UNIVERSIDAD POLITECNICA DE VALENCIA

Prácticas de Laboratorio

Práctica 3.
Arquitectura del Sistema: Capa de negocio

(Guía de práctica)

Ingeniería de software
ETS Ingeniería Informática
DSIC – UPV

Curso 2014-2015

Arquitectura de sistema - Capa de Negocio

Para el desarrollo del sistema propuesto, **Servicio de Llamadas de Emergencia**, adoptaremos la arquitectura de tres capas ya explicada en las clases de teoría. La Figura 1 muestra el esquema básico de dicha arquitectura, en el que se distinguen tres capas: **Persistencia**, **Negocio o Lógica de la Aplicación** y **Presentación**.

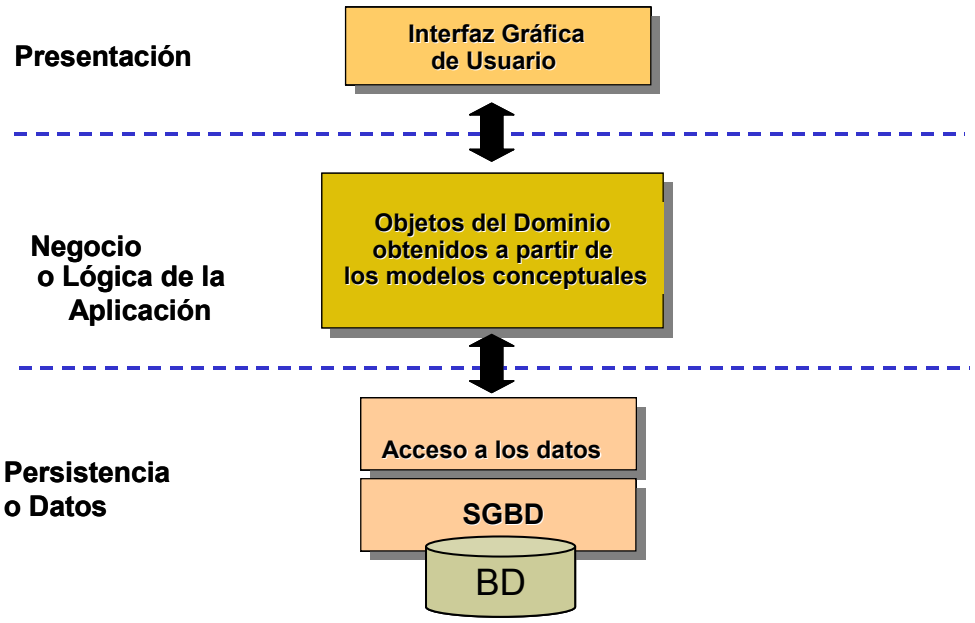

Figura 2. Arquitectura genérica de tres capas para nuestro sistema.

La aplicación a desarrollar, según la arquitectura propuesta, tendrá básicamente la estructura y los componentes siguientes:

- La capa de **presentación** estará constituida por una serie de formularios que permitirán al usuario interactuar con la aplicación, proporcionándole información relacionada con la introducción de pacientes, registro de llamadas de emergencia, etc.
- La capa de **negocio** o **lógica de la aplicación** contendrá la implementación de las clases del dominio. En nuestro caso serán las clases para paciente, hospital, ambulancia, especialidad, etc.
- La capa de **persistencia** estará formada por una base de datos que almacenará la información relevante del caso de estudio. En nuestro caso se incluirá información sobre hospitales, sus especialidades, las ambulancias, los pacientes, etc.

Es importante destacar que estamos utilizando arquitectura de capas cerrada, lo que significa que el sólo se permiten las interacciones entre las capas persistencia-negocio y negocio-presentación.

NOTA. En Java, el punto de entrada o inicio de una aplicación viene determinado por un método de clase (estático) con nombre Main(). Este método tiene que inicializar la aplicación, creando los objetos necesarios y el formulario principal. Esto implica que el formulario principal tendrá definido un atributo de tipo ***ServicioEmergencia*** y será el encargado de invocar al constructor de la clase ***ServicioEmergencia***, que a su vez iniciará la creación de todos los objetos necesarios para la aplicación.

En el resto del documento nos centramos en la capa de negocio, viendo qué clases con sus atributos y métodos tenemos que implementar para dar un soporte adecuado a esta arquitectura.

