

Ambiente de Desarrollo de Software Orientado a Objetos: Un Metamodelo para la Administración de Versiones.

Silvio Gonnet^{*†}, Hernán Melgratti^{*†} y Horacio Leone^{*†}

^{*}GIPSI- Dpto. Sistemas – Facultad Regional Santa Fe – Universidad Tecnológica Nacional – Argentina.
Lavaise 610. (3000) Santa Fe

[†]INGAR – CONICET – Fundación ARCIEN

[‡]Departamento de Computación – FCEyN – Universidad de Buenos Aires – Argentina
E-mail: hleone@alpha.arcrde.edu.ar

Resumen

En el presente trabajo se desarrolla el metamodelo por medio del cual se proveen capacidades para realizar el manejo de versiones de modelos en una herramienta que brinde soporte a las tareas de análisis y diseño de sistemas de información empleando modelos de objetos. El metamodelo será definido empleando el paradigma de objetos y se utiliza una sintaxis de cálculo situacional para expresar formalmente la existencia de un objeto, y su estructura, en una determinada versión del modelo. Uno de los principales objetivos es mantener consistencia y navegabilidad entre los distintos modelos y versiones de los mismos, los cuales se generan durante el proceso de desarrollo de software. La especialización del metamodelo permitirá utilizar una metodología de desarrollo dada con el grado de desagregación deseado.

Palabras claves: administración de versiones, cálculo situacional, proceso de producción de software, análisis y diseño orientado a objetos.

1. Introducción

En este trabajo se describen los avances alcanzados en la especificación de un esquema básico que permite mantener los distintos modelos, y sus versiones, que deben ser administrados en un ambiente computacional para ingeniería de software, fundamentalmente empleando un enfoque basado en objetos. Uno de los principales objetivos en un ambiente para ingeniería de software es poder establecer y mantener consistencia entre los múltiples modelos (y sus componentes) que se generan a lo largo del proceso de producción de software (Groth y otros, 1994). La satisfacción de este requerimiento para un ambiente de producción de software debe analizarse en las distintas dimensiones en que se crean los modelos y conceptos a rastrear a lo largo del proceso (Corriveau, 1996).

En primer lugar se puede mencionar que, obviando sus diferencias, las distintas metodologías de análisis y diseño orientado a objetos expresan de diversas formas: la estructura del sistema, a partir de clases y sus relaciones (diagramas de clases, etc.), la interacción entre objetos (diagramas de interacción, de objetos, etc.) y la estructura de comportamiento interno de cada objeto (diagrama de transición de estados, etc.). El conjunto de diagramas y modelos correspondiente a una etapa del proceso de desarrollo de software, representa de alguna manera una de las dimensiones en las cuales es necesario mantener la mencionada navegabilidad y consistencia, la cual en principio es satisfecha fundamentalmente por la metodología empleada.

Una segunda dimensión a considerar es la originada por las distintas *etapas* o *fases* en que se compone el proceso de desarrollo (análisis, diseño, testing, implementación, etc., las cuales pueden ser subdivididas en más sub-etapas). A lo largo de esta dimensión se producen creaciones, alteraciones, modificaciones, especializaciones o eliminaciones, de los modelos y conceptos empleados en cada fase, para las cuales se requiere mantener las relaciones que permitan rastrear entre fases lo que ha ocurrido. En general el

mantenimiento de las relaciones que permiten navegar en esta dimensión no se encuentran representadas en las metodologías.

Finalmente, una tercera dimensión surge al considerar que el proceso de desarrollo de software basado en el paradigma de objetos es inherentemente iterativo (Booch, 1994). En cada iteración se incluye la identificación de requerimientos, análisis, diseño, implementación y testeó. Esta secuencia de etapas, que se repite en cada iteración, puede caracterizarse como un ciclo de vida reducido del sistema. En cada iteración se subsanan errores y se incrementa la comprensión que se tiene del problema. Sin embargo, a medida que se avanza en el proceso iterativo, la secuencia debe converger hacia la solución deseada y por lo tanto cada iteración aumenta su dependencia de la iteración anterior. Aquí surge nuevamente la necesidad de mantener consistencia y navegabilidad entre los modelos y conceptos que se identifican en cada iteración, requerimiento que en general no está considerado en las metodologías.

En el presente trabajo se desarrolla un esquema básico que soportará el manejo de las versiones y alteraciones de modelos que se generan en la segunda y tercera dimensión en el proceso de desarrollo de software, en una herramienta de soporte al mismo empleando el modelo de objetos. El metamodelo se especializará de acuerdo a los componentes que considere la metodología en particular que se emplee (por ej.: clases, objetos, atributos, métodos, mensajes, relaciones, etc.), así como las etapas que se consideren para administrar el ciclo de vida del software. En este punto se pretende establecer una estructura sobre la cual luego se impondrán las restricciones referidas a la metodología y al ciclo de vida que se deseen emplear.

2. Conceptos Generales

A continuación se desarrolla el esquema propuesto para la administración de versiones de modelos. En éste se considera que cada versión se genera a partir de una versión precedente, sobre la cual se han realizado una serie de modificaciones que en general implican la eliminación, creación, modificación, etc., de los componentes del modelo (objetos, clases, relaciones, etc.). A partir de una versión inicial, pueden construirse sucesivas versiones de la misma. Esta evolución se concibe como una historia compuesta por situaciones discretas, tomándose el cálculo situacional para el modelado del proceso de generación de versiones. El cálculo situacional fue propuesto originalmente por McCarthy y Hayes en 1962, como un lenguaje de primer orden para razonar acerca de acciones, y si bien ha recibido muchas críticas acerca de su falta de generalidad para representar problemas del mundo real, recientes trabajos están revirtiendo estas críticas (Pinto y Reiter, 1993). La ontología básica del cálculo consiste en *situaciones*, las cuales corresponden a fotografías del universo estudiado en un instante dado, y *acciones*, que producen evoluciones de una *situación* a otra. Por lo tanto, se considera que cada nueva *situación* es generada a partir de una precedente por la aplicación de una *acción*, en el contexto dado por el proceso de producción de software, cada nueva versión que se genera se puede asociar con una *situación*, y una *acción* será el conjunto de operaciones que se realizaron sobre la versión precedente.

