Capitulo III.

Modelo de Datos

El Sistema Asistente para la Generación de Horarios de Curso tiene un dominio de datos complejo pero lo suficientemente bien limitado como para poder manejarlos de una manera ordenada. A continuación se explicará de manera simplificada el proceso para obtener el contexto completo de los datos a manejar.

Como ya bien sabemos, el propósito de este proyecto es el de asistir en la asignación de horarios a las secciones de las materias que se impartirán un determinado semestre. Analizando un poco más detenidamente podemos observar que un horario de curso no sólo está determinado por una hora de inicio y una hora de fin, sino que también es necesario determinar los días en los que será impartida la materia y a su vez es necesario involucrar un salón (en donde será impartido el curso), un profesor (el cuál impartirá las clases) y como es obvio, una materia. Hasta aquí tenemos tres grandes entidades: Profesor, Salón y Sección.

Dado el contexto de estudios universitarios, una Sección es una entidad derivada de una Materia y como tal, es necesario modelar la Materia independientemente ya que muchas Secciones pueden ser abiertas de la misma Materia.

Hasta aquí podemos modelar los elementos que conformar un Curso, pero además es necesario cumplir con las restricciones planteadas en el Capítulo I sobre traslapes de horarios. Recapitulando: Una Sección no puede tener traslape de horario con otra que se lleve a cabo en el mismo salón, y a su vez el Profesor no deberá tener problemas de la misma índole con otra Sección que imparta (en el mismo salón o cualquier otro).

Estas primeras dos restricciones pueden ser cumplidas con el modelo de datos que ya tenemos hasta el momento ya que la entidad Sección contiene toda la información necesaria para validar los requerimientos; sin embargo aún quedan restricciones pendientes por cumplir: No deben existir dos secciones de diferentes materias de un mismo semestre de un mismo plan de estudios que se traslapen en horario impidiendo así que un alumno pueda inscribirse a ambas secciones como es requerido por su plan de estudios.

Para entender el enunciado anterior es necesario comprender la estructura de los Planes de Estudio (una nueva entidad a modelar) y para esto empezaremos por una Carrera, por ejemplo Ing. En Sistemas Computacionales. Esta carrera pertenece al Departamento de Computación, Electrónica y Mecatrónica y a su vez tiene 2 Planes de Estudio vigentes: 2002 y 2006. Cada Plan de Estudios está dividido en semestres y cada semestre contiene un número fijo de Materias que el alumno debería cursar (según el semestre en el que vaya) para seguir a pie de letra el Plan de Estudios en el que está inscrito.

En resumen necesitamos modelar las siguientes entidades: Carrera, Plan de Estudio, Materia, Sección, Salón y Profesor. Finalmente se agregaron las entidades *Horario* y *Departamento* para cumplir con propósitos de usabilidad y extensibilidad. La entidad *Horario* tiene el propósito de almacenar las preferencias de horarios que los profesores para impartir sus clases. La entidad *Departamento* funciona como un simple contenedor para las múltiples instancias de la entidad *Carrera*.

3.1 Hibernate

Anteriormente se habló del Modelo como parte esencial de la arquitectura *MVC* y se dijo que el Modelo es la capa que contiene los datos y los procesa a petición del usuario para obtener la información necesaria que será desplegada finalmente al usuario. Es decir,

el Modelo es la parte que se encarga de los datos, y como tal, necesita mantenerlos accesibles de manera sencilla para su almacenamiento y recuperación. Con tal propósito se buscó una plataforma que facilitara las funciones de persistencia de datos en la Base de Datos y que permitiera transparentemente trabajar siempre con objetos de Java.

"Hibernate es un poderoso, eficiente sistema de persistencia y recuperación de objetos y relaciones [que] permite desarrollar clases persistentes siguiendo el paradigma de Programación Orientada a Objetos, incluyendo asociación, herencia, polimorfismo, composición y colecciones." (Red Hat, 2007).

