro y fuera del propio *framework* (de hecho Acegi, el sistema de seguridad empleado, fue realizado por colaboradores ajenos al *framework*). Actualmente dispone de soporte comercial a través de Interface21, la empresa creadora, así como otros fabricantes que dan soporte en su área. (10)

La utilización del *framework* Hibernate a través de Spring nos facilita más aun el trabajo ya que este libera en gran medida al programador de mantener el control sobre las sesiones. Una de las principales clases que nos brinda el *framework* para su integración es "*HibernateTemplate*" la cual nos brinda un gran número de métodos entre los que mencionaremos los principalmente los que más se utilizaron:

Método	Descripción
find(String queryString)	Devuelve el resultado de la query.
find(String queryString,Object value)	Devuelve el resultado de la query y pasa por parámetro el objeto que se le pase por valor.
find(String queryString,Object[] values)	Devuelve el resultado de la query y pasa por parámetros el arreglo de objetos que se le pasen por valor.
findByNamedParam(String queryString, String paramName,Object value)	Devuelve el resultado de la query y pasa por parámetro el objeto que se le pase por valor en la posición donde se encuentra el nombre del parámetro.
findByNamedParam(String queryString,	Devuelve el resultado de la query y pasa

String[] paramNames,Object[] values)	por parámetros el arreglo de objetos que se le pasen por valor en las posiciones donde se encuentran los nombres de los parámetros.
findByNamedQuery(String queryName)	Devuelve el resultado de la query pero en este caso el nombre pertenece a una query que se encuentra ubicada en los mapping.
findByNamedQuery(String queryName,Object value)	Devuelve el resultado de la query y pasa por parámetro el objeto que se le pase por valor pero en este caso el nombre pertenece a una query que se encuentra ubicada en los mapping.
findByNamedQuery(String queryName, Object[] values)	Devuelve el resultado de la query y pasa por parámetros el arreglo de objetos que se le pasen por valor pero en este caso el nombre pertenece a una query que se encuentra ubicada en los mapping.
findByNamedQueryAndNamedParam(String queryName, String paramName,Object value)	Devuelve el resultado de la query y pasa por parámetro el objeto que se le pase por valor en la posición donde se encuentra el nombre del parámetro pero en este caso el nombre pertenece a una query que se encuentra ubicada en los mapping.
findByNamedQueryAndNamedParam(String queryName,String[] paramNames,Object[] values)	Devuelve el resultado de la query y pasa por parámetros el arreglo de objetos que se le pasen por valor en las posiciones donde se encuentran los nombres de los parámetros pero en este caso el nombre

	pertenece a una query que se encuentra ubicada en los mapping.
delete(Object entity)	Elimina la entidad pasada por valor de la base de datos.
save(Object entity)	Salva la entidad pasada por valor en la base de datos.
update(Object entity)	Modifica la entidad pasada por valor en la base de datos.
saveOrUpdate(Object entity)	Salva o modifica dependiendo de la entidad que se pase por valor si existe ya o no en la base de datos.
load(Object entity,Serializable id)	Carga un objeto de la entidad pasada por valor de la base de datos cuyo id coincida con el indicado.
get(Class entityClass,Serializable id)	Carga un objeto de la clase de la entidad pasada por valor de la base de datos cuyo id coincida con el indicado.
bulkUpdate(String queryString)	Realiza una modificación en la base de datos según la query.
bulkUpdate(String queryString,Object value)	Realiza una modificación en la base de datos según la query y pasa por parámetro el objeto que se le pase por valor.
bulkUpdate(String queryString,Object[] values)	Realiza una modificación en la base de datos según la query y pasa por parámetros el arreglo de objetos que se le pasen por valor.

(11)

Estándares de codificación

Los estándares de codificación son de una gran importancia en los proyectos sobre todo para los desarrolladores. Los mismos facilitan en gran medida la comprensión del código escrito lo que hace más sencillo la revisión del código y permiten su posterior mantenimiento con un menor costo. Es de vital importancia para la correcta integración entre los diferentes componentes de la aplicación además de que muchos de los *framework* a los que uno normalmente se integra presuponen la correcta utilización de los mismos.

El estándar utilizado para el nombramiento de los paquetes clases etc. en el proyecto fue el que se brinda a continuación y esta vasado en el que ofrece para Java la empresa Sun Microsystems (Hommel).

Tipos de identificadores	Reglas para nombrar	Ejemplos
Paquetes	El prefijo del nombre de un paquete se escribe siempre con letras ASCII en minúsculas, y debe ser uno de los nombres de dominio de alto nivel, actualmente com, edu, gov, mil, net, org, o uno de los códigos ingleses de dos letras que identifican cada país como se especifica en el ISO Standard 3166, 1981. Los subsecuentes componentes del nombre del paquete variarán de acuerdo a las convenciones de nombres internas de cada organización. Dichas convenciones pueden especificar que algunos nombres de los directorios correspondan a divisiones, departamentos, proyectos o máquinas.	cu.uci.ccv.administracion
Clases	Los nombres de las clases deben ser sustantivos, cuando son compuestos tendrán la primera letra de cada palabra que lo forma en mayúsculas. Intentar mantener los nombres de las clases simples y descriptivas. Usar palabras completas, evitar acrónimos y abreviaturas (a no ser que la abreviatura sea mucho más conocida que el nombre completo, como URL o HTML).	class DUsuario class GestionarUsuarioDaoImpl
Interfaces	Los nombres de las interfaces siguen la misma regla que las clases.	Interface GestionarUsuarioDao
Métodos	Los métodos deben ser verbos, cuando son compuestos tendrán la primera letra en minúscula, y la primera letra de las siguientes	buscarUsuarios modificarRol

	palabras que lo forma en mayúscula.	
Variables	Excepto las constantes, todas las instancias y variables de clase o método empezarán con minúscula. Las palabras internas que lo forman (si son compuestas) empiezan con su primera letra en mayúsculas. Los nombres de variables no deben empezar con los caracteres subguión "_" o signo del dólar "$", aunque ambos están permitidos por el lenguaje. Los nombres de las variables deben ser cortos pero con significado. La elección del nombre de una variable debe ser un mnemónico, designado para indicar a un observador casual su función. Los nombres de variables de un solo carácter se deben evitar, excepto para variables índices temporales. Nombres comunes para variables temporales son i, j, k, m, y n para enteros; c, d, y e para caracteres.	String query List<NRol> rol
Constantes	Los nombres de las variables declaradas como constantes deben ir totalmente en mayúsculas separando las palabras con un subguión ("_"). (Las constantes ANSI se deben evitar, para facilitar su depuración.)	En este caso no se utilizo este recurso en el proyecto. static final int pi = 3.141593

En nuestro proyecto además se tomaron otros acuerdos como terminar todos los nombres de las interfaces de la capa de persistencia con la palabra “Dao” y en el caso de las clases que la implementaban adicionarles además la palabra “Impl”. Además en el caso de los “*value object*” también se tomo como acuerdo comenzar su nombre con la letra “N” en caso de que este fuera un nomenclador y “D” en el resto.

Conclusiones del Capítulo.

Se realizó un análisis independiente de cada uno de los puntos fundamentales que se plantearon como objetivos. Gracias a esto se comprendió la necesidad de uso de la metodología (RUP en este caso) y los aportes y ventajas que esta brindaba al proceso de desarrollo. Por otra parte se analizaron las técnicas de programación fundamentalmente la Programación Orientada a Objetos como la solución viable para dar solución a la complejidad del sistema específicamente el lenguaje Java. Además un estudio profundo en el cual se buscaron las principales características del *framework* Hibernate específicamente su integración con Sprint nos permitió conocer todas las ventajas de su uso para la implementación del proyecto. Todo lo anterior nos permitió modelar el marco teórico y el modelo conceptual sobre el cual se fundamentó la investigación.

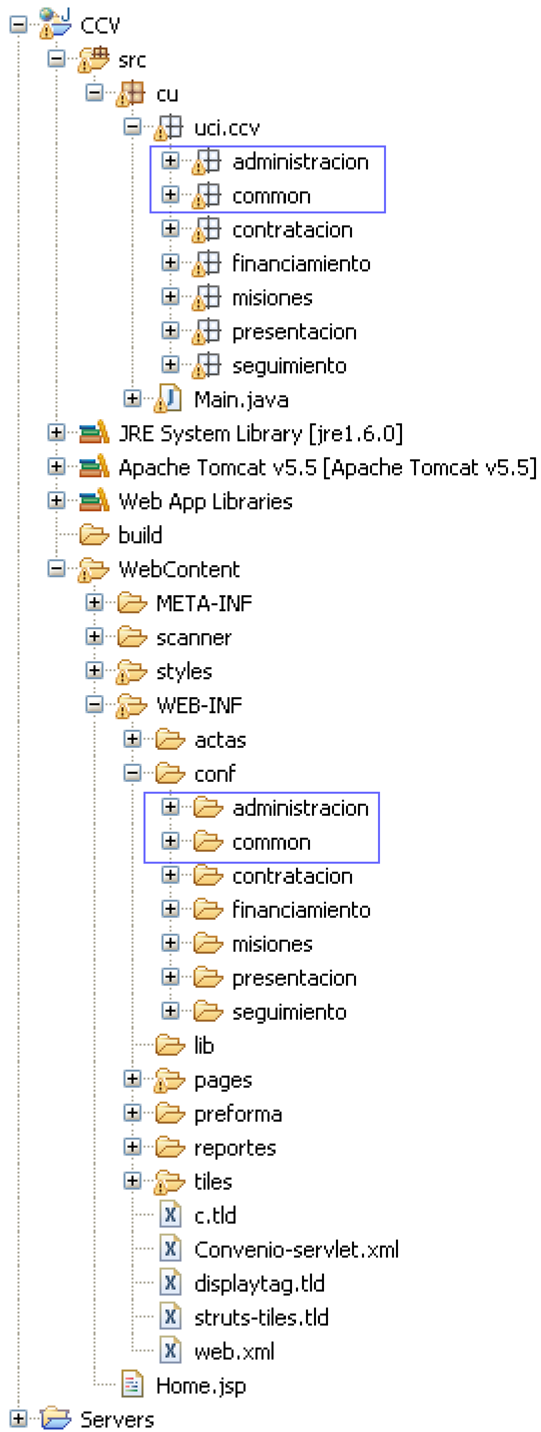
CAPÍTULO 2: IMPLEMENTACIÓN

Introducción al capítulo.

En el contenido de este capítulo se realizará un análisis con el objetivo de describir la solución dada al problema planteado. Para esto se irán describiendo primeramente las configuraciones y las localizaciones de los componentes dentro del espacio de trabajo, luego el mapeo que se realizó de las clases persistentes y la implementación de los DAOs las cuales partieron tanto del diseño de clases así como del modelo de diseño de la base de datos propuestos para los módulos de administración y configuración. También se realizará breve descripción de cada uno de los métodos que se implementaron y se abordarán algunos ejemplos representativos.

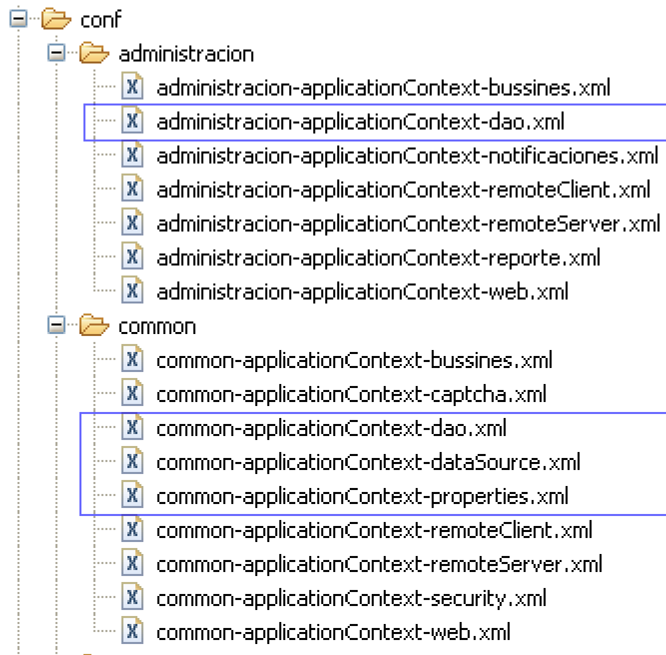
Estructura distribución y componentes de la capa de persistencia.

La capa de acceso a datos o capa de persistencia como también se le suele llamar es el grupo de interfaces, clases y componentes (XML, hbm) de configuración que le permiten manipular los objetos de persistencia de la aplicación; entiéndase por manipularlos principalmente guardarlos, eliminarlos, modificarlos y obtenerlos según sea al caso que se esté manejando. Además es esta capa la que vincula directamente a la aplicación con la base de datos y es la base de todas las demás capas superiores. En este caso se centrará la explicación sobre los módulos de administración y configuración:
Los componentes y clases de dichos módulos se encuentran dentro de la aplicación en los siguientes paquetes y carpetas:


CCV
src
cu
uci.ccv
administracion
common
contratacion
financiamiento
misiones
presentacion
seguimiento
Main.java
JRE System Library [jre1.6.0]
Apache Tomcat v5.5 [Apache Tomcat v5.5]
Web App Libraries
build
WebContent
META-INF
scanner
styles
WEB-INF
actas
conf
administracion
common
contratacion
financiamiento
misiones
presentacion
seguimiento
lib
pages
preforma
reportes
tiles
c.tld
Convenio-servlet.xml
displaytag.tld
struts-tiles.tld
web.xml
Home.jsp
Servers

El módulo se encuentra distribuido una parte en el paquete “administracion” que es donde se concentra el grueso del mismo y otra en “common” que es donde se encuentran algunas partes que por su carácter común para toda la aplicación se encuentran en este. Dentro de cada paquete se encuentran varios subpaquetes donde se ubican las interfaces y implementaciones de cada capa por separado y algunos componentes especiales a los cuales se les da tratamiento por separado de estos se prestará principal atención a los paquetes “dao” donde se encuentran las interfaces e implementaciones de la capa de acceso a datos y “vo” donde se encuentran ubicados los “*value objects*” o también conocidos como pojos que son los objetos persistentes y los .hbm que son los que contienen los mapeos de los objetos persistentes contra las tablas de la base de datos y además en los que se guardan las consultas en “hql” que se utilizan más tarde desde la implementación de los DAOs.

Otra parte muy importante del módulo se encuentra dentro de las carpetas “administracion” y “common” dentro de configuración en el “WEB-INF” del WebContent; en estas carpetas se encuentran los XMLs de configuración del módulo y de partes generales de la aplicación de las cuales depende el módulo para funcionar en estos se pueden encontrar principalmente los “*beans*”, siendo importantes para la capa de acceso a datos los siguientes:



Mapeo de las clases de persistencia, configuración del *framework*

La generación de los "*value object*" entiéndase los "*hbm*" con el mapeo y las clases persistentes se género utilizando para ello la herramienta Visual Paradin la cual te permite generar directamente a partir del modelo de diseño de las clases persistentes el cual dicho sea de paso se modeló en la misma herramienta, el código fuente de los ORM , el modelo de datos y del mismo generar la base de datos directamente, lo cual garantiza en cierto modo una perfecta correspondencia entre los mismos y a la vez mantener sincronizados estos ya que si ocurre algún cambio, primero se realiza en el modelo de diseño de las clases de persistencia y a partir de este se actualizan los demás esto ahora una buena parte del trabajo al programador el cual se puede concentrar más en la configuración de la navegabilidad, visibilidad etc. dentro de los mapping lo que hace viable el uso de los ORM ya que si toda esta configuración se hiciera manualmente se vería un poco discutible el tema del ahorro del código a la hora de la implementación.

Configuraciones.

Existen varios pasos que son inviolables a la hora de realizar un proyecto teniendo una arquitectura como la que se utilizó para el desarrollo del proyecto que consiste en la integración del *framework* Hibernate al *framework* Spring. Estos pasos consisten en la configuración de los mismos y a su vez la configuración de nuestras clases para que utilicen las ventajas de estos las cuales se hace mediante ficheros XML.
A continuación se explicarán en qué consisten estas configuraciones y se nombrarán uno a uno los ficheros XML que las contienen
Comenzaremos por el XML (common-applicationContext-properties.xml) en el cual se encuentra el siguiente "*bean*".

```
<bean id="propertyConfigurer"
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <list>
            <value>
                classpath:cu/uci/ccv/common/resources/database_connection.properties
            </value>
            <value>
                classpath:cu/uci/ccv/common/resources/ccvmessages.properties
            </value>
        </list>
    </property>
</bean>
```

En el cual se le inyecta la propiedad "*locations*" a la clase "*PropertyPlaceholderConfigurer*" la cual es una implementación muy útil de la interface "*BeanFactoryPostProcessor*" la cual nos permite cargar las propiedades de uno o más ficheros externos de propiedades (.properties) y utilizar variables dentro de los *beans*. Los dos valores que se ven en este son las direcciones donde se encuentran los ficheros ".properties" donde se encuentra algunos valores de configuración de la aplicación y que se explicará a continuación.

En el fichero (database_connection.properties) que es el de especial interés para nosotros de todos los ".properties" que existen en la aplicación se encuentra el siguiente contenido

```
hibernate.connection.url=jdbc:postgresql://10.7.13.111/ccv
hibernate.connection.driver_class=org.postgresql.Driver
hibernate.connection.username=prueba
hibernate.connection.password=prueba
hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
hibernate.cglib.use_reflection_optimizer=false
hibernate.show_sql=false

# La maxima cantidad inicial de conexiones que son creadas cuando el Pool es creado.
dbcp.initialSize=2
#La maxima cantidad de conexiones activas que el pool va a contener al mismo tiempo.
dbcp.maxActive=2
#La maxima cantidad de conexiones activas que permaneceran desocupadas en el pool.
dbcp.maxIdle=1
#La minima cantidad de conexiones activas que permaneceran desocupadas en el pool.
dbcp.minIdle=1
#El tiempo en milisegundos que el pool espera para lanzar una excepcion cuando no hay conexiones disponibles
dbcp.maxWait=-1
```

En este, como se explican en los comentarios (texto en verde iniciado por el símbolo #) se encuentra los valores que se le asignaran a varias de las propiedades del pool de conexiones de Hibernate además de los valores de la configuración de conexión necesarias para que la aplicación se pueda conectar a la base de datos.