3. Esquema Propuesto

Considerando el enfoque presentado en el punto 2, la transformación entre versiones consecutivas de modelos puede ser descrita a través de una función que a una determinada versión de modelo y una secuencia de operaciones le hace corresponder como imagen una nueva versión de modelo. Llamaremos *aplicar* a dicha función, y es la que en el cálculo situacional, dada una *situación* y una *acción*, denota la *situación* resultante.

$$\text{aplicar: } \Phi \times M \rightarrow M \quad (1.)$$

donde:

Φ : es el conjunto de todas las secuencias de operaciones posibles

M : el conjunto de versiones de modelos posibles.

m_0 : versión de modelo inicial ($m_0 = \emptyset$)

Por ejemplo:

$$aplicar(\phi_1, m_1) = m_2; \text{ donde } \phi_1 \in \Phi; m_1, m_2 \in M$$

Una secuencia de operaciones se define de la siguiente manera:

$$\phi = \begin{cases} \lambda \text{ secuencia vacía} \\ o \bullet \phi \text{ donde } o \text{ es una operación} \end{cases} \quad (2.)$$

A partir de la expresión 2. se puede definir por inducción a la función *aplicar*:

$$\begin{aligned} aplicar(\lambda, m) &= m \\ aplicar(o \bullet \lambda, m) &= m', \quad m \neq m' \\ aplicar(o \bullet \phi, m) &= aplicar(\phi, aplicar(o \bullet \lambda, m)) \end{aligned}$$

4. Operaciones Básicas

El término *operación* hace referencia a alguna acción primitiva que modifica a una versión de modelo. Se denota genéricamente a cada componente de una versión de modelo con el nombre *Versión de Entidad* (v_e). En este contexto una versión de modelo se define como un conjunto de versiones de entidad. En consecuencia, una *operación* se define como una transformación de conjuntos. Las operaciones primitivas propuestas inicialmente para representar la transformación de versiones de modelos son: *Agregar*, *Borrar*, *Simplificar*, *Refinar* y *Redefinir*. Las primeras cuatro operaciones permiten incorporar o eliminar conceptos de una versión de modelo, en tanto la última permite crear nuevas versiones de entidades existentes. En principio podrían definirse a las operaciones *Simplificar* y *Refinar* en términos de *Agregar* y *Borrar*, la distinción se realiza, sin embargo, con el objeto de mantener la historia de los cambios realizados en la versión de modelo.

A través de la operación *Agregar* se puede incorporar a una versión de modelo una v_e inexistente en la versión de modelo anterior. Inversamente, la operación *Borrar* elimina una v_e existente en la versión de modelo anterior. Una operación frecuente en la generación de modelos es la que permite detallar una entidad a partir de su descomposición en una estructura de nuevas entidades. En este contexto tal operación es denominada *Refinar*, y la misma permite que en el nuevo modelo aparezcan las v_e que desarrollan el refinamiento. La operación inversa a *Refinar* es *Simplificar*, mediante la cual una estructura de entidades (conjunto de v_e) se condensan en una v_e .

A partir de estas definiciones, se especificará formalmente cuando una v_e pertenece a una versión de modelo. El predicado $Pertenece(v_e, m)$ es verdadero cuando v_e pertenece a la versión de modelo m . El predicado 3. establece que el modelo inicial es una versión de modelo vacía.

$$\forall v_e \neg Pertenece(v_e, m_0) \quad (3.)$$

La expresión 4. establece la pertenencia de una v_e , utilizando el formato de los axiomas de estado sucesor propuesto por Reiter (1991). v_e pertenecerá al modelo que surge de aplicar ϕ al modelo m si y solo si v_e pertenece a m y no se borra al aplicar ϕ o si v_e se agrega al generar la nueva versión de modelo.

$$\begin{aligned} \forall \phi, v_e, m \quad & Pertenece(v_e, aplicar(\phi, m)) \\ \Leftrightarrow & \\ ((Pertenece(v_e, m) \vee (agregar(v_e) \in \phi)) \wedge borrar(v_e) \notin \phi) & \end{aligned} \quad (4.)$$

Refinamiento(v_e, ψ): denota la relación existente entre una v_e de una versión de modelo m y un conjunto de versiones de entidades ψ que pertenecen a una versión de modelo generado a partir de m . La expresión 5. expresa formalmente el concepto de *Refinamiento* y su vinculación con la operación *refinar*.

$$\begin{aligned} \forall \phi, v_e, m_i, \psi, \text{Refinamiento}(v_e, \psi) \wedge \text{Pertenece}(v_e, m_i) \wedge \\ \psi \subset \text{aplicar}(\phi, m_i) \wedge (\psi \cap m_i) = \emptyset \\ \Leftrightarrow \\ \text{refinar}(v_e, \psi) \in \phi \wedge \neg \text{Pertenece}(v_e, \text{aplicar}(\phi, m_i)) \end{aligned} \quad (5.)$$

Simplificación(ψ, v_e): denota la relación existente entre un conjunto de versiones de entidad ψ pertenecientes a una versión de modelo m y una versión de entidad v_e definida en una versión de modelo sucesor a m .

