Desarrollo de un framework de Persistencia para tecnología Java: WayPersistence ORM

Jorge Carneiro Denis

Ingeniería Informática

Josep María Camps Riva

14 de Enero de 2008

I. AGRADECIMIENTOS

A mi padre Manuel Carneiro, por ser de las personas que más me escucha y por haber confiado en mí desde el principio. A mi madre, Ana Denis, por su apoyo, dedicación, confianza durante toda mi vida, única persona por la que pondría la mano en el fuego.

A Carliña Portela, mi novia, por su paciencia, cariño, atención y ánimo tanto en la carrera como en mis últimos años.

Sin ellos 3, esto no sería un hecho y no lograría llegar a donde estoy tanto a nivel personal como profesional.

A mis hermanos, Ana y Manuel por estar ahí siempre y ayudarme en todo lo que pueden, a mi hermana que tiene un corazón de oro ya mi hermano que siempre una visión distinta de las cosas.

A la familia Portela Táboas, por sus opiniones, charlas, cafés, cenas y demás.

A Pedriño, mi mejor amigo, por su amistad, por siempre estar ahí, aunque ahora esté en Los Ángeles, gracias.

A amigos como Ángel Iglesias, Jorge Giráldez, Fernando Rodríguez que siempre me preguntan y se interesan por mí.

A los compañeros de trabajo que me han aportado ideas, opiniones y consejos, principalmente a Rubén, Alberto y Miguel Callón.

Y a los profesores de la universidad, que han hecho posible alcanzar un nivel de conocimientos suficiente para poder realizar este proyecto con todas las garantías.

A todos ellos, muchas gracias, os dedico este proyecto y os agradezco todo vuestro apoyo de corazón, porque sin vosotros, nada sería igual.

No sueñes tu vida, vive tú sueño.

II. RESUMEN

La implementación de un framework de persistencia para la tecnología Java requiere de un estudio previo de los distintos motores de persistencia en la actualidad.

Para realizar un motor de persistencia en condiciones, primeramente, debemos saber los requisitos mínimos que queremos implementar en nuestro framework porque depende de las características y el soporte que queramos dar a nivel persistencia para hacer un framework de mayor o menor magnitud.

Las características imprescindibles para cualquier framework desarrollado de forma personalizada son: escalabilidad, multiplataforma, adaptabilidad a cualquier sistema gestor de base de datos, válido para cualquier tipo de base de datos (relacional, en red, jerárquica, etc.) y fácil manejo y simplicidad para el equipo de desarrollo. En nuestro "WayPersistence v1.0" está validado para base de datos relacionales en un principio, mejorable en versiones posteriores.

Un framework de persistencia debe asegurar el mapeo entre el modelo de datos relacional y el modelo de datos orientado a objetos. Puesto que la mayor parte de las bases de datos de hoy en día son relacionales, WayPersistence está orientado a mapear este tipo de base de datos.

Para el mapeo con este ORM, tratamos de hacer una simetría entre el modelo de datos y el modelo de objetos de forma que una tabla sea una clase, una columna - una propiedad, una tupla o registro - una instancia, una relación entre tablas, una referencia a un objeto. Estos son algunas de las particularidades que ofrece
WayPersistence.

El motor de persistencia WayPersistence se engloba en varios paquetes. Un directorio principal, dataset, de donde cuelgan la mayor parte de los paquetes con las distintas funcionalidades implementadas y desarrolladas en clases java.

Dentro de estos paquetes tenemos: dao, paquete com.way.persistence.dao donde colocamos las clases de prueba o registros que serán mapeados en consonancia con el modelo relacional de la base de datos. En este caso, tenemos Role.java, Empleado.java, etc. que son clases java para mapear en relación con la base de datos definida en el sistema gestor elegido, por defecto en el framework, MySQL.

El paquete database, com.way.persitence.database posee el mayor número de clases desarrolladas con todos los tipos de datos utilizados por el framework y personalizados en cada caso: CampoEntero, CampoCadena, CampoHora, etc. Este tipo de objetos son subclases de ObjetoRaiz que a su vez extiende de MetaDato. Objetoraíz es la clase que almacena valores del tipo CampoCadena,
CampoLong, CampoEntero. La clase MetaDato instancia objetos que serán utilizados en RegistroInstancia. Complementando a estas dos clases genéricas, tenemos a RegistroInstancia que representa un registro individual en memoria que corresponde a una fila de la base de datos o un registro insertado. Y RegistroMetaDato representa los metadatos relacionados con el RegistroInstancia, nombre de la tabla y de los campos, en este caso son metadatos referenciados desde esta clase a la clase descrita en líneas anteriores, MetaDato.

La clase abstracta SessionImpl es utilizada en este ORM para unir la sesión con el dataset y las instancias de los registros. WayQuery es la principal clase de consultas y permite crear instancias de WayQuery y actualizar cada una de los métodos, basado en el lenguaje SQL.

La clase Referencia representa las claves foráneas de un registro a otro en una base de datos. Normalmente se especifican de una forma similar a la definición e foreign keys como CONSTRAINT en una base de datos. En relación a los CONSTRAINT también tenemos una clase CampoConstraint donde definimos las características de los campos representados en la base de datos tales como: clave primaria, no nulo, bloqueos, etc.

Como complementos de esta sección del framework, database, tenemos ModoSelect, ModoGenerator, ModoConsultas que nos permiten hacer filtrados y personalizar cada una de las consultas en función del tipo de consulta, del filtrado de los campos, así como el modo de las mismas.

Además, en database tenemos DataSet, una de las clases más importantes de este paquete del framework. Su principal acometido es almacenar los registros de varios tipos juntos con sus metadatos correspondientes. Estos registros son indexados con su base de datos correspondiente a través de su clave primaria. El Dataset suele asociarse con una sesión jdbc y accedemos al mismo a través de una clase que definiremos más adelante denominada sessionJdbc.

Otro de los paquetes definidos en el framework es com.way.persistence.database.validation que posee cuatro clases genéricas de validación para campo nulo, máxima longitud, mayor o igual y una master para validaciones más generales. Este paquete será mejorado en futuras versiones para darle más solvencia y fortaleza al ORM en lo que a validaciones se refiere.

Por otra parte, tenemos el paquete, com.way.persistence.drivers que contiene los drivers más comunes a utilizar para probar el ORM en esta versión 1.0. En este caso, hemos optado por los drivers de los principales y más conocidos sistemas gestores de bases de datos de la actualidad: MySQL (por defecto), Oracle, SqlServer, Informix y HSQL.

El corazón o espina dorsal de WayPersistence se define en el paquete siguiente, com.way.persistence.engine, es el motor del ORM y junto con com.way.persistence.databse son los dos paquetes fundamentales y más importantes del framework de acceso a datos.

En primer lugar, tenemos CargarDatos, que es una clase es utilizada para cargar los datos de las distintas operaciones del patrón CRUD que se den, al utilizar este framework. Estas operaciones básicas son la consulta, la inserción, el borrado y la actualización de tuplas de una base de datos persistente a través de este framework de persistencia.

La clase Driver contiene dependencias a la base de datos. Los drivers definidos en el paquete com.way.persistence.drivers dependen de este, es decir, extienden de esta clase genérica Driver.

Generador es una clase que genera claves utilizando filas para separar las secuencias de la tabla. Esta clase padre tiene varias clases hijas que extienden de esta que son: Generador Inserciones, GeneradorSelectMax y GeneradorSequencia. En el primer caso, utiliza columnas de la base de datos con sus valores y los añade a la base de datos cuando las filas

son insertadas. El segundo, es un básico generador de consultas que seleccionan el máximo.
El GeneradorSequencia es un método que genera secuencias para bases de datos que las soporten, ejemplo Oracle.

En este paquete tan importante, tenemos además, MotorQuery que actúa como gestionador o generador del SQL correspondiente e invocado por la clase WayQuery.

Por último, en este paquete, tenemos las dos clases más importantes de esta sección, SesionJdbcInterna y SesionManager.
La prime clase, contiene el acceso a las rutinas de bases de datos. La segunda ayuda a todas las interacciones entre el DataSet y la conexión JDBC con la base de datos. Estableces las conexiones con rutinas como OPEN, BEGIN, COMMIT, FLUSH, ETC. Funciona como la mayor parte de los manejadores de sesiones. Muy útil y fundamental para controlar las sesiones de WayPersistence.

Si seguimos con los paquetes del framework, tenemos com.way.persistence.model.utilidades que como su nombre indica, contiene clases útiles para la mejora del ORM. WayException controla la mayor parte de los errores que se producen en WayPersistence. En la mayor parte incluye información de los procesos que han fallado o no han podido continuar bien sea por errores de bugs, de JDBC, errores internos, test de pruebas, etc. Además de esta clase poseemos una SUte con métodos genéricos de utilidades, una de constantes para definir las constantes a nivel del proyecto, parte a mejorar y a retocar en versiones posteriores. Y por último el control de LOGs con SLog y SLogSlf4j que será el Log del que dependamos en un primer momento aunque sufrirá modificaciones en versiones posteriores.

El último de los paquetes a explicar del framework es: com.way.persistence.service con una clase con CasosDePrueba para probar el funcionamiento global del framework y otra DemostracionORM donde se hace una prueba de concepto de todo el framework para comprobar su funcionalidad, simplicidad, eficacia y eficiencia a partir de un buen ejemplo dentro del propio ORM.

Por otra parte, tendremos un fichero de configuración de tipo .properties en el cual definiremos las características de nuestra base de datos: driver, url, username y password. Estas características nos permitirán conectar nuestro ORM con el sistema gestor de base de datos a través del jar del propio gestor de bases de datos.

INDICE

IV. ORM – MAPEO DE MODELO DE OBJETOS AL MODELO RELACIONAL

1. Introducción ORM

La persistencia de la información es la parte más crítica en el desarrollo de proyectos software. Si la idea de la aplicación es orientarla a objetos, la persistencia se consigue con serialización de objetos o con el almacenamiento en base de datos. El modelo de datos relacional es el más utilizado hoy en día por la mayor parte de las empresas. Si comparamos el modelo de datos con el modelo relacional, se diferencia en bastantes características, por ello, para lograr que se conecten el uno con el otro, se creo en su día, ORM, que hace de interface entre estos dos modelos.

ORM (Objetc – Relational Mapping) o marco de mapeo relacional – objeto, es decir, mapeo del modelo de objetos al modelo relacional, es la herramienta a implementar y/o utilizar para lograr comunicar la base de datos con su modelo (Relacional) con la aplicación Orientada a Objetos (Modelo de Datos). Para lograr implementar ORM, utilizamos o creamos un motor de Persistencia, con él podremos buscar la solución al problema de la diferencia entre los dos modelos utilizados hoy en día para manipular y organizar datos en un proyecto de software, el utilizado en la memoria del ordenador (orientado a objetos) y el utilizado en la base de datos (modelo relacional).

Los aspectos que debemos considerar a la hora de escoger un marco ORM son: la memoria caché, las transacciones, la carga retardada, la concurrencia, etc.

2. Introducción a UML

Lenguaje Unificado Modelado (UML)

El "Unified Modelling Languaje" (UML) provee a los analistas y arquitectos de sistemas que trabajan en el diseño y análisis de objetos de un lenguaje consistente para especificar, visualizar, construir y documentar los artefactos de un sistema de software, así también es útil para hacer modelos de negocios.

Varios nuevos conceptos existen en UML, incluyendo: mecanismos de extensión (estereotipos, valores marcados y restricciones), procesos y ramas de procesamiento, distribución y concurrencia, patrones y colaboración, diagramas de actividad, refinamiento, interfaces y componentes, lenguaje de restricciones.

3. Concepto: Frameworks de Persistencia

1. Definiciones

Definición Framework

Un framework de persistencia es un conjunto de tipos de propósito general, reutilizable y extensible, que proporciona funcionalidad para dar soporte a los objetos persistentes. Un servicio de persistencia realmente proporciona el servicio, y se suele crear un framework de persistencia. Un servicio de persistencia se escribe normalmente para que trabaje con bases

de datos relacionales, en cuyo caso también se conoce como servicio de correspondencia O-R. Generalmente, un servicio de persistencia tiene que traducir los objetos a registros (o alguna forma estructurada como XML) y guardarlos en una base de datos, y traducir los registros a objetos cuando los recuperamos de la base de datos.

Definición ORM

El mapeo objeto-relacional (más conocido por su nombre en inglés, Object-Relational mapping, o sus siglas O/RM, ORM, y O/R mapping) es una técnica de programación para convertir datos entre el sistema de tipos utilizado lenguaje orientado a objeto y bases de datos relacionales. Si lo pensamos en la práctica, se crea una base de datos virtual sobre la base de datos relacional, lo que posibilita la utilización de la base de datos como un conjunto de clases que nos permiten aprovecharnos de las potentes características de la orientación a objetos, tales como: herencia, polimorfismo, etc. Existen gran cantidad de ORM en el mercado pero algunos desarrolladores no dudan en crearse el su propio ORM.

2. Requisitos para el servicio y framework de persistencia

Las funciones básicas que debe tener un framework son:

- Almacenar y recuperar los objetos en un mecanismo de almacenamiento persistente.
- Confimar y deshacer (commit y rollback) las transacciones.

El diseño debe ser extensible para dar soportes a los distintos mecanismos de almacenamiento, como bases de datos relacionales, registros en ficheros simples, o XML.

3. Ideas claves para un framework de persistencia

- Correspondencia: se debe establecer alguna correspondencia o mapeo entre una clase y su almacenamiento persistente. Es decir, debe haber una correspondencia de esquemas.
- Identidad de objeto: los registros y los objetos tienen un único identificador de objeto para relacionar fácilmente los registros con los objetos y asegurar que no hay duplicados inapropiados.
- Conversor de base de datos: la conversión 'mapper' es la encargada de la materialización y desmaterialización.
- Materialización y desmaterialización: la materialización es el acto de transformar una representación de datos no orientada a objetos de un almacenamiento persistente de objetos. La desmaterialización es la actividad opuesta.
- Caché: los servicios persistentes almacenan en una caché los objetos materializados por razones de rendimiento.
- Estado de transacciones de los objetos: es útil conocer el estado de los objetos en función de sus relaciones con la transacción actual.
- Operaciones de transacción: operaciones confirmar y deshacer (commit y rollback).
- Materialización perezosa: no todos los objetos se materializan de una vez, una instancia particular sólo se materializa bajo demanda, cuando es necesaria.

- Proxies virtuales: la materialización perezosa se puede implementar utilizando una referencia inteligente que se conoce como Proxy virtual.

4. Inconvenientes

Cuando trabajamos con objetos, el manejo de sus datos es de forma escalar. En algunos productos de bases de datos, los RDBMS más comunes que utilizan lenguaje SQL solo pueden almacenar valores comunes como enteros, cadenas, organizados según el modelo de datos en tablas.
La idea de ORM es solventar el problema de la conversión de valores simples a objetos y viceversa por parte de los programadores y ayudar con sus ventajas a solventar ese problema mapeando las relaciones a objetos y viceversa. Si se consigue que los objetos y la base de datos conserven las mismas propiedades, se dice que son persistentes.

5. Implementaciones

Los tipos de bases de datos más utilizadas hoy en día son las bases de datos en lenguaje SQL, independientemente del SGBD en el que estén implementadas. Normalmente, estas bases de datos se almacenan en tablas. Lo que en la BD se puede almacenar en varias tablas, en objetos se puede representar en sólo uno.
En la mayor parte de los ORM se han desarrollado multitud de paquetes para reducir el proceso de desarrollo de sistemas de mapeo relacional de objetos proveyendo librerías de clases capaces de realizar mapeos automáticamente.
Desde el punto de vista de un programador, el sistema debe lucir como un almacén de objetos persistentes. Uno puede crear objetos y trabajar normalmente con ellos, los cambios que sufran terminarán siendo reflejados en la base de datos.
Sin embargo, en la práctica no es tan simple. Todos los sistemas ORM tienden a hacerse visibles en varias formas, reduciendo en cierto grado la capacidad de ignorar la base de datos. Peor aún, la capa de traducción puede ser lenta e ineficiente (comparada en términos de las sentencias SQL que escribe), provocando que el programa sea más lento y utilice más memoria que el código "escrito a mano".
Un buen número de sistemas de mapeo objeto-relacional se han desarrollado a lo largo de los años, pero su efectividad en el mercado ha sido diversa. Considerado uno de los mejores fue NeXT's Enterprise Objects Framework (EOF), pero no tuvo éxito debido a que estaba estrechamente ligado a todo el kit de NeXT's, OpenStep. Fue integrado más tarde en NeXT's WebObjects, el primer servidor web de aplicaciones orientado a objetos. Desde que Apple compró NeXT's en 1997, EOF proveyó la tecnología detrás de los sitios web de comercio electrónico de Apple: los servicios .Mac y la tienda de música iTunes. Apple provee EOF en dos implementaciones: la implementación en Objective-C que viene con Apple Developers Tools y la implementación Pure Java que viene en WebObjects 5.2. Inspirado por EOF es el open source Apache Cayenne. Cayenne tiene metas similares a las de EOF e intenta estar acorde a los estándares JPA.
Más recientemente, un sistema similar ha comenzado a evolucionar en el mundo Java, conocido como Java Data Objects (JDO). A diferencia de EOF, JDO es un estándar, y muchas implementaciones están disponibles por parte de distintos distribuidores de software. La especificación 3.0 de Enterprise Java Beans (EJB) también cubre la misma área. Han existido algunos conflictos de estándares entre ambas especificaciones en términos de preeminencia. JDO tiene muchas implementaciones comerciales, mientras que EJB 3.0 está aún en desarrollo. Sin embargo, recientemente otro estándar ha sido anunciado por JCP para abarcar estos dos estándares de manera conjunta y lograr que el futuro estándar trabaje en diversas arquitecturas de Java. Otro ejemplo a mencionar es

Hibernate, el más usado framework de mapeo objeto-relacional en Java que inspiró la especificación EJB 3.
En el framework de desarrollo web Ruby on Rails, el mapeo objeto-relacional juega un rol preponderante y es manejado por la herramienta ActiveRecord. Un rol similar es el que tiene el módulo DBIx::Class para el framework basado en Perl Catalyst, aunque otras elecciones también son posibles.
Otras bases de datos sin SQL.

Las bases de datos como Caché no necesitan mapeo objeto-relacional manual. El acceso del SQL a los valores no escalares ya ha sido construido. Caché permite a los desarrolladores diseñar cualquier combinación de programación orientada a objetos y almacenamiento estructurado en tablas en la misma base de datos en lugar de depender de herramientas externas.

Otra solución puede ser el uso de un sistema de administración de base de datos orientada a objetos (OODBMS: Object-oriented database management system), lo cual, como el nombre lo sugiere, es una base de datos diseñada específicamente para trabajar con valores orientados a objetos. Usar un OODBMS puede eliminar la necesidad de convertir datos desde y hacia su forma SQL, y los datos pueden ser almacenados en su representación original como objetos.
Las bases de datos orientadas a objetos aún no han conseguido una alta aceptación y uso. Una de las principales limitaciones reside en que, cambiar de un sistema de administración de base de datos SQL a un sistema orientado totalmente a objetos implica que se pierde la capacidad de crear sentencias SQL, un método ya probado para obtener combinaciones específicas de datos. Por esta razón, muchos programadores se encuentran más a gusto trabajando con un sistema de mapeo de objetos y SQL, aún cuando la mayoría de las bases de datos comerciales orientadas a objetos son capaces de procesar consultas SQL de manera limitada.

4. Mapeo de objetos al modelo relacional

4.1. Visión general

La **persistencia de los objetos**, es el hecho de persistir la información de un objeto de forma permanente (guardar), pero también se refiere a recuperar y volver a ver la información introducida o guardada para poder se utilizada de nuevo.

La persistencia no es más que le mecanismo que se utiliza para persistir la información de un tipo determinado que puede serializar, guardar, etc. Los objetos en persistencia se clasifican en persistentes y transitorios. Los primeros, son almacenados para su posterior utilización y su tiempo de vida es distinto al del proceso que los instanció. Sin embargo, los transitorios, dependen directamente del ámbito del proceso que los instanció.

La persistencia permite al programador almacenar, transferir, recuperar el estado de los objetos a través de diferentes técnicas que se utilizan, como son: serialización, motores de persistencia, bases de datos orientadas a objetos, etc.

El modelo de objetos es muy diferente al modelo relacional. La interfaz que une a estos dos modelos es ORM. Equipos de desarrolladores de Java y .Net han popularizado la

utilización de modelos de datos orientados a objetos a través de UML al diseñar las aplicaciones, dejando de lado el enfoque de la aplicación.

UML (Unified Modeling Language), Modelado Unificado del Lenguaje, es la especificación más utilizada, certificada por OMG(Object Management Group) utilizada en la Ingeniería de Software no sólo para la estructura de un proyecto de software, sino también, su comportamiento, su arquitectura, su proceso de negocio y por supuesto, la estructura de datos.

OMG (ObjectManagement Group) es un consorcio dedicado al cuidado y el establecimiento de varios estándares de tecnologías orientadas a objetos, tales como UML, CORBA, XMI. Se trata de una organización sin ánimo de lucro que promueve la utilización de tecnologías orientadas a objetos mediante guías y especificaciones. El grupo está formado por grandes compañías y organizaciones de software como: HP, IBM, Sun Microsystems y Apple Computers.

Las bases de datos más populares hoy en día son relacionales. Oracle, SQLServer, Mysql, Postgress son los DBMS más usados.
En el momento de persistir un objeto, normalmente, se abre una conexión a la base de datos, se crea una sentencia SQL parametrizada, se asignan los parámetros y se ejecuta la transacción.

Cuando tenemos un objeto debemos preguntarnos cómo debemos asociarlos a nivel relacional, cómo almacenarlos (automáticamente, manualmente), que ocurre con las claves foráneas o secundarias, etc.

Para tratar los datos que recuperamos persistidos, nos fijamos en sí cargamos únicamente el objeto, sí cargamos sus asociaciones, sí cargamos el árbol completo, etc.

Está claro, que buena parte del tiempo de un proyecto de desarrollo de software se dedica a mapeo entre objeto y la relación. Debemos tener en cuenta la caché, las transacciones, la carga perezosa, puesto que no es igual que una tabla tenga múltiples tablas foráneas a que tenga una porque el cargo transaccional no es el mismo.

El ORM se encargará en la mayor medida, de resolver las cargas y permitirá:

- Mapear clases a tablas: una propiedad será una columna, una clase será una tabla.
- Persistir objetos. A través de un método del tipo orm.save(objeto) generará el SQL correspondiente que permitirá persistir un objeto.
- Recuperar los objetos persistidos, a través de un método objeto = orm.load (objeto class, clave primaria).
- Recuperar una lista de objetos a partir de un lenguaje de consulta especial, en algunos casos basado en SQL, en otros personalizado para cada ORM, casi siempre, similar a SQL. Por ejemplo, para Hibernate, existe un lenguaje personalizado creado por ellos mismos llamado HQL (Hibernate Query Language). En este caso, en lugar de hacer consultas SQL puras, se utiliza el propio lenguaje HQL que se basa en SQL pero en lugar de hacer referencia a las tablas del modelo relacional, se hace referencia al Modelo de Datos orientado a objetos, el definido en el propio proyecto.

4.2. Incongruencia entre el modelo relacional y el de objetos

Mientras que un modelo de objeto posee jerarquía de árbol, un modelo relacional con su base de datos, agrupa sus objetos en forma tabular. Esta incongruencia entre la tecnología de objetos y la relacional, fuerza al desarrollador a mapear el esquema de objetos a un esquema de datos.

Debido a este tipo de configuración tabular y orientada a objetos en árbol, ha llevado a múltiples desarrolladores de diferentes tecnologías de objetos a utilizar o construir un puente entre el mundo relacional y el mundo orientado a objetos. El marco de trabajo Enterprise Javabeans (EJB) proporciona uno de lo muchos métodos para reducir esta distancia, otra opción es utilizar Spring, entre otros, a través de beans.

4.3. Terminología

Modelo de objetos:

- **Identidad de objeto**. Propiedad por la que cada objeto es distinguible de otros aún si ambos tienen el mismo estado (o valores de atributos).
- **Atributo**. Propiedad del objeto al cual se le puede asignar un valor.
- **Comportamiento**. Es el conjunto de interfaces del objeto.
- **Interface**. Operación mediante la cual el cliente accede al objeto.
- **Encapsulación**. Es el ocultamiento de los detalles de implementación de las interfaces del objeto respecto al cliente.
- **Asociación**. Es la relación que existe entre dos objetos.
- **Clase**. Define como será el comportamiento del objeto y como almacenará su información. Es responsabilidad de cada objeto ir recordando el valor de sus atributos.
- **Herencia**. Especifica que una clase usa la implementación de otra clase, con la posible sobre escritura de la implementación de las interfaces.

Modelo relacional:

- **Base de datos**. Conjunto de relaciones variables que describen el estado de un modelo de información. Puede cambiar de estado (valores) y puede responder preguntas sobre su estado.
- **Relación variable (Tabla**). Mantiene tuplas relacionadas a lo largo del tiempo, y puede actualizar sus valores. Esquematiza como están organizados los atributos para todas las tuplas que contiene.
- **Tupla (fila).** Es un predicado de verdad que indica la relación entre todos sus atributos.
- **Atributo (columna).** Identifica un nombre que participa en una relación y especifica el dominio sobre el cual se aplican los valores.
- **Valor de atributo (valor de columna).** Valor particular del atributo con el dominio especificado.
- **Dominio.** Tipos de datos simples.

4.4. Mapeo de objetos

Un objeto se compone de propiedades y métodos. Las propiedades son las partes persistidas. Cada propiedad puede ser simple (cualquier tipo de dato primitivo) o compleja (definido por el usuario, bien sean objetos o estructuras).

En el modelo relacional cada final de una tabla se mapea como un objeto y cada columna como una propiedad. Normalmente cada objeto en nuestro modelo, representa una tabla del modelo relacional. Tendremos una clase por cada tabla del objeto relacional. Estas clases tendrán una serie de propiedades que serán las que representen las columnas de las tablas de la base de datos (modelo relacional). Si se crea una instancia de esta clase, tendremos un objeto del tipo que representa a la tabla del modelo relacional.

Así pues, cada propiedad del objeto se mapea de 0 a x columnas en una tabla. Cada columna de la tabla suele respetar el tipo de datos con su correspondiente propiedad del objeto. A veces por optimización es necesario ajustar en la base de datos relacional o incluso en el modelo de objetos.

Por ejemplo, en una base de datos que se mapea con el conocido Hibernate, si queremos definir en una propiedad de la clase que representa la tabla en base de datos, un trigger que genere a través de un secuencia un identificador de la tabla. El propio ORM Hibernate, no es capaz de interpretar que se ha definido en una base de datos Oracle un trigger para esta cuestión. En este caso particular, debemos editar manualmente el modelo de objetos y ponerle la sentencia para que genere dicho trigger. Este y muchos ejemplos similares son los que cada ORM lucha día a día para facilitar las cosas a las empresas que optan por utilizar cada ORM en cuestión.

Para acceder a las propiedades de los objetos, se suelen utilizar métodos especiales del tipo get y set, getters y setters que permiten editar y/o obtener los valores de dichas propiedades de cada uno de los objetos mapeados del modelo relacional.

4.5. Identidad del objeto

Para distinguir cada fila de las otras se necesita un identificador de objetos (OID) que es una columna más de nuestro objeto. Este OID siempre representa la clave primaria en la tabla y suele ser de tipo numérico.

5. Mapeo de Relaciones.

Denominamos relaciones entre objetos a las uniones que se producen entre ellos como la asociación, la herencia o la agregación. Para persistir las mismas, utilizamos transacciones, puesto que se utilizan varias tablas para realizar distintos cambios. En caso de un error en alguna transacción que utilice más de un objeto, se asegura deshacer dicha ejecución a través de una sentencia conocida en persistencia, denominada “rollback”.

5.1. Asociaciones entre objetos

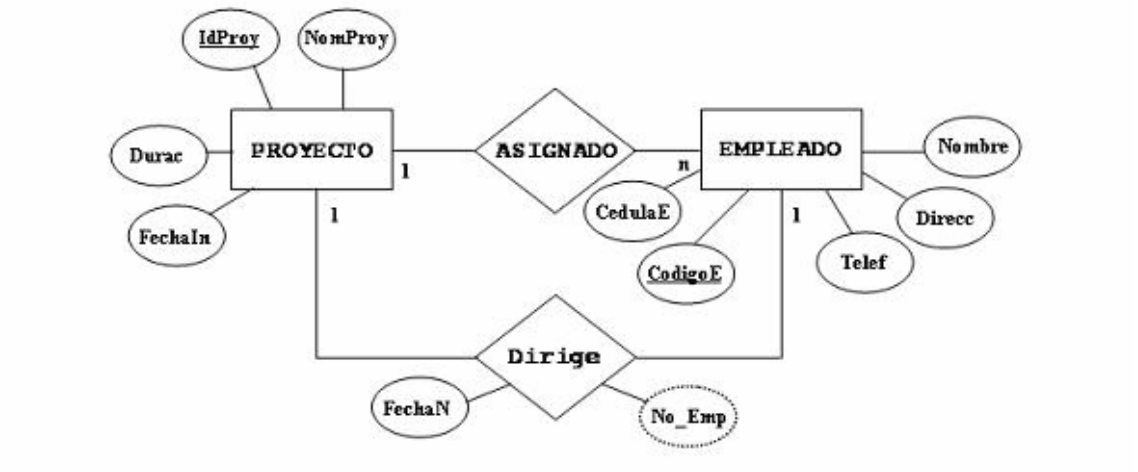
Una regla fundamental para el mapeo relacional es el tipo de multiplicidad en el modelo de objetos y el modelo relacional. Una relación 1-1 en el modelo de datos orientado a objetos, será igual en el modelo relacional, se corresponderá.

Las asociaciones se dividen según su multiplicidad: 1-1, 1-n, m-n. O según su navegabilidad: unidireccional o bidireccional. Esto se da a nivel modelo relacional y modelo de datos, pero en el modelo relacional sólo se representa la navegabilidad del tipo bidireccional.

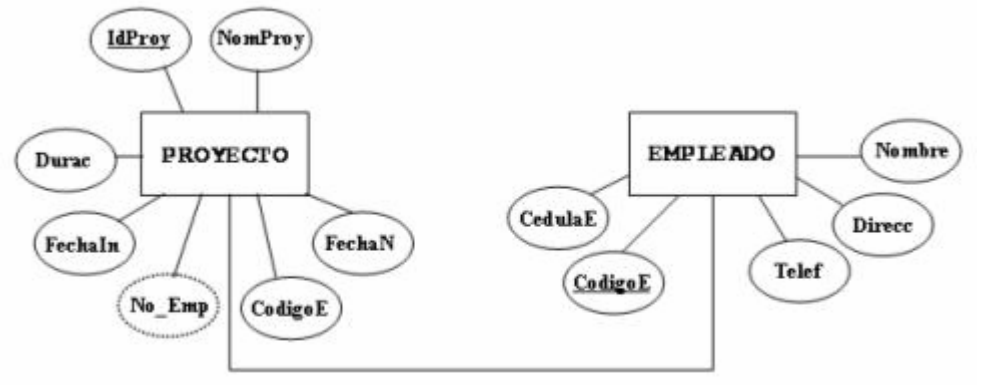
5.1.1. Multiplicidades:

- **Multiplicidad 1-1:** Cada objeto A1 está asociado con 0 o 1 objeto de tipo A2, y cada objeto A2 está con 0 o 1 objeto de tipo A1. En el modelo de objetos se representa como un tipo de datos definidos por el usuario.

En el ejemplo, podemos ver una relación 1 a 1, pero no es necesaria una tabla intermedia, así pues, la destruimos para optimizar el Modelo E/R.

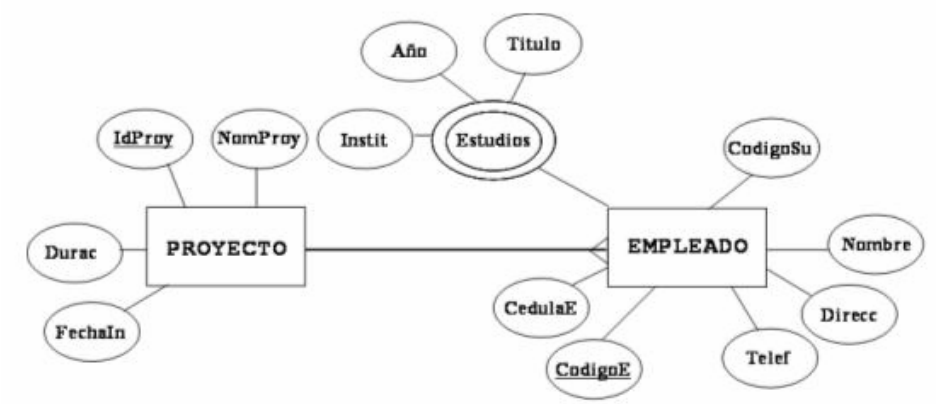


Modelo Entidad / Relación Optimizado

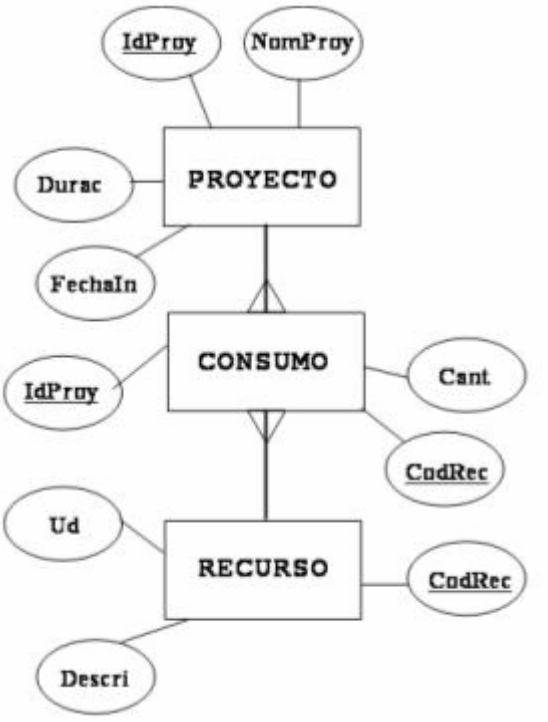


- **Multiplicidad 1- n:** cada objeto de tipo A1 puede estar asociado con 0 a n objetos de tipo A2, y cada objeto de tipo A2 está asociado 0 o n veces con un

objeto de tipo A1. Este tipo de relación puede darse en el siguiente ejemplo: Una ejemplo puede ser Un empleado que pertenece a un único departamento, tendremos una foreign key en la tabla Empleado que apuntará a la tabla Departamento. Un empleado sólo pertenecerá únicamente a un departamento pero un departamento tendrá 0 o muchos usuarios.



- **Multiplicidad n – m:** cada objeto A1 está asociado con 0 o n objetos de tipo A2, y a su vez cada objeto A2 está asociado con 0 o n objetos de tipo B. Un ejemplo claro de este tipo de multiplicidad sería un usuario con sus roles. Tendríamos una tabla Usuario, otra Roles y otra Usuario_Roles. En esta última uniríamos a través de las claves primarias de las dos tablas anteriores, la relación entre las mismas. De esta manera, tendríamos en el modelo relacional una relación de n a m, un usuario podrá tener 0 roles si no tiene ningún registro en la tabla Usuario_Roles, 1 rol si tiene por lo menos un registro, o múltiples roles que se corresponderán con los roles definidos en la tabla Roles. A su vez, podrá a ver varios usuarios con el mismo rol. Claro ejemplo de relación con multiplicidad n – m. Dicha tabla Usuario_Roles tendrá al menos 2 columnas, cada una representando la clave foránea a las 2 tablas que relaciona. De esta forma transformamos la relación n-m a dos relaciones 1 - n y 1 - m.



Según párrafos anteriores, decíamos que las asociaciones además de por su multiplicidad, se agrupan por su navegabilidad: unidireccional y bidireccional.

5.1.2. Navegabilidad.

Navegación unidireccional
Una relación unidireccional es aquella que conoce con que objetos está relacionado, pero esos objetos no conocen al objeto original. Representación '→'. El en modelo relacional, la navegabilidad está representada por una clave foránea y depende de la multiplicidad la tabla donde se representa.

Sin embargo, todas las relaciones en la base de datos son bidireccionales.

Navegabilidad bidireccional
Una relación bidireccional existe cuando los objetos en cambios extremos de la relación conocen el objeto del extremo. Ambos objetos deben implementar los métodos de acceso hacia el objeto con el cual está relacionado.

Asociación clave foránea
Para el mapeo de las relaciones, utilizamos los identificadores de objetos (OID) que suelen ser claves foráneas, una columna más agregada a la tabla. Las relaciones en el modelo relacional son siempre bidireccionales, cumplen el patrón "Foreign Key Mapping".

Asociación recursiva
Una asociación recursiva o reflexiva se produce cuando ambos extremos de la asociación son la misma entidad, clase o tabla. En el modelo de objetos se representa con una línea que sale de la entidad y llega a la misma entidad en forma cíclica. Puede tener cualquier multiplicidad y navegabilidad.

Agregación / Composición
Una agregación es una relación más fuerte que una asociación pero independiente.
Una composición es una relación fuerte y dependiente entre objetos.

A nivel modelo relacional, tanto la agregación, la composición, cómo la asociación, se representan como una relación.

La agregación se mapea como una relación n-m, es decir, suele haber una tabla auxiliar para implementarla. La composición puede mapearse como una relación 1 a n. En el modelo de datos orientado a objetos, las agregaciones y las composiciones se corresponden con un array o colección de objetos. Para el caso concreto de Hibernate, se utilizan 'hashset'.

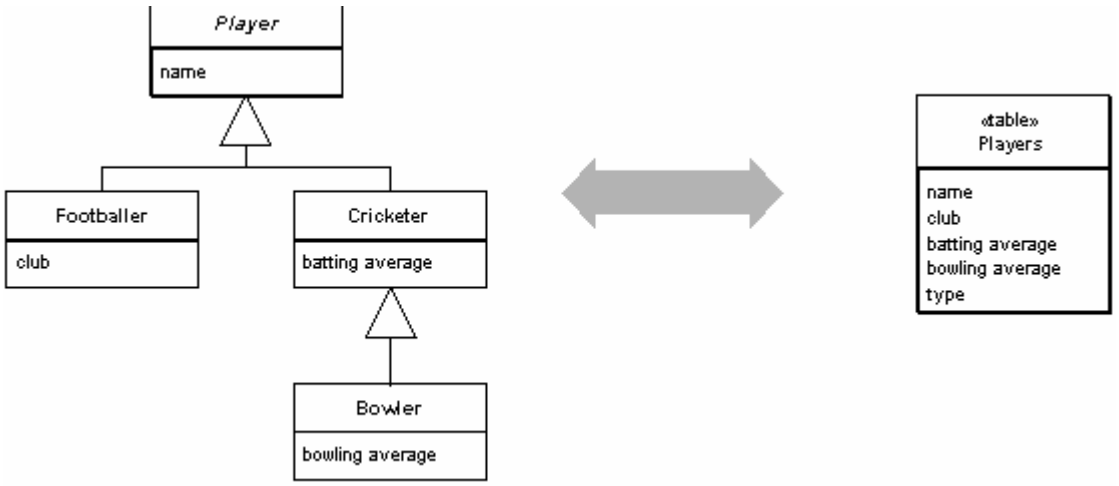
Herencia
Las asociaciones funcionan en ambos modelos. Para el caso de la herencia, el modelo Entidad / Relación no lo soporta.

Existen 3 tipos fundamentales de mapeos principales si modelamos su jerarquía a:
- Una sola tabla.
- Concreta en tablas.
- Completa en tablas.

Este tipo de implementación de la herencia dependerá del caso concreto en cuestión y de la escalabilidad y dificultad del proyecto, así como su magnitud.

- **Herencia con mapeo de la jerarquía a una tabla:** mapeamos todos y cada uno de los atributos que representan la herencia, sin repetir, en una única tabla.

En este ejemplo concreto, tenemos la clase 'Player' que hace de clase padre. De Placer heredan o son subclases, 'footballer' y 'cricketer'. Estos tendrán la propiedad name a no ser que la sobrescriban, que no es el caso. Este tipo de herencias a nivel programación van a depender del ámbito que tengan cada una de las clases, principalmente la clase padre. A su vez, de 'Cricketer' cuelga una clase 'bowler', que heredará de 'Cricketer' sus propiedades y/o métodos. Esta implementación de la herencia se da en el modelo de objetos, mientras en el modelo entidad / relación, se representa en una única tabla. Los jugadores o players tendrán un nombre (heredado de player en el caso del modelo de objetos), las propiedades: club, batting average, bowling average y type. Este type será probablemente un tipo numérico o enumerado que permitirá distinguir en esa tabla si estamos tratando con un 'player' de tipo: footballer, cricketer o bowler. Los campos de la tabla club, batting average y bowling average serán habilitados para que puedan ser nulos porque no todos los registros podrán tener todos los valores rellenados.



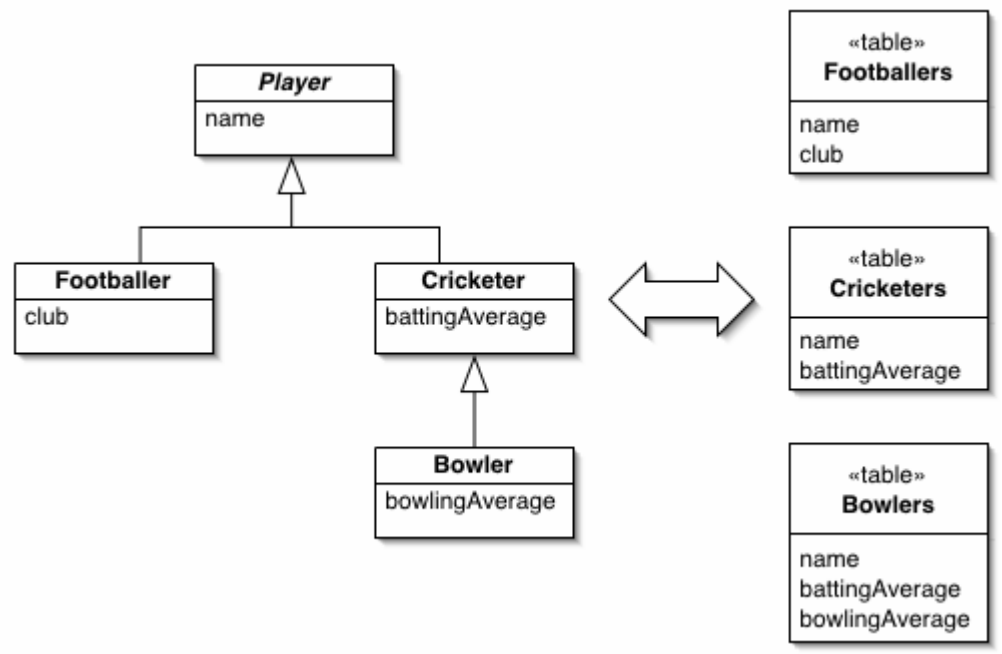
5.2. Ventajas y Desventajas

Las **ventajas** fundamentales de esta implementación son: aproximación simple, cada nueva clase se agregarán columnas para datos adicionales en el modelo relacional, soporta el polimorfismo cambiando el tipo de fila, el acceso a datos es rápido, el mostrado de datos en un informe será rápido por ser una única tabla.

Las **desventajas** son: mayor acoplamiento entre las clases, si se hace un cambio puede afectar notablemente, desperdicio de espacio en las tablas por tener demasiadas columnas con datos vacíos. Es decir, si una tabla tiene muchas columnas y no se rellenan en cada una de las tuplas. La clase crece exponencialmente.

- **Herencia con mapeo de la jerarquía concreta en tablas:** mapeamos cada clase concreta en una tabla. Cada tabla contendrá los atributos heredados más

los suyos propios. La clase padre abstracta, será mapeada en cada una de sus tablas derivadas.



Como vemos en el ejemplo, se representan las tablas en el modelo relacional con las propiedades de las clases padre más los valores de la clase hija. Esto supone un problema que se va incrementando cuanto mayor rango de jerarquía exista. Ejemplo 'Bowler', se incrementa considerablemente con las propiedades de todas las clases que va heredando.

Las **ventajas** son: informes fáciles y rápidos de obtener puesto que los datos concretos están en una única tabla, facilidad para acceder a los datos.

Las **desventajas** son: pobre escalabilidad, si se modifica la clase base, puesto que implica la modificación completa de todas las tablas que heredan de esta clase padre.

- **Herencia con mapeo de la jerarquía completa en tablas:** se crea una tabla por cada clase de la herencia. Se agregan las columnas a cada una de las clases agregadas para el control de la concurrencia.

Si es necesario leer el objeto heredado, se unen las dos tablas o bien se consultan por separado. Las claves primarias de las tablas heredadas serán las mismas que las de la clase base.

Las **ventajas** son: fácil de entender, porque es un mapeo uno a uno, soporta muy bien el polimorfismo, ya que tiene almacenado los registros en la tabla correspondiente y escalable.

Las **desventajas** son: demasiadas tablas en la base de datos, se necesita realizar operaciones en varias tablas, informes rápidos, aunque difíciles de montar por la complejidad de la relación entre las tablas y por ser informes que se realizan con la agrupación de múltiples tablas.

6. Persistencia

Persistir objetos java en una base de datos relacional es un reto único que implica serializar un árbol de objetos Java en una base de datos de estructura tabular y viceversa. Este reto actualmente está siendo corregido por varias herramientas diferentes. La mayoría de estas herramientas permite a los desarrolladores instruir a los motores de persistencia de cómo convertir objetos Java a columnas o registros de la base de datos y al revés. Es fundamental poder mapear objetos Java a columnas y registros de la base de datos de una forma optimizada en velocidad y eficiencia.

En general, existe una incongruencia entre la orientación a objetos y las bases de datos relacionales. Para evitar esto, como dijimos anteriormente, recurrimos a un ORM, que actuará de capa intermedia para mapear y comunicar a estos dos mundos para adaptarlos. Esta capa ORM (Object Relational Mapping) permite interpretar tablas y columnas del modelo E/R para que se comuniquen con entidades y propiedades del modelo de datos orientado a objetos.

Normalmente, para cargar un objeto persistido en memoria, los pasos a seguir son: abrir la conexión a la base de datos, crear una sentencia SQL parametrizada, llenar los parámetros (clave foránea, primaria, etc.), crear una transacción y ejecutar como tal los parámetros y el SQL. Posteriormente cerraremos la conexión. Con un framework de persistencia, el proceso de persistencia se reduce a: abrir una sesión con la base de datos, especificar el tipo de objeto que queremos, cerrar la sesión. De todas formas, un framework de persistencia deberá cumplir los siguientes requisitos:

- Transacciones. Asegurar la atomicidad, la transaccionalidad y hacer rollback si alguna de las transacciones falla para deshacer los cambios.
- Controlar la caché para no cargar la memoria con excesivos objetos utilizados.
- Asignar manual o automáticamente las claves primarias u OID.

6.1. Patrón CRUD

Acrónimo de CREATE-READ-UPDATE-DELETE. Conocido como el padre de todos los patrones de capa de acceso. Describe que cada objeto debe ser creado en la base de datos para que sea persistente. Una vez creado, la capa de acceso debe tener una forma de leerlo para poder actualizarlo o simplemente borrarlo. El borrado de un objeto se delega al programador para no tener problemas con el recolector de basura si encargamos esta tarea a la base de datos.

Las clases de operaciones conocidas como CRUD (Create, Read, Update y Delete), son las encapsuladas por el patrón DAO (acceso a datos). Para interactuar el DAO con la base de datos utilizará los métodos que cumplen el patrón CRUD.

Cuando la capa de negocios necesite almacenar datos (por ejemplo) solo tendrá que hacer referencia al método correspondiente para insertar los datos de la clase DAO, de esta manera si en algún momento se llegara a optar por usar otro motor de búsquedas, la capa del negocio no se debe preocupar ya que sólo bastará con actualizar la capa de datos. Si hablamos de patrones algunos detectarán por aquí que se están delegando responsabilidades, una buena práctica de la orientación a objetos.

Por cada tabla de una base de datos relacional existirá un DAO. Esto consiste básicamente en una clase que es la que interactúa con la base de datos. Los métodos de esta clase dependen de la aplicación y de lo que queramos hacer. Pero generalmente se implementan las 4 funciones básicas (también conocidas como métodos CRUD).

Una vez entendido el funcionamiento de DAO, es conveniente explicar un nuevo término que nos será de ayuda para completar la implementación de nuestra capa de persistencia. Se trata de los DTO's (Data Transfer IObject), los cuáles son utilizados por DAO para transportar los datos desde la base de datos hasta la capa de lógica de negocio o viceversa. Estos patrones no son utilizados por todos los ORM igualmente, pero por ejemplo, el más conocido, Hibernate, sigue estos patrones DTO y DAO.

6.2. Caché

En la mayor parte de las aplicaciones, se accede al 20% de los datos de la aplicación en mayor medida que el resto. Es decir, esto significa que hay una serie de datos relevantes que se consultan con más frecuencia. Normalmente, los grandes proyectos necesitan de una sincronización a nivel memoria caché que le permita manejar grandes cargas transaccionales y así poder procesar múltiples instancias simultáneamente.

En la mayor parte de los casos, se utiliza una única base de datos activa, con múltiples servidores conectados directamente a ella, soportando un número de variables de clientes.

Identificación de caché

Para no tener problema de caché en caso de tener varios servidores y para evitar la duplicidad de instancias, debemos consultar antes de crear una nueva instancia, comprobar si esa instancia del objeto ya existe en caché, en ese caso, no se duplica y obtiene una referencia a esa instancia creada inicialmente. Esto evita problemas en caché. Se suele aplicar la identidad en la misma caché a través de una tabla hashing en base a las claves primarias.

6.3. Cargas y sobrecargas

Carga de relaciones

El principal problema en la carga de relaciones se produce en el momento que cargamos una tabla de una base de datos y tiene múltiples relaciones. Normalmente no se suelen cargar todas sus relaciones porque ralentizaría muchísimo todos los procesos de consulta. Se realiza una "carga retardada" de las relaciones, y se implementa por algún tipo de objetos como "patrón Proxy" que lanzan la carga cuando se acceden.

Normalmente a nivel objetos, poseemos métodos set y get, que nos permiten en el caso del get, obtener el valor de un atributo simple, que verifica si el atributo ha sido inicializado y sino es así lo lee desde la base de datos.

Lo mismo ocurre para campos grandes como por ejemplo, campos del tipo memo o tipo blob que ocupan más. En algunos casos también se utilizará carga directa porque interesa pero dependerá del caso y siempre optimizando tiempos de consulta, transacción, etc.

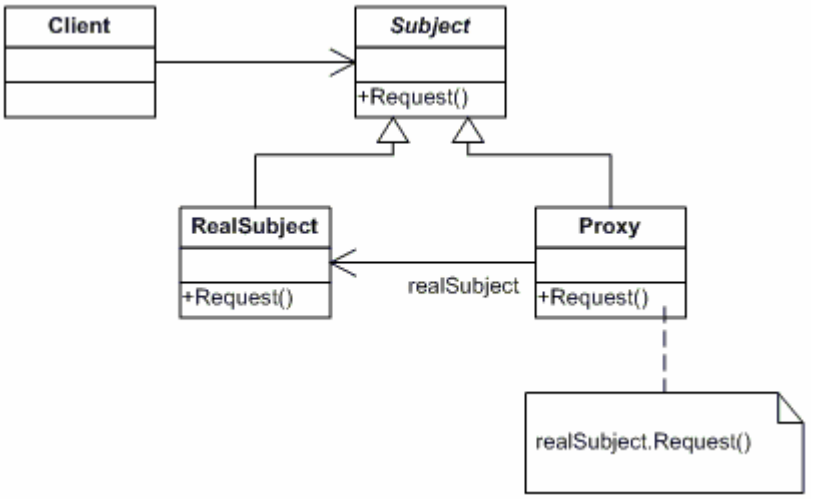
Sobrecargas innecesarias
Consejos para evitar sobrecargas en la capa de persistencia:

- Evitar mostrar al usuario todos los datos al principio, dando opción de búsqueda. No abusar de los combos o si se da el caso, para el caso de formularios web, controlarlos en sesión de manera que se llenen y/o vacíen en cada petición.
- Retornar los datos en forma de filas o tuplas.
- Unir los datos para informes que requieran múltiples datos con uniones hacia la izquierda.

6.4. Patrón Proxy

Mantiene una referencia al objeto real. Controla la creación y acceso a las operaciones del objeto real.

Un Proxy normalmente representa a un objeto pero sin contener la misma sobrecarga que el mismo. Suele contener sus datos más relevantes, como el OID para identificar al objeto y un dato concreto que permita reconocer a ese objeto Proxy. Con este patrón sólo se dará la información que el usuario necesita sin necesidad de sobrecargar la memoria caché y evitar traer todo el objeto real del usuario que provocaría una mayor sobrecarga.



6.5. Concurrencia

La capa de persistencia debe permitir que múltiples usuarios trabajen en la misma base de datos y proteger los datos de ser escritos equivocadamente. Se debe tener en cuenta cuando se están trabajando en la misma base de datos más de una sesión a la vez.

La integridad de datos es un riesgo cuando dos sesiones trabajan sobre la misma tupla: la pérdida de alguna actualización es muy probable.

Existen dos técnicas para tratar el problema de concurrencia:

- Bloqueo pesimista: se bloquean todos los accesos desde que el usuario empieza a cambiar los datos hasta que hace COMMIT en la transacción.
- Bloqueo optimista: el bloqueo se aplica cuando los datos son aplicados y se van verificando mientras los datos son escritos.

Bloqueo optimista

Cuando se detecta un conflicto entre transacciones concurrentes, se cancela alguna de estas. Para resolver el problema, valida que los cambios COMMIT por una sesión no entran en conflicto con los cambios hechos en otra sesión. Una validación exitosa PRE-COMMIT es obtener un bloqueo de los registros con una transacción simple.

El bloqueo optimista desacopla la capa de persistencia de la necesidad de mantener una transacción pendiente, chequeando los datos mientras se escribe.

Se usan varios esquemas de bloqueo optimista. A veces se utiliza un campo de estampa de tiempo (timestamp) o un simple contador.

Generalmente, la aplicación iniciará una transacción, leerá los datos desde la base, cerrará su transacción, seguirá con las reglas de negocio involucradas para volver a iniciar una transacción, esta vez con los datos a ser escritos. En el caso de la estampa de tiempo, la capa de persistencia verificará que sea la misma que existe en la base de datos, escribirá los datos y actualizará la estampa de tiempo a la hora actual.

Bloqueo pesimista

Evita que aparezcan conflictos entres transacciones concurrentes permitiendo acceder a los datos a solo una transacción a la vez.

La aproximación más simple, consiste en tener una transacción abierta para todas las reglas de negocio involucradas. Hay que tener precaución con transacciones largas.

El bloqueo pesimista evita el conflicto anterior en total. Fuerza a las reglas de negocios a adquirir el bloqueo de los datos antes de empezar a usarlo, así, la transacción se usa completamente sin preocuparse por los controles de concurrencia.

El bloqueo pesimista notifica a los usuarios tempranamente de la contención de los datos, pero para los propósitos de negocios la concurrencia es considerada altamente importante.

5.6. Transacciones

Introducción

Una transacción e un sistemas gestor de bases de datos, es un conjunto de órdenes que se ejecutan formando una unidad de trabajo, es decir, en forma indivisible y atómica. Si un SGBD es transaccional, quiere decir, que es capaz de mantener la integridad de los datos, haciendo que estas transacciones no puedan finalizar en un estado intermedio.

Gracias a las transacciones, los datos de una base de datos pueden ser accedidos para realizar inserciones, modificaciones, borrados con seguridad de que todo o se ejecuta completamente o falla y des deshace por completo (rollback). Las transacciones también permiten realizar bloqueo pesimista de tuplas mientras lo usuarios están trabajando con esos datos, evitando que otros usuarios comiencen a utilizar los mismos.

El uso de las transacciones tiene algún inconveniente: cada transacción requiere de una sesión separada, un alto aislamiento en la transacción y bloqueo basado en páginas puede evitar que los usuarios accedan a los datos que deberían estar legítimamente permitidos.

Una alternativa es que los datos sean puestos en un buffer en la capa de persistencia con todas las escrituras que se harán hasta que el usuario lo confirme. Esto requiere un esquema de bloqueo optimista donde las tuplas son chequeadas mientras se escriben.

Transacción como motor de bases de datos

Una transacción es un grupo de operaciones que se realizan juntas como un todo, como una lógica de trabajo, en caso de error de alguna de sus operaciones, no se realiza el resto de las operaciones. Por definición, una transacción tiene que cumplir cuatro propiedades: **atomicidad, coherencia, aislamiento y durabilidad (ACID)**.

- **Atomicidad:** una transacción debe ser atómica si se realizan cambios o no, es decir, si se realizan cambios que sean todas las operaciones definidas, en caso de que alguna de ellas falle, se deshacen todas las anteriores.

- **Coherencia:** una vez haya finalizado una transacción, los datos tienen que ser coherentes. Debe cumplirse la integridad referencial de todos los datos en la base de datos relacional.

- **Aislamiento:** las modificaciones realizadas por transacciones simultáneas deben estar aisladas de otras transacciones que se ejecuten al mismo tiempo.

- **Durabilidad**: una vez concluida la ejecución de la transacción, sus efectos se deben reflejar en la base de datos, aunque se produzca algún error posterior a la transacción.

V. FRAMEWORK DE PERSISTENCIA: WAY PERSISTENCIA

1. Justificación del Trabajo Fin de Carrera y Contexto en el que se desarrolla

Los frameworks de persistencia son una solución para la capa de datos de cualquier proyecto de software que permite abstraernos en gran medida del almacenamiento de datos.

En la programación en Java, puede optarse por la conexión a la base de datos a través de JDBC, pero no nos aportará las ventajas que nos ofrece un framework de persistencia. Ventajas como la velocidad de desarrollo, la eficiencia, la eficacia que hacen más fructífero y estable nuestro proyecto software.

En cuanto a persistencia, existen diferentes opiniones de cual es la solución más adecuada o correcta de implementar. En general, la utilización de un ORM u otro depende de la robustosidad y magnitud de un proyecto en cuestión, puesto que aunque un ORM pueda ser muy potente, para una aplicación sencilla, tardará excesivo tiempo en encargarse de gestionar la capa de persistencia por la gran cantidad de datos que tendrá que tratar con su propio motor de persistencia.