En el XML (common-applicationContext-dataSource.xml) se encuentran los siguientes *beans*:

```
<bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="url">
        <value>${hibernate.connection.url}</value>
    </property>
    <property name="driverClassName">
        <value>${hibernate.connection.driver_class}</value>
    </property>
    <property name="username">
        <value>${hibernate.connection.username}</value>
    </property>
    <property name="password">
        <value>${hibernate.connection.password}</value>
    </property>
    <property name="initialSize">
        <value>${dbcp.initialSize}</value>
    </property>
    <property name="maxActive">
        <value>${dbcp.maxActive}</value>
    </property>
    <property name="maxIdle">
        <value>${dbcp.maxIdle}</value>
    </property>
    <property name="minIdle">
        <value>${dbcp.minIdle}</value>
    </property>
    <property name="maxWait">
        <value>${dbcp.maxWait}</value>
    </property>
</bean>
```

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean"
    lazy-init="true">
    <property name="dataSource">
        <ref bean="dataSource" />
    </property>

    <property name="mappingDirectoryLocations">
        <list>
            <value>classpath:/cu/uci/ccv/common/vo</value>
            <value>classpath:/cu/uci/ccv/presentacion/vo</value>
            <value>classpath:/cu/uci/ccv/contratacion/vo</value>
            <value>classpath:/cu/uci/ccv/administracion/vo</value>
            <value>classpath:/cu/uci/ccv/seguimiento/vo</value>
            <value>classpath:/cu/uci/ccv/misiones/vo</value>
            <value>classpath:/cu/uci/ccv/financiamiento/vo</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">
                ${hibernate.dialect}
            </prop>
            <prop key="hibernate.cglib.use_reflection_optimizer">
                ${hibernate.cglib.use_reflection_optimizer}
            </prop>
            <prop key="show_sql">${hibernate.show_sql}</prop>
        </props>
    </property>
</bean>
```

En el primer "*bean*" se inyecta un "*BasicDataSource*" con todas las propiedades iniciadas según las variables que se cargan del ".propertie" las que contiene todas las configuraciones de Hibernate y de conexión a la base de datos. El segundo "bean" consiste en la inyección de la clase "*LocalSessionFactoryBean*" la cuan consiste en un "*factory*" de "*beans*" que produce una instancia de un "*seccionFactory*" de Hibernate el cual recopila todos los mapping que se les pasan. Para esto entre otras propiedades principalmente lo que necesita es que se le inyecte un "*datasource*" y una lista con los "*classpath*" de los paquetes que contienen los "*value object*".

Partiendo de un buen diseño de las clases persistente no se tendrá ningún problema a la hora de generar los pojos con el "Visual Paradin" el mismo tiene la facilidad de a la hora de generarlos incluso ser específico en la configuración del modo en que estarán configurados los "*lazy*" la forma en que se implementarán las referencias a los objetos de las clases y si se desea que los objetos implementen la interface "*Serializable*" entre otras.

A continuación una relación de todas las clases persistentes con sus respectivos archivos de mapeo.

Clase Persistente	XML de mapeo (.hbm)
DEnte.java	DEnte.hbm.xml
DMinisterio.java	DMinisterio.hbm.xml
DNivel.java	DNivel.hbm.xml
DParte.java	DParte.hbm.xml
DPersona.java	DPersona.hbm.xml
DUsuario.java	DUsuario.hbm.xml
NFuncionalidad.java	NFuncionalidad.hbm.xml
NRol.java	NRol.hbm.xml
NSector.java	NSector.hbm.xml
NTipo_Nivel.java	NTipo_Nivel.hbm.xml
DTrazas.java	DTrazas.hbm.xml
DNotificacion.java	DNotificacion.hbm.xml
DVisibilidadNotificacion.java	DVisibilidadNotificacion.hbm.xml
DSecretaria.java	DSecretaria.hbm.xml
NConfiguracion_Viaticos.java	NConfiguracion_Viaticos.hbm.xml
NTasa_Cambio.java	NTasa_Cambio.hbm.xml
NCategoria.java	NCategoria.hbm.xml
NFuente_Financiamiento.java	NFuente_Financiamiento.hbm.xml
NTipo_Recurso.java	NTipo_Recurso.hbm.xml
NModalidad.java	NModalidad.hbm.xml

Modelo de diseño y Modelo de datos.

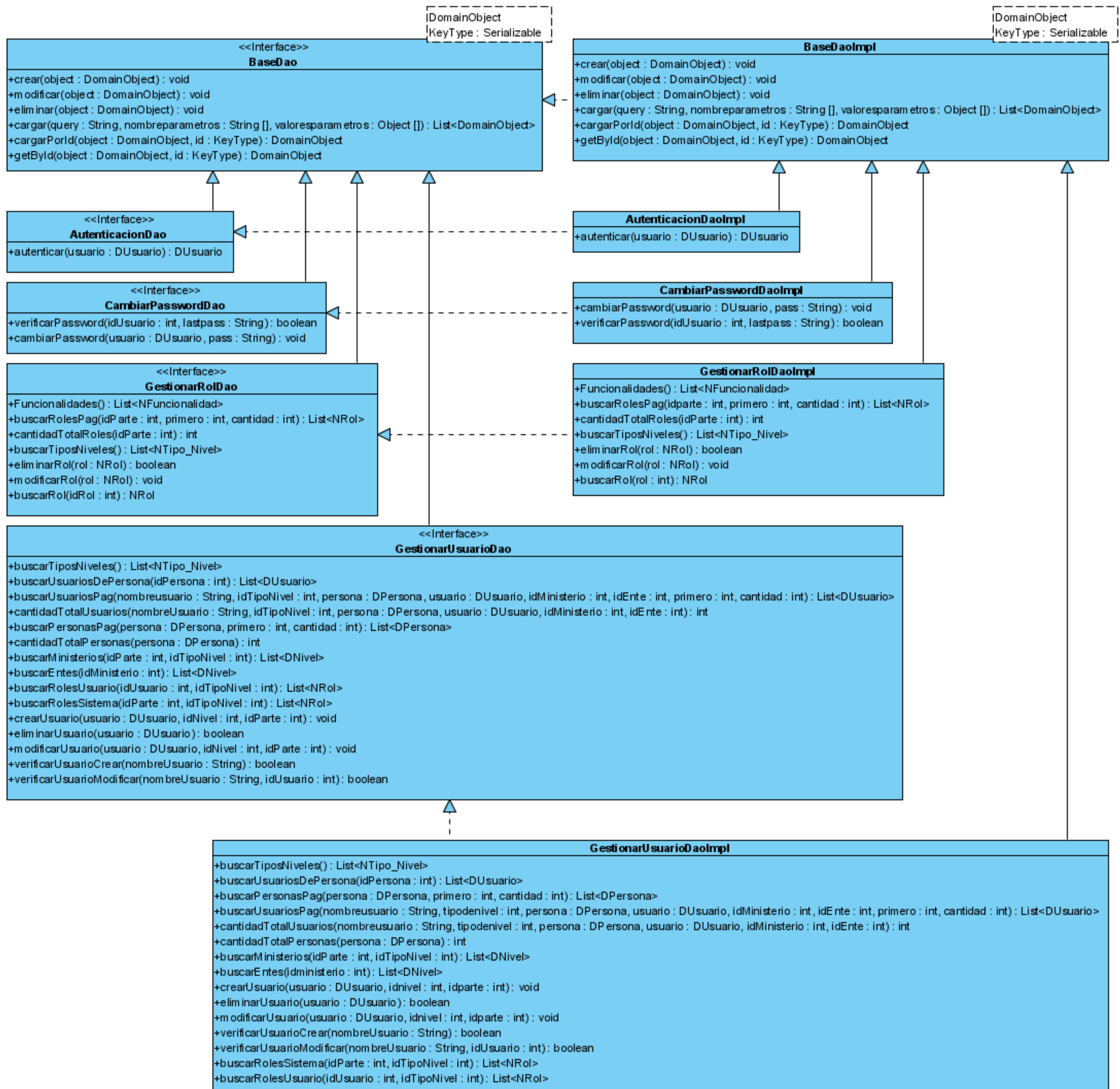
Unas de las entradas más importantes para el flujo de implementación lo constituyen los modelos de diseño ya sean estos el de diseño de las clases como los de clases persistentes y modelo de la base de datos. Estos constituyen la base sobre la que se comienza el desarrollo, estos garantizan la correcta organización de las mismas además de describir de antemano las realizaciones de los casos de uso. Estos sirven como una abstracción de la implementación y del código fuente que se genera. A partir de los mismos se generan los scripts que se utilizan posteriormente para una primera generación de la base de datos y además se generan los mapping y las clases e interfaces sobre los que se comenzará el desarrollo.

Estos además aseguran que las clases proporcionen el comportamiento que requieren las realizaciones de los casos de uso, así como también, la consistencia con la arquitectura de *software*, y la información suficiente para evitar ambigüedades en las clases implementadas.

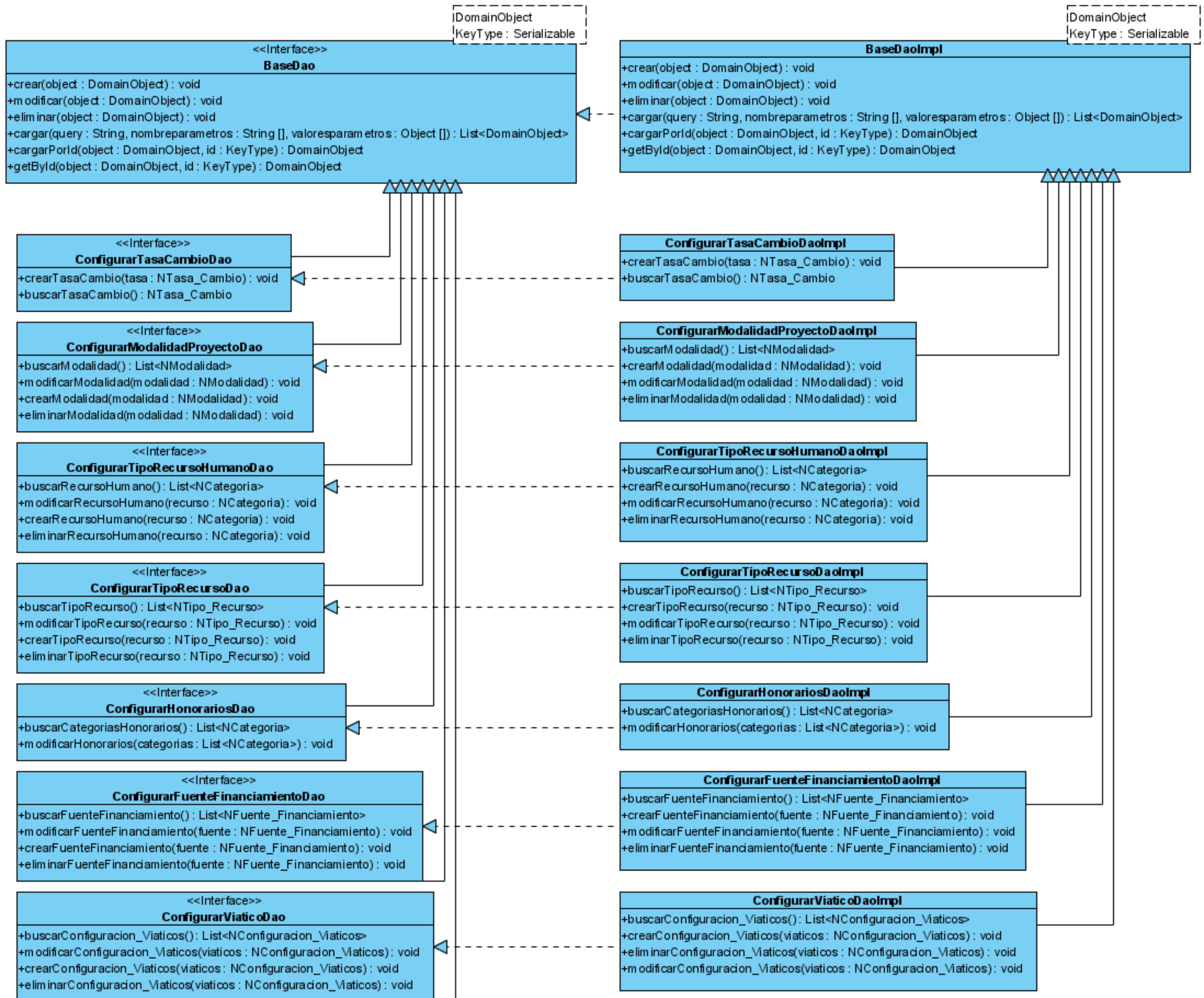