$$\begin{aligned} \forall \phi, v_e, m_i, \psi, \text{Simplificación}(\psi, v_e) \wedge \psi \subset m_i \\ \wedge \text{Pertenece}(v_e, \text{aplicar}(\phi, m_i)) \wedge \neg \text{Pertenece}(v_e, m_i) \\ \Leftrightarrow \\ \text{simplificar}(\psi, v_e) \in \phi \wedge (\psi \cap \text{aplicar}(\phi, m_i)) = \emptyset \end{aligned} \quad (6.)$$

Redefinición(v_e, v_e'): denota la relación entre una v_e que pertenece a una versión de modelo m_i y la versión de entidad que resulta de su redefinición v_e' , la cual pertenece a una versión de modelo sucesor de m_i . La expresión 7. expresa como se vincula la relación *Redefinición* con la operación *redefinir*.

$$\begin{aligned} \forall \phi, v_e, v_e', m_i, \text{Redefinición}(v_e, v_e') \wedge \text{Pertenece}(v_e, m_i) \\ \wedge \neg \text{Pertenece}(v_e', m_i) \wedge \text{Pertenece}(v_e', \text{aplicar}(\phi, m_i)) \\ \Leftrightarrow \\ \text{redefinir}(v_e, v_e') \in \phi \wedge \neg \text{Pertenece}(v_e, \text{aplicar}(\phi, m_i)) \end{aligned} \quad (7.)$$

Las expresiones planteadas anteriormente expresan formalmente como determinar el conjunto de v_e que forman parte de una versión de modelo. Cada versión de modelo puede construirse a partir de la versión anterior y la secuencia de operaciones aplicadas. Es posible reconstruir a una versión m_{i+1} aplicando todas las secuencias de operaciones realizadas a partir de la versión inicial m_0 . Esto surge de:

$$m_{i+1} = \text{aplicar}(\phi_i, m_i); m_i = \text{aplicar}(\phi_{i-1}, m_{i-1}); \dots; m_1 = \text{aplicar}(\phi_0, \emptyset)$$

$$m_{i+1} = \text{aplicar}(\phi_i, \text{aplicar}(\phi_{i-1}, \text{aplicar}(\dots \text{aplicar}(\phi_0, \emptyset) \dots)))$$

$$m_{i+1} = \text{aplicar}(\phi_0 \dots \phi_{i-1} \phi_i, \emptyset); \text{ donde } \phi_i \phi_j \text{ constituye la concatenación de las secuencias } \phi_i \text{ y } \phi_j.$$

Las expresiones presentadas permitirían reconstruir una versión de modelo dada y navegar a través de las operaciones que le dieron origen. A continuación se presenta un modelo de objetos que estructura la información de los modelos generados y soportan las operaciones antes definidas.

5. Estructura de Objetos que Soporta el Modelo

El esquema general que se maneja en esta propuesta considera que cada versión de modelo se genera a partir de vistas de un repositorio en el que se mantienen todos los objetos que han sido generados y modificados debido a la evolución del modelo. La estrategia surge de la siguiente observación: durante la construcción de un modelo, los distintos conceptos que se incorporan al mismo pueden, en versiones sucesivas: (i) modificarse (e incluso eliminarse) o (ii) asociarse con distintas entidades. Por ejemplo: el cambio de nombre de una clase, una alteración en la signatura de un método, son ejemplos de modificaciones que sufre una entidad. Por otro lado, en distintas versiones de un modelo una clase puede variar sus atributos o métodos, en un diagrama de interacción pueden variar los objetos y mensajes intervinientes, etc.

Considerando la observación anterior se organiza el manejo de versiones de modelos en un esquema de dos capas. Por un lado, un *Repositorio* (R) en el que se representan las entidades que surgen durante el proceso de generación de los sucesivos modelos, como así también las relaciones que se establecen entre las mismas. Las entidades que componen al repositorio se denominarán *Entidad Versionable* (e_v) y el conjunto de todas las e_v , *Entidades Versionables* (E_v). En la segunda capa, *Versiones* (V), están definidas las distintas versiones que se han generado para cada entidad existente en el repositorio, los objetos definidos en este nivel son del tipo v_e , presentadas en el punto 4. Operaciones Básicas.

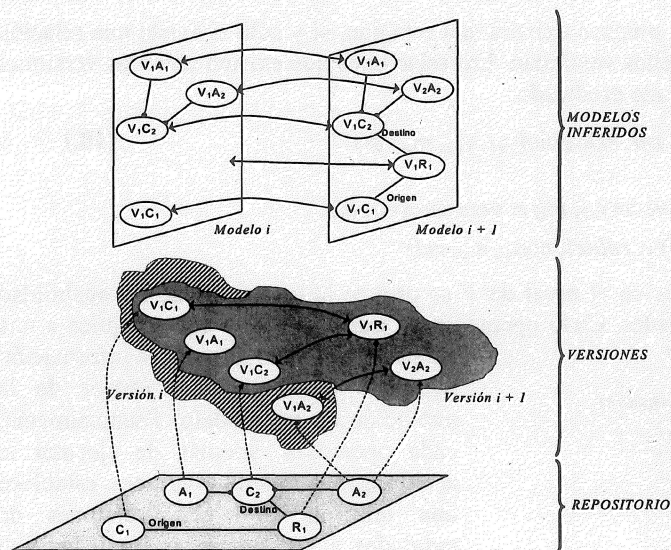


Figura 1: Esquema de Versionamiento

alguna versión del modelo. Las asociaciones que mantiene cada uno de estos objetos permiten determinar el conjunto de entidades con las que se asocia en distintas versiones de un modelo.

