Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de un módulo de gestión y consulta de pequeñas terminologías clínicas conforme con ISO 21090.

Proyecto Final de Carrera

Ingeniería Técnica en Informática de Gestión

Autor: Sergio Rodríguez Muñoz

Directores: Montserrat Robles Viejo, José A. Maldonado Segura

25 de Septiembre de 2012

Resumen

En la metodología de modelo dual para el modelado de la historia clínica electrónica los arquetipos definen los potencialmente numerosos conceptos del dominio como problema, hoja de interconsulta, presión sanguínea, etc. En la definición de arquetipos, a parte de las terminologías de propósito puramente clínico, se hace uso de una gran cantidad de pequeñas terminologías que sirven para definir el contexto clínico de la información. Este proyecto tiene como objetivo el desarrollar un sistema informático en Java basado en la norma ISO 21090 de tipos de datos en salud para la gestión y consulta de estas terminologías por parte de un editor de arquetipos.

Palabras clave: historia clínica electrónica, arquetipos, terminologías, modelos de información, tipos de datos en sanidad, ISO 21090, informática médica, JAVA, Hibernate, hypersql.

Tabla de contenidos

CAPÍTULO 1: INTRODUCCIÓN.

En la actualidad, dentro del ámbito clínico la necesidad de comunicación y transmisión de la información se hace cada día más necesaria. El uso de estándares para la comunicación de las historias clínicas electrónicas (HCE) es crucial para asegurar que la comunicación se realiza de manera segura y conservando el significado original de los datos, es decir con un nivel de interoperabilidad semántica suficiente. Pero la adopción de estos estándares es costosa, debido principalmente complejidad de propios estándares y del dominio clínico. Por todo ello, son necesarias herramientas que oculten lo máximo posible la complejidad de los estándares.

Una de las principales y primeras tareas en cualquier proyecto de intercambio de HCE es la definición de los conceptos del dominio (por ejemplo hoja de interconsulta, informe de alta, lista de medición, etc.) a comunicar y establecer su relación con el estándar de HCE utilizado. La metodología del modelo dual [1] de arquitectura de historia clínica electrónica facilita mecanismos para este fin. En este modelo existe una clara separación entre la representación de datos (el modelo de referencia) y la representación de los modelos clínicos (el modelo de arquetipos). Mientras que el primero proporciona una sintaxis para representar cualquier dato de la historia clínica electrónica así como de la información de contexto necesaria para interpretarla correctamente, los arquetipos permiten definir de manera formal los conceptos del dominio. Esta aproximación se utiliza en la norma ISO 13606 [2], así como por las especificaciones de openEHR [3]. También HL7 v3 [4] utiliza una metodología similar, si bien no emplea el concepto de arquetipo, la cual define un modelo de referencia y un mecanismo de refinamiento para crear modelos de dominio específicos.

1.1 - Problema a resolver

Los arquetipos son el vínculo de las estructuras de información con las terminologías (vocabulario) y potencialmente con las ontologías que describen la semántica de la información clínica. Existen diversos terminologías relevantes en el ámbito clínico, que van desde simple listas de términos a ontologías basadas en algún formalismo lógico como Snomed-CT. En la definición de arquetipos se hace uso de dos tipos de terminologías: las puramente clínicas y la de contexto clínico. En el caso de las terminologías puramente clínicas, como Snomed-CT o LOINC, y debido a su complejidad (en el caso de Snomed-CT estamos hablando de más de 300000 términos) es necesario utilizar servidores de terminología con funcionalidades avanzadas. Por el contrario las terminologías de contexto son pequeñas terminologías

que pueden ser externas o definidas en el propio modelo de referencia. Contienen términos que sirven para interpretar adecuadamente un extracto de HCE, clasificar los extractos o especificar información ético-legal como auditoría, idioma o control de cambios. Por tanto, están íntimamente relacionadas con el modelo de referencia y son básicas en la definición de arquetipos. Como consecuencia es recomendable que los editores de arquetipos sean capaces de gestionar estas pequeñas terminologías de manera independiente de los servidores de terminología, los cuales no están disponibles en la mayoría de las organizaciones y ofrecen interfaces de consulta variable. Por otro lado, los modelos de referencia en sanidad imponen el uso de un conjunto de tipos de datos, por ejemplo norma ISO 21090 [5], para la representación de la información que incluyen tipos para la representación de términos codificados lo cuales hay que tener en consideración.

1.2 – Objetivos y alcance del proyecto

Diseño e implementación de un sistema informático en Java basado en la norma ISO 21090 (Tipos de datos para la salud) para la gestión y consultas de pequeñas terminologías, concretamente:

- Terminologías locales de los modelos de referencia para la comunicación de la historia clínica electrónica como ISO13606 o CDA.
- Terminologías que aportan contexto básico a la información clínica como ISO 639-2 (Idiomas) o ISO 3166 (países).

Este sistema informático estará orientado para su uso en la definición formal de conceptos clínicos expresados como arquetipos. Su uso es relevante tanto para detallar los metadatos del arquetipo (por ejemplo idiomas soportados) como en la definición de los conceptos clínicos a partir de los modelos de referencia.

1.3 - Contexto de realización de proyecto

Este proyecto fin de carrera se ha realizado bajo la dirección de Montserrat Robles y José Alberto Maldonado miembros del grupo de Informática Biomédica (IBIME) del Instituto ITACA. El trabajo a desarrollar se encuadra dentro de la línea de investigación en el modelado semántico de historias clínicas electrónica de IBIME y pretende facilitar una herramienta para la consulta de pequeñas terminologías durante la edición de arquetipos.

CAPÍTULO 2: MODELADO DE LA HISTORIA CLÍNICA ELECTRONICA (HCE).

2.1 - Problemas en el modelado de la HCE

En la actualidad, el intercambio de información clínica entre profesionales sanitarios o entre sistemas informáticos es hoy en día un problema crítico para el sector sanitario. Este intercambio debe realizarse conservando fielmente el significado original de los datos (interoperabilidad semántica) y la integridad médico-legal. Este problema de comunicación no es trivial y es aún más complejo que en otros sectores. La principal razón es la propia complejidad y variabilidad de la medicina que da lugar a modelos muy grandes, cambiantes y difícilmente manejables. La medicina no es sólo muy amplia, sino que no tienen límites:

- En amplitud: porque cada día se descubre nueva información o información que no era relevante ahora si lo es.
- En profundidad: porque cada día se descubre información más detallada o ésta pasa a ser relevante.
- En complejidad: porque siempre se descubren nuevas relaciones o relaciones que no eran relevantes ahora si lo son.

A esto se debe añadir la gran heterogeneidad en cuanto a las necesidades de los usuarios de la información: pacientes, facultativos, investigadores, gestores, laboratorios, gestores, etc.

Un aspecto clave a la hora de intercambiar historias clínicas electrónicas (HCE) es el contexto, de hecho, se suele dividir cualquier anotación en una historia clínica electrónica en tres partes: contenedor (encabezamientos de secciones, subsecciones y elementos atómicos de información), contenido y contexto. Se puede definir el contexto como cualquier tipo de información que influye en la interpretación de una anotación. Esta información es básica para poder conservar fielmente el significado de los datos clínicos y para preservar la integridad médico-legal. La información de contexto puede ser de diversos tipos:

- Contexto estático o posición de la anotación dentro de la historia clínica.

- Contexto dinámico representado por todas aquellas anotaciones que influyen en el significado de la anotación considerada.
- Contexto de seguridad que engloba todos aquellos detalles que son esenciales para interpretar correctamente una anotación, por ejemplo: sujeto (paciente, donante, feto, familiar del paciente…), estado especial del paciente (diabético/a, embarazada, alérgico/a…), certeza (confirmado, suposición, descartado...).
- Contexto ético y legal que engloba toda aquella información relevante para cumplir los requerimientos de la legislación vigente y para dar soporte a posteriores labores de auditoria o control, como responsable de la atención sanitaria, persona o dispositivo que registró la información, firma digital, fechas y horas de las acciones sanitarias y de registro de la información en el sistema informático.

Es por todo esto, que uno de los aspectos más importantes a la hora de desarrollar cualquier sistema que intercambie o gestione información clínica es cómo organizar la información para que satisfagan todos estos los requerimientos básicos. Una arquitectura de información de la historia clínica electrónica (AHCE) modela las características genéricas aplicables a cualquier anotación en una historia clínica independientemente de la organización (primaria, especializada, etc.), del profesional (médico, enfermera, etc.), especialidad o uso (asistencial, investigación, salud pública, etc.). Una AHCE como un modelo conceptual de la información que puede estar contenida en cualquier HCE por tanto modela las características comunes a todas las HCE, pero no detalla qué información debe estar contenida en una historia ni cómo un sistema de HCE debe implementarse. El desarrollo de una AHCE precisa de un elevado nivel de abstracción para definir componentes de uso general que permitan describir cualquier entrada en una historia clínica. Las organizaciones de normalización en el sector sanitario como HL7 o CEN/TC 251 y de propósito general como ISO han sido conscientes de la importancia de las AHCE para facilitar la interoperabilidad semántica de la HCE. Una de las aportaciones más importantes de la última generación de AHCE es el uso de un modelo dual para su especificación. Concretamente, la norma ISO 13606 [1], la arquitectura propuesta por el consorcio OpenEHR [2] y en cierta medida HL7 CDA [3] se basan en un modelo dual.