Para realizar "WayPersistence" ORM he sopesado los pros y las contras de los distintos ORM que hay en la actualidad, fundamentalmente para la tecnología Java. Después del análisis de importantes motores de persistencia como Hibernate, Ibatis, Torque, TopLink, Cayenne, etc, me di cuenta que todos tenía sus particularidades, pero algunos de ellos tiene demasiadas dependencias con otras librerías o APIs de Java.

La relevancia y el enfoque que se quiere dar a este ORM no es otra más que ofrecer una capa de persistencia alternativa a los ORM actuales, que permita ofrecer con garantías un acceso a datos en un desarrollo de software sin demasiadas dependencias de otras apis, con eficiencia y rapidez.

Como fortaleza fundamental del ORM es que podemos realizar operaciones de consultas e inserciones a bases de datos sin necesidad de tener que cargar demasiadas dependencias en el proyecto, pudiendo acceder a la capa de datos con ayuda de un driver de la gran mayoría de Sistemas Gestores de Bases de Datos (SGBD) como: MySQL, Oracle, Informix, etc.

Como debilidades, la falta de un lenguaje personalizado de consultas que permita realizar las mismas sobre objetos y no sobre el modelo relacional (base de datos). Esta característica tan fundamental y potente la posee por ejemplo: Hibernate en su 3ª versión, y permite a un equipo de desarrollo de software mayor seguridad y menos dificultad en la generación de líneas de código.

Es necesario realizar este proyecto para contrastar los distintos ORM y no ORM del mercado para tener un conocimiento genérico de todos ellos y además, que se contemple una alternativa sencilla, simple, concreta y funcional de una framework que posee las características fundamentales para ser utilizado en cualquier aplicación de gestión.

2. Objetivos Generales y específicos

Los objetivos generales son adquirir y reforzar los conocimientos en las tecnologías estudiadas y los distintos frameworks de persistencia que existen en el mercado. No sólo el mero conocimiento de las mejores y más prácticas capas de persistencia del mercado, sino la investigación de la mayor parte de ellas, para extraer información que nos ayude a construir nuestro propio framework con una idea más general y global sin pecar de desconocimiento de todas las alternativas y posibilidades que existen en la actualidad. De esta manera, tendremos una visión de ORM mucho más específica y profundizada y podremos analizar cuando es conveniente utilizar un tipo de framework u otro en función de la base de datos, el proyecto en cuestión o la estructura del mismo.

El objetivo de la creación de un framework de persistencia no es otro que mejorar la productividad de mi aplicación a través de la reutilización de código con una librería de clases java que facilita la tarea al programador al permitirle guardar objetos en bases de datos relacionales de manera lógica y eficiente. El proyecto a realizar tiene el objetivo de investigar en los distintos frameworks de persistencia que existen en el mercado para Java y crear uno para integrarlo en el desarrollo de una aplicación en Java.

El beneficio de hacer una investigación sobre este tipo de frameworks es útil porque se logra demostrar la viabilidad de usar frameworks apoyados en patrones de diseño para el desarrollo de software para entorno web. Esta tecnología nos permitirá abstraer la capa de acceso a datos independientemente de la base de datos que utilizaremos con diferentes bases de datos sin que sea necesario hacer demasiados cambios en el entorno.

A nivel del proyecto como objetivo específico quiero conseguir que para cualquier aplicación Java de tamaño medio se pueda utilizar un framework de persistencia propio que no consuma tantos recursos como hibernate, pero que además de ser sencillo aporte rendimiento, abstracción y sobre todo utilidad. Que se pueda utilizar para cualquier base de datos, principalmente los Sistemas gestores de bases de datos más conocidos: Oracle, MySql, Access, Informix con tan la utilización agrupación de algunas clases Java que formarán el framework de persistencia.

Por otra parte, a nivel profesional creo que investigar sobre este tema me beneficiará de cara a enfrentarme en la vida profesional con cualquier tipo de ORM o capa de persistencia. Esto reforzará mucho mi nivel de capa de datos de cualquier proyecto en general. Tengo interés en realizarlo porque creo que el acceso a datos es fundamental para cualquier aplicación, no sólo la velocidad del acceso a datos, sino también la escalabilidad, la abstracción, la adaptación y posibilidad de reutilización del código para proyectos similares y por supuesto la optimización, el manejo y la precisión de las capas de persistencia en los proyectos. Es fundamental que el mapeo de los datos se pueda regenerar con la ayuda de herramientas como ant cuando la base de datos sufra cambios.

Objetivos generales

- Profundizar y conocer las soluciones de persistencia que existen en la actualidad y averiguar en que caso es conveniente utilizar cada una de ellas y cual es la mejor elección a escoger.

- Una vez estudiadas las alternativas disponibles, hacer el diseño y la implementación de un framework de mapeo entidad-relación sencillo y eficiente.
- Construir una aplicación de ejemplo que muestre el uso del framework implementado.

Objetivos específicos

- Conocer una alternativa a utilizar como motor de persistencia de acceso a datos para cualquier proyecto de gestión independientemente del sistema gestor de base de datos que se utilice trabajando con bases de datos relacionales.

- Aportar una alternativa en lo que a los motores de persistencia se refiere y dar una visión de que se puede personalizar la capa de datos para cualquier tipo de proyecto.

3. Enfoque y método seguido

Todo el desarrollo ha sido guiado con la metodología orientada a objetos y el enfoque que se le ha querido dar al proyecto no es otro más que hacer un framework de persistencia para poder cubrir la capa de acceso a datos de cualquier proyecto software de la magnitud que sea independientemente del sistema gestor que se utilice.

Independiente del sistema gestor de bases de datos a utilizar en el desarrollo, destacamos que el motor de persistencia es y será libre dependiente de sus tecnologías Java bajo licencia GNU GPL.

Sun Microsystems, desarrolló a principios de los años 90 el lenguaje de programación orientado a objetos Java. En el año 95 implementó la referencia del compilador, la máquina virtual y las librerías de clases java. Desde ese momento, Sun controla todas las especificaciones, el desarrollo y la evolución del lenguaje Java a través de una comunidad denominada *Java Community Process.* A finales del año 2006 y/o principios de 2007, Sun Microsystems liberó casi todas las tecnologías Java con la etiqueta de licencia GNU GPL, siguiendo las especificaciones del grupo – comunidad citada anteriormente.

Así pues, prácticamente casi todo el Java de Sun es software libre, a excepción de la biblioteca de clases que se requiere para ejecutar los programas Java que todavía sigue siendo de su propiedad.

Con todos estos datos, podemos afirmar que el proyecto "WayPersistence ORM" es un software libre que implementa un framework de persistencia desarrollado única y exclusivamente en Java para soportar la capa o motor de persistencia de datos de cualquier proyecto software independientemente del sistema gestor que se utilice. Este framework permitirá mapear el modelo entidad – relación o modelo relacional soportado por la base de datos a un modelo de datos orientado a objetos representado según las especificaciones y características de nuestro ORM "WayPersistence".

El enfoque fundamental y final que se pretende dar con este tipo de tecnología es tener un motor de capa de datos alternativo a los actuales en el mercado, con menor dependencia de librerías, con más simplicidad, escalable, multiplataforma, robusto y por supuesto muy

funcional y práctico. La versión a lanzar será la 1.0, pero tras un período de pruebas con esta versión, se podrá paliar los problemas que ocurran para una futura versión que garantice aún más la efectividad y eficacia de esta tecnología alternativa al JDBC estándar.

4. Planificación del Proyecto

4.1. Descripción general

En mi PFC he escogido el tema "Framework de Persistencia", es decir encargarme de una de las capas de cualquier aplicación desarrollada en J2EE que se encarga de los servicios de persistencia y la recuperación de la información para las capas superiores.

Antes de plantearse un acceso a datos propio debemos estudiar y analizar todas las posibilidades que nos ofrece el mercado principalmente para el desarrollo en aplicaciones Java, concretamente en el acceso a datos.

En un primer momento, dentro de las posibilidades que nos ofrecen las capas de persistencia tenemos las capas de EJBS de entidad (BMPs o CMPs), los beans normales de acceso a datos, los frameworks de persistencia de mapeo O/R, o incluso el conjunto de conectores a otros sistemas como los sistemas LEGACY, el acceso a SAP y BIW a través de su Business Connector o de JCO.

Si citamos los beans, su capa de acceso a datos se trata de ejecuciones de sentencias SQL contra el motor de la persistencia.

En general los frameworks de persistencia son herramientas de mapeo Objeto/Relación (O/R) aunque no todas, tienen fama de bajo rendimiento, desarrollo rápido de aplicaciones, se pueden desarrollar aplicaciones con más portabilidad a otras bases de datos, aunque siempre depende de la abstracción del propio framework.

Otra alternativa de acceso a datos es la externalización de SQL, puesto que es diferente en cada sistema gestor de bases de datos que utilices, principalmente si hablamos de consultas optimizadas, se pueden utilizar ficheros XML.

Para desarrollar mi proyecto en primer lugar como he comentado anteriormente haré un estudio de las distintas tecnologías y/o frameworks en java que se encargan de la parte de persistencia o acceso a datos.

Como framework de prueba en un primer lugar utilizaré un framework basado en Spring, o incluso el propio framework de Spring con algunas modificaciones para hacer pruebas con los diferentes frameworks de persistencia del mercado, las virtudes que aportan y los problemas que puedan ocasionar.

En un primer momento utilizaremos Hibernate con sus posibilidades de integración con los objetos relacionales a través de mapeados en xml, que junto con su HQL (Hibernate Quero Language) hacen el que las implementaciones DAO con este framework sean exitosas debido a la potencia y ventaja que aporta le HQL, lenguaje propio de hibernate basado en SQL que en lugar de hacer consultas directamente a la base de datos, lo hace a objetos relacionales o clases del proyecto en cuestión.
Hibernate es una ORM (Object Relational Mapping) de libre distribución de las más maduras y completas. Hace uso de APIs de Java tales como JDBC, JTA (Java Transaction

API), y JNDI (Java Naming Directory Interface), podemos nombrarlas las interfaces utilizadas por hibernate: Session, SessionFactory, Configuration y Query.

En un segundo lugar, probaremos JDO, es una API de SUN que permite trabajar con objetos normales de java también llamados POJOs (plain old java objects), en lugar de hacerlo con APIs propietarios. JDO corrige la persistencia en un paso d mejora de bytecodes posterior a la compilación, proporcionando una capa de abstracción entre el código de la aplicación y el motor de persistencia. Los problemas de JDO es que se solapa con EJB y CMP, a pesar de ser una enorme mejora sobre las tendencias actuales de tecnologías objeto – datos y datos - objeto.

Además, intentaremos probar con Oracle TopLink, que se basa en la arquitectura Objeto-a-relacional de la persistencia de Java, es flexible y productivo para almacenar objetos y Ejes de java en bases de datos relacionales y para convertir entre los objetos y los documentos de XML de java (JAXB). TopLink ofrece a los desarrolladores un funcionamiento excelente sea cual sea la base de datos, el servidor y cualquier tipo de herramientas de desarrollo dentro de J2EE.

Intentaremos hacer pruebas con Ibatis, que junto con Hibernate serán los frameworks a los que más tiempo le dedicaremos por su importancia. Por parte de Ibatis, al contrario de Hibernate, requiere que el programador conozca en profundidad SQL, permite la optimización de las consultas y contiene herramientas para evitar el problema de las "N+1 consultas" y para generar consultas dinámicas muy potentes. Aunque el modelo relacional cambie mucho, IBATIS es un claro caso de uso y puede adaptarse sin problema. Hoy en día tiene una gran importancia, pertenece al proyecto Apache y es de marcos de acceso a datos más utilizados. Está compuesto por dos partes: la capa DAO o capa general que abstrae el acceso a datos y el SQL Maps la que envuelve las llamadas a JDBC definidas en XML y realiza el mapeo entre objetos (javabeans) y SQL (Statements o StoredProcedures). No es un ORM puro, tenemos que escribir el SQL y el mapeo se realiza a partir de los resultados SQL. Sus grandes ventajas son la estabilidad y facilidad para encontrar el problema, simplicidad, rendimiento por tener caché configurable y gran nivel de abstracción.

Por último, JPA o Java Persistente API, conocida como JPA, es la API de persistencia desarrollada por la plataforma Java EE e incluida en el estándar EJB3. Su objetivo es unificar la manera en que funcionen las utilidades que proveen un mapeo objeto – relacional (O/R). El objetivo de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos y permitir utilizar objetos regulares conocidos como POJOs.

En principio estas tecnologías citadas anteriormente serán las pruebas de tecnologías que probaremos para ver las posibilidades que nos ofrecen, sus ventajas y problemas que puedan ocasionar. Además, probaremos con algún otro marco de persistencia como: Container-managed Entity EJB, Castor, Kodo, Torque, Cayenne, TJDO, JDBM, Jaxor, OJB, etc.

La prueba de mi aplicación en principio la haré utilizando la tecnología Spring que me aporta los beneficios: manejo de transacciones declarativo, cierre de de conexiones de bases de datos transparente al usuario y plantillas para generar los DAOs usando simplemente JDBC o los principales ORMs del mercado principalmente Hibernate y un no ORM, Ibatis que con conocimientos de SQL es muy potente y estable.

Comentaré las diferencias fundamentales entre los dos frameworks de persistencia más utilizados para Java combinándolos con Spring: Ibatis y Hibernate.

Capa de Persistencia utilizando Spring con Ibatis o Hibernate

Comentaré algunas diferencias fundamentales entre Ibatis e Hibernate combinándolos con Spring.

La capa de persistencia para Ibatis tiene una serie de pasos, si por ejemplo utilizamos una tabla de Usuarios los pasos serían:

- Generar el bean Usuario que mapee una fila de la tabla Usuarios (Patron Active Record).
- Generar una interfaz para la fachada (Patrón Facade) de la aplicación y una por cada tabla de la base datos (patrón DAO) con las operaciones habituales a realizar en cada una de las tablas.
- Realizar los archivos XML de Ibatis que mapeen la base de datos y las consultas de los distintos métodos de la lógica de negocio.
- Realizar el archivo de configuración de Ibatis.
- General la implementación de los DAOs utilizando Spring.

Capa de persistencia de Hibernate

Como comentamos anteriormente Hibernate es la herramienta ORM más comentada de los últimos años, aunque también la más compleja. Spring ofrece ayudas para utilizar hibernate. Si intentamos mapear una relación M:N supondremos la existencia de una tabla User que contendrá los datos de los usuarios. Cada tupla de los usuarios tendrá la forma {id, nombre, apellidos} donde id será una clave autonumérica. También tendremos la existencia de la tabla User_Role que servirá de enlace entre ambas tablas. La forma de cada una de las tuplas será {idUser_Role, user_id, role_id} donde idUserRole será la clave autonumérica. Por último tendremos Role que tendrá la estructura en sus tuplas {id_role, name_role} con id_role de tipo autonumérico. En la tabla User_Role, los campos user_id y role_id serán claves foráneas de las otras dos tablas.

Los pasos a realizar para configurar hibernate con Spring es.

- Generar un bean por cada tabla de la BD que mapee una fila de la tabla (Patrón Active Record). En este caso generaremos un bean por cada DAO de User, UserRole y Role. Este bean apuntará a una clase que será donde reflejaremos las operaciones básicas de acceso a datos: update, select, delete, etc.
- Generar una interfaz de la fachada de la aplicación (facade) y una interfaz por cada tabla de la base de datos con las operaciones a realizar sobre la tabla (patrón DAO).
- Generar los archivos de XML de mapeo de Hibernate. Aquí iremos introduciendo los beans necesarios de cada tabla que vayamos creando, cada una de las tablas en general tendrá un bean que llamará a una clase donde se representan sus operaciones más habituales, ejemplo: findById (Integer id), findAll(), getUserDetail(Integer id), etc.
- Realizar el archivo de configuración de Hibernate, normalmente se suele llamar application-context-hibernate.xml.
- Generar una implementación de la clase fachada e implementar los DAOs de la aplicación utilizando la plantilla de Spring para Hibernate.

En definitiva para la creación de un framework de persistencia propio en primer lugar se deben conocer la mayor parte de los existentes en la actualidad, analizar las ventajas y desventajas que tiene cada uno de ellos e intentar sacar alguna conclusión que nos lleve a pensar como hacer un conjunto de clases que realicen funcionalidades similares a bs estudiados, es decir, un framework de persistencia que nos permita probar una mini aplicación que puede ser el acceso de unos usuarios a un sistema, el mostrado de unos listado y las modificaciones y actualizaciones básicas de dichos usuarios.

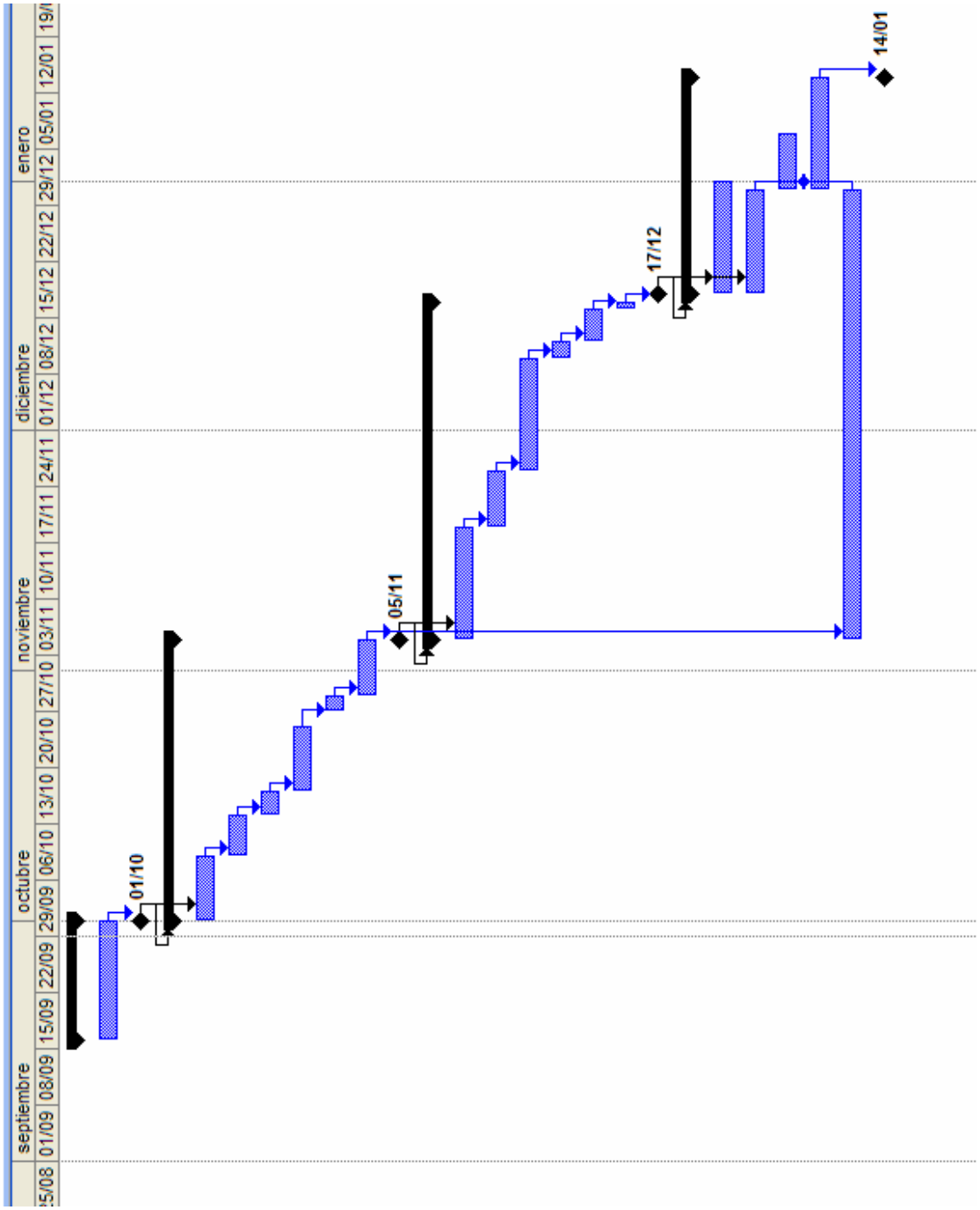
4.2. Plan de trabajo con Hitos y Temporización

La planificación que llevaremos a cabo en nuestro proyecto será de 11 días para el Plan de Trabajo Inicial con un hito de entrega el día 1 de Octubre de 2008. Para las comparativas de las distintas tecnologías que estudian frameworks en Java dedicaremos 25 días, siempre contabilizamos días laborables. El hito de entrega o entregable de esta tarea es el día 5 de Noviembre, coincidiendo con la entrega de la PEC2. Las subtareas que cuelgan de "Comparativas de Frameworks Java" son hibernate, JDO, TopLink, Ibatis, JPA y otros frameworks como Torque, Cayenne, etc.
Otra tarea dentro del Plan de Planificación del proyecto es el Diseño del Framework de Persistencia y la creación de la miniaplicación. Esta tarea llevará sobre 30 días, el entregable coincide con la PEC3 que será el 60% del código fuente y el diseño del framework.
Por último tenemos la entrega del proyecto en Enero donde concluiremos con la finalización de la memoria, la resolución de incidencias, y demás tareas relativas al proyecto. La fecha fin – entrega del proyecto es: 14 de Enero de 2008.

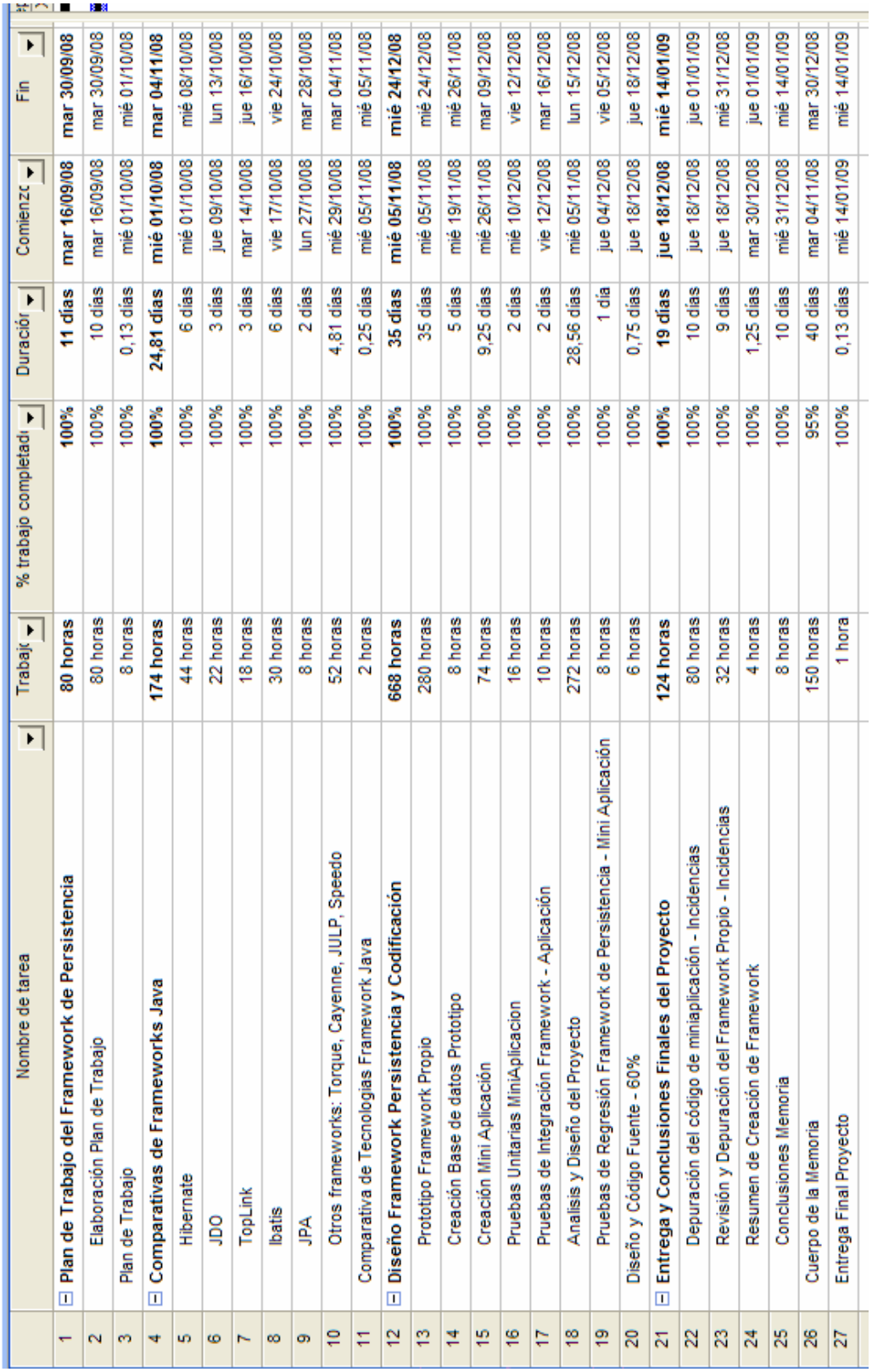
Planificación por Tareas y estimaciones.

Nombre de tarea	Duración	Comienzo	Fin	Predece
Plan de Trabajo del Framework de Persistencia	**11 días**	**mar 16/09/08**	**mar 30/09/08**	
Elaboración Plan de Trabajo	11 días	mar 16/09/08	mar 30/09/08	
Plan de Trabajo	0 días	mié 01/10/08	mié 01/10/08	2
Comparativas de Frameworks Java	**25 días**	**mié 01/10/08**	**mar 04/11/08**	**3**
Hibernate	6 días	mié 01/10/08	mié 08/10/08	3
JDO	3 días	jue 09/10/08	lun 13/10/08	5
TopLink	3 días	mar 14/10/08	jue 16/10/08	6
Ibatis	6 días	vie 17/10/08	vie 24/10/08	7
JPA	2 días	lun 27/10/08	mar 28/10/08	8
Otros frameworks: Torque, Cayenne, JULP, Speedo	5 días	mié 29/10/08	mar 04/11/08	9
Comparativa de Tecnologias Framework Java	0 días	mié 05/11/08	mié 05/11/08	10
Diseño Framework Persistencia y Codificación	**30 días**	**mié 05/11/08**	**mar 16/12/08**	**11**
Prototipo Framework Propio	10 días	mié 05/11/08	mar 18/11/08	11
Creación Base de datos Prototipo	5 días	mié 19/11/08	mar 25/11/08	13
Creación Mini Aplicación	10 días	mié 26/11/08	mar 09/12/08	14
Pruebas Unitarias MiniAplicacion	2 días	mié 10/12/08	jue 11/12/08	15
Pruebas de Integración Framework - Aplicación	2 días	vie 12/12/08	lun 15/12/08	16
Pruebas de Regresión Framework de Persistencia - Mini Aplicaci	1 día	mar 16/12/08	mar 16/12/08	17
Diseño y Código Fuente - 60%	0 días	mié 17/12/08	mié 17/12/08	18
Entrega y Conclusiones Finales del Proyecto	**19 días**	**jue 18/12/08**	**mar 13/01/09**	**19**
Depuración del código de miniaplicación - Incidencias	10 días	jue 18/12/08	mié 31/12/08	19
Revisión y Depuración del Framework Propio - Incidencias	10 días	jue 18/12/08	mié 31/12/08	19
Resumen de Creación de Framework	5 días	jue 01/01/09	mié 07/01/09	22
Conclusiones Memoria	10 días	mié 31/12/08	mar 13/01/09	25
Cuerpo de la Memoria	40 días	mié 05/11/08	mar 30/12/08	10
Entrega Final Proyecto	0 días	mié 14/01/09	mié 14/01/09	24

Diagrama de Gantt

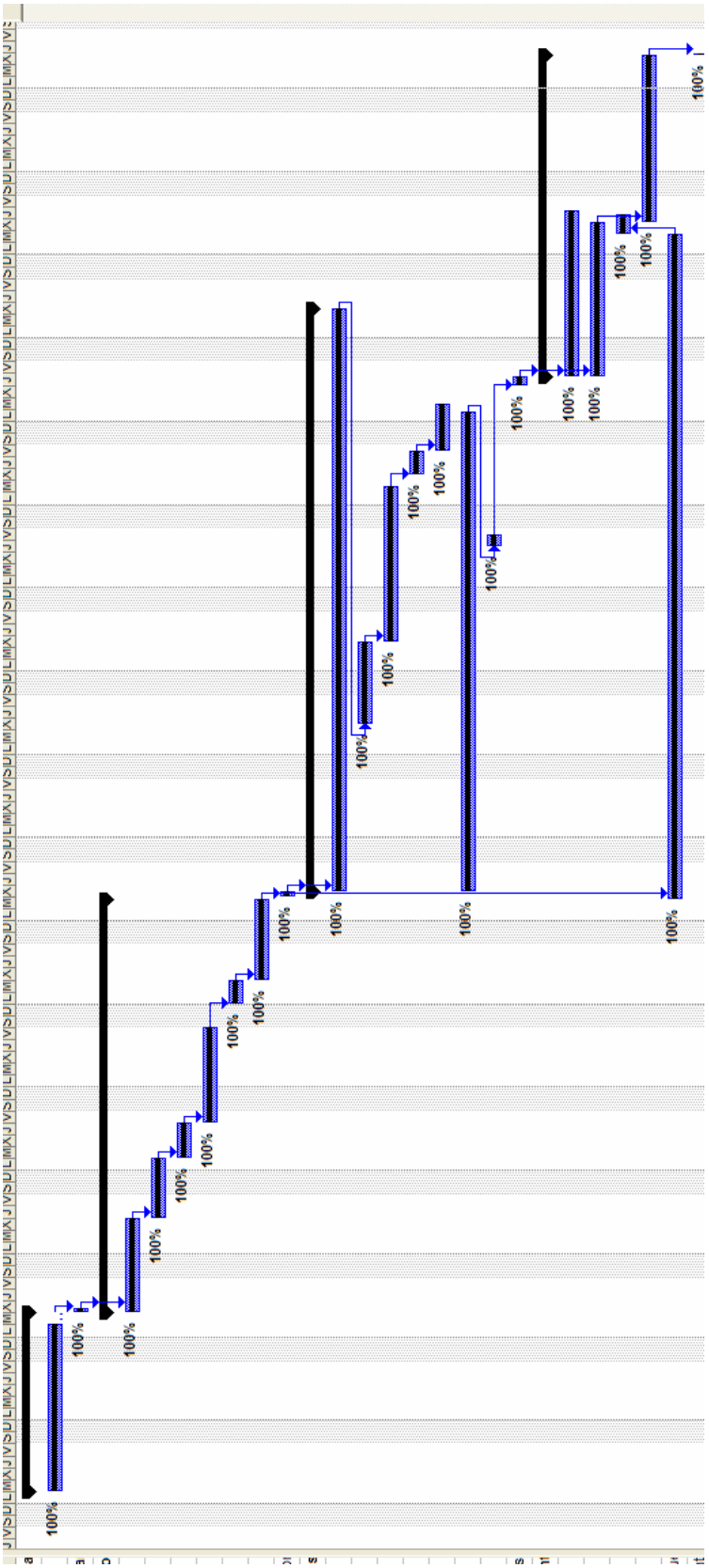


Plan de Seguimiento Final



	Nombre de tarea	Trabajo	% trabajo completado	Duración	Comienzo	Fin
1	**Plan de Trabajo del Framework de Persistencia**	**80 horas**	**100%**	**11 días**	**mar 16/09/08**	**mar 30/09/08**
2	Elaboración Plan de Trabajo	80 horas	100%	10 días	mar 16/09/08	mar 30/09/08
3	Plan de Trabajo	8 horas	100%	0,13 días	mié 01/10/08	mié 01/10/08
4	**Comparativas de Frameworks Java**	**174 horas**	**100%**	**24,81 días**	**mié 01/10/08**	**mar 04/11/08**
5	Hibernate	44 horas	100%	6 días	mié 01/10/08	mié 08/10/08
6	JDO	22 horas	100%	3 días	jue 09/10/08	lun 13/10/08
7	TopLink	18 horas	100%	3 días	mar 14/10/08	jue 16/10/08
8	Ibatis	30 horas	100%	6 días	vie 17/10/08	vie 24/10/08
9	JPA	8 horas	100%	2 días	lun 27/10/08	mar 28/10/08
10	Otros frameworks: Torque, Cayenne, JULP, Speedo	52 horas	100%	4,81 días	mié 29/10/08	mar 04/11/08
11	Comparativa de Tecnologías Framework Java	2 horas	100%	0,25 días	mié 05/11/08	mié 05/11/08
12	**Diseño Framework Persistencia y Codificación**	**668 horas**	**100%**	**35 días**	**mié 05/11/08**	**mié 24/12/08**
13	Prototipo Framework Propio	280 horas	100%	35 días	mié 05/11/08	mié 24/12/08
14	Creación Base de datos Prototipo	8 horas	100%	5 días	mié 19/11/08	mié 26/11/08
15	Creación Mini Aplicación	74 horas	100%	9,25 días	mié 26/11/08	mar 09/12/08
16	Pruebas Unitarias MiniAplicacion	16 horas	100%	2 días	mié 10/12/08	vie 12/12/08
17	Pruebas de Integración Framework - Aplicación	10 horas	100%	2 días	vie 12/12/08	mar 16/12/08
18	Analisis y Diseño del Proyecto	272 horas	100%	28,56 días	mié 05/11/08	lun 15/12/08
19	Pruebas de Regresión Framework de Persistencia - Mini Aplicación	8 horas	100%	1 día	jue 04/12/08	vie 05/12/08
20	Diseño y Código Fuente - 60%	6 horas	100%	0,75 días	jue 18/12/08	jue 18/12/08
21	**Entrega y Conclusiones Finales del Proyecto**	**124 horas**	**100%**	**19 días**	**jue 18/12/08**	**mié 14/01/09**
22	Depuración del código de miniaplicación - Incidencias	80 horas	100%	10 días	jue 18/12/08	jue 01/01/09
23	Revisión y Depuración del Framework Propio - Incidencias	32 horas	100%	9 días	jue 18/12/08	mié 31/12/08
24	Resumen de Creación de Framework	4 horas	100%	1,25 días	mar 30/12/08	jue 01/01/09
25	Conclusiones Memoria	8 horas	100%	10 días	mié 31/12/08	mié 14/01/09
26	Cuerpo de la Memoria	150 horas	95%	40 días	mar 04/11/08	mar 30/12/08
27	Entrega Final Proyecto	1 hora	100%	0,13 días	mié 14/01/09	mié 14/01/09

Diagrama de Gantt Final



5. Análisis y Diseño Orientado a Objetos

5.1. Introducción A/DOO

El análisis y diseño orientado a objetos para WayPersistence ORM requiere de varias premisas: por un lado, obviamente, conocer a fondo el lenguaje orientado a objetos, Java en este caso y como no, saber pensar en "objetos", tener la abstracción de saber que lo que estamos desarrollando o diseñando será implementado en objetos y no en la antigua programación estructurada.

El análisis y diseño orientado a objetos (A/DOO) es fundamental para crear cualquier tipo de software propio, como por ejemplo este ORM. Además, facilita su mantenimiento, su buena definición y garantiza una robustez que no se obtiene sin la utilización de estas herramientas.

Esta forma de analizar los distintos tipos de proyectos de desarrollo de software se centra fundamentalmente en asignar responsabilidades a los objetos aplicando el lenguaje UML (Unified Modeling Language) o Lenguaje Unificado de Modelado y algunos de los patrones de diseño más habituales. La idea fundamental de este análisis y diseño de objetos es como una forma de tener "planos de software" de un desarrollo o proyecto concreto, construir sistemas de objetos siguiendo unos patrones, más que una notación sin más.

El A/DOO está relacionado con el Análisis de Requisitos, que incluye dibujar casos de uso y transformar dichos requisitos en un lenguaje más acorde con los programadores para que se adapten a la tecnología en cuestión sin tener una visión global que no necesitan.

5.2. Definiciones

Análisis

Se centra en el descubrimiento y exploración del problema. No significa resolver el problema, sino hallar la manera de hacerlo, siguiendo unos pasos, con unos requisitos, después de haberlo estudiado. Podríamos decir que se realiza un análisis de requisitos previamente definidos para interpretarlos y transformarlos en algún tipo de representación UML o diseño, como por ejemplo, casos de uso.

Diseño

Una solución que cumpla con los requisitos a nivel esquemático. Un diseño podría ser el modelo de datos de una base de datos o el modelo de objetos de su mapeado en la codificación.

Si nos centramos en objetos:

- Análisis orientado a objetos: se centra en especificar que es cada objeto, de que se compone, qué conceptos tenemos como objetos.
- Diseño orientado a objetos: se presta atención a la definición de objetos de tipo software y cómo lograr satisfacer los requisitos.

Para realizar un análisis detallado del proyecto o producto en cuestión, en este caso un ORM, realizamos un análisis Funcional para determinar los Procesos a implementar representados en casos de uso con sus diagramas pertinentes.

5.3 ANÁLISIS FUNCIONAL: ESPECIFICACIÓN DE CASOS DE USO Y DIAGRAMAS DE SECUENCIA

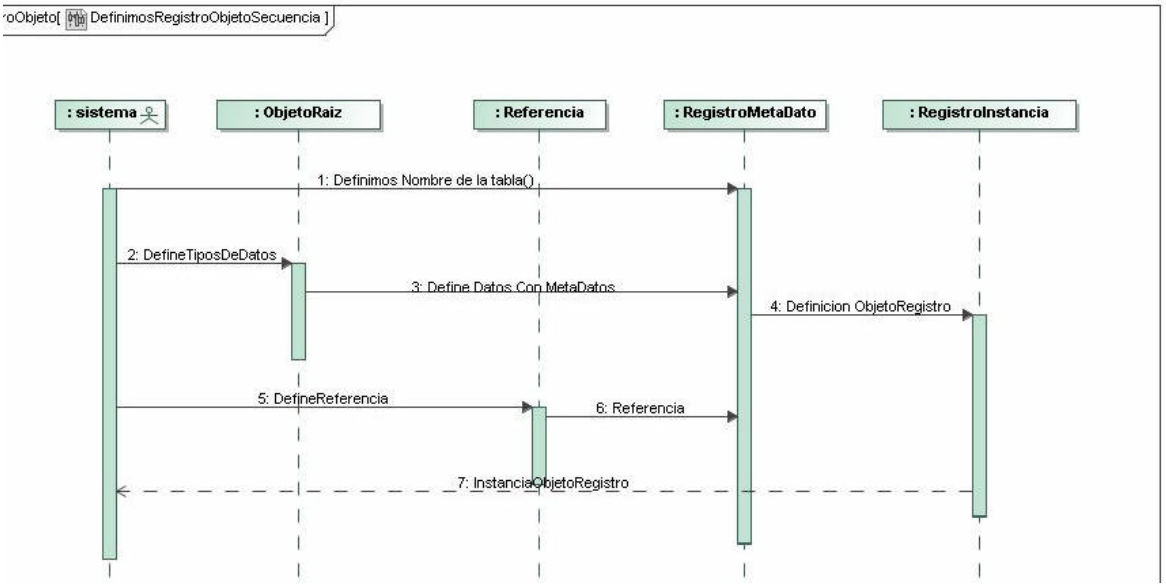
Introducción

WayPersistence ORM es un framework de persistencia desarrollado en Java para ofrecer una alternativa como capa de persistencia sobre los ORM de acceso a datos actuales.

Objetivo

El objetivo de este documento de Análisis Funcional consiste en presentar la descripción conceptual de aquellos procedimientos relacionados dentro del contexto de WayPersistence, empaquetados por grupos según su funcionalidad. Estos grupos, los denominamos Procesos en este análisis. Cada proceso estará agrupado por funcionalidades, y cada una de ellas tendrá un caso de uso asociado. Se representará tanto el diseño del diagrama de casos de uso como el diagrama de secuencia de cada una de las distintas funcionalidades.

Alcance



Dentro del alcance del desarrollo de este ORM, se incluyen los siguientes procedimientos:

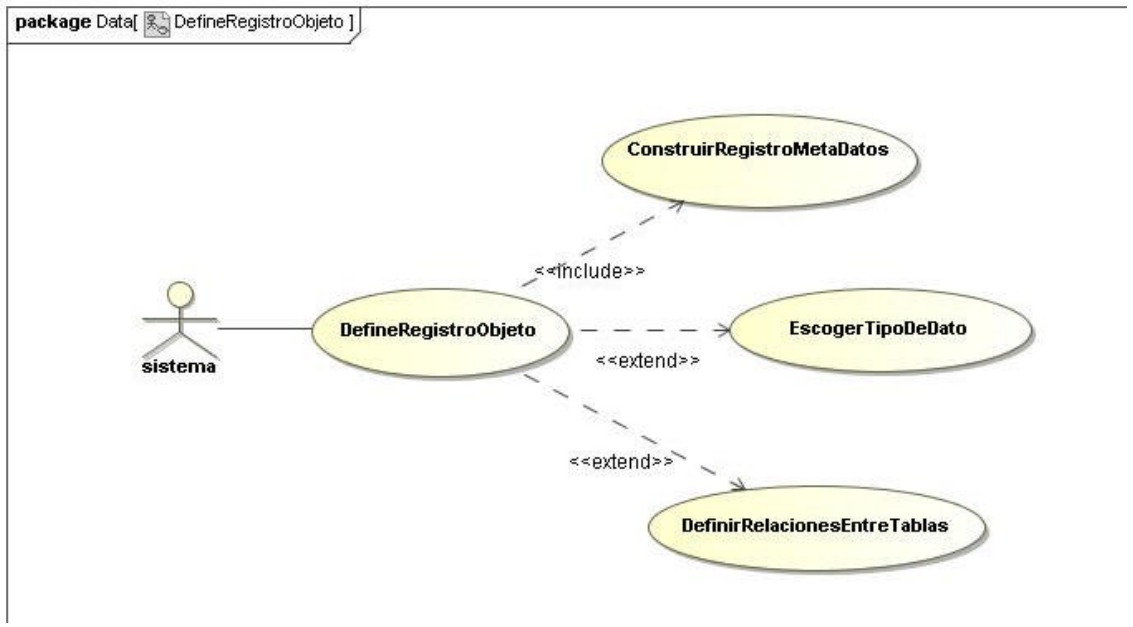
- Registro de Objetos.
- Acceso y tratamiento de datos.
- Validaciones de datos.
- Drivers de conexión.
- Motor del framework.
- Utilidades.

1. PROCESO DE REGISTRO DE OBJETOS

Descripción

El proceso de registro de objetos se basa en la definición del mapeo de objetos que será correspondido con similitud en el modelo relacional representado en la base de datos. El registro de objetos se centrará en la definición de un objeto que tenga simetría con lo definido en una base de datos. Esta simetría se produce a nivel tabla, siendo un objeto de tipo registro un mapeado objeto – relación de una tabla de una base de datos.

Especificación Caso de Uso: CU DefineRegistroObjeto



En este caso de uso, se especificarán los registros u objetos que serán similares a una tabla de la base de datos pero con representación a nivel de objetos. Esta representación será la que nos permita reconocer el mapeado de objetos de una tabla que a su vez es representada en al base de datos. En otros sistemas framework se suelen llamar Pojo's y en algunas ocasiones son mapeados con extensiones de tipo .hbm (ejemplo Hibernate).

Actor Principal: sistema que utiliza esta clase para especificar un objeto que será el que represente a nivel objetos la tabla de base de datos.

Actor Secundario: no tiene

Precondiciones: debería tener definido y referenciado una base de datos a la que se haga referencia para poder realizar el mapeado de objeto-relación.

Postcondiciones: el registro objeto será la representación de una tabla en el modelo de datos orientado a objetos. Nos permitirá trabajar sobre la base de datos sin tener que acceder a ella cada vez que queremos un dato de la misma.

Flujo principal:

Acción de actor	Respuesta del sistema
El sistema registra un objeto de tipo registro.	El sistema identifica que ese objeto es el mapeo de una tabla o entidad del modelo relacional. Se corresponderá en tipos de datos y tamaño de los mismos teniendo una completa simetría.

Casos de uso relacionados: todos los que se refieren a la definición de tipos de datos porque cuando definimos un objeto registro necesitamos especificar las propiedades de cada uno de los tipos de datos y también el CU RelaciónEntreObjetos o referencia.

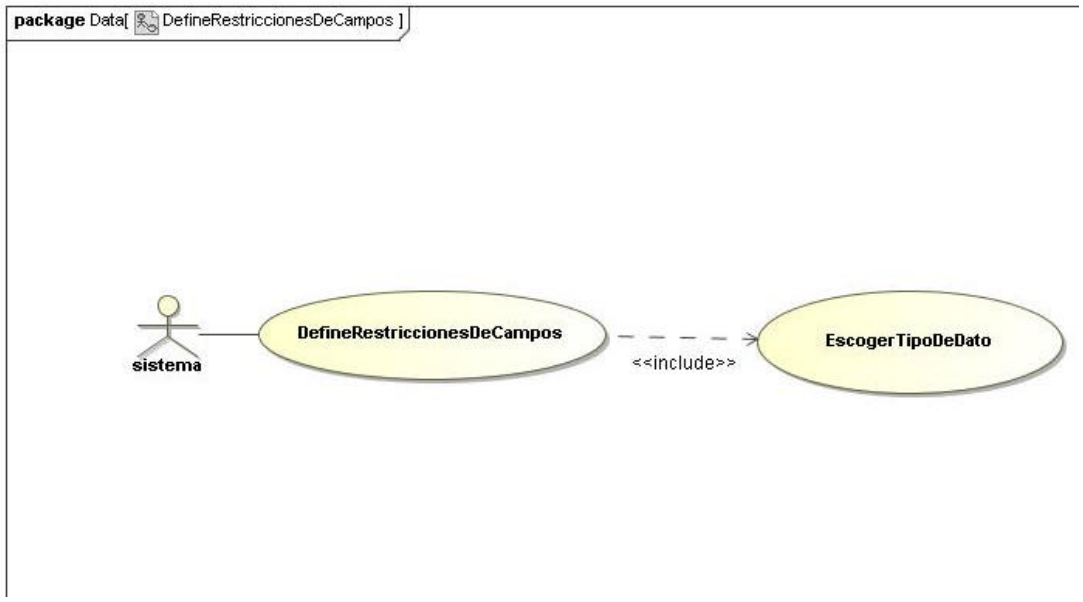
2. PROCESO DE ACCESO Y TRATAMIENTO DE DATOS

Descripción

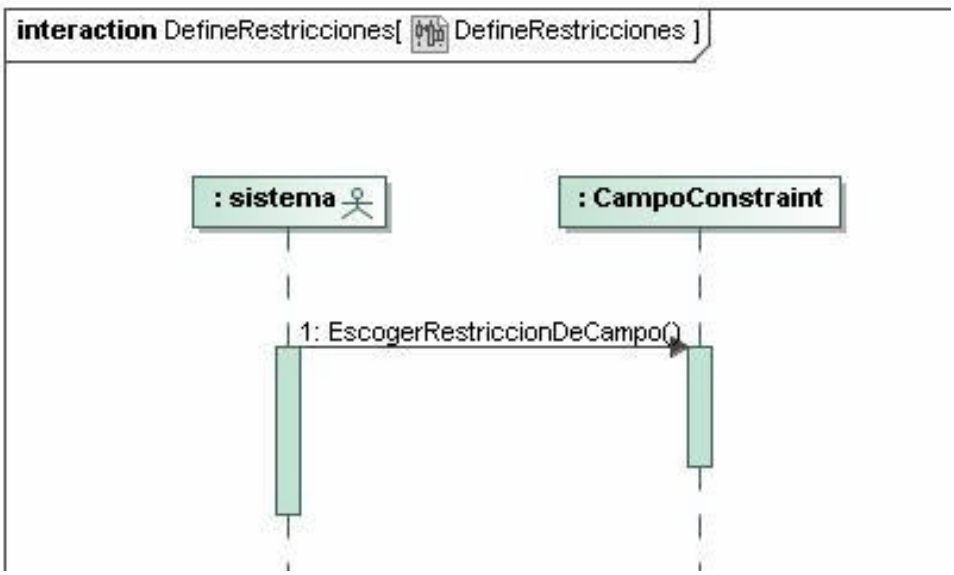
El proceso de acceso a datos y el tratamiento de los mismos se basan en el manejo de todos los tipos de datos a utilizar en el ORM, su estructuración y las relaciones entre ellos.

Especificación Caso de Uso: CU DefineRestriccionesDeCampos

En este caso de uso, se especificarán los distintos tipos de restricciones o características que pueden tener los campos en función de la base de datos que se haya escogido. Se estudiarán los tipos de restricciones más genéricas y comunes como “clave primaria”, “no nulo”, etc.



Actor Principal: sistema que utiliza esta clase para especificar alguna restricción a alguna



columna o propiedad.

Actor Secundario: no tiene

Precondiciones: debería tener definido algún tipo de dato en una tabla o entidad para poder especificar alguna restricción. Tiene que ser un tipo de dato como “CampoCadena”, “CampoDouble”, “CampoEntero”, etc.

Postcondiciones: la propiedad de la tabla contendrá la restricción especificada según lo escogido.

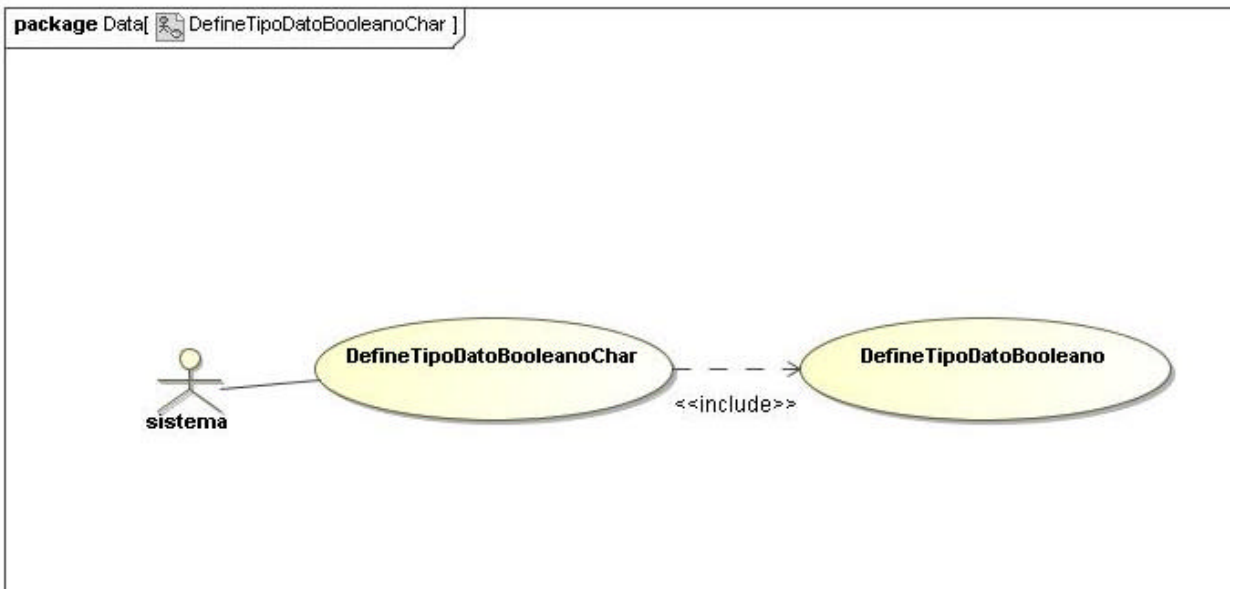
Flujo principal:

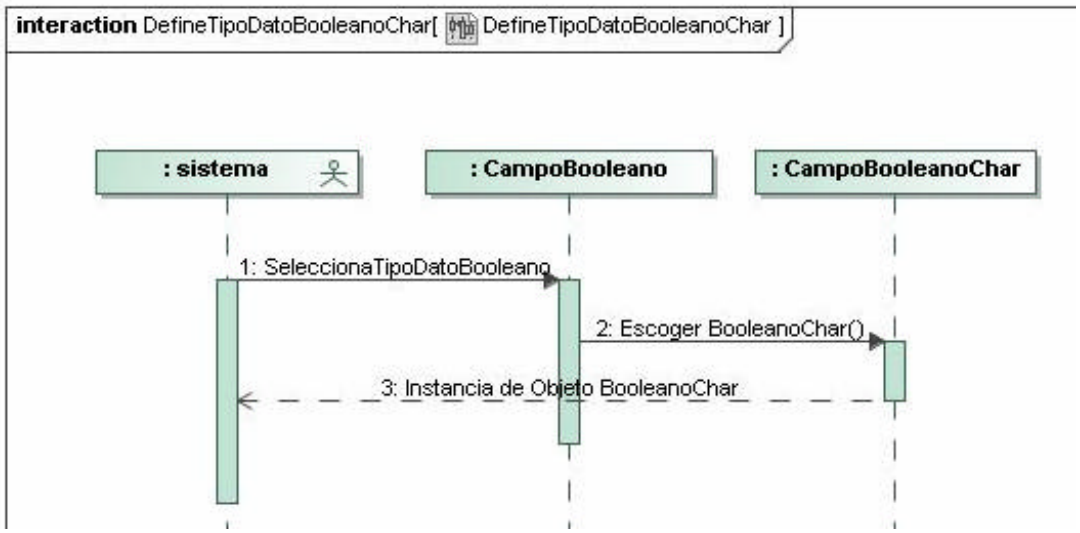
Acción de actor	**Respuesta del sistema**
El sistema solicita un cambio de restricción o constraint.	El sistema si identifica que es una propiedad de una tabla con un tipo de dato propio del framework, le añade una restricción.

Especificación Caso de Uso: CU DefineTipoDatoBooleanoChar

En este caso de uso, se definirá una clase que nos permita especificar un tipo de datos booleano siempre que este puede contener datos de tipo char o cadena, por ejemplo, “Y”, “N”, “Yes”, “No”, etc.

Este tipo de dato extenderá de uno genérico representado en otro caso de uso
<extend CU DefineTipoDatoBooleano>





Actor Principal: sistema que utiliza esta clase para especificar una columna de la base de datos de este tipo, o en su caso del modelo de objetos.

Actor Secundario: no tiene

Precondiciones: debería tener definido algún tipo de dato en una tabla o entidad para poder especificar un tipo de dato. Tiene que ser un tipo de dato que pueda ser mapeado por este en el modelo relacional.

Postcondiciones: el tipo de dato podrá ser booleano char, del tipo “Y”, ”N”, etc.

Flujo principal:

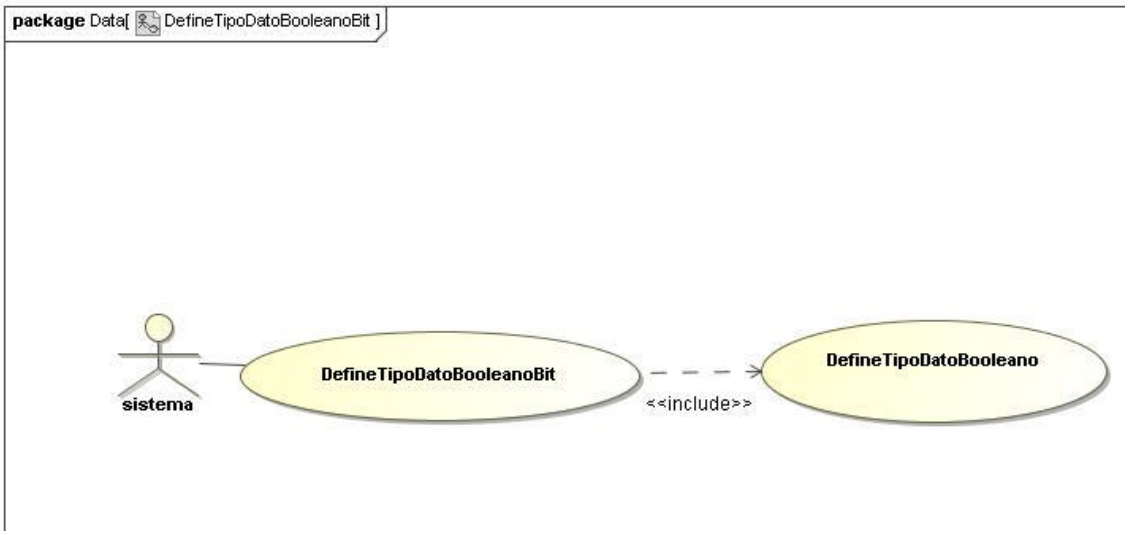
Acción de actor	**Respuesta del sistema**

El sistema propone definir un tipo de dato de tipo CampoBooleano Char	El sistema identifica el tipo de dato de tipo Char y Booleano poniéndole este tipo y permitiendo que actúe como tal en el ORM extendiendo de Booleano genérico que a su vez extenderá de una clase que define los tipos de datos de todo el motor de persistencia.

Especificación Caso de Uso: CU DefineTipoDatoBooleanoBit

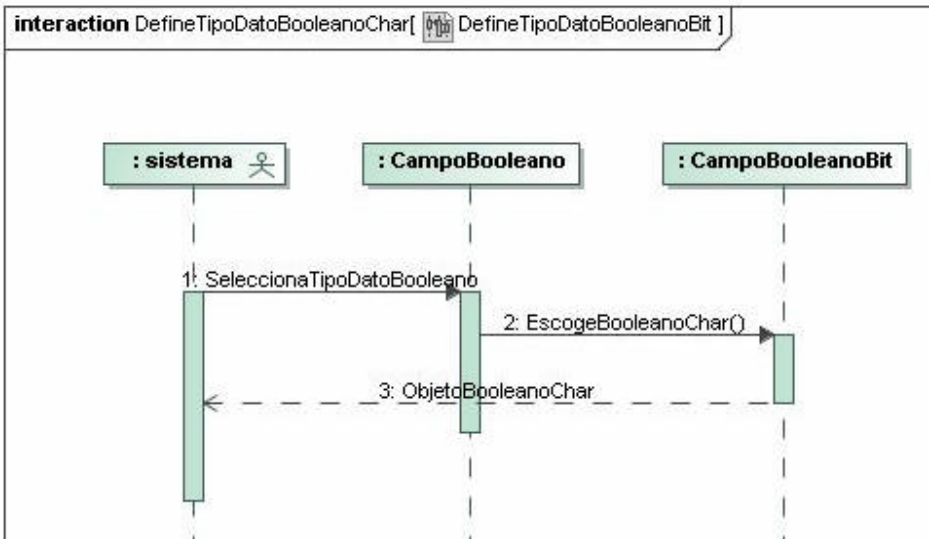
En este caso de uso, se definirá una clase que nos permita especificar un tipo de datos booleano siempre que este puede contener datos de tipo bit o numérico de un solo dígito.