En correspondencia con cada clase del repositorio, se define en el nivel de V una clase que especifica las características versionables de dicha entidad. En este sentido, cada v_e define aquellas propiedades de una entidad que pueden ser modificadas en las distintas versiones de modelos. Un conjunto de v_e constituye una versión de modelo. Es así como toda versión de modelo se puede inferir a partir de los niveles de conocimiento antes mencionados, como se esquematiza en la Figura 1. De esta manera un concepto formará parte de un modelo si una versión de dicho concepto pertenece al modelo, como lo establece la expresión 4. Por ello algunas entidades del repositorio pueden no formar parte de un modelo (en el caso que ninguna versión de la misma pertenezca al modelo). Además, por razones de consistencia, es necesario restringir a uno la cantidad máxima de versiones de una misma entidad que puedan pertenecer a un modelo.

Luego, la relación que existe entre los dos niveles y un modelo se puede expresar a través de la siguiente definición: cada e_v (instancia de alguna clase componente del repositorio) tiene asociado en un instante del tiempo una o más v_e , pero dado un modelo particular a lo sumo una de dichas versiones pertenece al mismo. Definida la función $w: E_v \rightarrow \mathcal{P}V_e$; que asocia a cada entidad el conjunto de versiones definidas, es decir: $w(e) = \{v_e / v_e \text{ es una versión de } e\}$, se debe cumplir la expresión 8.

$$\begin{aligned} & \forall e_v \in E_v \mid w(e_v) \mid \geq 1 \\ & \forall e_v \in E_v, m_i \in M \mid w(e_v) \cap m_i \mid \leq 1 \end{aligned} \quad (8.)$$

Establecidas las entidades que conforman un modelo se deben determinar las relaciones existentes entre ellas. Primero se debe notar que en la propuesta, las versiones de entidad pertenecientes a un modelo no se asocian **directamente** con otras versiones del mismo modelo, en cambio, como se mencionara anteriormente, las

entidades del repositorio mantienen la información de todas las entidades con las que se las ha relacionado durante las distintas versiones de un modelo. En consecuencia, la relación que existe entre dos v_e se debe inferir a partir de la relación establecida entre las entidades (pertenecientes al *Repositorio*) que las mismas versionan. Se debe notar que, dada la definición del metamodelo, para una versión particular siempre existe una entidad en el repositorio de la cual la misma es versión, es decir:

$$\forall v_e \in V_e \exists e_v \in E_v: v_e \in w(e_v) \quad (9.)$$

Luego, dada dos v_e de un modelo, entre las mismas existirá una relación, si y sólo si existe una relación entre las entidades del repositorio que cada una de ellas versionan. Las relaciones que existen entre las versiones de entidades se pueden establecer a través del siguiente predicado:

$$\begin{aligned} \forall v_{e1}, v_{e2} \in V_e, m_i \in M, rel: & \text{relación}(v_{e1}, v_{e2}, m_i, rel) \\ \Leftrightarrow & \\ \text{pertenece}(v_{e1}, m_i) \wedge \text{pertenece}(v_{e2}, m_i) \wedge & \text{versiona}(v_{e1}, e_{v1}) \\ \wedge \text{versiona}(v_{e2}, e_{v2}) \wedge \text{relación}(e_{v1}, e_{v2}, rel) & \end{aligned} \quad (10.)$$

A través de la definición de asociaciones explícitas en el nivel de V se provee la capacidad de navegabilidad entre las distintas versiones de modelos y de entidades. Cada operación de transformación que se aplica a una

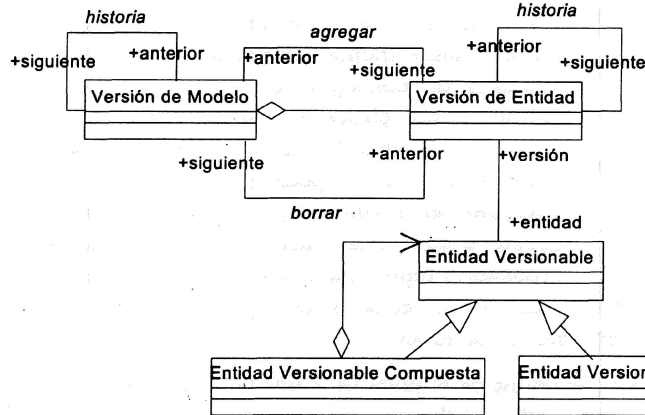


Figura 2: Modelo de objetos

versión de modelo, incorpora información necesaria para mantener rastros de la evolución de un modelo. Esencialmente, cada operación, además de ejecutar la acción que la misma involucra, establece una relación entre las versiones de entidades a las que se aplica y las que surgen como resultado de su ejecución. En el modelo de objeto de la **Figura 2** se pueden observar estas asociaciones entre versiones de entidades, denominada historia, y entre Versión de Modelo y Versión de Entidad, denominadas agregar y borrar.

6. Operaciones Básicas en el Modelo de Objetos

Las operaciones definidas inicialmente para el manejo de versiones (*Agregar*, *Borrar*, *Refinar*, *Simplificar*, *Redefinir*) se implementan de la siguiente manera.

Agregar: al aplicar la operación agregar a una versión de modelo, se crea una instancia en el repositorio de alguna subclase de *Entidad Versionable* y una instancia de la subclase de *Versión de Entidad* correspondiente. Es responsabilidad de las entidades versionables instanciar las relaciones que mantiene la nueva entidad con las existentes en el repositorio. Se instancia la relación que vincula a una entidad versionada con su nueva versión, y esta última se agrega a la versión de modelo que se está modificando. Por último se deben establecer los vínculos explícitos para mantener la evolución del modelo.

```

agregar: nombre deTipo: tipoEntidad
|unaVersionDeEntidad|
unaVersionDeEntidad := self agregarElemento: nombre deTipo: tipoEntidad.
unaVersionDeEntidad historia: self anterior.