A continuación se muestran los diseños de clases y de datos propuestos los mismo se enmarcan solo en los que tienen relación directamente con las clases de persistencia de datos y los DAOs, más precisamente solo de aquellos que tienen una relación directa con los módulos de administración y configuración, el diseño en su versión completa comprende todas las partes de la aplicación.

Modulo de Administración y Configuración (Diagrama de clases de diseño)

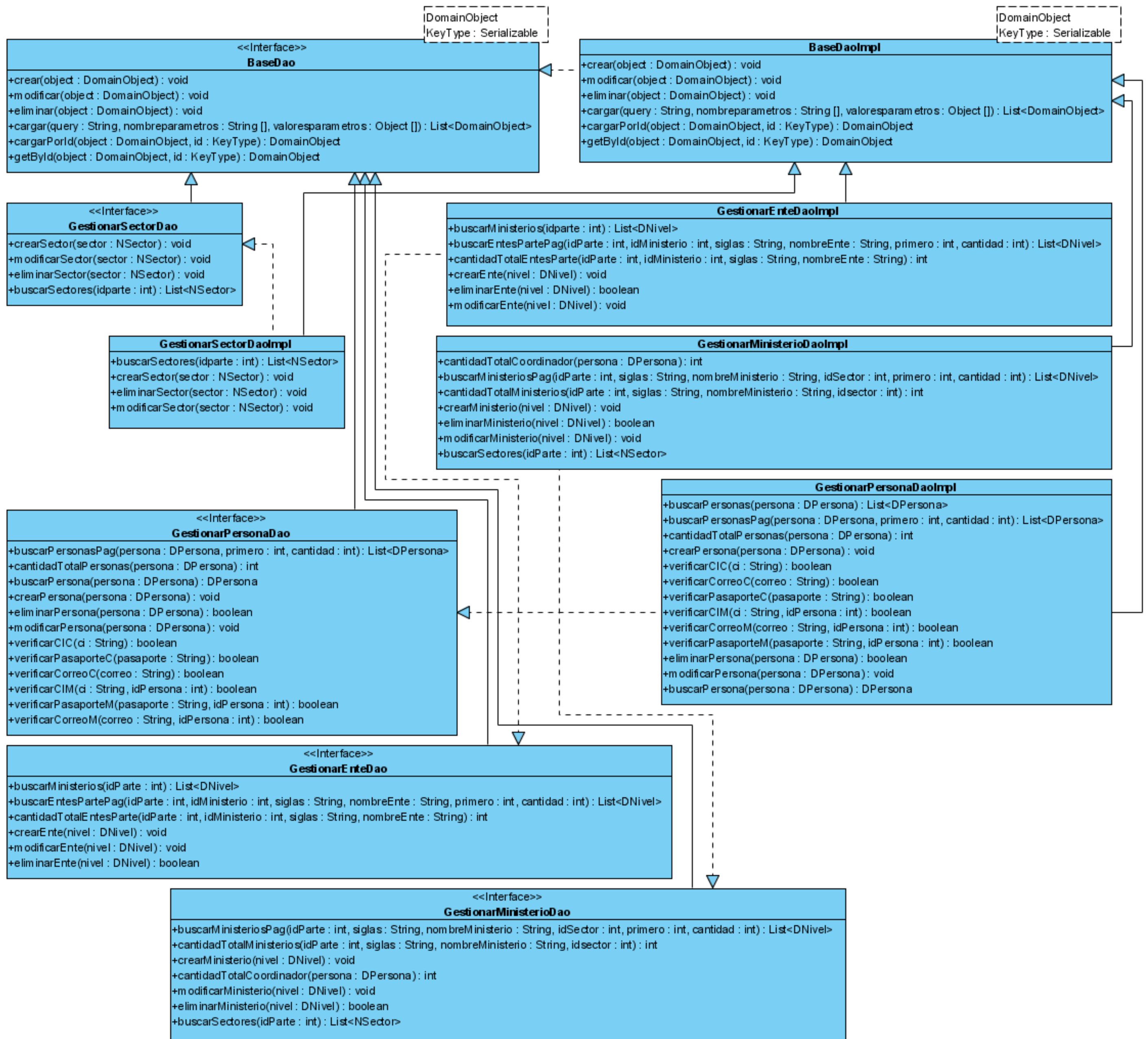
Casos de Usos (Administración).



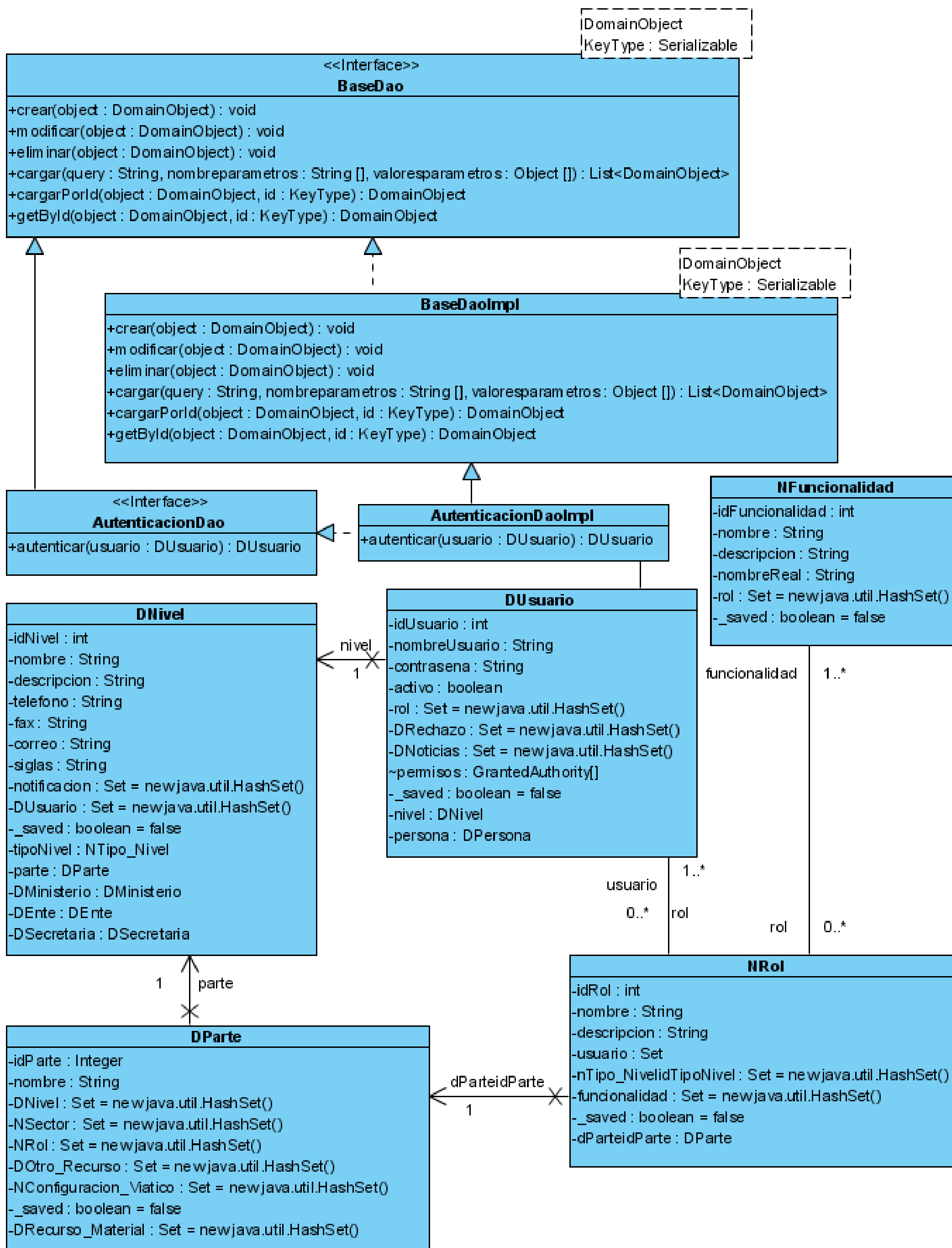
Casos de Usos (Configuración Nomencladores).



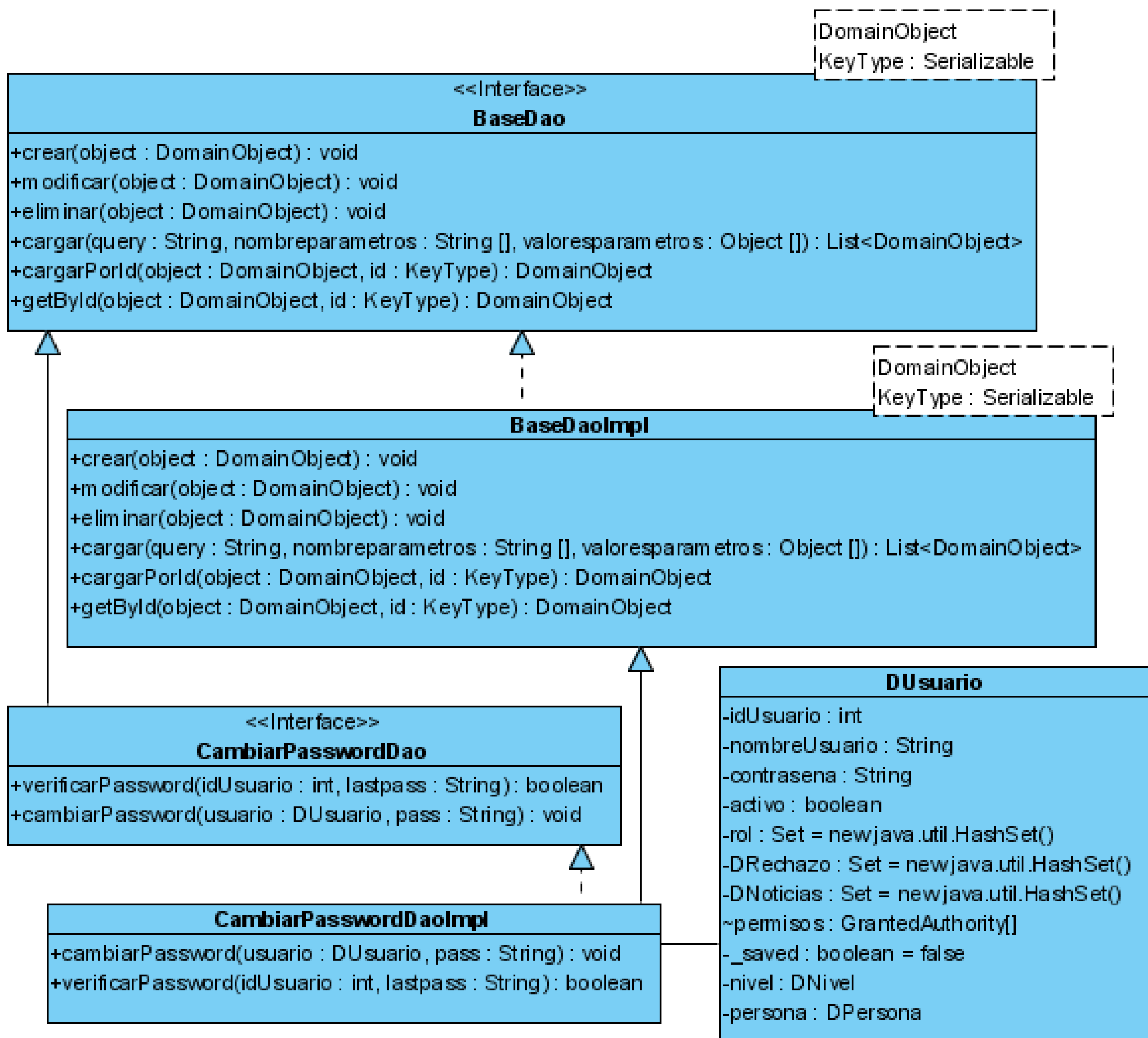
Casos de Usos (Configuración Sistema).



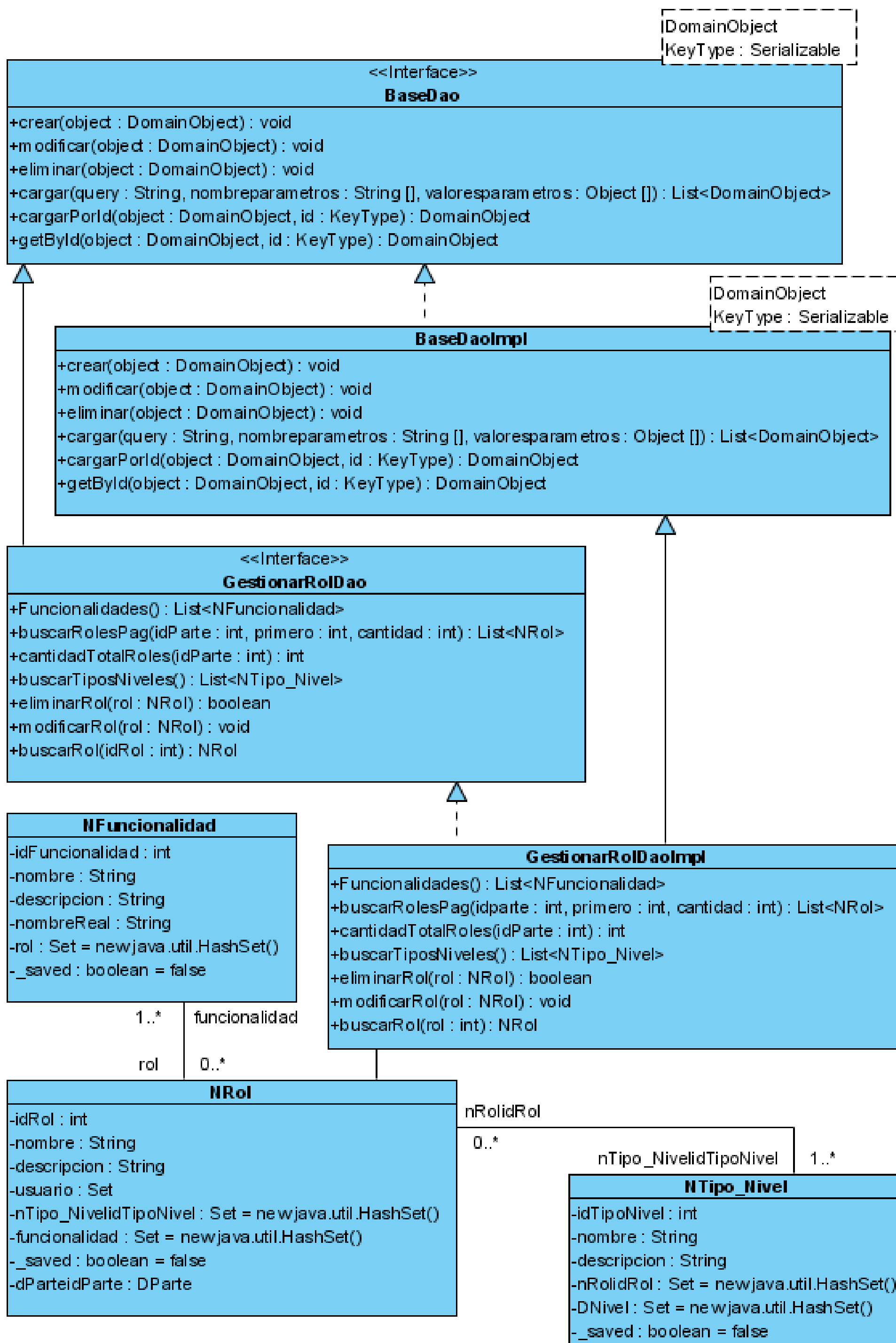
Caso de uso (Autentificar Usuario)



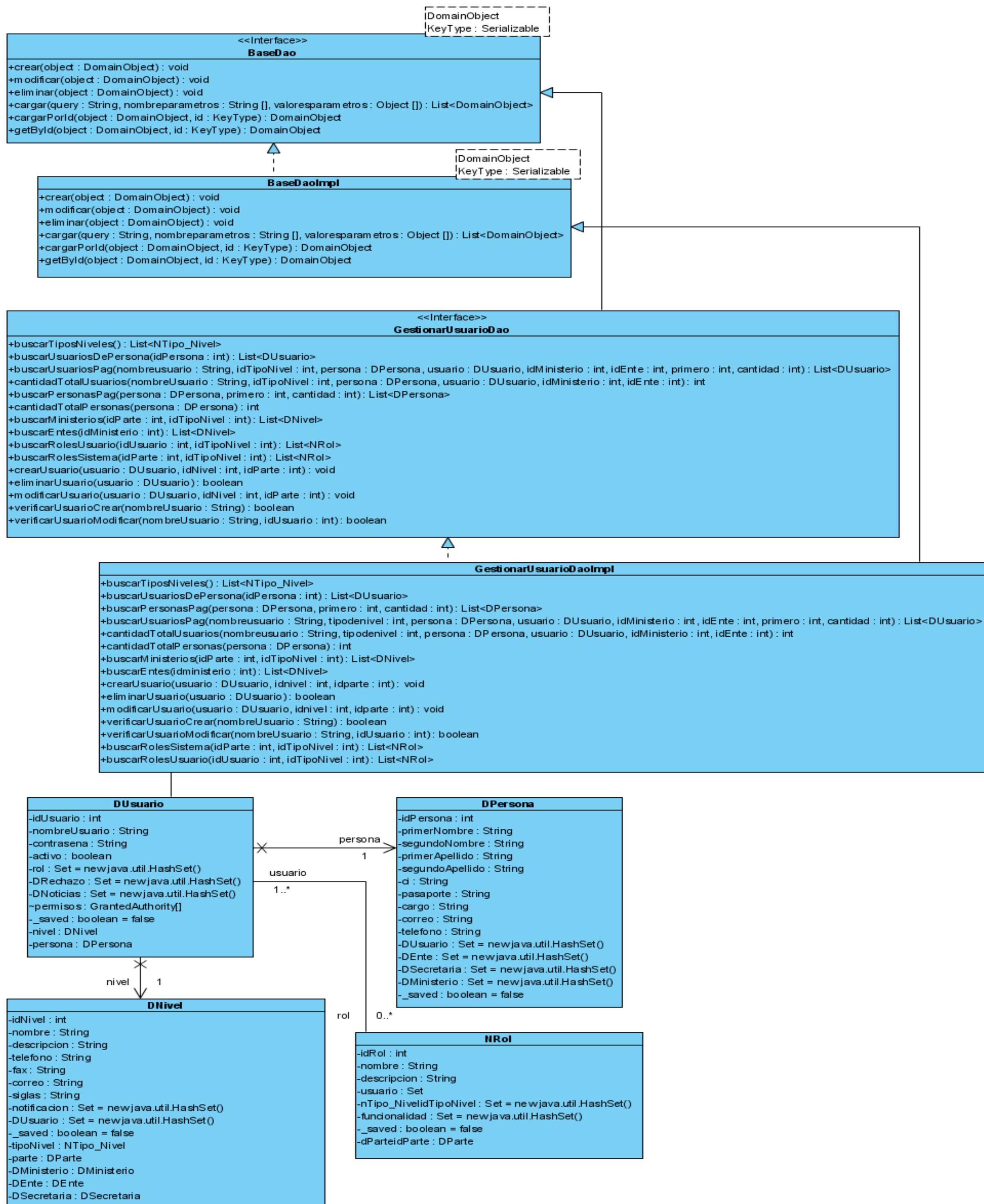
Caso de uso (Cambiar contraseña)



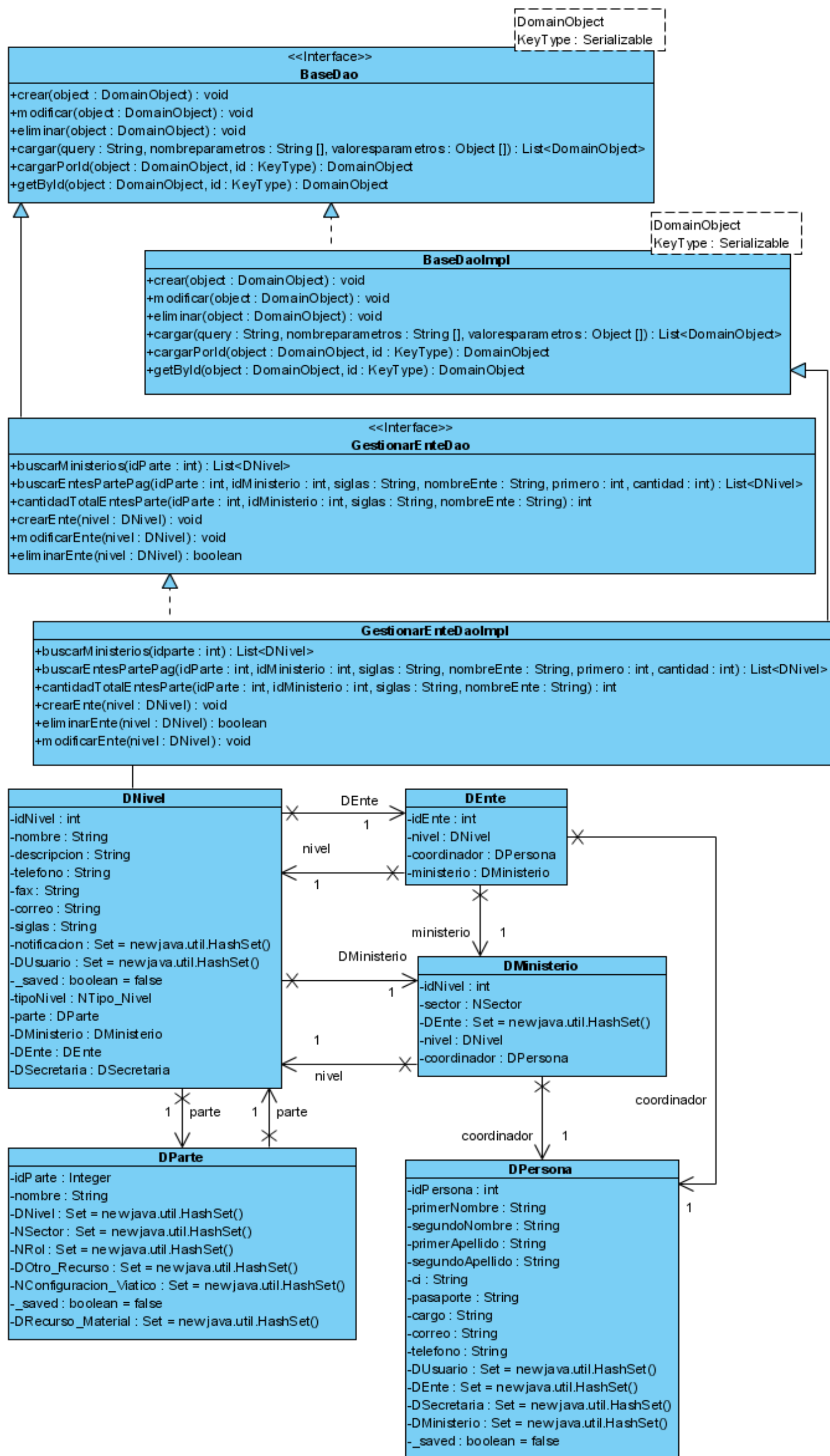
Caso de uso (Gestionar Rol)



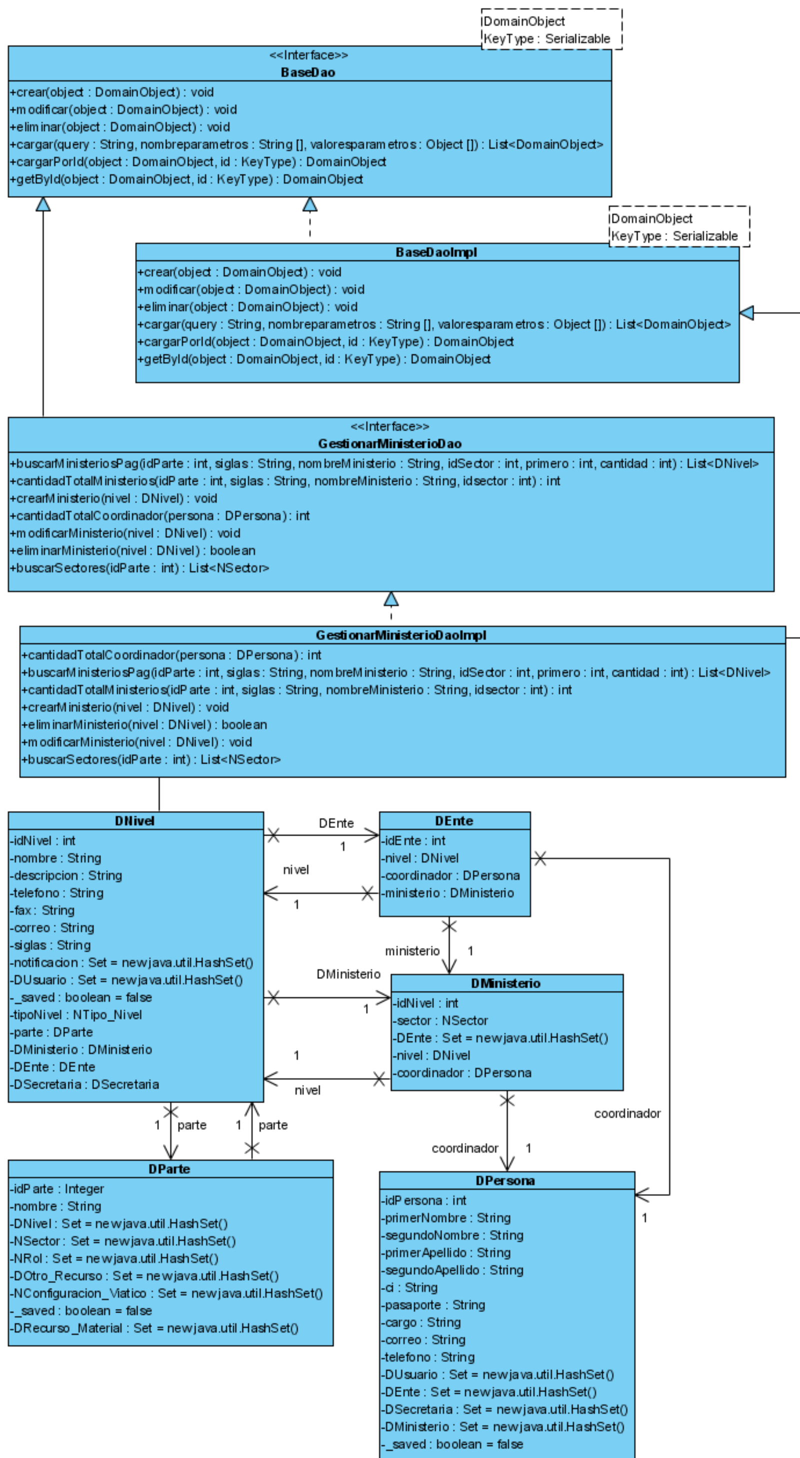
Caso de uso (Gestionar Usuario)



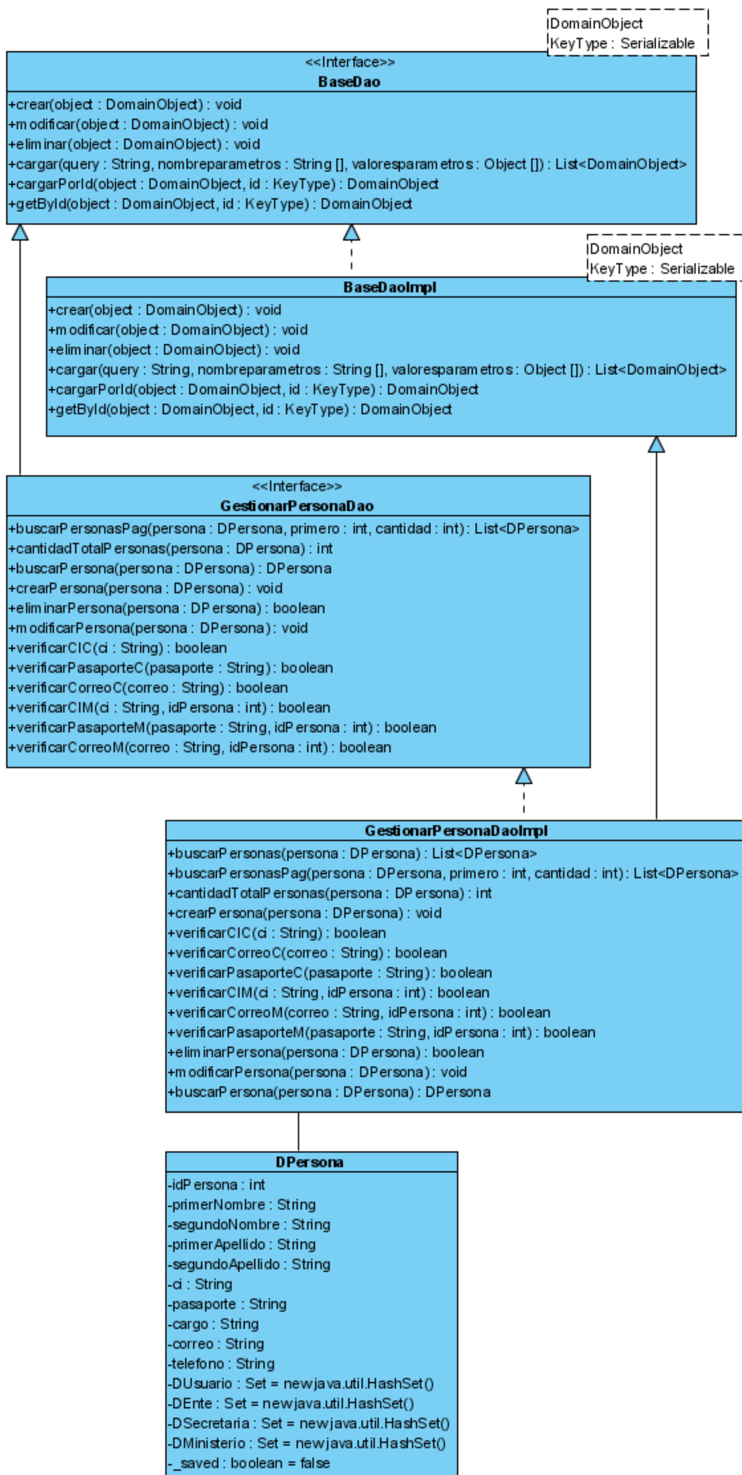
Caso de uso (Gestionar Ente)



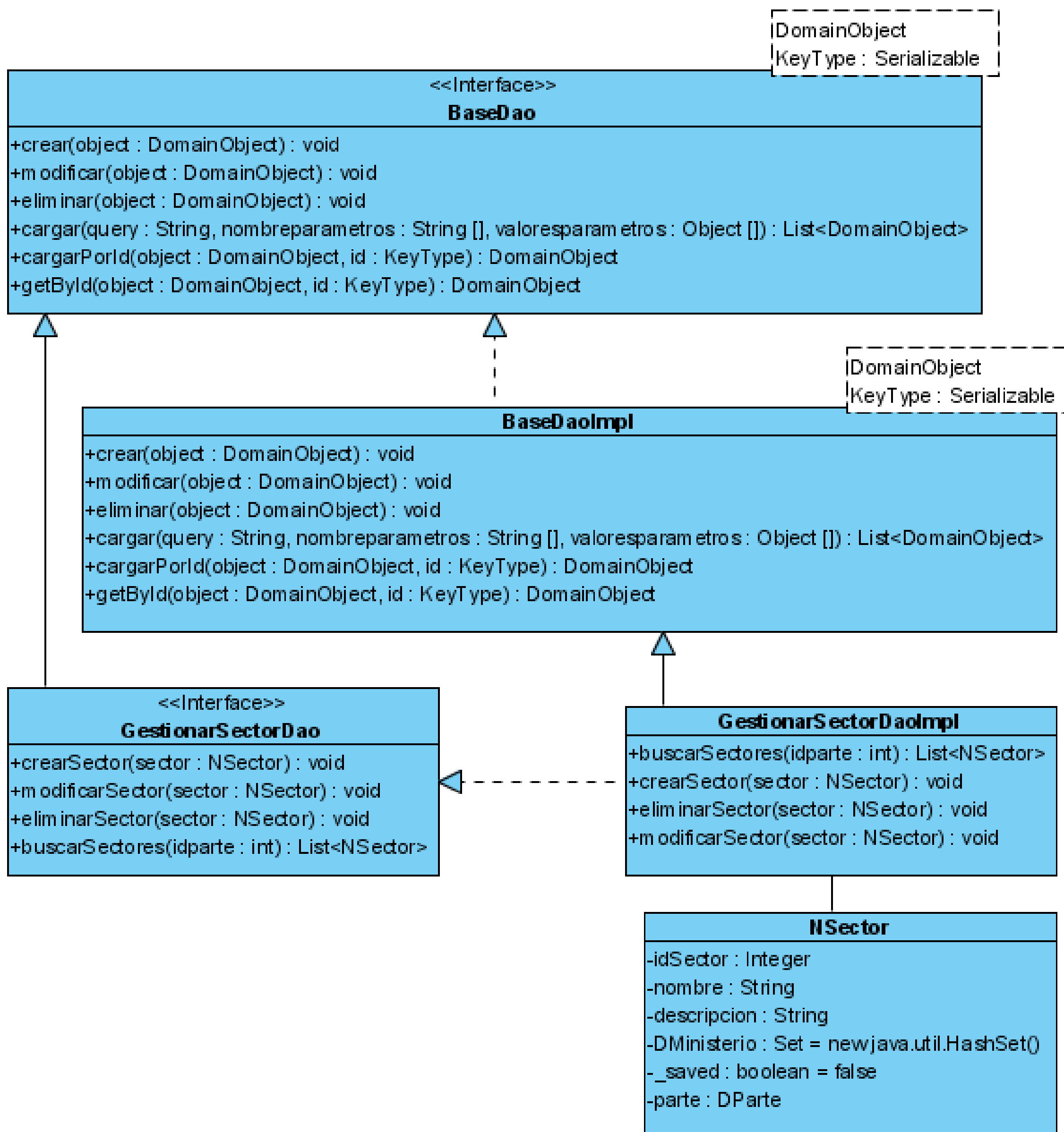
Caso de uso (Gestionar Ministerio)



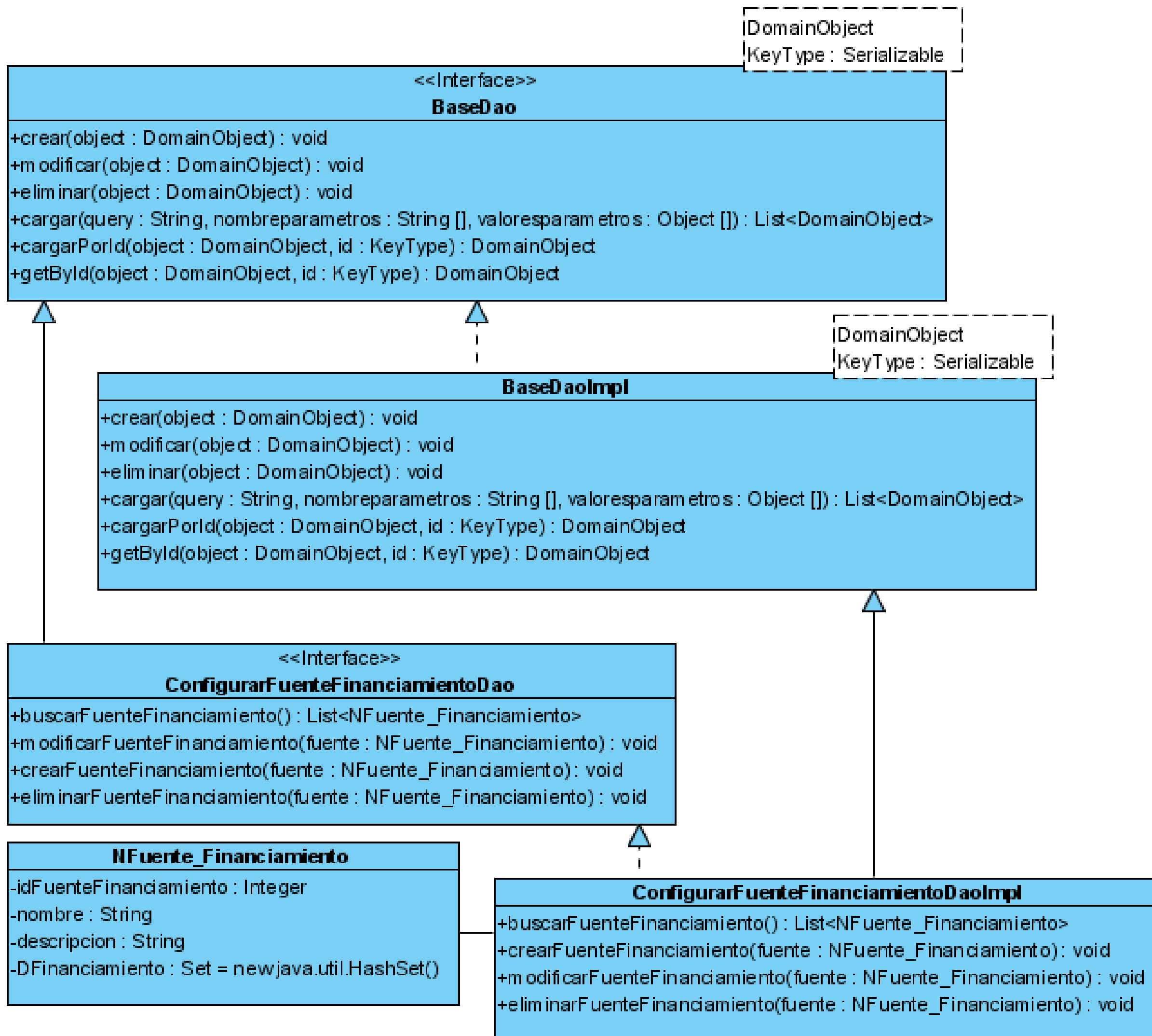
Caso de uso (Gestionar Persona)



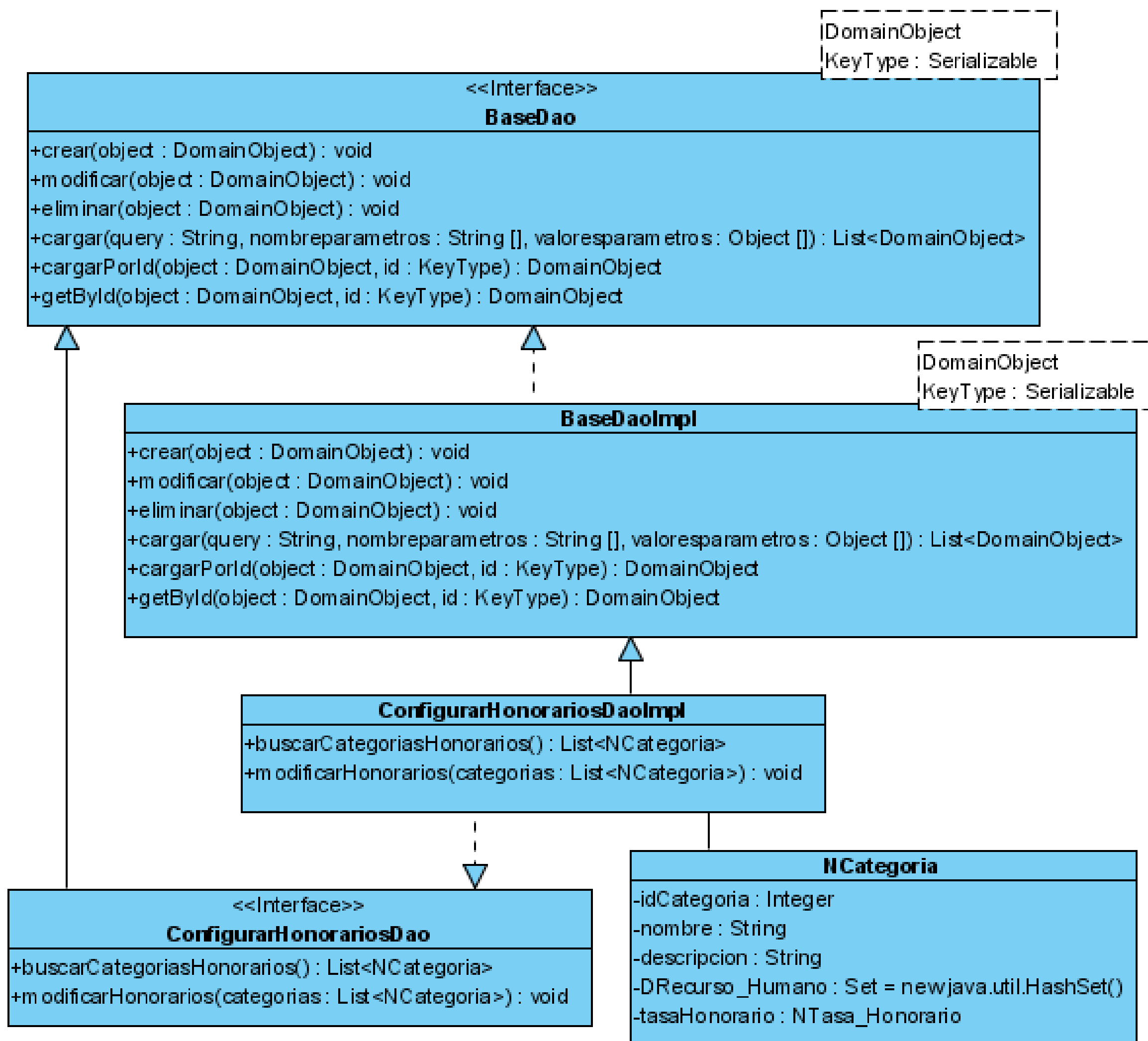
Caso de uso (Gestionar Sector)



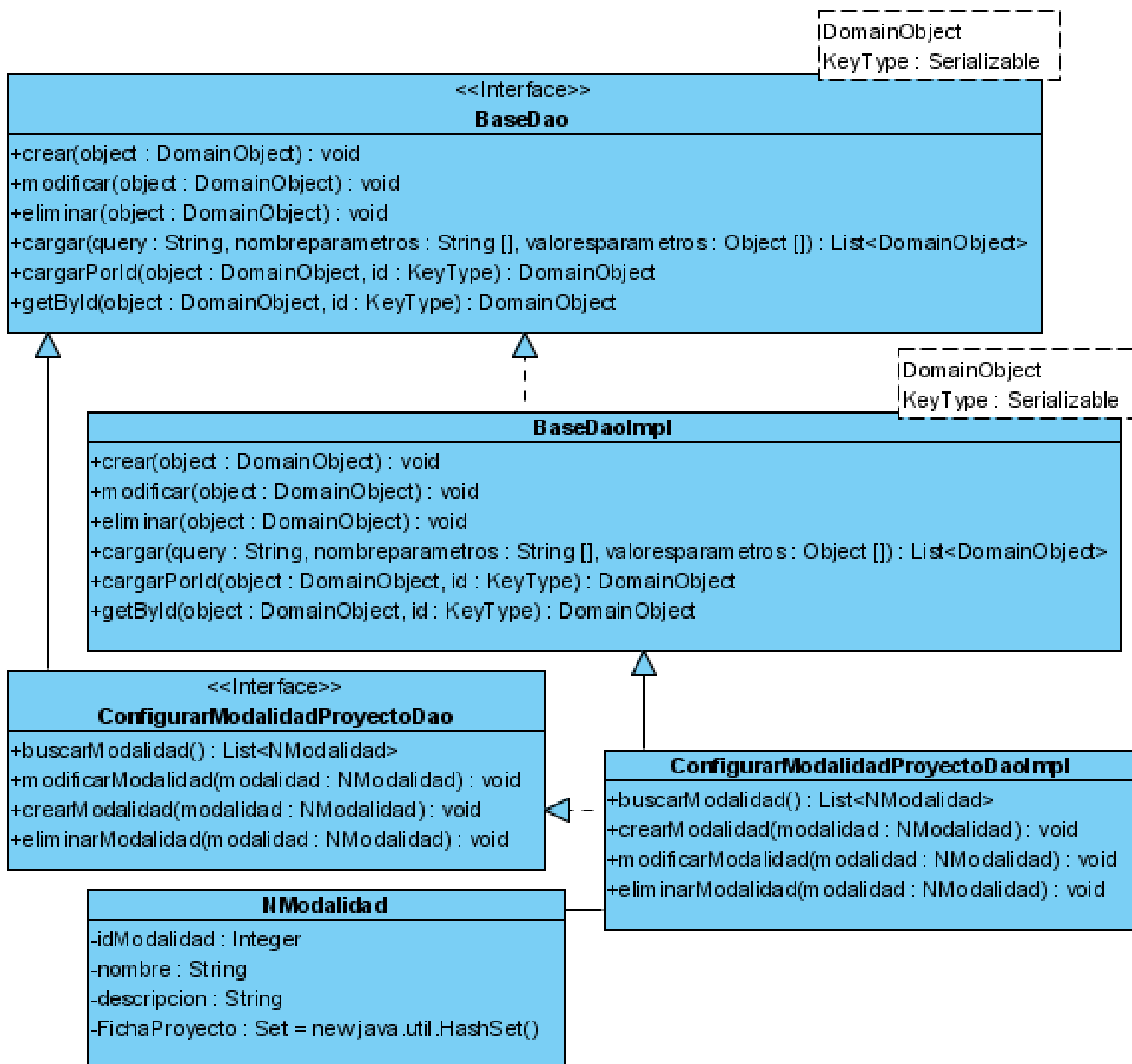
Caso de uso (Configurar Fuente de Financiamiento)



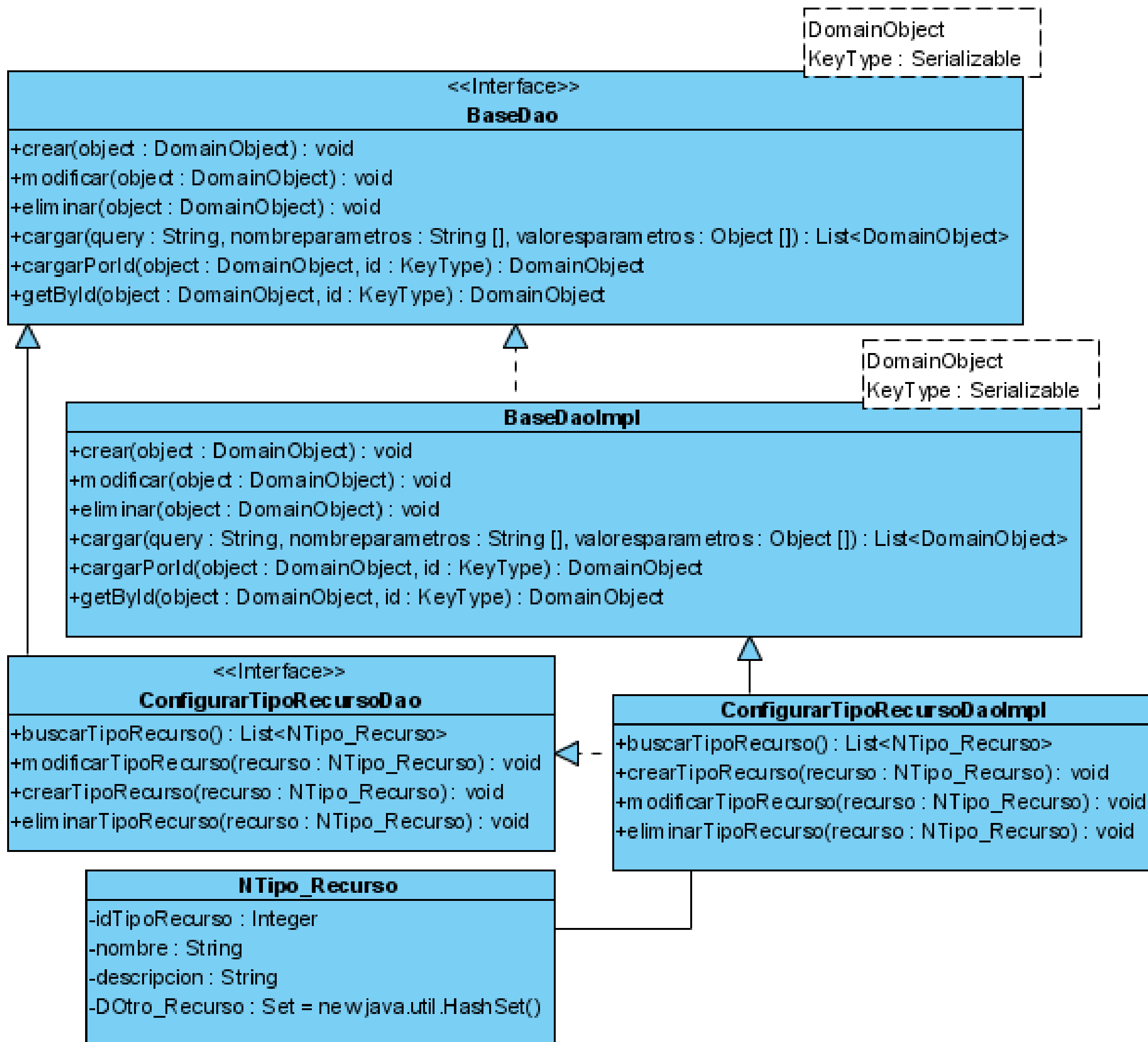
Caso de uso (Configurar Honorarios)



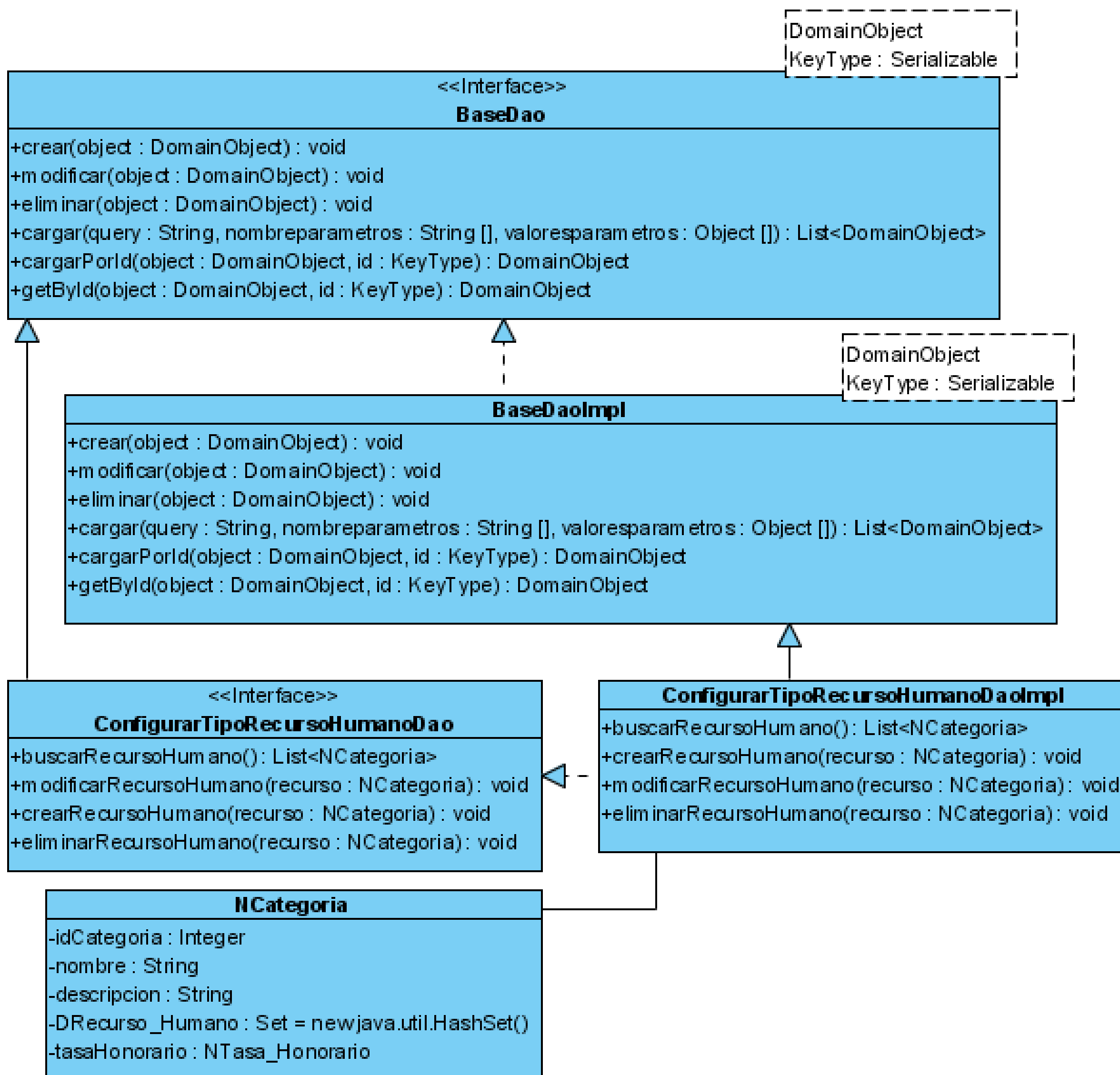
Caso de uso (Configuración Modalidad Proyecto)



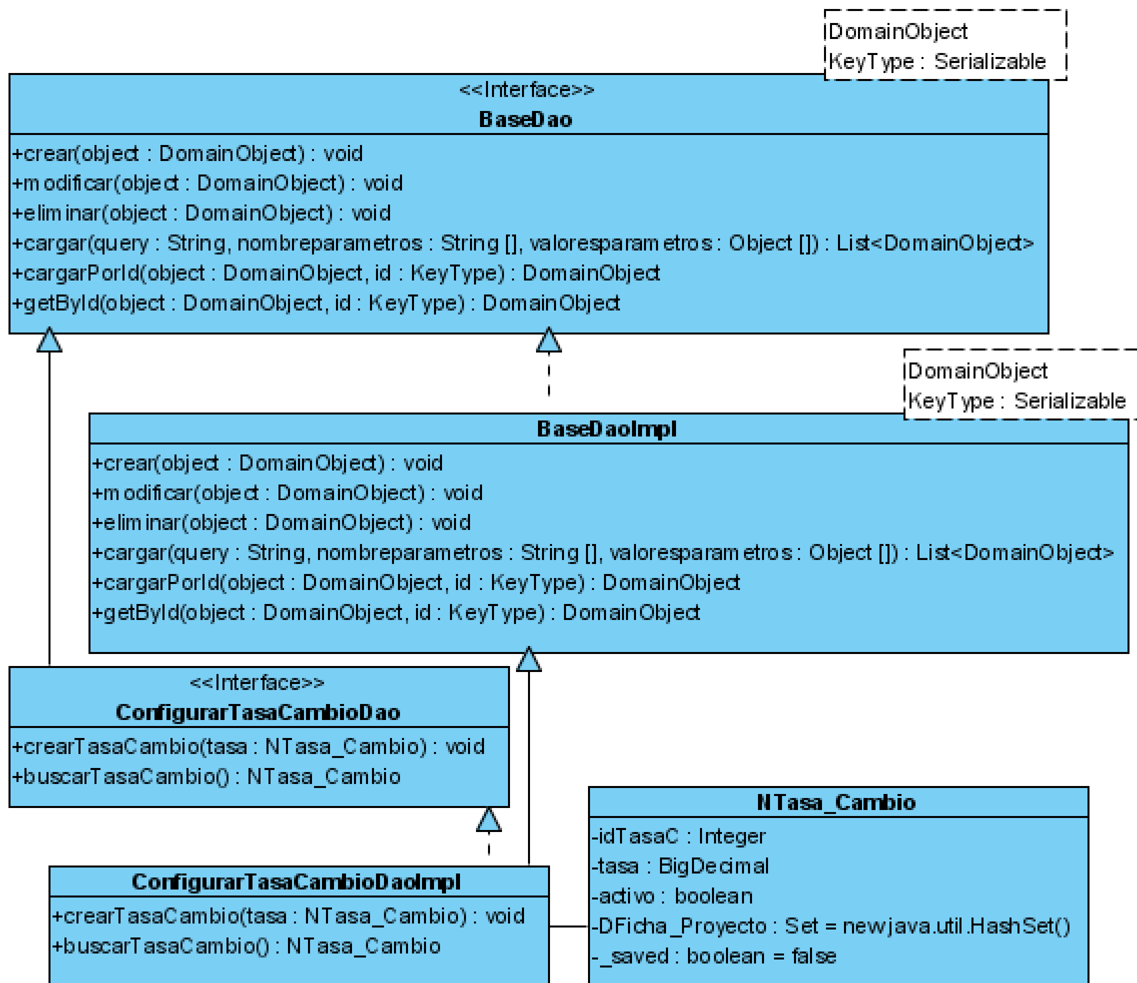
Caso de uso (Configuración Tipo Recurso)



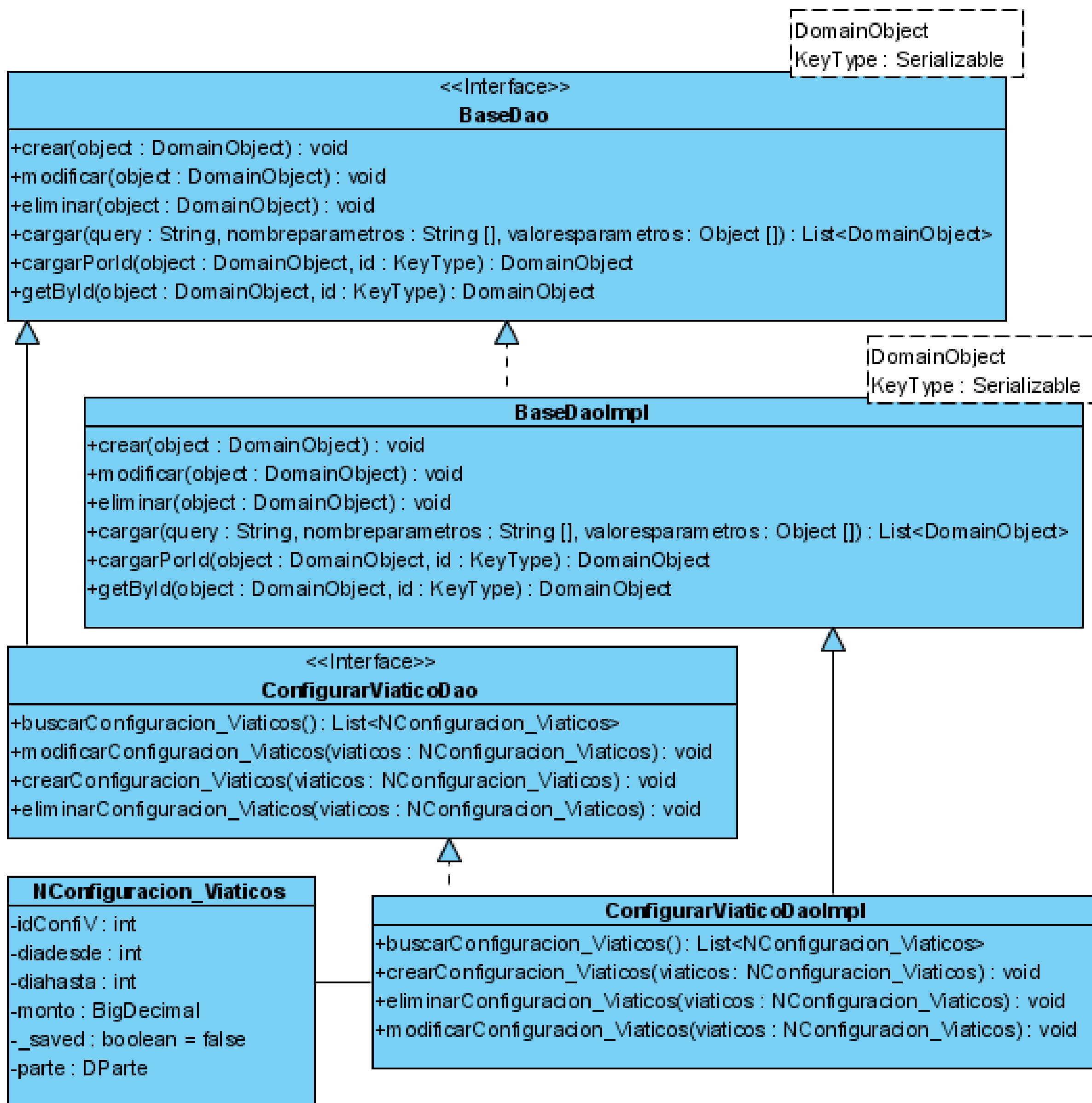
Caso de uso (Tipo de Recurso Humano)



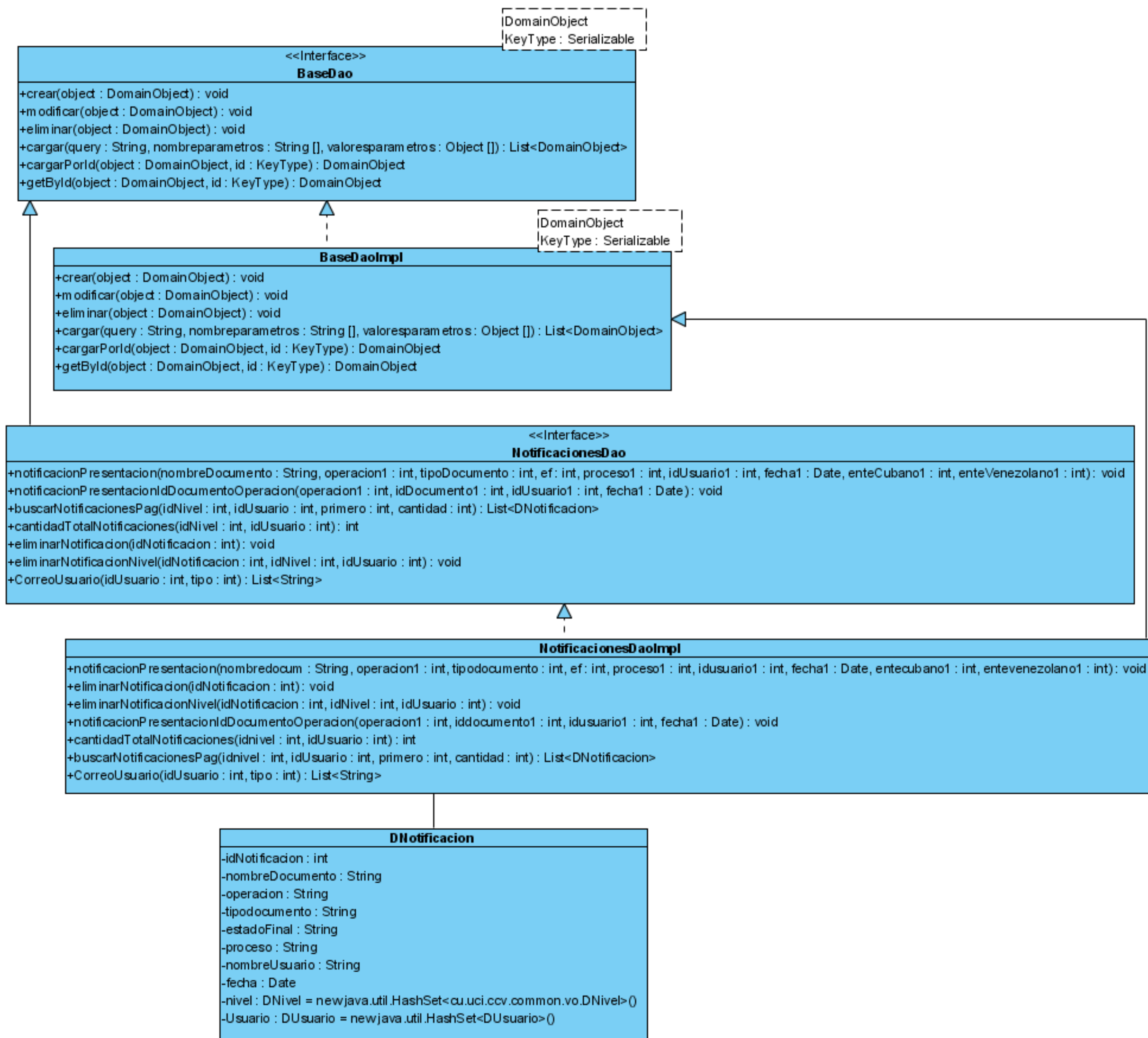
Caso de uso (Configuración Tasa de Cambio).



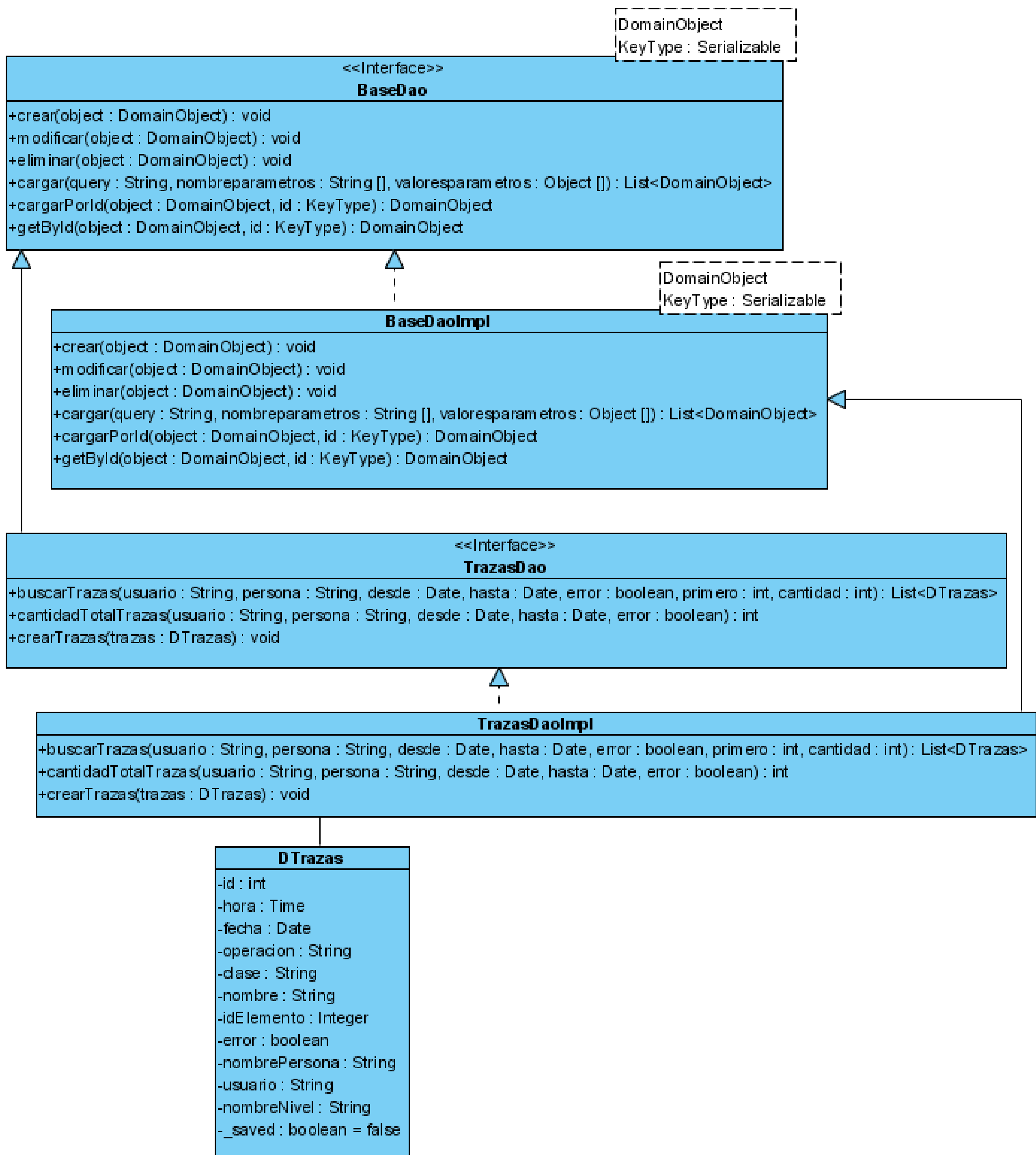
Caso de uso (Configuración Viático)



Caso de uso (Notificaciones)



Caso de uso (Trazas)



A continuación se presentan los modelos de clases persistentes y el modelo de entidades de los cuales se ha tratado de eliminar el contacto con otros módulos y que traería ambigüedades y no aportan a este trabajo. A partir de los mismos se pasara a explicar cómo es su correlación y la correspondencia entre ambos modelos.

Modelo de diseño de las clases persistentes

DPersona
-idPersona : int
-primerNombre : String
-segundoNombre : String
-primerApellido : String
-segundoApellido : String
-ci : String
-pasaporte : String
-cargo : String
-correo : String
-telefono : String
-DUsuario : Set = newjava.util.HashSet()
-DEnte : Set = newjava.util.HashSet()
-DSecretaria : Set = newjava.util.HashSet()
-DMinisterio : Set = newjava.util.HashSet()
-_saved : boolean = false

DUsuario
-idUsuario : int
-nombreUsuario : String
-contrasena : String
-activo : boolean
-rol : Set = newjava.util.HashSet()
-DRechazo : Set = newjava.util.HashSet()
-DNoticias : Set = newjava.util.HashSet()
~permisos : GrantedAuthority[]
-_saved : boolean = false
-nivel : DNivel
-persona : DPersona

NRol
-idRol : int
-nombre : String
-descripcion : String
-usuario : Set
-nTipo_NivelidTipoNivel : Set = newjava.util.HashSet()
-funcionalidad : Set = newjava.util.HashSet()
-_saved : boolean = false
-dParteidParte : DParte

1 persona
usuario rol 0..* 0..* funcionalidad rol 1..*
1..* Usuario *
nivel 1

NFuncionalidad
-idFuncionalidad : int
-nombre : String
-descripcion : String
-nombreReal : String
-rol : Set = newjava.util.HashSet()
-_saved : boolean = false

DNivel
-idNivel : int
-nombre : String
-descripcion : String
-telefono : String
-fax : String
-correo : String
-siglas : String
-notificacion : Set = newjava.util.HashSet()
-DUsuario : Set = newjava.util.HashSet()
-_saved : boolean = false
-tipoNivel : NTipo_Nivel
-parte : DParte
-DMinisterio : DMinisterio
-DEnte : DEnte
-DSecretaria : DSecretaria

1 1 coordinador 1
coordinador coordinador

DEnte
-idEnte : int
-nivel : DNivel
-coordinador : DPersona
-ministerio : DMinisterio

1 nivel
1 DEnte
ministerio 1

NConfiguracion_Viaticos
-idConfiV : int
-diadesde : int
-diahasta : int
-monto : BigDecimal
-_saved : boolean = false
-parte : DParte

DMinisterio
-idNivel : int
-sector : NSector
-DEnte : Set = newjava.util.HashSet()
-nivel : DNivel
-coordinador : DPersona

nivel 1
DMinisterio 1 nivel nivel *
ministerio 1 0..* sector 1

DVisibilidadNotificacion
-idVisibilidad : int
-estadoInicial : NEstado
-estadoFinal : NEstado
-visibilidad : Integer
-_saved : boolean = false

NSector
-idSector : Integer
-nombre : String
-descripcion : String
-DMinisterio : Set = newjava.util.HashSet()
-_saved : boolean = false
-parte : DParte

DNotificacion
-idNotificacion : int
-nombreDocumento : String
-operacion : String
-tipodocumento : String
-estadoFinal : String
-proceso : String
-nombreUsuario : String
-fecha : Date
-nivel : DNivel = newjava.util.HashSet<cu.uci.ccv.comm...
-Usuario : DUsuario = newjava.util.HashSet<DUsuario>()

DTrazas
-id : int
-hora : Time
-fecha : Date
-operacion : String
-clase : String
-nombre : String
-idElemento : Integer
-error : boolean
-nombrePersona : String
-usuario : String
-nombreNivel : String
-_saved : boolean = false

DSecretaria 1

DSecretaria
-nivel : DNivel
-coordinador : DPersona

1 tipoNivel

NTipo_Nivel
-idTipoNivel : int
-nombre : String
-descripcion : String