Este tipo de dato extenderá de uno genérico representado en otro caso de uso <extend CU DefineTipoDatoBooleano>



Actor Principal: sistema que utiliza esta clase para especificar una columna de la base de datos de este tipo, o en su caso del modelo de objetos.

Actor Secundario: no tiene



Precondiciones: debería tener definido algún tipo de dato en una tabla o entidad para poder especificar un tipo de dato. Tiene que ser un tipo de dato que pueda ser mapeado por este en el modelo relacional.

Postcondiciones: el tipo de dato podrá ser booleano bit con valor 0 u 1.

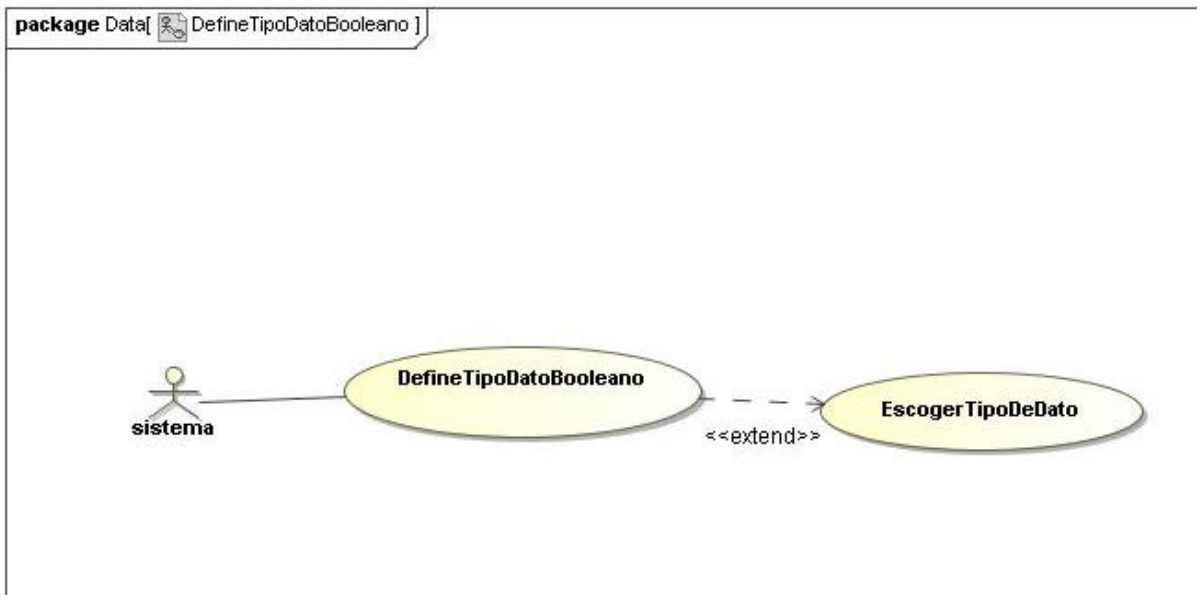
Flujo principal:

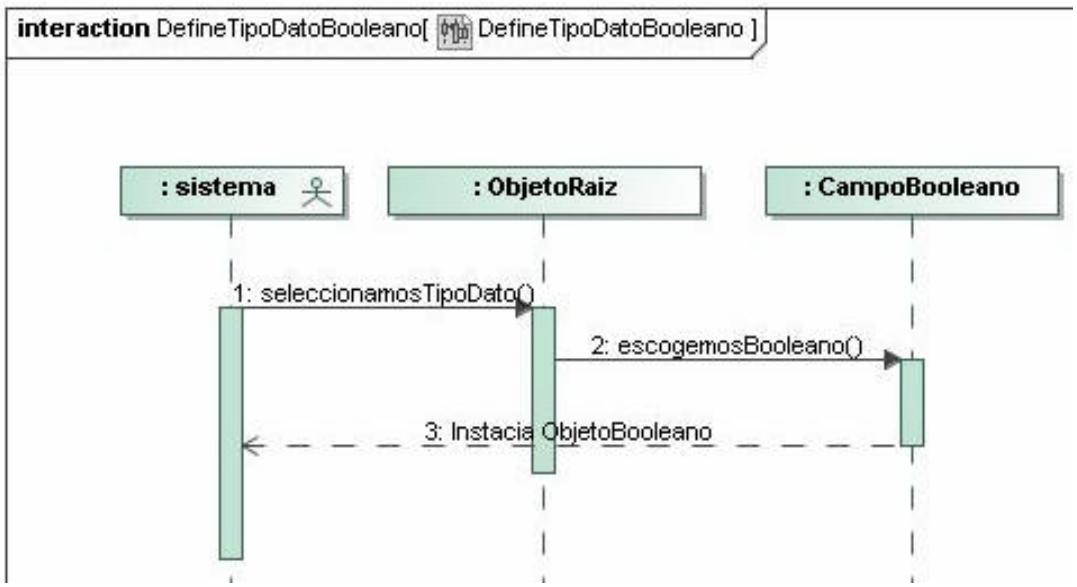
Acción de actor	Respuesta del sistema
El sistema propone definir un tipo de dato de tipo CampoBooleano Bit	El sistema identifica el tipo de dato de tipo bit y Booleano poniéndole este tipo y permitiendo que actúe como tal en el ORM extendiendo de Booleano genérico que a su vez extenderá de una clase que define los tipos de datos de todo el motor de persistencia.

Especificación Caso de Uso: CU DefineTipoDatoBooleano

En este caso de uso, se definirá una clase que nos permita especificar un tipo de datos booleano que será la clase padre de dos de los casos de uso de los que ya hemos hablado. CampoBooleano (así se llamará esta funcionalidad), será la clase padre de dos de las clases que comentamos anteriormente. Esta caso de uso extenderá de un caso de uso genérico que comentaremos más adelante que implementará una clase denominada “ObjetoRaiz” <extends CU EscogeTipoDeDato>

Normalmente no existe un mapeado para tipos de datos boléanos, sólo para cadenas y números. En este caso, mapearemos en un principio los boléanos como campos de tipo String tal como comentamos en líneas anteriores y en casos de uso dependientes de este.





Actor Principal: sistema que utiliza esta clase para especificar una columna de la base de datos de este tipo, o en su caso del modelo de objetos.

Actor Secundario: no tiene

Precondiciones: debería tener definido algún tipo de dato en una tabla o entidad para poder especificar un tipo de dato. Tiene que ser un tipo de dato que pueda ser mapeado por este en el modelo relacional.

Postcondiciones: el tipo de dato podrá ser booleano bit con valor 0 u 1, o bien, booleano de tipo String como "Y" o "N". Dependerá de la implementación y la utilización que se prefiera en función del sistema gestor de bases de datos.

Flujo principal:

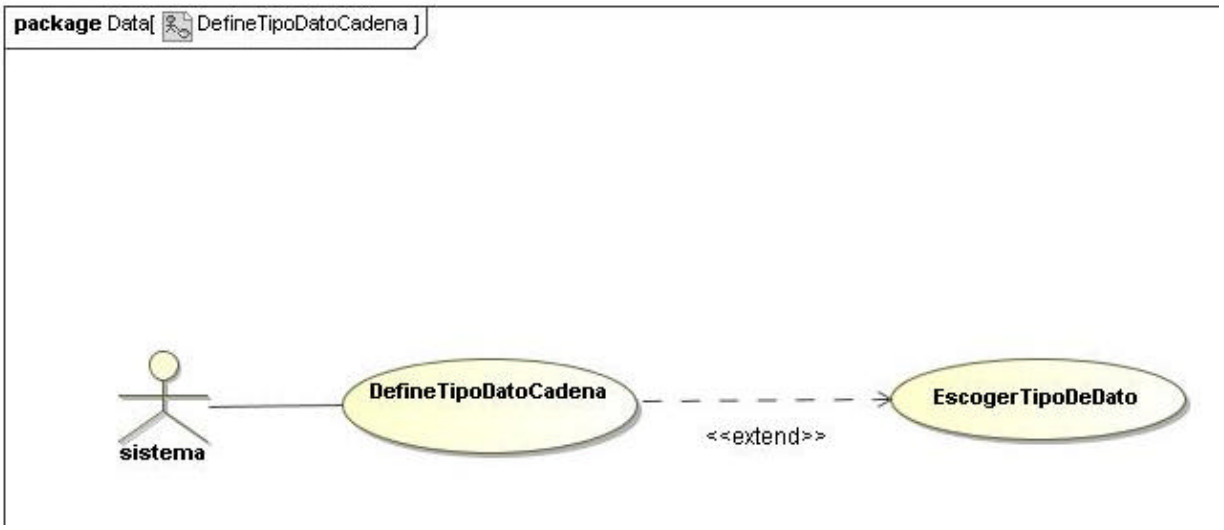
Acción de actor	**Respuesta del sistema**
El sistema propone definir un tipo de dato de tipo CampoBooleano	El sistema identifica el tipo de dato de tipo booleano que será utilizado en función de las necesidades del momento de tipo cadena o de tipo bit (char o bit).

Especificación Caso de Uso: CU DefineTipoDatoCadena

En este caso de uso, se definirá un clase que nos permita representar en el proyecto WayPersistence un tipo de dato cadena similar a String. Este tipo de dato será uno de los más utilizados para el mapeo de objeto – relacional y nos permitirá representar en nuestro

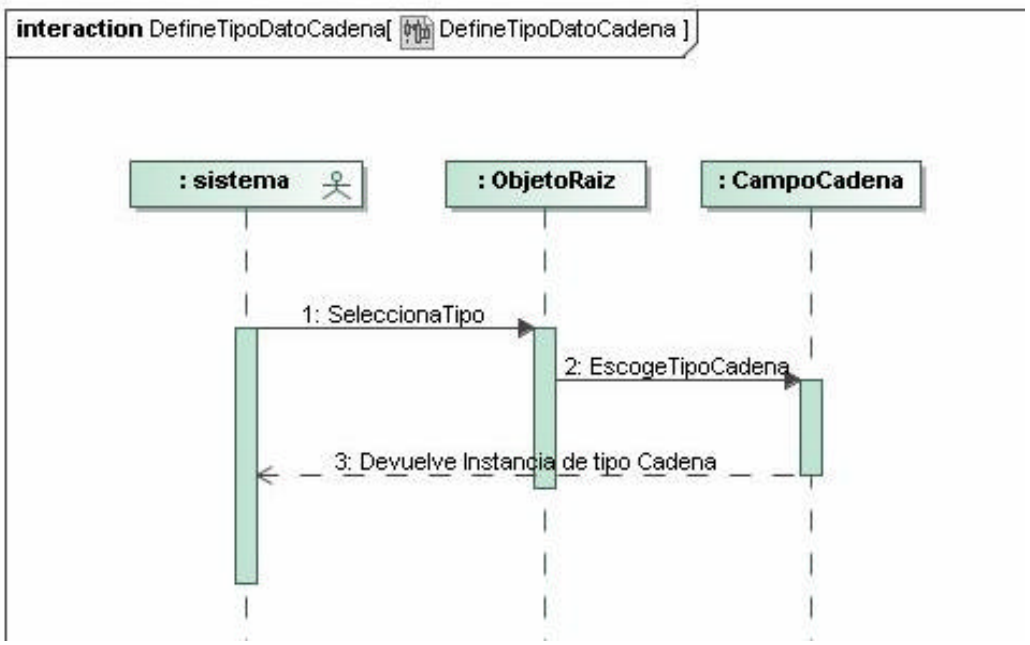
modelo de objetos los datos de tipo cadena y/o string definidos en nuestra base de datos representada en un modelo entidad – relación.

Representará un metadato de tipo cadena y dependerá de los desarrolladores su implementación y su dependencia de una clase genérica. En un principio, dependerá un caso de uso <extends CU EscogeTipoDeDato>



Actor Principal: sistema que utiliza esta clase para especificar una columna de la base de datos de este tipo, o en su caso del modelo de objetos.

Actor Secundario: no tiene



Precondiciones: debería tener definido algún tipo de dato en una tabla o entidad para poder especificar un tipo de dato. Tiene que ser un tipo de dato que pueda ser mapeado por este en el modelo relacional.

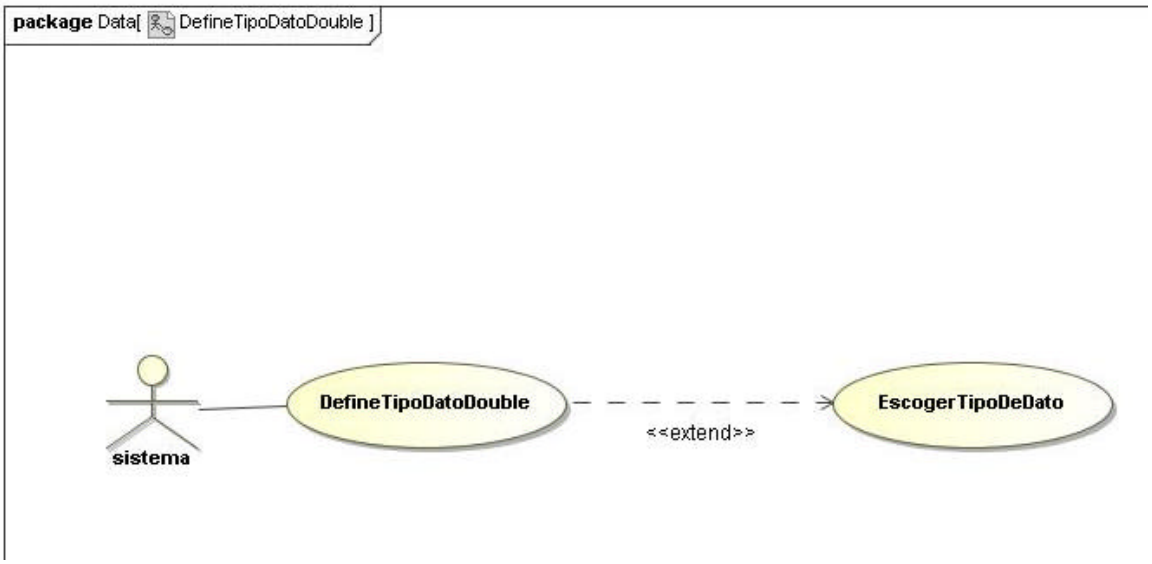
Postcondiciones: el tipo de dato podrá ser de tipo cadena o string definido en el sistema gestor de bases de datos. Para algunos varchar, char, etc.

Flujo principal:

Acción de actor	Respuesta del sistema
El sistema propone definir un tipo de dato de tipo CampoCadena	El sistema interpreta este tipo de dato cadena o string y lo transformará en un tipo de dato cadena, lo interpretará de manera adecuada independientemente del sistema gestor de bases de datos que se esté utilizando.

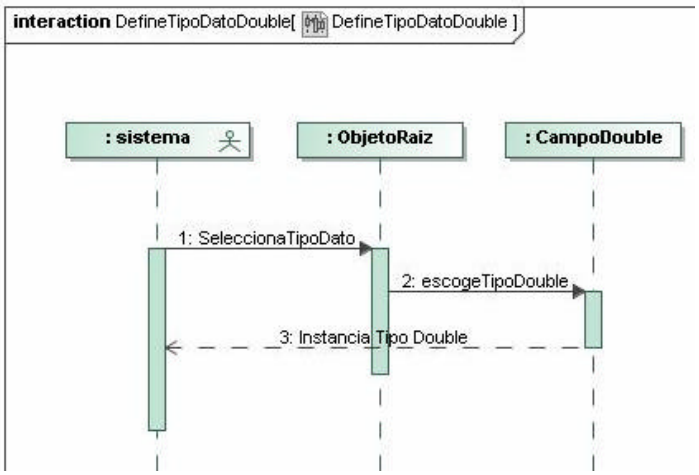
Especificación Caso de Uso: CU DefineTipoDatoDouble

En este caso de uso, se definirá una clase que nos permita representar en el proyecto WayPersistence un tipo de dato de coma flotante o Double. Este tipo de dato se trata de un valor numérico real, es decir, con decimales. En función de cómo interprete el sistema el tipo de dato que venga del modelo relacional, interpretaremos que es un tipo de dato de tipo Decimal y lo convertiremos a CampoDouble, que será el campo con el que será reconocido en WayPersistence.



Representará un metadato de tipo double o de coma flotante y dependerá de los desarrolladores su implementación y su dependencia de una clase genérica. En un principio, dependerá un caso de uso <extends CU EscogeTipoDeDato> que permitirá que implementemos algunos métodos genéricos que pueden ser necesarios en nuestra funcionalidad.

Actor Principal: sistema que utiliza esta clase para especificar una columna de la base de datos de este tipo, o en su caso del modelo de objetos.



Actor Secundario: no tiene

Precondiciones: debería tener definido algún tipo de dato en una tabla o entidad para poder especificar un tipo de dato. Tiene que ser un tipo de dato que pueda ser mapeado por este en el modelo relacional.

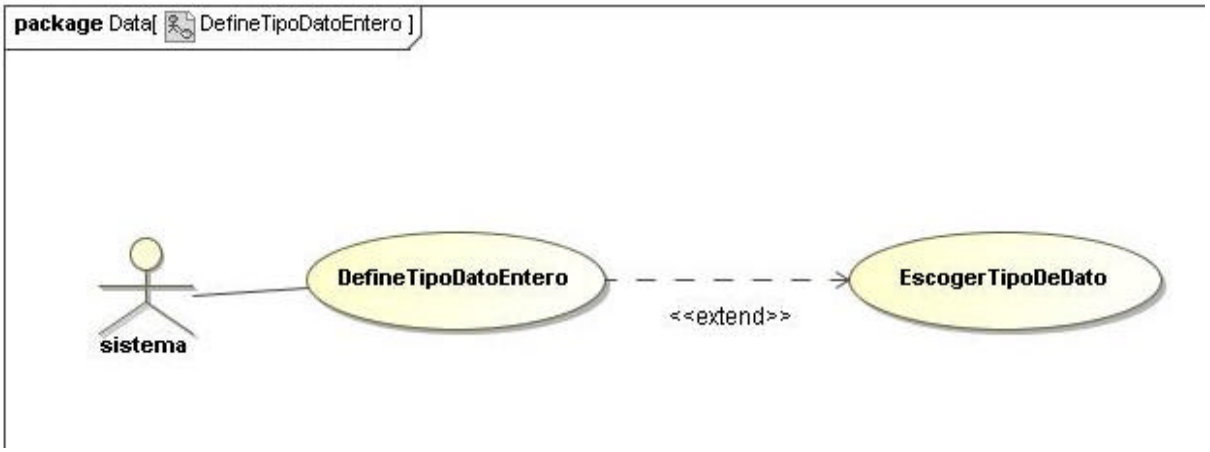
Postcondiciones: el tipo de dato podrá ser de tipo double o float y será definido en el ORM como CampoDouble.

Flujo principal:

Acción de actor	**Respuesta del sistema**
El sistema propone definir un tipo de dato de tipo CampoDouble	El sistema interpreta este tipo de dato float o double y lo transformará en un tipo de dato double o CampoDouble en el proyecto, lo interpretará de manera adecuada independientemente del sistema gestor de bases de datos que se esté utilizando.

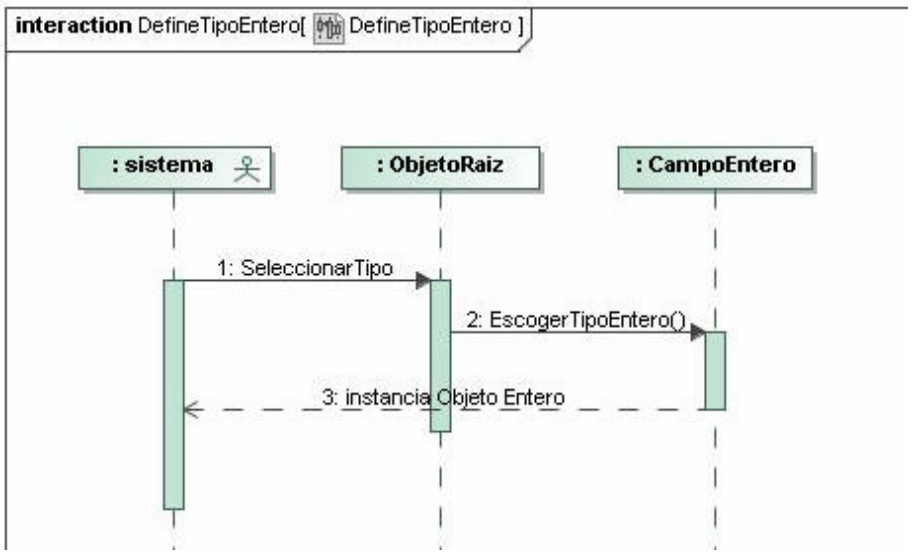
Especificación Caso de Uso: CU DefineTipoDatoEntero

En este caso de uso, se definirá una clase que nos permita representar en el proyecto WayPersistence un tipo de dato de entero o numérico. Este tipo de dato se trata de un valor numérico o entero, es decir, sin decimales. En función de cómo interprete el sistema el tipo de dato que venga del modelo relacional, interpretaremos que es un tipo de dato de tipo Entero y lo convertiremos a CampoEntero, que será el campo con el que será reconocido en WayPersistence.



Representará un metadato de tipo entero y dependerá de los desarrolladores su implementación y su dependencia de una clase genérica. En un principio, dependerá un caso de uso <extends CU EscogeTipoDeDato> que permitirá que implementemos algunos métodos genéricos que pueden ser necesarios en nuestra funcionalidad.

Actor Principal: sistema que utiliza esta clase para especificar una columna de la base de datos de este tipo, o en su caso del modelo de objetos.



Actor Secundario: no tiene

Precondiciones: debería tener definido algún tipo de dato en una tabla o entidad para poder especificar un tipo de dato. Tiene que ser un tipo de dato que pueda ser mapeado por este en el modelo relacional.

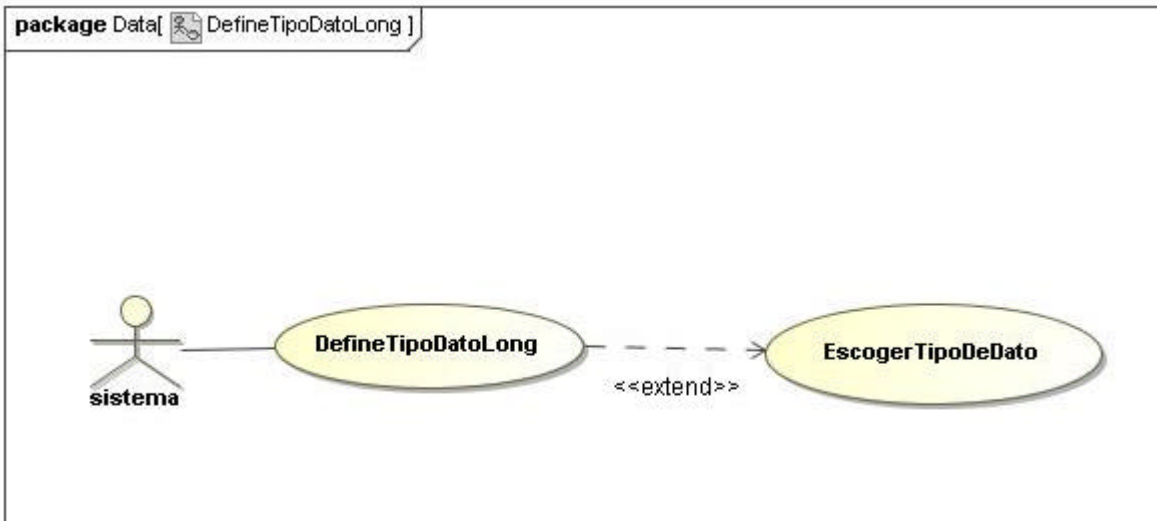
Postcondiciones: el tipo de dato podrá ser de tipo entero, int, Integer, number y será interpretado en el ORM como un tipo de dato de tipo CampoEntero.

Flujo principal:

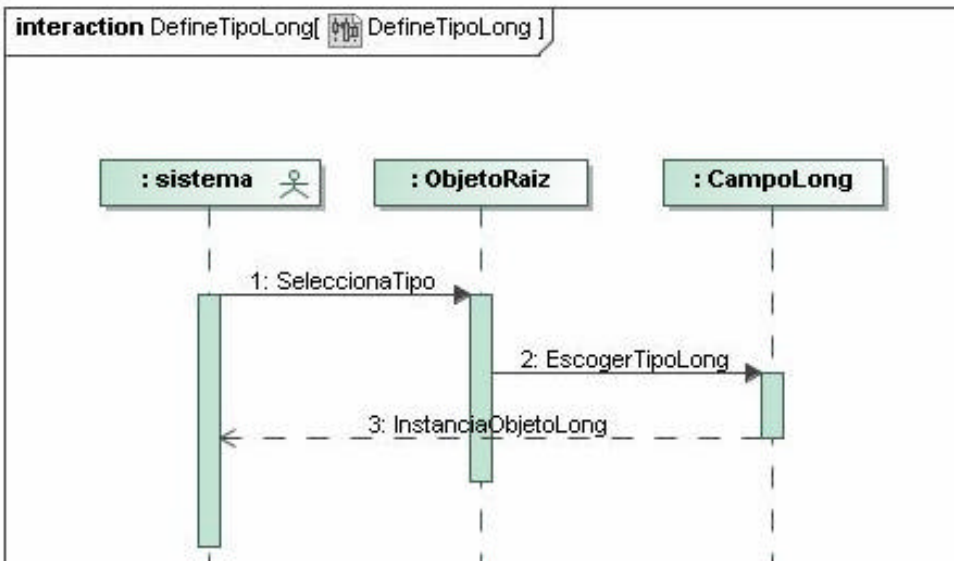
Acción de actor	Respuesta del sistema
El sistema propone definir un tipo de dato de tipo CampoEntero	El sistema interpreta este tipo de dato entero y lo transformará en un tipo de dato entero o CampoEntero en el proyecto, lo interpretará de manera adecuada independientemente del sistema gestor de bases de datos que se esté utilizando.

Especificación Caso de Uso: CU DefineTipoDatoLong

En este caso de uso, se definirá una clase que nos permita representar en el proyecto WayPersistence un tipo de dato de long o numérico. Este tipo de dato se trata de un valor



similar al entero, sin decimales, pero con mucha más capacidad para almacenar un número.



Su representación en SQL por ejemplo, es de, NUMERIC (18, 0), así pues, será representado para números de elevada longitud de dígitos... En función de cómo interprete el sistema el tipo de dato que venga del modelo relacional, interpretaremos que es un tipo de dato de tipo Entero largo y lo convertiremos a CampoLong, que será el campo con el que será reconocido en WayPersistence.

Representará un metadato de tipo entero largo y dependerá como los anteriores de un caso de uso <extends CU EscogeTipoDeDato> que permitirá que implementemos algunos métodos genéricos que pueden ser necesarios en nuestra funcionalidad.

Actor Principal: sistema que utiliza esta clase para especificar una columna de la base de datos de este tipo, o en su caso del modelo de objetos.

Actor Secundario: no tiene

Precondiciones: debería tener definido algún tipo de dato en una tabla o entidad para poder especificar un tipo de dato. Tiene que ser un tipo de dato que pueda ser mapeado por este en el modelo relacional.

Postcondiciones: el tipo de dato podrá ser de tipo entero largo o number largo y será interpretado en el ORM como un tipo de dato de tipo CampoLong.

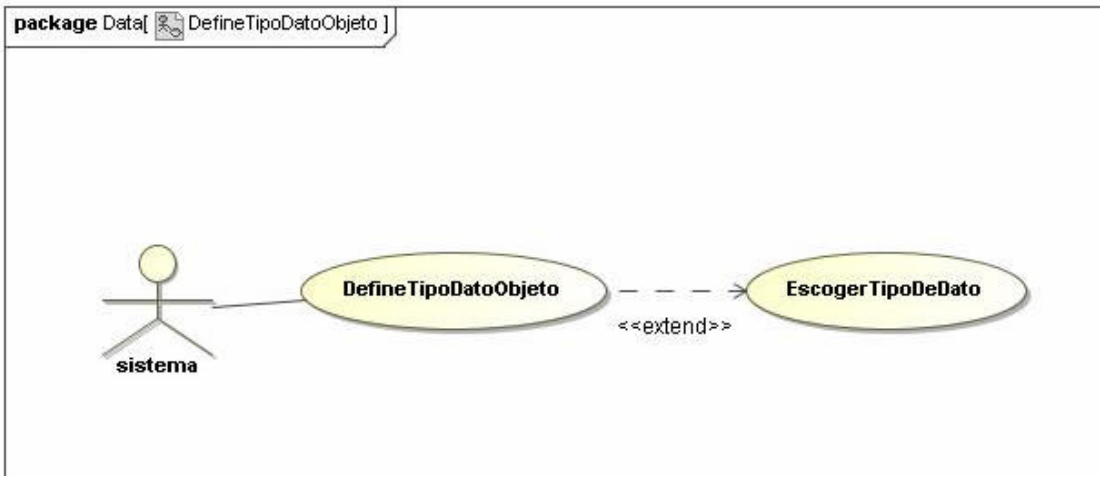
Flujo principal:

Acción de actor	**Respuesta del sistema**
El sistema propone definir un tipo de dato de tipo CampoLong	El sistema interpreta este tipo de dato entero largo y lo transformará en un tipo de dato entero o CampoLong en el proyecto, lo interpretará de manera adecuada independientemente del sistema gestor de bases de datos que se esté utilizando.

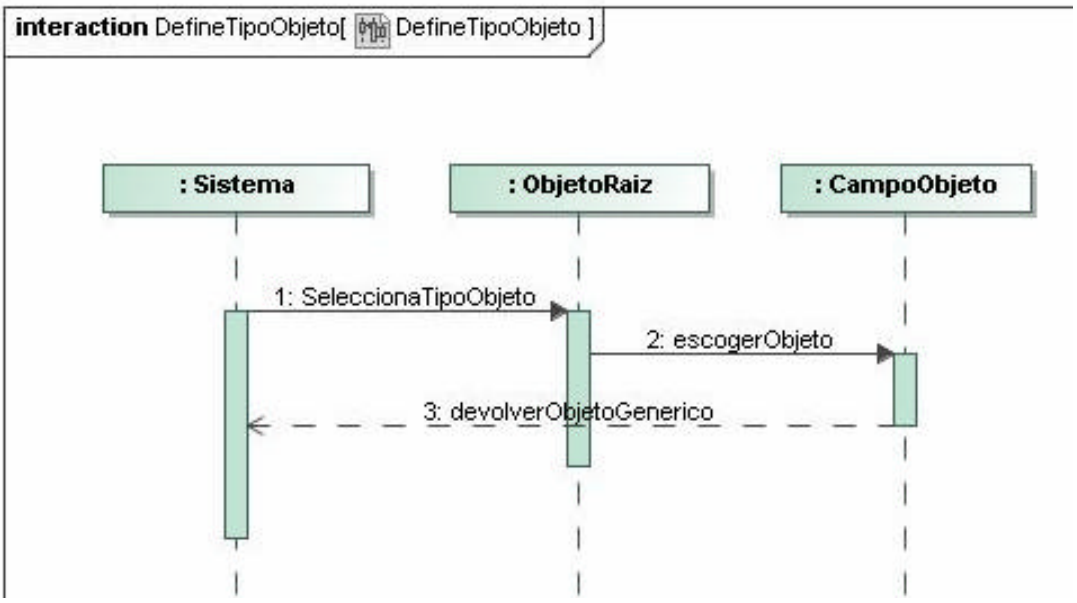
Especificación Caso de Uso: CU DefineTipoDatoObjeto

En este caso de uso, se definirá una clase que nos permita representar en el proyecto WayPersistence un tipo de dato objeto. Se utilizará para columnas que no son objetos conocidos como los referentes a datos primitivos como son Integer, String, Boolean, etc. Este tipo de dato se utilizará siempre y cuando no podamos utilizar alguno de los anteriores.

Representará un metadato de tipo entero largo y dependerá como los anteriores de un caso de uso <extends CU EscogeTipoDeDato> que permitirá que implementemos algunos métodos genéricos que pueden ser necesarios en nuestra funcionalidad.



Actor Principal: sistema que utiliza esta clase para especificar una columna de la base de datos de este tipo, o en su caso del modelo de objetos.



Actor Secundario: no tiene

Precondiciones: debería tener definido algún tipo de dato en una tabla o entidad para poder especificar un tipo de dato. Tiene que ser un tipo de dato que pueda ser mapeado por este en el modelo relacional.

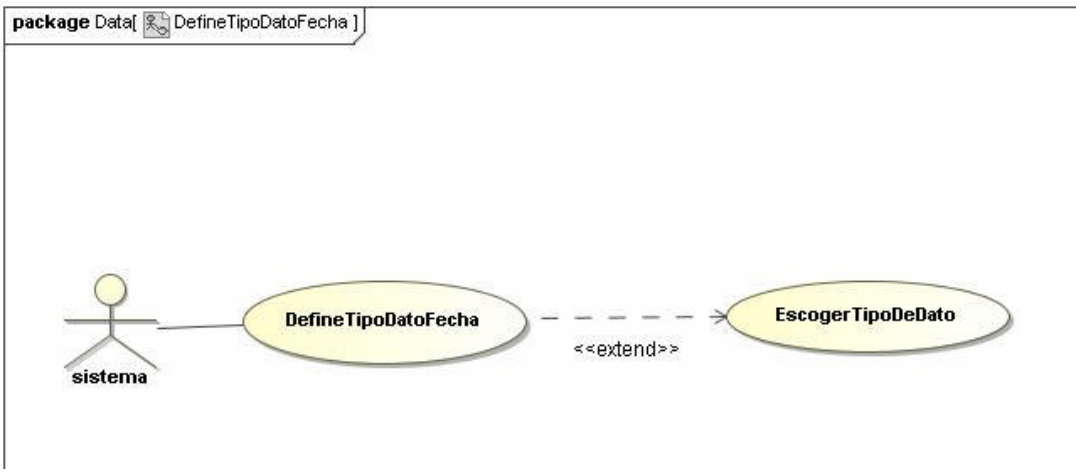
Postcondiciones: el tipo de dato podrá ser de tipo objeto y será necesario por si tenemos algún problema en los tipos de datos habituales de los sistemas gestores, en otro caso, como en tipos de datos como los primitivos, será necesario CampoObjeto.

Flujo principal:

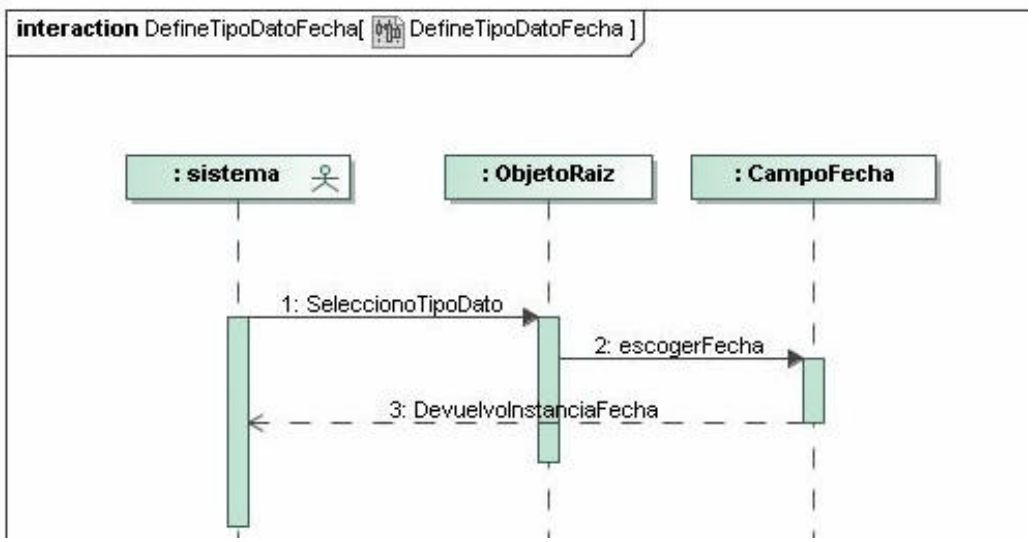
Acción de actor	**Respuesta del sistema**
El sistema propone definir un tipo de dato de tipo CampoObjeto	El sistema interpreta este tipo de dato primitivo y lo transformará en un tipo de dato CampoObjeto en el proyecto, lo interpretará de manera adecuada independientemente del sistema gestor

	de bases de datos que se esté utilizando.

Especificación Caso de Uso: CU DefineTipoDatoFecha



En este caso de uso, se definirá una clase que nos permita representar en el proyecto WayPersistence un tipo de dato fecha o date. Se utilizará para columnas que suelen estar definidas en los sistemas gestores como Date o fecha corta. Este tipo de dato se utilizará siempre y cuando sea necesario en función de lo que venga definido del modelo relacional.



Representará un metadato de tipo fecha corta o date y dependerá como los anteriores de un caso de uso <extends CU EscogeTipoDeDato> que permitirá que implementemos algunos métodos genéricos que pueden ser necesarios en nuestra funcionalidad.

Actor Principal: sistema que utiliza esta clase para especificar una columna de la base de datos de este tipo, o en su caso del modelo de objetos.

Actor Secundario: no tiene

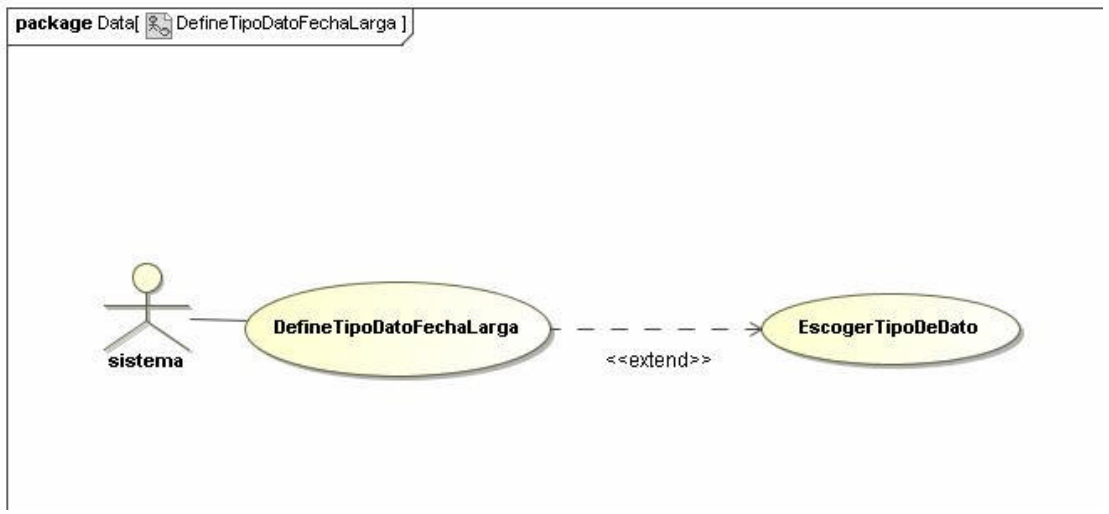
Precondiciones: debería tener definido algún tipo de dato en una tabla o entidad para poder especificar un tipo de dato. Tiene que ser un tipo de dato que pueda ser mapeado por este en el modelo relacional.

Postcondiciones: el tipo de dato podrá ser de tipo date en la mayor parte de los casos y será necesario para hacer un mapeo de un tipo de dato fecha con un formato similar a, por ejemplo, 12/12/2008.

Flujo principal:

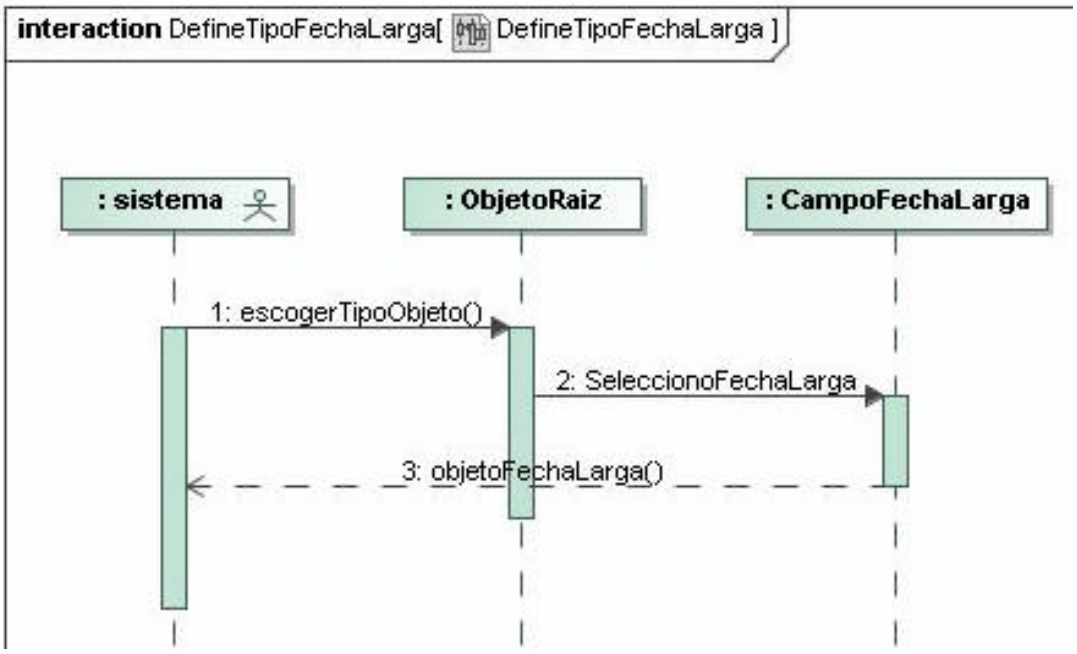
Acción de actor	Respuesta del sistema
El sistema propone definir un tipo de dato de tipo CampoFecha	El sistema interpreta este tipo de dato primitivo y lo transformará en un tipo de dato CampoFecha en el proyecto, lo interpretará de manera adecuada independientemente del sistema gestor de bases de datos que se esté utilizando.

Especificación Caso de Uso: CU DefineTipoDatoFechaLarga



En este caso de uso, se definirá una clase que nos permita representar en el proyecto WayPersistence un tipo de dato fecha larga, timestamp o datetime. Se utilizará para columnas que suelen estar definidas en los sistemas gestores como Timestamp o datetime en función del sistema gestor que se esté utilizando, también denominadas fechas largas. Este tipo de dato se utilizará siempre y cuando sea necesario en función de lo que venga definido del modelo relacional.

Representará un metadato de tipo fecha larga, timestamp o datetime. Por ejemplo en Oracle es TimeStamp y representa una fecha de la forma 12/12/2008 10:00:00. Dependerá como los anteriores de un caso de uso <extends CU EscogeTipoDeDato> que permitirá que implementemos algunos métodos genéricos que pueden ser necesarios en nuestra funcionalidad.



Actor Principal: sistema que utiliza esta clase para especificar una columna de la base de datos de este tipo, o en su caso del modelo de objetos.

Actor Secundario: no tiene

Precondiciones: debería tener definido algún tipo de dato en una tabla o entidad para poder especificar un tipo de dato. Tiene que ser un tipo de dato que pueda ser mapeado por este en el modelo relacional.

Postcondiciones: el tipo de dato podrá ser de tipo timestamp o datetime en la mayor parte de los casos y será necesario para hacer un mapeo de un tipo de dato fecha con un formato similar al comentado en líneas anteriores.

Flujo principal:

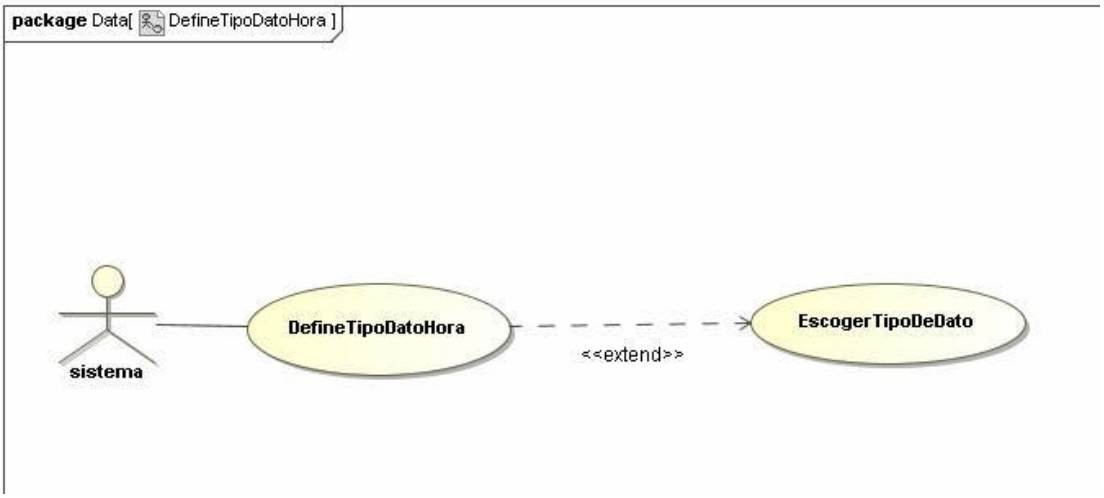
Acción de actor	**Respuesta del sistema**
El sistema propone definir un tipo de dato de tipo CampoFechaLarga	El sistema interpreta este tipo de dato primitivo y lo transformará en un tipo de dato CampoFechaLarga en el proyecto, lo interpretará de manera adecuada independientemente del sistema gestor de bases de datos que se esté utilizando.

Especificación Caso de Uso: CU DefineTipoDatoHora

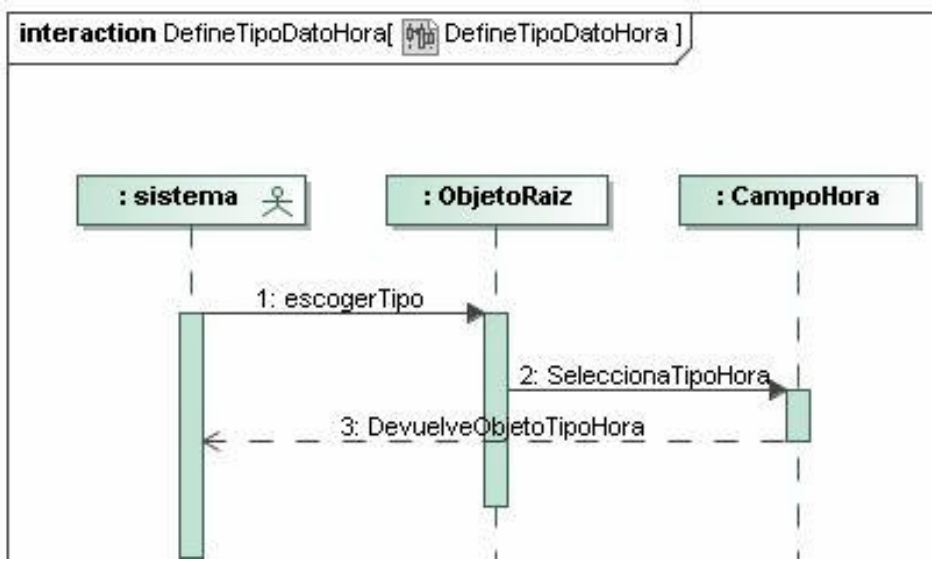
En este caso de uso, se definirá una clase que nos permita representar en el proyecto WayPersistence un tipo de dato hora o time. Se utilizará para columnas que suelen estar

definidas en los sistemas gestores como time, aunque no funciona para todos. Este tipo de dato se utilizará siempre y cuando sea necesario en función de lo que venga definido del modelo relacional.

Representará un metadato de tipo time. Dependerá como los anteriores de un caso de uso <extends CU EscogeTipoDeDato> que permitirá que implementemos algunos métodos genéricos que pueden ser necesarios en nuestra funcionalidad.



Actor Principal: sistema que utiliza esta clase para especificar una columna de la base de datos de este tipo, o en su caso del modelo de objetos.



Actor Secundario: no tiene

Precondiciones: debería tener definido algún tipo de dato en una tabla o entidad para poder especificar un tipo de dato. Tiene que ser un tipo de dato que pueda ser mapeado por este en el modelo relacional.

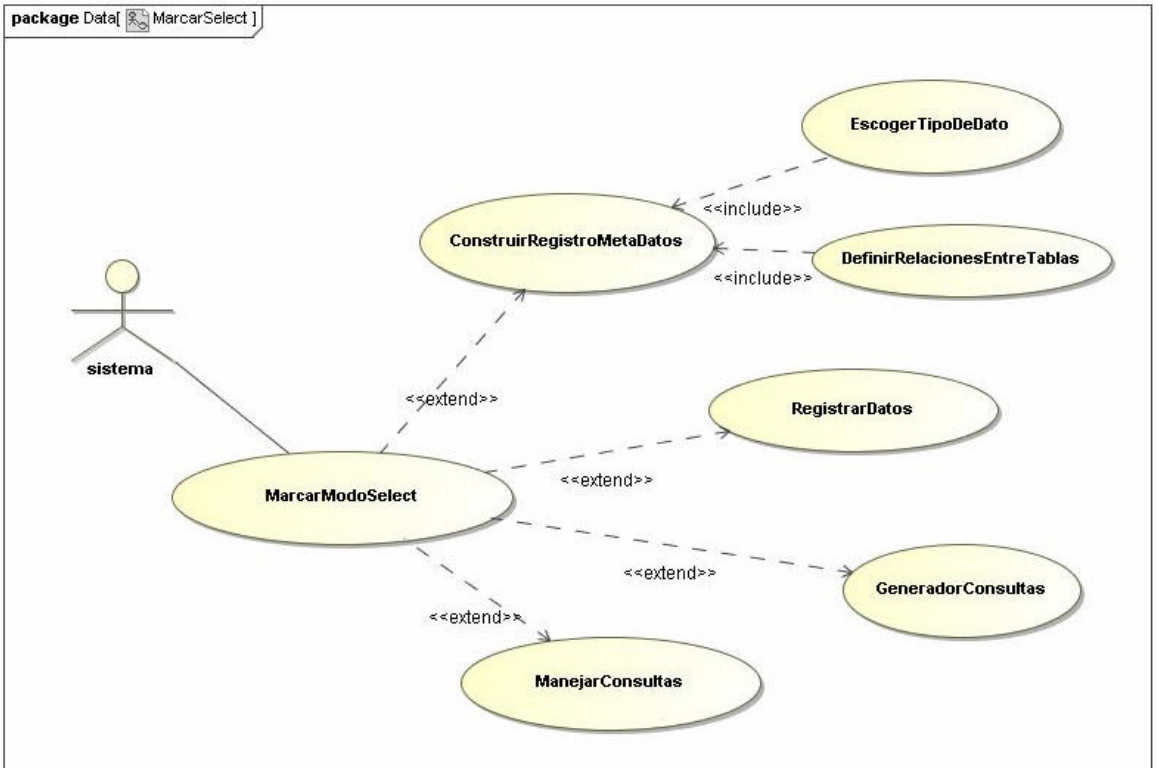
Postcondiciones: el tipo de dato podrá ser de tipo time en la mayor parte de los casos y será necesario para hacer un mapeo de un tipo de dato hora con un formato similar al comentado en líneas anteriores.

Flujo principal:

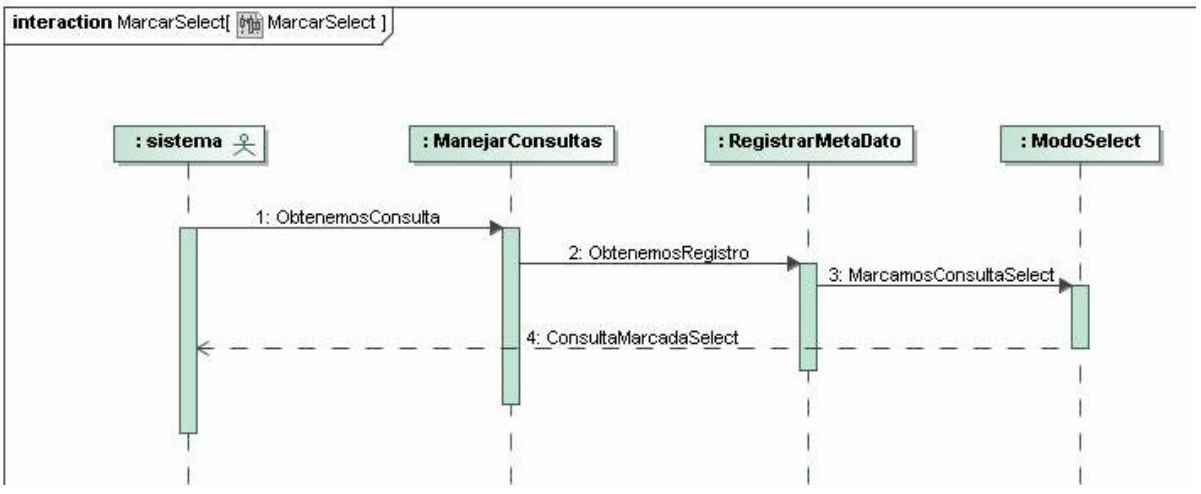
Acción de actor	Respuesta del sistema
El sistema propone definir un tipo de dato de tipo CampoHora	El sistema interpreta este tipo de dato primitivo y lo transformará en un tipo de dato CampoHora en el proyecto, lo interpretará de manera adecuada independientemente del sistema gestor de bases de datos que se esté utilizando.

Especificación Caso de Uso: CU MarcarSelect

En este caso de uso, se utilizará para filtrar los select de la consulta en función de la necesidad de cada momento. Podemos devolver una consulta de tipo select con todos sus campos (TODO), con los que son claves primarias o claves foráneas (DESCRIPTIVO), los campos excepto los marcados como NO_CONSULTADOS (NORMAL), o incluso sin añadir ninguna columnas para hacer consultas anidadas con join (NADA).



Actor Principal: sistema de consultas que marca una consulta select en función de la necesidad.



Actor Secundario: no tiene

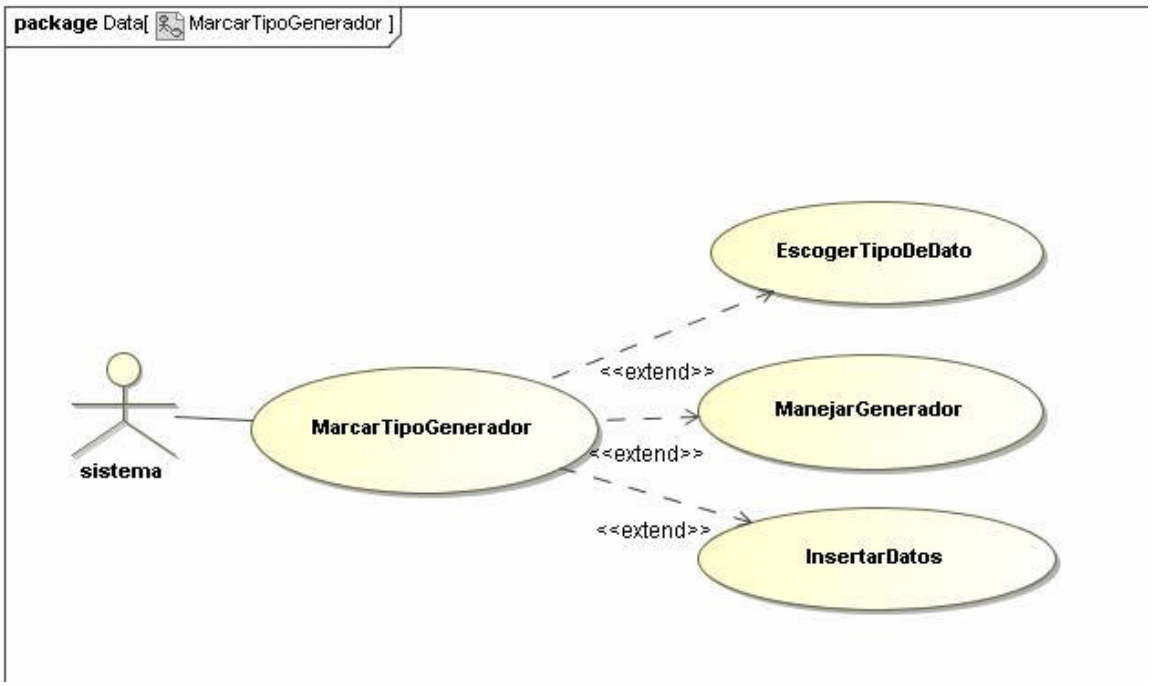
Precondiciones: debería estar haciendo alguna consulta con un tipo de dato WayQuery, para poder filtrar con este tipo de marcado tipo enumerado.

Postcondiciones: se especificará que tipo de consulta select se quiere marcar. Estará marcada al final del caso de uso con la opción elegida.

Flujo principal:

Acción de actor	Respuesta del sistema
El sistema indica un tipo de consulta a especificar de tipo enumerado	El sistema interpreta el tipo de consulta a establecer en función de lo que se haya especificado.

Especificación Caso de Uso: CU MarcarTipoGenerador



En este caso de uso, se utilizará para marcar que tipo de operación se hará sobre la base de datos. Es decir, a nivel consulta si el campo va a ser seleccionado, insertado, para tipos de datos genéricos. Actuará al igual que el caso de uso anterior como un enumerado y puede ser de 3 tipos: SELECT_MAX, SEQUENCIA, INSERCCIÓN.