```

```

agregarElemento: nombre deTipo: tipoEntidad
|unaEntidadVersionable unaVersionDeEntidad|
unaEntidadVersionable := tipoEntidad entidadVersionable new: nombre.
unaVersionDeEntidad := tipoEntidad versionEntidad new: nombre.
unaEntidadVersionable versionadaPor: unaVersionDeEntidad.
self agregarComponente: unaVersionDeEntidad.
^ unaVersionDeEntidad

```

Borrar: La acción de borrar una entidad de un modelo consiste en eliminar la versión de entidad correspondiente del modelo, y en forma similar a *Agregar* establece los vínculos para mantener la evolución del modelo.

```

borrar: unaVersionDeEntidad
self borrarElemento: unaVersionDeEntidad.
self historia: unaVersionDeEntidad.

borrarElemento: unaVersionDeEntidad
self borrarComponente: unaVersionDeEntidad.

```

Simplificar: El efecto de realizar una simplificación consiste en eliminar del modelo a las entidades que están siendo simplificadas, y agregar la entidad que se genera como resultado de la simplificación. Por último se deben establecer los vínculos explícitos para mantener la evolución del modelo.

```

simplificar: listaVersionesEntidad por: nombre deTipo: tipoEntidad
|nuevaVersionDeEntidad|
listaVersionesEntidad do:[x| self borrar: x].
nuevaVersionDeEntidad := self agregarElemento: nombre deTipo: nombre.
listaVersionesEntidad do:[x| nuevaVersionDeEntidad historia: x].

```

Refinar: Un refinamiento, se puede definir de manera análoga a la simplificación.

```

refinar: unaVersionDeEntidad por: listaVersiones
|nuevasVersionesDeEntidades|
self borrarElemento: unaVersionDeEntidad.
nuevasVersionesDeEntidades := listaVersiones do: [x| self agregarElemento: x].
nuevasVersionesDeEntidades do:[x| x historia: unaVersionDeEntidad].

```

Redefinir: esta operación reemplaza una versión en el modelo sin generar una nueva entidad en el repositorio. Al igual que las restantes operaciones deben mantener los vínculos que provean la capacidad de navegabilidad.

```

redefinir: unaVersionDeEntidad por: nombre
|nuevaVersionDeEntidad|
self borrarElemento: unaVersionDeEntidad.
nuevaVersionDeEntidad:= unaVersionDeEntidad tipo versionEntidad new: nombre.
self agregarComponente: nuevaVersionDeEntidad.
unaVersionDeEntidad versionaA versionadaPor: nuevaVersionDeEntidad.
unaVersionDeEntidad historia: nuevaVersionDeEntidad.

```

7. Aplicación de la Estructura Propuesta

Con el objeto de ejemplificar un modelo capaz de soportar el versionamiento de modelos muy elementales de Análisis y Diseño Orientado a Objetos, se especializa el esquema presentado en la **Figura 2**.

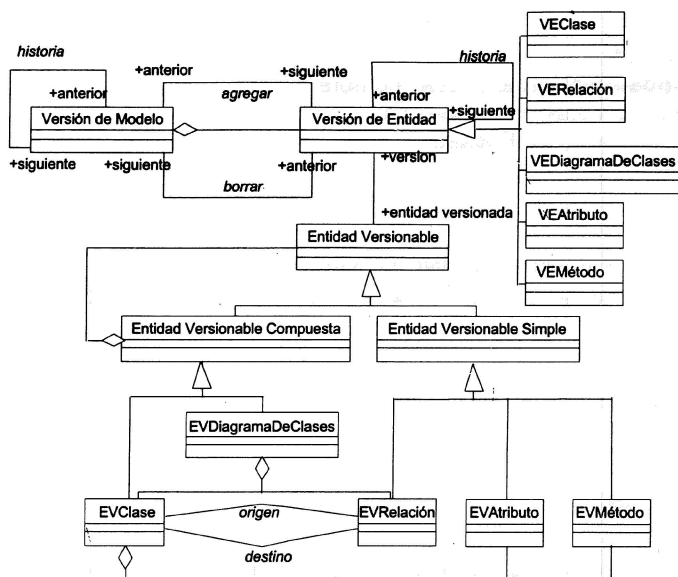


Figura 3

Las entidades de la notación que puedan estar compuestas por otras entidades se definen como una especialización de *Entidad Versionable Compuesta*; el resto de las entidades especializan la clase *Entidad Versionable Simple*. En correspondencia con cada refinamiento de *Entidad Versionable* se especializa *Versión de Entidad*. En la **Figura 3** se presenta una especialización (parcial) para el caso particular de una metodología de análisis y diseño orientada a objetos, en la que los modelos están compuestos sólo por Diagramas de Clases, Clases, Relaciones, Atributos y Métodos.

Al iniciar la construcción de un nuevo modelo se genera una instancia de *Versión de Modelo* (Versión de Modelo 1) que se corresponde con el modelo vacío inicial. Si se define luego el diagrama de clases presentado en la **Figura 4**, el *Repositorio* estará formado por las instancias de las entidades versionables que se muestran en la

Figura 5. En la misma, se muestran las asociaciones que mantiene cada uno de estos objetos. Las mismas se han establecido a medida que se ha incorporado las entidades versionables.