-nRolidRol : Set = newjava.util.HashSet()
-DNivel : Set = newjava.util.HashSet()
-_saved : boolean = false

NTasa_Cambio
-idTasaC : Integer
-tasa : BigDecimal
-activo : boolean
-DFicha_Proyecto : Set = newjava.util.HashSet()
-_saved : boolean = false

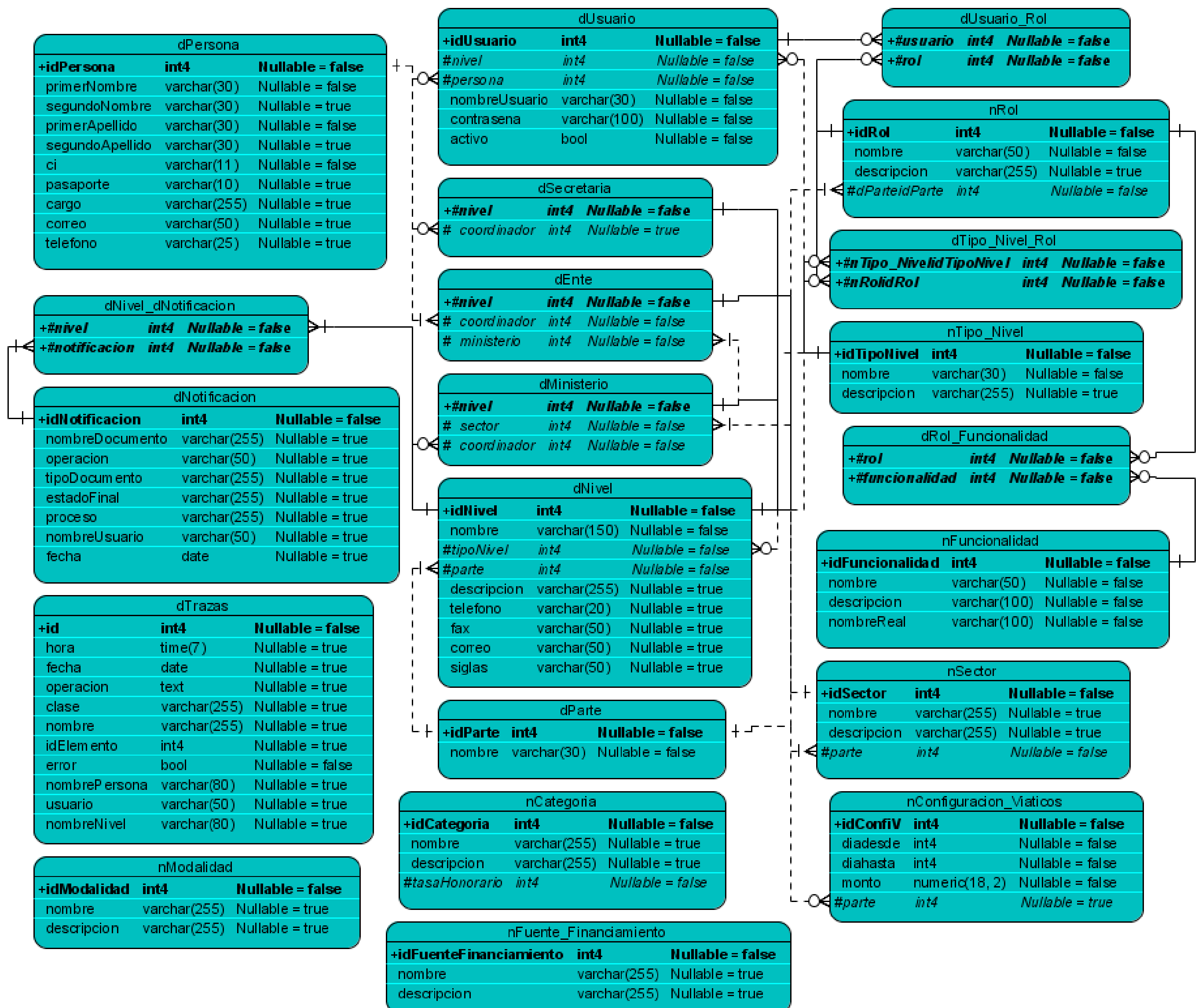
NCategoria
-idCategoria : Integer
-nombre : String
-descripcion : String
-DRecurso_Humano : Set = newjava.util.HashSet()
-tasaHonorario : NTasa_Honorario

NFuente_Financiamiento
-idFuenteFinanciamiento : Integer
-nombre : String
-descripcion : String
-DFinanciamiento : Set = newjava.util.HashSet()

NTipo_Recurso
-idTipoRecurso : Integer
-nombre : String
-descripcion : String
-DOtro_Recurso : Set = newjava.util.HashSet()

NModalidad
-idModalidad : Integer
-nombre : String
-descripcion : String
-FichaProyecto : Set = newjava.util.HashSet()

Modelo de datos



Implementación.

Con la utilización de los DAOs se crea una separación entre la aplicación y la base de datos; la utilización del patrón DAO nos permite encapsular todos los requerimientos para acceder a la base de datos y ocultar de la capa de negocios la forma de acceder a los mismos. En su interior se definen todas las operaciones de persistencia (CRUD) las cual nos permiten hacer la mayoría de las operaciones sobre las bases de datos.

La implementación de los mismos se realizó buscando en todo momento la mayor posibilidad de mantenimiento posterior ya que el sistema se puede ver sometido a cambios para esto se aprovecharon las capacidades que nos ofrece el *framework* de trabajo Spring el cual mediante la declaración de beans y el hecho de que estos estén declarados en XMLs nos permite su posterior modificación y aprovechándonos de la inyección de instancias se puede en cualquier momento cambiar la implementación que se le inyecta a una interfaz.

Partiendo del diseño de clases propuestos y se declararon las siguientes interfaces con los métodos que se describirán a continuación.

Interfaz	public interface GestionarRolDao extends BaseDao
Métodos	
Nombre	public List<NFuncionalidad> Funcionalidades()
Descripción	Devuelve una lista con las funcionalidades del sistema.
Nombre	public List<NRol> buscarRolesPag(int idParte,int primero,int cantidad)
Descripción	Devuelve una lista limitada de roles de una de las partes.
Nombre	public int cantidadTotalRoles(int idParte)
Descripción	Devuelve la cantidad total de roles de una de las partes.
Nombre	public List<NTipo_Nivel> buscarTiposNiveles()
Descripción	Devuelve una lista de los tipos de nivel existentes en el sistema.
Nombre	public boolean eliminarRol(NRol rol)

Descripción	Intenta eliminar un rol, lanza una excepción si este todavía conserva alguna relación.
Nombre	public void modificarRol(NRol rol)
Descripción	Modifica un rol.
Nombre	public NRol buscarRol(int idRol)
Descripción	Devuelve un rol específico.
Interfaz	public interface GestionarUsuarioDao extends BaseDao
Métodos	
Nombre	public List<NTipo_Nivel> buscarTiposNiveles()
Descripción	Devuelve una lista con los tipos de nivel.
Nombre	public List<DUsuario> buscarUsuariosDePersona(int idPersona)
Descripción	Devuelve una lista con los usuarios de una persona determinada.
Nombre	public List<DUsuario> buscarUsuariosPag(String nombreusuario,int idTipoNivel,DPersona persona,DUsuario usuario,int idMinisterio , int idEnte,int primero,int cantidad)
Descripción	Devuelve una lista de usuarios limitada dado un grupo de criterios.
Nombre	public int cantidadTotalUsuarios(String nombreUsuario,int idTipoNivel,DPersona persona,DUsuario usuario,int idMinisterio , int idEnte)
Descripción	Devuelve la cantidad total de usuarios dado un grupo de criterios.
Nombre	public List<DPersona> buscarPersonasPag(DPersona persona,int primero,int cantidad)
Descripción	Devuelve una lista limitada de personas.
Nombre	public int cantidadTotalPersonas(DPersona persona)
Descripción	Devuelve la cantidad total de personas.
Nombre	public List<DNivel> buscarMinisterios(int idParte, int idTipoNivel)
Descripción	Devuelve una lista de ministerios dada una parte y una visibilidad dada por el tipo de nivel.
Nombre	public List<DNivel> buscarEntes(int idMinisterio)

Descripción	Devuelve una lista de entes de un ministerio.
Nombre	public List<NRol> buscarRolesUsuario(int idUsuario , int idTipoNivel)
Descripción	Devuelve una lista de roles dado un usuario y una visibilidad dada por el tipo de nivel.
Nombre	public List<NRol> buscarRolesSistema(int idParte ,int idTipoNivel)
Descripción	Devuelve una lista de roles dado una parte y un tipo de nivel específico.
Nombre	public void crearUsuario(DUsuario usuario, int idNivel, int idParte)
Descripción	Crea un usuario en un nivel y para una parte específica.
Nombre	public boolean eliminarUsuario(DUsuario usuario)
Descripción	Elimina un usuario.
Nombre	public void modificarUsuario(DUsuario usuario, int idNivel, int idParte)
Descripción	Modifica un usuario.
Nombre	public boolean verificarUsuarioCrear(String nombreUsuario)
Descripción	Verifica que un nombre de usuario no exista.
Nombre	public boolean verificarUsuarioModificar(String nombreUsuario , int idUsuario)
Descripción	Verifica que un nombre de usuario no exista excepto para el usuario que está realizando el cambio.
Interfaz	public interface CambiarPasswordDao extends BaseDao
