Grafos versionables: un soporte para el manejo de la evolucion del esquema de una BDOO y para el manejo de versiones de un hipertexto.

Trabajo de grado

Realizado por: Maria Eugenia Berizonce
Maria Vanesa Cabral
Maria Alejandra Ripa Alberdi

Director: Silvia Gordillo

Septiembre, 1992

PREFACIO

Presentamos en el informe los conceptos básicos sobre Bases de Datos Orientadas a Objetos (BDOO), Bases de Datos Orientadas a Objetos para CAD, Hipertextos y Versiones.

Describimos el modelo general, desarrollado para soportar el manejo de versiones y motivado por los requerimientos de las áreas antes mencionadas.

Presentamos un prototipo que captura las características fundamentales del modelo.

Mostramos la generalidad del modelo y planteamos posibles extensiones al prototipo al utilizar grafos versionables como un soporte para :

* la evolución del esquema de una BDOO

* la evolución del esquema de una BDOO extendido para soportar objetos compuestos

* la evolución en el tiempo de un hipertexto

Finalmente, extendemos el prototipo y lo especializamos para soportar versiones de hipertextos.

El punto 1 contiene temas introductorios como BDOO, BDOO para CAD, Hipertextos, Versiones y el uso de versiones en cada una de las áreas mencionadas.

El punto 2 describe el modelo de versionamiento desarrollado, su fundamentación y características. Así mismo, presenta un prototipo que implementa el modelo.

El punto 3 muestra la utilización de los grafos versionables como soporte para la evolución de una BDOO, para la evolución de un esquema de una BDOO para objetos compuestos y para la evolución en el tiempo de un hipertexto.

El punto 4 contiene una descripción del desarrollo del prototipo extendido para soportar, en particular, versiones de hipertextos.

Para el lector no familiarizado con *grafos* y *programación orientada a objetos*, recomendamos leer el apéndice A previo a la lectura del informe.

AGRADECIMIENTOS

A Gustavo Rossi por su ayuda con el tema de hipertextos y la corrección final del informe.

A Ricardo Guido Lavalle por sus aportes sobre visualización del grafo.

A Gabriela Lopez Uhalde por su colaboración en el momento preciso.

A Adolfo Nagy por la búsqueda de papers del tema.

INDICE

Capitulo 4: Una aplicacion: extension del prototipo

INTRODUCCION

Las aplicaciones que utilizan una BDOO como soporte tecnológico, se caracterizan por estar sujetas a cambios constantes. En una aplicación muchas veces es necesario mantener varias versiones de las estructuras de los tipos que se han ido manejando y que han sufrido algún tipo de evolución. Que un tipo evolucione no significa que los usuarios no quieran seguir manipulando versiones de un tipo, ni que los objetos instanciados en versiones anteriores desaparezcan del sistema.

Si especializamos las BDOO para soportar CAD, encontramos que los objetos de diseño atraviesan distintas fases en el proceso de diseño que dan lugar a múltiples versiones del objeto.

Por otro lado, los sistemas de hipertextos en los últimos años han puesto su énfasis en la información compartida y la colaboración. Entonces, es necesario que los hipertextos sean capaces de mantener configuraciones que varíen en el tiempo. Además, en muchos casos la organización de los datos de los hipertextos varía entre usuarios o aplicaciones.

Presentamos un modelo general que es un manejador de versiones de grafos de contenido arbitrario. El propósito del modelo es reflejar la evolución de una jerarquía de tipos de una BDOO, de una jerarquía de tipos de complejidad arbitraria de una BDOO para CAD y de un hipertexto.

Mostramos, además, que el modelo es implementable con la realización de un prototipo. Efectuamos un análisis de las áreas de aplicación presentadas previamente como una extensión al prototipo general expuesto.

En particular, se detalla mediante un ejemplo, la especificación del prototipo para soportar versiones de hipertexto.

Parte

1

Conceptos basicos

Capitulo 1: Conceptos basicos

1.1 - Bases de Datos Orientadas a Objetos

1.2 - Bases de Datos Orientadas a Objetos para Cad

1.3 - Hipertexto

1.4 - Versiones

1.4.1 - Conceptos generales

1.4.2 - Versionamiento del esquema de una BDOO

1.4.3 - Versionamiento del esquema de una BDOO que soporta objetos compuestos

1.4.4 - Versionamiento de un sistema de hipertexto

1- CONCEPTOS BASICOS

En esta sección describiremos, en principio, las nociones fundamentales de las siguientes áreas de aplicación: Bases de Datos Orientadas a Objetos, Bases de Datos Orientadas a Objetos para CAD e Hipertextos. Luego detallaremos los conceptos básicos y los distintos enfoques para la implementación de versiones en general y, en particular, el uso de versiones en cada una de las áreas mencionadas.