2.2 - Modelo dual de AHCE

En la base del modelo dual se encuentra la separación entre conocimiento e información. Por información se entiende datos u opiniones sobre una entidad del mundo real. Por ejemplo, Luis Vila el día 20/08/2002 tenía una cantidad de Fibrinógeno en sangre de 256 mg/dl, esta información se refiere a Luis Vila y no a otra persona. Mientras que conocimiento se define como los datos y hechos que son ciertos para todas las instancias de las entidades a las que se refieren. Por ejemplo, el fibrinógeno es una proteína sintetizada por el hígado que se utiliza en el proceso de coagulación de la sangre, su cantidad se mide en miligramos por decilitro y el intervalo de referencia va desde los 200 mg/dl a los 450 mg/dl, esta información se puede aplicar a todas las personas.

En el modelo dual, el modelo único previo que incluía todos los conceptos del dominio se transforma en un “pequeño” modelo, denominado modelo de referencia, que incluye exclusivamente los conceptos no volátiles que permiten describir cualquier anotación en la historia clínica. En el ámbito de la historia clínica electrónica un modelo de referencia debe contener básicamente:

- Un conjunto de conceptos de negocio, cada uno de los cuales representa un tipo de bloque constitutivo de la historia clínicas con diversa granularidad.
- Los tipos de relaciones posibles entre los conceptos de negocio (herencia, agregación, composición, etc.).
- Estructuras de datos como valores simples, listas, tablas, árboles, series temporales, etc.
- Un conjunto de tipos de datos básicos como texto, término codificado, multimedia, cantidad física, rango, etc.

Los profesionales sanitarios suelen manejar un conjunto más o menos fijo de estructuras de información que representan conceptos médicos para la realización de sus actividades, por ejemplo, un informe de alta, historia clínica de primaria, resultados bioquímicos, diagnóstico, presión sanguínea etc. Los arquetipos son los modelos que permiten definir formalmente estos conceptos en relación con el modelo de referencia. Podemos definir formalmente un arquetipo como un modelo del nivel de conocimiento que define una estructura de información utilizada en un dominio particular por medio de restricciones sobre las componentes del modelo de referencia. Cuando un arquetipo restringe un componente del modelo de referencia se suele decir también

que el arquetipo extiende o especializa el componente. De manera intuitiva podemos hacer una analogía con la arquitectura, mientras que el modelo de referencia se correspondería con los tipos de “ladrillos” disponibles, los arquetipos definirían “planes de construcción” de tipos de elementos arquitectónicos, tales como un ábside o un tejado de doble vertiente. La definición de arquetipos es tarea de los especialistas en el campo de interés.

Entre otras ventajas, diremos que los arquetipos son fácilmente comprensibles por los profesionales de la salud. Además, estos son reutilizables, pueden especializarse para satisfacer mejor los requerimientos, se pueden agregar para crear grandes arquetipos y pueden ser versionados para adoptar nuevas funcionalidades. Los arquetipos son también multilingües, ya que pueden ser definidos para facilitar la legibilidad humana en diferentes idiomas. Las instancias de arquetipos se especifican por medio de un lenguaje formal de descripción de arquetipos, conocido como ADL (del inglés *Archetype Description Language*).

2.3 - Uso de terminologías en las AHCE

La utilización de vocabularios normalizados junto con los modelos de referencia y los modelos de dominio (arquetipos) es vital para alcanzar un alto grado de interoperabilidad semántica de la HCE. Por tanto, no es de extrañar la existencia de una relación muy estrecha entre las terminologías y los modelos de referencia y los arquetipos. Si nos centramos en los modelos de referencia, podemos distinguir dos tipos de terminologías: las puramente clínicas y las de contexto clínico (objeto de este proyecto fin de carrera). Las primeras son terminologías externas al modelo de referencia que contienen términos directamente relacionados con el cuidado y tratamiento del paciente (anatomía, enfermedades, procedimientos, etc.). Se han llevado a cabo muchos esfuerzos por conseguir la normalización de la terminológica clínica, que se han plasmado en una gran cantidad de terminologías muchas de ellas complejas y que abarcan diversos dominios. Las segundas son pequeñas terminologías que pueden ser externas o definidas en el propio modelo de referencia que contienen términos que sirven para interpretar adecuadamente un extracto de HCE, clasificar los extractos o especificar información ético-legal como auditoría, idioma o control de cambios. Por ejemplo el atributo “territory” de la clase CLINICAL_SESSION de ISO 13606 contiene un código para el territorio en el que se creó el extracto de HCE extraído de la norma ISO3166 que codifica los nombres de países y áreas dependientes y sus principales subdivisiones.

2.4 -Tipos de datos en sanidad

El propósito de esta norma internacional ISO 21090. (Health informatics - Harmonized data types for information interchange) es proporcionar un conjunto de definiciones de tipos de datos para la representación y el intercambio de conceptos básicos que se usan comúnmente en entornos sanitarios. En consecuencia, esta norma específica la semántica compartida para una colección de tipos de datos relacionados con la salud y que son adecuados para su uso en normas relacionadas con la información sanitaria. La norma también contiene la definición de los tipos de datos utilizando la notación UML, además de incluir una representación basada en XML para su uso en el intercambio de información. Los tipos de datos contenidos son resultado de los requerimientos recogidos en HL7 V3, también de ISO 13606, openEHR, y el trabajo anterior de ISO sobre tipos de datos.

La norma contiene tipos de datos para la representación de texto y datos binarios, identificadores y localizadores, nombres y direcciones, cantidades, colecciones, fechas, horas e intervalos, datos inciertos y códigos. Estos últimos los relevantes para este proyecto fin de carrera. La norma define diversos tipos de datos para la representaciónd e valores codificados como: tipos de datos para la representación de valores codificados: CD (descriptor de concepto), CR (Rol de un concepto), CE (valor codificado con equivalencias), CV (valor codificado) y CS (valor codificado simple) los cuales se describirán con detalle en el siguiente apartado de la presente memoria.

2.4.1 - Tipos de datos para la representación de terminologías

El objetivo del proyecto es la implementación de un sistema informático para la gestión y consulta de las pequeñas terminologías utilizadas por los modelos de referencia y arquetipos. Para la representación de los conceptos contenidos en estas terminologías se utiliza la norma ISO 21090, la cual describe un conjunto amplio de tipos de datos para el dominio sanitario. En concreto en este proyecto se hace uso de los tipos HXIT, ANY, CD, CS y CD.CV. Los dos primeros, son tipos de datos abstractos que contienen atributos comunes a todos los tipos de datos mientras que los tres últimos son específicos para la representación de códigos.

HXIT

Recoge información sobre la historia del valor. Contiene datos tales como el período de validez y una referencia al evento que representa en el caso de que este valor

establecido como válido.

HXIT se define como una clase privada y abstracta por tanto, este tipo de datos no es para su uso fuera de los tipos de datos en la especificación. Debido también a dichas cualidades no existe una definición de igualdad entre variables de tipo HXIT.

Atributos:

- **validTimeLow** <<string>>: Representa el instante de tiempo en el que la información se hizo o se marco como válida.
- **validTimeHigh** <<string>>: El tiempo en que la información dada cesará de ser correcta. El valor de ambas variables descritas anteriormente tiene que ser una marca de tiempo valida que siga el formato TS.value (03 de julio 2005 3:00 pm).
- **controlActRoot** <<string>>: Es el identificador del evento asociado con el tipo de datos definido y su valor especificado.
- **controlActExtension** <<string>>: La extensión del identificador del evento asociado con el tipo de datos definido y su valor especificado.
- Los atributos de HXIT (validTimeLow, validTimeHigh, controlActRoot, controlActExtension) nunca participar en la determinación de la igualdad de las especialidades de HXIT.

ANY

Define las propiedades básicas de cada valor de datos. Especializa HXIT y por ello hereda sus atributos. Es conceptualmente un tipo abstracto, lo que significa que no tiene valor adecuado puede ser sólo un valor de datos sin pertenecer a ningún tipo concreto. En ocasiones valores nulos pueden ser de tipo ANY, a excepción “nullFlavor” INV, ya que este atributo requiere un valor de la enumeración de valores definidos como validos para que tenga sentido.

Atributos:

- **nullFlavor** <<string>>: Si el valor es nulo, indica la razón. Este atributo tiene limitado su valor. NI: not information, INV: invalid, OTH: other, NINF: negative infinity, PINF: positive, UNC: unencoded, DER: derived, UNK: unknown, ASKU: ask but unknown, NAV: temporality unavailable, QS: sufficient quantity, NASK: not asked, TRC: trace, MSK: nasked, NA: not

applicable.

- **updateMode** <<string>>: Permite identificar el papel que juega el atributo en el procesamiento de la instancia que se está representando. El valor de este atributo se tomarán a partir del código HL7 “updateMode” sistema. Los valores actuales son: A, D, R, AR, N, U y K (add, delete, replace, add or replace, no change, unknonw y key).
- **flavorId** <<string>>: Identificador de la imposición de un conjunto de restricciones sobre el tipo de datos. El único propósito de especificar que una restricción se ha utilizado para restringir aún más el tipo de datos es para permitir la validación de las instancias: un motor de validación puede consultar las normas expresadas para el tipo y confirmar que la instancia se ajusta a las normas.