Métodos	
Nombre	public boolean verificarPassword(int idUsuario, String lastpass)
Descripción	Verificar la contraseña anterior de un usuario.
Nombre	public void cambiarPassword(DUsuario usuario, String pass)
Descripción	Modificar la contraseña de un usuario
Interfaz	public interface AutenticacionDao extends BaseDao<DUsuario, Serializable>
Métodos	
Nombre	public DUsuario autenticar(DUsuario usuario)
Descripción	Autenticar un usuario y devolverlo con todos sus permisos.

Interfaz	public interface GestionarPersonaDao extends BaseDao
Métodos	
Nombre	public List<DPersona> buscarPersonasPag(DPersona persona,int primero,int cantidad)
Descripción	Devolver una lista de personas limitada que cumplan con características en específico.
Nombre	public int cantidadTotalPersonas(DPersona persona)
Descripción	Devuelve la cantidad total de personas que cumplan con características en específico.
Nombre	public DPersona buscarPersona(DPersona persona)
Descripción	Devuelve una persona con todas sus relaciones.
Nombre	public void crearPersona(DPersona persona)
Descripción	Crea una persona.
Nombre	public boolean eliminarPersona(DPersona persona)
Descripción	Intenta eliminar una persona, lanza una excepción si este todavía conserva alguna relación.
Nombre	public void modificarPersona(DPersona persona)
Descripción	Modifica una persona.
Nombre	public boolean verificarCIC(String ci)
Descripción	Verifica que un Carnet de Identidad no exista.
Nombre	public boolean verificarPasaporteC(String pasaporte)
Descripción	Verifica que un pasaporte no exista.
Nombre	public boolean verificarCorreoC(String correo)
Descripción	Verifica que un correo electrónico no exista.
Nombre	public boolean verificarCIM(String ci,int idPersona)
Descripción	Verifica que un Carnet de Identidad no exista excepto parra una persona específica.
Nombre	public boolean verificarPasaporteM(String pasaporte,int idPersona)
Descripción	Verifica que un pasaporte no exista excepto parra una persona

	específica.
Nombre	public boolean verificarCorreoM(String correo,int idPersona)
Descripción	Verifica que un correo electrónico no exista excepto parra una persona específica.
Interfaz	public interface GestionarEnteDao extends BaseDao
Métodos	
Nombre	public List<DNivel> buscarMinisterios(int idParte)
Descripción	Devuelve una lista de ministerios dada una parte específica.
Nombre	public List<DNivel> buscarEntesPartePag(int idParte, int idMinisterio, String siglas, String nombreEnte, int primero, int cantidad)
Descripción	Devuelve una lista limitada de entes dado un grupo de características.
Nombre	public int cantidadTotalEntesParte(int idParte,int idMinisterio , String siglas,String nombreEnte)
Descripción	Devuelve la cantidad total de entes dado un grupo de características.
Nombre	public void crearEnte(DNivel nivel)
Descripción	Crea un ente
Nombre	public void modificarEnte(DNivel nivel)
Descripción	Modifica un ente.
Nombre	public boolean eliminarEnte(DNivel nivel)
Descripción	Intenta eliminar un ente, lanza una excepción si este todavía conserva alguna relación.
Interfaz	public interface GestionarMinisterioDao extends BaseDao
Métodos	
Nombre	public List<DNivel> buscarMinisteriosPag(int idParte,String siglas,String nombreMinisterio,int idSector,int primero,int cantidad)
Descripción	Devuelve una lista limitada de ministerios dado un grupo de características.
Nombre	public int cantidadTotalMinisterios(int idParte,String siglas,String nombreMinisterio,int idsector)

Descripción	Devuelve la cantidad total de ministerios dado un grupo de características.
Nombre	public void crearMinisterio(DNivel nivel)
Descripción	Crea un ministerio.
Nombre	public int cantidadTotalCoordinador(DPersona persona)
Descripción	Devuelve la cantidad total de personas que pueden ser coordinadores dado un grupo de características.
Nombre	public void modificarMinisterio(DNivel nivel)
Descripción	Modifica un ministerio.
Nombre	public boolean eliminarMinisterio(DNivel nivel)
Descripción	Intenta eliminar un ministerio, lanza una excepción si este conserva todavía alguna relación.
Nombre	public List<NSector> buscarSectores(int idParte)
Descripción	Devuelve una lista de sectores dada una parte.
Interfaz	public interface GestionarSectorDao extends BaseDao
Métodos	
Nombre	public void crearSector(NSector sector)
Descripción	Crea un sector.
Nombre	public void modificarSector(NSector sector)
Descripción	Modifica un sector.
Nombre	public void eliminarSector(NSector sector)
Descripción	Intenta eliminar un sector, lanza una excepción si este todavía conserva alguna relación.
Nombre	public List<NSector> buscarSectores(int idparte)
Descripción	Devuelve una lista de sectores dada un parte.
Interfaz	public interface ConfigurarFuenteFinanciamientoDao extends BaseDao
Métodos	
Nombre	public List<NFuente_Financiamiento> buscarFuenteFinanciamiento()
Descripción	Devuelve una lista de fuentes de financiamientos.

Nombre	public void modificarFuenteFinanciamiento(NFuente_Financiamiento fuente)
Descripción	Modifica una fuente de financiamiento.
Nombre	public void crearFuenteFinanciamiento(NFuente_Financiamiento fuente)
Descripción	Crea una fuente de financiamiento.
Nombre	public void eliminarFuenteFinanciamiento(NFuente_Financiamiento fuente)