Hibernate, en pocas palabras, es un sistema de persistencia de objetos a través de mapeo de variables en un archivo XML[6]. Para entender mejor el funcionamiento de Hibernate analicemos el siguiente diagrama:

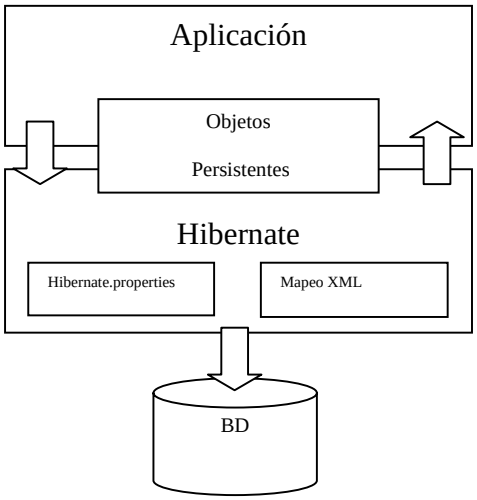

Diagrama 2 Arquitectura simplificada de Hibernate

Como podemos apreciar, la aplicación sólo se preocupa de mandar los objetos al *framework* de Hibernate, el cuál tiene la responsabilidad de persistir de manera correcta los

[6] También es posible realizar los mapeos directamente en el código Java haciendo uso de Java Annotations.

objetos en una Base de Datos Relacional (como lo es MySQL), y de igual manera los recupera para que la aplicación pueda manipularlos nuevamente.

Esta simple explicación encapsula el propósito completo de usar Hibernate en una aplicación, ya que el programador no se tiene que preocupar por desarrollar mecanismos de persistencia y recuperación y puede continuar trabajando en modelos de objetos representando todas las entidades persistentes que desee así cualquier tipo de relaciones.

En el diagrama anterior, el recuadro nombrado “hibernate.properties” representa las propiedades de configuración pertinentes a Hibernate, mientras que el recuadro “Mapeo XML” hace referencia a los archivos que utiliza Hibernate para saber qué objetos persistir, así como especificaciones de relaciones con otros objetos[7].

A continuación se mostrará el código fuente de la entidad *Materia* junto con su correspondiente archivo de mapeo XML:

```
package db.entities;

import java.util.*;

public class Materia {

    private int id;
    private Carrera carrera;
    private int clave;
    private String nombre;
    private int unidades;
    private Map<PlanDeEstudios,Integer> planesDeEstudio = new HashMap<PlanDeEstudios,Integer>();

    public Materia(){

    }

    public Materia(Carrera carrera, int clave, String nombre, int unidades){
        this.setClave(clave);
        this.setNombre(nombre);
        this.setUnidades(unidades);
        this.setCarrera(carrera);
        carrera.addMateria(this);
    }

    public int getClave() {
        return clave;
    }

    public void setClave(int clave) {
        this.clave = clave;
    }
```

[7] Para más información acerca de los mapeos XML consulte el manual de Hibernate en http://www.hibernate.org/hib_docs/v3/reference/en/html/

```
    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getUnidades() {
        return unidades;
    }

    public void setUnidades(int unidades) {
        this.unidades = unidades;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String toString(){
        return this.getCarrera().getId() + this.getClave() + " (" + this.getNombre() + ")";
    }

    public Carrera getCarrera() {
        return carrera;
    }

    public void setCarrera(Carrera carrera) {
        this.carrera = carrera;
    }


    public void addPlanDeEstudios(PlanDeEstudios plan, int semestre){
        this.getPlanesDeEstudio().put(plan, new Integer(semestre));
        plan.getMaterias().put(this, new Integer(semestre));
    }

    public Map<PlanDeEstudios, Integer> getPlanesDeEstudio() {
        return planesDeEstudio;
    }

    public void setPlanesDeEstudio(Map<PlanDeEstudios, Integer> planes) {
        this.planesDeEstudio = planes;
    }


}
```

Código fuente 1 Materia.java

Como se puede apreciar, una entidad es simplemente una clase normal de Java que debe cumplir con los requisitos mínimos de un Java Bean (tener un constructor vacío y métodos *setters* y *getters*) además de métodos asistentes (como *addPlanDeEstudios()* el cual se asegura de crear vínculos bi-direccionales entre las clases relacionadas).