1.1- BASES DE DATOS ORIENTADAS A OBJETOS

En un Sistema de información, el proceso de especificación de software puede ser visto como el proceso de construir un modelo sobre alguna porción del mundo real.

Este modelo debería reflejar, lo más naturalmente posible, la conceptualización del usuario del universo de discurso.

En la década del 60, aparecían los primeros *Manejadores de Bases de Datos* (Jerárquico y en Red), que se basaban en la noción de registros lógicos con registros padres y registros hijos con distintas restricciones según el modelo.

Es en la década del 70 que aparece el *Modelo Relacional*, que despierta gran interés entre los diseñadores de sistemas, principalmente de sistemas "comerciales".

La simplicidad del modelo (sólo existen tablas conformadas por tuplas), su sólida sustentación matemática, la provisión de un poderoso lenguaje de consultas, y la completa independencia de los datos, hicieron que este modelo se convierta en "el modelo", y aún hoy es el elegido para resolver esta clase de aplicaciones.

Aunque estos modelos han permitido capturar el mundo a modelizar más naturalmente (principalmente el Relacional), así como facilitar el proceso de diseño, el abaratamiento de los costos de los sistemas de cómputo y el aumento de su capacidad y velocidad, han traído aparejado la posibilidad del desarrollo de nuevas aplicaciones en áreas no convencionales para las Bases de Datos, como Inteligencia

Artificial, Automatizacion de Oficinas, Procesamiento de Imágenes, Multimedia, etc. Además de los requerimientos usuales en estas áreas (persistencia, recuperación de fallas, back-up) se requiere una buena capacidad de modelización de datos, soporte para inferencias, nuevos tipos de datos como imágenes gráficas, voz, texto, múltiples versiones de objetos, etc. [BAN88]

LIMITACIONES DEL MODELO RELACIONAL

Detallaremos a continuación algunos de los problemas que presenta el Modelo Relacional:

- Imposibilidad de representar *objetos compuestos*.

Las aplicaciones no convencionales introducidas utilizan frecuentemente objetos complejos, mientras que en los modelos tradicionales los items de datos están uniformemente estructurados y tienden a ser orientados a registros, donde el contenido de un item es un dato atómico. Cuando se debe representar una entidad compleja se la debe dividir en varias entidades. Sin embargo, hay ciertos tipos de datos en los que ésto no sólo sería una tarea tediosa, sino casi imposible; por ejemplo: un módulo de software, una imagen de video, etc.

Vemos entonces, que el mecanismo de abstracción del modelo sacrifica "expresividad".

- Imposibilidad de representar el comportamiento de los datos.

Para brindarle semántica a la Base de Datos es necesario reflejar toda la interpretación de los datos con los que se está trabajando. Para esto, se debe representar a los datos no solo en función de su estructura sino también en función de su comportamiento. En el Modelo Relacional las operaciones sobre los datos están dadas solamente por los programas de aplicación.

- Tipos de datos no extensibles.

Solo se tienen los tipos primitivos y las aplicaciones se deben restringir a usar esos tipos. No se le permite al usuario definir sus

propios tipos de acuerdo con las características propias de cada aplicación.

- Incompatibilidad entre el lenguaje de programación y el de la Base de Datos.

En los modelos convencionales se accede a la información a través de un lenguaje de consultas que puede resultar incompleto, ya que no tiene el poder computacional de los lenguajes de programación. Otra forma de acceso es mediante un lenguaje "host", agregando primitivas para acceder a la Base de Datos. Sin embargo, las Bases de Datos y los lenguajes de programación no están totalmente integrados, sino que existen distintas incompatibilidades:

* *Incompatibilidad estructural:* los tipos de datos de los lenguajes de programación y los de las Bases de Datos son distintos; las Bases de Datos usan conjuntos de tuplas y este tipo no existe en los lenguajes de programación.

* *Incompatibilidad operacional:* los operadores relacionales son orientados a conjuntos y los lenguajes de programación son orientados a registros.

* *Incompatibilidad de paradigmas:* los operadores relacionales son declarativos (el usuario especifica qué datos quiere sin especificar cómo acceder a ellos). Los de los lenguajes de programación son imperativos (el usuario especifica qué dato quiere y cómo recuperarlo).

EL MODELO ORIENTADO A OBJETOS EN BASES DE DATOS

Los requerimientos antes mencionados hacen necesario el estudio de nuevos modelos conceptuales que permitan modelizar las situaciones y capturar la interpretación de los datos en términos de lo que ellos representan en el mundo de la aplicación.