Capa de Negocio (o Lógica) de la Aplicación.

La Figura 2 muestra la solución propuesta para el desarrollo del caso de estudio (fase de modelado conceptual). Partiendo del diagrama de clases UML generado durante la fase de modelado conceptual, una primera tarea importante a realizar sobre este diagrama es la obtención de un modelo de diseño ya en el espacio de la solución (solución a implementar). En este diseño se obtienen las clases que formarán la capa de negocio; además se deben elegir los tipos de colección adecuadas (colección genérica, pila, lista, cola, etc.).

1. Decisiones de diseño

En la obtención del modelo de diseño nos debemos plantear si adoptamos alguna decisión de diseño y así modificar el diagrama de clases propuesto en la etapa de análisis (Figura 2). Por ejemplo, ahora es posible:

- **Crear nuevas clases**. En algunas situaciones es necesario añadir clases que no son propiamente de modelado, como por ejemplo clases para almacenar colecciones o clases que faciliten la tarea de implementación. En nuestro caso, añadiremos únicamente la clase ***ServicioEmergencia***.

- **Eliminar clases y/o fusionarlas con otras**. Para simplificar el diseño y la implementación, es posible eliminar algunas clases y/o fusionar sus atributos con las de otras. Esto suele ser frecuente en aquellas situaciones donde las clases están relacionadas con cardinalidades 1..1. Obviamente los atributos no se pierden, si no que pasan a estar en la clase fusionada. Por ejemplo, en nuestro modelo podríamos eliminar la clase ***Privada*** (Ambulancia Privada), pero se ha decidido añadir un nuevo atributo a dicha clase (la compañía) y mantenerla. Respecto al resto del diagrama, no necesitamos eliminar ni fusionar otras clases.

- **Crear nuevas relaciones**. Por motivos de eficiencia es habitual añadir nuevas relaciones al modelo de clases. En concreto podemos crear nuevas relaciones que, aunque no aportan ninguna nueva semántica al modelo, sí ofrecen un acceso más

sencillo y con menor complejidad temporal. Por ejemplo, si quisiésemos saber los síntomas registrados en el servicio de emergencias, podríamos acceder a través de los registros de emergencia a los síntomas, o bien, añadir una relación directa entre la clase ***ServicioEmergencia*** y ***Síntoma***. En nuestro caso, únicamente creamos las relaciones que conectan la clase ***ServicioEmergencia*** con ***RegistroEmergencia***, ***Ambulancia*** y ***Hospital***.

- **Modificar relaciones existentes**, bien restringiendo su navegabilidad (en asociaciones inicialmente bidireccionales) haciéndola ahora unidireccional. A menudo las relaciones entre clases no precisan ser implementadas en ambos sentidos por lo que se pueden sugerir restricciones de navegabilidad para simplificar el modelo. En nuestro caso, dado que tenemos sólo un Servicio de Emergencia, no necesitamos saber el servicio de emergencia de un hospital, de una ambulancia o de un registro de emergencia. Por lo tanto, las agregaciones de ***ServicioEmergencia*** con ***RegistroEmergencia***, ***Ambulancia*** y ***Hospital*** son unidireccionales. En la Figura 3 esto lo representamos mediante una flecha, indicando el sentido que se ha privilegiado en el diseño. De esta forma una asociación/agregación, en principio bidireccional, será finalmente implementada atendiendo únicamente a un sentido. Por otra parte, también se restringe la relación "realizada" entre ***RegistroEmergencia*** y ***Paciente***.

Finalmente, al tener la solución de diseño es conveniente repasar los atributos de las clases por si necesitamos más información, por ejemplo en nuestro caso se han añadido: a) atributos *fecha* y *hora* a la clase ***RegistroEmergencia***; b) atributo *descripción* a la clase ***Síntoma***.

Como puede observarse, estas decisiones de diseño son sólo una guía y dependen de la solución concreta que queramos ofrecer en cada caso. En particular, para un mismo modelo de clases de análisis pueden aparecer distintos modelos correctos de clases de diseño.

NOTA: Si durante el desarrollo del caso de estudio el alumno considera que es conveniente añadir/eliminar/modificar alguna relación más al diagrama de clases propuesto, debe consensuar con su profesor esa modificación.