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="db.entities.Materia" table="materia"  >
        <id name="id" type="integer" column="materia_id">
            <generator class="native" />
        </id>

        <properties name="carrera_clave" unique="true">
```

```
            <many-to-one    name="carrera"
                            class="db.entities.Carrera"
                            column="carrera_id"
                            not-null="true"
            />
            <property       name="clave"
                            type="integer"
                            column="clave"
                            not-null="true"
            />
        </properties>

        <property       name="nombre"
                        type="string"
                        column="nombre"
        />

        <property       name="unidades"
                        type="integer"
                        column="unidades"
        />

        <map            name="planesDeEstudio" order-by="semestre, plan_id" table="Materias_Planes">
            <key        column="materia_id" />
            <map-key-many-to-many
                        class="db.entities.PlanDeEstudios"
                        column="plan_id"
            />
            <element    column="semestre"
                        type="integer"
            />
        </map>

    </class>
</hibernate-mapping>
```

Código fuente 2 Materia.hbm.xml

El archivo XML que se presentó anteriormente sigue las reglas establecidas por Hibernate referente a los mapeos de clases persistentes que establece las propiedades y relaciones con otras clases que deberán ser guardadas en la base de datos. Este archivo XML lo utiliza Hibernate como su guía para la generación del esquema de base de datos y deducir qué tipo de tablas serán necesarias para almacenar las relaciones.

3.2 Data Access Objects

Los *Data Access Objects* (*DAOs* por sus siglas en inglés) son otro patrón de diseño destinado a encapsular la información que se desea guardar o recuperar para mantener una separación de conceptos más cristalino. Los *DAOs* en este proyecto se utilizan como capa intermediaria entre Hibernate y la aplicación, y están encargados de manipular todas las transacciones de Hibernate para guardar y recuperar objetos de la base de datos.

Durante el diseño de la aplicación se decidió crear un *DAO* para cada entidad persistente, heredando un comportamiento genérico para todos los *DAOs* como el de guardar (actualizar) y eliminar objetos; sin embargo cada uno tiene funciones específicas particularmente desarrolladas particularmente para la recuperación de las entidades según parámetros de decisión.

El siguiente diagrama muestra la estructura de DAOs principales:

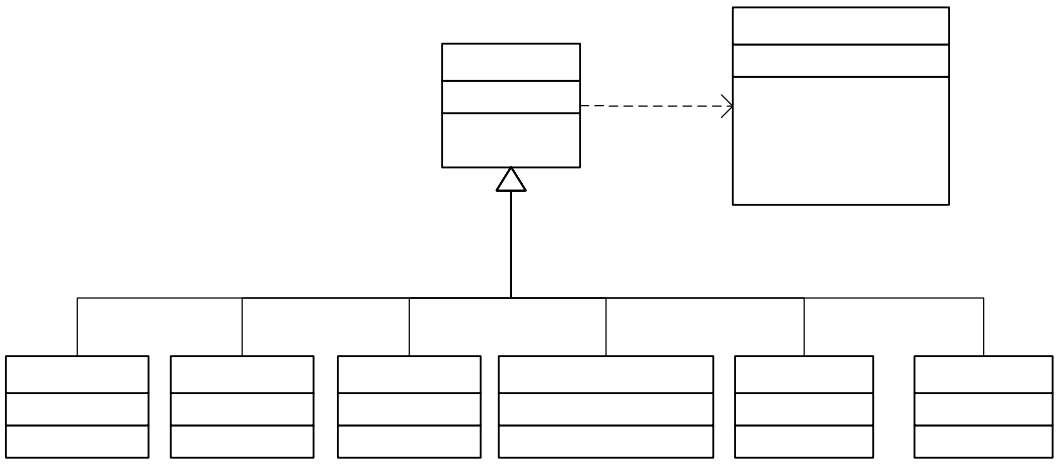

Diagrama 3 Arquitectura de DAOs

Cuando la capa de Modelo desea guardar una instancia de una entidad persistente (como Profesor, Salón, etc.) debe de comunicarse a través de un DAO para que éste solicite a Hibernate (a través de la clase de utilería HibernateUtil) el inicio de una transacción. Una vez abierta la transacción Hibernate (internamente) se encarga de convertir el objeto en enunciados SQL con lo que se guarda únicamente la información necesaria en la base de datos. Cuando todo el proceso se ha realizado con éxito la cadena de respuestas llega hasta la aplicación que solicitó el proceso de guardado para que pueda notificar (a través de la capa de Vista) al usuario del éxito de la operación. En el siguiente diagrama se representa el proceso completo de persistencia de un objeto:

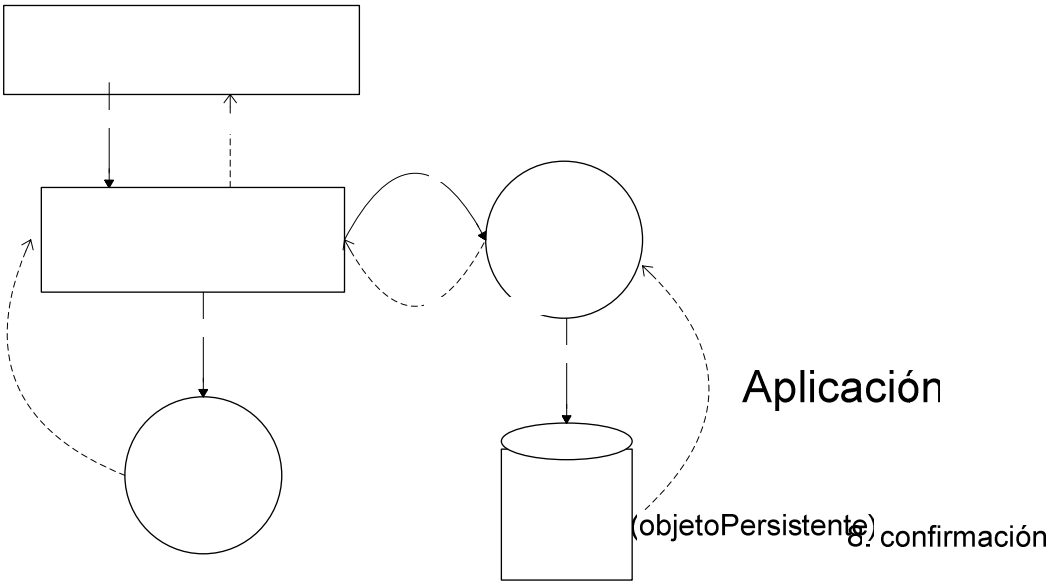

Diagrama 4 Persistencia en la BD