Descripción	Intenta eliminar una fuente de financiamiento, lanza una excepción si esta conserva todavía alguna relación.
Interfaz	public interface ConfigurarHonorariosDao extends BaseDao
Métodos	
Nombre	public List<NCategoria> buscarCategoriasHonorarios()
Descripción	Devuelve una lista de categorías con sus honorarios.
Nombre	public void modificarHonorarios(List<NCategoria> categorias)
Descripción	Modifica una lista de categorías con sus honorarios.
Interfaz	public interface ConfigurarModalidadProyectoDao extends BaseDao
Métodos	
Nombre	public List<NModalidad> buscarModalidad()
Descripción	Devuelve una lista de modalidades con sus honorarios.
Nombre	public void modificarModalidad(NModalidad modalidad)
Descripción	Modifica una modalidad.
Nombre	public void crearModalidad(NModalidad modalidad)
Descripción	Crea una modalidad.
Nombre	public void eliminarModalidad(NModalidad modalidad)
Descripción	Intenta eliminar una modalidad, lanza una excepción si este todavía conserva alguna relación.
Interfaz	public interface ConfigurarTasaCambioDao extends BaseDao
Métodos	
Nombre	public void crearTasaCambio(NTasa_Cambio tasa)

Descripción	Crea una tasa de cambio y actualiza la tasa actual.
Nombre	public NTasa_Cambio buscarTasaCambio()
Descripción	Devuelve la tasa de cambio actual.
Interfaz	public interface ConfigurarTipoRecursoDao extends BaseDao
Métodos	
Nombre	public List<NTipo_Recurso> buscarTipoRecurso()
Descripción	Devuelve una lista de tipos de recursos.
Nombre	public void modificarTipoRecurso(NTipo_Recurso recurso)
Descripción	Modifica un tipo de recurso.
Nombre	public void crearTipoRecurso(NTipo_Recurso recurso)
Descripción	Crear un tipo de recurso.
Nombre	public void eliminarTipoRecurso(NTipo_Recurso recurso)
Descripción	Intenta eliminar un tipo de recuro, lanza una excepción si este todavía conserva alguna relación.
Interfaz	public interface ConfigurarTipoRecursoHumanoDao extends BaseDao
Métodos	
Nombre	public List<NCategoria> buscarRecursoHumano()
Descripción	Devuelve una lista de categorías.
Nombre	public void modificarRecursoHumano(NCategoria recurso)
Descripción	Modifica una categoría.
Nombre	public void crearRecursoHumano(NCategoria recurso)
Descripción	Crea una categoría.
Nombre	public void eliminarRecursoHumano(NCategoria recurso)
Descripción	Intenta eliminar una categoría, lanza una excepción si este todavía conserva alguna relación.
Interfaz	public interface ConfigurarViaticoDao extends BaseDao
Métodos	
Nombre	public List<NConfiguracion_Viaticos> buscarConfiguracion_Viaticos()
Descripción	Devuelve una lista de configuraciones de viáticos.

Nombre	public void modificarConfiguracion_Viaticos(NConfiguracion_Viaticos viaticos)
Descripción	Modifica una configuración de viáticos.
Nombre	public void crearConfiguracion_Viaticos(NConfiguracion_Viaticos viaticos)
Descripción	Crea una configuración de viáticos.
Nombre	public void eliminarConfiguracion_Viaticos(NConfiguracion_Viaticos viaticos)
Descripción	Intenta eliminar una configuración de viáticos.
Interfaz	public interface NotificacionesDao extends BaseDao<DNotificacion, Integer>
Métodos	
Nombre	public void notificacionPresentacion(String nombreDocumento , int operacion1, int tipoDocumento, int ef, int proceso1, int idUsuario1, Date fecha1, int enteCubano1, int enteVenezolano1)
Descripción	Ejecuta el procedimiento de la base de datos que crea una notificación de presentación de un proyecto.
Nombre	public void notificacionPresentacionIdDocumentoOperacion(int operacion1, int idDocumento1, int idUsuario1, Date fecha1)
Descripción	Ejecuta el procedimiento de la base de datos que crea una notificación de alguna operación sobre algún documento.
Nombre	public List<DNotificacion> buscarNotificacionesPag(int idNivel, int idUsuario, int primero, int cantidad)
Descripción	Devuelve una lista limitada de notificaciones dado un usuario y una visibilidad dada par el nivel.
Nombre	public int cantidadTotalNotificaciones(int idNivel, int idUsuario)
Descripción	Devuelve la cantidad total de notificaciones dado un usuario y una visibilidad dada par el nivel.
Nombre	public void eliminarNotificacion(int idNotificacion)

Descripción	Elimina una notificación.
Nombre	public void eliminarNotificacionNivel(int idNotificacion , int idNivel, int idUsuario)
Descripción	Elimina la visibilidad de una notificación dado un nivel.
Nombre	public List<String> CorreoUsuario(int idUsuario, int tipo)
Descripción	Devuelve el correo de un usuario.
Interfaz	public interface TrazasDao extends BaseDao<DTrazas, Integer>
Métodos	
Nombre	public List<DTrazas> buscarTrazas(String usuario,String persona,Date desde,Date hasta,boolean error,int primero,int cantidad)
Descripción	Devuelve una lista limitada de trazas dada un grupo de características.
Nombre	public int cantidadTotalTrazas(String usuario,String persona,Date desde,Date hasta,boolean error)
Descripción	Devuelve la cantidad total de trazas dado un grupo de características
Nombre	public void crearTrazas(DTrazas trazas)
Descripción	Crea una traza.

Estas son las interfaces que consume directamente el negocio y que permiten enmascarar las implementaciones, de las cuales se explicará su funcionamiento luego de abordar la interfaz BaseDao y su implementación BaseDaoImpl las cuales son vitales y constituyen la base de los demás DAOs.