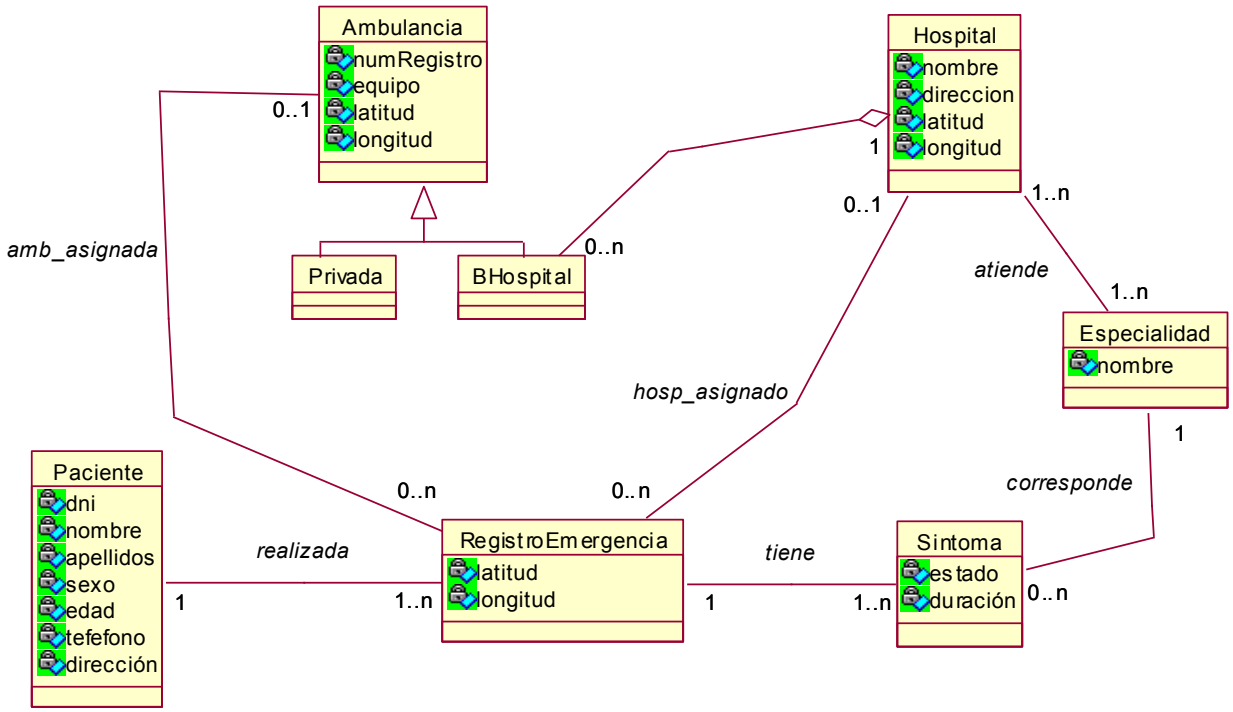

Figura 2. Solución de modelado conceptual propuesta en la etapa de análisis.