```
package db.DAO;

import db.entities.*;
import org.hibernate.Criteria;
import org.hibernate.criterion.Property;

public class MateriaDAO extends GenericDAO{

    public MateriaDAO() {
        super();
    }

    public Materia getMateriaById(int id){
        return (Materia) session.get(Materia.class, new Integer(id));
    }

    public Materia getMateriaByCarreraAndClave(Carrera carrera, int clave){
        Property ptyCarrera = Property.forName("carrera");
        Property ptyClave = Property.forName("clave");
        Materia result =
                (Materia) this.createCriteria()
                                .add(ptyCarrera.eq(carrera))        // Compara el valor de la carrera
                                .add(ptyClave.eq(clave))            // Compara el valor del año
                                .uniqueResult();                    // Ejecuta el query
        return result;
    }

    private Criteria createCriteria(){
        return session.createCriteria(Materia.class);
    }

}
```


Código fuente 3 MateriaDAO.java

En el archivo anterior podemos apreciar un ejemplo del poderoso sistema de lo que Hibernate llama *Criteria Queries* en el método *getMateriaByCarreraAndClave()*. Este método de consulta permite establecer reglas tipo SQL de manera programática; sin embargo este no es el único modo de consultar información en la base de datos. Hibernate

permite el uso de enunciados *SQL* directos o a través de *HQL* (*Hibernate Query Language)* o de *Example API*. Este último basa su funcionamiento en crear un objeto parecido al que deseamos encontrar y Hibernate se encargará de buscarlo.

3.3 Validators

Para asegurar el cumplimiento de las restricciones requeridas para el correcto funcionamiento del sistema se utilizan los objetos *Validators*. Dichos objetos sirven como filtros para decidir si una entidad persistente debe ser permitida que ingrese al modelo ya almacenado en la base de datos. Dada la función de los *Validators* es fácil pensar que estos objetos deberían de estar contenidos dentro de la capa del Controlador (a veces también llamada Lógica de Negocios), sin embargo los objetos en cuestión pertenecen al Modelo porque es esta capa la que debe regular la validez de la información que contiene.

El ciclo de vida de un *Validator* empieza cuando el controlado crea una nueva instancia de la clase que desea validar. Por ejemplo, si se desea guardar una nueva Sección, se deberá crear una nueva instancia de SecciónValidator y se le provee acceso al objeto para verificar que se cumplan con todas las restricciones necesarias (atributos requeridos, requerimientos de integridad, etc.). Si un objeto no cumpliese con las reglas impuestas por el *Validator*, el objeto que invocó su aplicación puede solicitar también la lista de errores que se encontraron durante su inspección para poder retroalimentar al usuario sobre los mismos. Si el *Validator* da su aprobación con el objeto que se desea salvar, se continúa con el proceso explicado anteriormente.