CD - Concept Descriptor

Esta terminología puede contener un código simple es decir, una referencia a un concepto definido directamente por el sistema de codificación, o puede contener una expresión en una sintaxis definida por el sistema de codificación que puede ser evaluada.

Un CD puede contener uno o más traducciones en múltiples sistemas de codificación. Las traducciones son todas las representaciones del mismo concepto en sistemas de códigos diferentes. Sólo hay un concepto, y sólo el CD raíz puede contener un texto original.

Atributos:

- **validTimeLow** <<string>>.
- **validTimeHigh** <<string>>.
- **controlActRoot** <<string>>.
- **controlActExtension** <<string>>.
- **nullFlavor** <<string>>.
- **updateMode** <<string>>.
- **flavorId** <<string>>.

Estos primeros atributos proceden de la herencia de la especialización de ANY. El valor de estos atributos será nulo tal y como indica la especificación de esta terminología.

- **code** <<string>>: Código simple definido por el sistema de códigos, o una

expresión en una sintaxis definida por el sistema de código que describe el concepto e identifica a la terminología.

- **codeSystem** <<string>>: Especifica el sistema de codificación usado en la propiedad “code”. Es un atributo con restricción de valor no nulo y va ligado al atributo “code”.
- **codeSystemName** <<string>>: Contiene el nombre común del sistema de codificación.
- **codeSystemVersion** <<string>>: Indica la versión del sistema de codificación utilizado en el atributo que identifica a la terminología.
- **valueSet** <<string>>: Esta propiedad complementa y forma parte del código que identifica la terminología. Se trata de una fecha en entrada en el sistema.
- **valueSetVersion** <<string>>: Indica el formato definido en TS.DATETIME.FULL en el que esta representada la fecha contenida el la propiedad “valueSet”. Este atributo no tiene sentido sin el anterior sin embargo si existe valor para “valueSet” conlleva a que la propiedad que se esta definiendo no sea nula.
- **displayName** <<string>>: Es la traducción del código asignado a la terminología al lenguaje natural, de forma que represente algo para un humano y con lo cual pueda identificarla.
- **originalText** <<string>>: Es el texto recogido desde la interfaz y que el usuario ha introducido o seleccionado de una lista y forma la base para asignar el código a la terminología CD.
- **codingRationale** <<string>>: Proporciona información acerca de un determinado CD, ya sea para indicar que es el CD raíz o que forma parte de las traducciones de otra terminología CD. Original (O), post-coded (P) y required (R) estos son los posibles valores que puede tomar este atributo.
- **translation** <<HashSet<CD>>>: Representa un conjunto de otros CDs que representan cada uno una traducción de este CD en códigos equivalentes dentro del mismo sistema o en los conceptos correspondientes de otros sistemas de código.
- Las traducciones deben de ser exactas sin embargo, rara vez existe sinonimia exacta entre dos sistemas de codificación estructuralmente diferentes. Por esta razón, no todas las traducciones son iguales de manera estricta.
- Las traducciones no contendrán otras traducciones, es decir su atributo

"translation" será una lista vacía. Sólo el CD raíz tiene un conjunto de traducciones que enumera todas las posibles traducciones.

- **source** <<CD>>: Una referencia a la terminología CD que es la fuente de esta traducción. Se empleará este atributo si este CD fue creado para traducirlo de otro CD. Esta propiedad es una referencia, y se utiliza en los CDs dentro del grupo de CDs que representan el mismo concepto.

Ilustrando el tipo de dato de codificación descrito anteriormente se encuentra este ejemplo.

```
<CD>
   <Code>784.0</Code>
   <CodeSystem>2.16.840.1.113883.6.42</CodeSystem>
   <CodeSystemName>ICD-9</CodeSystemName>
   <CodeSystemVersion>3.0</CodeSystemVersion>
   <ValueSet>2.16.840.1.113883.19.11.1</ValueSet>
   <ValueSetVersion>20070711</ValueSetVersion>
   <CodingRationale>12</CodingRationale>
   <Translation>
      <Tranlations>
            <CD>
               <Code>784.0</Code>
               <CodeSystem>2.16.840.1.113883.6.42</CodeSystem>
               <CodeSystemName>ICD-9</CodeSystemName>
               <CodeSystemVersion>3.0</CodeSystemVersion>
               <ValueSet>2.16.840.1.113883.19.11.1</ValueSet>
               <ValueSetVersion>20070711</ValueSetVersion>
               <CodingRationale>12</CodingRationale>
               <DisplayName>Headache</DisplayName>
               <OriginalText>""</OriginalText>
               <Source>784.0</Source>
            </CD>
      </Tranlations>
   </Translation>
   <DisplayName>Headache</DisplayName>
   <OriginalText>Burnt ear with iron. Burnt other ear calling for ambulance</OriginalText>
   <Source>""</Source>
</CD>
```

Figura 1. Representación del arquetipo CD.

CS – Coded Simple Value

Codificado de datos en su forma más simple, donde el atributo "code" por si sólo está predeterminado. Las propiedades "codeSystem" y "codeVersion" del sistema están implícitas y establecidas por el contexto en el que el CS se produce. Debido a que está altamente restringida su funcionalidad, CS sólo se utilizará para simple atributos estructurales con terminologías muy controladas y estables.

Respecto a la propiedad de igualdad entre dos valores de CS se determina únicamente en base al código explícito y el "codeSystem." Otro dato relevante es que los valores de CS pueden ser iguales a los valores de CD si ambos especifican el mismo "code" y "codeSystem", por tanto pueden ser comparados.

Atributos:

- **validTimeLow** <<string>>.
- **validTimeHigh** <<string>>.
- **controlActRoot** <<string>>.
- **controlActExtension** <<string>>.
- **nullFlavor** <<string>>.
- **updateMode** <<string>>.
- **flavorId** <<string>>.

Estos atributos proceden de la herencia de la especialización de ANY, por el contrario el siguiente es propio de la clase CS.

- **code** <<string>>: Código simple definido por el sistema de codificación. Si el valor está vacío o es nulo, entonces no hay código en el sistema de código que representa el concepto. El código se define a partir de la combinación de estos caracteres: una letra, un dígito, un '-', '_' , '.' o ':'. La expresión que forma código no contendrá espacios en blanco ni otros caracteres que no están en esta lista.

CD.CV - Coded Value

Restringe a la terminología de la que se especializa, CD. Los datos son codificados, especificando sólo un código, código de sistema y, opcionalmente el nombre y el texto original. Sólo se utiliza como tipo de propiedades de otras terminologías de sistema. CV se utiliza para cualquier caso de uso que requiera un valor de código único para ser enviados. Por lo tanto, no debe utilizarse en circunstancias donde un determinado valor tenga varias alternativas que lo representen. No puede ser un requisito para migrar a un nuevo sistema de codificación.

Atributos:

Los atributos de esta terminología proceden de la especialización de CD en CD.CV. Ambas terminologías comparten los mismos atributos salvo porque la lista de traducciones y el atributo “originalText” para CD.CV tendrán valor nulo.

- **validTimeLow** <<string>>.
- **validTimeHigh** <<string>>.
- **controlActRoot** <<string>>.
- **controlActExtension** <<string>>.

- **nullFlavor** <<string>>.
- **updateMode** <<string>>.
- **flavorId** <<string>>.
- **code** <<string>>.
- **codeSystem** <<string>>.
- **codeSystemName** <<string>>.
- **codeSystemVersion** <<string>>.
- **valueSet** <<string>>.
- **valueSetVersion** <<string>>: .
- **displayName** <<string>>.
- **originalText** <<string>>.
- **codingRationale** <<string>>.
- **translation** <<HashSet<CD>>>.
- **source** <<CD>>.

CAPÍTULO 3: DISEÑO.

En la etapa de diseño de un proyecto se dan respuesta a preguntas como la forma en la que se va desarrollar el proyecto para que cumpla con todos los requerimientos especificados, en que partes se divide, etc. En el módulo gestor de terminologías sobre el que se ha trabajado podemos distinguir que hay una parte de representación del concepto terminología y de los tipos de datos de codificación; una parte de almacenamiento de los datos; y por último unos servicios que ofrecer sobre ellas.

El trabajo derivado de la etapa de diseño es necesario ya que es tan importante tener claro el trabajo a realizar como el saber la manera en la que se va a realizar.

3.1 - Planificación del proyecto mediante modelos.

Los modelos suponen un estándar en la representación visual del proyecto. Es una herramienta de apoyo para los programadores con la que a través de diagramas conocen la envergadura del proyecto, sus componentes y como se estructura. Puesto que son estándares también son una buena herramienta para la documentación de los proyectos.

En el sistema desarrollado aparecen los conceptos de terminología y de tipo de dato de codificación. Terminologías identificadas por un valor único para cada una de ellas, que tienen las propiedades nombre y versión y que a su vez contienen tipos de datos de codificación que amplían su información. Cada tipo de datos estándar de representación de códigos contiene información adicional que se especificada en el documento ISO 21090, estándar de comunicación de historias clínicas (HCE).

Entre los lenguajes de modelado esta UML (Lenguaje Unificado de Modelado), el cual se usa para especificar, visualizar, modificar, construir y documentar los artefactos de un software orientado a objetos de obra sistema en desarrollo. Como la intención es representar los conceptos mencionados anteriormente se ha empleado el sistema de representación visual llamado Diagrama de Clases. El Diagrama de Clases es el diagrama principal para el análisis y diseño. Un diagrama de clases presenta las clases del sistema con sus relaciones estructurales y de herencia. La definición de clase incluye definiciones para atributos y operaciones. La figura 2 muestra el diagrama de clases del servidor de terminologías.