Actor Principal: sistema de consultas que marca el tipo de generador para operaciones especiales

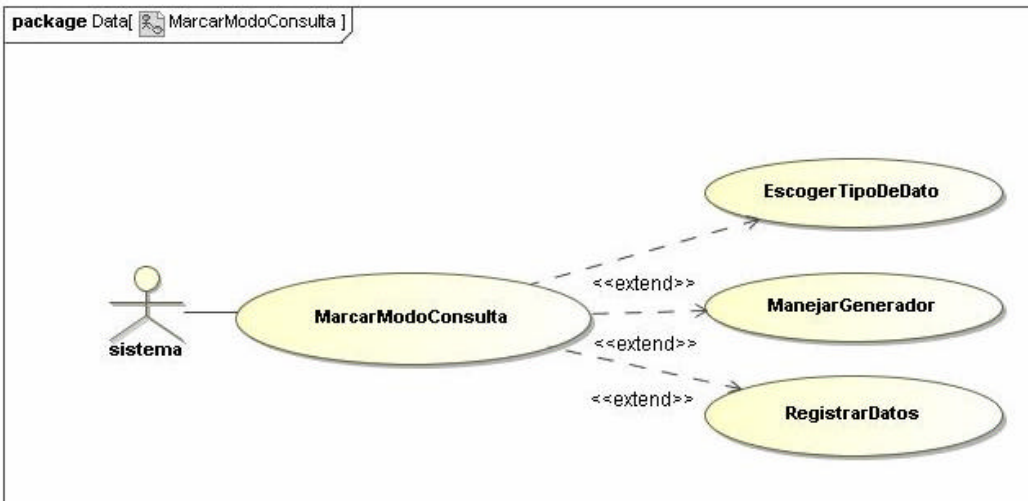
Precondiciones: debería estar haciendo alguna consulta específica.

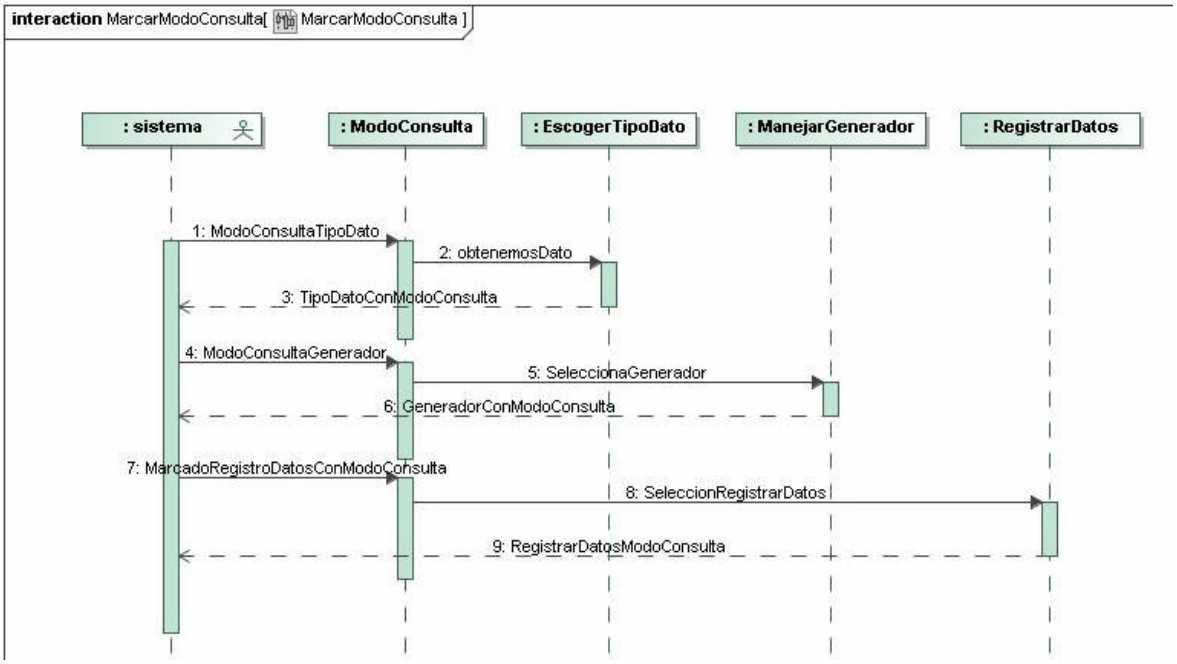
Postcondiciones: se especificará que tipo generador se quiere utilizar, se ha seleccionado.

Flujo principal:

Acción de actor	Respuesta del sistema
El sistema indicará un tipo de generador específico.	El sistema interpreta el tipo de generador que se ha escogido e interpreta el mismo para realizar su particularidad.

Especificación Caso de Uso: CU MarcarModoConsulta





En este caso de uso, se utilizará para marcar que tipo de modo consulta se ha escogido. Es decir, si se va a realizar una inserción, una lectura o consulta, una actualización, etc. Se podrá escoger dentro de un tipo enumerado las siguientes opciones: marcar la consulta como FOR_UPDATE (con bloqueo adicional sobre la base de datos), marcar la consulta como BASICO (sin bloqueo), SOLO_LECTURA (fuerza a que el registro no sea modificado), SOLO_INSERTA (sólo hará una inserción), y marcar la consulta como REFERENCIA_NO_CONSULTA.

Actor Principal: sistema de consultas que marca el modo de la consulta en función de la operación que vayamos a realizar.
Actor Secundario: no tiene

Precondiciones: debería estar haciendo alguna consulta específica.

Postcondiciones: se especificará que modo de consulta escogido y se marcará como tal.

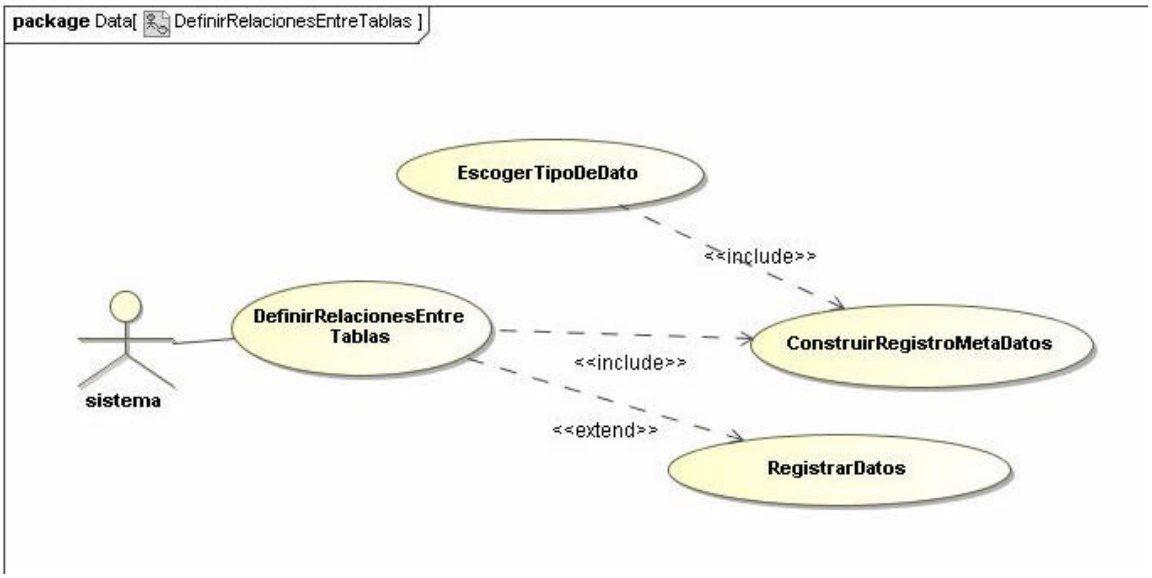
Flujo principal:

Acción de actor	Respuesta del sistema
El sistema indicará un modo de consulta a escoger	El sistema interpreta el modo de consulta escogido en función de lo que se haya seleccionado.

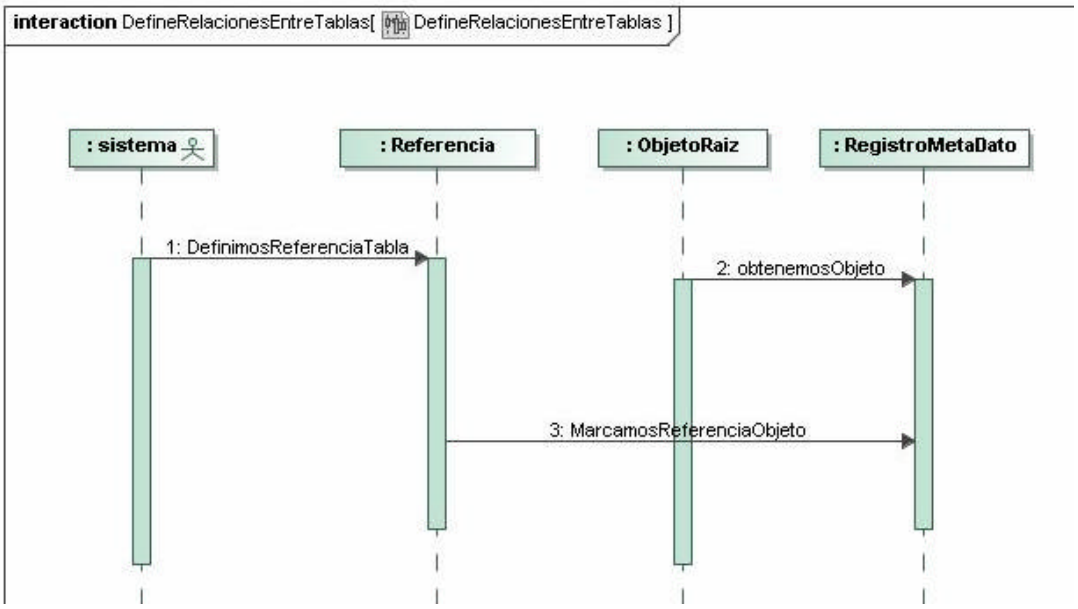
Especificación Caso de Uso: CU DefinirRelacionesEntreTablas

En este caso de uso, se representan las claves foráneas de un registro a otro. Es decir, las relaciones que puede haber entre tablas en el modelo relacional y que representaremos en el modelo orientado a objetos. Agrupará las claves foráneas que corresponden a claves primarias de otros objetos y tratará de hacer la relación entre esos objetos en el modelo de objetos.

Se realizará una simetría entre las claves foráneas definidas en el sistema gestor de bases de datos escogido para que tenga correlación y completa apariencia con las definidas en el modelo del ORM.



Tendrá relación con un caso de uso que se definirá más adelante que se denomina CU ConstruyendoMetaDato



Actor Principal: el sistema define un tipo de dato de tipo Referencia en el registro (definido en el propio ORM) y hará alusión a la clave primaria de la tabla a la que hace referencia. Es decir, Referencia actuará como una clave foránea.

Actor Secundario: no tiene

Precondiciones: debería estar definiendo un registro, y este registro tendrá un tipo de dato Referencia.

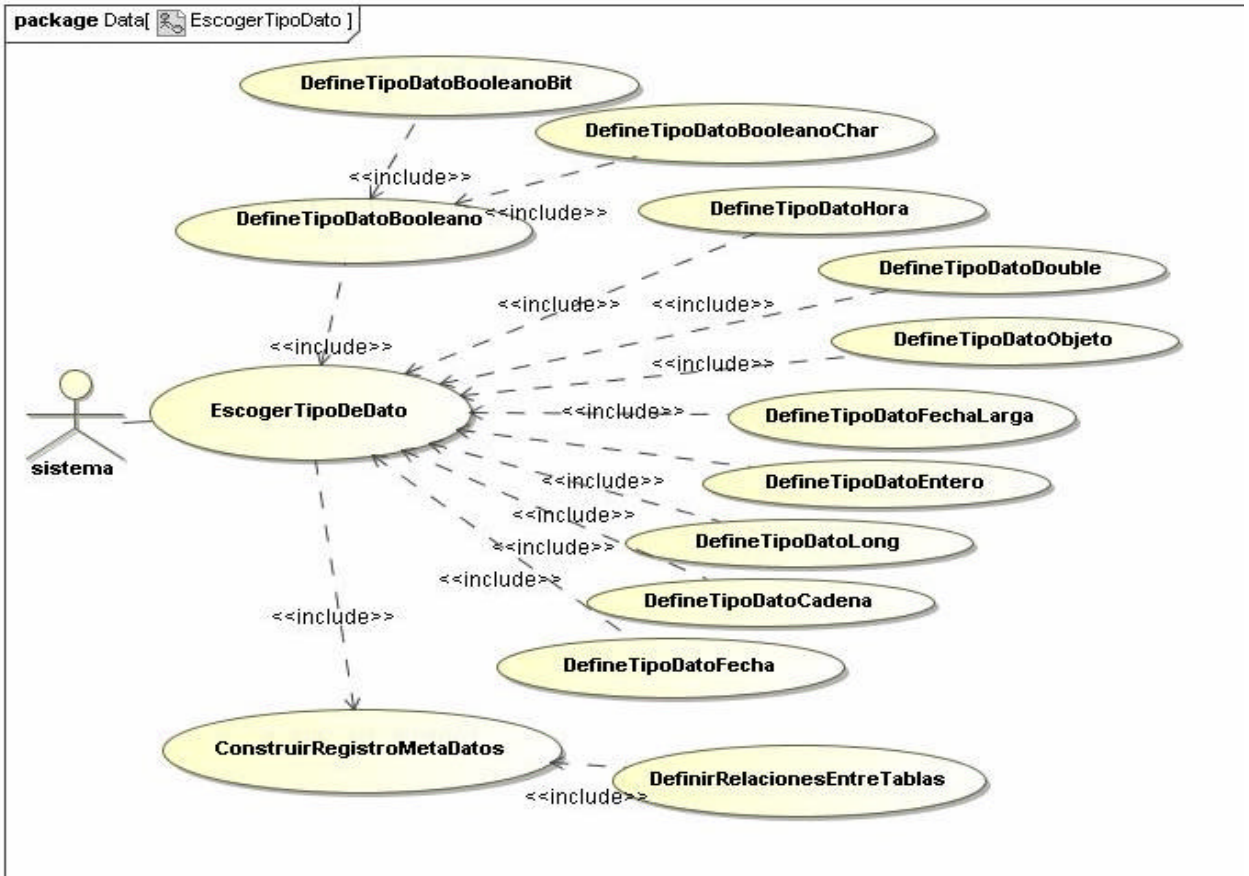
Postcondiciones: se especificará un tipo de dato de tipo Referencia que será una clave foránea a nivel de representación de objetos y no de modelo relacional como es la propia foreign key por definición

Flujo principal:

Acción de actor	Respuesta del sistema
El sistema selecciona o define una clave foránea.	El sistema interpreta que esa clave foránea es una Referencia a la clave Primaria indicada siempre con relación al modelo relacional.

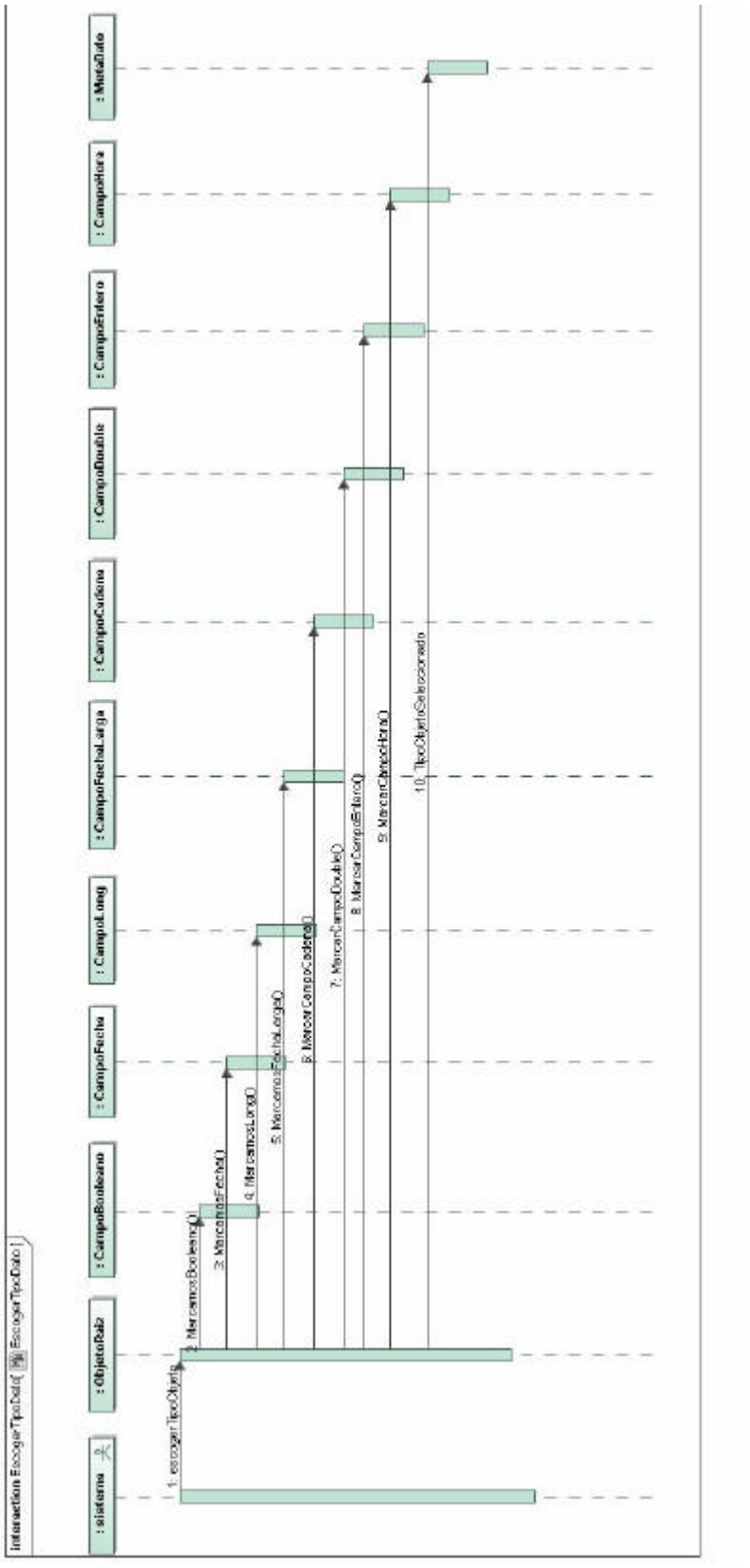
Especificación Caso de Uso: CU EscogerTipoDato

En este caso de uso, se define un objeto de tipo genérico denominado ObjetoRaiz. Se trata de hacer una funcionalidad genérica que permita tener una clase padre o genérica de la que extenderán la gran mayoría de casos de uso comentados anteriormente. Esto permitirá hacer herenciable el caso de uso y tener un control total sobre los tipos de datos a la hora de realizar el modelo orientado a objetos.



Actor Principal: el sistema define un tipo de dato genérico y lo extiende al tipo de objeto al que pertenezca dicho dato a partir de este objeto raíz u objeto genérico.

Actor Secundario:



Precondiciones: debería estar definiendo un registro, y este registro tendrá un objeto de cualquiera de los tipos que cuelgan de este mismo

Postcondiciones: se especificará un tipo de dato de cualquiera de los tipos a partir de este, siempre se pasará por este y todos los objetos que heredan de este implementarán sus métodos.

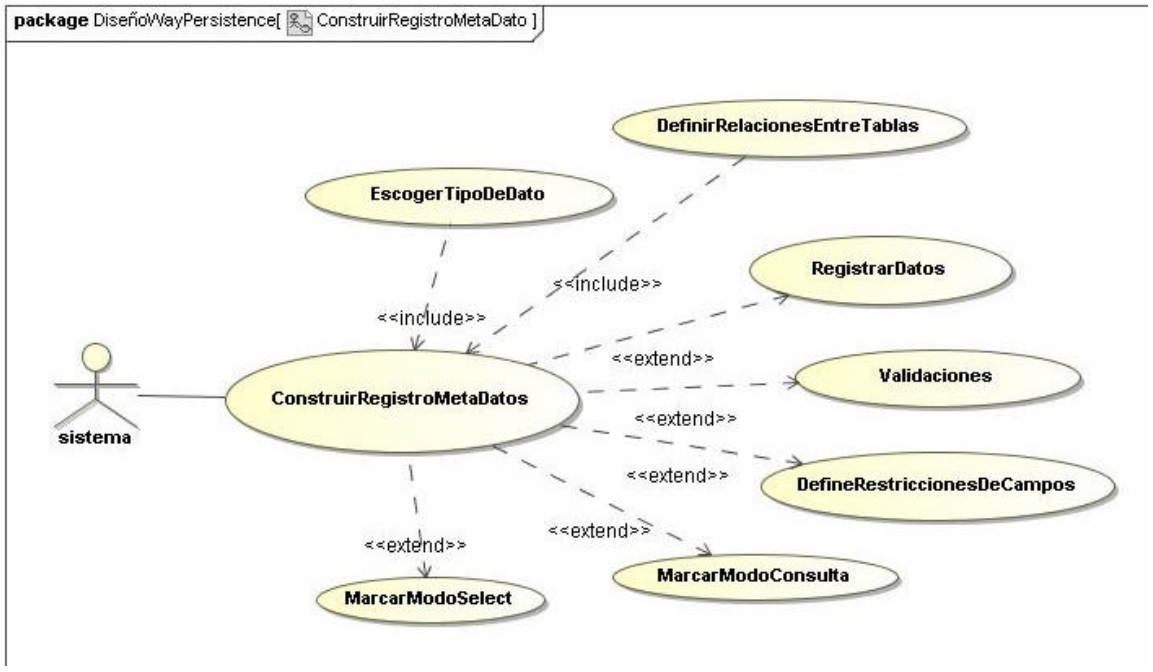
Flujo principal:

Acción de actor	Respuesta del sistema
El sistema define un tipo de objeto del tipo que sea.	El sistema interpreta el tipo de objeto en cuestión e implementa los métodos de este objeto padre.

Casos de uso relacionados: DefineTipoDatoBooleano, DefineTipoDatoCadena, DefineTipoDatoDouble, DefineTipoDatoFecha, DefineTipoDatoEntero, DefineTipoDatoHora, DefineTipoDatoFechaLarga, DefineTipoDatoHora, DefineTipoDatoLong, DefineTipoDatoObjeto. Todos y cada uno de estos casos de uso necesitan de la implementación de este caso de uso para que tengan "vida propia", pues implementan métodos definidos en este caso de uso concreto.

Especificación Caso de Uso: CU ConstruirRegistroMetaDato

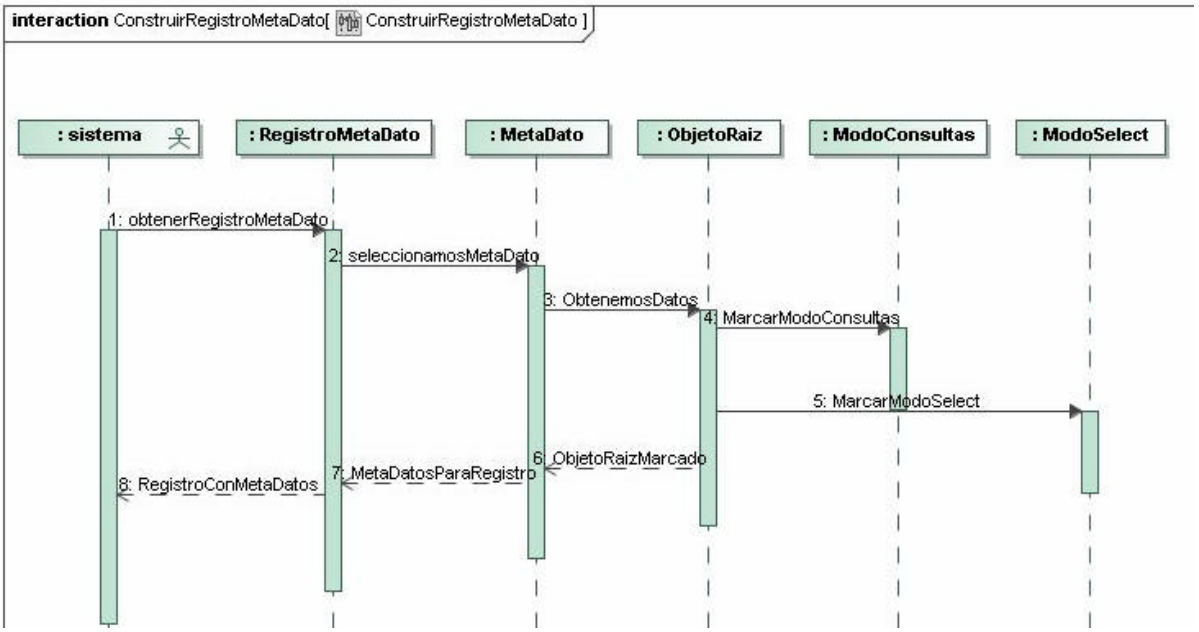
En este caso de uso, se trata de un caso de uso que define RegistroMetaDato como el nombre de la tabla. Se almacenan instancias de un MetaDato (estructura que almacenará objetos de tipo raíz o genéricos definidos en casos de uso anteriores).de uso cuya función es generar instancias que serán almacenadas en un Registro de tipo MetaDato. El principal objetivo de esta variable no es otra que describir el registro y su definición, las instancias o conexiones serán instanciadas por MetaDato.



Cuando definimos una clase de tipo Registro, como los DAO's en otros frameworks de persistencia, como por ejemplo, Hibernate, definimos en nuestro caso una línea como:

```
public static final RegistroMetaDato<Usuario> USUARIO
 = new RegistroMetaDato<Usuario> (Usuario.class, "USER");
```

En este caso, para realizar el mapeo especificamos el nombre del metadato que será utilizado en el modelo de objetos y hacemos referencia al nombre del metadato que tiene en la base de datos, en este caso "USER".



Actor Principal: el sistema define un metadato, encabezado o etiqueta de los datos que permitirá identificar a cada uno de los datos definidos dentro del caso de uso anterior <DefinirTipoDeDato>. Estas etiquetas serán útiles para agrupar los tipos de datos junto con el CU <DefinirRelacionesEntreTablas> en registros de datos que serán tratados por el motor de persistencia para el tratamiento de datos en un modelo orientado a objetos mapeado según el modelo relacional de una base de datos relacional implementada en cualquier sistema gestor actual. A su vez, los metadatos creados serán instanciados por RegistroMetaDato que definirá entre otras cosas el nombre con que reconoceremos a la tabla en nuestro modelo orientado a objetos.

Actor Secundario: el metadato actúa como actor secundario o índice que permitirá agilizar la búsqueda de los datos en función del metadato al que pertenezca y el registroMetaDato ayudará a hacer el mapeo entre modelo relacional y modelo de objeto en lo que a nombre de la tabla se refiere.

Precondiciones: deberá previamente estar definidos datos de tipo objeto y referencia (aunque no sería estrictamente necesario en este caso) para poder crear una instancia de un objeto MetaDato. Y a su vez para tratar un registroMetaDato ser deberá tener Metadatos para que puedan ser instanciados.

Postcondiciones: se encargará de implementar los metadatos o etiquetas de los campos que identifican las tablas (en el modelo relacional) mapeando con la ayuda de WayPersistence al modelo de objetos. Y a su vez, estos metadatos serán instanciados para poder crear registros con todos los metadatos que identifiquen y puedan ser mapeados del modelo de datos.

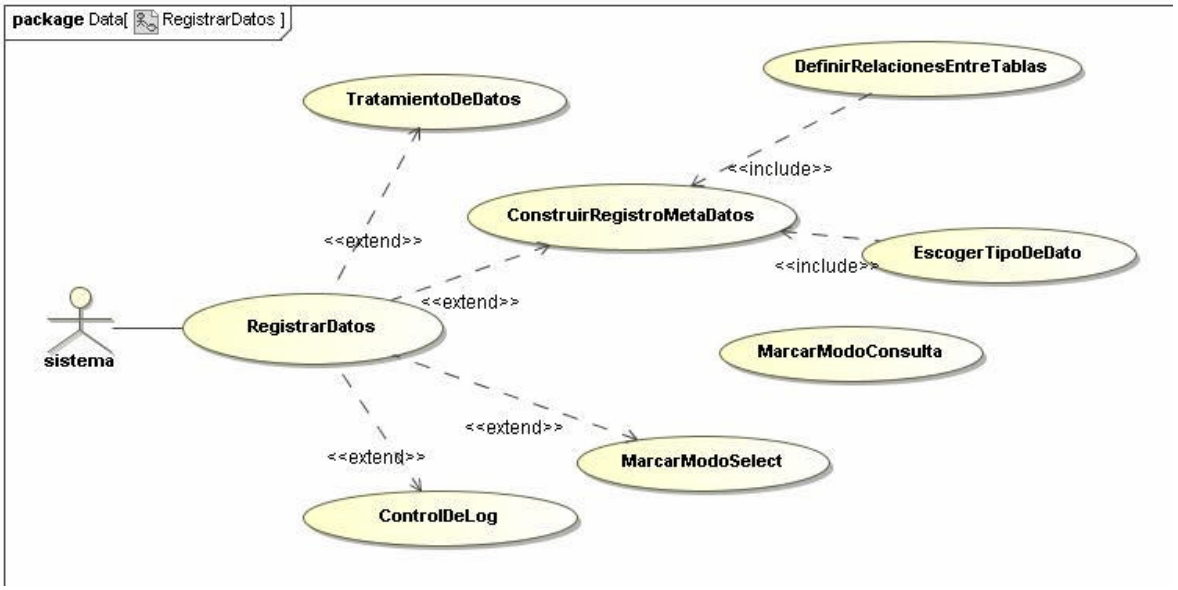
Flujo principal:

Acción de actor	Respuesta del sistema
El sistema define un registro metadato con instancias de objetos metadatos.	El sistema interpreta los metadatos instanciados que formarán un registro para ser tratado y colaborar en la representación del mapeo de objetos – relacional.

Casos de uso relacionados: <DefinirTipoDeDato>, <DefinirRelacionesEntreTablas>.

Especificación Caso de Uso: CU RegistrarDatos

En este caso de uso, se trata de un caso de uso cuya función es registrar los datos de una instancia de un registro como si de una fila de la base de datos se tratara. Esta instancia de registro es creada por la sesión <CU ManejarSession>. Solo almacenaremos una vez una misma instancia en memoria caché. Este caso de uso tratará de registrar los datos de un objeto que representa el mapeado de una tabla en el modelo relaciona. Este objeto se denomina en WayPersistence Registro <CU RegistrarDatos>.

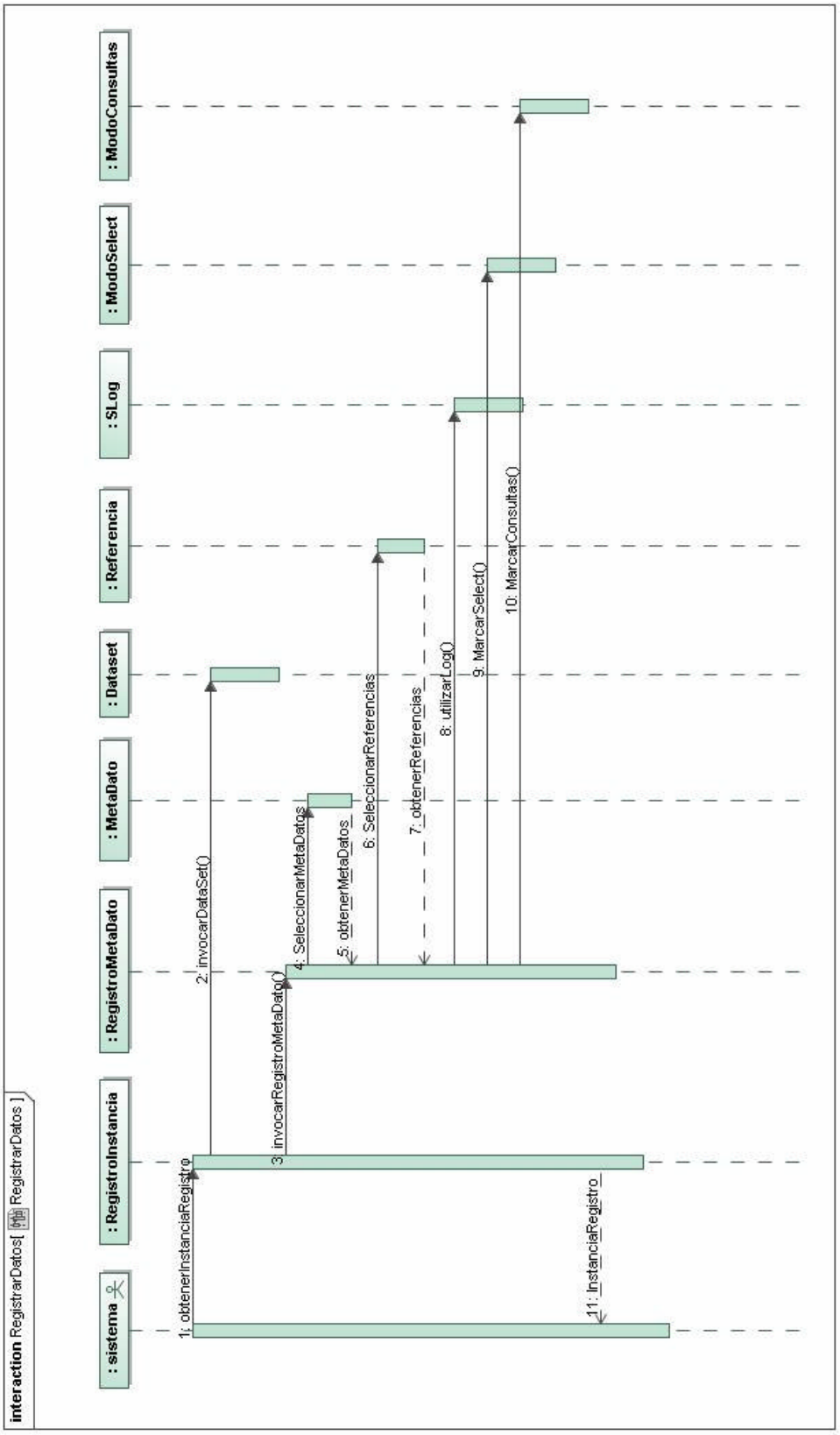


Actor Principal: el sistema define un registro de datos, equivalente a una fila de la base de datos, una tupla es insertada en memoria caché para ser tratada por la sesión.

Actor Secundario: Se utilizan los tipos de objetos definidos anteriormente en función de la necesidad de la fila que se genere.

Precondiciones: deberá previamente estar definidos datos de tipo objeto y datos para tratar en una instancia. El mapeo de los Registros permitirá obtener instancias de los registros.

Postcondiciones: se encargará de generar las filas o instancias de registros representadas en una tupla o fila de la base de datos que tendrá datos de los tipos definidos por el propio framework.



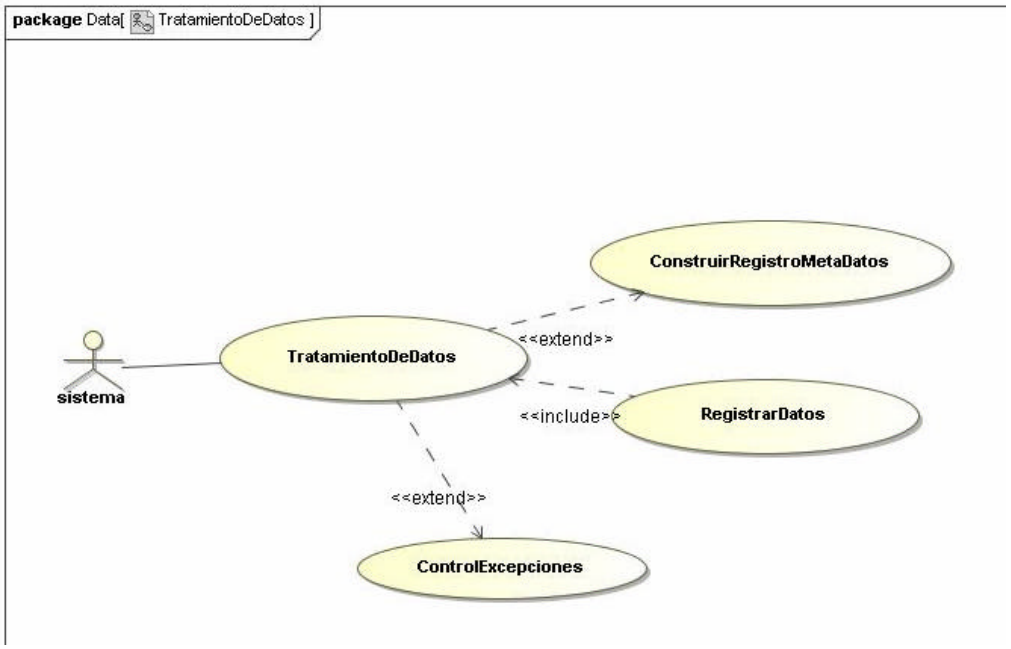
Flujo principal:

Acción de actor	Respuesta del sistema
El sistema genera una instancia de RegistroInstancia con datos de distintos tipo definidos en WayPersistence.	El sistema interpreta los tipos de datos definidos dentro de la instancia y el valor de los mismos.

Casos de uso relacionados: DefineTipoDatoBooleano, DefineTipoDatoCadena, DefineTipoDatoDouble, DefineTipoDatoFecha, DefineTipoDatoEntero, DefineTipoDatoHora, DefineTipoDatoFechaLarga, DefineTipoDatoHora, DefineTipoDatoLong, DefineTipoDatoObjeto, ConstruccionRegistroDatos.

Especificación Caso de Uso: CU TratamientoDeDatos

En este caso de uso, se manejan los datos que hayan sido utilizados en la sesión. Se crear registros, se modifican, se guarda, se hace commit, se hace flush. Toda y cada una de las operaciones básicas sobre sesiones son realizadas en el dataset, ese dataset es el encargado de dirigir el tratamiento de los datos y las órdenes que se realizan sobre ellos. El dataset contiene registros de varios tipos con sus metadatos. Estos registros son indexados pro su clave primaria.

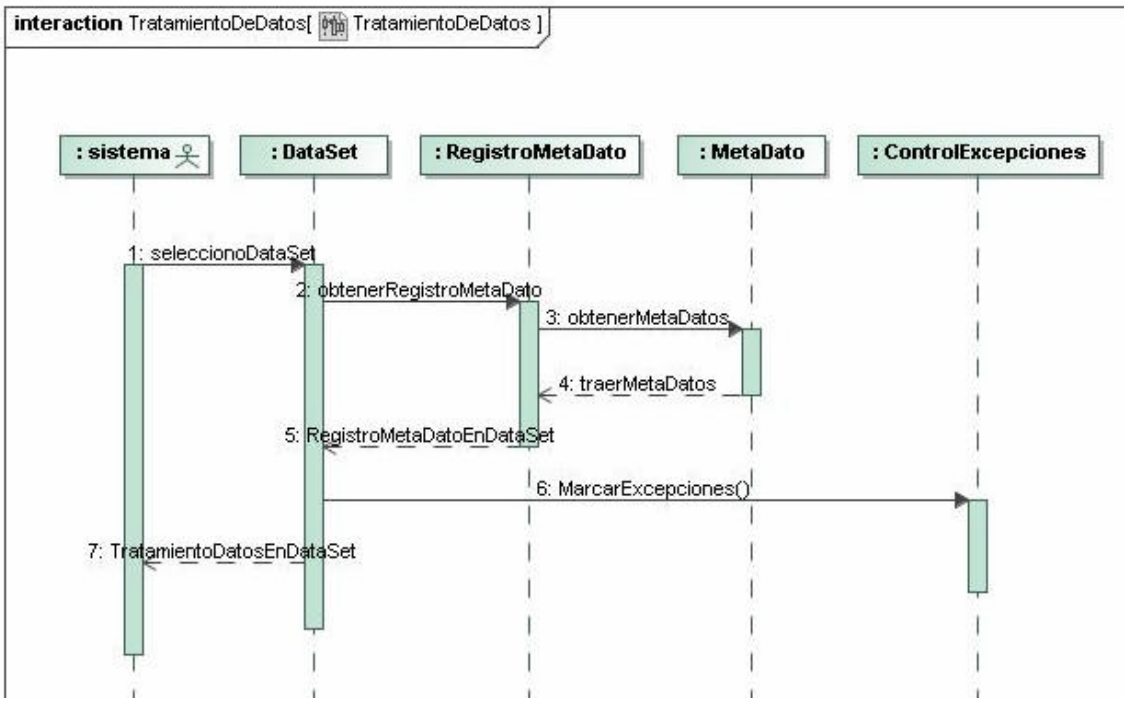


Actor Principal: el sistema a través de la sesión realiza todo tipo de operaciones en el dataset con las intancias de los datos y el registro de los mismos, junto con sus metadatos.

Actor Secundario: la sesión jdbc hace de actor ejecutante que realiza las peticiones.

Precondiciones: deberá previamente tener datos asociados, instanciados, registrados etc. para poder tratarlos.

Postcondiciones: se encargará del manejo de datos y su almacenamiento en memoria caché para cuando llegue el momento los borre de la misma y pasen a la base de datos o se deshagan y se pierdan.



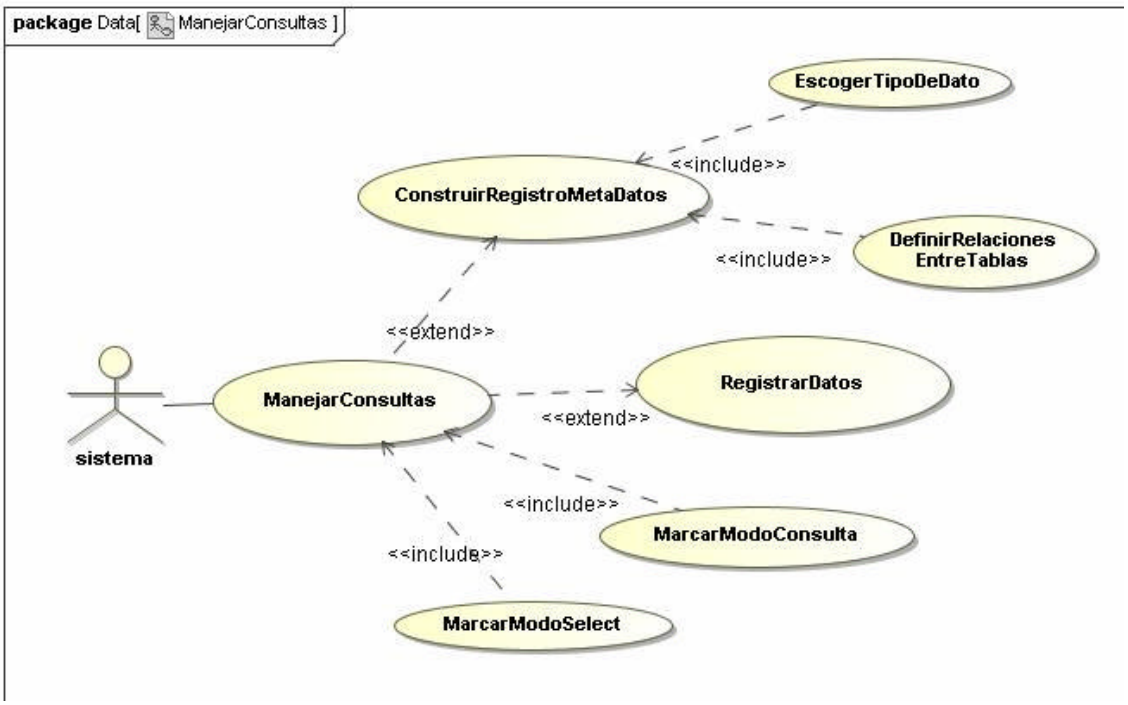
Flujo principal:

Acción de actor	Respuesta del sistema
El sistema genera una instancia de DataSet a través de la sesión para realizar operaciones básicas.	El sistema interpreta los movimientos efectuados por la sesión y realiza operaciones sobre la memoria caché en función de las órdenes realizadas a nivel de thread o hilo ejecutado desde la sesión.

Casos de uso relacionados: ConstruccionRegistroDatos, RegistrarDatos, ObjetoRaiz, ManejarSession.

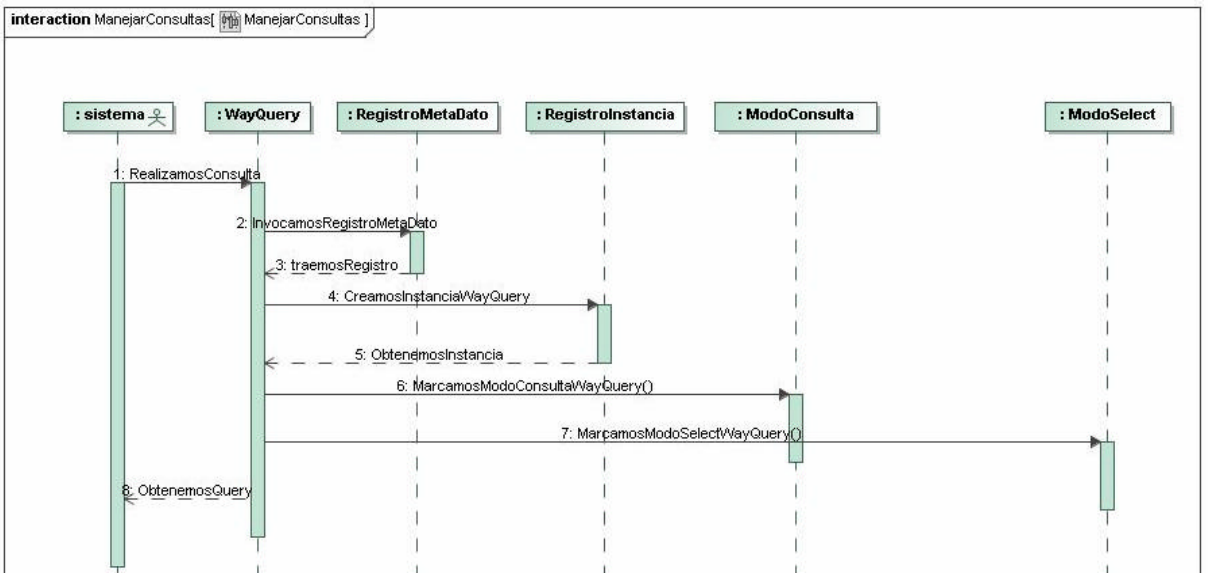
Especificación Caso de Uso: CU ManejarConsultas

En este caso de uso, se realizan todo tipo de operaciones relacionas con las consultas sobre la base de datos a nivel SQL. Permite crear instancias de WayQuery y realizar operaciones siguiendo el patrón CRUD, para operar con datos contra el modelo de objetos y modelo relacional.



Actor Principal: el sistema realiza operaciones básicas de tipo SQL sobre los datos de caché y base de datos.

Actor Secundario: no



Precondiciones: deberá previamente tener datos en base de datos y/o en caché.

Postcondiciones: se encargará del manejo de consultas sobre registros de datos almacenados en memoria o datos de bases de datos referenciadas y mapeadas contra el WayPersistence ORM.

Flujo principal:

Acción de actor	Respuesta del sistema
El sistema realiza consultas contra el framework	El sistema interpreta la petición de las consultas en función de los datos almacenados en caché y en la base de datos.

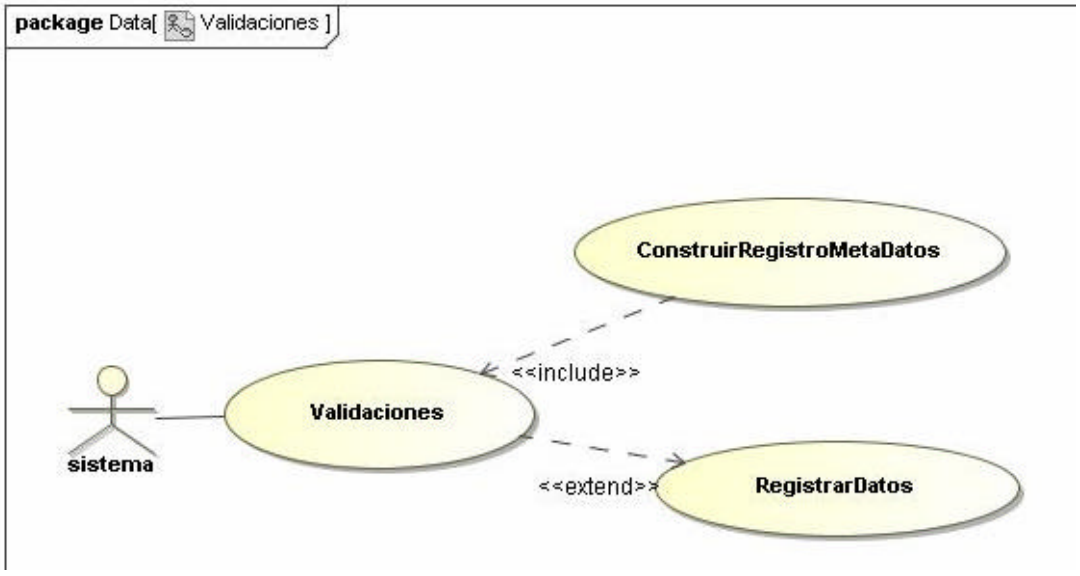
Casos de uso relacionados: ConstruccionRegistroDatos, RegistrarDatos, ManejarSession y todos los Casos de Uso de tipo CampoCadena, CampoEntero, etc. que dependen del caso de uso EscogerTipoObjeto

3. PROCESO DE VALIDACIONES DE DATOS.

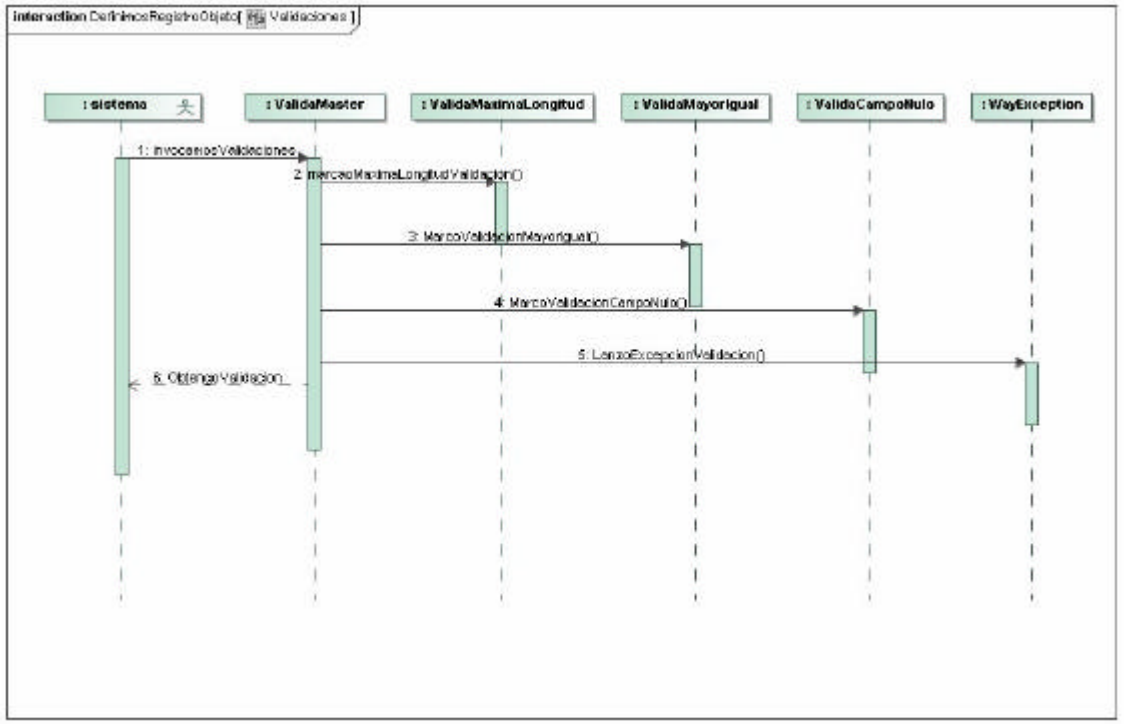
En este proceso se describen las validaciones que se van a producir a nivel del ORM tratando los datos que se utilicen definidos en el propio motor de persistencia. Se realizan unas validaciones genéricas como son las validaciones de máxima longitud de campo, las validaciones de un valor mayor o igual a otro y las validaciones de que un campo o tipo de dato no sea nulo. En versiones posteriores probablemente se enfatice la mejora de este proceso, pero de momento se creen suficientes las validaciones citadas.

Especificación Caso de Uso: CU Validaciones

En este caso de uso, se definirán varias clases que implementarán las funcionalidades de validar cualquiera de los campos que hayamos definidos en el modelo de objetos de la base de datos. Tendrán una clase padre o maestra que actuará como genérica y otras que colgarán de la misma que extenderán de ella e implementarán sus métodos. La idea de validación se basa en validar que los campos no sean nulos, que sean mayor o igual que otro valor y que se valide la longitud máxima.



Este caso de uso podrá sufrir variaciones de definición funcional en futuras versiones del ORM puesto que no se han definido todas las validaciones definitivamente.



Para asegurar que los campos no sean nulos, se implementa una funcionalidad que obtendrá una instancia del registro <CU RegistrarDatos> y un metadato, <CU Tratamiento de Datos>. Se controlará si falla alguna de los fallos de validación a través de excepciones, implementadas en <CU ControlExcepciones>.

Actor Principal: sistema que utiliza esta caso de uso para comprobar si ha validado según el tipo de validación que haya escogido

Actor Secundario: el ORM lanzará alguna excepción en caso de que se produzca un fallo en la validación.

Precondiciones: debería tener definido algún tipo de dato en una tabla o entidad para poder validar y además, deberá haber alguna instancia de registro en sesión.

Postcondiciones: el tipo de dato saldrá validado después de pasar por la funcionalidad de validación en función del tipo de validación que se haya ejecutado.

Flujo principal:

Acción de actor	**Respuesta del sistema**
El sistema lanza una validación de un campo.	El sistema identifica a que tipo de validación se refiere a partir de la clase genérica.

Flujo secundario:

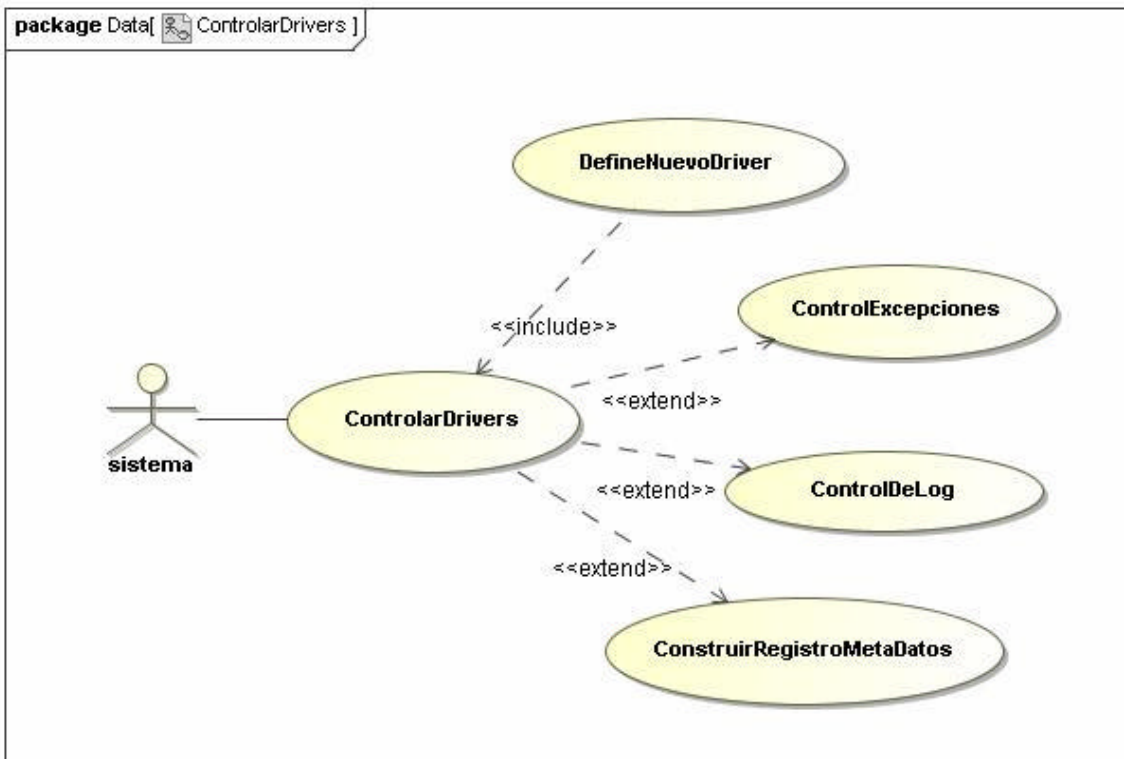
Acción de actor	**Respuesta del sistema**
El sistema valida que no sea nulo	El sistema interpreta que el valor es un nulo y devuelve la respuesta al sistema después de haber tratado con un metadato y una instancia de un

	registro.
El sistema valida que la longitud no sea la máxima	El sistema interpreta que el valor que le pasa no exceda del tamaño máximo predefinido en el tipo de dato en cuestión. Para ello también trata un metadato y una instancia de un registro sobrescribiendo el método genérico.
El sistema valida que un tipo de dato sea mayor o igual que un valor dado	El sistema interpreta si el tipo de dato que se le pasa es mayor o igual que un valor dado, que normalmente puede ser 0. sobrescribe el método genérico de validación

4. PROCESO DE DRIVERS DE CONEXIÓN

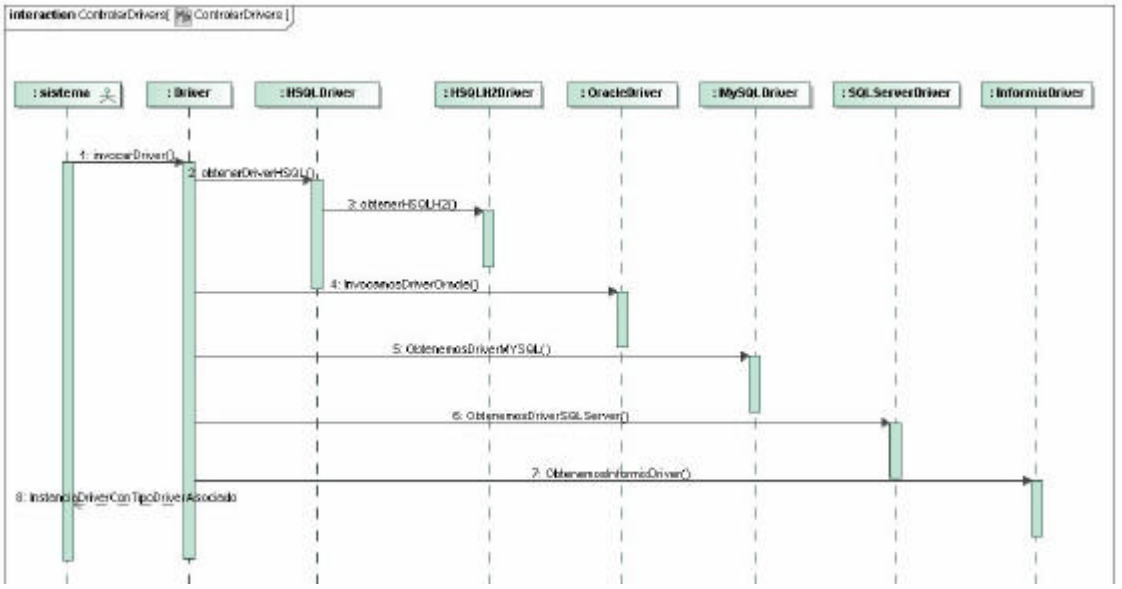
Se realiza el proceso de conexión mediante drivers con la definición de varios de ellos y su adaptabilidad a la capa de persistencia propia denominada como dijimos, WayPersistence. El acometido de este proceso consiste en realizar casos de uso que permitan definir unas pautas a seguir y unas funcionalidades que nos permitan implementar las conexiones de drivers de los distintos sistemas gestores de bases de datos. Por supuesto, para controlar cada uno de los drivers, debemos tener una clase genérica de la que extiendan los anteriores, donde se realizarán e implementarán las operaciones más habituales.

