Trabajo Práctico
Organizador de Futbol 5

Entrega Nº 9

Materia: Diseño de Sistemas
Profesor: Nicolas Passerini
Ayudante: Gisela Decuzzi
Alumnos:

- **Acosta Naiara**
- **Brandoni Agustin**
- **Coiro Tomas**
- **Leder Brian**

2014

1. Justificación de la estrategia de mapeo definida para cada uno de los siguientes casos
    - ***Herencia***

      Si bien en nuestro modelo no contamos con herencia pura, si se tiene una interfaz que refleja un impedance mismatch similar al de la herencia. En nuestro caso es la interfaz TipoDeInscripción donde las clases Condicional, Estandar y Solidaria implementan la misma. El mapeo al modelo de datos se realizó utilizando una única tabla (Tipo_inscripcion) ya que los atributos entre las diferentes clases eran los mismo lo que variaba era el comportamiento de estas, por lo tanto no fue necesario utilizar diferentes tablas que mapeen las subclases o alguna estrategia alternativa. Los tipos de inscripción poseen un id y una descripción donde se detalla a que tipo de inscripción corresponde, de todos modos el id las identifica unívocamente.

    - ***Manejo de identidad***

      Para el manejo de identidad en las tablas del modelo de datos todas debieron incluir un ID (Identificador) para identificar los registros unívocamente. En el caso de las tablas que ayudaban a la implementación de una relación de muchos a muchos el ID es compuesto. Como consecuencia en el modelo de objetos se debió agregar el atributo ID que se mapea directamente con el de la tabla que mapea. Por ejemplo para la clase Jugador se agrego el atributo ID_Jugador y en la tabla Jugador uno de sus campos es id_jugador. Antes los objetos no necesitaban ese atributo, porque dentro de la aplicación cada uno de estos es único para la VM.

    - ***Relaciones***

    - <u>Uno a muchos</u>

      La mayoría de nuestras relaciones son uno a muchos, por lo que se va a comentar como se resolvió esta situación para una de nuestras clases principales: Jugador. La tabla Jugador tiene una relación de uno a muchos con Calificaciones y con Infracciones. En el modelo de datos el jugador tenía una colección de calificaciones y una colección de infracciones. Para el modelo de aplicación estas dos colecciones pasaron a ser tablas que conocen al jugador (es decir que tiene el id_jugador como atributo). Cabe aclarar que esta clase tiene otras relaciones de uno a muchos que se resuelven de la misma manera.

    - <u>Muchos a muchos</u>

      Este tipo de relación se puede ver por ejemplo entre Jugadores y Equipos, en el modelo de datos se debió crear una tabla que relacione la Tabla Equipo y la Tabla Jugador, donde esta tabla posee como PK la composición de las PK de las otras dos tablas. Esto se debió realizar de esta forma ya que en la Base de Datos no se puede representar “directamente” la relación muchos a muchos, en cambio en el modelo de objetos si. En la aplicación no esta la clase Equipo, sino que esto es únicamente un atributo de la clase Partido que contenía una colección de jugadores. Previamente se tomó la decisión de crear la Tabla Equipo para identificar qué jugadores estaban en cada equipo y luego al notar el tipo de relación se creó la tabla Equipo_Jugador que relaciona, como se mencionó antes, Equipo y Jugador.

    - <u>Recursivas</u>

      Si se reifica el amigo (que actualmente es un mail) y se creara la clase Amigo, esta clase sería muy similar a la clase Jugador, por lo que en vez de tener dos clases similares se podría tener una única clase, por ejemplo Persona, donde las

personas que sean jugadores tendrían una relación recursiva con esta clase para representar los amigos de los jugadores. Para mapear esto en la base de datos, se optaría por tener una única tabla Persona que contiene entre sus atributos el id_persona como PK. Y un campo amigoDe que almacene el id_persona de quien es amigo una Persona que es Amigo y no Jugador.

- ***Tipos de datos***

  En el caso de los Strings nos encontramos que en xtend estos no tienen límite en cambio en la base datos debemos especificarle un límite, esto nos sucedió por ejemplo con el atributo nombre de la clase Partido. Por lo tanto desde la aplicación validaremos que un string que almacenemos en la Base de Datos no supere el límite puesto en el campo.
  Por otro lado en la base de datos no existe el tipo de dato Boolean que nosotros utilizamos por ejemplo en la inscripción para saber si se encuentra confirmada o no. Decidimos en la Base de Datos utilizar un atributo de tipo String (o varchar en realidad) con limite 1, donde se almacena 'S' o 'N' para representar 'True' o False respectivamente.
  Por último las fechas no son las mismas las que maneja SQL respecto a las que maneja Xtend (o Java).
  No tuvimos que manejar o decidir sobre los tipos de datos numéricos ya que no contamos con números decimales, sólo tenemos números de tipo int para la base de datos que se corresponden con los int de la aplicación.

2. Componentes del sistema afectados utilizando el Framework ORM
   - A todas las clases que se impactan directamente como ser: Jugador, Partido, Infraccion, Inscripcion, Propuesta, Administrador, Calificación se le debió agregar el atributo ID tal como se mencionó anteriormente.
   - Xtend tiene la propiedad de que por medio de la directa @Property antes de un atributo no es necesario realizar el Getter y Setter de ese atributo. Para la implementación de Hibernet se necesitaban los Getter y Setter explícitos por lo que se debió implementar estos métodos para todos los atributos que se mapean en la Base de Datos.
   - A los atributos de las clases que se impactan directamente en la Base de Datos que son ID se debió agregar delante del atributo la directiva @ID
   - A los atributos de la clases que se impactan direcamtente en la Base de Datos se debió agregarles delante del atributo la directiva @Column
3. Cómo ayudó la arquitectura presentada en el trabajo práctico 2.1 para minimizar el impacto de los cambios. Relacionarlo con el concepto acoplamiento

   En ese punto del proyecto se propusieron dos alternativas de solución para el problema presentado, las mismas eran implementar la solución utilizando el patrón Observer o utilizando el patrón Decorator. Nuestra elección definitiva fue implementar el patrón Observer. Esta solución nos permitió tener bien distribuidas las responsabilidades aumentando la cohesión de los componentes. En especial en la clase Partido que es quien tiene los observer principales, si el requerimiento no lo hubiéramos resuelto utilizando el patrón, las clase Partido iba a tener la responsabilidad de comunicar a quien correspondía los eventos que sucedían. De ese modo posiblemente esa clase iba a tener más atributos (porque hubiera necesitado conocer a los objetos que tenía que informar los eventos), estando mas acoplado a los objetos que debería notificar, donde en esta instancia deberíamos tomar la decisión de qué atributos se impactaban en la base y cuales no.

Con la decisión tomada, la decisión de qué atributos se impactaban de la clase Partido resultó sencilla ya que todos los atributos se encuentran mapeados.