```
package db.validators;

import db.entities.Materia;
import java.util.Hashtable;

public class MateriaValidator {

    private Hashtable errors = new Hashtable();
    private Materia materia;
```

```
    private boolean valid = true;

    public MateriaValidator(Materia materia) {
        this.setMateria(materia);
    }

    public void validate(){
        checkRequired();
        if (this.isValid())
            this.checkIntegerValues();
    }

    public Hashtable getErrors() {
        return errors;
    }

    public void setErrors(Hashtable errors) {
        this.errors = errors;
    }

    public Materia getMateria() {
        return materia;
    }

    public void setMateria(Materia materia) {
        this.materia = materia;
    }

    private void checkRequired() {
        if (materia.getCarrera() == null){
            this.setValid(false);
            errors.put("carrera","No ha elegido una carrera para esta materia");
        }
        if (materia.getClave() == 0){
            this.setValid(false);
            errors.put("clave","No ha llenado este campo.");
        }
        if (materia.getNombre() == null || materia.getNombre().compareTo("") == 0){
            this.setValid(false);
            errors.put("nombre","No ha llenado este campo.");
        }
        if (materia.getUnidades() == 0){
            this.setValid(false);
            errors.put("unidades","No ha llenado este campo.");
        }
    }

    private void checkIntegerValues(){
        if (materia.getClave() < 1){
            this.setValid(false);
            errors.put("clave","La clave no puede ser menor o igual a 0 (cero)");
        }
        if (materia.getClave() < 100 || materia.getClave() > 999){
            this.setValid(false);
            errors.put("clave","La clave debe ser un n&uacute;mero entero de 3 d&iacute;gitos.");
        }
        if (materia.getUnidades() < 1){
            this.setValid(false);
            errors.put("unidades","Las unidades no pueden ser menor o igual a 0 (cero)");
        }
        if (materia.getUnidades() > 9){
            this.setValid(false);
            errors.put("unidades","Las unidades debe ser un dígitos entre 1 y 9.");
        }
    }