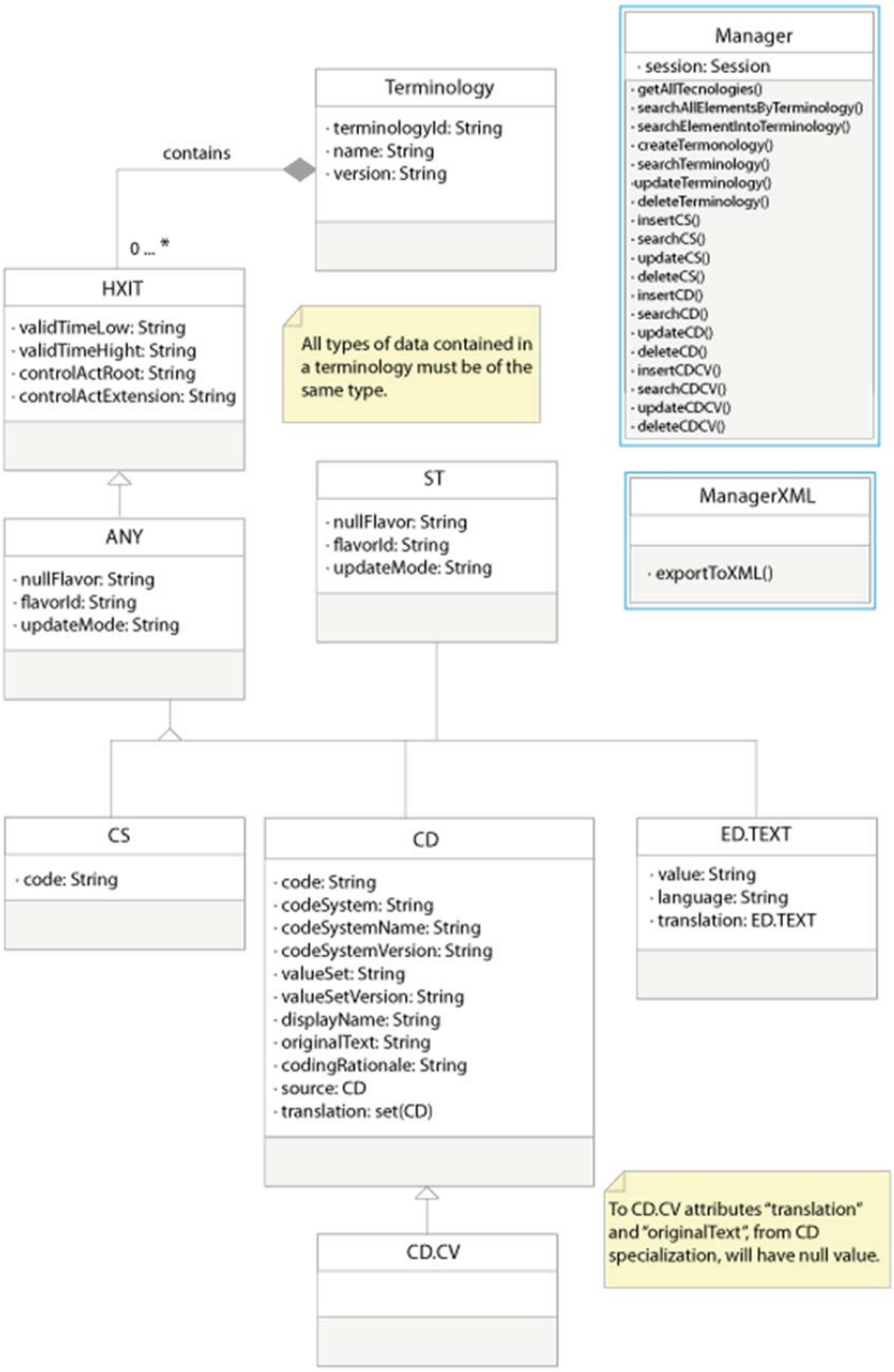

Figura 2. Diagrama de Clases del proyecto sobre la gestión de terminologías clínicas.

Tras analizar y estudiar cada una de las clases y las relaciones que existen entre ellas la implementación es más sencilla. Aunque esta la posibilidad de usar un programa que dado el diagrama genere de forma automática el código de las clases para este proyecto la implementación será manual.

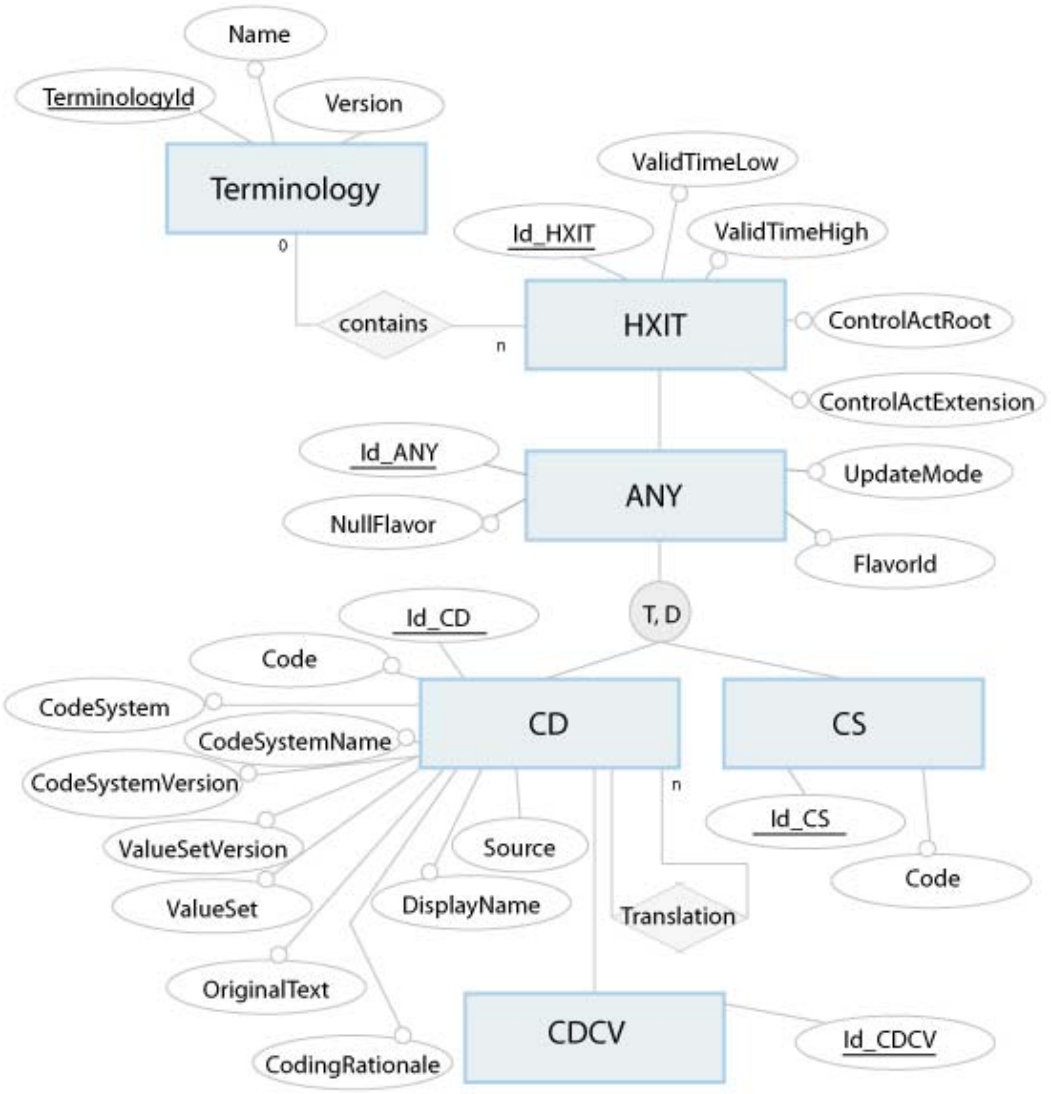

Figura 3. Diagrama del clases del servidor de terminologías de contexto clínico.

3.2 - Planificación de la distribución y la persistencia de datos.

Un vez que ya se ha dejado claro los tipos de datos de codificación que se van a trabajar hay que diseñar y estructurar conforme a la especificación de los tipos de datos el lugar de almacenamiento de las instancias de los objetos en el sistema. Lo habitual en este tipo de proyectos sería hacer uso de un servidor de terminologías pero en el caso tratado es inviable por su complejidad y coste. En su lugar, se empleará una base de datos que cumpla con todos los requerimientos especificados para las terminologías. El esquema entidad-relación del modelo conceptual de la base de datos se muestra en la Figura 3.

Este diagrama recoge todas las entidades relevantes junto con sus atributos y las relaciones entre ellas con sus respectivas restricciones de cardinalidad. En el diseño se aprecia la jerarquía que se da en la entidad HXIT y como se especializan de ella diversas entidades. Estas entidades especializadas tienen restricciones de especialización total y disyunta para el caso de la entidad ANY y su descendencia.