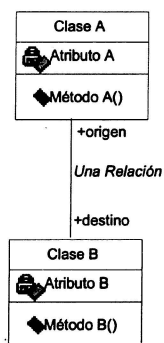


Figura 4

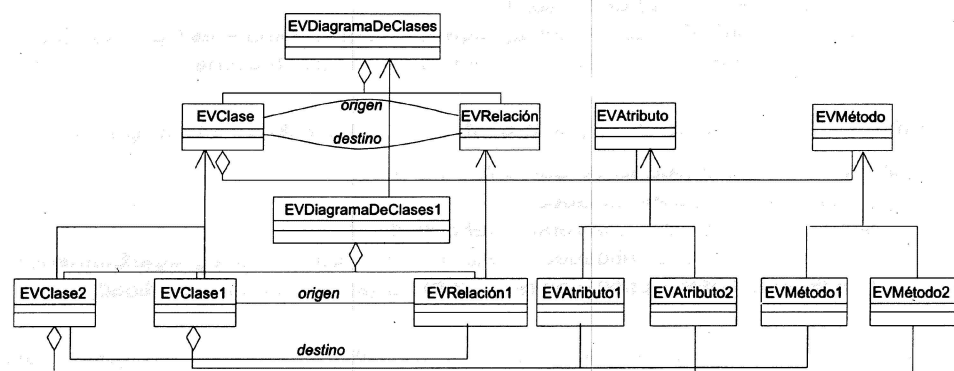


Figura 5

8. Especificación de la Metodología

El metamodelo de Versionamiento planteado, provee capacidades que permiten el manejo de versiones para la construcción de modelos. Para que el mismo pueda ser empleado, se deberá especializar de acuerdo a las entidades que considere la metodología en particular que se desee emplear (por ejemplo: clases, objetos, atributos, métodos, mensajes, relaciones, etc.). Asimismo, las relaciones que se definen entre las entidades del repositorio estarán dadas por la especialización del metamodelo. Luego, si se cuenta con una descripción de la metodología a emplear, se puede obtener una especialización de este metamodelo de versionamiento. En este punto se describe un *Metamodelo de Metodologías*, cuyas instancias serán utilizadas para especificar a las metodologías deseadas, y obtener a partir de éstas las especializaciones del metamodelo de Versionamiento correspondientes que permitan administrar versiones de los modelos generadas con dichas metodologías.

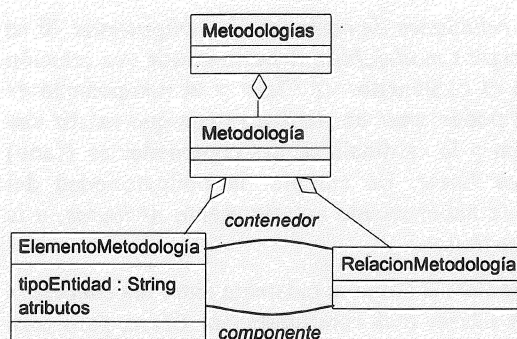


Figura 6: Diagrama de clases del Metamodelo de Metodologías

En el diseño del *Metamodelo de Metodologías* se contempla la posibilidad de trabajar con un conjunto de metodologías, donde cada metodología se describe a través de los diversos conceptos modelables que utiliza y las relaciones existentes entre ellos. Dicha estructura se presenta en el Diagrama de Clases de la **Figura 6**.

Para los fines de versionamiento, un aspecto importante es la granularidad de la descripción de la metodología. Es decir, la especificación de los elementos de la misma y los atributos de estos elementos. Por ejemplo, en una descripción de una metodología Orientada a Objetos se podría considerar a los parámetros formales de un método como elementos de la metodología, en tanto que en otra descripción podrían ser considerados como atributos del concepto método, en lugar de

ser elementos básicos de la metodología. En este sentido, asociada a toda descripción de una metodología existirá un grado de descomposición de cada uno de sus elementos. Este vendrá dado por el conjunto de *conceptos modelables* que se especifiquen. A los fines de versionamiento, un *concepto modelable* es una entidad de la metodología a la que se le puede aplicar, a lo largo del proceso del desarrollo del software, algunas de las operaciones agregar, borrar, simplificar o refinar. Por otra parte, un concepto considerado como atributo de un concepto modelable dado se puede redefinir a lo largo del proceso de desarrollo. Por ejemplo, al describir una metodología se puede especificar a las relaciones y a cada uno de sus propiedades como conceptos modelables distintos, es decir utilizar los conceptos modelables *Relación*, *cardinalidadOrigen*, *cardinalidadDestino*, *rolOrigen* y *rolDestino*. Se deberán describir además, las formas en que estos conceptos se asocian, creando para cada asociación una instancia de la clase *RelaciónMetodología*. Otra definición posible de la misma metodología podría utilizar sólo el concepto *Relación*, cuyos atributos sean *cardinalidadOrigen*, *cardinalidadDestino*, *rolOrigen* y *rolDestino*.

Una instancia del *metamodelo de Metodologías* (que define una metodología en particular), contiene una instancia de *ElementoMetodología* por cada *concepto modelable* de la metodología (cada una de las cuales permitirá definir una Entidad Versionable del Repositorio). El valor del atributo *atributos* de cada una de estas instancias, define las características versionables del concepto. Por otra parte, para describir cada una de las relaciones existentes entre los *conceptos modelables* se genera una instancia de *RelaciónMetodología*, donde se especifican los siguientes atributos:

- ❑ *Contenedor de la relación*, elemento de la metodología que contiene al componente a través de la relación.
- ❑ *Obligatoriedad del Contenedor*, especifica la necesidad de la existencia del Contenedor de la relación en el caso de crear al Componente, especificando TRUE en el caso que sea necesario y FALSE en el otro caso.
- ❑ *Cardinalidad del Contenedor*, especifica cuantos contenedores puede tener un componente.
- ❑ *Componente de la relación*, elemento de la metodología contenido en componente a través de la relación especificada.
- ❑ *Obligatoriedad del Componente*, especifica la necesidad de la existencia del Componente de la relación en el caso de crear al Contenedor, especificando TRUE en el caso que sea necesario y FALSE en el otro caso.
- ❑ *Cardinalidad del Componente*, especifica cuantos componentes puede tener un contenedor.