    public boolean isValid() {
        return valid;
    }

    public void setValid(boolean valid) {
        this.valid = valid;
    }

}
```

Código fuente 4 MateriaValidator.java

Como este archivo lo demuestra, el proceso de validación se realiza en etapas, siendo la primera la más sencilla por tratarse únicamente de validar campos requeridos. A continuación cada *Validator* se encarga de definir las reglas que los objetos deben de cumplir como restricciones de traslapes (en el caso de tratarse de horarios o asignaciones)

3.4 Hibernate en el Web

Hibernate es un *framework* que está diseñado para trabajar en cualquier aplicación y su funcionamiento está basado en transacciones (también llamadas *unidades de trabajo*), las cuales normalmente son realizadas en la capa de la Modelo de Datos únicamente (bajo el esquema *MVC)* ya que ninguna otra capa debe de tener acceso a la manipulación directa de los datos si no es a través de esta.

Como anteriormente fue explicado en ciclo de vida *MVC*, se empieza cuando el usuario solicita una acción por parte del servidor, los *Servlets* se encargan de recibir y procesar la petición (redirigiéndola a quien sea que le corresponda el trabajo), después se realiza la transacción en la base de datos (leer, guardar o actualizar datos) y finalmente se le regresan al controlador que despliega los resultados al cliente. Es importante mencionar que bajo este esquema, la transacción termina en cuanto el Modelo termina su trabajo; sin embargo Hibernate maneja objetos de tipo *proxy* para representar los objetos recuperados de la base de datos.

Los objetos *proxies* son representaciones de los objetos reales que cuando se requiere información específica de alguna entidad, se encargan de recuperarla dinámicamente. El propósito de usar *proxies* yace en hacer más eficiente el uso de memoria. Cuando se recupera un objeto de la base de datos usando Hibernate es posible que se recupere con ese objeto todo un árbol de relaciones con otros objetos (colecciones, composiciones, etc.) y

traer a la memoria todo la información es considerado altamente ineficiente además de peligroso ya que puede dejar un sistema sin recursos. Para solucionar el problema, los diseñadores de Hibernate decidieron utilizar un esquema llamado *Lazy Loading* que funciona como un *Load on demand* (Cargado bajo demanda). Cuando la aplicación requiere leer atributos de alguna instancia de la entidad – que en realidad es un objeto *Proxy* – éste se encarga de recuperarlo de la base de datos en el momento.

La idea es muy buena y en situaciones normales funciona de maravilla, sobre todo en el contexto de aplicaciones de escritorio, sin embargo en una aplicación Web, es posible que los objetos recuperados durante la consulta en la capa de *Modelo*, no se hayan "inicializado" (es decir, cargar todos los atributos del objeto) que serán utilizados posteriormente al ser presentada la información al usuario.