```java
public interface BaseDao<DomainObject, KeyType extends Serializable> {

    public void crear(DomainObject object) throws DAOException;

    public void modificar(DomainObject object) throws DAOException;

    public void eliminar(DomainObject object) throws DAOException;

    public List<DomainObject> cargar(String query, String[] nombreparametros,
            Object[] valoresparametros) throws DAOException;

    public DomainObject cargarPorId(DomainObject object, KeyType id)
            throws DAOException;

    public DomainObject getById(DomainObject object, KeyType id);

}
```

En la interfaz BaseDao se encuentran declarados la mayoría de los métodos que pueden ser comunes para los demás DAOs además de que estos tienen los tipos de datos declarados de forma genérica para hacer que estos admitan cualquier "*value object*" por referencia.

```java
public class BaseDaoImpl<DomainObject, KeyType extends Serializable> extends
        HibernateDaoSupport implements BaseDao<DomainObject, KeyType> {

    public void crear(DomainObject object) throws DAOException {
    public void modificar(DomainObject object) throws DAOException {
    public void eliminar(DomainObject object) throws DAOException {
    public List<DomainObject> cargar(String query, String[] nombreparametros,
    public DomainObject cargarPorId(DomainObject object, KeyType id)
    public DomainObject getById(DomainObject object, KeyType id) {
}
```

En BaseDaoImpl hay que destacar la propiedad que nos permite realmente la integración con el *framework* Hibernate y esta es la extensión (heredar) de la clase que contiene Spring para permitir su integración con Hibernate "HibernateDaoSupport" la cual es la que contiene el *sessionFactory* tiene cargadas todas las propiedades que se necesitan como el "*connection*" los mapping etc. Además contiene el HibernateTemplate del cual se habló más detalladamente en el capitulo anterior.

A continuación se explicarán algunos de los ejemplos más ilustrativos de las implementaciones de los métodos de los DAOs.

```
public List<NFuncionalidad> Funcionalidades() throws Exception{

    List <NFuncionalidad> func = null;
    String query = "from NFuncionalidad";
    func = this.getHibernateTemplate().find(query);
    return func;
}
```

En este método se pueden ver una de las formas más simples de utilización del *framework* y en él se obtiene una lista de funcionalidades. Se utiliza el método "*find*" en su primera variante explicada anteriormente.

```
public NRol buscarRol(int rol) throws Exception{
    NRol roll = new NRol();
    List<NRol> roles = new ArrayList<NRol>();
    roles = this.getHibernateTemplate().
    findByNamedQueryAndNamedParam("rolfuncionalidad", "idrol1", rol);
    if(roles.size()> 0) roll = roles.get(0);else return null;
    return roll;
}
```

En este método se obtiene un solo rol o ninguno para esto se realizó una query que garantizara tal resultado, la misma se encuentra declarada dentro de un XML de mapeo y tiene parámetros que restringen la búsqueda por lo que se hiso necesario utilizar el método "*findByNamedQueryAndNmedParam*" explicado anteriormente.

```
public List<NRol> buscarRolesPag(int idparte, int primero, int cantidad)
throws Exception{
    List<NRol> rol = null;
    Session session = this.getSessionFactory().openSession();
        rol = (List<NRol>)session.getNamedQuery("rolesparte")
        .setParameter("idparte1", idparte)
        .setFirstResult(primero)
        .setMaxResults(cantidad)
        .list();
        session.close();
    return rol;
}
```

En este método se obtiene una lista de roles que se desea limitar la cantidad que se devuelve y además cual será la posición a partir de la cual se empezara a contar los resultados deseados. En este se recurrió a uno de las funciones propias del *framework* Hibernate. Primero se obtuvo la sesión para realizar las operaciones directamente sobre ella, luego se invoco el método "*getNamedQuery*" el cual permite cargar la query creada en un XML y sobre la instancia la clase "Query" la cual nos permite antes de ejecutarla con el método "*list*" de la misma asignarle los parámetros que vienen contenidos dentro de la query los valores de las propiedades "*firstResult*" (primer resultado) y "*maxResult*" (máximo resultado), luego se debe serrar la sesión ya que como se está trabajando directamente sobre ella el "*HibernateDaoSuport*" no tiene control directo sobre la misma.

```
public boolean eliminarRol(NRol rol) throws Exception {
    boolean respons = true;
    NRol roll = new NRol();
    List<NRol> roles = new ArrayList<NRol>();
    try {
        roles = this.getHibernateTemplate().
        findByNamedQueryAndNamedParam("rolid", "idrol1", rol.getIdRol());
        if(roles.size()> 0) roll = roles.get(0);else return false;
        if(roll!= null){
        for (Iterator i = roll.getUsuario().iterator(); i.hasNext();) {
            DUsuario user = (DUsuario) i.next();
            respons = false;
        }
        if(respons){
              this.getHibernateTemplate().delete(roll);
              return respons;
        }
        }
        return respons;
    } catch (Exception e) {
        throw e;
    }
}
```

En este método se va a eliminar un rol pero antes de eliminarlo se debe comprobar que este no está asignado a ningún usuario en caso contrario no se puede eliminar el mismo para ello primero se realiza una consulta en la que se obtiene el rol indicado de la base de datos luego se verifica que este no tenga ningún usuario asignado, en caso de que esto se cumpla se procede a eliminarlo en caso contrario no se elimina.

```
public void modificarRol(NRol rol) throws Exception {
    try {
            this.getHibernateTemplate().update(rol);
    } catch (Exception e) {
        throw e;
    }
}
```

En este método se modifica un rol, para esto se utiliza el método "*update*" del "HibernateTemplate".

```
public void crearEnte(DNivel nivel) throws Exception {
    this.getHibernateTemplate().save(nivel);
}
```

En este método se crea un rol, para esto se utiliza el método “*save*” del “HibernateTemplate”.

Conclusiones del Capitulo.

Al finalizar este capítulo ya se ha implementado la capa de acceso a datos de los módulos de administración y configuración que se había propuesto en un principio. En este se fueron abarcando todos los pasos fundamentales para su creación y se describieron cada uno de sus componentes. Se generaron y configuraron correctamente permitiendo una correcta correspondencia entre la base de datos y las cases de persistencia todos los *mappings*. Se utilizaron para ello los beneficios del *framework* Hibernate y las plantillas que nos ofrece el *framework* Spring para su integración al *framework* anterior permitiéndonos con esto aprovechar todas las facilidades de los mismos. Se logró cumplir con todos los estándares de Java se utilizó con éxito el patrón DAO facilitando así la implementación y organización del código.

CAPÍTULO 3: PRUEBAS DE UNIDAD

Introducción al capítulo.

En este capítulo se realizarán pruebas de unidad sobre los distintos métodos de la capa de persistencia de los módulos de administración y configuración con el objetivo de comprobar y validar la integridad y correcto funcionamiento de los mismos. Para ello se utilizará el *framework* de pruebas JUnit y se irán llenando las planillas que nos permiten realizar las mismas de forma metódica y ordenada.

Pruebas de Unidad con JUnit.

Para garantizar la fiabilidad de un componente de *software* el mismo debe ser sometido a pruebas que lo garanticen, estas se deben ir realizando a la par de la evolución del desarrollo para garantizar que los mismos no provoquen fayas en otros componentes y además se deben ir corrigiendo a medida que aparezcan. Es importante resaltar que las pruebas no garantizan que no existan errores pero siempre y cuando se verifiquen detalladamente todos los componentes del sistema y cada una de las capas garantice la integridad de las salidas de sus componentes se disminuirá grandemente la aparición posterior de fayas en el sistema.

En la realización de estas pruebas y para garantizar que las mismas se realicen de una forma homogénea y especializada es donde entra la utilización de JUnit, el cual es un *framework* de pruebas que nos permite la realización de forma rápida y sencilla de pruebas a nuestro sistema. El mismo existe para varios lenguajes no solo para Java y en estos momentos el mismo goza de gran preferencia entre los programadores por su eficacia y sencillez.

A continuación primeramente se explicará la manera en que se debe configurar el *framework* y la manera en que el mismo muestra los resultados de la pruebas en las que no basaremos para llenar las plantillas con los mismos. Las pruebas que se realizarán a continuación serán sobre la capa de persistencia

específicamente sobre los métodos de los DAOs y se buscar comprobar que los mismos realizan las funcionalidades para los que fueron concebidos.

Por experiencia de otros compañeros anteriormente y por experiencia propia a lo largo de la interacción con el *framework* conocemos que con el mismo en el caso de la capa de persistencia que aunque las pruebas arrojen resultados positivos no se puede asegurar el buen funcionamiento del mismo el programador tiene que por su parte verificar los resultados de las modificaciones en la base de datos ya que el mismo no realiza esta funcionalidades y así completar y corroborar el resultados de las pruebas.
La utilización del *framework* comienza con la elaboración de la clase donde se implementaran los casos de prueba la cual se muestra a continuación.

```
public class AdministracionTest extends TestCase
```

Se crea la clase heredando de la "*TestCase*" lo que garantiza que el *framework* compruebe nuestra clase.

```
private ApplicationContext factory = null;
private ConfigurarTazaCambioDao
    configurarTazaCambioDao = null;
private ConfigurarFuenteFinanciamientoDao
    fuenteFinanciamientoDao = null;
.
.
.

public AdministracionTest(String name) {
    super(name);
    String[] contexts = new String[] {
            "common-applicationContext-dataSource.xml"
            ,"common-applicationContext-dao.xml"
            ,"common-applicationContext-properties.xml"
            ,"administracion-applicationContext-dao.xml" };
    factory = new ClassPathXmlApplicationContext(contexts);
}
```

A continuación declaramos variables como una instancia del "*AplicationContext*" el cual nos servirá para inicializar las funcionalidades de Spring e Hibernate realizando la carga inicial de los *mappings* y los XMLs

de los *beans* con los que trabajaremos posteriormente, seguidos por instancias de los DAOs sobre los que realizaremos pruebas. La instancia del "*AplicationContext*" se inicializa en el constructor, para lo cual utilizamos la implementación "*ClassPathXmlApplicationContext*" el cual nos permite pasarle la lista de ubicaciones de los XMLs a partir de los cuales se construirá.

```
@Before
protected void setUp() throws Exception {
    super.setUp();
    configurarTazaCambioDao =
        (ConfigurarTazaCambioDao) factory
        .getBean("tazaDeCambioDao");
    fuenteFinanciamientoDao =
        (ConfigurarFuenteFinanciamientoDao) factory
        .getBean("fuenteDeFinanciamientoDao");
    .
    .
    .
    .

}
```

Se sobrescribe el método "*setUp*" el cual se ejecuta antes de los métodos de prueba y en el cual le inyectamos los beans a las instancias de los DAOs los cual nos permite inicializarlos.

```
@After
protected void tearDown() throws Exception {
    super.tearDown();
    configurarTazaCambioDao = null;
    fuenteFinanciamientoDao = null;
    .
    .
    .
    .
}
```

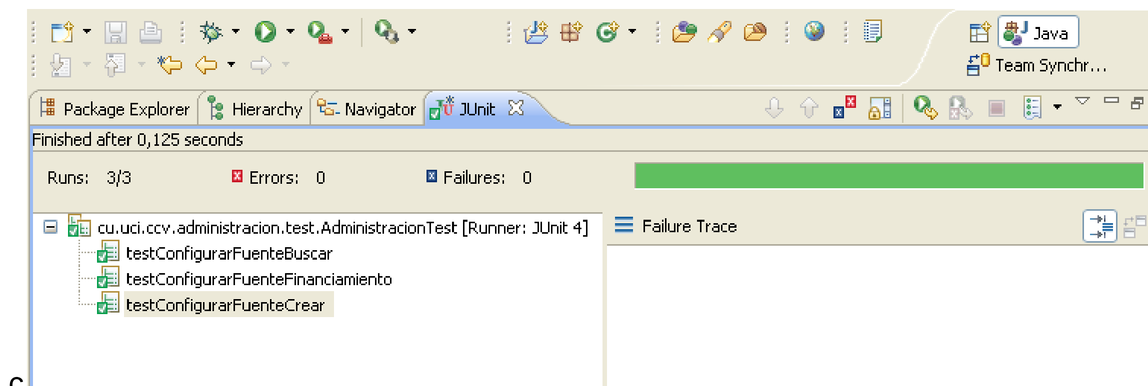
Se sobrescribe el método "*tearDown*" el cual se ejecuta después de los métodos de prueba y con el cual se liberan los recursos.