Especificación Caso de Uso: CU ControlarDrivers



En este caso de uso, se realizan todas las operaciones relacionadas con los drivers. Este caso de uso tendrá relacionados varios casos de uso, concretamente los que hayamos definido para cada sistemas gestor. Funciona como driver genérico para las bases de datos y contiene dependencias a las mismas. Existirá una instancia de este driver para cada conexión que se produzca sobre el sistema de bases de datos que estemos utilizando.

Actor Principal: el sistema realiza operaciones de control de drivers y operaciones básicas sobre una misma conexión.



Actor Secundario: el sistema busca el driver correspondiente en función de la base de datos escogida.

Precondiciones: deberá previamente tener inyectado algún driver con su jar correspondiente para proceder a realizar operaciones sobre la base de datos que corresponda.

Postcondiciones: se realizarán operaciones sobre la base de datos que corresponda con la dependencia del driver del sistema gestor de bases de datos que se haya escogido.

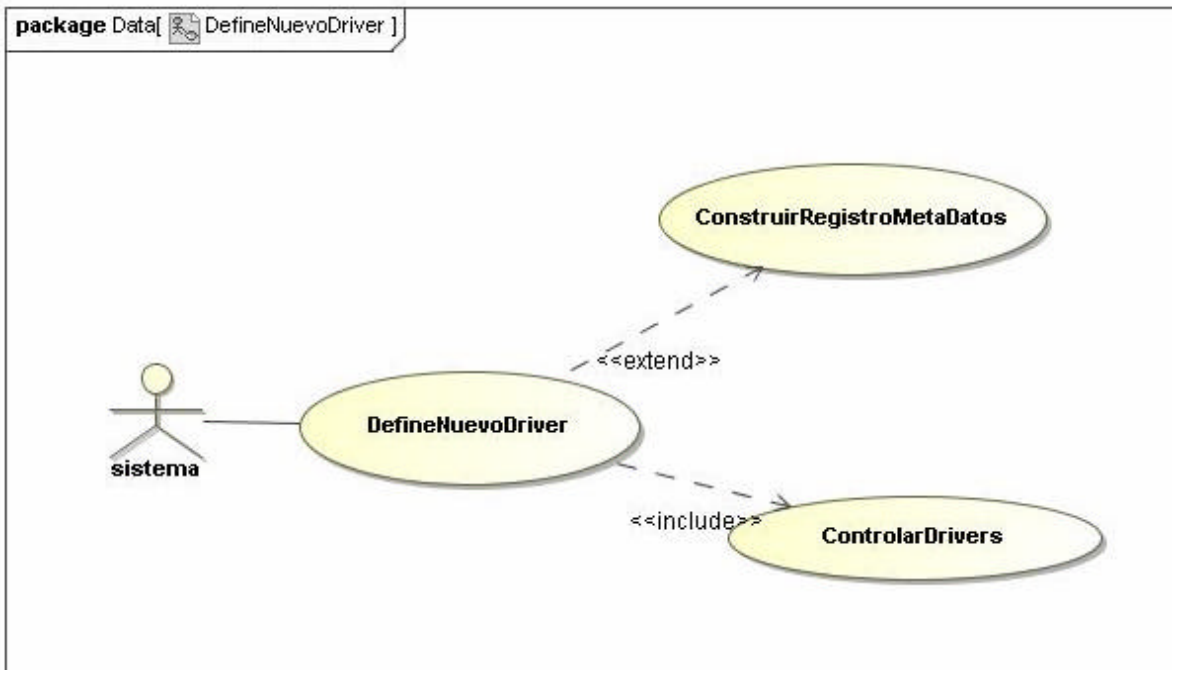
Flujo principal:

Acción de actor	**Respuesta del sistema**
El sistema realiza una conexión a una base de datos a través de un driver.	El sistema comprueba que se tiene inyectado el driver en el proyecto y que se hace referencia al caso de uso perteneciente al driver insertado. En caso contrario se producirá un error de conexión.

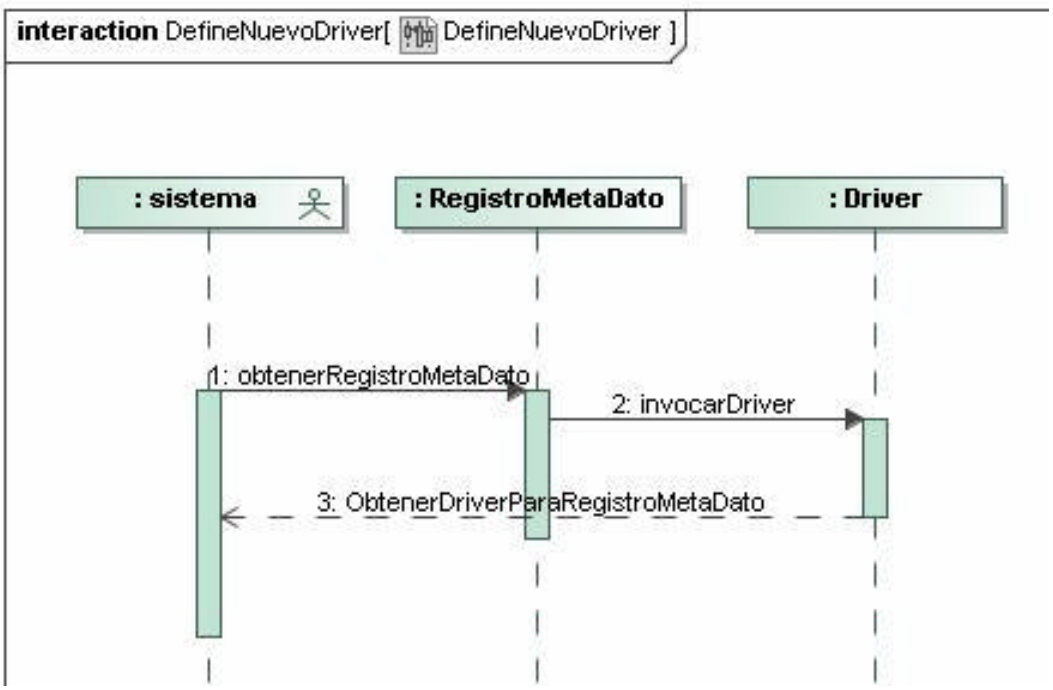
Casos de uso relacionados: CU DefineNuevoDriver

Especificación Caso de Uso: CU DefineDriverNuevo

En este caso de uso, se realizan todas las operaciones relacionadas con las particularidades de cada uno de los drivers a implementar. Se trata de implementar cada uno de los drivers que queremos utilizar en nuestro ORM ahora o e un futuro. Se trata de dejar definidas todas las implementaciones con las distintas pautas adaptadas a los sistemas gestores de bases de datos que correspondan.



Actor Principal: el sistema realiza operaciones para definir un nuevo driver en función de las características del sistema gestor de bases de datos al que pertenezcan



Actor Secundario: el sistema busca el driver correspondiente en función de la base de datos escogida.

Precondiciones: deberá previamente tener inyectado algún driver con su jar correspondiente al sistema gestor o base de datos que hayamos escogido.

Postcondiciones: se realizarán operaciones sobre la base de datos que corresponda con la dependencia del driver del sistema gestor de bases de datos que se haya escogido.

Flujo principal:

Acción de actor	Respuesta del sistema
El sistema realiza una conexión a una base de datos a través de un driver.	El sistema comprueba que se tiene inyectado el driver en el proyecto y que se hace referencia al caso de uso perteneciente al driver insertado. En caso contrario se producirá un error de conexión.

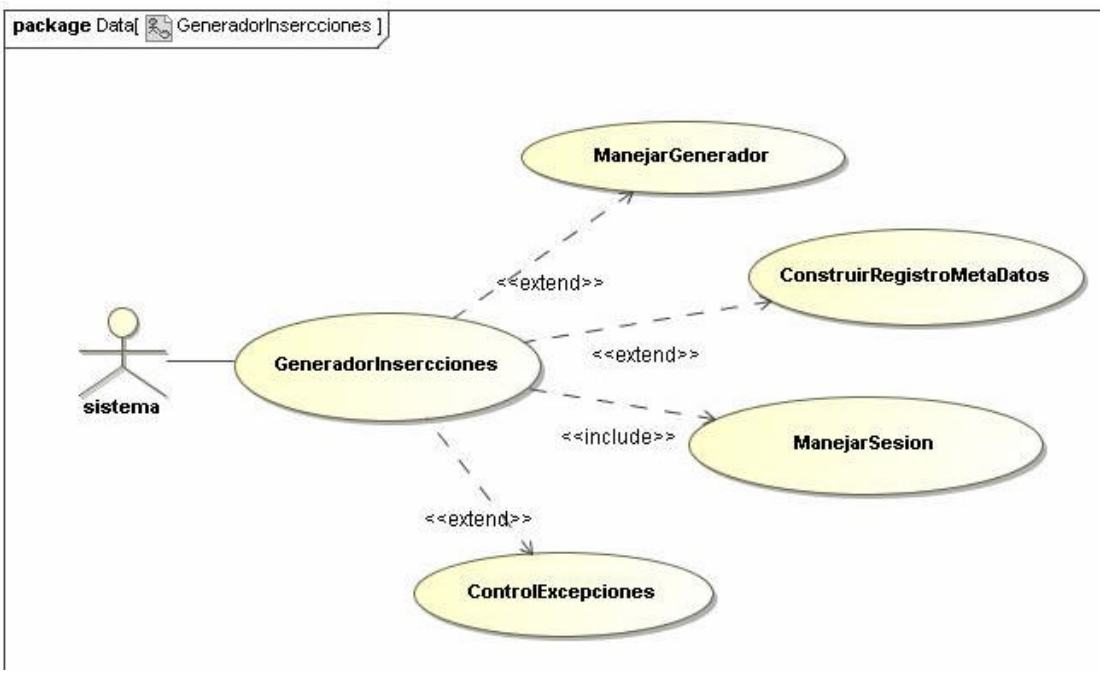
5. PROCESO DE MOTOR DEL FRAMEWORK.

Descripción
Esta parte del proceso ORM junto con la parte de database son el corazón del framework de persistencia y permiten realizar las interacciones a la base de datos a partir del Dataset y la conexión con el jdbc. Destaca la sesión interna y manejo de la misma, así como el driver (comentado anteriormente) y la carga de datos que permite realizar inserciones al dataset para posteriormente pasarlo a la base de datos.

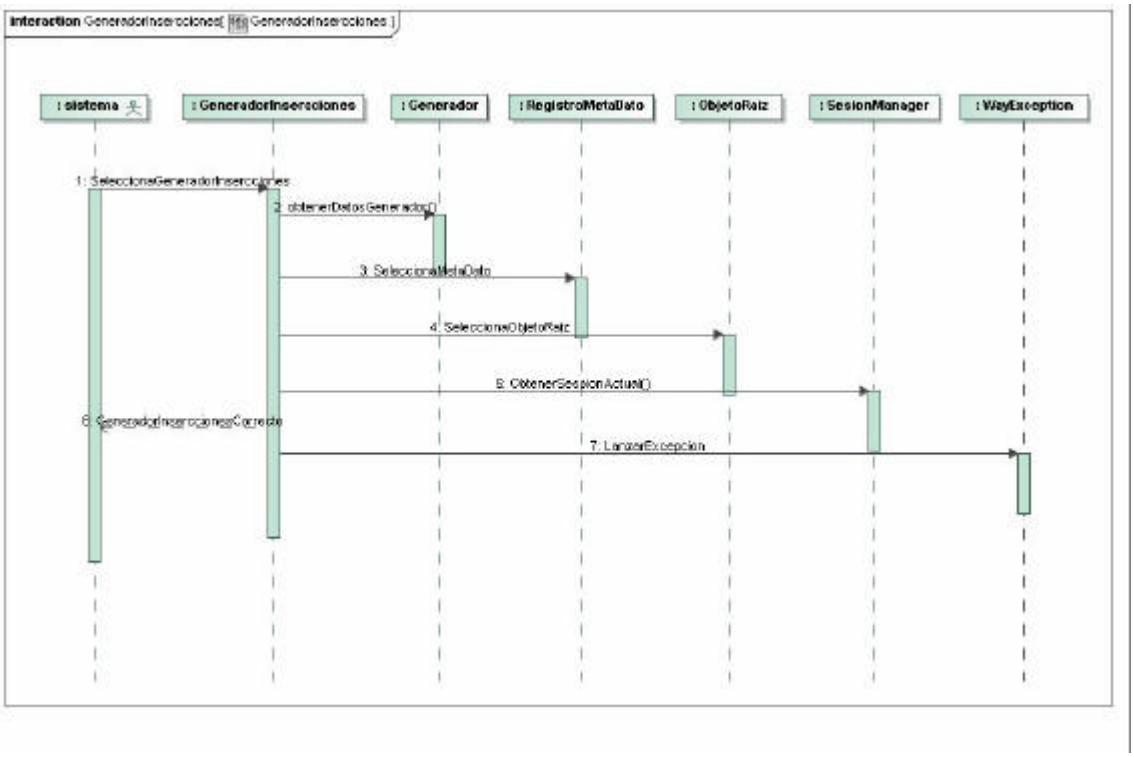
Este proceso de motor del framework puede separarse de la parte de bases de datos, y probar de forma independiente mediante test unitarios como por ejemplo junit. Aun así, con la unión entre los mismos se realiza la unión fundamental para que el ORM tenga la fortaleza, simplicidad y seguridad que venimos contando hasta ahora.

Especificación Caso de Uso: CU GeneradorInsercciones

En este caso de uso, se utiliza un generador para realizar las inserciones con las columnas con sus valores y los añade a la base de datos cuando las filas son insertadas. No es soportado por todas las bases de datos.



Actor Principal: el sistema realiza un generador de inserciones con un objeto de tipo raíz o genérico que puede ser cualquier tipo de dato y el nombre del mismo.



Actor Secundario:

Precondiciones: deberá previamente tener un objeto raíz y su nombre y dependerá del caso de uso CU ManejarGenerador.

Postcondiciones: se realizará el generador de inserciones para añadir a la base de datos y controlar cuando se realizan dichas inserciones.

Flujo principal:

Acción de actor	Respuesta del sistema
El sistema realiza la petición de ejecutar un generador de inserciones.	El sistema comprueba si existe un valor de tipo dato especificado con su nombre y si se puede insertar correctamente.

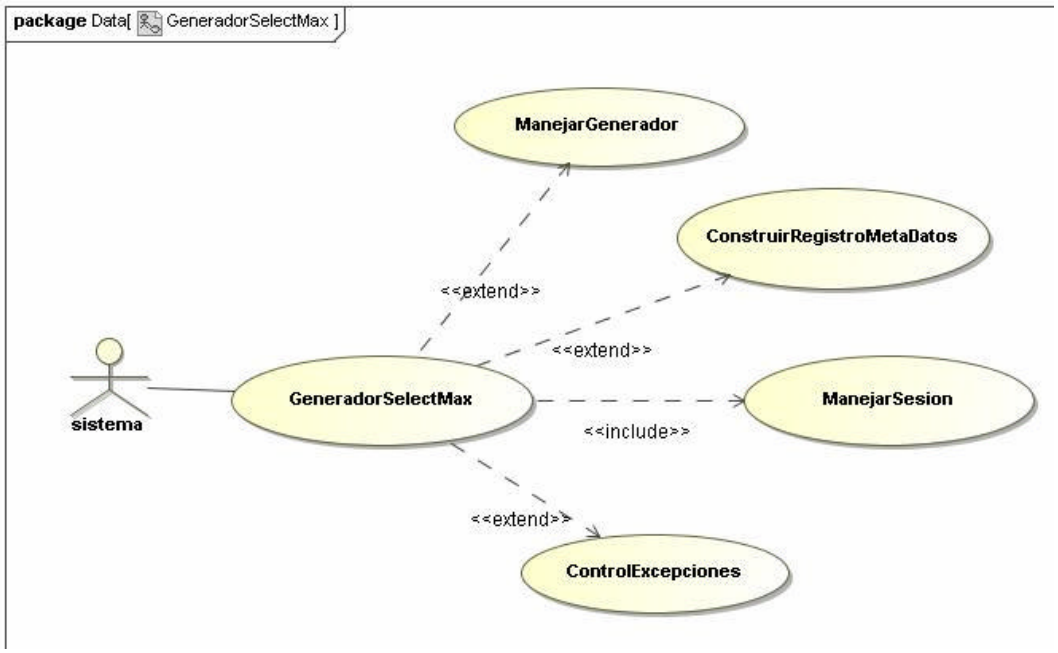
Especificación Caso de Uso: CU GeneradorSelectMax

En este caso de uso, se utiliza un generador para realizar la máxima selección del campo seleccionado. Se utiliza similar al select de SQL. No es soportado por todas las bases de datos.

Actor Principal: el sistema realiza un generador de select máximo para obtener un objeto que cumpla con las condiciones.

Actor Secundario:

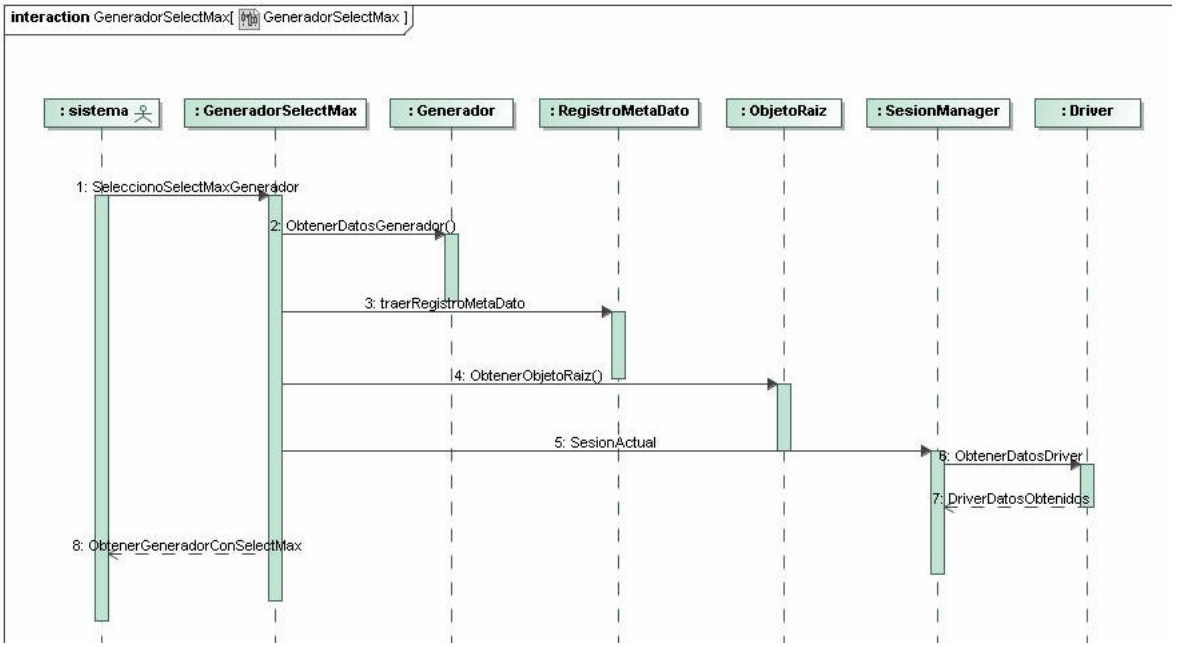
Precondiciones: deberá previamente tener un objeto raíz y su nombre y dependerá del caso de uso CU ManejarGenerador.



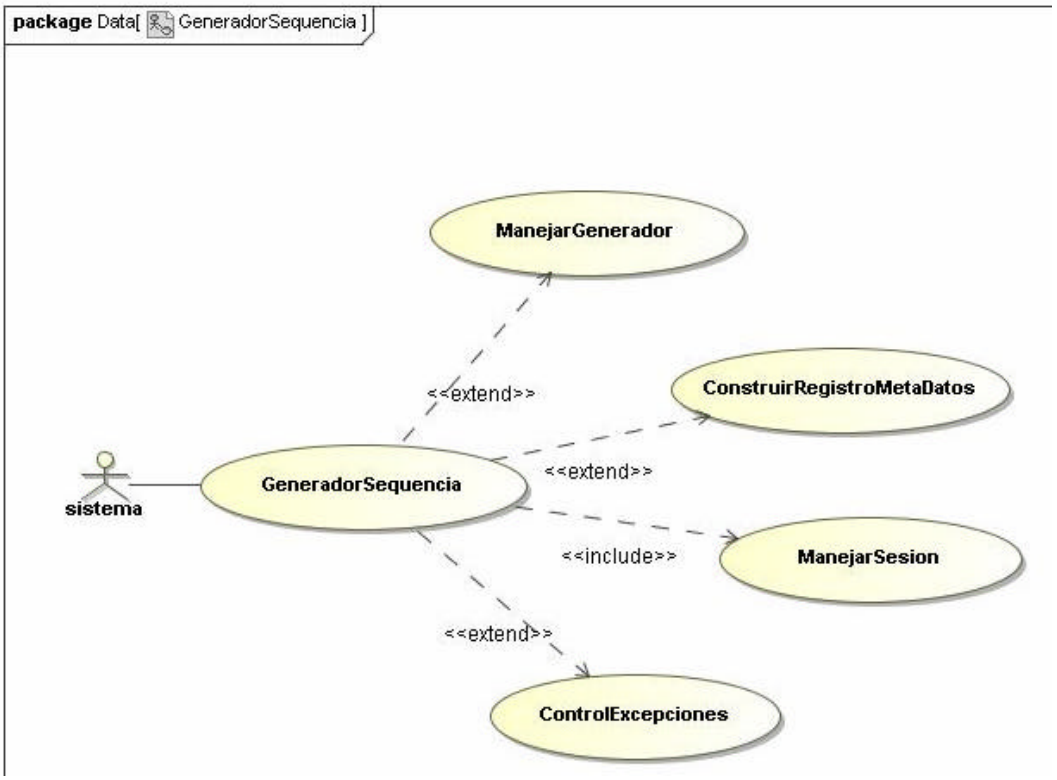
Postcondiciones: se realizará el generador de SelectMax para añadir a la base de datos y controlar si devuelve el máximo o no, o si no devuelve nada por no haber datos.

Flujo principal:

Acción de actor	Respuesta del sistema
El sistema realiza la petición de ejecutar un generador de select max	El sistema comprueba si existe un valor de tipo dato especificado con su nombre y si se puede realizar la generación de este tipo de generador con select max.



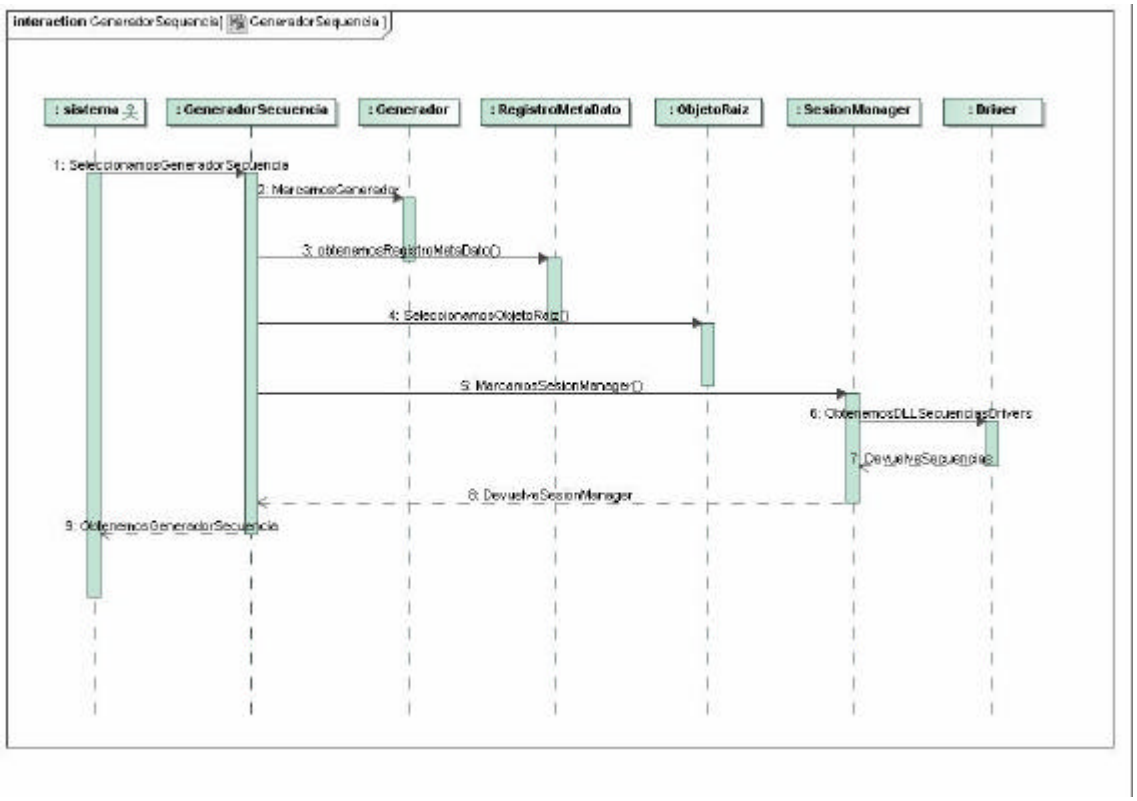
Especificación Caso de Uso: CU GeneradorSe quencia



En este caso de uso, se utiliza un generador para realizar generación de secuencias con sus iteraciones en las claves primarias de los objetos del modelo de datos orientado a objetos. No es soportado por todas las bases de datos. La utilización más común es para bases de datos Oracle.

Actor Principal: el sistema realiza un generador de secuencias para los sistemas gestores de bases de datos que lo soporten.

Actor Secundario:



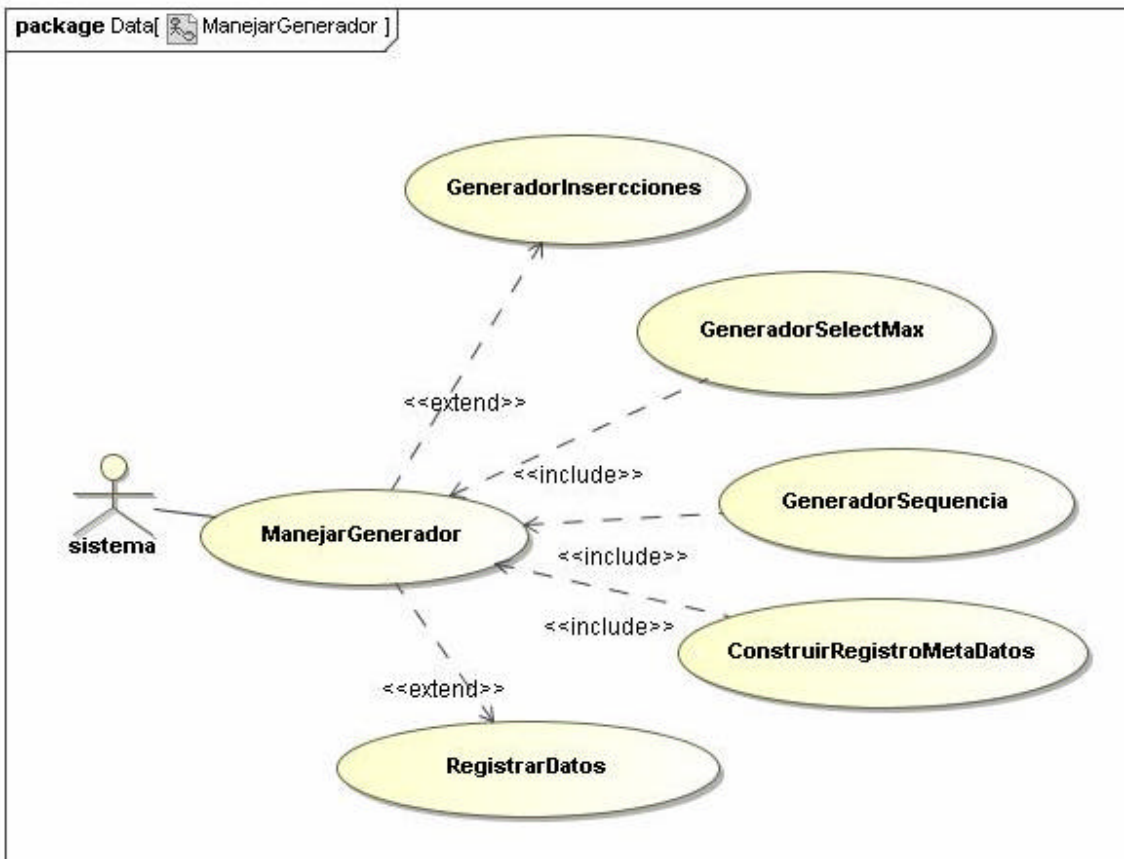
Precondiciones: deberá previamente tener un objeto raíz y su nombre y dependerá del caso de uso CU ManejarGenerador.

Postcondiciones: se realizará el generador de secuencias para añadir a los campos de la base de datos que especifiquemos, en la mayor parte de los casos para tipos de datos incrementales como claves primarias.

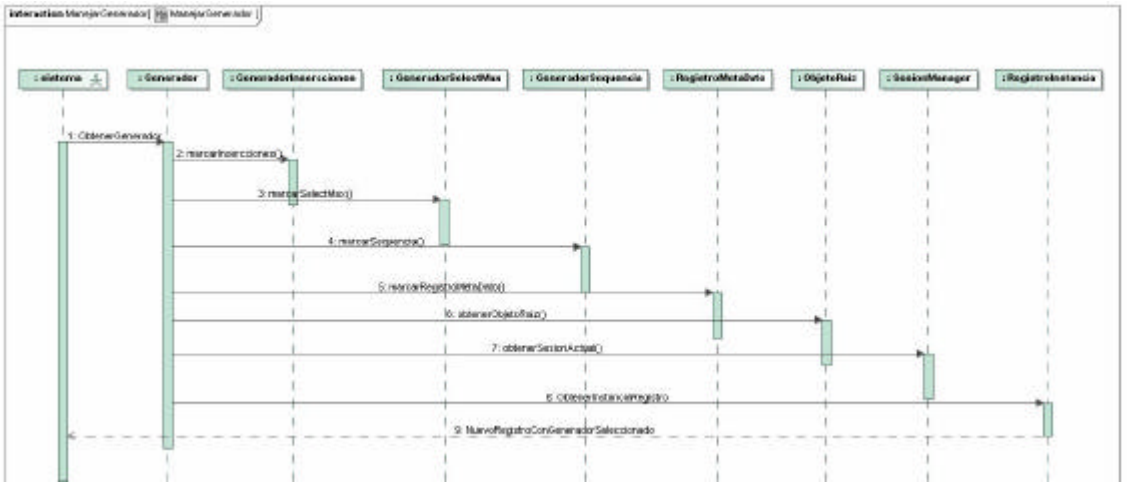
Flujo principal:

Acción de actor	Respuesta del sistema
El sistema realiza la petición de ejecutar un generador de secuencias.	El sistema comprueba si el tipo de sistema gestor de bases de datos soporta secuencias y si es así, realiza las generaciones que se hayan solicitado.

Especificación Caso de Uso: CU ManejarGenerador



En este caso de uso, se utiliza un generador para realizar ciertas particularidades de operaciones a nivel base de datos, pero siempre en orientación a objetos. En este caso, se trata de realizar 3 tipos de generadores: generador de Inserciones, generador de SelectMax y generador de secuencias. Cada uno de ellos estará implementado en un caso de uso porque cada uno tiene unas particularidades.



Actor Principal: el sistema realiza operaciones básicas para seleccionar un tipo de generación de operaciones que puede ser inserción, selectmax o secuencias, todos ellos no soportados por todas las bases de datos.

Actor Secundario:

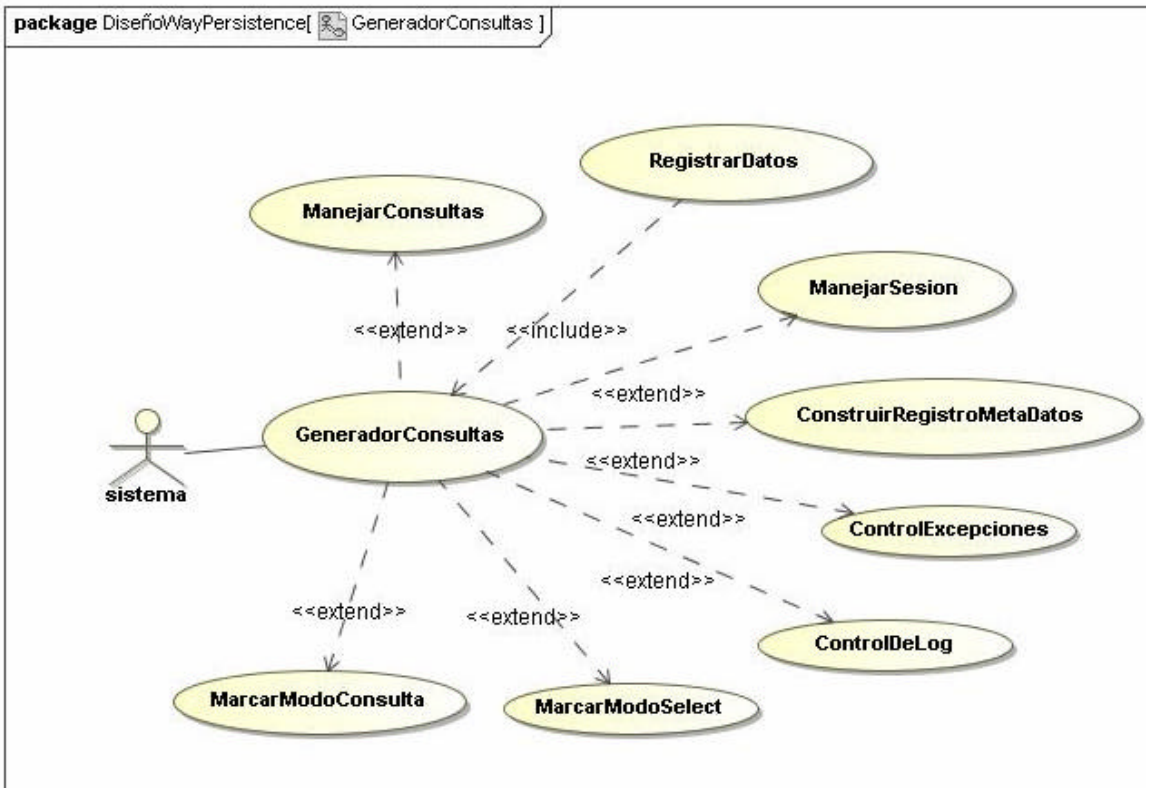
Precondiciones: deberá previamente tener un metadato para poder aplicarle cualquier tipo de generador siempre que se quiera realizar.

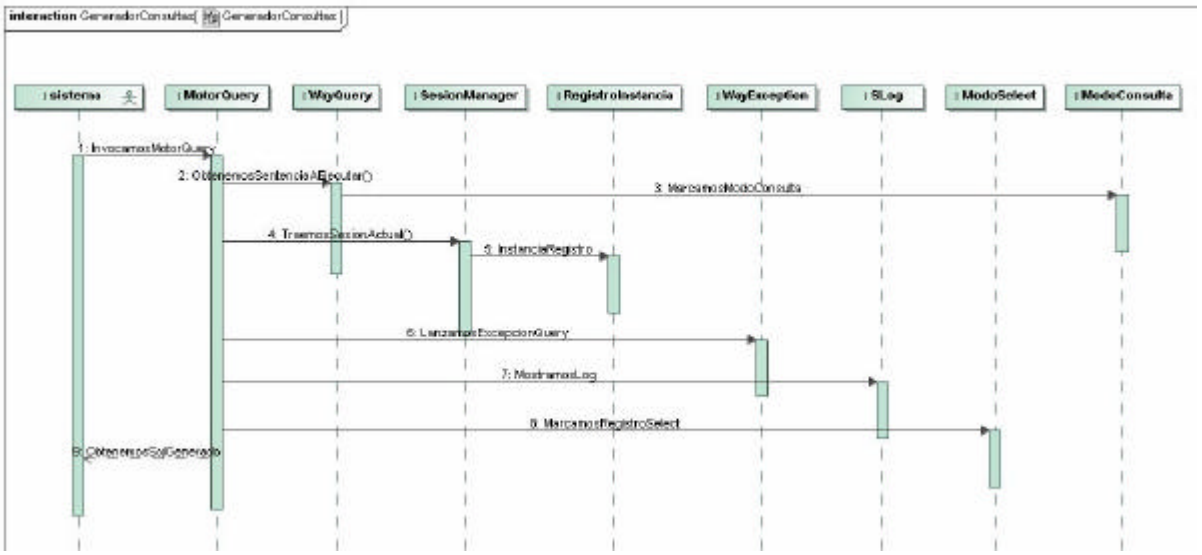
Postcondiciones: se realizarán operaciones generando claves utilizando filas para separar las secuencias de la tabla. Estas operaciones pueden ser las citadas anteriormente.

Flujo principal:

Acción de actor	Respuesta del sistema
El sistema realiza la petición de ejecutar un generador especificando su tipo	El sistema comprueba si el metadato puede soportar el tipo elegido y lo redirige al tipo escogido
El sistema selecciona el tipo de generador: Inserciones	El sistema redirige al caso de uso <CU GenerarInsercciones>
El sistema selecciona el tipo de generador: SelectMax	El sistema redirige el caso de uso <CU GenerarSelectMax>
El sistema selecciona el tipo de generador. Secuencia	El sistema redirige al caso de uso <CU GenerarSequencia>

Especificación Caso de Uso: CU GeneradorConsultas





En este caso de uso, se utiliza un generador de consultas para realizar la ejecución de consultas en lenguaje SQL. Actuará de motor de consultas que hayan sido manejadas previamente y actuará como motor ejecutor de las mismas para que sean interpretadas en la sesión y pasadas a bases de datos. Se trata de todas las consultas que cumplan con el patrón CRUD y realicen alguna acción sobre la base de datos.

Depende del caso de uso <CU ManejarConsultas> que realiza buena parte de la lógica de las consultas de todo tipo y del caso de uso <CU ManejarSesion>.

Actor Principal: el sistema realiza operaciones de generación de SQL correspondiente a operaciones básicas sobre la base de datos.

Actor Secundario:

Precondiciones: deberá previamente tener realizadas operaciones de consultas de cualquier tipo SQL para realizar el manejo de las mismas internamente a través de esta funcionalidad.

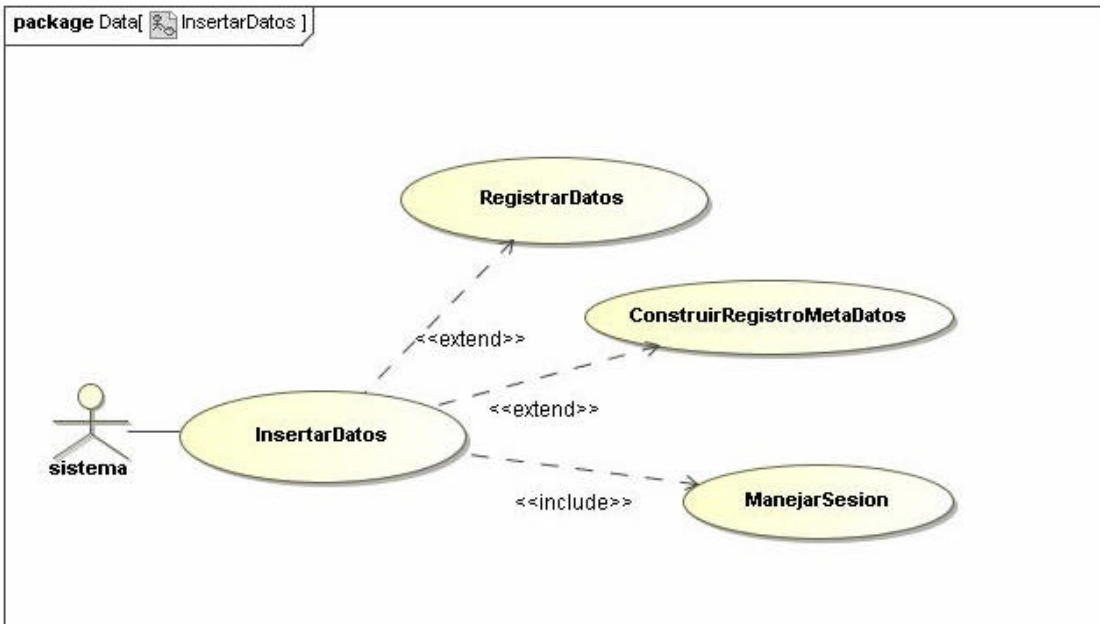
Postcondiciones: se realizará la funcionalidad que permita generar el SQL que corresponde a las consultas realizadas en los casos de uso relacionados, principalmente el de manejo de consultas.

Flujo principal:

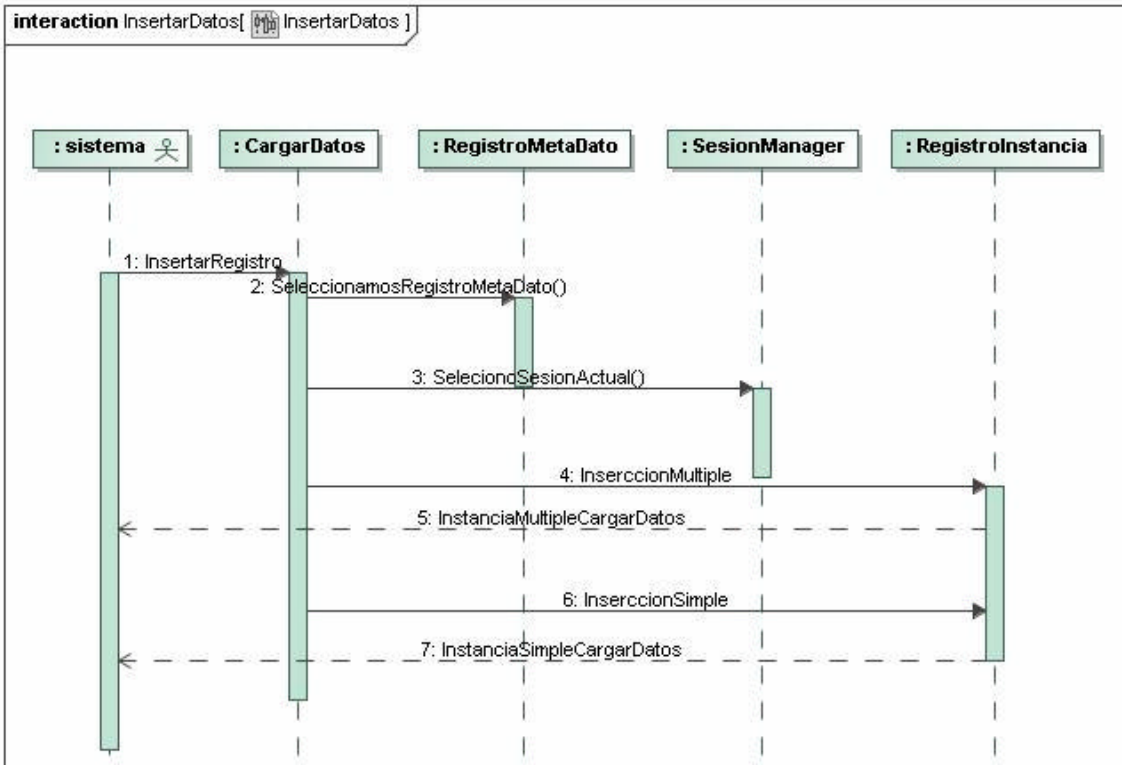
Acción de actor	**Respuesta del sistema**
El sistema realiza la generación de consultas SQL	El sistema comprueba si hay operaciones de SQL realizadas y comprueba que se pueden manejar las mismas para su ejecución inmediata.

Especificación Caso de Uso: CU InsertarDatos

En este caso de uso, se creará una funcionalidad que a partir de una instancia de Registro de datos nos permita ejecutar la inserción de datos de todo tipo bien de un único registro o de múltiples. Esta inserción será dependiente de los casos de uso: <CU ConstruccionRegistroDatos>, <CU RegistrarDatos>, <CU ManejarSession>, <CU EscogerTipoObjeto>.



Actor Principal: el sistema realiza inserciones sobre la sesión con inserciones de múltiples



registros o tuplas o de una única inserción.

Actor Secundario:

Precondiciones: deberá previamente tener un objeto de tipo Registro para poder insertar sobre la sesión.

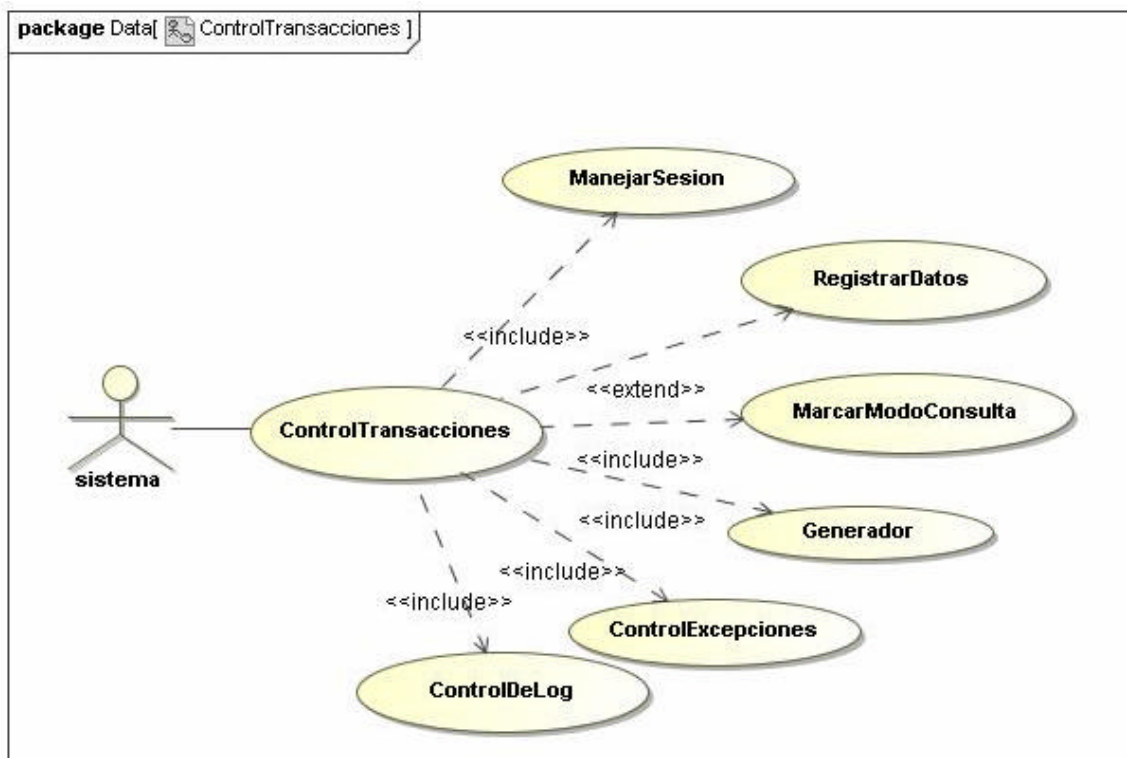
Postcondiciones: se realizará la funcionalidad que permita insertar en sesión sobre la memoria caché para posteriormente con las herramientas que poseerá el ORM, hacer commit() en la base de datos para realizar el guardado de la acción efectuada por esta funcionalidad.

Flujo principal:

Acción de actor	Respuesta del sistema
El sistema realiza la inserción de un registro o varios en la memoria caché.	El sistema realiza la inserción y lo guarda en memoria caché. Posteriormente se guardará en bases de datos o se rechazará de memoria caché si así se cree conveniente.

Especificación Caso de Uso: CU ControlTransacciones

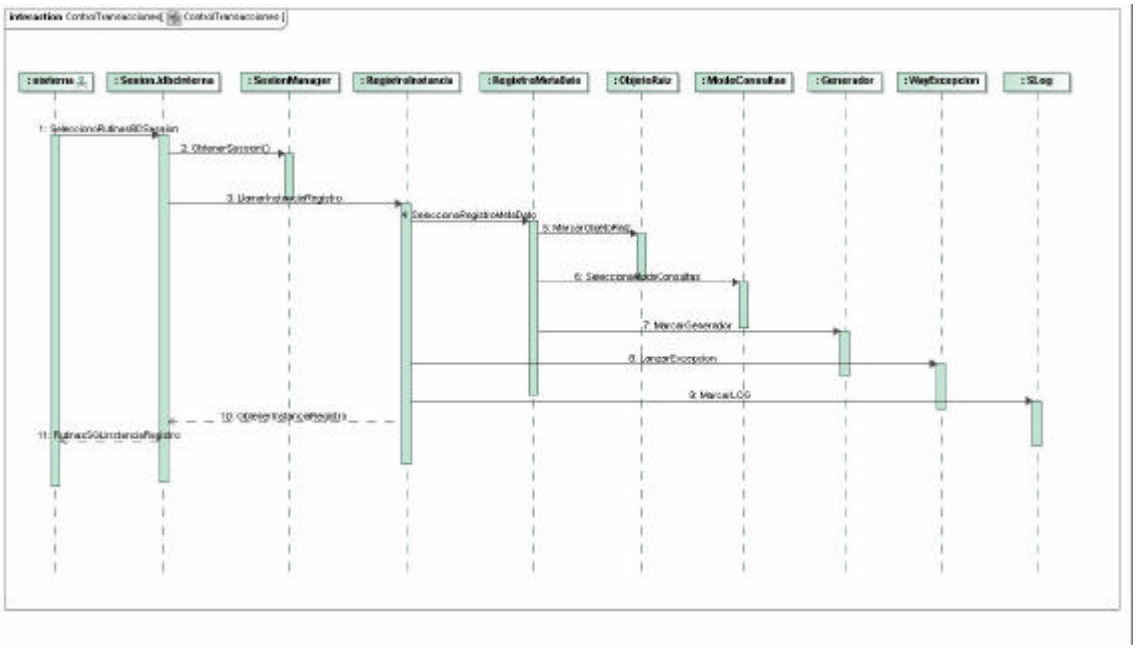
En este caso de uso, se implementa una funcionalidad que se encarga de llevar un control generalizado de las transacciones del framework de persistencia. Contiene el acceso a todas y cada una de las rutinas a base de datos. Controla el estado de la memoria caché en todo momento con sus datos, inserciones y modificaciones que se van produciendo. No permite que existan repetidas instancias en la memoria caché para gestionar la misma con mayor eficiencia.



Actor Principal: el sistema realiza un manejo de la memoria caché y las transacciones que se producen en todo el ORM independientemente de la base de datos que se utilice.

Actor Secundario:

Precondiciones: deberá previamente tener instancias de registros, conexión con el manejo



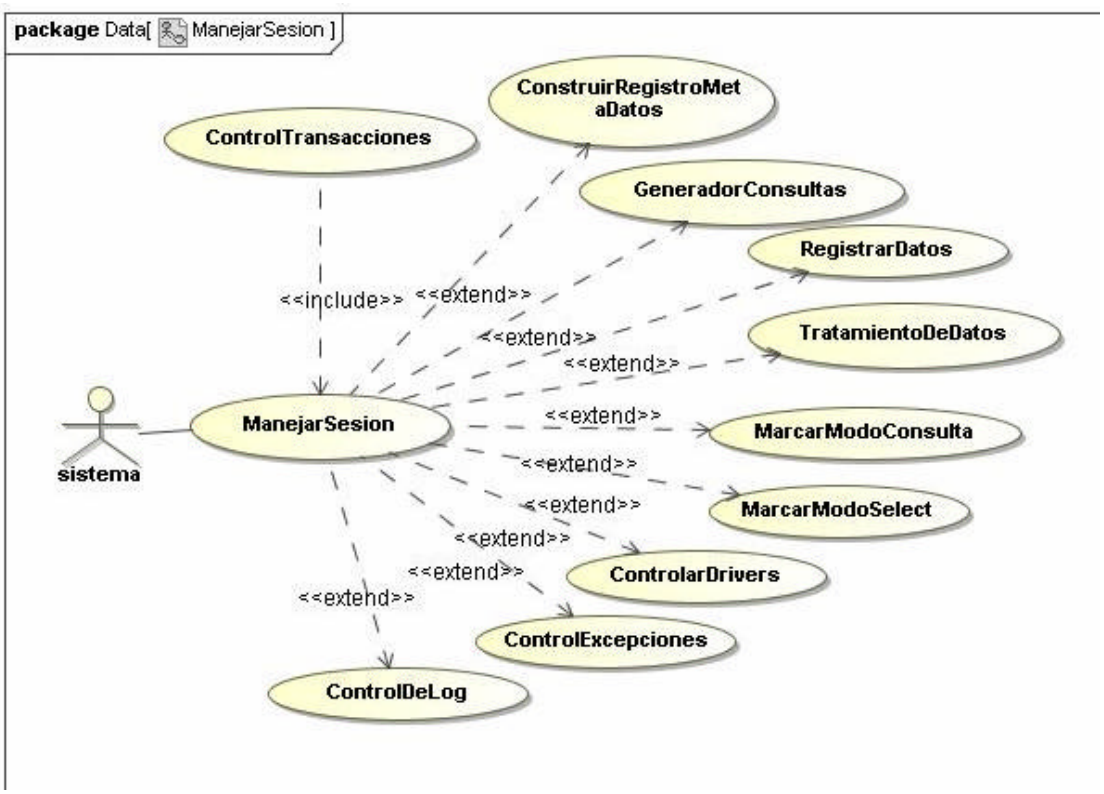
de sesiones, registro de metadatos, y especificaciones de tipos de datos.
Postcondiciones: se realizará el manejo transaccional del framework de persistencia con control absoluto de la memoria caché para evitar mezclas entre conexiones y duplicidad de datos. Los casos de uso relacionados son: <CU ManejarSesión>, <CU ConstrucciónRegistroDatos>, <CU RegistroInstancia>, etc.

Flujo principal:

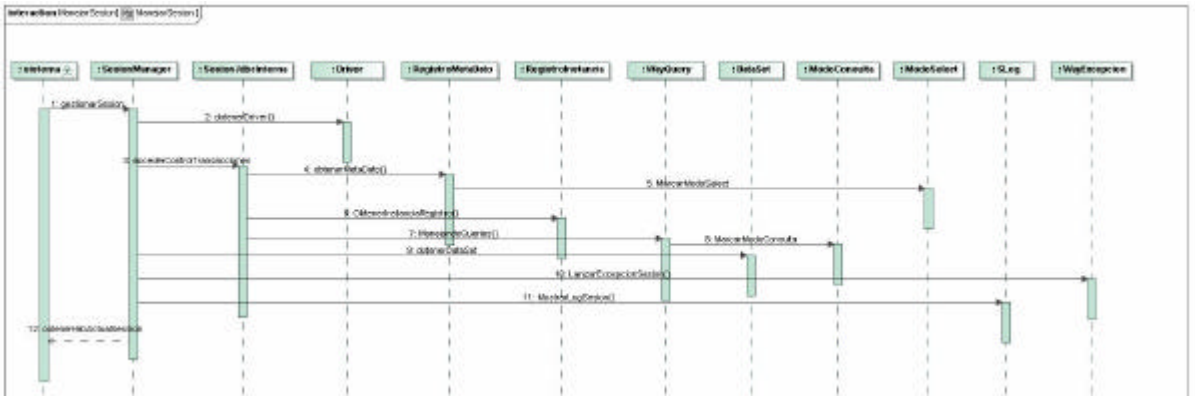
Acción de actor	Respuesta del sistema
El sistema realiza el manejo de transacciones.	El sistema comprueba el estado de la memoria caché, las transacciones y contrasta con el manejo de la sesión el estado actual.

Especificación Caso de Uso: CU ManejarSesión

En este caso de uso, se creará una funcionalidad que será uno de los pilares fundamentales del framework. Este pilar se basa en el manejo de las operaciones sobre el dataset y la conexión con la base de datos. El manejador de sesiones permitirá unir la sesión con el dataset y las instancias de los registros. Es dependiente del caso de uso definido anteriormente denominado < CU ControlTransacciones>.



Permite realizar operaciones entre conexiones del tipo open, begin, commit, flush. Es decir, permite manejar las sesiones de la base de datos, abrilas, cerrarlas, desecharlas, hacer commit y por consiguiente, permitir una interacción absoluta entre el modelo de objetos y el modelo de datos. Sin esta clase, el framework no llegaría a conectarse en ningún momento con la base de datos, pues el hilo conductor principal para que se produzca el mapeo objeto - relación.



Los casos de uso que dependen de este mismo, son: <CU ControlTransacciones>, <CU ConstruccionRegistroDatos>, <CU RegistrarDatos>, <CU ManejarSession>, <CU ObjetoRaiz>, <CU ManejarConsultas>, etc.