3.3 - Métodos y consultas de la base de datos.

A continuación se desglosan cada uno de los métodos diseñados para llevar a cabo las operaciones pertinentes de la interfaz de usuario. Son métodos de consulta, inserción, búsqueda, modificado y borrado sobre los objetos contenidos en el sistema gestor de terminologías. En definitiva constituyen el listado de operaciones que el usuario del sistema podrá realizar.

- **getAllTerminologies:** Método público que devuelve una colección con las terminologías almacenadas en la base de datos. Esta función no recibe parámetros de entrada.
- **getAllElementsByTerminology**: Método público que dado el identificador de una terminología devuelve el listado de elementos de tipo CD, CS o CDCV contenidos en ella. La colección devuelta puede ser una colección vacía si se da el caso de que a la terminología indicada no se le han añadido ningún tipo de datos de los mencionados.
- **searchElementIntoTerminology**: Dado un identificador de un tipo de dato (CS, CD o CDCV) a encontrar y un identificador de un terminología donde buscarlo se devuelve el objeto solicitado en caso de encontrarse en la lista de elementos de la terminología o un nulo junto a un mensaje de error si no se ha encontrado. Es un método público y forma parte de los métodos para la interfaz de usuario.
- **createTerminology**: Método público cuya función es insertar entrada en la tabla de almacenamiento de terminologías. Los parámetros de entrada son los siguiente: el identificador de la terminología, el nombre y la versión.
- **searchTerminology**: Dado un identificador de una terminología devuelve si lo encuentra el objeto terminología correspondiente al identificador. En el caso contrario devuelve un valor nulo.
- **updateTerminology**: Método encargado de modificar una determinada terminología indica en la entrada estándar a la función modificándole los valores de sus atributos y poniendo en su lugar los valores

correspondientes pasados al método. Este método devuelve un valor buleano dependiendo de si se ha llevado a cabo la operación o no.

- **deleteTerminology**: Método que permite eliminar una terminología del sistema. El identificador de la terminología será el único parámetro de entrada. El método devuelvo verdadero o falso en función de si se ha realizado la operación con existo o no se ha encontrado la terminología.
- **insertCS**: Método pensado para introducir un elemento CS en la terminología indicada. Los parámetros de entrada son el identificador de la terminología y los atributos para crear el elemento CS. Antes de crear el objeto se hará una comprobación para ver si la terminología con la que se esta trabajando permite elementos de tipo CS.
- **searchCS**: Permite buscar un elemento CS dentro de una terminología. Para ello se pasan como parámetros de entrada el identificador del elemento y el de la terminología.
- **updateCS**: Método que sirve para modificar un elemento CS dentro de una terminología. Se pasan como parámetros de entrada el identificador del elemento y el de la terminología. Devuelve un booleano con el resultado de la operación; true operación correcta, false ha habido algún problema.
- **deleteCS**: Método que dada un terminología elimina de su lista de elementos el CS indicado en los parámetros de entrada de la función. Esta función devuelve true o false según el resultado de la operación.
- **insertCD**: Método pensado para introducir un elemento CD en la terminología indicada. Los parámetros de entrada son el identificador de la terminología y los atributos para crear el elemento CD. Antes de crear el objeto se hará una comprobación para ver si la terminología con la que se esta trabajando permite elementos de este tipo.
- **searchCD**: Permite buscar un elemento CD dentro de una terminología. Para ello se pasarán los parámetros que identifique ambos conceptos por la entrada de la función. Este método devuelve el objeto CD en caso de encontrarlo o en caso de no hacerlo devuelve nulo.
- **updateCD**: Método para modificar un elemento CD que esta contenido dentro de la terminología indicada. Los parámetros de entrada son el id de la terminología, el identificador del CD y los nuevos valores de los atributos del elemento CD.
- **deleteCD**: Método que dada un terminología elimina de su colección de elementos el CD indicado en los parámetros de entrada de la función. Esta

función devuelve true o false según el resultado de la operación.

- **insertCDCV**: Método para insertar un nuevo CDCV en una terminología ya existente. Se pasan como parámetros de entrada el id de la terminología y los atributos para crear el CDCV.
- **searchCDCV**: Método que permite buscar un elemento CDCV en una terminología a través del identificador del elemento y de la terminología.
- **updateCDCV**: Método que sirve para modificar un elemento CD dentro de una terminología. Se pasan como parámetros de entrada el identificador del elemento y el de la terminología. Devuelve un booleano con el resultado de la operación.
- **deleteCDCV**: Función pensada par eliminar un elemento CDCV de una terminología en concreto. Por la entrada estándar del método se pasan los identificadores del CDCV y de la terminología. La función indicará si la operación se ha realizado con éxito mediante la salida estándar.

CAPÍTULO 4: IMPLEMENTACIÓN DEL PROYECTO.

Tras el exhaustivo análisis realizado y el posterior diseño puede comenzar la etapa de implementación. La implementación del sistema de gestión y consulta de terminologías se podría dividir en tres partes. Una primera fase en la que se implementarían los objetos que representan a las terminologías y a los tipos de datos de codificación, otra fase en la que se llevaría a cabo la definición de una base de datos donde almacenar las instancias de los objetos y por último la implementación de métodos para consultar a esa base de datos, métodos que representarían cada una de la funcionalidades ofrecidas en la interfaz de usuario.

A continuación se desglosan esas fases en las que se divide la etapa de implementación.

4.1 - Fase 1: implementación objetos java.

JAVA es el lenguaje de programación escogido para la representación de los objetos de este proyecto. El motivo principal es que el proyecto desarrollado forma parte de un proyecto mayor LinkEHR [6][7] usa este lenguaje. Por tanto y puesto que la intención es integrar el proyecto de consulta y gestión de pequeñas terminologías clínicas, la implementación se ha hecho mediante ese lenguaje. Es necesario que fuese un lenguaje de programación orientado a objetos. Además es un lenguaje de creación de software multiplataforma lo cual proporciona la ventaja de poder funcionar en diversas plataformas.

Los objetos a implementar representan los conceptos de terminología y de tipos de datos para la codificación de estas. Partiendo de la representación de estos conceptos se realiza en la fase de análisis, concretamente en el diagrama de clases de la aplicación, crearemos una clase para cada uno de ellos.

Terminology: Clase pública que recoge la información necesaria acerca de una terminología. Con atributos TerminologyId que la identifica, Name, Version los tres de tipo String y una colección de objetos de codificación de tipo CD, CS o CDCV. Esta colección sólo puede contener elementos del mismo tipo.

HXIT: Clase pública y abstracta por tanto no instanciable y que servirá para complementar la información de codificación de las clases que se especializan de ella.

Los atributos de esta clase son: validTimeLow,, validTimeHigh, controlActRoot y controlActExtension todos de tipo String.

ANY: Clase pública y abstracta que se especializa de HXIT y complementa la información de esta. Además de los atributos heredados de su clase padre añade flavorId, nullFlavor y updateMode todos de tipo String. Los dos últimos tienen su valor limitado y esto queda patente en sus métodos de modificado (setter).

CD: Clase pública que se especializa de ANY. CD son la siglas de Descriptor de Concepto. Esta clase incluye los siguientes atributos: code, codeSystem, codeSytemName, codeSystemVersion, valueSet, valueSetVersión, codingRationale, originalText, displayName, todos estos de tipo String, Source de tipo CD y una colección de elementos CD que son traducciones de este tipo de datos.

CS: Representac una Codificación Simple con respecto a la que ofrece CD. Solo añade un nuevo atributo con nombre *code* de tipo String.

CDCV: Clase pública que se especializa de CD. Esta clase es igual que su clase padre salvo que pone a nulo el atributo codingRationale y la colección de traducciones también.

Para todas estas clases los atributos serán privados y podrán ser consultados o modificados mediante los métodos *setters* y *getters* implementados. A parte de estos métodos estas clases no incluyen ningún método adicional. Las seis clases incluyen un atributo adicional de tipo Long el cual se explicará en el apartado de implementación de la base de datos.

Tras este trabajo se da por implementada la jerarquía de tipos de datos y la relación de agregación existente entre la clase Terminology y los tipos de datos de codificación.

4.2 - Fase 2: Base de datos.

A la hora de determinar el sistema gestor de base de datos más adecuado para el almacenamiento de las terminologías se ha de tener en cuenta dos criterios: el sistema gestor debe funcionar como una librería java y que no será necesario almacenar una gran cantidad de datos. El sistema gestor escogido es HyperSQL [8] debido

principalmente a que está desarrollo en Java, y es una buena opción para trabajar con bases de datos con este mismo lenguaje, además es software libre. HyperSQL es un motor de base de datos que tiene la posibilidad de funcionar en memoria. En los casos en los que se necesite una forma rápida y sencilla de probar algún proyecto, es bueno tener a mano una herramienta que no requiera de una infraestructura demasiado compleja y que sea fácil de utilizar.

La implementación de esta base de datos no se ha hecho de forma convencional, la idea era facilitar el trabajo de programación mediante archivos de mapeo que partiendo de los objetos java implementados crearán las entidades y las relaciones de la base de datos. Ha sido posible gracias al uso de la librería Hibernate [9]. Esta librería es compatible con java debido a que este lenguaje permite el uso de librerías. La ventaja que ofrece el hecho de crear de este modo la base de datos es que cualquier variación en los objetos java la repercusión a la base de datos se reduce a una modificación de un pedazo de código en los archivos de mapping.