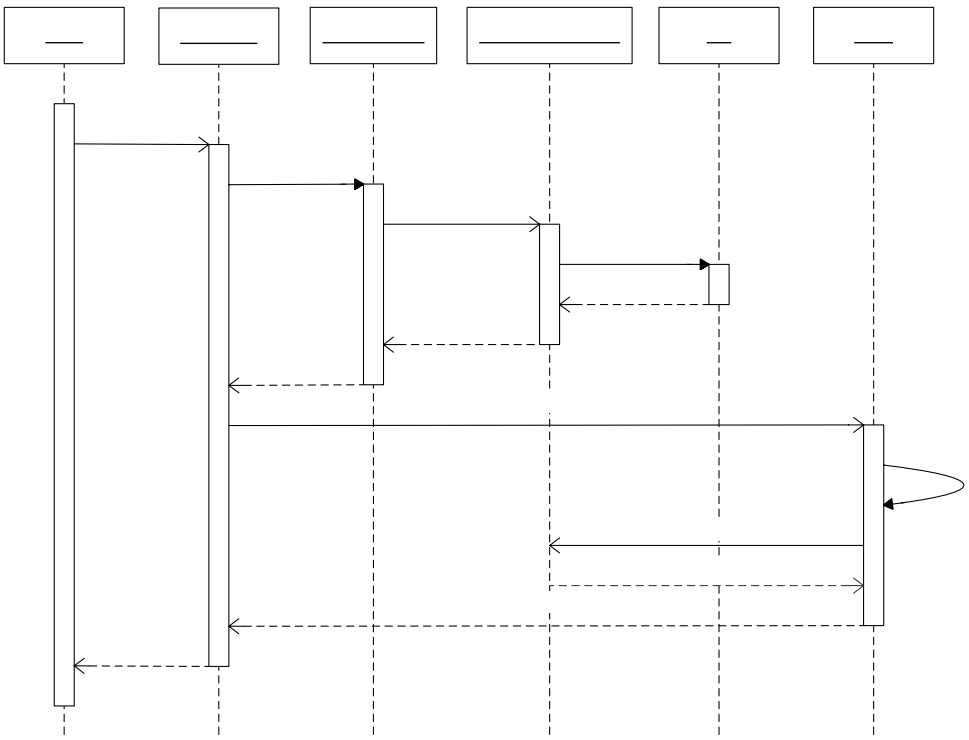

Diagrama 5 Secuencia normal de Hibernate en MVC

Como el diagrama claramente lo demuestra, la Vista es incapaz de recuperar los atributos del objeto tipo *Proxy* ya que en el momento en que los solicita, la unidad de trabajo con la base de datos terminó y se cerró la conexión que requiere el *Proxy*.

Algunas posibles soluciones a este problema son:

a) Cada vez que la Vista requiera inicializar la(s) entidad(es) en cuestión podrían solicitar que se abra una nueva sesión de Hibernate para que el *Proxy* pueda tener comunicación con la base de datos. Esta solución parece ser simple y sencilla de implementar, sin embargo rompería completamente la filosofía de separación de conceptos. La capa de Vista nunca debe de tener control sobre el acceso a la base de datos.

b) Inicializar completamente cada objeto que se transmite fuera de la capa de Modelo, de esta manear no es necesario que exista una sesión activa de Hibernate para recuperar los atributos requerido por la Vista. Nuevamente, esta solución no es viable para todos los escenarios (altos requerimientos de memoria) y anula el propósito del *Lazy Loading*; sin embargo es posible implementar esta solución para objetos que se sabe de antemano no tendrán relaciones complejas con otros objetos.

c) Mantener abierta la sesión de Hibernate hasta que se termina de crear la respuesta visual y es transmitida al usuario.

Al optar por la tercera solución, se tiene la ventaja de que sólo se requiere una sola conexión con la base de datos (a través de Hibernate) para realizar todas las transacciones que se requieran para completar la solicitud del cliente. Esta solución se clasifica como el patrón de diseño *Sesion-per-application-transaction* que a su vez utiliza el patrón *Chain of Command* para delegar responsabilidades.

Para implementar satisfactoriamente esta solución es necesario hacer uso de los *Servlet Filters*, los cuales son simplemente objetos que encapsulan todo la petición del cliente antes de que el Controlador inicie sus labores. Específicamente en este proyecto se creó el archivo *HibernateFilter.java*, el cuál se asegura de iniciar una sesión de Hibernate **antes** de que el procesamiento empiece y persiste todos los cambios en la base de datos (en caso de ser necesario) **después** de que se completa el ciclo de vida de *MVC* o deshace los cambios en caso de ocurrir un error irrecuperable relacionado con la base de datos.

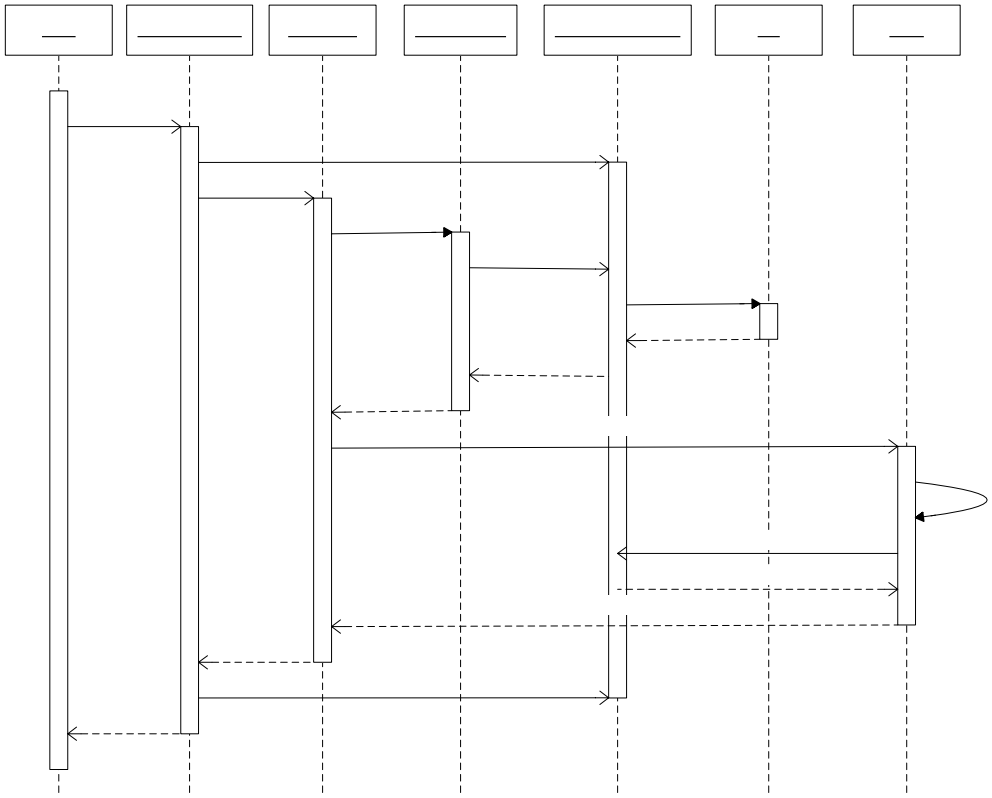

Diagrama 6 Secuencia de Hibernate usando Servlet Filters