Actor Principal: el sistema realiza operaciones en las sesiones que permiten realizar transacciones sobre la sesión y pasarlas a la base de datos.

Actor Secundario:

Precondiciones: deberá tener instancias de objetos, con registros, transacciones y consultas realizadas para poder introducirlas en la sesión y ejecutarlas para plasmarlas en las base de datos.

Postcondiciones: se realizará la funcionalidad que permita manejar sesiones sin mezclarse entre ellas con la ayuda del nivel transaccional del caso de uso anterior y el manejo de sesiones optimizadas gracias a esta funcionalidad.

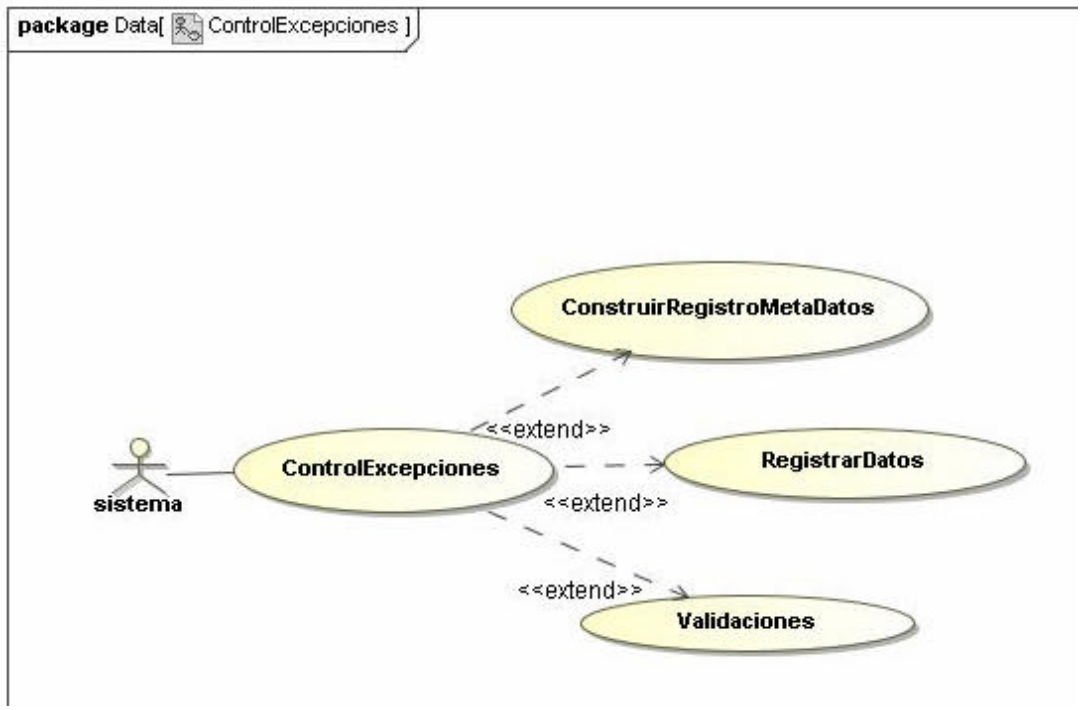
Flujo principal:

Acción de actor	Respuesta del sistema
El sistema realiza el manejo de sesiones.	El sistema controla las transacciones efectuadas sobre las sesiones con un control absoluto con operaciones que permiten realizar apertura, cerrado, desechado y confirmado de transacciones en las sesiones.

6. PROCESO DE UTILIDADES.

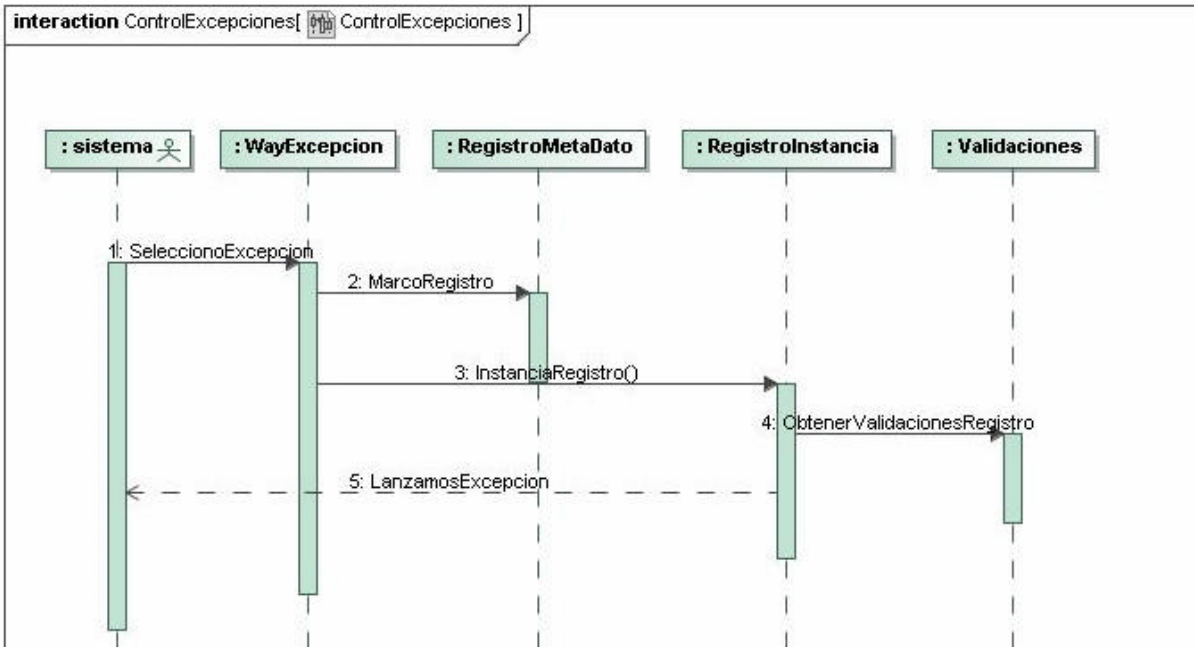
Especificación Caso de Uso: CU ControlExcepciones

En este caso de uso, se trata el manejo de excepciones de todo el ORM. El tipo de



excepciones puede ser: jdbc, bug, interno o de pruebas. En la mayor parte de los procesos se utilizan para transacciones que no han podido continuar.

Actor Principal: el sistema lanza excepciones siempre que produzcan errores en cualquier tipo de procesos de los tipos citados anteriormente.



Actor Secundario:

Precondiciones: deberá previamente haber habido un error de los tipos indicados y se utilizará el control de excepciones para lanzar la que se considere adecuada.

Postcondiciones: se realizará la funcionalidad que permita controlar las excepciones y lanzar la que corresponda en función del bug que haya salido. Este tipo de inserciones será controlado en este caso de uso.

Flujo principal:

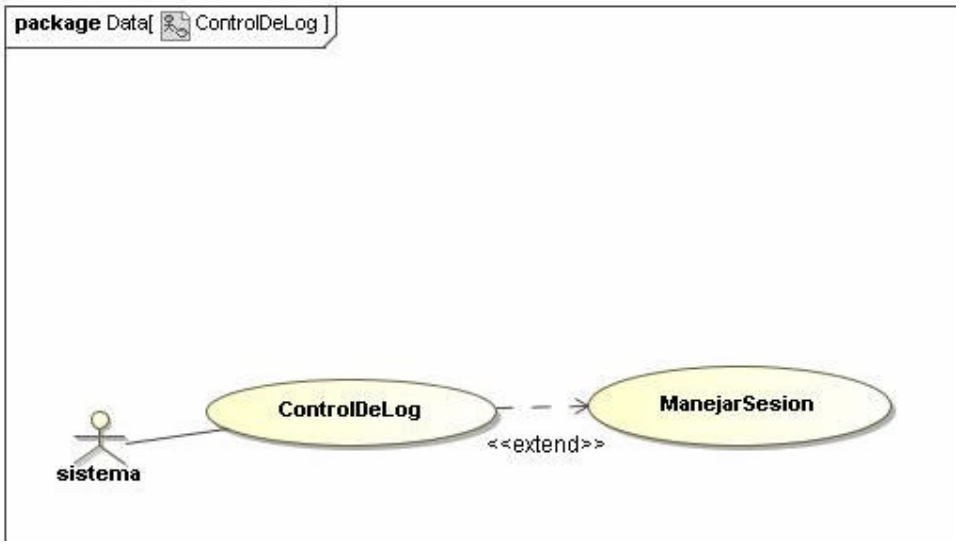
Acción de actor	**Respuesta del sistema**
El sistema realiza el lanzado de una excepción por un error del ORM.	El sistema devuelve la excepción capturada en función del tipo o proceso que haya fallado previamente.

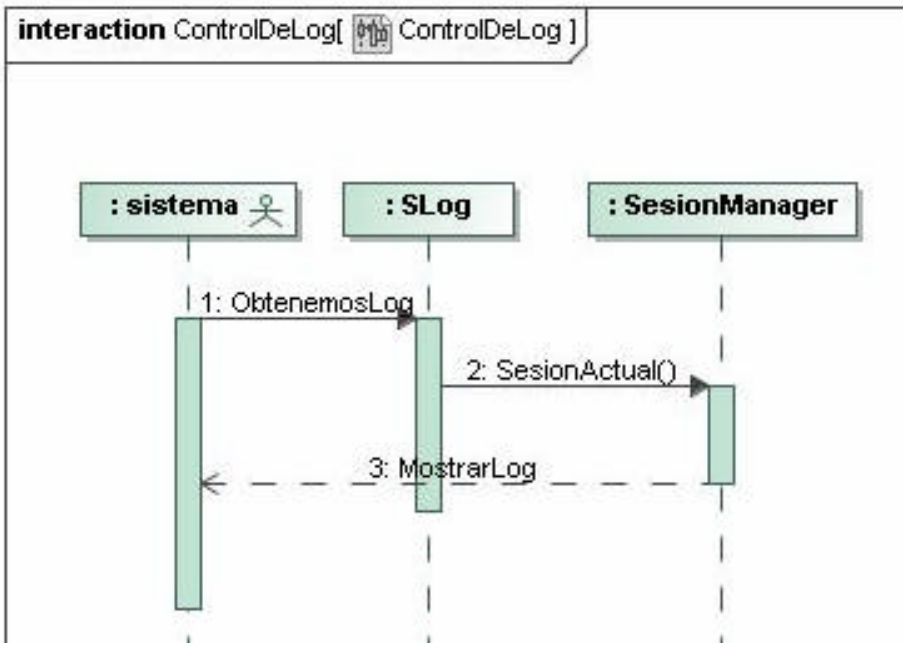
Especificación Caso de Uso: CU ControlDeLog

En este caso de uso, se creará una funcionalidad que permita mostrar mensajes de distintos niveles independientemente de la parte del framework que se esté ejecutando. Se tendrán varios niveles definidos en función de la severidad de los mismos. El log en un principio extenderá de Slf4j.

Actor Principal: el sistema mostrará mensajes de log según hayamos definido en los momentos adecuados, según niveles.

Actor Secundario:
Precondiciones: deberá previamente tener inyectado el jar de slf4j del que extiende nuestro log personalizado.





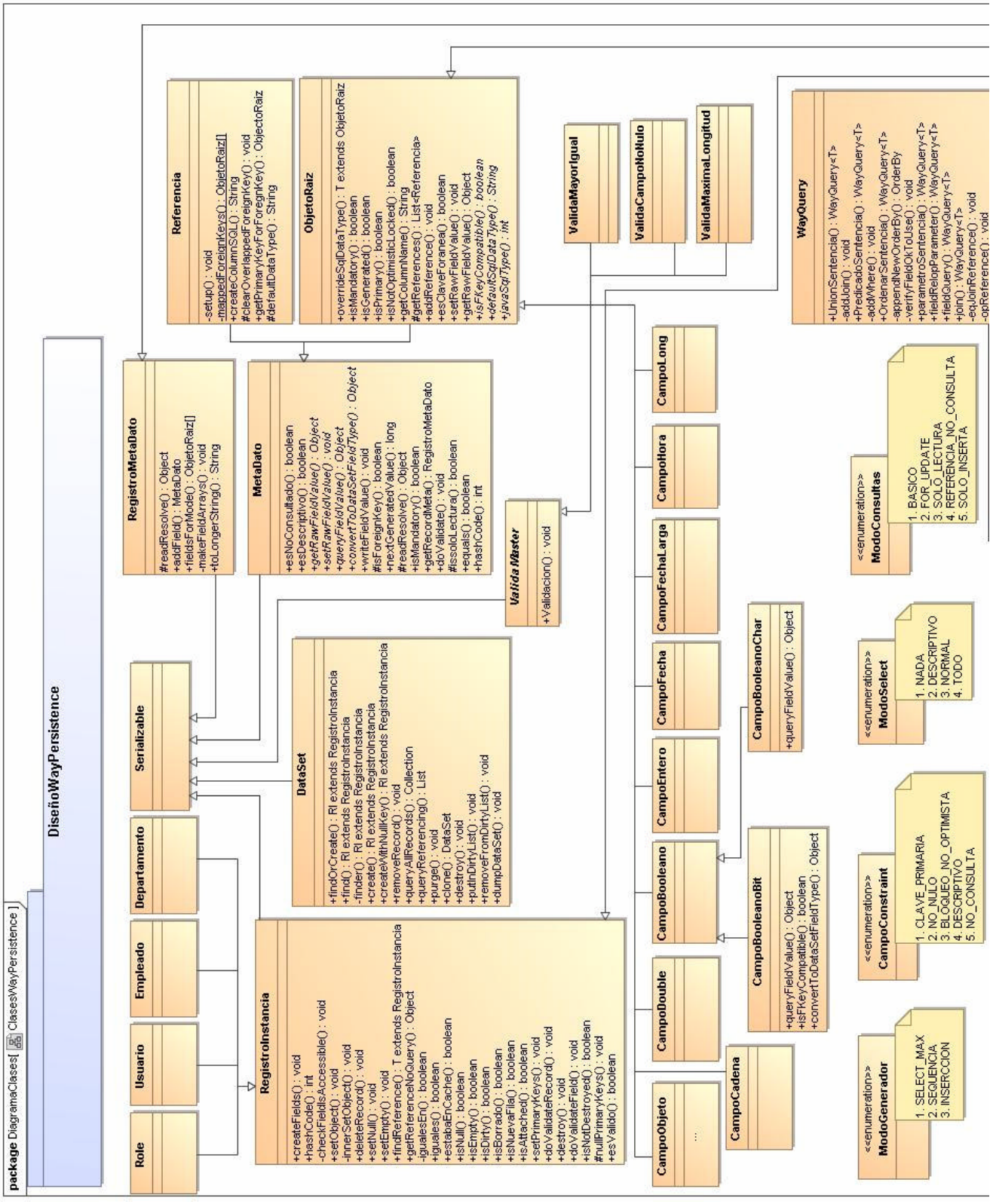
Postcondiciones: se realizará la funcionalidad que permita mostrar mensajes de log de nuestro personalizado log que extiende de slf4j. Estos mensajes y niveles serán personalizados en función de las necesidades que tengamos en cada momento.

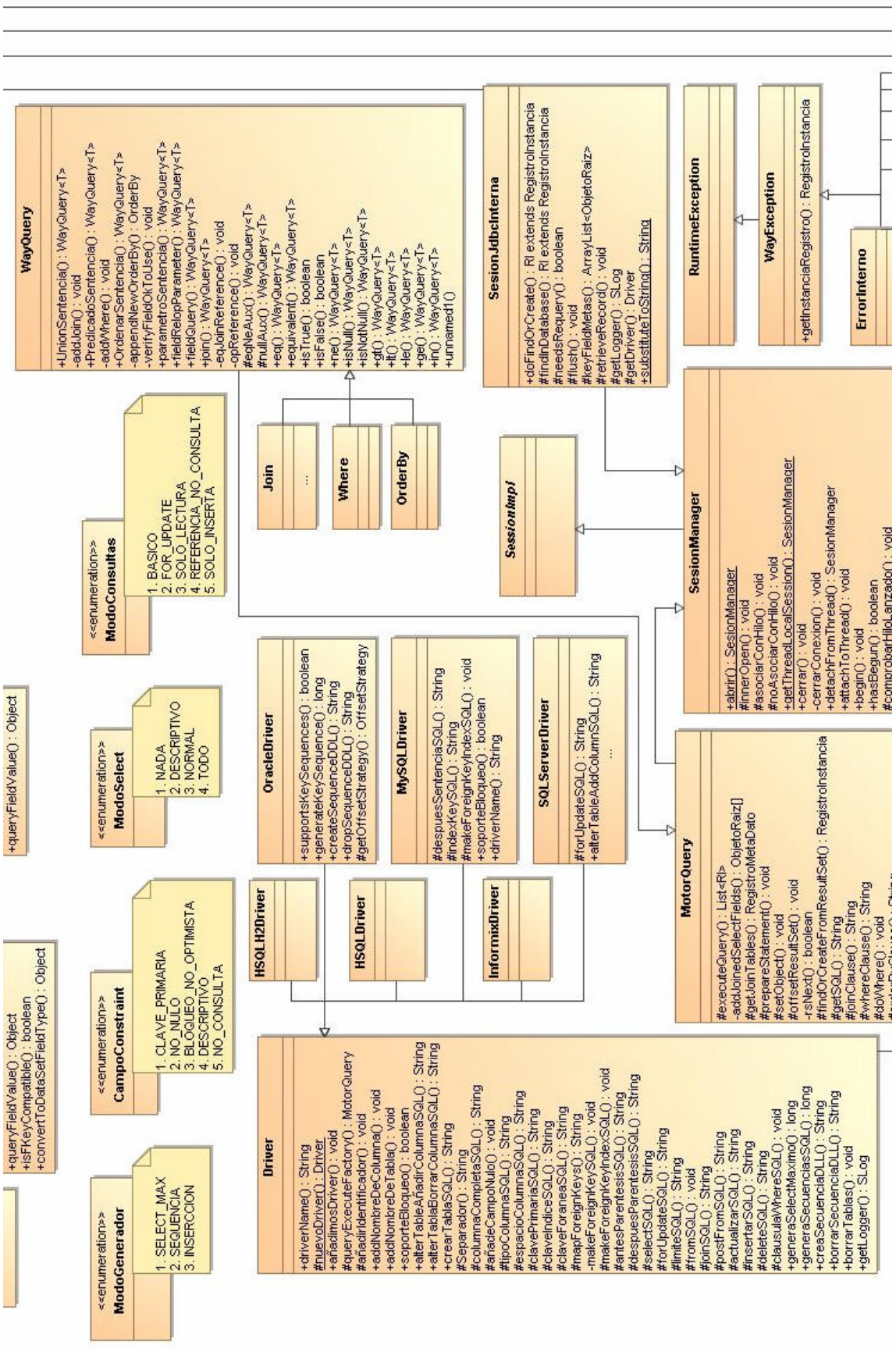
Flujo principal:

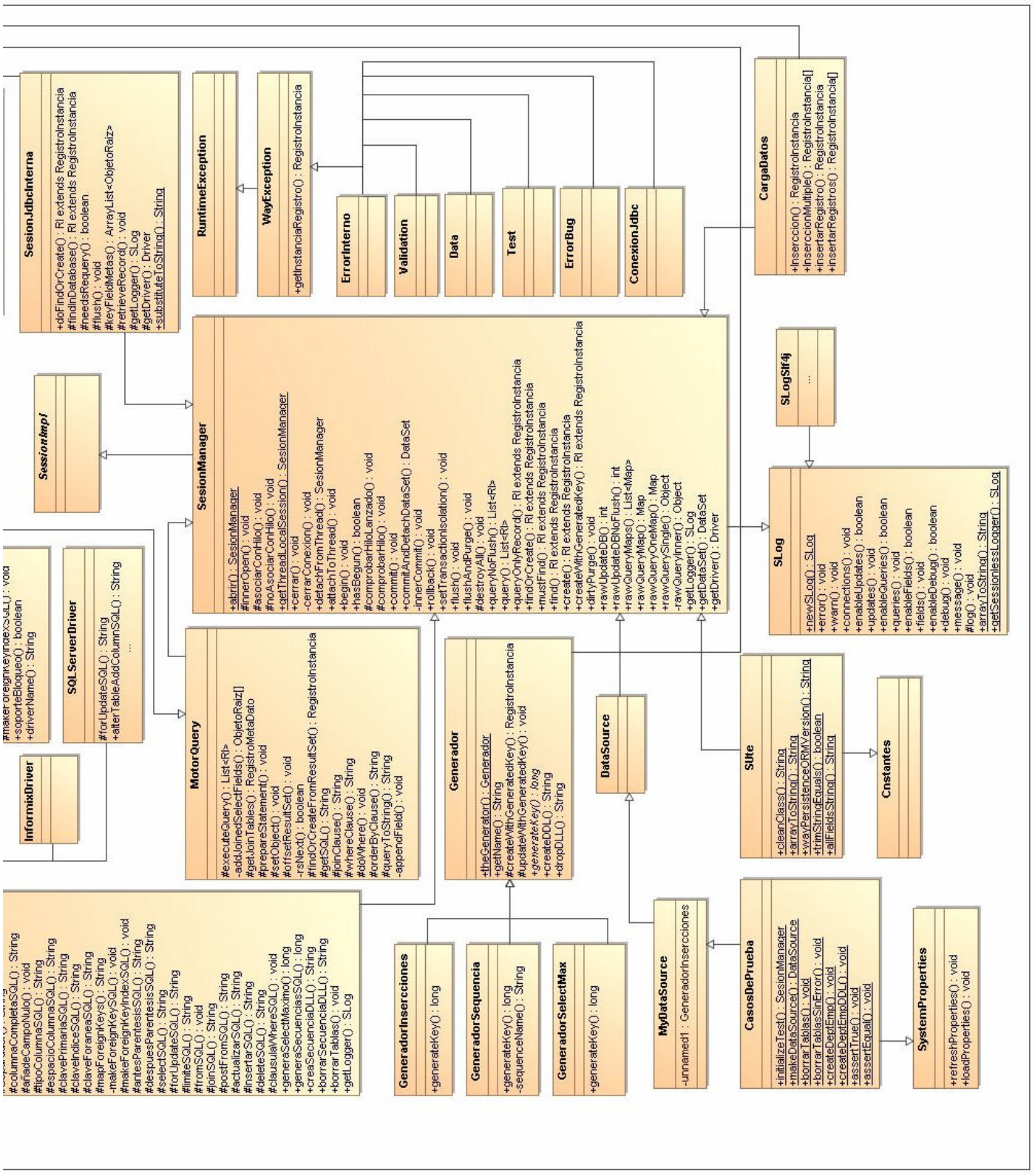
Acción de actor	**Respuesta del sistema**
El sistema llamará al log para mostrar un mensaje.	El sistema realiza la inserción de log's que muestren la situación del ORM a todos los niveles.

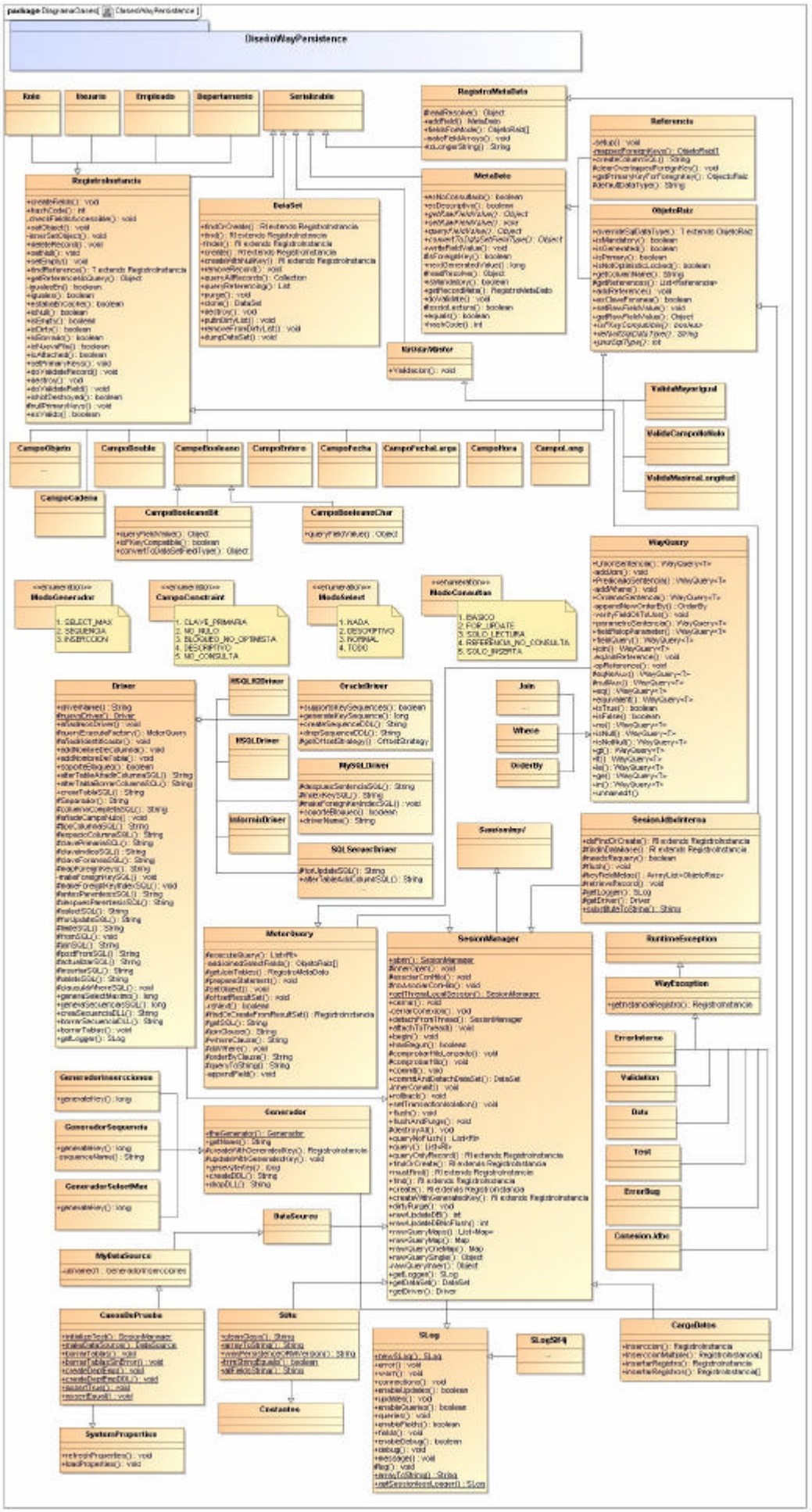
5.4. DISEÑO: DIAGRAMA DE CLASES WAYPERSISTENCE

Se realizará un diseño de las clases que se implementarán en el ORM con sus métodos.




WayQuery
+UnionSentencia() : WayQuery<T>
-addJoin() : void
+PredicadoSentencia() : WayQuery<T>
-addWhere() : void
+OrdenarSentencia() : WayQuery<T>
-appendNewOrderBy() : OrderBy
-verifyFieldOkToUse() : void
+parametroSentencia() : WayQuery<T>
+fieldRelopParameter() : WayQuery<T>
+fieldQuery() : WayQuery<T>
+join() : WayQuery<T>
-equJoinReference() : void
-opReference() : void
#eqNeAux() : WayQuery<T>
#nullAux() : WayQuery<T>
+eq() : WayQuery<T>
+equivalent() : WayQuery<T>
+isTrue() : boolean
+isFalse() : boolean
+ne() : WayQuery<T>
+isNull() : WayQuery<T>
+isNotNull() : WayQuery<T>
+gt() : WayQuery<T>
+lt() : WayQuery<T>
+le() : WayQuery<T>
+ge() : WayQuery<T>
+in() : WayQuery<T>
+unnamed1()
Sesion.JdbcInterna
+doFindOrCreate() : RI extends RegistroInstancia
#findInDatabase() : RI extends RegistroInstancia
#needsRequery() : boolean
#flush() : void
#keyFieldMetas() : ArrayList<ObjetoRaiz>
#retrieveRecord() : void
#getLogger() : SLog
#getDriver() : Driver
+substituteToString() : String
RuntimeException
WayException
+getInstanciaRegistro() : RegistroInstancia
ErrorInterno
Join
...
Where
OrderBy
SessionImpl
SesionManager
+abrir() : SesionManager
#innerOpen() : void
#asociarConHilo() : void
#noAsociarConHilo() : void
+getThreadLocalSession() : SesionManager
+cerrar() : void
-cerrarConexion() : void
+detachFromThread() : SesionManager
+attachToThread() : void
+begin() : void
+hasBegun() : boolean
#comprobarHiloLanzado() : void
<<enumeration>>
ModoConsultas
1. BASICO
2. FOR_UPDATE
3. SOLO_LECTURA
4. REFERENCIA_NO_CONSULTA
5. SOLO_INSERTA
<<enumeration>>
ModoSelect
1. NADA
2. DESCRIPTIVO
3. NORMAL
4. TODO
OracleDriver
+supportsKeySequences() : boolean
+generateKeySequence() : long
+createSequenceDDL() : String
+dropSequenceDDL() : String
#getOffsetStrategy() : OffsetStrategy
MySQLDriver
#despuesSentenciaSqlQ() : String
#indexKeySqlQ() : String
#makeForeignKeyIndexSqlQ() : void
+soporteBloqueo() : boolean
+driverName() : String
SQLServerDriver
#forUpdateSQL() : String
+alterTableAddColumnSQL() : String
...
HSQLH2Driver
HSQLDriver
InformixDriver
MotorQuery
#executeQuery() : List<RI>
-addJoinedSelectFields() : ObjetoRaiz[]
#getJoinTables() : RegistroMetaDato
#prepareStatement() : void
#setObject() : void
#offsetResultSet() : void
-rsNext() : boolean
#findOrCreateFromResultSet() : RegistroInstancia
#getSQL() : String
#joinClause() : String
#whereClause() : String
#doWhere() : void
<<enumeration>>
CampoConstraint
1. CLAVE_PRIMARIA
2. NO_NULO
3. BLOQUEO_NO_OPTIMISTA
4. DESCRIPTIVO
5. NO_CONSULTA
<<enumeration>>
ModoGenerador
1. SELECT_MAX
2. SEQUENCIA
3. INSERCCION
Driver
+driverName() : String
#nuevoDriver() : Driver
+añadimosDriver() : void
#queryExecuteFactory() : MotorQuery
#añadirIdentificador() : void
+addNombreDeColumna() : void
+addNombreDeTabla() : void
+soporteBloqueo() : boolean
+alterTableAñadirColumnaSQL() : String
+alterTableBorrarColumnaSQL() : String
+crearTablaSQL() : String
#Separador() : String
#columnaCompletaSQL() : String
#añadeCampoNulo() : void
#tipoColumnaSQL() : String
#espacioColumnaSQL() : String
#clavePrimariaSQL() : String
#claveIndiceSQL() : String
#claveForaneaSQL() : String
#mapForeignKeys() : String
-makeForeignKeySQL() : void
#makeForeignKeyIndexSQL() : void
#antesParentesisSQL() : String
#despuesParentesisSQL() : String
#selectSQL() : String
#forUpdateSQL() : String
#limiteSQL() : String
#fromSQL() : void
#joinSQL() : String
#postFromSQL() : String
#actualizarSQL() : String
#insertarSQL() : String
#deleteSQL() : String
#clausulaWhereSQL() : void
+generaSelectMaximo() : long
+generaSecuenciasSQL() : long
+creaSecuenciaDLL() : String
+borrarSecuenciaDLL() : String
+borrarTablas() : void
+getLogger() : SLog
+queryFieldValue() : Object
+isFKeyCompatible() : boolean
+convertToDataSetFieldType() : Object
+queryFieldValue() : Object


SesionJdbcInterna
+doFindOrCreate() : RI extends RegistroInstancia
#findInDatabase() : RI extends RegistroInstancia
#needsRequery() : boolean
#flush() : void
#keyFieldMetas() : ArrayList<ObjetoRaiz>
#retrieveRecord() : void
#getLogger() : SLog
#getDriver() : Driver
+substituteToString() : String
RuntimeException
WayException
+getInstanciaRegistro() : RegistroInstancia
ErrorInterno
Validation
Data
Test
ErrorBug
Conexion.Jdbc
CargaDatos
+inserccion() : RegistroInstancia
+inserccionMultiple() : RegistroInstancia[]
+insertarRegistro() : RegistroInstancia
+insertarRegistros() : RegistroInstancia[]
SesionManager
+abrir() : SesionManager
#innerOpen() : void
#asociarConHilo() : void
#noAsociarConHilo() : void
+getThreadLocalSesion() : SesionManager
+cerrar() : void
-cerrarConexion() : void
+detachFromThread() : SesionManager
+attachToThread() : void
+begin() : void
+hasBegun() : boolean
#comprobarHiloLanzado() : void
#comprobarHilo() : void
+commit() : void
+commitAndDetachDataSet() : DataSet
-innerCommit() : void
+rollback() : void
+setTransactionIsolation() : void
+flush() : void
+flushAndPurge() : void
#destroyAll() : void
+queryNoFlush() : List<RI>
+query() : List<RI>
+queryOnlyRecord() : RI extends RegistroInstancia
+findOrCreate() : RI extends RegistroInstancia
+mustFind() : RI extends RegistroInstancia
+find() : RI extends RegistroInstancia
+create() : RI extends RegistroInstancia
+createWithGeneratedKey() : RI extends RegistroInstancia
+dirtyPurge() : void
+rawUpdateDB() : int
+rawUpdateDBNoFlush() : int
+rawQueryMaps() : List<Map>
+rawQueryMap() : Map
+rawQueryOneMap() : Map
+rawQuerySingle() : Object
-rawQueryInner() : Object
+getLogger() : SLog
+getDataSet() : DataSet
+getDriver() : Driver
SessionImpl
SLogSlf4j
...
SLog
+newSLog() : SLog
+error() : void
+warn() : void
+connections() : void
+enableUpdates() : boolean
+updates() : void
+enableQueries() : boolean
+queries() : void
+enableFields() : boolean
+fields() : void
+enableDebug() : boolean
+debug() : void
+message() : void
#log() : void
+arrayToString() : String
+getSessionlessLogger() : SLog
SQLServerDriver
#forUpdateSQL() : String
+alterTableAddColumnSQL() : String
...
InformixDriver
MotorQuery
#executeQuery() : List<RI>
-addJoinedSelectFields() : ObjetoRaiz[]
#getJoinTables() : RegistroMetaDato
#prepareStatement() : void
#setObject() : void
#offsetResultSet() : void
-rsNext() : boolean
#findOrCreateFromResultSet() : RegistroInstancia
#getSQL() : String
#joinClause() : String
#whereClause() : String
#doWhere() : void
#orderByClause() : String
#queryToString() : String
-appendField() : void
Generador
+theGenerator() : Generador
+getName() : String
#createWithGeneratedKey() : RegistroInstancia
#updateWithGeneratedKey() : void
+generateKey() : long
+createDDL() : String
+dropDLL() : String
DataSource
SiUte
+cleanClass() : String
+arrayToString() : String
+wayPersistenceORMVersion() : String
+trimStringEquals() : boolean
+allFieldsString() : String
Cnstantes
GeneradorInserccione
+generateKey() : long
GeneradorSequencia
+generateKey() : long
-sequenceName() : String
GeneradorSelectMax
+generateKey() : long
MyDataSource
-unnamed1 : GeneradorInserccione
CasosDePrueba
+initializeTest() : SesionManager
+makeDataSource() : DataSource
+borrarTablas() : void
+borrarTablasSinError() : void
+createDeptEmp() : void
+createDeptEmpDDL() : void
+assertTrue() : void
+assertEqual() : void
SystemProperties
+refreshProperties() : void
+loadProperties() : void


DiseñoWayPersistence
Rate
Usuario
Empleado
Departamento
Serializable
RegistroMetaData
Referencia
RegistroInstancia
DataSet
MetaData
ObjetoRaiz
ValidaMayorIgual
ValidaCampoNoNulo
ValidaMaximaLongitud
CampoObjeto
CampoDouble
CampoBoolean
CampoEntero
CampoFecha
CampoFechaLarga
CampoHora
CampoLong
CampoCadena
CampoBooleanoBit
CampoBooleanoChar
WayQuery
ModoGenerador
CampoConstraint
ModoSelect
ModoConexion
Driver
MSQLXDDriver
MSQLDriver
InformixDriver
OracleDriver
MySQLDriver
SQLServerDriver
Join
Where
OrderBy
SesionAbierta
SesionImpl
MotorQuery
SessionManager
RuntimeException
WayException
ErrorInterno
Validation
Data
Test
ErrorBug
ConexionJdbc
GeneradorIncrCounter
Generador
GeneradorSequence
GeneradorSelectMax
DataSource
MyDataSource
CasosDePrueba
Sitio
Constantes
SLog
SLogSFG
CargaDatos
SystemProperties

VI. RESUMEN

Visión global

Para la implementación de un framework de persistencia se ha pretendido realizar un ORM propio que cumpliese con unas mínimas características como escalabilidad, multiplataforma y adaptabilidad a cualquier sistema gestor de bases de datos, principalmente, a las bases de datos relacionales. Además, se requiere que posea la especial característica de ser transacional y que el ORM asegure que un proyecto software cualquiera, pueda realizar operaciones desde la lógica de negocio sobre la capa de datos, asegurando y garantizando sus transacciones para que sean independientes, con aislamiento entre ellas, que permitan el bloqueo de registro y que aseguren una integridad de los datos. En definitiva, que el motor de persistencia sea transacional y cumpla con el patrón ACID que garantiza atomicidad, coherencia, aislamiento y durabilidad.

En un primer momento, para la creación de un ORM propio se estudiaron los distintos motores de persistencia de la actualidad, fundamentalmente, los utilizados para tecnología Java. Nos centramos principalmente en los frameworks de persistencia que permitiesen realizar un mapeo entre el modelo relacional y el modelo de objetos, es decir, un ORM propiamente dicho.

A nivel de la metodología y la gestión de proyecto, destacar que el seguimiento del proyecto sufrió variaciones con respecto a la planificación y las estimaciones realizadas en el plan de trabajo inicial. En un principio, no se habían contemplado algunas de las tareas o funcionalidades que posteriormente fueron parte importante del proyecto.

Con respecto al análisis y diseño orientado a objetos, se realizó exhaustivamente para WayPersistence con lenguaje UML. Se representaron los distintos casos de uso localizados, diagramas de secuencia y diagrama de clases de toda la aplicación que sufrió modificaciones constantemente hasta conseguir una versión estable y definitiva.

Con respecto a la valoración económica del proyecto, es demasiado arriesgado realizarla, pero podemos decir que el total aproximado de las horas es de 940 horas. Teniendo en cuanta que 300 horas fueron de análisis y diseño, 100 de dirección de proyecto, y el resto de programación, la estimación económica se dispararía demasiado.

Valoración Personal

El desarrollo del proyecto fin de carrera en J2EE ha sido una experiencia enriquecedora para mi, que me ha ayudado a tener una idea más general de la gestión de proyectos, análisis, diseño y programación. Creo que con más tiempo, podría mejorarse muy mucho la eficiencia, eficacia y calidad de este framework de persistencia, pero para haber sido tan sólo 4 meses, creo que el resultado es bueno y positivo.
Sería interesante realizar proyectos similares a lo largo de la carrera que aportaran una visión más cercana del mundo del desarrollo de software y la persistencia. El tema de los frameworks, tanto de presentación como de persistencia se obvia demasiado en la ingeniería, y podría ser interesante orientar al alumnado en este tipo de herramientas que posteriormente utilizará en el mundo laboral.
Estoy satisfecho plenamente con la UOC por ser una universidad adaptada al mundo de hoy, para trabajadores que quieran complementar sus estudios y con un estupendo modelo de estudios.

VII. GLOSARIO

Persistencia de la información: parte crítica en desarrollo de proyectos Software.

Materialización: la materialización es el acto de transformar una representación de datos no orientada a objetos de un almacenamiento persistente de objetos.

Desmaterialización: La desmaterialización es el acto de transformar la representación de datos orientada a objetos persistentes en una representación de datos no orientada a objetos

Persistencia de objetos: Es el hecho de persistir la información de un objeto de forma permanente (guardar) y además, se refiere a recuperar y volver a ver la información introducida o guardada para poder ser utilizada de nuevo.

Framework de Persistencia: Un framework de persistencia es un conjunto de tipos de propósito general, reutilizable y extensible, que proporciona funcionalidad para dar soporte a los objetos persistentes.

ORM: es una técnica de programación para convertir datos entre el sistema de tipos utilizado lenguaje orientado a objeto y bases de datos relacionales.

UML (Unified Modeling Language), Modelado Unificado del Lenguaje, es la especificación más utilizada en la Ingeniería de Software pare representación de diseño y estructuras de datos en los proyectos de software.

OMG (ObjectManagement Group) es un consorcio dedicado al cuidado y el establecimiento de varios estándares de tecnologías orientadas a objetos, tales como UML, CORBA, XMI.

Persistencia: Persistir objetos java en una base de datos relacional es un reto único que implica serializar un árbol de objetos Java en una base de datos de estructura tabular y viceversa.

Transacción: es un grupo de operaciones que se realizan juntas como un todo, como una lógica de trabajo, en caso de error de alguna de sus operaciones, no se realiza el resto de las operaciones.

Java Community Process: comunidad que controla todas las especificaciones, el desarrollo y la evolución del lenguaje Java.

VIII. BIBLIOGRAFÍA

[Arlow & Neustadt, 2008] Jim Arlow & Ila Neustadt. "UML 2 AND THE UNIFIED PROCESS Second Edition", 2008. Ed. Addison Wesley. (Practical Object – Oriented Analysis and Design)

[Bobadilla & Sancho, 2003] Jesús Bobadilla & Adela Sancho. "COMUNICACIONES Y BASES DE DATOS CON JAVA", 2003. Editorial RA-MA.

[Cuevas, 2003] Gonzalo Cuevas Agustín. "GESTIÓN DEL PROCESO SOFTWARE", 2003. Editorial Centro de Estudios Ramón Areces, S.A.

[Durán & Ruiz, 2002] A. Durán Toro, A. Ruíz Cortés, R. Corchuelo Gil y M. Toro Bornilla. "Identificación de Patrones de Reutilización de Requisitos de Sistemas de Información". Departamento de Lenguajes y Sistemas Informáticos. Universidad de Sevilla.

[Eckel, 2004] Bruce Eckel. "PIENSA EN JAVA 2ª Edición", 2004. Editorial Prentice Hall.

[Lamarca y Rodríguez, 2007] Ignacio Lamarca y José Ramón Rodríguez. "Apuntes de Metodología y Gestión de Proyectos Informáticos - TIC". UOC. Septiembre 2007.

[Larman, 2002] Craig Larman. "UML y PATRONES 2ª Edición", 2002. Ed. Prentice Hall. (Una introducción al análisis y diseño orientado a objetos y al proceso unificado).

[Montesa Andrés, 2007] José Montesa Andres. "Identificación de fases, actividades y entregables en proyectos informáticos". Universidad Politécnica de Valencia. Octubre 2007

[Polo, Gómez, Piattini y Ruiz, 2002] Macario Polo, Juan Ángel Gómez, Mario Piattini y Francisco Ruiz. "Una herramienta para la enseñanza de patrones en Ingeniería del software". Escuela Superior de Informática. Universidad de Castilla – La Mancha. 2002.

[Schildt, 2003] Herbert Schildt. "JAVA 2 – MANUAL DE REFERENCIA - 4ª Edicción", 2003. Editorial Mc Graw Hill.

[Wiley, 2003] John Wiley & Sons. "AGILE DATABASE TECHNIQUES", 2003. Editorial Ambysoft. (Effective Strategies for the Agile Software Developer).

http://www.uml.org/ → información general de UML
http://www.geocities.com/txmetsb/UML-Use-cases.htm → Casos de uso con UML
http://www.omg.org/ → Información OMG
http://es.wikipedia.org/wiki/Persistencia_de_objetos → Información general de Persistencia de objetos
http://www.agiledata.org/essays/mappingObjects.htm → http://www.virtual.unal.edu.co/cursos/sedes/manizales/4060029/lecciones/cap2-6.html → Información diagramas Entidad Relación
http://www.programacion.com/bbdd/articulo/jap_persis_jdo/ → Persistencia de objetos utilizando JDO.
http://www.alfrek.net/blog/category/programacion/designpatterns/ → diseño de patrones

http://blogs.3devnet.com/blogs/starrillo/archive/2006/03/28/70.aspx → información sobre patrones, principalmente patrón Proxy.
http://www3.uji.es/~mmarques/f47/apun/node73.html → información de transacciones.
http://www.programacion.com/bbdd/articulo/joa_persistencia/ → información acerca de motores de persistencia.
http://cayenne.apache.org/doc/jpa-guide.html → framework de persistencia cayenne, de Apache.
http://openjpa.apache.org/ → información acerca de motor de persistencia OpenJPA de Apache.
http://www.hibernate.org/ → Información acerca de Hibernate
http://ibatis.apache.org/index.html → información sobre Ibatis Data Mapper framework.
http://wiki.eclipse.org/Introduction_to_EclipseLink_(ELUG) → información acerca del ORM EclipseLink (Eclipse) – TopLink de Oracle (antes)
http://julp.sourceforge.net → documentación acerca de Java Ultra – Lite Persistence (JULP)
http://www.castor.org → información acerca del proyecto Castor y su persistencia.
http://db.apache.org/ojb/ →
http://www.onjava.com/pub/a/onjava/2003/01/08/ojb.html?page=1 → documentación acerca del ORM ObjectRelationalBridge – OJB
http://www.jpox.org/docs/1_0/configuration.html → información acerca de JPOX Java Persistent Objects.
http://sites.google.com/a/cynosuredev.com/simple-orm/Home/Documentation → documentación de este proyecto
http://www.visual-paradigm.com/VPGallery/diagrams/index.html → Información acerca de la representación de de todo tipo de diagramas UML

ANEXO I. TIPOS DE FRAMEWORKS DE PERSISTENCIA

BuzzSQL

Qué es
BuzzSQL es una simple API Java para acceder a una base de datos relacional. Funciona como una fina capa sobre JDBC. Posee un sistema automático de instalación y conexión de la base de datos con pool de conexiones configurable, con logging, etc. Está orientado para aplicaciones independientes (stand-alone) en las cuales se necesita rápidas consultas o actualizaciones a la base de datos.

Qué no es
BuzzSQL no es un ORM (Object-relational mapping). No examina la estructura de la base de datos ni genera clases. Tampoco escribe SQL de forma automática, es necesario entender la sintaxis SQL y el conocimiento de cómo hacer las consultas a la base de datos.

Objetivos
El objetivo de BuzzSQL es conseguir un medio perfilado entre la utilización de conexiones JDBC convencionales para acceso a base de datos y los más complejas librerías de mapeo objeto – relación tal como Hibernate, Torque o Cayenne.
BuzzSQL fue escrito por completo como una necesidad del autor para acceder mediante una librería a una base de datos Java que es menos compleja que la gran mayoría de librerías ORM, pero simplifica las los estados de JDBC y provee pool de conexiones automáticas y utiliza configuración utilizando ficheros de propiedades. Está diseñado para trabajar en servidor de aplicaciones J2EE y aplicaciones J2SE independientes, autónomas o stand-alone.
La utilización más habitual de BuzzSQL es para desarrolladores que buscan una rápida y eficiente capa de acceso a datos swing sobrecarga y cabeceras de una herramienta OML. No se recomienda utilizar BuzzSQL si buscas ingeniería inversa para la base de datos, generación de script SQL, datos orientados a objetos, alto rendimiento de caché o multitud de ficheros de configuración XML.

Arquitectura
La estructura del objeto de BuzzSQL es limpia y directa, construida una fácil librería para aprender a utilizar por desarrolladores novatos de Java. La mayoría de objetos son modelados de acuerdo con el estándar SQL, Select para consultas, Insert para inserciones, Update para actualizaciones, etc.
Buzzsql es una capa fina por encima de JDBC, y además provee acceso para todos los objetos por debajo de JDBC. Conexiones a la base de datos son siempre obtenidas hacia un contexto de nombrado JNDI. Como resultado, integramos casi los servidores de aplicaciones. En efecto, en la mayoría de entornos de aplicaciones de servidor no son necesarios configuraciones de ficheros adicionales. Cuando un contexto de nombrado no es viable como es común en una aplicación independiente J2SE, BuzSQL crea una automáticamente y carga en con el datasource de acuerdo a tu fichero de confirmación.

Java versión
Se requiere J2SE 5.0 o una más actual. No funciona con versiones anteriores.

Ejemplo de utilización de dicha librería BuzzSQL.

```
public class Example1
{
	public static void main(String args[]) throws SQLException
	{
		new Example1();
	}

	public Example1() throws SQLException
	{
		// Release info usage
		MyReleaseInfo releaseInfo = MyBuzzSQL.getReleaseInfo();
		System.out.println(releaseInfo.getWelcome());

		System.out.println();
		System.out.println("---------------------------------");
		// simple delete
		MyDelete delete1 = new MyDelete("delete from example1.table_example1");
		delete1.execute();
		delete1.close();
		System.out.println("borrado finalizado");
		System.out.println("---------------------------------");

		// insert with retrieval of auto-increment primary key
		MyInsert insert1 = new MyInsert("insert into example1.table_example1(col_string, col_int) values
(?,?)");
		insert1.setArgs("jorge", 1234567);
		insert1.execute();
		int pk1 = insert1.getGeneratedKey();
		System.out.println("clave primaria --> " + pk1);
		insert1.close();
		System.out.println("insertado");
		System.out.println("---------------------------------");

		// insert with retrieval of auto-increment primary key
		insert1 = new MyInsert("insert into example1.table_example1(col_string, col_int) values (?,?)");
		insert1.setArgs("ana", 475757);
		insert1.execute();
		pk1 = insert1.getGeneratedKey();
		System.out.println("clave primaria --> " + pk1);
		insert1.close();
		System.out.println("insertado");
		System.out.println("---------------------------------");

		// simple select with result set iteration
		Select query1 = new Select().setSQL("select col_pk, col_string, col_int from
example1.table_example1");
			query1.execute();
			while (query1.next()) {
				System.out.println(query1.getLine());
			}
		System.out.println("Select finalizado");
		System.out.println("---------------------------------");
		query1.close();



		// update with setArgs method chaining
		MyUpdate update1 = new MyUpdate("update example1.table_example1 set col_int = ? where
col_string = ?").setArgs(98765, "jorge");
		update1.execute();
		update1.close();
		System.out.println("update finalizado");
		System.out.println("---------------------------------");


		// simple select with result set iteration
		query1 = new Select().setSQL("select col_pk, col_string, col_int from example1.table_example1");
			query1.execute();
			while (query1.next()) {
```

```
                                System.out.println(query1.getLine());
                        }
                query1.close();
                System.out.println("Select finalizado");
                System.out.println("--------------------------------");

                // simple select with result set iteration
                query1 = new Select().setSQL("select * from example1.table_example1");
                        query1.execute();
                        while (query1.next()) {
                                System.out.println(query1.getLine());
                        }
                query1.close();
                System.out.println("Select finalizado");
                System.out.println("--------------------------------");
        }
```

El fichero de configuración: BuzzSQL.properties

```
<initial-context>
<context name="java:comp">
        <context name="env">
                <context name="jdbc">
                <reference name="example"  className="javax.sql.DataSource"
        factoryClassName="org.shiftone.ooc.factory.PooledDataSourceObjectFactory">
                <refAddr name="driver"       value="com.mysql.jdbc.Driver" />
                <refAddr name="url" value="jdbc:mysql://localhost:3306/example1" /
        <refAddr name="user" value="root" />
                <refAddr name="password" value="root" />
</reference>
                </context>
        </context>
</context>
</initial-context>
```

Cayenne (Objeto Relacionales, la persistencia y el almacenamiento en caché de Java*)*

Apache Cayenne es un código abierto de persistencia con licencia Apache, proporcionan mapeado de objeto-relacional (ORM) y servicios de Remoting. Con gran riqueza y potentes características, Cayenne puede abordar una amplia gama de necesidades de persistencia. Se une a la perfección con uno o más esquemas de bases de datos directamente con objetos Java, la gestión atómica, rollbacks, generación de SQL, sucesión de secuencias y mucho más. Cayenne permite la persistencia de objetos a distancia, incluso a cliente a través de Web Services. También permite la serialización XML nativo.
Cayenne está diseñador par ser fácil de utilizar, sin sacrificar la flexibilidad o el diseño. Con ese fin, apoya Cayenne bases de datos con ingeniería inversa y la generación, así como una velocidad basada en la generación de motor de clase. Todas estas funciones pueden ser controladas directamente a través de la CayenneMOdeler, uno funcional y completa herramienta GUI. No es necesaria una configuración encriptada en xml. Todo un esquema de la base de datos puede ser asignado directamente a los objetos java en cuestión de minutos, todo desde la comodidad de la interfaz gráfica basada en CayenneModeler.

Cayenne apoya muchas otras características, incluyendo el almacenamiento en caché, una completa sintaxis de consulta de objetos, relación pre búsqueda, inherencia de objetos, auto-detección de la base de datos y persistencia genérica de objetos. Lo que es más importante es la escalabilidad que aporta Cayenne para cualquier tamaño del proyecto. Con una gran madurez, el 100% de código abierto, una comunidad de usuario potente y un

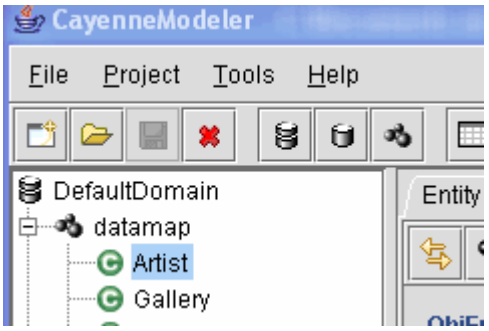
historial sólido de alto rendimiento en grandes entornos. Así pues, Cayenne es una excepcional opción para los servicios de búsqueda.

El equipo de Cayenne está en proceso de construcción en: http://cayenne.apache.org/doc/jpa-guide.html

Actualmente se encuentra publicada la versión "Cayenne 3.0M4 Released"

Cayenne Modeler

Cayenne se distribuye con CayenneModeler, una completa herramienta de mapeado GUI que apoya la ingeniería inversa de esquemas RDBMS (Sistema Administrador de Bases de datos relacionales), en colaboración con las asignaciones de bases de datos y generación de código fuente de Java para la persistencia de objetos.



La persistencia de clases Java se genera y sincroniza con el capeado utilizando el Modelar o, alternativamente con una tarea de Apache Ant. Un esquema de la base de datos SQL puede ser generado a partir de la Modeler y también con simples llamadas a las API.

Más información en: http://cayenne.apache.org/

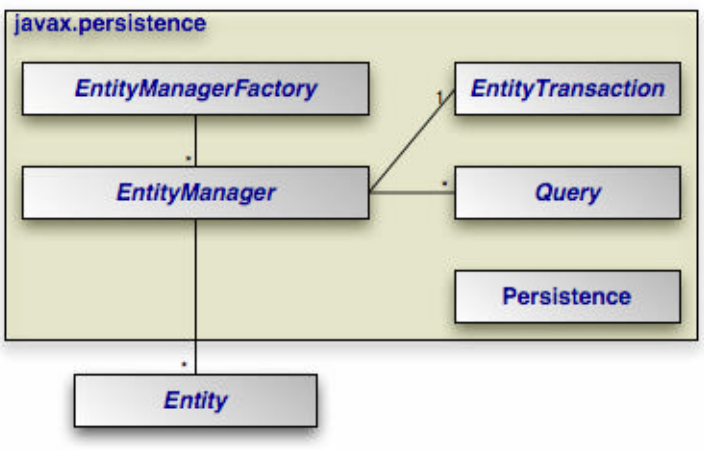
Apache OpenJPA

OpenJPA Apache es un proyecto de persistencia Java de la fundación Apache Software Foundation. Es una gran característica de la implementación de la parte de la persistencia de EJB 3.0. También conocida como Persistencia de Java API (APP) y está disponible bajo los términos de la Licencia Apache Software.

OpenJPA puede ser utilizado como un simple POJO de la capa de persistencia, o puede ser integrado en cualquier contenedor ligero EJB 3.0 y muchos ligeros marcos de trabajo.

JPA es una especificación de Sun Microsysems para la persistencia de objetos Java a cualquier base de datos relacional. Esta API fue desarrollada para la plataforma JEE e incluida en el estándar de EJB 3.0, formando parte de la JAVA Specification Request JSR 220.

Para su utilización, esta API necesita o requiere J2SE 1.5 (también conocida como Java 5) o superior, ya que hace uso intensivo de las nuevas características de lenguaje



Java, como las anotaciones y los genéricos.

Arquitectura

Varias de las interfaces del diagrama anterior son solo necesarias para su utilización fuera de un servidor de aplicaciones que soporte EJB 3, como es el caso del EntityManagerFactory que es ampliamente usado en desarrollo de aplicaciones de escritorio. En un servidor de aplicaciones, una instancia de EntityManager típicamente suele ser inyectada, haciendo así innecesario el uso de un EntityManagerFactory. Por otra parte, las transacciones dentro de un servidor de aplicaciones se controlan mediante un mecanismo estándar de controles de, por lo tanto la interfaz EntityTransaction también no es utilizada en este ambiente.

- **Persistence:** La clase javax.persistence.Persistence contiene métodos estáticos de ayuda para obtener una instancia de EntityManagerFactory de una forma independiente al vendedor de la implementación de JPA.
- **EntityManagerFactory:** La clase javax.persistence.EntityManagerFactory nos ayuda a crear objetos de EntityManager utilizando el patrón de diseño del Factory (fábrica). **EntityManager:** La clase javax.persistence.EntityManager es la interfaz principal de JPA utilizada para la persistencia de las aplicaciones. **Cada EntityManager puede realizar operaciones CRUD** (Create, Read, Update, Delete) sobre un conjunto de objetos persistentes.
- **Entity:** La clase javax.persistence.Entity es una anotación Java que se coloca a nivel de clases Java serializables y que cada objeto de una de estas clases anotadas representa un registro de una base de datos.
- **EntityTransaction:** Cada instancia de EntityManager tiene una relación de uno a uno con una instancia de javax.persistence.EntityTransaction, permite operaciones sobre datos persistentes de manera que agrupados formen una unidad de trabajo transaccional, en el que todo el grupo sincroniza su estado de persistencia en la base de datos o todos fallan en el intento, en caso de fallo, la base de datos quedará con su estado original. Maneja el concepto de todos o ninguno para mantener la integridad de los datos.
- **Query:** La interface javax.persistence.Query está implementada por cada vendedor de JPA para encontrar objetos persistentes manejando cierto criterio de búsqueda. JPA estandariza el soporte para consultas utilizando **Java Persistence Query Language (JPQL) y Structured Query Language (SQL).** Podemos obtener una instancia de Query desde una instancia de un EntityManager.