Por tanto, para el sistema de almacenamiento y gestión de datos serán necesario un archivo de mapping para representar las entidades y sus relaciones, la librería Hibernate la cual se añadirá al proyecto java y una nueva clase que llamaremos Manager / Manejador. Esta nueva clase contendrá los métodos de la interfaz de usuario y el método *main* que permite poner en funcionamiento Hibernate y por consiguiente la base de datos.

Hibernate requiere dos pequeñas modificaciones en las clases que definen los objetos: se añadirá un nuevo atributo “Id” de tipo Long el cual supone un valor identificador único para un evento en particular. Todas las clases de entidad persistentes necesitarán tal propiedad identificadora si queremos utilizar el grupo completo de funcionalidades de Hibernate (también algunas clases dependientes menos importantes); La segunda modificación es añadir un constructor sin argumentos, es un requerimiento para todas las clases persistentes, Hibernate tiene que crear objetos utilizando Java Reflection.

4.2.1 - Hibernate.

Hibernate es una herramienta de Mapping objeto-relacional para Java que facilita la definición de correspondencias entre los atributos y tablas de una base de datos relacional y el modelo de objetos de una aplicación. Hibernate no sólo se ocupa de la

asignación de clases Java a tablas de bases de datos (y de Tipos de datos Java con tipos de datos SQL), sino que también proporciona consulta de datos y sistemas de recuperación. Por tanto, ayuda reducir significativamente el tiempo de desarrollo que de no hacerse mediante una librería de este tipo se implementaría de forma manual los datos en SQL y JDBC. La meta del diseño de Hibernate es simplificar en un gran porcentaje las tareas relacionadas con la programación de la persistencia de datos, eliminando la necesidad de un manual y una programación artesanal. Sin embargo, a diferencia de muchas otras soluciones de persistencia, Hibernate no oculta el poder de SQL y garantiza que su inversión en tecnología relacional y conocimiento es tan válida como siempre.

Hibernate sin duda puede ayudar a eliminar o encapsular código específico del proveedor SQL y ayudará con la tarea común de traducción del conjunto de resultados desde una representación tabular a un grafo de objetos, como en el caso que nos acontece.

4.2.2 - Archivos de Mapping de Hibernate.

Hibernate necesita saber cómo cargar y almacenar objetos a persistir. En este punto es donde entra en juego el archivo de mapeo de Hibernate. Este archivo le dice a Hibernate a que tabla tiene que acceder en la base de datos, y que columnas debe utilizar en esta tabla.

La estructura básica de un archivo de mapeo consiste en la declaración del DTD y en colocar las etiquetas de inicio y fin del mapeo.

```
<?xml version=”1.0”?>
<!DOCTYPE hibernate-mapping PUBLIC "
        -//Hibernate/Hibernate Mapping DTD 3.0//EN"
         "http://www. hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="org.hibernate.tutorial .domain">
...
</hibernate-mapping>
```

El archivo DTD se encuentra incluido en hibernate-core.jar. El DTD de Hibernate es sofisticado. Puede utilizarlo para autocompletar los elementos y atributos XML de mapeo. en su editor.

Entre las dos etiquetas hibernate-mapping se incluye un elemento “class” por cada una de las entidades a representar. Como hay clases dependientes de otras o especializadas a la clase raíz, HXIT, se le pone la etiqueta “class” y dentro de ella se incluyen las clases derivas de ella mediante etiquetas “joined-class” dando lugar de esta forma a jerarquía de herencia.

Por cada clase a habrá que añadir una serie de etiquetas “*property*” que representan a cada uno de los atributos de la clase en cuestión. El formato de esta etiqueta es el siguiente: <property name="code" type="string" column="Code"/>. Esta etiqueta es para tipos de atributos simples cuando se trata de mapear una colección se emplea una etiqueta especifica según el tipo de colección, por ejemplo para un *HashSet* la etiqueta se llama set. Es el caso de la propiedad *translation* de la clase CD.

```
<set name="translations" inverse="false">
        <key column="translation_id"/>
        <one-to-many class="CD"></one-to-many>
</set>
```

Las colecciones persistentes en Hibernate se comportan como HashMap, HashSet, TreeMap, TreeSet o ArrayList, dependiendo del tipo de interfaz. Las instancias de colecciones tienen el comportamiento usual de los tipos de valor. Son automáticamente persistidas al ser referenciadas por un objeto persistente y se borran automáticamente al desreferenciarse.

Una pieza fundamental en el mapeo mediante Hibernate es el fichero de configuración. Hibernate está diseñado para operar en muchos entornos diferentes y por lo tanto hay un gran número de parámetros de configuración. Afortunadamente, la mayoría tiene valores predeterminados sensibles. Una instancia de org.hibernate.cfg.Configuration representa un conjunto entero de mapeos de los tipos Java de una aplicación a una base de datos SQL. La org.hibernate.cfg.Configuration se utiliza para construir una org.hibernate.SessionFactory inmutable. Los mapeos se compilan desde varios archivos de mapeo XML. Hibernate ofrece un modelo de fichero de configuración desde el cual aplicando pequeñas modificaciones como el nombre de los ficheros de mapping se obtiene el fichero de configuración de la aplicación. El archivo tiene esta forma:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>
      <!-- Database connection settings -->
      <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
<property
name="connection.url">jdbc:hsqldb:file:terminologyDB/terminologyDB;shutdown=true;hsqldb.write_delay=
false;</property>
      <property name="connection.username">admin</property>
      <property name="connection.password">nimda</property>
      <!-- JDBC connection pool (use the built-in) -->
      <property name="connection.pool_size">1</property>
      <!-- SQL dialect -->
      <property name="dialect">org.hibernate.dialect.HSQLDialect</property>
      <!-- Enable Hibernate's automatic session context management -->
      <property name="current_session_context_class">thread</property>
      <!-- Disable the second-level cache  -->
      <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
      <!-- Echo all executed SQL to stdout -->
      <property name="show_sql">true</property>
      <!-- Drop and re-create the database schema on startup -->
      <property name="hbm2ddl.auto">update</property>
      <mapping resource="term.hbm.xml"/>
      <mapping resource="terminology.hbm.xml"/>
   </session-factory>
</hibernate-configuration>
```

Figura 4. Fichero de configuración del mapeo de los objetos mediante Hibernate.

Por último, la figura 5 muestra en detalle el archivo de mapeo necesario para crear el servidor de terminologías a partir de los objetos Java.

4.2.3 -Volcado de datos a un XML.

Hibernate no sólo sirve para crear un almacenaje de objetos java, otra de las múltiples funcionalidades que ofrece es la de exportar el contenido de la base de datos a un fichero XML. Un árbol XML analizado semánticamente se puede considerar como otra manera de representar los datos relacionales a nivel de objetos.

Hibernate soporta dom4j como API para manipular árboles XML. Puede escribir consultas que recuperen árboles dom4j de la base de datos y puede tener cualquier modificación que realice al árbol sincronizada automáticamente con la base de datos. Incluso puede tomar un documento XML, analizarlo sintácticamente utilizando dom4j, y escribirlo a la base de datos con cualquiera de las operaciones básicas de Hibernate.

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

   <!-- Class HXIT, abstract -->
   <class name="HXIT" table="HXIT" abstract="true">

      <id name="id"  column="EVENT_ID" type="long">
         <generator class="native"/>
      </id>
      <property name="validTimeLow" type="string" column="validTimeLow"/>
      <property name="validTimeHigh" type="string" column="validTimeHigh"/>
      <property name="controlInformationRoot" type="string" column="controlInformationRoot"/>
      <property name="controlActExtension" type="string" column="controlActExtension"/>

      <!-- Class ANY, abstract, extends HXIT-->
      <joined-subclass  name="ANY" table="ANYTable" abstract="true" >
         <key column="EVENT_ID"></key>

         <property name="flavorId" type="string" column="FLAVORID"/>
         <property name="validTimeLow" type="string" column="validTimeLow"/>
         <property name="validTimeHigh" type="string" column="validTimeHigh"/>
         <property name="controlInformationRoot" type="string" column="controlInformationRoot"/>
         <property name="controlActExtension" type="string" column="controlActExtension"/>

         <!-- Class CS extends ANY-->
         <joined-subclass name="CS" table="CS">
            <key column="EVENT_ID"></key>
            <property name="code" type="string" column="Code"/>
         </joined-subclass>

         <!-- Class CD extends ANY-->
         <joined-subclass name="CD" table="CD">

            <key column="EVENT_ID"></key>

            <property name="code" type="string" column="Code"/>
            <property name="codeSystem" type="string" column="CodeSystem"/>
            <property name="codeSystemName" type="string" column="CodeSystemName"/>
            <property name="codeSystemVersion" type="string" column="CodeSystemVersion"/>
            <property name="valueSet" type="string" column="ValueSet"/>
            <property name="valueSetVersion" type="string" column="ValueSetVersion"/>
            <property name="displayName" type="string" column="DisplayName"/>
            <property name="originalText" type="string" column="OriginalText"/>
               <property name="codingRationale" type="string" column="CodingRationale"/>

            <set name="translations" inverse="false">
               <key column="translation_id"/>
               <one-to-many class="CD"></one-to-many>
            </set>

            <!-- <property name="source" type="CD" column="Source"/>-->

            <!-- Class CDCV extends CD-->
            <joined-subclass name="CDCV" table="CDCV">
             <key column="EVENT_ID"></key>
                                          </joined-subclass>

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

Figura 5. Archivos de mapping de Hibernate.

El igual que con los objetos java serán necesarios unos ficheros de mapeo de la estructura de los objetos. Se disponen dos ficheros de mapping uno para la clase *Terminology* y otro para la jerarquía de herencia que representa a los tipos de datos de codificación para las terminologías. Estos ficheros no difieren mucho de los creados para los objetos java. La principal diferencia es que se añade “node” junto a “name” en cada etiqueta (<class **name**="Customer", **table**="CUSTOMER", **node**="customer">).

Aparte de los ficheros de mapeo será necesaria una clase que de forma al árbol de representación de la estructura XML. En esta clase se incluye un método que recorrerá las terminologías e irá añadiendo la información de cada una de ellas, recuperando los datos mediante consultas a la base de datos y las propiedades de los elementos, a un String. Esta función será llamada desde el método *main* de la clase Manager. El método descrito se llama exportToXML y se detalla a continuación:

```
public String exportToXML() {

            String result = "<TERMINOLOGIES>\n";
            Terminology t = new Terminology();
             session = HibernateUtil.getSessionFactory().openSession();
            Transaction tx = null;

            for(int i = 0; i < terminologies.size(); i++){
               t = (Terminology) terminologies.get(i);
               result += "<TERMINILOGY>\n<TerminologyId>"+t.getTerminologyId()+"</TerminologyId>/n" +
                           "<Name>"+t.getName()+"</Name>\n<Version>"+t.getVersion()+"</Version>\n"+
                           "<Elemenents>\n";
                        try {
                           tx = session.beginTransaction();
                           Iterator it = t.getElements().iterator();
                           result +="<"+t.getElementType()+">";
                           Element element = null;
                           while (it.hasNext()) {
                              element = (Element)it.next();
                              result += element.asXML();
                           }
                           tx.commit();
                        }
                        catch (RuntimeException e) {
                           if (tx != null) tx.rollback();
                           System.err.println("No se ha podido exportar la base de datos a un xml");
                           return "";
                        }
                        result += "\n</Elements>\n</TERMINOLOGY>";
             }
             result += "</TERMINOLOGIES>";
             session.close();
             return result;
  }
```

Figura 6. Método para el volcado de la base de datos.

4.3 - Fase 3: Métodos para la interfaz de usuario.

En el segundo capítulo de la presente memoria se describen los diferentes métodos y consultas que formarán la interfaz de usuario de este sistema gestor de terminologías. En este apartado se va a describir la implementación de los métodos agrupados en los siguientes grupo: métodos de inserción, de búsqueda, de modificado y de borrado. La interfaz se compone de los siguientes métodos: getAllTerminoligies, getAllElementsByTerminology, searchElementIntoTerminilogy, createTerminology, searchTermonology, updateTerminology, deleteTerminology, insertCS, searchCS, updateCS, deleteCS, insertCD, searchCD, updateCD, deleteCD, insertCDCV, searchCDCV, updateCDCV y deleteCDCV.

Métodos de inserción: Mediante los parámetros de entrada se identifica la terminología en la cual se va a insertar y se crea el elemento a insertar. Una vez recuperado el objeto terminología se comprueba que el tipo de datos que se intenta añadir a ella es el tipo de datos permitido por la terminología. Se creará el objeto y se incluirá en la lista de tipos de codificación. En la Figura 7 se muestra un ejemplo de método para la inserción.

Métodos de búsqueda: Estos métodos comprueban si un elemento se encuentra dentro de una terminología y en caso de encontrarse en ella lo devuelven. Este método se apoya en otro método implementado, searchElementIntoTerminilogy como se muestra en la Figura 8.

Métodos de modificado: Este tipo de métodos se emplea para modificar un elemento contenido en una terminología, ambos son parámetros de entrada del método. Se busca el elemento y si se encuentra se modifica. Un ejemplo de ello es el método updateCS que se muestra en la Figura 9.

Métodos de borrado: Su utilidad es eliminar un elemento dentro de la lista de tipos de datos para la codificación en una determinada terminología, un ejemplo se muestra en la Figura 10.

```
public boolean insertCS(String terminologyId, String code, String nullFlavor, String updateMode,
                String flavorId, String validTimeLow, String validTimeHigh,
                String controlInformationRoot,  String controlActExtension)
  {
          session = HibernateUtil.getSessionFactory().openSession();
          Transaction tx = null;
          /*Se busca y se comprueba que existe la terminologia*/
          Terminology t = searchTerminology(terminologyId);
          if(t == null){
            System.err.println("Terminologia no encontrada: Error al recuperar la terminologia: "+terminologyId);
            return false;
           }
          /*Comprobación del tipo de datos permitido por la termiologia*/
          if(t.getElementType().equals("") || t.getElementType().equals("CS")){
                    t.setElementType("CS");
                    try {
                      tx = session.beginTransaction();
                      CS elementCS = new CS();                    /*Creación del elemento CS*/
                      elementCS.setCode(code);
                      elementCS.setControlActExtension(controlActExtension);
                      elementCS.setControlInformationRoot(controlInformationRoot);
                      elementCS.setFlavorId(flavorId);
                      elementCS.setNullFlavor(nullFlavor);
                      elementCS.setUpdateMode(updateMode);
                      elementCS.setValidTimeHigh(validTimeHigh);
                      elementCS.setValidTimeLow(validTimeLow);
                      session.save(elementCS);          /*guarda en la base de datos*/

                      tx.commit();
                        return true;
                    }
                    catch (RuntimeException e) {
                       if (tx != null) tx.rollback();
                       System.err.println("Error al insertar el CS en la terminologia: "+terminologyId);
                       return false;
                    }
                    finally {session.close();}
          } else
          {
          System.err.println("Error al insertar el CS en la terminologia: "+terminologyId+". Esta terminologia no
          admite CSs.");
           return false;
          }
  }
```

Figura 7. Ejemplo de implementación de un método de inserción.

```
public CS searchCS(Long id, String terminologyId)
{
          if(searchTerminology(terminologyId).getElementType().equals("CS")){
                    return (CS)searchElementIntoTerminology(terminologyId, id);
          }
          else {
                    System.err.println("No se encuentra el CS solicitado.");
                    return null;
          }
}
```

Figura 8. Ejemplo de implementación de un método de búsqueda.

```
public boolean updateCS(Long id, String terminologyId, String code)
  {
        session = HibernateUtil.getSessionFactory().openSession();
        Transaction tx = null;

        Terminology t = searchTerminology(terminologyId);
        if(t == null) return false;
          try {
        tx = session.beginTransaction();
                CS elementCS = searchCS(id, terminologyId);
                if(elementCS == null) return false;

                elementCS.setCode(code);
                session.saveOrUpdate(elementCS);
        tx.commit();
                return true;
            }  catch (RuntimeException e) {
                if (tx != null) tx.rollback();
                System.err.println("Error al modificar el CS "+id+" en la terminologia: "+terminologyId);
                return false;
              }
              finally {session.close();}
  }
```

Figura 9. Ejemplo de implementación de un método de modificado.

```
public boolean deleteCS(Long id, String terminologyId)
  {
        session = HibernateUtil.getSessionFactory().openSession();
        Transaction tx = null;

        Terminology t = searchTerminology(terminologyId);
        if(t == null) return false;
          try {
                tx = session.beginTransaction();
                CS elementCS = searchCS(id, terminologyId);
                session.delete(elementCS);

                tx.commit();
                return true;
              }
              catch (RuntimeException e) {
                if (tx != null) tx.rollback();
                System.err.println("Error al eliminar el CS "+id+" en la terminologia: "+terminologyId);
                return false;
              }
              finally {
          session.close();
              }
  }
```

Figura 10. Ejemplo de implementación de un método de borrado.

Métodos de listado: Este tipo devuelven una lista de objetos en función de la consulta realizada a la base de datos. Podemos hacer cualquier consulta mediante la función createQuery de la clase *Session* de Hibernate como se muestra en la Figura 11.