La forma propuesta para definir las relaciones entre los conceptos modelables establece un esquema básico para verificar la consistencia de los modelos que se definen. De esta manera se pueden definir

restricciones para la construcción de modelos, como por ejemplo *relaciones de composición obligatorias*. Si se trabajara en una metodología en la cual *Clase* y *Atributo* son *conceptos modelables*, debería existir una relación entre *Clase* y *Atributo*, que denominamos *ClaseAtributo*, donde el contenedor es *Clase* y el componente es *Atributo*. La obligatoriedad del contenedor es TRUE porque para poder crear un *Atributo* tiene que existir una *Clase* a la que se asocia a través de una instancia de esta relación y la cardinalidad del contenedor es 1(unos) porque al *Atributo* solamente se lo puede asociar a una única *Clase*. En cambio, la obligatoriedad del Componente es FALSE porque para crear una *Clase* no se necesita haber creado anteriormente *Atributos*, y la cardinalidad es *n* debido a que una *Clase* puede contener muchos *Atributos*.

Además permite definir *relaciones de composición no obligatorias* como la existente entre los *conceptos modelables* *Diagrama de Clases* y *Clase*, donde un *Diagrama de Clases* está compuesto por *Clase*, pero para crear una *Clase* no es necesario incluirla dentro de un *Diagrama de Clases*, esta relación se define con *Diagrama de Clases* como contenedor, *Clase* como componente, FALSE en ambos atributos de obligatoriedad y *n* los atributos de cardinalidad (una clase pueda estar en *n* Diagramas de Clases y un Diagrama de Clases puede contener *n* Clases).

Por último, es posible definir *otros tipos de restricciones* a la forma en que se asocian los *conceptos modelables*. Por ejemplo, si se considera que una relación puede establecerse entre dos clases (una clase *origen* y otra *destino*), y que una relación sólo puede tener una clase origen y una destino, pero que una clase puede ser origen y destino de distintas relaciones, esto se puede expresar de la siguiente manera: definir los *conceptos modelables* *Clase* y *Relación* instanciando a la clase *ElementoMetodología*. Asimismo se deberán definir las instancias de *RelaciónMetodología* que representen que una clase es origen de una relación y que una clase es destino de una relación, que llamaremos *RelaciónOrigen* y *RelaciónDestino* respectivamente. Se define además que la clase *Relación* es el contenedor y *Clase* es el componente. Como para crear una *Clase* no necesito de la existencia de una *Relación* la *obligatoriedadContenedor* en ambas relaciones es FALSE, en cambio, para poder crear una *Relación* entre dos *Clases* se requiere de la existencia de dos *Clases*, la *obligatoriedadComponente* en ambas relaciones es TRUE. La *cardinalidadContenedor* es *n* debido a que una *Clase* puede estar involucrada en diversas relaciones, en cambio la *cardinalidadComponente* es 1 en ambas relaciones ya que una única *Clase* es el origen de una *Relación* y una única *Clase* es el destino de una *Relación*.

Si ejemplificamos una metodología que utiliza Diagramas de Clase, los cuales están compuestos por Clases y Relaciones, donde las Relaciones tienen como origen y destino a Clases y las Clases están compuestas por Atributos, se obtiene la especialización del Metamodelo de Metodologías que se muestra en la Figura 7.

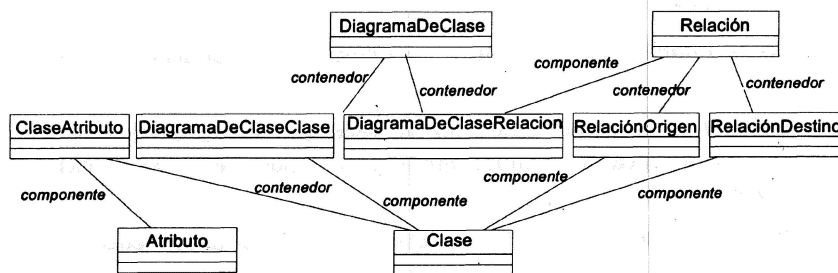


Figura7: Una Especialización del Metamodelo de Metodologías.

9. Algoritmo de Especialización de Metamodelo de Versionamiento.

El Metamodelo de Versionamiento se especializa a partir de la especificación de una metodología, creándose las clases en el nivel Repositorio y en nivel Versión.

Por cada *concepto modelable* de la metodología definido como instancia de *ElementoMetodología* se especializa el nivel de *Versión de Entidad* generando una subclase de *VersionDeEntidad*. En la Figura 8 se

muestran las instancias correspondientes a los *conceptos modelables Clase y Relación*. El algoritmo toma estas clases y especializa la clase *Versión de Entidad* (**Figura 8 – (1)**) generando las subclases *VEClase* y *VERelación*. La estructura de estas nuevas clases consistirá en los atributos definidos en el atributo *atributos* de la instancia de *ElementoMetodologia*. En el ejemplo, los atributos de la clase *VERelación* están determinado por el valor del atributo *atributos* de la clase *Relación*, como por ejemplo, *RolOrigen*. Asimismo, se especializa el Repositorio, generando las subclases correspondientes de *EntidadVersionable*. En el ejemplo, (**Figura 8 – (2)**) se generan las subclases *EVClase* y *EVRelación* a partir de las instancias *Clase* y *Relación*. Al generar estas clases en el repositorio, se definen además las relaciones existentes entre las mismas. Por ejemplo (**Figura 8 – (3)**), a partir de la instancia *RelaciónDestino* del *metamodelo de metodologías*, se establece en el repositorio la relación *destino* entre las clases *EVClase* y *EVRelación*.