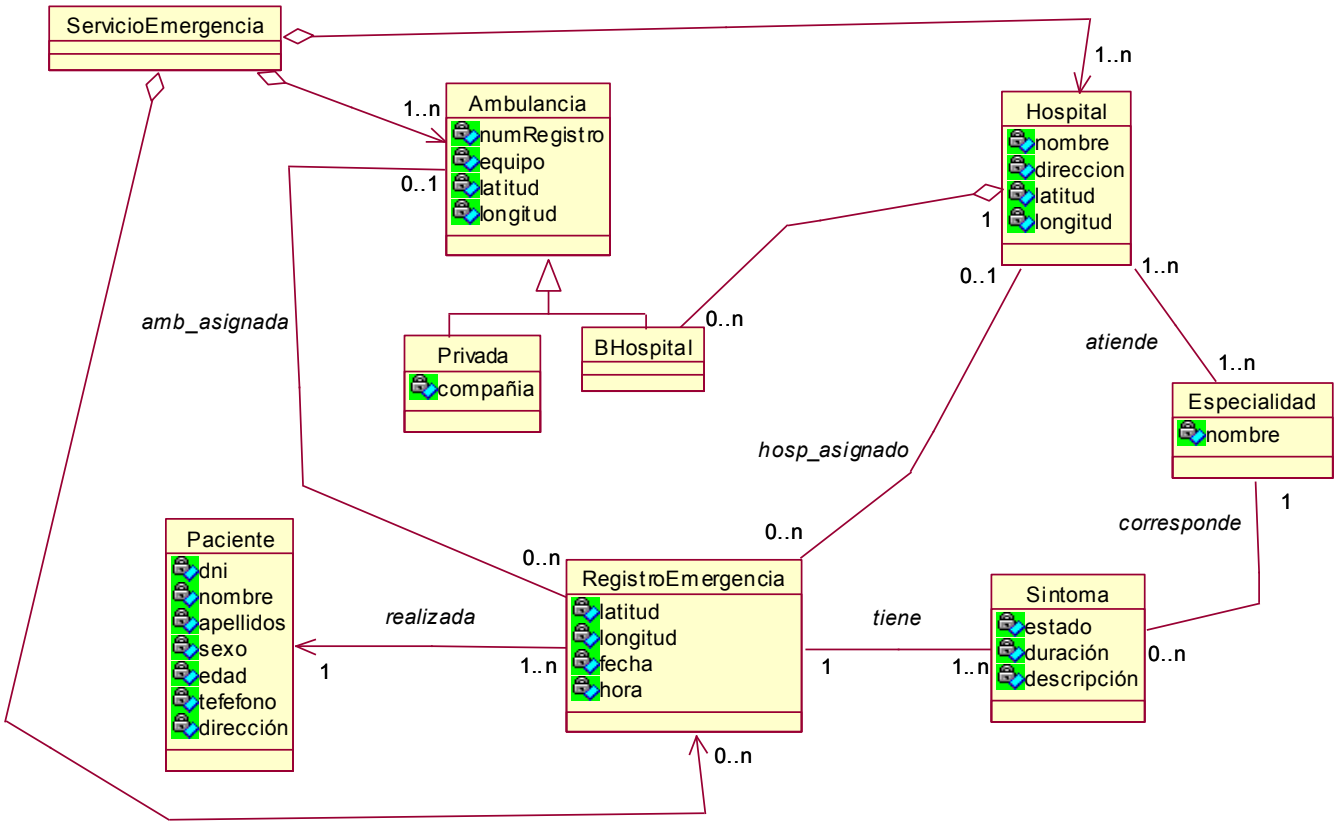

Figura 3. Solución propuesta para la etapa de diseño.

2. Polimorfismo en las Colecciones Java

En Java podemos utilizar, o extender, colecciones predefinidas (colas, listas, pilas, vectores, listas ordenadas, diccionarios, etc.) para almacenar objetos de tipos diferentes. Gracias al polimorfismo de inclusión estas colecciones pueden incluir/almacenar en sus variables todo tipo de objetos de nuestra aplicación

Un aspecto a tener en cuenta cuando trabajemos con este tipo de colecciones es que al obtener o extraer un elemento de la colección nos devolverá un Objeto, por lo que tendremos que forzar el tipo (realizar un *casting* o conversión explícita) a la clase de elemento que queramos manipular. Para saber a qué clase pertenece un Objeto se puede utilizar el operador **instanceof** (por ejemplo, *if (ValorObjeto instanceof NombreClase)*). Esta situación es habitual en colecciones que almacenan objetos de clases relacionadas mediante generalización o herencia.

No obstante, existe la posibilidad de utilizar **colecciones genéricas** de elementos. Así, podemos crear listas (ArrayList<Tipo>), listas ordenadas o diccionarios (TreeMap<TipoClave,TipoValor>), colas (ArrayDeque<Tipo>) o pilas (Stack<Tipo>) de elementos de un tipo determinado evitándonos tener que realizar castings a la hora de recuperar los elementos y asegurándonos en tiempo de compilación de que el contenido de las colecciones es correcto. En nuestro caso de estudio, un ArrayList<Piso> no aceptará un elemento de la clase Casa.

NOTA. Si a priori conocemos el tipo de valor (o clave) que vamos a utilizar es, por tanto, **muy recomendable** utilizar este tipo de colecciones genéricas.

Ejercicio: Realizar el diseño en Java de la solución propuesta (ver Figura 3) siguiendo las técnicas y patrones de diseño vistos en las clases teóricas.

NOTA. Se deben implementar todas las clases con sus atributos y los métodos consultores/modificadores. Para el caso de los atributos que representan colecciones, se deben implementar los métodos para añadir/eliminar elementos y los típicos de búsqueda por el atributo que lo identifique unívocamente.
Para este ejercicio **no es necesario** implementar los constructores.