```
public void testXXX() throws Exception {
      .
      .
      .
}
```

Se crean los casos de prueba para lo cual tenemos que crear nuestros métodos iniciando su nombre con la palabra test seguido del nombre del caso de prueba y para lo cual el *framework* nos brinda entre otros muchos los siguientes métodos:

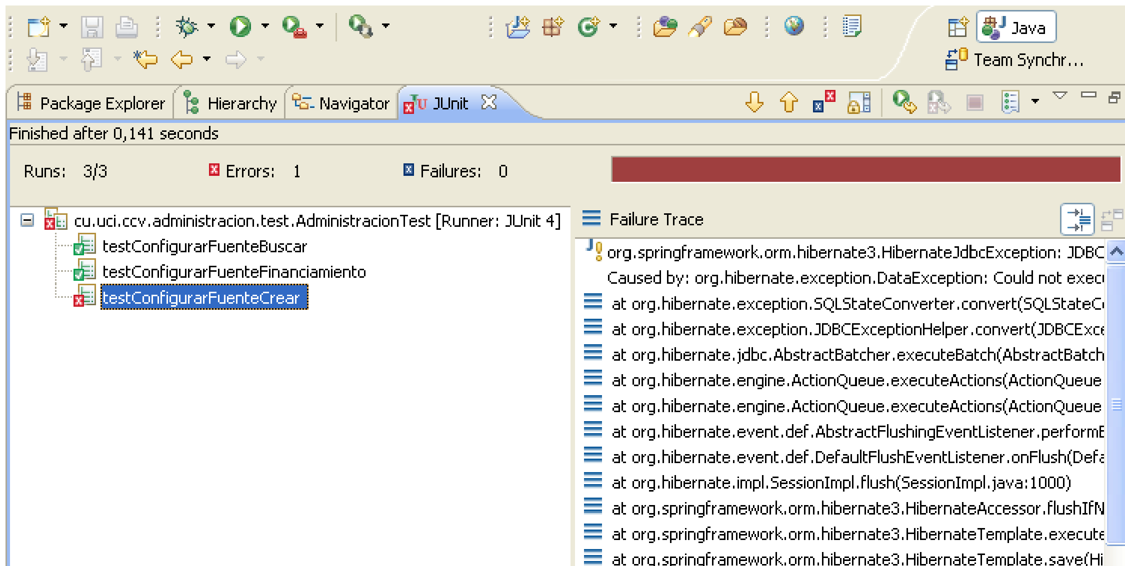
Método	Descripción
assertTrue	Acepta el resultado si la condición es verdadera
assertFalse	Acepta el resultado si la condición es falsa
assertEquals	Acepta el resultado si los objetos son iguales
asserNotNull	Acepta el resultado si el objeto no es null
assertNull	Acepta el resultado si el objeto es null
asserSame	Acepta el resultado si la dos referencias apuntan al mismo objeto
assertNotSame	Acepta el resultado si las dos referencias no apuntan al mismo objeto

A continuación se mostraran algunos ejemplos de cómo el *framework* visualiza los resultados de las pruebas.



c

Resultado positivo.



Resultado negativo.

Casos de Prueba.

La siguiente planilla recoge los principales pasos y aspectos que se tuvieron en cuenta a la hora de de realizar las pruebas sobre los distintos métodos de la capa de persistencia de los módulos de administración y configuración. Específicamente se incluyeron los referentes a los casos de uso gestionar roles, gestionar usuarios y gestionar personas que constituyen los pilares principales del modulo.

Caso de Prueba	GestionarRolDao	
Método	funcionalidades	
Condiciones		
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se Obtiene una lista de funcionalidades almacenadas en la base de datos	Se esperan obtener una lista con todas las funcionalidades almacenadas en la base de datos	Se obtuvo la lista satisfactoriamente La lista contenía todas las funcionalidades
Método	BuscarRolesPag	
Condiciones	Se deben especificar la cantidad de roles a buscar la “parte” de la cual se están buscando	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se Obtiene una lista de roles almacenadas en la base de datos	Se espera un listado dividido en porciones de los roles almacenados en la base de datos	Se obtuvo la lista satisfactoriamente La lista contenía solo la cantidad de elementos especificados en el paginado
Método	cantidadTotalRoles	
Condiciones	Se debe especificar la “parte” de la cual se está buscando	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se obtiene una cantidad real de los roles que existen en la base de datos	Se espera la cantidad real de roles que existe en la base de datos	Se obtuvo la cantidad real de roles que existían en la base de datos
Método	buscarTiposNiveles	
Condiciones		

Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se obtiene un listado con todos los tipos de nivel almacenados en la base de datos	Se esperan obtener una lista con todos los tipos de nivel almacenadas en la base de datos	Se obtuvo la lista satisfactoriamente La lista contenía todos los tipos de nivel almacenados en la base de datos
Método	eliminarRol	
Condiciones	Se debe especificar el rol que se desea eliminar	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se elimina exitosamente un rol de la base de datos	Se espera eliminar un rol especifico de la base de datos	Se elimino el rol de la base de datos
Método	modificarRol	
Condiciones	Se debe brindar un rol con todos sus tributos y relaciones con otras clases especificadas	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se modifica el rol especifico en la base de datos	Se espera que se modifiquen los atributos un rol especifico en la base de datos	Se modificaron los atributos del rol en la base de datos
Método	buscaRol	
Condiciones	Se debe especificar el rol que se desea buscar	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se obtiene el rol correcto de la base de datos	Se espera obtener un rol especifico de la base de datos	Se obtuvo el rol indicado de la base de datos

Caso de Prueba	GestionarUsuarioDao	
Método	buscarUsuariosDePersona	
Condiciones	Se debe especificar la persona de la que se desean obtener los usuarios	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se obtiene un listado de los usuarios de una persona en especifica de la base de datos	Se espera obtener una lista con todos los usuarios de una persona especifica guardados en la base de datos	Se obtuvo la lista satisfactoriamente La lista contenía todos los usuarios de la persona indicada que existían en la base de datos.
Método	buscarUsuariosPag	
Condiciones	Se deben especificar o pasar los valores por defecto en caso de que no se desee incluir en el filtro el nombre de usuario nombre o apellidos de la persona tipos de nivel ministerio ente tipo de nivel del los usuario y además un usuario que debe ser el registrado en la aplicación con su parte nivel y tipo de nivel.	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se obtiene un listado de usuarios de la base de datos	Se espera obtener un listado dividido en porciones de los usuarios que se encuentran en la base de datos y cumplen con las condiciones	Se obtuvo la lista satisfactoriamente La lista contenía todos los usuario que cumplían con las condiciones
Método	cantidadTotalUsuarios	
Condiciones	Se deben especificar o pasar los valores por defecto en caso de que no se desee incluir en el filtro el nombre de usuario nombre o apellidos de la persona tipos de nivel ministerio	

	ente tipo de nivel del los usuario y además un usuario que debe ser el registrado en la aplicación con su parte nivel y tipo de nivel.	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se obtiene una cantidad real de los usuarios que cumplen las condiciones y están en la base de datos	Se espera la cantidad real de usuarios que existen en la base de datos y cumplen con las condiciones	Se obtuvo la cantidad real de usuarios que existían en la base de datos y cumplían con las condiciones.
Método	buscarMinisterios	
Condiciones	Se deben especificar la parte y el tipo de nivel	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se obtiene una lista de ministerios almacenados en la base de datos	Se espera obtener una lista con todos los ministerios que estén en la base de datos y cumplan con la condición	La lista contenía todos los ministerios almacenados en la base de datos y que cumplían con las condiciones
Método	buscarEntes	
Condiciones	Se debe especificar el ministerio al que pertenece	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se obtiene una lista de entes almacenados en la base de datos	Se espera obtener una lista con todos los entes que estén en la base de datos y cumplan con la condición	La lista contenía todos los entes almacenados en la base de datos y que cumplían con las condiciones
Método	crearUsuario	
Condiciones	Se necesita una instancia de un usuario deben especificar además nivel y parte	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba

Comprobar que se crea el usuario especifico en la base de datos	Se espera que se cree un nuevo usuario con los datos especificados en la base de datos	Se creó un nuevo usuario en la base de datos con los datos especificados
Método	eliminarUsuario	
Condiciones	Se debe especificar el usuario que se desea eliminar	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se elimina exitosamente un usuario de la base de datos	Se espera eliminar un usuario especifico de la base de datos	Se elimino el usuario de la base de datos
Método	modificarUsuario	
Condiciones	Se debe especificar el usuario que se desea modificar	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se modifica el usuario especifico en la base de datos	Se espera que se modifiquen los atributos de un usuario especifico en la base de datos	Se modificaron los atributos del usuario en la base de datos
Método	verificarUsuarioCrear	
Condiciones	Se debe especificar el nombre de usuario.	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se busca un nombre de usuario en especifico	Se espera que se encuentre en caso de que exista un nombre de usuario en especifico	Se verifico con éxito la existencia de un usuario en la base de datos
Método	verificarUsuarioModificar	
Condiciones	Se debe especificar el nombre de usuario y su identificador.	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se busca un nombre de usuario en	Se espera que se encuentre en caso de que exista un	Se verifico con éxito la existencia de un usuario en

especifico	nombre de usuario en especifico y que se excluya el usuario sobre el que se realiza la búsqueda	la base de datos y el mismo no se tuvo en cuenta cuando este pertenecía al usuario sobre el que se realizaba la búsqueda
Método	buscarRolesSistema	
Condiciones	Se debe especificar la parte y el tipo de nivel.	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se obtiene una lista de roles del sistema almacenados en la base de datos	Se espera obtener una lista con todos los roles que cumplan con las condiciones que están almacenados en la base de datos	La lista contenía todos los roles almacenados en la base de datos que cumplían con las condiciones
Método	buscarRolesUsuario	
Condiciones	Se debe especificar el usuario y el tipo de nivel	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se obtiene una lista de roles almacenados en la base de datos	Se espera obtener una lista con todos los roles que cumplan con las condiciones que están almacenados en la base de datos	La lista contenía todos los roles almacenados en la base de datos que cumplían con las condiciones
Caso de Prueba	GestionarPersonaDao	
Método	buscarPersonasPag	
Condiciones	Se necesita una instancia de persona que contenga datos como el nombre, primer y segundo apellido, CI, correo y pasaporte	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se obtiene una lista de personas	Se espera obtener un listado dividido en porciones con	Se obtuvo la lista satisfactoriamente y los

almacenados en la base de datos	todos los personas que están en la base de datos que cumplan con las condiciones	elementos cumplían con las condiciones La lista contenía solo la cantidad de elementos especificados en el paginado
Método	cantidadTotalPersonas	
Condiciones	Se necesita una instancia de persona que contenga datos como el nombre, primer y segundo apellido, CI, correo y pasaporte	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se obtiene una cantidad real de las personas que existen en la base de datos	Se espera la cantidad real de personas que existe en la base de datos y que además cumplen con las condiciones	Se obtuvo la cantidad real de personas que existían en la base de datos y que además cumplen con las condiciones
Método	crearPersona	
Condiciones	Se necesita una instancia de persona con todos sus datos	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se crea la persona especifico en la base de datos	Se espera que se cree una nueva persona con los datos especificados en la base de datos base de datos	Se creó una nueva persona en la base de datos con los datos especificados
Método	verificarCIC	
Condiciones	Se debe especificar el CI	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se busca un CI especifico en la base de datos	Se espera que se encuentre en caso de que exista un CI especifico	Se verifico con éxito la existencia de un CI ya existente en la base de

		datos
Método	verificarCorreoC	
Condiciones	Se debe especificar el correo	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se busca un correo especifico en la base de datos	Se espera que se encuentre en caso de que exista un correo especifico	Se verifico con éxito la existencia de un correo ya existente en la base de datos
Método	verificarPasaporteC	
Condiciones	Se debe especificar el pasaporte	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se busca un pasaporte especifico en la base de datos	Se espera que se encuentre en caso de que exista un pasaporte especifico	Se verifico con éxito la existencia de un pasaporte ya existente en la base de datos
Método	verificarCIM	
Condiciones	Se debe especificar el CI y el identificador de la persona	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se busca un CI especifico en la base de datos	Se espera que se encuentre en caso de que exista un CI especifico y que se excluya la persona sobre el que se realiza la búsqueda	Se verifico con éxito la existencia de un CI ya existente en la base de datos y el mismo no se tuvo en cuenta cuando este pertenecía a la persona sobre la que se realizaba la búsqueda
Método	verificarCorreoM	
Condiciones	Se debe especificar el correo y el identificador de la persona	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba

Comprobar que se busca un correo especifico en la base de datos	Se espera que se encuentre en caso de que exista un correo especifico y que se excluya la persona sobre el que se realiza la búsqueda	Se verifico con éxito la existencia de un correo ya existente en la base de datos y el mismo no se tuvo en cuenta cuando este pertenecía a la persona sobre la que se realizaba la búsqueda
Método	verificarPasaporteM	
Condiciones	Se debe especificar el pasaporte y el identificador de la persona	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se busca un pasaporte especifico en la base de datos	Se espera que se encuentre en caso de que exista un pasaporte especifico y que se excluya la persona sobre el que se realiza la búsqueda	Se verifico con éxito la existencia de un pasaporte ya existente en la base de datos y el mismo no se tuvo en cuenta cuando este pertenecía a la persona sobre la que se realizaba la búsqueda
Método	eliminarPersona	
Condiciones	Se necesita una instancia de persona que contenga su identificador	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se elimina exitosamente una persona de la base de datos	Se espera eliminar una persona especifica de la base de datos	Se elimino la persona de la base de datos
Método	modificarPersona	
Condiciones	Se necesita una instancia de persona que contenga su	

	identificador	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se modifica la persona especifica en la base de datos	Se espera que se modifiquen los atributos de una persona especifica en la base de datos	Se modificaron los atributos de la persona en la base de datos
Método	buscarPersona	
Condiciones	Se necesita una instancia de persona que contenga su identificador	
Objetivo de la Prueba	Resultados Esperados	Resultado de la Prueba
Comprobar que se obtiene la persona especifica de la base de datos	Se espera obtener una persona especifica de la base de datos	Se obtuvo la persona indicada con todos sus atributos.

Ejemplos de pruebas.

Método “crearUsuario”

```
DUsuario usuario = new DUsuario();
usuario.setActivo(true);
usuario.setNombreUsuario("usuarioprueba");
DPersona persona = new DPersona();
persona.setIdPersona(2);

usuario.setPersona(persona);
NRol rol = new NRol();
rol.setIdRol(2);
Set<NRol> set = new HashSet<NRol>();
set.add(rol);
usuario.setRol(set);

usuario.setContrasena("asdasdasdasdasdasdasdasd");

gestionarUsuarioDao.crearUsuario(usuario, 2, 1);
```

De lo que se obtuvo el siguiente resultado:

idusuario	nivel	persona	nombreusuario	contrasena	activo
685	2	2	usuarioprueba	asdasdasdasdasdasdasdasd	✓
684	174	700	jluis	651b8d8478a096e46d997953f57d323dd03e92ef	✓
683	281	60	carlacarolina	f8a8bcfac087bfe2c08ca34d2660349aead32fb4	✓
682	1	699	juancarlos	fd903d84e564a8c52888531414a6957fe0613672	✓
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	✓

En esta imagen se puede ver la nueva tupla creada con los valores deseados.

Método “modificarUsuario”

```
DUsuario usuario = (DUsuario) gestionarUsuarioDao.getById(new DUsuario(), 685);
usuario.setContrasena("qweqweqweqweqweqweqweqweqwe");
NRol rol = new NRol();
rol.setIdRol(4);
Set<NRol> set = new HashSet<NRol>();
set.add(rol);
usuario.setRol(set);
gestionarUsuarioDao.modificarUsuario(usuario, 2, 1);
```

De lo que se obtuvo el siguiente resultado:

idusuario	nivel	persona	nombreusuario	contrasena	activo
685	2	2	usuarioprueba	qweqweqweqweqweqweqweqweqwe	✓
684	174	700	jluis	651b8d8478a096e46d997953f57d323dd03e92ef	✓
683	281	60	carlacarolina	f8a8bcfac087bfe2c08ca34d2660349aead32fb4	✓

usuai	rol
685	3
684	2
684	10
684	25
684	28

En estas imágenes se pueden ver las tuplas modificadas con los valores deseados.

Método “eliminarUsuario”

```
DUsuario usuario = (DUsuario) gestionarUsuarioDao.getById(new DUsuario(), 685);
boolean aux = gestionarUsuarioDao.eliminarUsuario(usuario);
```

De lo que se obtuvo el siguiente resultado:

idusuario	nivel	persona	nombreusuario	contrasena	activo
684	174	700	jluis	651b8d8478a096e46d997953f57d323dd03e92ef	✓
683	281	60	carlacarolina	f8a8bcfac087bfe2c08ca34d2660349aead32fb4	✓
682	1	699	juancarlos	fd903d84e564a8c52888531414a6957fe0613672	✓
681	2	582	dpardo	03eb7310fd222420d13873ab4df820795a7f1bd7	✓

En esta imagen se puede ver como la tupla fue eliminada.

Método “buscarUsuariosDePersona”.

```
List<DUsuario> usuarios =  gestionarUsuarioDao.buscarUsuariosDePersona(671);
for (DUsuario usuario : usuarios) {
    System.out.println(usuario.getUsername() + "  \t\t" + usuario.getContrasena());
}
```

Lo cual debía obtener los siguientes resultados.

idusuario	nivel	persona	nombreusuario	contrasena	activo
672	349	672	campismo_ministerio	5883f89844e96a1fec1cf686dcd637803f1a72c8	✓
661	347	671	arley	188282149eac95abf20c43e113601931b3b63e09	✓
667	354	671	hermanos_sainz	59d4bddd1d279b7e338b23e7c74047b4be6107f4	✓
668	351	671	opjm	01c343093238381f4ee18730a94f7c002f83359a	✓
673	352	671	opjm_ministerio	21e1ea673a85a2846ce97c67d42dfbdc2418614c	✓
674	353	671	ahs_min	49902be2d42f74f6ebded0f6914dc75f9241706e	✓
675	349	671	campismo_min	8c62e60ad2c222aedc316dbf22b61a967ac5cc65	✓
676	346	671	cesj_min	383f8a2523d7d7ef1ec441bde322f047ea787613	✓
669	346	670	dcesj_ministerio	a60f039ada9b6eb578b114441cb1ec9759c0d8c0	✓

Se obtuvo el siguiente resultado.

```
arley             188282149eac95abf20c43e113601931b3b63e09
ahs_min                   49902be2d42f74f6ebded0f6914dc75f92417O6e
cesj_min                  383f8a2523d7d7ef1ec441bde322f047ea787613
opjm_ministerio                   21e1ea673a85a2846ce97c67d42dfbdc2418614c
campismo_min              8c62e60ad2c222aedc316dbf22b61a967ac5cc65
hermanos_sainz                    59d4bddd1d279b7e338b23e7c74047b4be6107f4
opjm              01c343093238381f4ee18730a94f7c002f83359a
```

En esta imagen se pueden ver como coinciden todos los resultados con los almacenados en la base de datos.

Método “buscarPersonasPag”.

```
DPersona persona = new DPersona();
persona.setPrimerNombre("al");
persona.setPrimerApellido("a");
persona.setSegundoApellido("");
persona.setCi("");
persona.setCorreo("");
persona.setPasaporte("");
List<DPersona> personas = gestionarPersonaDao.buscarPersonasPag(persona, 10, 5);
for (DPersona persona2 : personas) {
    System.out.println(persona2.getPrimerNombre()+ " "+persona2.getPrimerApellido()
            + " " + persona2.getCi());
}
```

Delo que se obtuvo el siguiente resultado

```
Luis Alfredo  Ramos Figueredo V5150193
Carlos Alberto Blanco Freeman 68090132627
Jesus Salvador  Ramos Diaz 12186184
Oscar Valentin Acosta Hernández 53120100789
Valery Blanco v14869023
```

En este caso son importantes peculiaridades como la cantidad y el primero de los resultados posibles que se obtuvo además del de que el nombre y los apellidos contengan las subcadenas y que se tomen todos los demás parámetros como que cualquier valor posible es admitido.

Conclusiones del capítulo.

Después de la realización de todas las pruebas sobre los métodos de la capa de persistencia se logró una validación de los mismos garantizando su estabilidad y correcto funcionamiento. No hay dudas de que la utilización del *framework* JUnit y la normalización de las pruebas mediante la utilización de una planilla agilizaron en gran medida el proceso.

CONCLUSIONES

Se realizó un estudio del estado del arte sobretodo asiendo un mayor enfoque en las técnicas de programación orientadas a objetos utilizando el lenguaje Java específicamente, junto a esto se estudiaron también los *frameworks* de desarrollo Hibernate y Spring sobretodo su integración con el *framework* anterior. A partir de este estudio se pudieron cumplir los objetivos y tareas especificas y se llegó a los siguientes resultados:

A partir de los modelos de diseño clases persistentes propuestos se genero la base de datos y las clases persistentes además de los mapeos que especifican la correspondencia entre ambos.

Se configuraron todos los XML de la aplicación permitiendo la conexión con éxito a la base de datos y la correcta integración de la aplicación al *framework* Spring y la configuración para el correcto y óptimo funcionamiento del *framework* Hibernate.

A partir del modelo de diseño de clases propuesto se implementaron los componentes que encapsulan las operaciones CRUD para persistir objetos de clases entidades específicas.

Se probaron todos los métodos implementados para la persistencia de objetos, utilizando el *framework* JUnit para realizar pruebas de unidad.

BIBLIOGRAFÍA

1. **Programación en castellano.** Programación en castellano. *La Nueva Metodología.* [En línea] 2006. http://www.programacion.com/tutorial/nuevametodologia/1/.

2. **Jacobson, Ivar, Booch, Grady y Rumbaugh, James.** *El Proceso Unificado de Desarrollo de Software.* Madrid : PEARSON EDUCACIÓN, 1999. 84-7829-036-2.

3. Teleformación. *Historia de la Informática.* [En línea] http://teleformacion.uci.cu/file.php/258/Bibliografia_Basica/Tema_1/Que_son_los_paradigmas.pdf.

4. **Beserril, Francisco.** *Java a su alcanse.* Mexico : The McGraw-Hill Companies, Inc., 1998.

5. Manual de Java. *Características de Java.* [En línea] http://www.manual-java.com/manualjava/caracteristicas-java.html.

6. **Lallana, Vicent-Ramon Palasí.** Programación en castellano. *Motores de Persistencia.* [En línea] 2006. H:\DATA\Docencia\Bibliografia\Programación en castellano_ Motores de Persistencia programacion gratis.htm.

7. **González, Héctor Suárez.** Java Hispano. *Manual Hibernate.* [En línea] 2007. http://www.javahispano.org/contenidos.item.action?id=1082&menuId=ARTICLES.

8. **Hanson, Jeff.** Programación en castellano. *Persistencia de Objetos Java utilizando Hibernate.* [En línea] 2006. H:\DATA\Docencia\Bibliografia\Programación en castellano_ Persistencia de Objetos Java utilizando Hibernate programacion gratis.htm.

9. **Red Hat Middleware.** JBoss Comunity. *Hibernate Reference Documentation.* [En línea] 2007. http://docs.jboss.org/hibernate/stable/core/reference/en/html/.

10. **Medín Piñeiro, Juan y García Figueras, Antonio.** *Hacia una arquitectura con JavaServer Faces, Spring, Hibernate y otros frameworks.* Sevilla : s.n., 2006.

11. **WALLS, CRAIG y Breidenbach, Ryan.** *Spring in Action.* s.l. : MANNING, 2008.

12. **Lago, Ramiro.** proactiva-calidad. *Patrones de diseño software.* [En línea] Abril de 2007. http://www.proactiva-calidad.com/java/patrones/index.html.

13. **BAUER, CHRISTIAN y KING, GAVIN.** *Java Persistence with Hibernate,REVISED EDITION OF HIBERNATE IN ACTION.* New York, NY 1002 : MANNING, 2007.

14. **Peak, Patrick y Heudecker, Nick.** *Hibernate Quickly.* s.l. : MANNING, 2006.

15. **MASSOL, VINCENT y HUSTED, TED.** *JUnit in Action.* s.l. : MANNI NG, 2006.

GLOSARIO DE TÉRMINOS

Mixta: Reuniones que se realizan anualmente entre los países Cuba y Venezuela en el cual se acuerdan los presupuestos y los proyectos que serán impulsados durante ese periodo.

Personal Software Process: Es una metodología que se encarga de medir y estimar principalmente los tiempos de desarrollo.

Frameworks: En el desarrollo de *software*, un *framework* es una estructura de soporte definida en la cual otro proyecto de *software* puede ser organizado y desarrollado. Típicamente, un *framework* puede incluir soporte de programas, bibliotecas y un lenguaje interpretado entre otros *software* para ayudar a desarrollar y unir los diferentes componentes de un proyecto.

Hibernate: es una herramienta de Mapeo objeto-relacional para la plataforma Java que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) que permiten establecer estas relaciones.

Spring: El Spring *Framework* (también conocido simplemente como Spring) es un *framework* de integración que se utiliza para el desarrollo de aplicaciones para la plataforma Java.

Proyecto Convenio Integral de Cooperación Cuba Venezuela: En el año 2000 el Presidente de Cuba Comandante en Jefe Fidel Castro y el Presiente de Venezuela Comandante Hugo Rafael Chávez Frías firmaron en Caracas el Convenio Integral de Cooperación Cuba - Venezuela, mediante el cual los gobiernos de ambas naciones se comprometieron a elaborar de común acuerdo, programas y proyectos de cooperación.

Java: Java es toda una tecnología orientada al desarrollo de *software* con el cual podemos realizar cualquier tipo de programa. La tecnología Java está compuesta básicamente por 2 elementos: el lenguaje Java y su plataforma. Con plataforma nos referimos a la máquina virtual de Java (Java Virtual Machine).

Visual Paradigm: es una herramienta UML profesional que soporta el ciclo de vida completo del desarrollo de *software*: análisis y diseño orientados a objetos, construcción, pruebas y despliegue. El *software* de modelado UML ayuda a una más rápida construcción de aplicaciones de calidad, mejores y a un menor coste. Permite dibujar todos los tipos de diagramas de clases, código inverso, generar código desde

diagramas y generar documentación. La herramienta UML CASE también proporciona abundantes tutoriales de UML, demostraciones interactivas de UML y proyectos UML.

Java 2 Enterprise Edition: Es una plataforma diseñada para la computación mainframe, escala típica de las grandes empresas. Diseñada para simplificar el desarrollo de aplicaciones en un cliente ligero en niveles medio ambiente.

Persistencia: Es uno de los conceptos más importantes in el desarrollo de aplicaciones. Se refiere a la capacidad de estas de preservar la información de sus datos incluso cuando hayan sido cerradas.