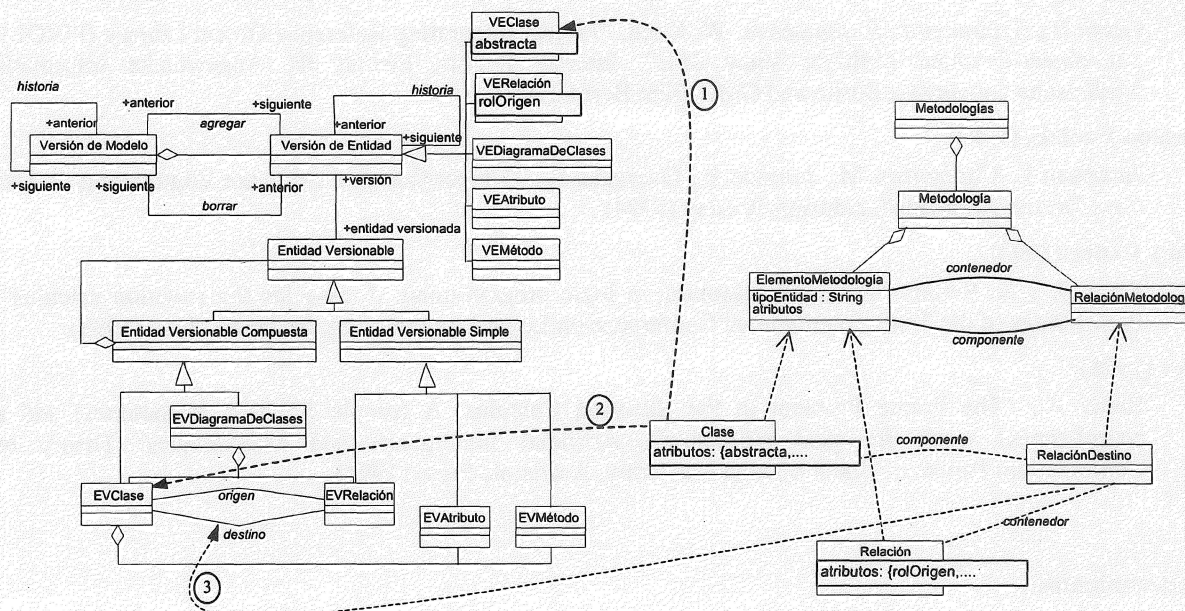


Figura 8: Una Especialización del Metamodelo de Versionamiento.

10. Conclusiones

Se ha presentado un metamodelo básico para implementar a partir del mismo el manejo de versiones en una herramienta que soporte el proceso de desarrollo de software empleando la especificación formal del cálculo situacional y el modelo de objetos. Este desarrollo se encuentra en el marco de un proyecto cuyo objetivo es el desarrollo de un ambiente que soporte integralmente el proceso de desarrollo de software. En el mismo se detectó como una necesidad básica contar con un esquema que permita manejar consistentemente y navegar a través de los modelos generados en el proceso. Se pretende a través de ésta propuesta documentar la evolución de un modelo de forma tal de poder rastrear cómo entidades identificadas inicialmente evolucionan a lo largo del proceso de desarrollo de software. Análogamente, tomando un concepto del modelo final, sería posible explicar su generación mediante la recuperación de los axiomas aplicados en su génesis.

A partir de las operaciones básicas presentadas se pretende modelar las actividades más complejas que se ejecutan en las distintas etapas del proceso de desarrollo de software, de forma tal de registrar su empleo en la generación de las distintas versiones.

Bibliografía**Booch (1994)**

Booch G., "Object-oriented analysis and design with applications", Benjamin-cummings (1994).

Booch, Rumbaugh y Jacobson (1999)

Booch G., Rumbaugh J., Jacobson I., "The unified modelling language user guide", Addison-Wesley (1999).

Corriveau (1996)

Corriveau, J., "Traceability Process for Large OO Projects", Computer, 29, 9, pág. 63 (1996).

Groth y otros (1994)

Groth B., S. Hermann, S. Jähnichen, W. Koch, "Project Integrating Reference Object Library (PIROL): An object-Oriented Multiple View SEE", Internal Report, Institut für Angewandte Informatik Technische Universität Berlin and GMD-First Berlin (1994).

Jacobson y colab. (1994)

Jacobson I., Christerson M., Jonsson P., Övergaard G., "Object-Oriented Software Engineering. A Use Case Driven Approach", Addison-Wesley (1994).

Pinto y Reiter (1993)

Pinto, J. y R. Reiter, "Temporal reasoning in logic programming: A case for the situation calculus", Proceedings of the Tenth International Conference on Logic Programming, Budapest, June (1993).

Reiter (1991)

Reiter R., "The Frame Problem in the Situation Calculus: A Simple Solution (sometimes) and a completeness result for goal regression", Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy, Academic Press (1991).

Agradecimientos

Este trabajo ha sido financiado en forma conjunta con los proyectos "Ambiente para el Desarrollo de Modelos de Empresas de Producción", homologado por la Universidad Tecnológica Nacional, el PIA de CONICET N° 6346 y el proyecto PICT 14-00000-01143 de la Agencia Nacional de Promoción Científica y Tecnológica, SCyT de la República Argentina. Se agradece el apoyo brindado por estas instituciones.