A continuación mostramos con un ejemplo como las interfaces de JPA interactúan para ejecutar una consulta con JPQL y actualizar objetos persistentes. En este ejemplo estamos utilizando una ejecución fuera de un contenedor de EJB 3.0:

```
// obtener una instancia de EntityManagerFactory usando la clase
// Persistence
// Como consejo debemos guardar la instancia de este factory para
// que no vuelva a ser creado con el objeto Persistence debido a que
// tiene un costo considerable de construcción
EntityManagerFactory factory = Persistence.createEntityManagerFactory("MyPersistenceUnitName");
// obtener una instancia de EntityManager del factory
EntityManager em = factory.createEntityManager();
// Inicio de transacción
em.getTransaction().begin();
```

```
// Consulta para obtener todos los empleados que trabajan en una
// división de trabajo especificada y tienen un promedio de 40
// horas trabajadas a la semana
Query query = em.createQuery("SELECT e " +
" FROM Employee e " +
" WHERE e.division.name = 'Research' " +
" AND e.avgHours > 40");
// obtener una lista de resultados
List results = query.getResultList ();
// establecer un monto de salario a todos los empleados
for (Object res : results) {
Employee emp = (Employee) res;
emp.setSalary(emp.getSalary() * 1.1);
}
// Comprometer todos los cambios que han sido detectados dentro de
// la transacción en el que se actualizarán todos los objetos
// persistentes que hayamos modificado em.getTransaction().commit();
// liberar los recursos
em.close();
```

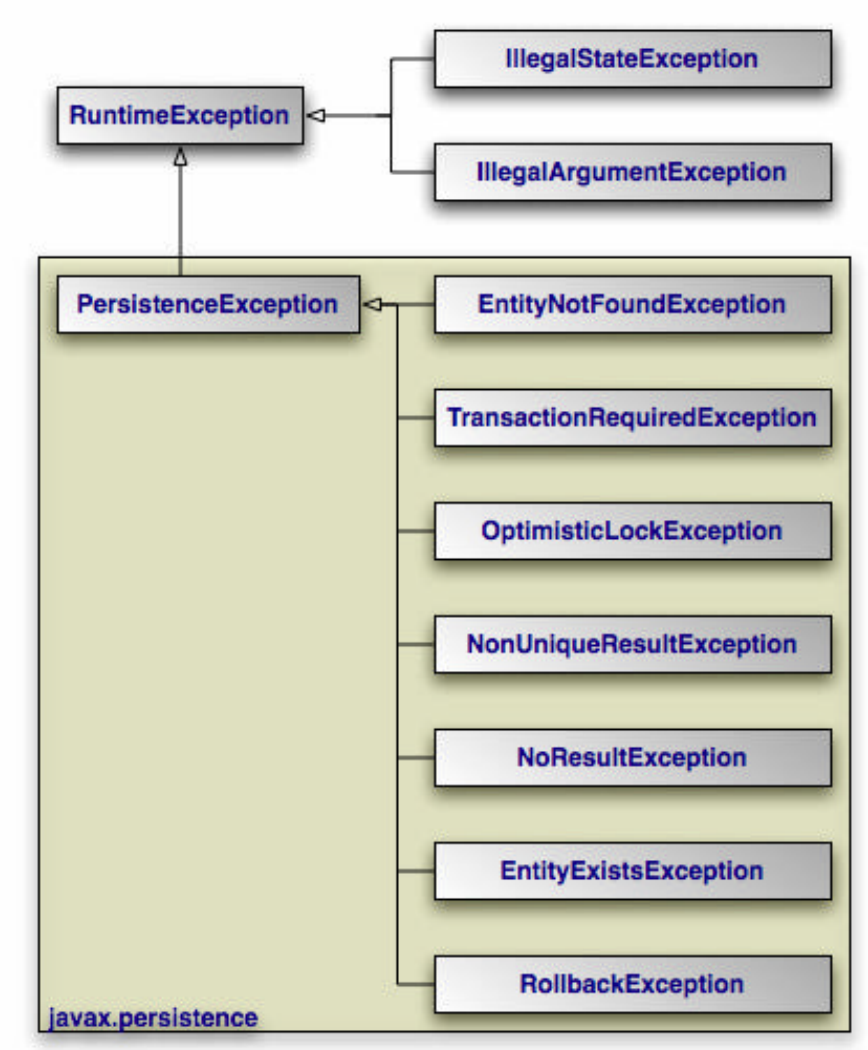
En el siguiente ejemplo realizaremos lo mismo que el ejemplo anterior pero suponiendo que estamos desarrollando una aplicación que reside en un contenedor de EJB 3.0:

```
// Consulta para obtener todos los empleados que trabajan en una
// división de trabajo especificada y tienen un promedio de 40
// horas trabajadas a la semana
Query query = em.createQuery("SELECT e " +
" FROM Employee e " +
" WHERE e.division.name = 'Research' " +
" AND e.avgHours > 40");
// obtener una lista de resultados
List results = query.getResultList ();
// establecer un monto de salario a todos los empleados
for (Object res : results) {
Employee emp = (Employee) res;
emp.setSalary(emp.getSalary() * 1.1);
}
```

Como observamos en los dos ejemplos la gran diferencia entre utilizar JPA dentro y fuera de un contenedor de EJB 3.0, es que en las aplicaciones que residen en un contenedor de EJB 3.0, el contenedor es quien se encarga de la creación de la instancia del EntityManager utilizando el concepto de inyección de dependencias y contexto de persistencia, y la parte transaccional es transparente al desarrollador debido a que es el contenedor de EJB 3.0 quien se encarga del manejo de las transacciones.

Excepciones

Excepciones utilizadas en JPA.



Beans de Entidad

Anotaciones de EJB 3.0 para definir clases de entidad persistentes.

Mapeo con anotaciones EJB 3.0 / JPA

Las entidades de JPA no son más que simples POJOs (Plain Old Java Objects) que tienen definidos sobre su clase, propiedad o métodos unas anotaciones especiales de EJB3.0. Las anotaciones de JPA se clasifican en dos categorías:

- **Anotaciones de mapeo lógico** que permiten describir modelo de objeto, asociaciones de clase, etc.
- **Anotaciones de mapeo físico** que describen esquemas físicos de base de datos, tablas, columnas, índices, etc.

JPA reconoce dos tipos de clases persistentes: las clases entidad y las clases embebidas (embebidas).
Las anotaciones JPA, conocidas también como anotaciones EJB 3.0, se encuentran en el paquete javax.persistence.*. Muchos IDEs que soportan a JDK5 como Eclipse, Netbeans,

IntelliJ IDEA, poseen herramientas y plugins para generar clases de entidad con anotaciones de JPA a partir de un esquema de base de datos.

Declaración de un Bean con Entidad.

Para declarar un bean de entidad agregamos la anotación @Entity a la clase que va a ser persistente, hay tener en cuenta que una clase persistente además de tener esta anotación, debe implementar la interfaz Serializable.
Para declarar el identificador de la clase que representará la clave primaria en la tabla de base de datos, lo hacemos utilizando la anotación @Id.

Ejemplo:

```
@Entity
public class Flight implements Serializable {
Long id;
@Id
public Long getId() { return id; }
public void setId(Long id) { this.id = id; }
}
```

Dependiendo de donde declaremos la anotación, sea en propiedades o métodos, el acceso para el motor de persistencia JPA será correspondientemente accediendo a una propiedad o a un método. Lo más aconsejable es que las anotaciones se las declare a nivel de métodos getters ya que son los más visitados. Debemos evitar tener mezclado anotaciones tanto en propiedades como en métodos.

Declaración de Tablas.
Para definir los datos de una tabla de base de datos, utilizamos la anotación @Table a nivel de la declaración de la clase, esta anotación nos permite definir el nombre de tabla con la que se está mapeando la clase, el esquema, el catálogo, constraints de unicidad. Si no se utiliza esta anotación y se utiliza únicamente la anotación Entity, el nombre de la tabla corresponderá al nombre de la clase.

```
@Entity
@Table(name="tbl_sky")
public class Sky implements Serializable {
```

La anotación Table contiene los atributos de catalogo y schema, en caso de que necesitemos definirlos. Usted también puede definir constraints de unicidad utilizando la anotación @UniqueConstraint en conjunción con la anotación @Table

```
@Table(name="tbl_sky",
uniqueConstraints = {@UniqueConstraint(columnNames={"month", "day"})}
)
```

Más información y documentación: http://openjpa.apache.org/

IBATIS

IBATIS es un framework de código abierto basado en capas desarrollado por Apache Software Foundation, que se ocupa de la capa de Persistencia (se sitúa entre la lógica de Negocio y la capa de la Base de Datos). Puede ser implementado en Java y .NET (también existe un port para Ruby on Rails llamado RBatis).

Características

Es posible subdividir la capa de Persistencia en tres subcapas: **Abstracción, capa de Framework de Persistencia y la capa de Driver.**

- **La capa de Abstracción:** será el interfaz con la capa de la lógica de negocio, haciendo las veces de "facade" entre la aplicación y la persistencia. Se implementa de forma general mediante el patrón **Data Access Object (DAO)**, y particularmente en iBATIS se implementa utilizando su framework DAO **(ibatis-dao.jar).** Se configura mediante el fichero **dao.xml**

- **La capa de Framework de Persistencia:** será el interfaz con el gestor de Base de Datos ocupándose de la gestión de los datos mediante un API. Normalmente en Java se utiliza JDBC, IBATIS utiliza su framework **SQL-MAP (ibatis-sqlmap.jar).** Se configura mediante un fichero XML de configuración, **sql-map-config.xml.** Además cada **objeto de modelo**, que representa al objeto en la aplicación, se relaciona con un fichero del tipo **sqlMap.xml**, que contiene sus sentencias SQL. Por ejemplo, un objeto Java Persona con un objeto XML *persona.xml.*

- La capa de **Driver:** se ocupa de la comunicación con la propia Base de Datos utilizando un Driver específico para la misma.

Toda implementación de iBATIS incluye los siguientes componentes: Data Mapper y Data Access Object.

- **Data Mapper**: proporciona una forma sencilla de interacción de datos entre los objetos Java, .NET y bases de datos relacionales.
- **Data Access Object**: abstracción que oculta la persistencia de objetos en la aplicación y proporciona un API de acceso a datos al resto de la aplicación

El framework Ibatis Data Mapper facilita el uso de la base de datos para aplicaciones Java y .NET. Se caracteriza por su simplicidad, toda su potencia en SQL y en procedimientos almacenados a su alcance.
Los componentes básicos de Ibatis son: "iBatis datos Mapper", "iBatis Data Access Objects" y "PetStore Demo".

El Ibatis SQL Maps proporciona una gran relación con los objetos y la base de datos tanto en Java como en .Net. Se utiliza toda la potencia real de SQL sin utilizar ni una sola línea de JDBC. El SQL Maps ayuda a reducir significativamente la cantidad de código que normalmente se necesita para acceder a una base de datos relacional. Se utiliza un simple descriptor XML combinándolo con javaBeans y SQL. El Ibatis data Mapper permite adaptarse a cualquier modelo de base de datos y es muy tolerante a cambios de diseños e incluso malos diseños, esto es logrado sin la generación de código.

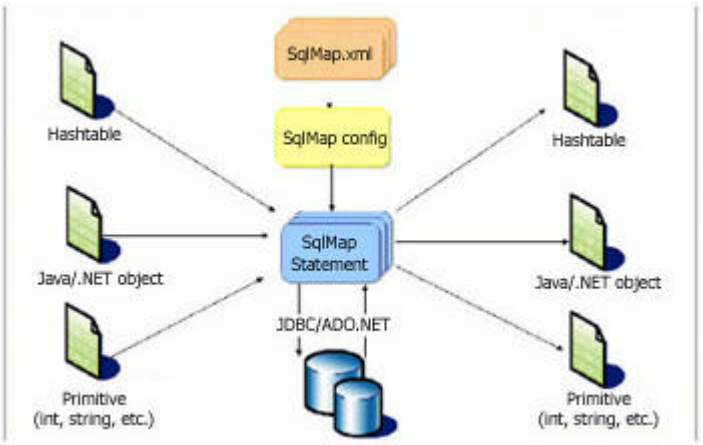
iBatis Data Access Objects. Es una capa de abstracción que oculta detalles de la solución de persistencia y proporciona una API común para lanzar la aplicación. Cuando desarrollamos aplicaciones sólidas en un potente lenguaje de programación como Java, a menudo se aislan los detalles de la implementación de persistencia en una API común. El marco Data Access Objects se incluye como parte de la capa de base de datos ibatis, que incluye SQL Maps. Aunque vayan juntos, el marco DAO es completamente independiente y puede ser utilizado sin SQL Maps.

Ibatis PetStore

PetStore es una web totalmente funcional basada en la capa de persistencia de productos de código abierto Ibatis. jPetStore es un excelente ejemplo de cómo el framework iBatis puede ser implementado en una típica aplicación web .NET y J2EE.

La versión más actual de IBatis es la 2.3.4, con JDK 1.5 obligatorio para su funcionamiento. En cuanto al ejemplo de aplicación PetStore, está la versión 5.0.

La página oficial para más información:



http://ibatis.apache.org/index.html

Ibatis no pretende sustituir herramientas Objeto-Relacionales Mapping (ORM) tales como Hibernate, OJB, TopLink para nombrar unos pocos. Más bien se trata de un bajo nivel de marco que le permite manejar su código SQL y el mapa-objeto Java. Una vez que el mapa de su capa de persistencia a su modelo de objetos, lo tiene todo configurado. No es necesario buscar DataSource, conseguir conexiones, crear declaraciones preparadas, analizar resultados o incluso la caché de los resultados - iBatis lo hace todo por ti. En cuanto a las capas, crea un iBatis PreparedStatement, establece los parámetros (si procede), ejecuta las declaraciones y construye un mapa o un objeto JavaBean de los resultados.

Ejemplo Completo de iBatis

Person.java

```
package examples.domain;
//imports omitidos….
public class Person {
        private int id;
        private String firstName;
        private String lastName;
        private Date birthDate;
        private double weightInKilograms;
        private double heightInMeters;

public int getId () {
        return id;
}
public void setId (int id) {
        this.id = id;
}
//…asumimos que aquí tendríamos otros métodos get y set.…

}
```

Utilizaremos la siguiente tabla, que podría ser apropiada para una relación tabla-por-clase.

Person.sql

```
CREATE TABLE PERSON(
PER_ID NUMBER (5, 0) NOT NULL,
PER_FIRST_NAME VARCHAR (40) NOT NULL,
PER_LAST_NAME VARCHAR (40) NOT NULL,
PER_BIRTH_DATE DATETIME ,
PER_WEIGHT_KG NUMBER (4, 2) NOT NULL,
PER_HEIGHT_M NUMBER (4, 2) NOT NULL,
PRIMARY KEY (PER_ID))
```

El fichero de configuración SQL Maps.

El fichero de configuración es un fichero XML. Dentro de el configuraremos ciertas propiedades, el DataSource JDBC y los mapeos SQL que utilizaremos en nuestra aplicación. Este fichero te ofrece un lugar central donde configurar tu DataSource. El marco de trabajo puede manejar varias implementaciones de DataSources, dentro de los que se incluyen iBATIS están SimpleDataSource, Jakarta DBCP (Commons), y cualquier DataSource que se pueda obtener a través de un contexto JNDI (p.e. Un DataSource configurado dentro de un servidor de aplicaciones).

SqlMapConfigExample.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMapConfig
PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

<!-- Asegúrate siempre de usar las cabeceras XML correcta como la de arriba! -->

<sqlMapConfig>
<!--
El fichero de properties especificado aquí es el lugar donde poner las propiedades (name=value)
que se usarán después en este fichero de configuración donde veamos elementos de configuración
como por ejemplo ”${driver}”. El fichero suele ser relativo al classpath y es opcional.-->

<properties resource="examples/sqlmap/maps/SqlMapConfigExample.properties" />

<!-- Estas propiedades controlan los detalles de configuración de SqlMap, principalmente las
relacionadas con la gestión de transacciones. Todas son opcionales -->

<settings
cacheModelsEnabled="true"
enhancementEnabled="true"
lazyLoadingEnabled="true"
maxRequests="32"
maxSessions="10"
maxTransactions="5"
useStatementNamespaces="false"
/>

<!-- typeAlias permite usar un nombre corto en referencia a un nombre cualificado de una clase–>
<typeAlias alias="order" type="testdomain.Order"/>


<!-- Configura un DataSource que podrá ser usado con SQL Map usando SimpleDataSource.. -->

<transactionManager type="JDBC" >
<dataSource type="SIMPLE">
<property name="JDBC.Driver" value="${driver}"/>
<property name="JDBC.ConnectionURL" value="${url}"/>
<property name="JDBC.Username" value="${username}"/>
<property name="JDBC.Password" value="${password}"/>
</dataSource>
</transactionManager>

<!—Identificamos todos los ficheros de mapeos XML usados en SQL Map para cargar los mapeos SQL.
Los paths son relativos al classpath. Por ahora, solo tenemos uno... -->
<sqlMap resource="examples/sqlmap/maps/Person.xml" />
</sqlMapConfig>
```

SqlMapConfigExample.properties

```
# Este es un ejemplo de fichero de properties que simplifica la automatización de la configuración
#del fichero de configuración de SQL Maps (ej. A través de Ant o de herramientas para la
#integracióncontinua para diferentes entornos... etc.).
#Estos valores pueden ser usados en cualquier valor de propiedad en el fichero de arriba (ej.
#“${driver}”.
#El uso del fichero de propiedades es completamente opcional.
```

```
driver=oracle.jdbc.driver.OracleDriver
url=jdbc:oracle:thin:@localhost:1521:oracle1
username=jsmith
```

Los ficheros SQL Map

Ahora que tenemos configurado un DataSource y listo el fichero de configuración central, necesitaremos proporcionar al fichero de SQL Map nuestro código SQL y los mapeos para cada uno de los objetos parámetro y de los objetos resultado (entradas y salidas respectivamente).
Continuando con nuestro ejemplo de arriba, vamos a construir un fichero SQL Map para la clase Person y la tabla PERSON. Empezaremos con la estructura general de un documento SQL, y una sencilla sentencia SQL:

Person.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap namespace="Person">
<select id="getPerson" resultClass="examples.domain.Person">
        SELECT PER_ID as id,
        PER_FIRST_NAME as firstName,
        PER_LAST_NAME as lastName,
        PER_BIRTH_DATE as birthDate,
        PER_WEIGHT_KG as weightInKilograms,
        PER_HEIGHT_M as heightInMeters
        FROM PERSON WHERE PER_ID = #value#
</select>
</sqlMap>
```

El ejemplo de arriba muestra la forma más sencilla de un mapeo SQL. Usa una característica del marco de trabajo SQL Maps que permiten mapear de forma automática columnas de un ResultSet a propiedades de un JavaBean (o keys de un objeto java.util.Map, etc.) basándose en el encaje de los nombres, es decir
que encajen los nombres de los campos del ResultSet con los de los atributos de JavaBean.
El símbolo #value# es un parámetro de entrada. Más específicamente, el uso de “value” implica que estamos usando un recubrimiento de un tipo primitivo (ej. java.lang.Integer; pero no estamos limitados solo a este).

Completamos nuestro mapeo SQL para la clase Person con mas mapeos que proporciona un conjunto de sentencias para acceder y modificar los datos en la base de datos, es decir operaciones INSERT, UPDATE Y DELETE.

Person.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap namespace="Person">
<!-- Usa un recubrimiento de un tipo primitivo (ej. java.lang.Integer) como parámetro y permite
que los resultados sean auto-mapeados a las propiedades del JavaBean Person -->
<select id="getPerson" parameterClass=”int” resultClass="examples.domain.Person">
SELECT
PER_ID as id,
PER_FIRST_NAME as firstName,
```

```
PER_LAST_NAME as lastName,
PER_BIRTH_DATE as birthDate,
PER_WEIGHT_KG as weightInKilograms,
PER_HEIGHT_M as heightInMeters
FROM PERSON
WHERE PER_ID = #value#
</select>

<!-- Usa las propiedades del JavaBean Person como parámetro para insertar. Cada uno de los
parámetros entre los símbolos #almuadilla# es una propiedad del JavaBean. -->
<insert id="insertPerson" parameterClass="examples.domain.Person">
INSERT INTO
PERSON (PER_ID, PER_FIRST_NAME, PER_LAST_NAME,
PER_BIRTH_DATE, PER_WEIGHT_KG, PER_HEIGHT_M)
VALUES (#id#, #firstName#, #lastName#,
#birthDate#, #weightInKilograms#, #heightInMeters#)
</insert>

<!-- Usa las propiedades del JavaBean person como parámetro para actualizar. Cada uno de los
parámetros entre los símbolos #almuadilla# es una propiedad del JavaBean. -->
<update id="updatePerson" parameterClass="examples.domain.Person">
UPDATE PERSON
SET PER_FIRST_NAME = #firstName#,
PER_LAST_NAME = #lastName#, PER_BIRTH_DATE = #birthDate#,
PER_WEIGHT_KG = #weightInKilograms#,
PER_HEIGHT_M = #heightInMeters#
WHERE PER_ID = #id#
</update>

<!-- Usa las propiedades del JavaBean person como parámetro para borrar. Cada uno de los
parámetros entre los símbolos #almuadilla# es una propiedad del JavaBean. -->
<delete id="deletePerson" parameterClass="examples.domain.Person">
DELETE PERSON
WHERE PER_ID = #id#
</delete>
</sqlMap>
```

Programando con el marco de trabajo SQL Map

Ahora que tenemos todo configurado y mapeado, todo lo que necesitamos es programar nuestra aplicación
Java. El primer paso es configurar SQL Map. Es simplemente una forma de de cargar nuestra configuración
del fichero XML de SQL Map que creamos antes. Para cargar el fichero XML, haremos uso de la clase
Resources incluida en el marco de trabajo.

```
String resource = “com/ibatis/example/sqlMap-config.xml”;
Reader reader = Resources.getResourceAsReader (resource);
SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMapClient(reader);
```

Un objeto de la clase SqlMapClient es thread safe y será un servicio de larga duración . Solo es necesario instanciarlo/configurarlo una vez para todo el ciclo de vida de una aplicación. Esto lo convierte en un buen candidato para ser un miembro estático de una clase base (ej. una clase base DAO), o si prefieres mantenerlo configurado de forma mas centralizada y disponible de forma más global, podrías crear una clase wrapper para mayor comodidad. Aquí hay un ejemplo de una clase que podría servir para este propósito:

```
public MyAppSqlConfig {
```

```
private static final SqlMapClient sqlMap;
static {
try {
String resource = “com/ibatis/example/sqlMap-config.xml”;
Reader reader = Resources.getResourceAsReader (resource);
sqlMap = SqlMapClientBuilder.buildSqlMapClient(reader);
} catch (Exception e) {

// Si hay un error en este punto, no importa cual sea. Será un error irrecuperable del cual
// nos interesará solo estar informados.
// Deberás registrar el error y reenviar la excepción de forma que se te notifique el
// problema de forma inmediata.
e.printStackTrace();
throw new RuntimeException (“Error initializing MyAppSqlConfig class. Cause: ” + e);
}
}
public static SqlMapClient getSqlMapInstance () {
return sqlMap;
}
}
```

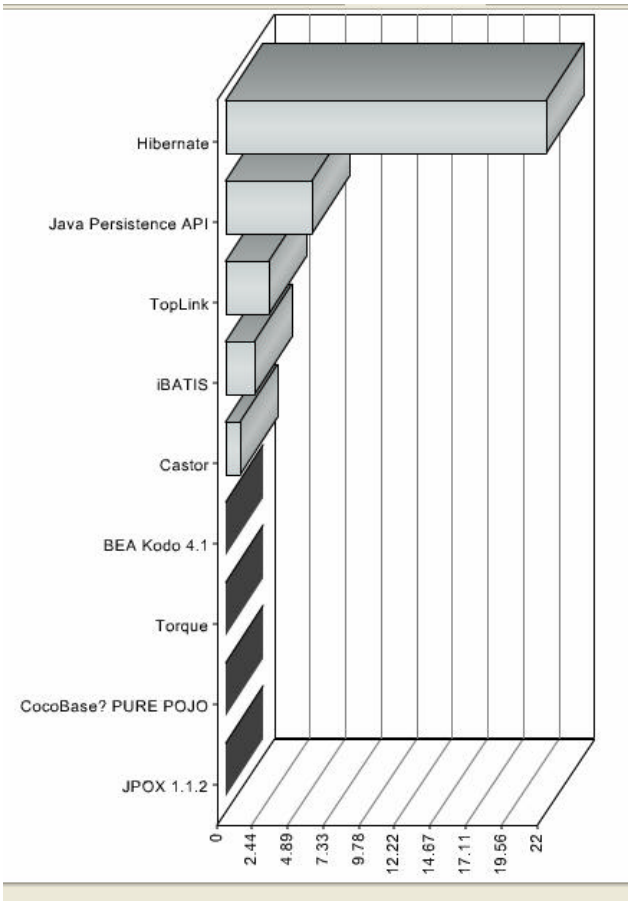
Leyendo Objetos de la Base de Datos

Ahora que la instancia de SqlMap está inicializada y es accesible de forma sencilla, podemos hacer uso de
ella. La usaremos primero para obtener un objeto Person de la base de datos. (Para este ejemplo, asumiremos que hay 10 registros en la tabla PERSON con rangos de PER_ID del 1 al 10).
Para obtener un objeto Person de la base de datos, simplemente necesitamos la instancia de SqlMap, el
nombre de la sentencia a ejecutar y un Person ID. Carguemos por ejemplo el objeto Persona cuyo “id” es el número 7.

```
SqlMapClient sqlMap =
MyAppSqlMapConfig.getSqlMapInstance(); //
Como se codificó arriba
Integer personPk = new Integer(7);
Person person = (Person) sqlMap.queryForObject
(“getPerson”, personPk);
```

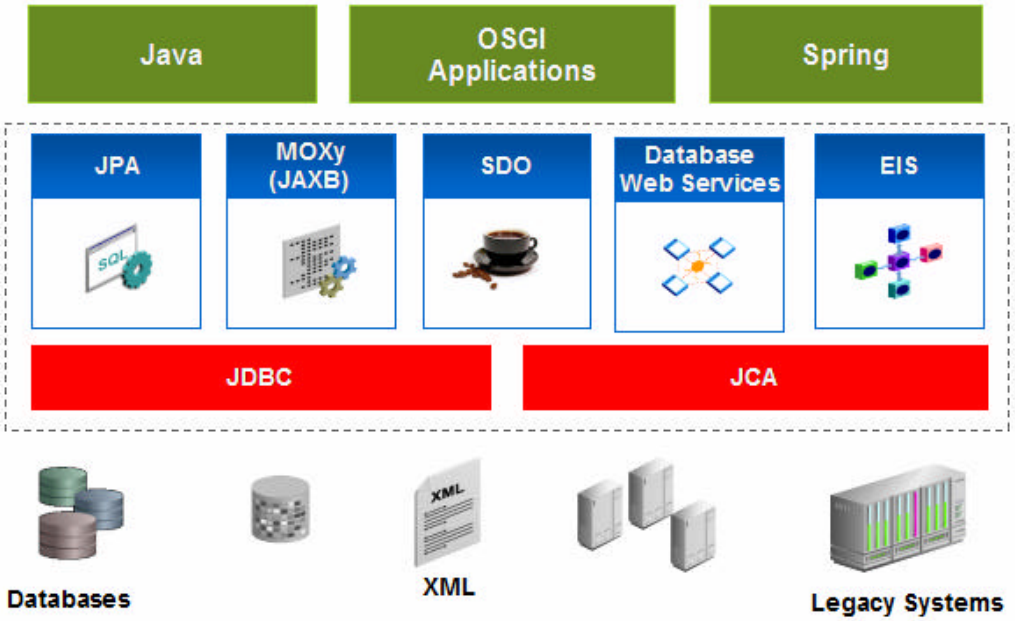
TopLink

TopLinks fue originalmente desarrollado por desarrolladores de Smaltalk en el año 1990. El “TOP” en TopLink es un acrónico para la “The Object People” y el nombre fue originalmente denominado como “TopLink”. De 1996 – 1998 la versión de Java del producto fue añadida a la línea de producto, nombrada por TopLink para Java. Después de la adquisición de “The Object People” en Abril 2000 por parte de Bea Systems y WebGain, el producto Toplink pasó a ser propiedad de WebGain.



En 2002, TopLink fue adquirido por "Oracle Corporation" y continuó para ser desarrollado como miembro de la familia de productos "Oracle Fusion Middleware". TopLink continuó creciendo ganando múltiples premios incluidos: Java Pro Reader, a la mejor herramienta o driver de acceso a datos (Julio 2003), también como mejor herramienta de acceso a datos en JavaWorld 2003 y votado como el cuarto mejor arquitectura de persistencia de Java por "Java Developers Journal" en 2004.

TopLink es un paquete ORM para desarrolladores Java. Provee un potente y flexible framework para almacenar objetos java en una base de datos relacional o para convertir objetos java a documentos XML.
TopLink Essentials es la Implementación de Referencia de EJB 3.0 JPA y la edición de la comunidad de código abierto del producto de TopLink Oracle. TopLink Essentials es una versión límite del producto propietario. Por ejemplo, TopLink Essentials no provee



sincronización de caché entre aplicación de clusters, consultas a caché, etc.

En 2006, Oracle donó el código fuente del producto TopLink y recursos de desarrollo para el código abierto Sun Microsystems GlashFish proyect. Este proyecto fue nombrado "TopLinks Essentials" y fue Java EE EJB 3.0 JPA implementación.

En 2007, Oracle donó el código fuente del producto TopLink y sus recursos al proyecto EclipseLink Fundación Eclipse.

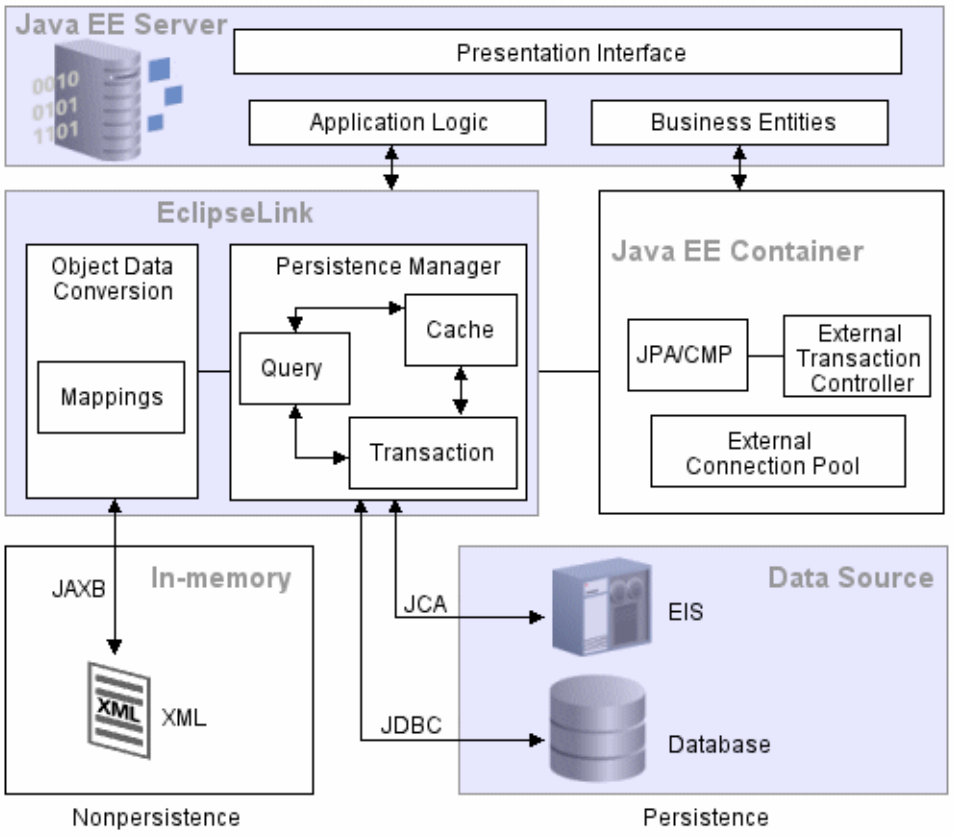
En marzo de 2008, la fundación Eclipse, anunció que Sun Microsystems ha seleccionado al proyecto EclipseLink como la implmentacion de referencia para JPA 2.0, el comité JSR 317 sucedió este proyecto por TopLink Essentials.

Así pues, la versión actual de TopLink es denominada EclipseLink.

Características
Aunque es conocido comúnmente como una herramienta ORM (Object Relational Mapping), TopLink tiene otras características claves incluidas:

- Framework rico en consultas que soporta expresiones para framework orientado a objetos, QBE (Quero By Example), EJB QL, SQL, y procedimientos almacenados.
- Framework de transacciones a nivel de objetos.
- Motor de objetos de identidad con avanzada caché.
- Completa asignación o direccionamiento del mapeo relacional.
- Mapeo de objetos a XML, y suporte JAXB.
- Soporte para bases de datos no relacionales EIS/JCA.
- Editor de Mapeo visual (Mapping Workbench).
- Soporte limitado para consultas en memoria.

Para más información:



- Una breve historia de TopLink
- EclipseLink

EclipseLink

http://wiki.eclipse.org/Introduction_to_EclipseLink_(ELUG)

Hibernate

Historia

Hibernate fue una iniciativa de un grupo de desarrolladores dispersos alrededor del mundo conducidos por Gavin King.
Tiempo después, Jboss Inc. (comprado recientemente por Red Hat) contrató a los principales desarrolladores de Hibernate y trabajo con ellos en brindar soporte al proyecto.
La rama actual de desarrollo de HIbernate es la 3.x, la cual incorpora nuevas características, como una nueva arquitectura Interceptor/Callback, filtros definidos por el usuario, y de forma opcional el uso de anotaciones para definir la correspondencia bien sustituyendo a los archivos XML o conjuntamente con ellos. Hibernate 3 también guarda cercanía con la especificación EJB 3.0 y actúa como eje principal de esta implementación en JBoss.
Hibernate es un entorno de trabajo que tiene como objetivo facilitar la persistencia de objetos Java en bases de datos relacionales y al mismo tiempo la consulta de estas bases de datos para obtener objetos. Esta disponible también para .NET con el nombre NHibernate y 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.
Hibernate es software libre, distribuido bajo los términos de la licencia GNU LGPL.

Características

Como la mayor parte de las herramientas de su mismo tipo, busca solucionar el problema de las grandes diferencias entre los modelos utilizados hoy en día para organizar y manipular objetos. El modelo orientado a objetos y el utilizado en la base de datos, modelo relacional.
Para lograr esto permite al desarrollador detallar cómo es su modelo de datos, qué relaciones existen y qué forma tienen. Con esta información
Hibernate le permite a la aplicación manipular los datos de la base operando sobre objetos, con todas las características de la Programación Orientada a Objetos. Hibernate convertirá los datos entre los tipos utilizado por Java y los definidos por SQL. Genera las sentencias SQL y libera al desarrollador del manejo manual de los datos que resultan de la ejecución de dichas sentencias, manteniendo la portabilidad entre todas las bases de datos con un ligero incremento en tiempo de ejecución.
Hibernate está diseñado para ser flexible en cuanto al esquema de las tablas utilizado, para poder adaptarse a su utilización sobre la base de datos ya existente. También tiene la funcionalidad de crear la base de datos a partir de la información disponible.
Hibernate ofrece también un lenguaje de consulta de datos llamado HQL (Hibernate Query Language), al mismo tiempo que una API para construir las consultas programadas automáticamente (conocido comúnmente como 'Criteria').
Hibernate para Java puede ser utilizado en aplicaciones Java independiente o en aplicaciones Java EE, mediante le componente Hibernate Annotations que implemente el estándar JPA, que es parte de esta plataforma.

Bases de datos que soporta Hibernate

- Oracle 8i, 9i, 10g
- DB2 7.1, 7.2, 8.1, 9.1 DB2 7.1, 7.2, 8.1, 9.1
- Microsoft SQL Server 2000 Microsoft SQL Server 2000
- Sybase 12.5 (JConnect 5.5) Sybase 12.5 (JConnect 5,5)
- MySQL 3.23, 4.0, 4.1, 5.0 MySQL 3.23, 4.0, 4.1, 5.0
- PostgreSQL 7.1.2, 7.2, 7.3, 7.4, 8.0, 8.1 PostgreSQL 7.1.2, 7.2, 7.3, 7.4, 8.0, 8.1
- TimesTen 5.1, 6.0 TimesTen 5.1, 6.0
- HypersonicSQL 1.61, 1.7.0, 1.7.2, 1.7.3, 1.8 HypersonicSQL 1,61, 1.7.0, 1.7.2, 1.7.3, 1.8
- SAP DB 7.3 SAP PP 7,3
- InterSystems Cache' 2007.1 InterSystems Caché '2007,1

Hibernate también ha sido probado y se cree que es compatible con las versiones actuales de:

- Apache Derby Apache Derby
- HP NonStop SQL/MX 2.0 (requires Dialect from HP) HP NonStop SQL / MX 2.0 (requiere dialecto de HP)
- Firebird (1.5 with JayBird 1.01 tested) Firebird (1,5 con la prueba JayBird 1,01)
- FrontBase
- Informix
- Ingres
- Interbase (6.0.1 tested) Interbase (prueba 6.0.1)
- Mckoi SQL Mckoi SQL
- Pointbase Embedded (4.3 tested) Pointbase Embedded (4,3 probado)
- Progress 9 Progreso 9
- Microsoft Access version from 95, 97, 2000, XP, 2002, to 2003 (requires Dialect from HXTT) Microsoft Access versión de 95, 97, 2000, XP, 2002, 2003 (dialecto requiere de HXTT)
- Corel Paradox version from 3.0, 3.5, 4.x, 5.x, 7.x to 11.x (requires Dialect from HXTT) Corel paradoja de la versión 3.0, 3.5, 4.x, 5.x, 7.x a 11.x (requiere dialecto de HXTT)
- flat text , CSV file, TSV file, fixed-length, and variable-length binary file (requires Dialect from HXTT) texto plano, CSV, TSV archivo, de longitud fija y longitud variable archivo binario (requiere dialecto de HXTT)
- Xbase database (dbase, Visual DBASE, SIx Driver, SoftC, Codebase, Clipper, Foxbase, Foxpro, VFP(3.0,5.0,7.0,8.0,9.0, 10), xHarbour, Halcyon, Apollo, Goldmine, and BDE) (requires Dialect from HXTT) Xbase base de datos (dBase, Visual DBASE, SEIS Conductor, SoftC, código, Clipper, FoxBase, FoxPro, VFP (3.0,5.0,7.0,8.0,9.0, 10), xHarbour, Halcyon, Apollo, Goldmine y BDE) (requiere Dialecto de HXTT)
- Microsoft Excel version from 5.0, 95, 97, 98, 2000, 2001, 2002, 2003, to 2004 (requires Dialect from HXTT) Microsoft Excel desde la versión 5.0, 95, 97, 98, 2000, 2001, 2002, 2003, a 2004 (dialecto requiere de HXTT)

Ejemplo

Ejemplo de una aplicación de **gestión de Infracciones**.

Archivo de configuración de Hibernate.

Hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
                    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
                    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
```

```
        <!-- Driver class para ORACLE -->
    <property name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
    <!-- Poner nuestro password -->
    <property name="hibernate.connection.password">root </property>
    <!-- Formato url para ORACLE, poner nuestra ip, el puerto y el nombre de la BD -->
    <property name="hibernate.connection.url">jdbc:oracle:thin:@192.168.2.7:1521:INFRACCIONES</property>
    <!-- Poner nuestro username -->
    <property name="hibernate.connection.username">root</property>
    <!-- Dialecto de hibernate para ORACLE -->
    <property name="hibernate.dialect">org.hibernate.dialect.2OracleDialect</property>
  </session-factory>
</hibernate-configuration>
```

HiberanteMM.reveng.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-reverse-engineering PUBLIC "-//Hibernate/Hibernate Reverse Engineering DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-reverse-engineering-3.0.dtd" >

<hibernate-reverse-engineering>
        <!-- nombre del esquema de BD, poner nuestro esquema -->
        <schema-selection match-schema="INFRACCIONES" />
        <!-- Mapeo de tipos, en principio no tocar -->
        <type-mapping>
                <sql-type jdbc-type="NUMERIC" hibernate-type="java.lang.Long" />
                <sql-type jdbc-type="NUMERIC" not-null="true" hibernate-type="java.lang.Long" />
                <sql-type jdbc-type="NUMERIC" length="1" hibernate-type="java.lang.Boolean" />
                <sql-type jdbc-type="NUMERIC" not-null="true"  length="1" hibernate-type="java.lang.Boolean"
                />
                <sql-type jdbc-type="SMALLINT" hibernate-type="java.lang.Short" />
                <sql-type jdbc-type="SMALLINT" not-null="true" hibernate-type="java.lang.Short" />
                <sql-type jdbc-type="INTEGER" hibernate-type="java.lang.Integer" />
                <sql-type jdbc-type="INTEGER" not-null="true" hibernate-type="java.lang.Integer" />
                <sql-type jdbc-type="DECIMAL" hibernate-type="java.lang.Integer" />
                <sql-type jdbc-type="DECIMAL" not-null="true" hibernate-type="java.lang.Integer" />
                <sql-type jdbc-type="NUMERIC" not-null="true" hibernate-type="java.lang.Long" />
                <sql-type jdbc-type="VARCHAR" length="1" hibernate-type="java.lang.Character"/>
                <sql-type jdbc-type="VARCHAR" not-null="true" length="1" hibernate-
                type="java.lang.Character"/>
                <sql-type jdbc-type="CHAR" length="1" hibernate-type="java.lang.Character"/>
                <sql-type jdbc-type="CHAR" not-null="true" length="1" hibernate-type="java.lang.Character"/>
                <sql-type jdbc-type="VARCHAR" hibernate-type="string"/>
        </type-mapping>
        <!-- tablas de las que queremos hacer ingenieria inversa -->
        <!-- cambiar las del ejemplo por nuestras tablas -->
        <table-filter match-name="OFFENDER"/>
        <table-filter match-name="VIOLATION"/>
</hibernate-reverse-engineering>
```

BaseDaoHibernate.java

```
package com.model.dao.hibernate;
import java.io.Serializable;
import java.util.List;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.orm.ObjectRetrievalFailureException;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;

import com.model.dao.Dao;
import com.model.exceptions.DAOException;

/**
 * This class serves as the Base class for all other DAOs - namely to hold
 * common methods that they might all use. Can be used for standard CRUD
```

```
* operations.
* </p>
*
* <p>
* <a href="BaseDAOHibernate.java.html"><i>View Source</i></a>
* </p>
*
* @author <a href="mailto:matt@raibledesigns.com">Matt Raible</a>
*/

public class BaseDaoHibernate extends HibernateDaoSupport implements Dao {
        protected final Log log = LogFactory.getLog(getClass());

        /**
         * @see com. model.dao.DAO#saveObject(java.lang.Object)
         */
        public void saveObject(Object o) throws DAOException  {
                try {
                        log.debug("Inicio metodo saveObject(Object o)");
                        getHibernateTemplate().saveOrUpdate(o);
                } catch (Exception ex) {
                        log.error("Error metodo saveObject(Object o) :"+ ex.getStackTrace());
                        throw new DAOException(ex);
                } finally {
                        log.debug("Fin metodo saveObject(Object o)");
                }
        }


        /**
         * @see com.dao.DAO#getObject(java.lang.Class, java.io.Serializable)
         */
        public Object getObject(Class clazz, Serializable id) throws DAOException {
                try {
                        log.debug("Inicio metodo getObject(Class clazz, Serializable id)");
                        Object o = getHibernateTemplate().get(clazz, id);
                        if (o == null) {
                                throw new ObjectRetrievalFailureException(clazz, id);
                        }
                        return o;
                } catch (Exception ex) {
                        log.error("Error metodo getObject(Class clazz, Serializable id) :"
                                        + ex.getStackTrace());
                        throw new DAOException(ex);
                } finally {
                        log.debug("Fin metodo getObject(Class clazz, Serializable id)");
                }
        }




        /**
         * @see com.model.dao.DAO#getObjects(java.lang.Class)
         */
        public List getObjects(Class clazz) throws DAOException {
                try {
                        log.debug("Inicio metodo getObjec2ts(Class clazz)");
                        return getHibernateTemplate().loadAll(clazz);
                } catch (Exception ex) {
                        log.error("Error metodo getObjects(Class clazz) :"
                                        + ex.getStackTrace());
                        throw new DAOException(ex);
                } finally {
                        log.debug("Fin metodo getObjects(Class clazz)");
                }
        }
```

```
	/**
	 * @see com.telvent.en4.en4sys.model.dao.DAO#removeObject(java.lang.Class,
	 *      java.io.Serializable)
	 */
	public void removeObject(Class clazz, Serializable id) throws DAOException  {
		try {
			log.debug("Inicio metodo removeObject(Class clazz, Serializable id)");
			getHibernateTemplate().delete(getObject(clazz, id));
		} catch (Exception ex) {
			log.error("Error metodo removeObject(Class clazz, Serializable id) :"
								+ ex.getStackTrace());
			throw new DAOException(ex);
		} finally {
			log.debug("Fin metodo removeObject(Class clazz, Serializable id) ");
		}
	}
}
```

InfractionDaoHibernate.java

```
package com.dao.hibernate;

import static org.hibernate.criterion.Example.create;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.hibernate.LockMode;
import com.model.Infraction;
import com.model.dao.InfractionDao;

/**
 * Clase que implementa InfractionDao.
 *
 * @see com.model.Infraction
*/
public class InfractionDaoHibernate extends BaseDaoHibernate implements
		InfractionDao {
	/**
	 * Logger
	 */
	private static final Logger LOGGER = Logger
			.getLogger("InfractionDaoHibernate");

	/**
	 * Constructor
	 */
	public InfractionDaoHibernate() {
		super();
	}

	/**
	 * @param aTransientInstance
	 *            Instance of {@link Infraction}
	 */
	public void add(final Infraction aTransientInstance) {
		if (LOGGER.isLoggable(Level.FINEST)) {
			LOGGER.log(Level.FINEST, "persisting Infraction instance");
		}
		try {
			getHibernateTemplate().getSessionFactory().getCurrentSession()
					.persist(aTransientInstance);
			if (LOGGER.isLoggable(Level.FINEST)) {
				LOGGER.log(Level.FINEST, "persist successful");
			}
		} catch (RuntimeException re) {
			if (LOGGER.isLoggable(Level.ALL)) {
				LOGGER.log(Level.ALL, "persist failed");
```

```
                }
                throw re;
        }
}

/**
 * @param anInstance
 *          Instance of {@link Infraction}
 */
public void update(final Infraction anInstance) {
        if (LOGGER.isLoggable(Level.FINEST)) {
                LOGGER.log(Level.FINEST, "attaching dirty Infraction instance");
        }
        try {
                getHibernateTemplate().getSessionFactory().getCurrentSession()
                        .saveOrUpdate(anInstance);
                if (LOGGER.isLoggable(Level.FINEST)) {
                        LOGGER.log(Level.FINEST, "attach successful");
                }
        } catch (RuntimeException re) {
                if (LOGGER.isLoggable(Level.ALL)) {
                        LOGGER.log(Level.ALL, "attach failed");
                }
                throw re;
        }
}


/**
 * @param anInstance
 *          Instance of {@link Infraction}
 */
public void attachClean(final Infraction anInstance) {
        if (LOGGER.isLoggable(Level.FINEST)) {
                LOGGER.log(Level.FINEST, "attaching clean Infraction instance");
        }
        try {
                getHibernateTemplate().getSessionFactory().getCurrentSession()
                                .lock(anInstance, LockMode.NONE);
                if (LOGGER.isLoggable(Level.FINEST)) {
                        LOGGER.log(Level.FINEST, "attach successful");
                }
        } catch (RuntimeException re) {
                if (LOGGER.isLoggable(Level.ALL)) {
                        LOGGER.log(Level.ALL, "attach failed");
                }
                throw re;
        }
}

/**
 * @param aPersistentInstance
 *          Instance of {@link Infraction}
 */
public void delete(final Infraction aPersistentInstance) {
        if (LOGGER.isLoggable(Level.FINEST)) {
                LOGGER.log(Level.FINEST, "deleting Infraction instance");
        }
        try {
                getHibernateTemplate().getSessionFactory().getCurrentSession()
                                .delete(aPersistentInstance);
                if (LOGGER.isLoggable(Level.FINEST)) {
                        LOGGER.log(Level.FINEST, "delete successful");
                }
```

```
		} catch (RuntimeException re) {
			if (LOGGER.isLoggable(Level.ALL)) {
				LOGGER.log(Level.ALL, "delete failed");
			}
			throw re;
		}
	}

	/**
	 * @param anId Instance of {@link Integer}
	 * @return Instance of {@link Infraction}
	 */
	public Infraction findById(final java.lang.Integer anId) {
		if (LOGGER.isLoggable(Level.FINEST)) {
			LOGGER.log(Level.FINEST, "getting Infraction instance with id: "
					+ anId);
		}
		try {
			final Infraction instance = (Infraction) getHibernateTemplate()
				.getSessionFactory().getCurrentSession()
					.get("com.model.Infraction", anId);
			if (instance == null) {
			if (LOGGER.isLoggable(Level.FINEST)) {
			    LOGGER.log(Level.FINEST,  "get successful, no instance found");
			}
			} else {
				if (LOGGER.isLoggable(Level.FINEST)) {
					LOGGER.log(Level.FINEST, "get successful, instance
found");
				}
			}
			return instance;
		} catch (RuntimeException re) {
			if (LOGGER.isLoggable(Level.ALL)) {
				LOGGER.log(Level.ALL, "get failed");
			}
			throw re;
		}
	}

	/**
	 * @param anInstance
	 *          Instance of {@link Infraction}
	 * @return List of {@link Infraction}
	 */
	public List<Infraction> findByExample(final Infraction anInstance) {
		if (LOGGER.isLoggable(Level.FINEST)) {
			LOGGER.log(Level.FINEST, "finding Infraction instance by example");
		}
		try {
			final List<Infraction> results = (List<Infraction>) getHibernateTemplate()
				.getSessionFactory().getCurrentSession().createCriteria(
					"com.model.Infraction").add(
					create(anInstance)).list();
			if (LOGGER.isLoggable(Level.FINEST)) {
				LOGGER.log(Level.FINEST,
						"find by example successful, result size: "
							+ results.size());
			}
			return results;
		} catch (RuntimeException re) {
			if (LOGGER.isLoggable(Level.ALL)) {
```

```
                                LOGGER.log(Level.ALL, "find by example failed");
                        }
                        throw re;
                }
        }
}
```

OffenderDaoHibernate.java

```
package com.model.dao.hibernate;

import java.util.List;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.LockMode;
import static org.hibernate.criterion.Example.create;


/**
 * Clase que implementa OffenderDao.
 * @see com.telvent.en4.en4sys.model.Offender
 * @author (with Hibernate Tools)
 */

import com.model.dao.OffenderDao;
import com.model.Offender;

public class OffenderDaoHibernate extends BaseDaoHibernate implements
                OffenderDao {

        private static final Log log = LogFactory
                .getLog(OffenderDaoHibernate.class);
        public void add(Offender transientInstance) {
                log.debug("persisting Offender instance");
                try {
                        getHibernateTemplate().getSessionFactory().getCurrentSession()
                                .persist(transientInstance);
                        log.debug("persist successful");
                } catch (RuntimeException re) {
                        log.error("persist failed", re);
                        throw re;
                }
        }

        public void update(Offender instance) {
                log.debug("attaching dirty Offender instance");
                try {
                        getHibernateTemplate().getSessionFactory().getCurrentSession()
                                        .saveOrUpdate(instance);
                        log.debug("attach successful");
                } catch (RuntimeException re) {
                        log.error("attach failed", re);
                        throw re;
                }
        }

        public void attachClean(Offender instance) {
                log.debug("attaching clean Offender instance");
                try {
                        getHibernateTemplate().getSessionFactory().getCurrentSession()
                                        .lock(instance, LockMode.NONE);
                        log.debug("attach successful");
                } catch (RuntimeException re) {
                        log.error("attach failed", re);
                        throw re;
                }
```

```
	}

	public void delete(Offender persistentInstance) {
		log.debug("deleting Offender instance");
		try {
			getHibernateTemplate().getSessionFactory().getCurrentSession()
					.delete(persistentInstance);
			log.debug("delete successful");
		} catch (RuntimeException re) {
			log.error("delete failed", re);
			throw re;
		}
	}

	public Offender findById(java.lang.Integer id) {
		log.debug("getting Offender instance with id: " + id);
		try {
			Offender instance = (Offender) getHibernateTemplate()
					.getSessionFactory().getCurrentSession().get(
							"com.model.Offender", id);
			if (instance == null) {
				log.debug("get successful, no instance found");
			} else {
				log.debug("get successful, instance found");
			}
			return instance;
		} catch (RuntimeException re) {
			log.error("get failed", re);
			throw re;
		}
	}

	public List<Offender> findByExample(Offender instance) {
		log.debug("finding Offender instance by example");
		try {
			List<Offender> results = (List<Offender>) getHibernateTemplate()
				.getSessionFactory().getCurrentSession().createCriteria(
				"com.Offender").add(create(instance)).list();
			log.debug("find by example successful, result size: "
					+ results.size());
			return results;
		} catch (RuntimeException re) {
			log.error("find by example failed", re);
			throw re;
		}
	}
}
```

El archivo de contexto de hibernate, para configurar los daos de acceso a datos.

applicationContext-hibernate.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns:aop="http://www.springframework.org/schema/aop"
	xmlns:tx="http://www.springframework.org/schema/tx"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
	http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-
2.0.xsd
	http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">

	<!-- Hibernate SessionFactory -->
	<bean id="sessionFactory"
		class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
```

```
                <property name="dataSource" ref="dataSource" />
                <property name="mappingResources">
                        <list>
                                <value>
                                        com /model/Offender.hbm.xml
                                </value>
                                <value>
                                        com /model/Infraction.hbm.xml
                                </value>
                        </list>
                </property>

                <property name="hibernateProperties">
                        <props>
                                <prop key="hibernate.dialect">
                                        org.hibernate.dialect.OracleDialect
                                </prop>
                                <prop key="hibernate.query.substitutions">
                                        true 'Y', false 'N'
                                </prop>
                                <prop key="hibernate.show_sql">true</prop>
                                <!-- Create/update the database tables automatically when the JVM starts up
                                        <prop key="hibernate.hbm2ddl.auto">update</prop>-->
                                <!-- Turn batching off for better error messages under PostgreSQL
                                        <prop key="hibernate.jdbc.batch_size">0</prop>-->
                        </props>
                </property>
        </bean>

        <!-- Generic Dao - can be used when doing standard CRUD –>

        <bean id="dao"
                class="com.model.dao.hibernate.BaseDaoHibernate">
                <property name="sessionFactory" ref="sessionFactory" />
        </bean>


        <!-- Transaction manager for a single Hibernate SessionFactory (alternative to JTA) -->

        <bean id="transactionManager"
                class="org.springframework.orm.hibernate3.HibernateTransactionManager">
                <property name="sessionFactory" ref="sessionFactory" />
        </bean>

        <bean id="infractionDao” class="com.model.dao.hibernate.InfractionDaoHibernateImpl">
                <property name="sessionFactory" ref="sessionFactory" />
        </bean>

        <bean id="offenderDao"
                class="com.model.dao.hibernate.OffenderDaoHibernateImpl">
                <property name="sessionFactory" ref="sessionFactory" />
        </bean>
</beans>
```

InfractionDao.java

```
package com.model.dao;
import java.util.List;
import com.model.Infraction;

public interface InfractionDao extends Dao {
        /**
         *
         * @param transientInstance an Infraction instance
         */
   void add(Infraction transientInstance);
   /**
```

```
     *
     * @param instance an Infraction instance
     */
    void update(Infraction instance);

    /**
     *
     * @param instance an Infraction instance
     */
    void attachClean(Infraction instance);

    /**
     *
     * @param persistentInstance an Infraction persisted instance
     */
    void delete(Infraction persistentInstance);

    /**
     *
     * @param id an id
     * @return The infraction that corresponds to the given id
     */
    Infraction findById(java.lang.Integer id);

    /**
     *
     * @param instance an Infraction instance
     * @return a list of Infraction
     */
    List<Infraction> findByExample(Infraction instance);

}
```

OffenderDao.java

```
package com.model.dao;

import java.util.List;


import com.model.Offender;

public interface OffenderDao extends Dao {

/**
 *
 * @param transientInstance an Offender instance
 */
    void add(Offender transientInstance);

    /**
     *
     * @param instance an Offender instance
     */
    void update(Offender instance);

    /**
     *
     * @param instance an Offender instance
     */
    void attachClean(Offender instance);
```

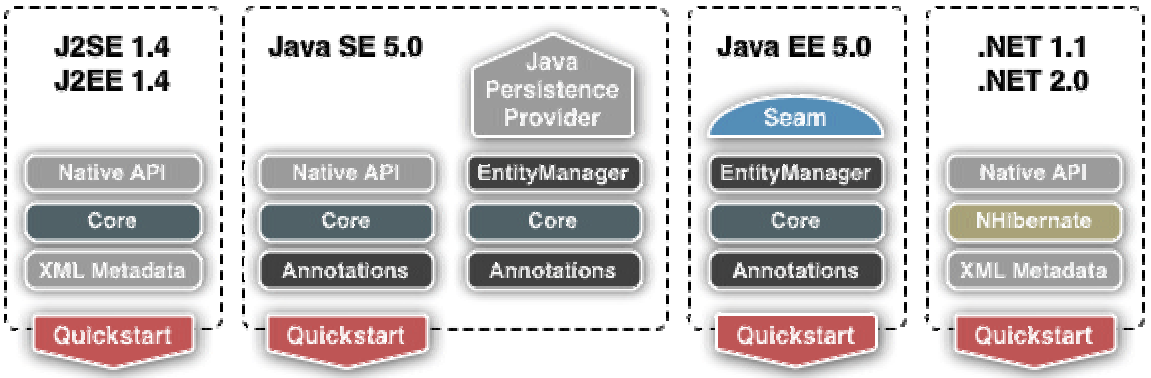
```
    /**
     *
     * @param persistentInstance an Offender persistent instance
     */
    void delete(Offender persistentInstance);

    /**
     *
     * @param id an id
     * @return The offender that corresponds to the given id
     */
    Offender findById(java.lang.Integer id);

    /**
     *
     * @param instance an Offender instance
     * @return a list of Offender
     */
    List<Offender> findByExample(Offender instance);

}
```

Necesarios los jar de Hibernate 3.2 así como el hibernate-tools para generar los archivos con ant, previamente construyendo el build.xml quenos permita a partir de nuestra tabla generar los objetos o pojos.