```
public HashSet<Terminology> getAllTecnologies()
  {
        return (HashSet<Terminology>)session.createQuery("from Terminology");
  }
```

Figura 11. Ejemplo de implementación de un método de listado.

CAPÍTULO 5: PRUEBAS DE FUNCIONAMIENTO.

Llegados a este punto la implementación del módulo de gestión y consulta de pequeñas terminologías se da por concluida pero no se puede decir que el software este terminado. Es necesario realizar una serie de pruebas para comprobar el correcto funcionamiento de la aplicación y el cumplimiento de todos los requisitos especificados en la fase de análisis.

Partiendo de los requerimientos especificados para esta aplicación se deben diseñar una serie de casos de estudio y una planificación de pruebas para cada uno. Tras determinar los casos a probar y llevar a cabo las pruebas el resultado puede desencadenar en la necesidad de depurar el código o en la confirmación de que es correcto.

Cada caso de prueba consiste en un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular a estudiar.

5.1 - Pruebas funcionales.

Determinar el correcto funcionamiento de la aplicación dependerá de confirmar que los siguientes cuatro puntos son correctos: la implementación de los objetos java que representan los conceptos de terminologías y los arquetipos, verificar el correcto mapeo de los objetos java para formar la base de datos, comprobar los cinco tipos de consultas de la interfaz de usuario y cerciorarse de que se puede exportar el contenido de la base de datos a un fichero XML.

El primer caso de estudio consiste en la comprobación de los objetos java creados en representación de los conceptos de terminología y arquetipos. Verificar la correcta implementación de dichos objetos se limita a comprobar que las clases representan exactamente los conceptos según la especificación y con sus debidas restricciones, y comprobar la sintaxis de la implementación para que se corresponda con la del lenguaje java.

El resto de comprobaciones se realizaran a partir de la ejecución de diferentes métodos en el *main* de la clase Manager verificando que se cumplan los requisitos de la aplicación y que no den lugar a excepciones o errores no controlados.

La comprobación del mapeo de objetos mediante Hibernate y la correcta implementación de las consultas van ligadas. Las pruebas consistirán en la creación de diferentes objetos, usando los métodos creados, y la manipulación de estos. Para ello se ejecutarán una serie de órdenes desde la raíz de la aplicación y mediante un gestor de base de datos se observarán los resultados de la ejecución. Esto permite comprobar la estructura de tablas con sus relaciones derivada de los objetos java junto con los archivos de mapeo y la correcta implementación de los métodos para la gestión y consulta a la base de datos.

1 Ejecución de las siguientes órdenes:

```
/*Creación de dos terminologías*/
mgr.createTerminology("100", "terminologia1", "1.1");
 mgr.createTerminology("200", "terminologia2", "1.1");
 /*Comprobación de método que devuelve las terminologías*/
HashSet<Terminology> list = mgr.getAllTecnologies();
for (int i = 0; i<list.size(); i++) {
            System.out.println("TEMINOLOGIA" + i+"\n");
}

CD source = new CD("230","codeSystem", "codeSystemName",  "codeSystemVersion","valueSet",
"valueSetVersion","displayName",  "originalText", "", new HashSet<CD>(), null,"", "", "", "", "", "", "");
 /* Inserción de diferentes arquetipos en las terminologías */
mgr.insertCS("100", "234.3", "", "", "", "", "", "", "");
mgr.insertCS("100", "456.1", "", "", "", "", "", "", "");
mgr.insertCS("300", "234.3", "", "", "", "", "", "", "");
mgr.insertCD("100", "789", "codeSystem", "codeSystemName", "codeSystemVersion", "valueSet", "valueSetVersion",
displayName",  "originalText", source, "O");
mgr.insertCD("200","789", "codeSystem", "codeSystemName", "codeSystemVersion",  "valueSet", "valueSetVersion",
"displayName",  "originalText", source, "O");
```

2 Comprobación mediante el gestor de base de datos(RazorSql en este caso):

La inserción de las terminologías es correcta y aparecen en su correspondiente tabla. Las inserciones de los dos primeros arquetipos y el cuarto son correctas pero el tercero y el cuarto producen errores que son capturados por el método. En el primer caso la terminología indicada no se encuentra en la base de datos y en el segundo se intenta insertar un elemento CD en una terminología que solo admite CSs.

Figura 12. Pruebas de funcionamiento de los métodos.

La figura 12 muestra el proceso de prueba de algunas de las consultas implementadas. La prueba comienza con la creación de varias terminologías mediante

el método “createTerminology”. La inserción de estas terminologías al servidor se comprueba mediante las cuatro líneas de código posteriores a la creación, además al mismo tiempo se comprueba el funcionamiento de la función de listado “getAllTerminologies”. A continuación se insertan arquetipos a las terminologías probando diferentes casos de estudio: qué ocurre si se intenta insertar un arquetipo en una terminología que no está creada, qué sucede si se intenta insertar un tipo de datos no permitido por la terminología, etc. La figura 12 sólo muestra las pruebas de inserción y listado pero del mismo modo se ha realizado las de búsqueda, modificación y borrado.

Por último se debe probar el volcado de datos de la base de datos de terminologías a un fichero XML. Esta tarea es esencial puesto que proporciona un copia de seguridad de los datos almacenados. Para su comprobación y verificado del correcto funcionamiento se debe ejecutar el método exportToXML, desde la clase Manager, y comparar la estructura de datos resultante con el contenido de la base de dados. Es importante comprobar que se imprimen la etiquetas adecuadas, en el orden correcto, con sus correspondientes cierres y cumpliendo las reglas de un fichero XML. La estructura general del documento XML resultante del volcado se muestra en la Figura 13.

```
<TERMINOLOGIES>
        <TERMINOLOGY>
                <TerminologyID>...<TerminologyID>
                <Name>...<Name>
                <Version>...<Version>
                <Archetypes>
                        <CS>
                                … información del elemento CS...
                        </CS>
                        … los demás arquetipos ...
                </Archetypes>
        </TERMINOLOGY>
        … resto de terminologías de la base de datos ...
</TERMINOLOGIES>
```

Figura 13. Estructura del fichero de volcado del servidor de terminologías.

CAPÍTULO 6: CONCLUSIONES.

6.1 - Conclusiones sobre el trabajo realizado.

El desarrollado de la aplicación para la gestión y consulta de pequeñas terminologías clínicas se ha llevado a cabo cumpliendo con los requisitos especificados de la manera más eficiente gracias a la combinación de tecnologías y lenguajes empleada. Se ha querido seguir un modelo de desarrollo divido en cuatro partes planificación, diseño, implementación y pruebas lo cual da lugar a un trabajo eficiente, bien calculado y meticulosamente probado. La base de un buen desarrollo de un proyecto está en la planificación y en el posterior diseño, puesto que se ha dedicado la mitad del tiempo a planificación y dado el resultado se puede concluir que ha sido todo un acierto.

Hibernate ha supuesto también un gran punto a favor de este proyecto. La combinación Java, Hibernate y Hypersql ha sido clave en la implementación y desarrollado de este módulo. Se ha tenido que estudiar la librería Hibernate desde cero y estudiar el sistema gestor de base de datos Hypersql ya que no nos queríamos limitar a los conocimientos ya adquiridos si no que la idea era explorar y aprender algo nuevo a la vez que se realizaba este proyecto final de carrera.

6.2 - Conclusiones personales.

La experiencia de haber desarrollado este proyecto con los miembros del departamento IBIME del instituto ITACA ha sido grata y es de agradecer el haberme otorgado esta oportunidad de ampliar mi formación con ellos y el poder colaborar en su proyecto LinkEHR realizando este proyecto final de carrera el cual se integrará y formará parte de él.

Como ya se ha mencionado anteriormente el desarrollo de este módulo ha supuesto un trabajo por mi parte para aprender a usar Hibernate lo cual supone para mi una ventaja de cara a futuros proyectos.

Este trabajo ha supuesto también el conocer y experimentar el hecho de estar en un departamento con un grupo de trabajo y un director que te ayuden, te guíen y te controlen el trabajo que vas realizando.

Por tanto y para concluir que muy satisfecho con el trabajo realizado, con los conocimientos aprendidos durante su desarrollado y con la dirección de este proyecto final de carrera.

6.3 - Posibles ampliaciones y mejoras.

El trabajo desarrollado en el presente proyecto puede ser continuado en distintas direcciones con el fin de ofrecer nuevas funcionalidades a las aplicaciones clientes como el editor de arquetipos LinkEHR del grupo de Informática Biomédica del Instituto ITACA. Estas nuevas funcionalidades son:

- Diseñar e implementar una interfaz gráfica para la gestión de las terminologías integrada con el editor de arquetipos LinkEHR.
- Aumentar la metainformación sobre las terminologías gestionadas para soportar la existencia de múltiples versiones de una misma terminología. Actualmente, es posible tener distintas versiones de una misma terminología pero son tratadas con terminologías independientes y por tanto sin relación entre ellas.
- Desarrollar un módulo que facilite la importación de terminologías expresadas en un XML propietario, es decir, no conforme con ISO 21090.

CAPÍTULO 7: BIBLIOGRAFÍA.

[1] CEN 13606-2: 2008. Health informatics – Electronic health record communication—Part 2: Archetypes.

[2] ISO 13606-1:2008. Health informatics - Electronic health record communication.

[3] http://www.openehr.org/releases/1.0.2/roadmap.html. Último acceso 20/09/2012.

[4] Health Level Seen International, "HL7 Clinical Document Architecture Release 2.0 (CDA R2)". Disponible en: http://www.hl7.org/implement/standards/cda.cfm. Último acceso: septiembre 2012.

[5] ISO 21090:2011. Health informatics – Harmonized data types for information interchange.

[6] Maldonado JA, Moner D, Boscá D, Fernández-Breis, JT, Angulo C, Robles M. LinkEHR-Ed: A multi-reference model archetype editor based on formal semantics. Int. Journal of Medical Informatics, 2009; 78(8):559-570.

[7] Maldonado JA, Moner D, Boscá D, Angulo C, Marco L, Reig E, Robles. Concept-based exchange of healthcare information: the LinkEHR approach. Proceedings book of the First HBSI IEEE conference, pp. 150-157, Julio 2011.

[8] El manual de uso de HSQLDB. Disponible en: http://hsqldb.org/doc/2.0/guide/index.html. Último acceso: septiembre 2012.

[9] Hibernate Core Reference Manual. Disponible en: http://www.hibernate.org/docs.html. Último acceso: septiembre 2012.