Castor

Castor es un código abierto de enlace a datos para Java [TM]. Es el camino más corto entre los objetos Java, documentos XML y tablas relacionales. Castor ofrece una vinculación Java-XML, persistencia de Java-SQL y mucho más.
La versión actual de Castor es “Castor 1.3 RC1 Release”

Más información en: http://www.castor.org/

Java Ultra Lite Persistente - JULP

Herramienta de mapeo objeto relacional alternativa a herramientas de persistencia como BMP, CMP EntityBeans, CachedRowSet, JDBC que posee los siguientes requerimientos:

- Código abierto.
- Ejecución en contenedor y stand-alone.
- Base de datos independiente.
- Utilización de inherencia.
- Concurrencia optimista de boqueo.
- Uso de muchas clases por tabla.
- Uso de muchas tablas por clase.
- No utiliza muchas complicados ficheros XML para el mapeo.
- Ejecución en modo desconectado.

Después de comparar este framework con otros similares, este personalizado framework (con menos de 50kb) JULP (Java Ultra-Lite Persistente) ha sido probado con JBoss, Tomcat y stand-alone que está bajo licencia GPL y lo puede utilizar cualquier usuario que lo crea útil.

JLUP está basado en el patrón DAO y si algún día se decide unir a un framework es relativamente sencillo.

Por ejemplo, si tu quieres utilizar TIMESTAMP or DATETIME or versión columna para concurrencia bloqueante optimista puede usar este trick. JULP supone automáticamente la consulta de la información de la primaryKey de DataBaseMetaData, y tu puedes añadir tu TimeStamp/DateTime/version columna a la información de la clave primaria y esta columna será utilizada para crear una cláusula hiere fpar la actualización o el borrado (Update or Delete).

```
CREATE TABLE PRODUCT(ID INTEGER NOT NULL PRIMARY KEY,NAME VARCHAR,PRICE DECIMAL,
..., TIMESTAMP DATE_MODIFIED)
...
String catalog;
String schema;
String table = "PRODUCT";
List pk;
...
pk = PKCache.getInstance().getPrimaryKey(catalog, schema, table);
pk.add("PRODUCT.DATE_MODIFIED");
PKCache.getInstance().setPrimaryKey(catalog, schema, table, pk);
...
factory.remove(product);
...
Generated SQL would be: DELETE FROM PRODUCT WHERE ID = ? AND DATE_MODIFIED = ?;
```

La versión actual de JULP es: Java Ultra-Lite Persistence (JULP) v2.
Para más información: http://julp.sourceforge.net

OBJ - Apache ObjectRelationalBridge

En aplicaciones grandes, los cambios en el mapeo de los modelos relacionales de las tablas de las bases de datos son muy frecuentes. Es común la utilización de EJB, entity beans, por

parte de los desarrolladores de aplicaciones J2EE. Los EJB permiten abstraer los objetos, aunque tiene capacidades limitadas de mapeo, si hablamos de EJB 2.0, ya que, por ejemplo la conversión de datos personalizados o la herencia de componentes no son soportados directamente. Cuando se requiere para los modelos cualesquiera de estas características, los desarrolladores tienen que escribir montones de líneas de código, incluso con frecuencia los resultados no son del todo satisfactorios.

Afortunadamente, tenemos alternativas. Los frameworks de persistencia permiten a los desarrolladores mapear objetos java a tablas relacionales con mínimo esfuerzo. El estándar Java para cada framework se llama JDO (Java DataObjects) el cual ha madurado mucho en los últimos años. Desafortunadamente, no es completa la implementación libre de JDO de la actualidad. De todas formas, con alguna planificación y uso de buenas prácticas, no es difícil desarrollar tu prototipo de aplicación con un framework de persistencia no estándar.

Los más populares frameworks de persistencia de la comunidad de código abierto son con ya dije anteriormente Hibernate, también Castor y OBJ. En este apartado, nos centramos en OBJ, el cual integra contenedores J2EE con soporte completo para JTA y JCA, y como una alternativa válida EJB o entity beans.

Con OBJ no modificas tu código fuente de ninguna manera, tu puedes hacer la persistencia a objetos incluso sin acceder al código original. Destacamos también la capacidad de este framework a soportar componentes inherentes. Además tenemos buenas prácticas en la web oficial de OBJ de cómo acoplar a tu aplicación la capa de persistencia utilizando este framework.

Flexibilidad - Múltiples API's de Persistencia

Destacamos que apache ObjectRelationalBridge (OJB) es una herramienta de mapeo objeto/relacional que permite la transparencia de la persistencia de objetos java contra las bases de datos relacionales. Soporta múltiples APIs de persistencia como:

- Persistente Broker API: la cual sirve como kernel de persistencia de OJB. Las implementaciones JDO son construidas por encima de su kernel. Puede ser utilizada directamente por aplicaciones que no necesitan un completo nivel de transacciones. Para más información podemos consultar: http://db.apache.org/ojb/docu/tutorials/pb-tutorial.html
- Las resistentes API ODMG 3.0 que aporta completas características. Para más información: http://db.apache.org/ojb/docu/tutorials/odmg-tutorial.html
- La API JDO. Actualmente OJB provee un plugin que hace referencia a la implementación JDO (RI). Combinando JDO RI y el plugin obtenemos la solución O/R JDO 1.0. Una completa implementación de JDO está fechada para la versión OJB 2.0. Para más información sobre JDO: **http://db.apache.org/ojb/docu/tutorials/jdo-tutorial.htmL**
- Un manejador de transacciones de objetos OTM (Object Transaction Manager), capa que contienen todas las características que JDO y ODMG tienen en común. Más información en: **http://db.apache.org/ojb/docu/tutorials/otm-tutorial.html**

Escalabilidad
OBJ ha sido diseñada por un gran número de aplicaciones, desde sistemas embebidos hasta ricas aplicaciones cliente multitarea basadas en arquitecturas J2EE.
OBJ integra servidores de aplicaciones J2EE. Soporta tecnología 'lookup' JNDI de datasources. Soporta e integra JTA y JCA. Puede ser utilizado dentro de JSPs, Servlets y sesiones de beans. Además, provee especial soporte para BMP (Bean Manager EntityBeans).

Funcionalidad
OJB utiliza XML basado en mapeo Objeto/Relacional. El mapeo reside en la capa dinámica Metadata, la cual puede ser manipulada en tiempo de ejecución hacia un MOP (Meta-Object Protocol) y cambiar el comportamiento de la persistencia del kernel.
OJB provee una flexible configuración y un mecanismo plugin que permite seleccionar componentes predefinidos o implementaciones de propias extensiones y plugins.
Para profundizar aun en más características de OJB podemos acceder a: Features OJB

Más información en:

http://db.apache.org/ojb/

http://www.onjava.com/pub/a/onjava/2003/01/08/ojb.html?page=1

Torque

Torque es una capa de persistencia que incluye un generador para generar todos los recursos requeridos de base de datos para que tu aplicación e incluye un entorno de tiempo de ejecución para generar las clases en tiempo de ejecución a partir de la base de datos.

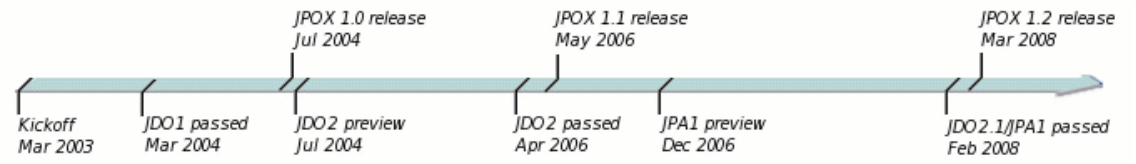
Torque fue desarrollado como parte del framework Turbina. Actualmente está desacoplado de dicho framework y puede ser utilizado independientemente. Comenzando con la versión 2.2 de Turbina, Torque ya está desacoplado. El entorno de ejecución de Torque contiene todo lo necesario para utilizar el generador de clases OM/Peer. Además, incluye un pool de conexiones jdbc al igual que hibernate.

JPOX – Java Persistent Objects

JPOX es plenamente compatible con una aplicación de la API Java Data Objects (JDO). El API JDO es una interfaz estándar basada en el modelo de abstracción de persistencia en Java. Es libre bajo una licencia OpenSource y por lo tanto el código fuente esta disponible para su descarga junto con la aplicación JDO.

JPOX soporta la persistencia de todos los Sistemas gestores de bases de datos del mercado de hoy en día, persistencia de almacenes de datos DB40, soporta los principales patrones ORM demandados por las aplicaciones de la actualidad, permite realizar consultas utilizando el lenguaje JDOQL, SQL o JPGQ, y tiene su propio analizador de bytes de código.

Está disponible bajo la licencia OpenSource de Apache 2, que permite el acceso no sólo a una capa superior de persistencia de aplicación java, sino también al código fuente, lo que permite contribuir al éxito de la historia principal compatible con los estándares de código abierto de persistencia de la actualidad.



JPOX es una solución heterogénea de persistencia Java. Permite a un equipo de trabajo tomar un modelo de datos y persistir los objetos a una base de datos sin tener que malgastar gran cantidad de tiempo en la definición o forma en la que debemos persistir. Esto significa que te centraras en el desarrollo de tu aplicación y su lógica de negocio en lugar de la rutina de la tarea de almacenamiento y recuperación de sus objetos.

JPOX está diseñado para cumplir con todas las normas persistentes en la persistencia de dominio. Es totalmente compatible con una aplicación de JDO1, JDO2 y cumple con la JPA1 y JDO2.1 en sus JPOX 2.1. Esto quiere decir, que escogiendo JPOX tenemos persistencia confiable y segura. Además, es extensible utilizando el mecanismo plugin de la norma OSGi y prove muchos puntos de extensión donde puedes potenciar la capacidad de tu capa de persistencia llevándola aun más lejos.

Diferencias entre JPOX 1.1 y JPOX 1.2

JPOX 1.1

- Fecha de publicación: mayo de 2006
- Cumplimiento de especificaciones: JDO1, JDO2
- Soporte: Sólo con el apoyo de donaciones
- Desarrollo: Sólo con donaciones
- Comentarios: JDO2 apoyo añadido. JPOX 1.1.0 es SUN JDO2 "Reference Implementation"

JPOX 1.2

- Fecha de publicación: marzo 2008
- Cumplimiento de especificaciones: JDO1, JDO2, JDO2.1, JPA1
- Soporte: Sólo con el apoyo de donaciones
- Desarrollo: Sólo con donaciones
- Comentarios: JPA1 apoyo añadido, JDO2.1 apoyo añadido, soporte espacial, DB4O datastore añadió. JPOX 1.2 será SUN JDO2.1 "Reference Implementation"

La Documentación de JPOX podemos encontrarla en: Documentación JPOX

TJDO

TriActive JDO (TJDO) es una implementación de código abierto de Sun, de la especificación JSR 12. Posee un diseño el soporte de persistencia transparente de JDO que se adapta a cualquier base de datos compatible con JDBC. TJDO ha sido desplegado y puesto en funcionamiento con éxito en muchas instalaciones comerciales desde el año 2001.

JDBM

JDBM es un motor de persistencia para Java. Su objetivo es ser de Java un rápido, simple motor de persistencia como GDBM lo es para otros lenguajes como C/C++, Python, Perl, etc. Es posible utilizarlo para almacenar una mezcla de objeto y blob, todas las actualizaciones se realizan transacionalmente de manera segura y asegurando la atomicidad de las mismas.
JDBM proporciona estructuras de datos escalables, como HTree y B+Tree, apoyo a la persistencia de de grandes colecciones de objetos.

Prevayler

Es un frecuente software libre para una capa de persistencia para java. Ridículamente simple, es con creces la más rápida y transparente capa de persistencia, con tolerancia a fallo y arquitectura de balanceado de carga para la arquitectura antigua de objetos Java o POJOS (Plain Old Java Objects).

Speedo

Es una aplicación de código abierto de JDO, su especificación.

Space4J

También llamado S4J, Es una aplicación libre de prevalencia en Java. La prevalencia es un concepto iniciado por Klaus Wuestefeld sobre la forma de almacenar datos en una real forma orientada a objetos, utilizando sólo la memoria instantáneas, registro de transacciones y serialización. Además de la funcionalidad básica, ofrece transparencia para soportar clustering, pasivación y la indexación.

Jaxor

Jaxor es un código de generación de mapeos O/R, herramientas las cuales llevan la información definida en XML que relacionan las entidades relacionales con los mapeados y la generación de clases, interfaces y objetos encontrados, los cuales puede ser utilizados desde una aplicación Java (incluyendo JFC/Swing, J2EE y herramientas en línea de comandos). La generación actual de código es manejada por plantillas de velocidad en lugar de fijarse en mecanismos de la herramienta. Esta flexibilidad permite preparar y modificar el formato o incluso el código que se genera.

pBeans

pBeans es un framework de persistencia basado en Java y la capa que mapea la base de datos Objeto/Relacional. Está diseñada para ser fácil de utilizar y completamente automatizada.

Easy ORM

Object Relational Mapping – Easy ORM

La principal idea de este ORM es un proyecto que tiene una clase principal. El resto de las clases ORM a través de la herencia quedan habilitadas para guardar, actualizar datos sin escribir una sólo línea de código.

Ejemplo:

```
public class User extends Table {}
```

Table es la clase mágica de la que extiende User. Para este ejemplo tenderemos dos campos de User que son: username y password.

```
public static void main(String args[]) {
        User u = new User();
        u.put("username","sapan");
        u.put("password","pass);
        u.save();
}
```

El proyecto contiene dos ficheros Table.java y MySqlConnection.java.

Para cargar datos utilizamos u.loadColumns como por ejemplo:

```
public static void main(String [] args) {
        User u = new User();
        u.loadColumns("1", "username,password");
        System.out.println(u.get("username));
        System.out.println(u.get("email"));
}
```

Ejemplo completo utilizando **Sorm**:

```
package sorm;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

import sun.text.CompactShortArray.Iterator;

public class User extends Table{

        public static int insertUser(String nombre, String apes, String pass) {
                int result = 0;
                User u = new User();
                u.put("nombre", nombre);
```

```
		u.put("apellidos", apes);
		u.put("pass", pass);
		try {
			u.save();
			String id = (String) u.get("id");
			result = Integer.parseInt(id);
			System.out.println("insertado con éxito " + u.get("nombre") + " " + result);
		} catch (SQLException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
			result = 0;
			System.out.println("no insertado con éxito");
		}
		return result;
	}

	public static void listUser(int value) {

		User u = new User();
		String value1 = String.valueOf(value);
		System.out.println("valor de value1 " + value1);
		try {
			System.out.println("mostrar usuario actual");
			u.loadColumns(value1,"nombre,apellidos,pass");

		} catch (SQLException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		try {
			System.out.print("nombre -> " + u.get("nombre"));
			System.out.print(", apellidos -> " + u.get("apellidos"));
			System.out.print(", password -> " + u.get("pass"));
		} catch (SQLException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}

	public static HashMap[] listAll() {
		HashMap[] results = null;
		try {
			ArrayList users = new User().select("id>1","id");
//			for (int i=0; i < sorm.User user = (sorm.User) users.get(i); out.println(user);)
			results = new Query("select * from user").runQuery();

		} catch (SQLException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} catch (Exception e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}


		return results;
	}
	public static void main(String [] args) {
		int insertado = 0;
		String texto = null;
		String apes = null;
		String pass = null;

		InputStreamReader isr = new InputStreamReader(System.in);
		BufferedReader br = new BufferedReader (isr);


		InputStreamReader apes2 = new InputStreamReader(System.in);
		BufferedReader br2 = new BufferedReader (apes2);
```

```
                InputStreamReader pass2 = new InputStreamReader(System.in);
                BufferedReader br3 = new BufferedReader (pass2);

                try {
                        System.out.print("introduce por favor un nombre: ");
                        texto = br.readLine();
                        System.out.print("introduce por favor los apellidos: ");
                        apes = br2.readLine();
                        System.out.print("introduce por favor un password: ");
                        pass = br3.readLine();
                        insertado = insertUser(texto, apes, pass);
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }


                listUser(insertado);

        }

}
```

SimpleORM

SimpleORM es Mapeador Objeto-Relacional de un proyecto de código abierto con licencia Apache. Provee una simple pero efectiva implementación de mapeado objeto/relacional por encima de JDBC a más bajo coste y menor sobrecarga. No necesita de ficheros de configuración para configurar que suelen ser tediosos y dar algún problema en herramientas similares.

SimpleORM evita exóticas tecnologías como la generación de bytes de código. Tiene funcionalidades similares a Hibernate para el mapeado de datos en una base de datos relacional para objetos Java en memoria. Las consultas pueden ser especificadas en términos de objetos Java, identidad de objetos es ajustada con claves de la base de datos, posee una relación entre objetos que son modificados y mantenidos automáticamente con bloqueos optimizadores.

Pero a diferencia de Hibernate, SimpleORM utiliza una estructura de objetos simples y arquitectura que evita la necesidad de un parseado complejo, procesado de bytes por codigo, etc. SimpleORM es pequeño y transparente y empaquetado en 2 jars de simplmenete 79 Kb y 52 Kb, con sólo una pequeña dependencia (Slf4j). Hibernate esta por encima de 2400 Kb a mayores sobre las 2000 kb de dependencias Jar. Esto hace que SimpleORM sea fácil de entender y con una magnífica reducción de riesgo técnico. Además, utiliza una muy simple arquitectura y estructura de objetos, y su versión más actual es 3.* que provee una diferencia en el componente DataSet que simplifica la definidos de registros.

Utilización con SimpleOrm con un framework

SimpleWebApp es un framework que puede utilizar SimpleORM para implemntar aplicaciones web muy fácilmente. Estas simples aplicaciones no suelen ser complejas de implementar.

La versión ligera de este framework, SimpleWebApp's lightweight, aporta alta productividad de diseño y analogía a Ruby on Rails. El uso de sus metadatos es más común para tipos de sistemas más estructurados en Java, pero posee un alto nivel de abstracción para operaciones comunes.

A diferencia de otros ORM's como Hibernate, la forma de agrupar los datos es muy distinta en este framework. En lugar de utilizar POJO 's se utilizan Records tal que así:

```
class Employee...
        private static final SRecordMeta EMPLOYEE // the record
        = new SRecordMeta(Employee.class, "EMPLOYEE_TABLE");
        ...
        public static final SFieldString PHONE_NR // a field
        = new SFieldString(EMPLOYEE, "PHONE_NR", 30);

        public String getPhoneNr() { // Optional get method if you like them
                return getString(PHONE_NR);
        }
```

Este Employee contiene un objeto que representa el registro Empleado (Tabla SQL), el metadato Phone_NR, que contiene el dato de la columna SQL Phone Number y un método Get que personaliza la obtención de los datos, que también se pueden recoger con 'employee.getString(Employee.PHONE_NR).

Esta estructura generalizada tiene las ventajas siguientes:

- Las declaraciones ORM son muy concisas y de este modo de fácil mantenimiento. No son necesarios ficheros XML.
- Los registros son independientes que los Pojos.
- Tenemos un acceso a los objetos que representan campos independientemente de los getters y los setters.
- La enriquecida estructura de los registros permite distinguir con mayor claridad un valor nulo, un valor cero o un valor inválido.
- Los registros de SimpleOrm viven dentro de un DataSet. A diferencia de Hibernate, los padres de los hijos y los hijos de los padres deben ser consistentes
- Los registros puede ser accedidos sin ninguna conexión a la base de datos.
- Es fácil relacionar Pojos con registros de SimpleOrm pero es una práctica poco habitual.

Así pues, SimpleOrm es un real ORM. Tiene un DataSet que provee propia caché de objetos, y cada registro solo es representado una vez dentro de cada transacción. Las reglas de negocio pueden ser escritas independientemente de otras sin necesidad de coordinar con los datos recibidos. Esto distingue a SimpleORM de Ibatis, Apache DbUtils, etc.

Ejemplo: **Data Definition Example**

```
public class Employee extends SRecordInstance {
// ie. a class mapped to a table.

        private static final SRecordMeta EMPLOYEE = new SRecordMeta(Employee.class, "XX_EMPLOYEE");

        public static final SFieldString EMPEE_ID
          = new SFieldString(EMPLOYEE, "EMPEE_ID", 20, SPRIMARY_KEY);
        public static final SFieldString NAME
          = new SFieldString(EMPLOYEE, "ENAME", 40, SMANDATORY, SDESCRIPTIVE);
```

```
        public static final SFieldString PHONE_NR
          = new SFieldString(EMPLOYEE, "PHONE_NR", 20)
            .putUserProperty("DISPLAY_LABEL", "Telephone Number");
        public static final SFieldDouble SALARY
          = new SFieldDouble(EMPLOYEE, "SALARY").addValidator(new SValidatorGreaterEqual(0));
        public static final SFieldString DEPT_ID = new SFieldString(EMPLOYEE, "DEPT_ID", 10);
        static final SFieldReference<Department> DEPARTMENT
          = new SFieldReference(EMPLOYEE, Department.DEPARTMENT, "DEPT");
        public enum EType{PERMANENT, CASUAL, CONTRACT};
        static final SFieldEnum<Type> ETYPE = new SFieldEnum(EMPLOYEE, "ETYPE", EType.class);
        static final SFieldString RESUME // Curriculum Vitae
          = new SFieldString(EMPLOYEE, "RESUME", 200, SUNQUERIED)
            .overrideSqlDataType("VARCHAR ( 200)"); // Maybe LONG VARCHAR for Oracle etc.
        public @Override SRecordMeta<Employee> getMeta() {return EMPLOYEE;};

        // Completely optional get/set methods.
        public String getPhoneNumber() {return getString(PHONE_NR);}
        public void setPhoneNumber(String value) {setString(PHONE_NR, value);}
}
```

Data Manipulation Example

```
SSessionJdbc session = SSessionJdbc.open(myDataSource, "MyInformativeLoggingLabel");
session.begin();
...
Employee employee = session.mustFind(Employee.EMPLOYEE, SFOR_UPDATE, "123-45-6789");

String name = employee.getString(Employee.NAME);
String phone = employee.getPhoneNr(); // if getter defined
employee.setPhoneNr("(123) 456 7890");
Employee.EType etype = e100a.getEnum(e100a.ETYPE);
...
println(employee.getString(Employee.DEPT_ID));
Department department = employee.findReference(employee.DEPARTMENT);
...
oldEmployee.delete();
...
session.commit();
session.begin(); ... session.commit();
session.close();
```

En conclusión, SimpleORM es fácil de entender, no demasiado elaborado, modesto y sencillo y no complicado.

Versión actual: SimpleORM 3.0 ‘released’ con una nueva arquitectura DataSet.

Para más información: http://www.simpleorm.org/

Smyle

Simple API de almacenamiento par ayudar a desarrollador a centrarse en las aplicaciones en lugar de escribir código JDBC.

XORM

XORM es una capa extensible O/R para aplicaciones Java.

Ammentos

Un framework de persistencia ligero para Java (JDK5.0 en adelante). Extremadamente sencillo de utilizar, trabaja con anotaciones, no requiere ninguna configuración o fichero de mapeo, y se ejecuta en cualquier base de datos compatible con JDBC (no dialectos SQL para configurar y apoya multi-dispositivo de persistencia.

JDBCPersistence

JDBCPersistence es otra capa de mapeo OR. Se diferencia de sus pares tanto en la implementación como en su API. Utiliza generación de bytes de código en el núcleo del framework que genera clases que implementan la lógica JDBC, la cual es específica para un par de tablas/bean. JDBCPersistence genera clases persistentes en tiempo de ejecución sobre la demanda de no incurrir notables por encima del proceso de desarrollo. El marco de toda la configuración se realiza en la API, que mejora significativamente la puesta en marcha y reduce el tamaño de la librería.

Fue creado con los siguientes requerimientos:

- Ser más rápido al cargarse.
- Soportar CLOBs y BLOBs.
- Cargar persistencia de objetos desde java.sql.ResultSet.
- Tener API compactada.
- Tener unas mínimas dependencias en sus proyectos.
- Suportar configuración vía API.

PAT

PAT significa “Persistent Applications Toolkit”, es decir, kit de herramientas de aplicaciones persistentes. Al igual que otro software, simplifica el desarrollo de las capas de persistencia de las aplicaciones de negocio. Lo hace por proveer entorno de persistencia orientado a objetos de sus objetos: pojos, etc. PAT ofrece casi una transparente capa de datos par aúna aplicación empresarial. Emplea el estado de las técnicas para lograrlo. Estas técnicas son OO, AOP (JBossAOP), Java, PRevayler, Ant, JUnit, Log4j, @@annotations y otros. Coopera con las aplicaciones web: struts, Tomcat, JBoss.

O/R Broker

Es un framework convencional para aplicaciones que utilizan JDBC. Te permite externalizar tus sentencias SQL en ficheros individuales, para facilitar la lectura y sencilla manipulación, y permite el mapeo declarativo de las tablas y objetos. No simplemente los objetos javabeans, pero cualquier POJO arbitrariamente. Esto es la fuerte de O/R Broker en comparación con otras herramientas similares de persistencia. Permite desarrollar un diseño propio con las clases orientadas a objetos para la persistencia, sin tener que estar

forzado a utilizar javabens solamente, y tener que soportarías jerarquías inherentes y referencias circulares. Uno de los objetivos del diseño para O/R Broker fue la simplicidad, la cual significa que hay sólo 4 clases públicas, y un fichero de esquema XML para la validación del simple fichero de configuración XML.

En resumen, O/R Broker es un framework JDBC en el que no se dice el diseño de las clases de dominio, si que tiene que ser conforme a las especificación de los beans, teniendo que extender a una superclase o una implantación de una interfaz, pero permite la persistencia independiente y un propio diseño de las clases orientada a objetos.

Super CSV

Super CSV es un paquete CSV para procesar ficheros CSV. Super CSV está diseñado alrededor de objetos sólidos orientados hacia los principios orientados a objetos, para aprovechar su código orientado a objetos, y para hacer más fácil su escritura y mantenimiento. CSV ofrece las siguientes características:

- Capacidad de lectura y escritura de los POJOS beans, mapas y listas de cadenas.
- La capacidad de convertir fácilmente de entrada/salida de números enteros, fechas, cadenas de corte, etc.
- La posibilidad de verificar fácilmente los datos se ajusta a alguna especificación, con el tamaños de cadenas, singularidad e incluso columnas opcionales.
- La habilidad para leer o escribir datos de cualquier codificación.
- Soporte para los saltos de línea de Windows, Mac y Linux.
- Configurable separación de carácter, carácter de espacio final, final de línea de caracteres, etc.

JGrinder

JGrinder es un framework de mapeo de objetos en Java y almacenamiento de varias persistencias. Estos incluyen bases de datos relacionales, en la memoria de almacenamiento, en los ficheros planes. La arquitectura permite la persistencia de almacenamiento adicional (cada objeto de almacenamiento sobre un middleware de mensajería).

Odal

Odal es un framework de persistencia de datos con énfasis en el mantenimiento y el rendimiento. Entre sus características incluyen consulta de la API, ORM, de validación y conversión de datos, el suporte de procedimiento almacenados, la generación de código. Mínimo de dependencias. Corto tiempo de inicio.

BeanKeeper

Netmind Beankeeper es una librería de mapeo de O/R. su tarea es mapear objetos java a bases de datos relacionales, y frecuentemente un potente servicio de consultas de gran alcance para recuperarlas.

Persist

Persist es un minimalista ORM /DAO Java, diseñado para altor rendimiento y facilidad de uso. A diferencia de la mayoría de herramientas ORM, Persist no pretende aislar competencia del código de las aplicaciones de bases de datos relacionales. En lugar de eso, la intención es reducir el esfuerzo de manejar los datos almacenados en los sistemas gestores de bases de datos por medio de JDBC, sin hacer compromisos en el rendimiento y la flexibilidad.

Ebean ORM

Ebean es una capa de persistencia ORM. Está diseñada para aprender a utilizarla fácilmente. Ello sigue la especificación de mapeo de JPA con anotaciones tales como @Entity, @OneToOne, @OneToMany. EBean tiene un API relacional cuando se desea omitir ORM a favor de la utilización de su propio SQL para buscar, actualizar y realizar llamadas a procedimientos almacenados.

Velosurf

Velosurf es una librería de la capa de mapeo de bases de datos para el motor de plantillas de Apache Velocity. Dispone de un mapeo automática de las tablas de la base de datos y su relación sin ninguna generación de código. Se trata de un ligero marcador de persistencia de sistemas.

jPersist

JPersist es una API de persistencia de objetos – relación de bases de datos extremadamente potente que administra para evitar la necesidad de configuración y anotación, mapeo automático. Utiliza JDBC, y puede trabajar con una base de datos relacional y cualquier tipo de recurso de conexión. Utiliza información obtenida desde la base de datos para manejar el mapeo entre los objetos Java y la misma base de datos, así pues, la configuración de mapeo no es necesaria y la anotación tampoco. De hecho, no hay ninguna configuración necesaria en absoluto.

QLor

QLor (Q-Logic Object Relational Mapping Framework), framework de mapeo O/R que permite una persistencia con un alto rendimiento. Es fácil de utilizar e implementar con otras tecnologías. El altamente estructurado y diseñado. Sus características:

- O2O, O2M y M2M mapeos de base de datos.
- Múltiples claves primarias sin modificación de diagrama de clases.

- Claves primarias en cascada sobre las asociaciones.
- Fácil inherencia y múltiple mapeo de inherencia.
- Mapeos programados.
- Limpia y fácil de leer la capa de persistencia de registro.

Daozero

Daozero reduce los códigos DAO en Spring e Ibatis. La antigua forma de hacerlo era de escribir códigos e invocar explícitamente la API iBatis, pero Daozero implementa interfaces DAO en tiempo de ejecución e invoca la API de iBatis para los desarrolladores automáticamente. Sustituye los viejos DAOs con directamente.

SeQuaLite

SeQuaLite es un framework de acceso a datos sobre JDBC. Entre sus características incluye las operaciones CRUD (operaciones básicas de inserción, borrado, modificación), cargas, cascadas, paginado y generación de código SQL sobre la marcha. Ayuda a reducir el tiempo de desarrollo eficazmente.

Mr. Persister

Mr. Persister es una API de persistencia de objetos que hace posible leer y escribir objetos de cualquier tamaño desde cualquier base de datos relacional. La implementación / planificación de 8 características es una fácil inclusión de operaciones JDBC vía plantillas JDBC (estilo Spring), conexiones automáticas, manejos transacionales, mapeo de objeto – relación, informes dinámicos, pool de conexiones, almacenamiento en caché, replicaciones, depuración y mucho más.

LightweitghtModelLayer

LightweigthModelLayer es un pequeño pero fiable motor de persistencia (82 KB) basado en anotaciones y reflexión. No tiene dependencias y pueden ser utilizados por autónomos y aplicaciones web. Una limpia y fácil librería utilizada por desarrolladores que se olvidan de la preocupación habitual de la comunicación con la bases de datos.

ANEXO II:
MANUAL DEL PROGRAMADOR –WAYPERSISTENCE

Escogiendo Motor de Persistencia

WayPersistence es un framework de persistencia para tecnología Java multiplataforma que posee características que permiten el mapeo objeto-relacional entre bases de datos con su modelo relacional o entidad-relación y el modelo de datos orientado a objetos. Se trata de un ORM fácil de entender, simple y sobretodo muy funcional.

No necesita de engorrosos ficheros XML para configurar y está totalmente desarrollado en Java. Como ORM que es, realiza un mapeo entre el modelo de datos relacional y el modelo de datos orientado a objetos haciendo simetría entre tablas con clases, columnas por propiedades, etc. WayPersistence empaqueta objetos java dentro de un DataSet que permite realizar operaciones de registros con mayor facilidad y más eficiencia.

A diferencia de otros ORMs, WayPersistence posee una característica destacable que no utiliza técnicas de riesgo, parseado de consultas generación de bytes de código lo que demuestra una transparencia y simplicidad considerable.

La idea fundamental de la API de WayPersistence es destacar esta capa de acceso a datos como uno de las más ligeras con apenas dependencias, excepto los jars de los propios drivers a utilizar según el sistema gestor de base de datos elegido. Esto hace a este framework uno de los más ligeros con 126k de tamaño. Por ejemplo, el ORM más conocido de todos, Hibernate en su versión 3 ocupa casi 3 MB y por encima de 2 MB todas sus dependencias.

Destacamos de WayPersistence su capacidad para realizar consultas de muchos tipos con relaciones, su potente dataset con posibilidad de deshacer consultas anteriores y su simplicidad y eficiencia por encima de todo. WayPersistence pretende ser una alternativa a motores de persistencia como pueden ser Cayenne, Ammentos, iBatis, etc.

Mapeado de Tablas en Registros (Clases)

WayPersistence representa una clase java por cada una de las tablas de la base de datos, independientemente del sistema gestor de bases de datos que se esté utilizando. Esta característica de poder tratar toda la capa de persistencia mediante objetos permite un mejor control sobre los datos y una mejor fluidez y simplicidad para el desarrollo de cualquier proyecto de software realizado en Java. Cada una de las clases, que definimos en simetría con las tablas o entidades del modelo de entidad-relación, son instanciadas como objetos java con sus propiedades, que equivalen a las columnas de la base de datos. Estos objetos se declaran como constantes y son registros. Por ejemplo cualquiera de las clases java definidas en el paquete com.way.persistence.dao:

```
public class Usuario extends RegistroInstancia {
        public static final RegistroMetaDato<Usuario> USUARIO = new RegistroMetaDato<Usuario>(
                        Usuario.class, "USER");
        public static final CampoCadena IDUSER =
                new CampoCadena(USUARIO, "ID", 3, CampoConstraint.CLAVE_PRIMARIA);
        public static final CampoCadena NOMBRE =
                new CampoCadena(USUARIO, "NOMBRE", 40, NO_NULO, DESCRIPTIVO);
        public static final CampoCadena APELLIDOS = new CampoCadena(USUARIO, "APELLIDOS", 50);
        public static final CampoCadena PASSWORD = new CampoCadena(USUARIO, "PASS",9);
        public static final CampoCadena ROLE_ID = new CampoCadena(USUARIO, "ROLE_ID", 3);
```

```
        public static final Referencia<Role> ROLE =
                new Referencia(USUARIO, Role.ROLE, "USER_REFERENCE_ROLE_ID");
        public @Override() RegistroMetaDato<Usuario> getMeta() {
                return USUARIO;
        };
}
```

En este ejemplo podemos apreciar que Usuario es un registro que representa la tabla Usuario de la base de datos. Cuando definimos RegistroMetaDato le estamos dando un nombre al objeto que será instanciado y haciendo referencia al nombre que tiene asignado en base de datos, en este caso, "USER". A continuación definimos los atributos de cada uno de los registros que se tendrán simetría con las propiedades de la tabla User de la base de datos. Así, si un tipo de datos como "NOMBRE" es definido en WayPersistence como CampoCadena (tipo de dato propio del framework que equivale a un tipo de dato de tipo String), en la base de datos debería ser de tipo cadena. Es decir, la representación del modelo de objetos tiene que ser casi igual que la del modelo relacional en cuanto a tipo de datos se refiere. Además de tipos de datos propios como "CampoCadena", destacamos CampoEntero (no representado en este ejemplo), Referencia (que equivale a un campo que hace referencia a la clave primaria en otra table, permite hacer una foreign key), y otros tipos de datos que iremos viendo.

Una prueba de funcionamiento de uno de estos registros es una inserción y una consulta sobre la inserción, como por ejemplo:

```
LOGGER.log(Level.INFO, "---------- Test de Empleado ---------- ");
SesionManager sessionActual = SesionManager.getThreadLocalSession();
sessionActual.getLogger().enableDebug();
// Comenzamos la session actual
sessionActual.begin();
CargaDatos usuarios = new CargaDatos(sessionActual, Usuario.USUARIO);
Object[][] usuariosVarios = new Object[][]{
        { "100", "Ana", "Lopez", "analop", "100"},
        { "200", "Jorge", "Antunez", "jorantun", "200"},
        { "300", "Pedro", "Alvarez", "pe300", "300"},
};
usuarios.insertarRegistros(usuariosVarios);
sessionActual.commit();

sessionActual.begin();
Usuario user = sessionActual.mustFind(Usuario.USUARIO, "100");
String nombre = user.getString(Usuario.NOMBRE);
LOGGER.log(Level.INFO, "El nombre del usuario encontrado es: " + nombre);
sessionActual.commit();
```

el resultado es: INFO: El nombre del usuario encontrado es: Ana

Es decir, aseguramos que en este método empezamos una sesión, modificamos los datos insertando 3 registros y hacemos un commit. Posteriormente buscamos con la sentencia ".mustFind" uno de los registros insertados a partir de su clave primaria. A partir de eso, obtenemos el objeto y su nombre y lo mostramos.

Podemos destacar que la definición de un registro, la inserción y estas operaciones básicas que hemos probado a realizar hace del ORM un motor de persistencia sencillo y fácil, en el que no hay que escribir grandes líneas de código para poder hacer una consulta. Además, no necesita de tediosos archivos XML y sus configuraciones que en alguno de los frameworks de persistencia más conocidos suponen cierto calvario para los usuarios noveles.

Alguna de las características en comparación con frameworks como Hibernate son las siguientes.

- Se puede acceder al objeto en cuestión de un registro, es decir, a una propiedad de la base de datos, por ejemplo el APELLIDO de un usuario, sin necesidad de crear objetos get y set. Se utiliza una sentencia como la que a diferencia a como sería por ejemplo en Hibernate con la sentencia "CreateQuery".

```
WayQuery consulta = sessionActual.new WayQuery(Usuario.USUARIO);
consulta.gt(Usuario.APELLIDOS, "B");
```

- También nos permite poner un valor del campo a nulo, a válido o incluso validarlo.
- Podemos también obtener una instancia de un objeto que tenga claves foráneas y consultar por ejemplo un usuario que pertenezca aun departamento y traer solo los datos del usuario forzando con un bloqueo optimista.
- La clase WayException permite evitar un fallo grave en caso de inconsistencia de datos o problemas en alguna consulta, inserción, etc. Esa clase de control de excepciones nos cubrirá en caso de que alguno de los fallos más comunes ocurran, bien sean de errores internos en el propio framework, de jdbc o de un bug.
- Todos y cada uno de los registros definidos en el framework, se almacenan en el dataset para tener una mayor consistencia de datos al realizar operaciones básicas.
- Los registro pueden ser accedidos sin tener una conexión a la base de datos, a diferencia de los POJOs de Hibernate.

Una potente virtud de WayPersistence es la posibilidad de realizar test unitarios, por ejemplo con JUnit si necesidad de tener que estar conectado a la base de datos. Esto permite una mayor rapidez y una eficiencia considerable para un equipo de desarrollo. Permite ahorrar tiempo de desarrollo y por lo tanto recursos.

WayPersistence solo almacena un registro por cada transacción y permite separar la lógica de negocio o la capa "service" siguiendo el modelo Vista – Controlador de la capa de datos que estaría ocupada con un real ORM como WayPersistence. Además, WayPersistence es código abierto.

Ahora procedo a poner un ejemplo concreto tanto de definición de registro como de las tareas más básicas que se suelen realizar con un registro dentro de este ORM.

```
package dao;

import com.way.persistence.database.CampoCadena;
import com.way.persistence.database.CampoConstraint;
import com.way.persistence.database.RegistroInstancia;
import com.way.persistence.database.RegistroMetaDato;

/**
 * La clase que representa la tabla Role
 */
public class Role extends RegistroInstancia {
        public static final RegistroMetaDato <Role> ROLE
    = new RegistroMetaDato(Role.class, "ROLE");

        public static final CampoCadena ROLE_ID
    = new CampoCadena(ROLE, "ROLE_ID", 3, CampoConstraint.CLAVE_PRIMARIA);
```

```
        public static final CampoCadena NOMBRE_ROL
    = new CampoCadena(ROLE, "ROLE", 30, CampoConstraint.DESCRIPTIVO);

        public static final CampoCadena DESCRIPCION
    = new CampoCadena(ROLE, "DESCRIPCION", 70);

        public @Override() RegistroMetaDato<Role> getMeta() {
                return ROLE;
        };

        / /     CREATE TABLE "ROLE"(
        / /     "IDROLE" VARCHAR(3) NOT NULL,
        / /     "ROLE" VARCHAR(30) NOT NULL,
        / /     "DESCRIPCION" VARCHAR(70) NULL,
        / /     PRIMARY KEY("IDROLE"));
}
```

En este ejemplo destacamos varias cosas. Por un lado, definimos una clase Role que extiende de RegistroInstancia (Clase padre de la definición de registros). A partir de ella, definimos una clase o registro que representará la tabla Role en el modelo relacional para cualquier base de datos o sistema gestor. Por un lado definimos la constante RegistroMetaDato con el nombre del objeto que hará referencia al nombre de la tabal, en este caso se llamarán igual "ROLE". Luego definimos las propiedades o atributos de Role, ROLE_ID, NOMBRE_ROL, DESCRIPCION. En este caso, todos los campos los hemos considera de tipo cadena, pero podrían haber sido de tipo Double, Entero, etc. Como comentario del código java aparece la sentencia SQL que definimos en la base de datos para ver las diferencias y sobre todo saber el nombre de los campos y sus tipos de datos para que no haya problemas en ese sentido. Destacamos las propiedades de tipo "CampoContraint" que tienen algunos de los atributos que significa que poseen alguna característica a mayores definidas normalmente en lenguajes de bases de datos como Constraints, en este caso de tipo Descriptivo y Clave Primaria para el identificador del Role.

En este caso mapeamos la tabla ROLE con este registro. Tiene 3 campos uno de ellos es clave primaria. No posee ninguna referencia a ninguna tabla, o clave foránea, cosa que no podemos decir del ejemplo de páginas más atrás, que posee una referencia o clave foránea a este registro definido ahora.

La manipulación de los datos con los objetos registros.

```
        /**
         * Poblamos los roles en BD
         */
        static void poblamosRoles() throws Exception {
                LOGGER.log(Level.INFO, "---------- deptTest ---------- ");
                SesionManager ses = SesionManager.getThreadLocalSession();
                ses.getLogger().enableDebug();
                ses.begin();

                / / Creamos algunos Departamento utilizando CargaDatos
                CargaDatos roles = new CargaDatos(ses, Role.ROLE);
                Object[][] rolesVarios = new Object[][]
                      {{"100", "Admin", "Administrador"},
                                      { "200", "Usuario", "Usuario final" },
                                      { "300", "mantenimiento", "Mantenimiento" } };
                roles.InserccionMultiple(ses, Role.ROLE, rolesVarios);

                ses.commit();
        }

        /**
         * Poblamos los Usuarios relacionandolos con al menos un rol (no obligatorio)
         * */
        static void poblamosUsuarios() throws Exception {
                LOGGER.log(Level.INFO, "---------- Test de Empleado ---------- ");
                SesionManager sessionActual = SesionManager.getThreadLocalSession();
```

```
                sessionActual.getLogger().enableDebug();
                // Comenzamos la session actual
                sessionActual.begin();
                CargaDatos usuarios = new CargaDatos(sessionActual, Usuario.USUARIO);
                Object[][] usuariosVarios = new Object[][]{
                        { "100", "Ana", "Lopez", "analop", "100"},
                        { "200", "Jorge", "Antunez", "jorantun", "200"},
                        { "300", "Pedro", "Alvarez", "pe300", "300"},
                };
                usuarios.insertarRegistros(usuariosVarios);
                sessionActual.commit();

                sessionActual.begin();
                Usuario user = sessionActual.mustFind(Usuario.USUARIO, "100");
                String nombre = user.getString(Usuario.NOMBRE);

                LOGGER.log(Level.INFO, "El nombre del usuario encontrado es: " + nombre);
                sessionActual.commit();

        }
```

En este caso lo que hemos hecho ha sido poblar la tabla de roles y partir de ella, poblar la tabla de usuarios, por supuesto, para poblar la tabla de usuarios, tiene que haberse poblado la tabal de roles para relacionar a un usuario con un rol.
Destacamos en estas líneas la utilización de "CargaDatos", la que utilizamos para insertar registros en base de datos pasándole dos parámetros: la session y un RegistroMetaDato que será el nombre del registro, en este caso Usuario.USUARIO. Además, probamos a encontrar un usuario encontrado y a mostrar su nombre como hicimos anteriormente.

Para realizar un borrado de un registro con el ORM actual, mostramos un ejemplo:

```
/**
         * Consulta un usuario por id
         */
        static void borrarUsuarioPorId(String idUsuario) {
                SesionManager session = SesionManager.getThreadLocalSession();
                session.begin();

                Usuario user = session.mustFind(Usuario.USUARIO, idUsuario);
                user.deleteRecord();
                LOGGER.log(Level.INFO, "Usuario borrado");
                session.commit();
        }
```

En este caso, creamos la sesion, comenzamos la transacción y obtenemos un usuario que exista en el sistema (en base de datos), invocamos a "deleteRecord" y realizamos un borrado lógico del registro en base de datos. Hacemos commit y solucionada la acción de borrar.

Si por el contrario, nos interesa crear un nuevo usuario si necesidad de introducir gran cantidad de registros, podemos utilizar el método Inserccion implementado en la clase CargaDatos o bien podemos utilizar la manera que representamos en este ejemplo:

```
static void crearNuevoUsuario(String idUsuario, String nombre, String apes, String pass, String role) {
                SesionManager session = SesionManager.getThreadLocalSession();
                session.begin();

                CargaDatos usuarios = new CargaDatos(session, Usuario.USUARIO);
                Object[] user = new Object[]
                        { idUsuario, nombre, apes, pass, role};
                usuarios.insertarRegistro(user);

                session.commit();
        }
```

Esta es una manera similar o prácticamente igual, pero se invocan de distinta manera. Al igual que el borrado, cogeríamos el hilo actual, comenzaríamos una nueva transacción, cargamos los datos con el metadato o nombre del registro a modificar y la sesion. Creamos

un array de objetos que almacenará los valores a insertar en base de datos. Posteriormente, insertamos el registro en base de datos y hacemos un commit.

Consultas

La clase fundamental de las consultas en el ORM es WayQuery la que puede ser ejecutada en la session para realizar cualquier tipo de consulta en la base de datos.

Por ejemplo, siguiendo con el mismo ejemplo, para poder sacar un listado de usuarios con su nombre y apellidos ordenados de forma descendente por su nombre sería:

```
/**
* Listado de usuarios
*/
static void listadodeUsuarios() {
                        SesionManager session = SesionManager.getThreadLocalSession();
                        session.begin();
                        WayQuery<Usuario> consultaListado = new WayQuery(Usuario.USUARIO).descending(Usuario.NOMBRE);
                        List<Usuario> listadoUsuarios = session.query(consultaListado);
                        Iterator<Usuario> iterador = listadoUsuarios.iterator();
                        while (iterador.hasNext()) {
                                Usuario usuario = (Usuario) iterador.next();
                                LOGGER.log(Level.INFO, "Usuario: " + usuario.getString(Usuario.NOMBRE) + " " +
                                                usuario.getString(Usuario.APELLIDOS));
                                }
                        session.commit();
}
```

En este ejemplo observamos que definimos un objeto de tipo WayQuery, este objeto será la consulta en cuestión, invocamos a descending para ordenar deforma descendiente esta consulta por nombre. Desde la session invocamos a Query pasándole por parámetro esta consulta que nos devolverá un listado de usuarios. Creamos un iterador para recorrer la lista y así ir mostrando registro a registro de la base de datos con su nombre y apellidos. Hacemos commit y visualizaremos algo como:

```
INFO: Usuario: Pedro Alvarez
INFO: Usuario: Juan Lopez
INFO: Usuario: Ana Lopez
```

Cuando realizamos una consulta, como el caso anterior, no es necesario devolver todos los valores, puesto que sólo queremos mostrar el nombre y los apellidos del objeto. Se utilizan sentencias como rawQueryMap o rawQueryMaps definidas en la session.

Manejo de transacciones en WayPersistence

Para realizar una transacción en este ORM se tienen que seguir una serie de pasos. En primer lugar, se tiene que abrir un datasource a una base de datos, es decir, establecer una conexión a la base de datos o sistema gestor de bases de datos utilizado. En nuestro caso concreto, en WayPersistence, se obtiene el hilo que está actualmente en sesión. Posteriormente, se comienza la sesión con la sentencia “begin()” y se realizan las sentencias que creamos convenientes. Después de haber realizado las operaciones básicas sobre la propia transacción, introducir o crear datos en base de datos, borrado, consultas, deshacer operaciones con flush(), etc. Una vez que estemos conformes con los datos modificados que permanecen en memoria caché, debemos confirmarmos. Para ello, con la sentencia “commit()” confirmamos los cambios en nuestra base de datos, y finalizamos la transacción con close() o cierre. Ejemplo:

```
public static void main(String[] argv) throws Exception {
        CasosDePrueba.initializeTest(DemostracionORM.class);
        try {
                LOGGER.log(Level.INFO, "Estamos Inicializando ");
                pruebaInicia();
                poblamosRoles();
                poblamosUsuarios();
                Usuario usuario = obtenerUsuarioPorId("100");
                if (usuario != null) {
                        LOGGER.log(Level.INFO, "No existe ningun usuario con ese id");
                }
                borrarUsuarioPorId("200");
                crearNuevoUsuario("400","Juan", "Lopez", "juan8", "300");
                listadodeUsuarios();
        } finally {
                SesionManager.getThreadLocalSession().cerrar();
        }
}
```

Las manipulaciones de bases de datos se manejan en el paquete "com.way.persistence.engine". En este paquete tenemos dos clases principales que se encargan de manejar y gestionar el proceso transaccional del framework u ORM. Estas dos clases que se encargan de esta serie de operaciones son: SessionManager y SessionJdbcInterna. Estas dos clases contienen la configuración JDBC. Siempre que realizamos alguna operación en una sesión, se asociará al hilo actual en el que estemos operando, por ejemplo en nuestro caso sería: SesionManager.getThreadLocalSession(). En un principio cada sesión se corresponderá con un único hilo.

Si invocamos varias veces a un mismo registro de la base de datos en la misma sesión, permanecerá una única vez en sesión con una instancia de ese registro sin repetirse ninguna vez más que la propia instancia. Si consultamos el mismo registro a la vez desde varias sessiones, se permitirá tendrá que esperar debido al bloqueo optimista para poder acceder a dicha instancia si ya está siendo utilizada por otra operación de otra sesión.

Los registros que permanezcan en memoria caché serán limpiados de la misma una vez hayamos hecho commit() de la transacción en el orden que fueron insertados en la sesión. WayPersistence nos permite además, hacer un barrido o limpiado de la memoria caché de algunas operaciones que queremos desechar por el motivo que sea.

Cumpliendo el Patrón ACID en las transacciones

El patrón ACID garantiza la atomicidad, coherencia, aislamiento y durabilidad.

El framework WayPersistence garantiza transacciones independientes con aislamiento entre ellas, permite el bloqueo de registros y asegura una integridad de la base de datos multitransaccional.

Si se produce algún tipo de problema en la creación de alguna sentencia de consulta SQL, se lanza una excepción y cancela la transacción. Este tipo de excepciones puede ser evitado especificando ModoConsulta.FOR_UPDATE.

Este ORM no tiene activado la opción de commit automático, puesto que no se considera necesario y puede ser un riesgo para las transacciones. Pare evitar problemas con las transacciones simultáneas, debemos utilizar FOR_UPDATE, de esta forma, aseguraremos que se bloque una transacción si estamos accediendo a datos que están siendo utilizados

por otra transacción. Es decir, antes de acceder a datos que estén siendo utilizados por otra transacción, esperamos a que se haga commit de esa última transacción, para poder proceder a retomar la transacción anterior que está esperando a que termine la actual. Así pues, destacamos que FOR_UPDATE realiza un bloqueo para permitir realizar una única transacción.

Operaciones con DataSet

A parte del paquete engine que posee toda las clases que controlan como su nombre indica el motor de la aplicación, las consultas, los drivers, principalmente a nivel genérico, y por supuesto, como comentamos en líneas anteriores la sesión actual y sus operaciones habituales.

Por otra parte, tenemos otro paquete que es de drivers, que extienden de la clase driver definida también a nivel de paquete "…engine". Además de estos paquetes, tenemos el paquete dao que contiene la definición de registros que mapean la representación de tablas del modelo entidad-relación en objetos java. También tenemos el paquete "..service" y el "…utilidades": el primero de ellos contiene la "lógica de negocio" del ORM, es decir, las operaciones más alejadas de la propia persistencia en la que se representan pruebas de concepto para el funcionamiento y puesta a punto de WayPersistence. El segundo de ellos, contiene una clase genérica de constantes, clase mejorable en futuras versiones donde se agruparán todas las constantes genéricas del ORM, una clase excepciones y otras de control de Log. Todas ellas, ayudan a mejorar y reforzar las utilidades de esta herramienta.

Por último, tenemos dos paquetes aún no mencionados: validation y database. El primero de ellos contribuye a que todos los tipos de datos estén validados en función de cómo lo definamos. El segundo, el paquete "com.way.persistence.database" junto con el paquete "com.way.persistence.engine" son los dos pilares básicos del ORM. "…database" contiene la mayor parte de los tipos de datos personalizados y utilizados en esta capa de persistencia. Además de los tipos de datos comunes, se posee ObjetoRaiz que es una clase genérica de la que dependen todos los tipos de datos, es decir, de la que extienden, y a su vez, ella extiende de MetaDato. En ObjetoRaiz se almacenan todos los datos de tipo genérico como CampoCadena, CampoEntero, etc. Cada instancia de un metaDato se almacena e un RecordMetaDato. RegistroMetadato define el metadato como por ejemplo el nombre de la tabla. RegistroInstancia. Cada RegistroInstancia representa un reigstor individual en memoria que corresponde a una fila de la base de datos o un un registro que será insertado. Esta instancia de registros es creada por sessionJdbc, de aquí la unión entre los dos "módulos" del ORM. Además, WayQuery es la principal clase de consultas que posee WayPersistence. Permite crear instancia de WayQuery y actualizar cada una de los métodos basados en el lenguaje SQL. Esto se ayuda del MotorQuery para realizar todas las Queries del ORM.

Ejemplo:

```
/**
 *      Consulta un usuario por id
 */
static Usuario obtenerUsuarioPorId(String idUsuario) {
                        SesionManager session = SesionManager.getThreadLocalSession();
                        session.begin();
                        Usuario usuario = session.mustFind(Usuario.USUARIO, idUsuario);
                        LOGGER.log(Level.INFO, "Mostramos el usuario buscado: " +
                        usuario.getString(Usuario.NOMBRE));
                        session.commit();
```

```
        return usuario;
}
```

En la carpeta build tenemos un jar con el ORM empaquetado, en este caso se tiene como un todo porque se quiere probar la funcionalidad y la eficiencia de WayPersistence. Se podría optar por separar los paquetes para utilizar de forma independiente, por ejemplo, utilizar el database se podría utilizar sin necesidad de un gestor de bases de datos, por ejemplo para probar con test unitarios.

Relaciones entre registros de distintas tablas

Para realizar las relaciones entre distintas tablas utilizamos la clase Referencia. Esta clase hará que en el modelo de datos orientado a objetos se haga referencia a otra tabla (clase o registro en este caso) para unir varias tablas. En el ejemplo que se está utilizando hasta ahora, Usuario tendría una propiedad de tipo Referencia que apuntaría a la clave primaria de el registro Role. Esto representaría las claves foráneas del modelo relacional de la base de datos en cuestión.

```
public static final Referencia<Role> ROLE =
        new Referencia(USUARIO, Role.ROLE, "USER_REFERENCE_ROLE_ID");
```

La forma habitual de encontrar todos los usuarios con un determinado role sería:
Role role = usuario.findRefrence(usuario.roleid);
List<Usuarios> listadoUsuarios = dataset.queryReferencing(role, Usuario.ROLE);

Este manual es una guía para poder aprender las funcionalidades básicas y el manejo simple y básico del framework de persistencia WayPersistenceORM. En versiones posteriores se procederá a actualizar este manual y aportar mayores características y aportaciones que nos puede aportar este motor de persistencia nuevo, sencillo y por supuesto, funcional.

ANEXO III. GUÍA RÁPIDA PARA UTILIZAR WAYPERSISTENCE

Para utilizar el framework de persistencia WayPersistence en un proyecto cualquiera, se debe tener en cuenta lo siguiente:

1. Colocar el fichero way.persistence.properties en el home del usuario del equipo en el que estemos /../home (Linux) o C:/Documents And Settings/<nombreusuario>/ en Windows. Este fichero tendrá las especificaciones de la base de datos, será como el fichero de configuración de nuestro motor de persistencia. Ejemplo:

   database.driver=com.mysql.jdbc.Driver
   database.url=jdbc:mysql://localhost:3306/demo
   database.username=root
   database.password=root

   Siendo database.driver el driver inyectado como jar en nuestro proyecto, es decir, el gestor de bases de datos escogido.
   Url: la dirección a donde apunta la base de datos y el esquema de la misma, en este caso es una base de datos denominada “demo”.
   Además el usuario y la contraseña.

2. Una vez configurado el fichero properties con los datos que conciernen a nuestro proyecto, procedemos a inyectar el jar necesario correspondiente a nuestro sistema gestor utilizado, en este caso MySQL.

3. Inyectamos el jar del propio motor de persistencia en el proyecto que lo queramos utilizar. En nuestro caso, la mini – aplicación, proyecto Java desarrollado con Eclipse, “PruebaWayPersistenceORM”. El jar “WayPersistenceORM.jar” generado por el proyecto java “WayPersistence” por una de sus tareas del “ant” denominada “jar”.

4. Una vez realizado esto pasamos a probar el proyecto en la mini – aplicación citada anteriormente.