En este contexto aparece el *Modelo Orientado a Objetos* como uno de los más poderosos para utilizar en Bases de Datos, ya que satisface plenamente las necesidades mencionadas.

El principio fundamental es la correspondencia uno a uno entre las entidades de la situación a modelizar y los objetos del modelo. Así cada entidad del mundo real se representa por exactamente un objeto en el modelo, independientemente de su complejidad, sin necesidad de hacer descomposiciones artificiales en conceptos simples.

Un objeto es una máquina abstracta, que define un protocolo a través del cual el objeto puede interactuar con otros. Puede tener un estado que está encapsulado. El protocolo está definido por un conjunto de mensajes, que se responden mediante métodos, que son piezas de código que implementan las acciones deseadas. Un mensaje se envía a un objeto para que éste realice una acción; sólo por medio de estos mensajes se puede acceder al objeto.

Los *objetos* permiten modelizar tanto la estructura de los datos como su comportamiento, en una misma entidad. Cuando almacenamos un dato en la Base de Datos, almacenamos al mismo tiempo los datos y los programas de aplicación. Así, hay un modelo simple para datos y operaciones.

Un *tipo* define un "molde" para un conjunto de objetos con las mismas características. Consta de dos partes, la interfase del objeto y la implementación. Solamente la interfase es visible al usuario del tipo y consiste de un conjunto de operaciones (incluyendo los tipos de los parámetros). La implementación consta de una parte de definición de datos (donde se describe la estructura interna que tendrán los objetos de ese tipo) y una definición de operaciones (programas que implementan cada una de las operaciones de la interfase).

Una de las ventajas del mecanismo de definición es la de tener un sistema en el cual los tipos son extensibles, lo que brinda nuevas funcionalidades a los datos. Los tipos pueden agruparse en *jerarquías* que establecen una relación IS_A entre los tipos.

La *herencia* permite la modificación incremental de las definiciones de los tipos, ya que provee la capacidad de que nuevos tipos compartan las definiciones previas y reusen código.

La noción de *Clases* en Bases de Datos es distinta de la noción de Clases en los lenguajes de programación orientada a objetos y, en particular distinta de la noción de tipo. Una clase es una colección de objetos del mismo tipo (la extensión del tipo). Las clases se utilizan para crear y manipular objetos.

La noción de *identidad* es otra característica fundamental para el modelo. Un objeto tiene una existencia independiente de sus valores; esto es, un objeto no solo puede referenciarse a través de los valores de su estado, sino fundamentalmente, a través de su identidad, por lo tanto la identidad de un objeto debe ser inmutable; ésta debe persistir mientras persista el objeto.

Los Sistemas de Bases de Datos tradicionales están orientados a valores; esto significa que la referencia a un objeto está dada por el valor de algún o algunos de sus atributos (llamados claves); lo cual hace difícil expresar cierto tipo de información. La noción de identidad permite distinguir entre objetos idénticos (cuando son el mismo objeto) y objetos iguales (cuando los valores de sus estados son los mismos); esta diferencia hace posible modelizar situaciones como:

Tipo Silla	Silla1	Silla2
Atributos		
Material: Integer	madera	madera
Color : String	marrón	marrón
Tipo : String	inglesa	inglesa

Los objetos Silla1 y Silla2 tienen valores iguales, sin embargo no dejan de ser dos objetos distintos. La existencia de identidad nos permite modelizar a estos objetos como dos objetos distintos, ya que tendrán distinta identidad, y asociarle a cada uno (por ejemplo en alguna situación) su respectivo dueño.

La identidad nos permite también establecer la noción de estructura compartida. Podemos distinguir entre referirnos a dos cosas que son iguales o referirnos a dos cosas que de hecho son la misma.

Aunque las Bases de Datos Orientadas a Objetos (BDOO) se basan en la noción de identidad es importante tener en cuenta que la manera de relacionar objetos puede basarse o no en esta característica. Podemos resumir entonces, que una de las principales diferencias entre una BDOO y una Relacional es que mientras que en la última un esquema se define a partir de construcciones primitivas como tablas (cuyos atributos son atómicos), en la primera un esquema está dado por la definición de un conjunto de tipos cuyos atributos pueden ser complejos, que pueden relacionarse a través de la relación IS_A, etc.

OTRAS CARACTERISTICAS

Además de las nociones básicas del modelo de objetos, existen características inherentes a las Bases de Datos que deben ser tenidas en cuenta.

Una de ellas es cómo hacer que los objetos persistan, esto es, cómo se determina que un objeto es persistente o transitorio.

El manejo de persistencia es una de las diferencias más importantes entre el modelo Orientado a Objetos y el Relacional. El Relacional provee un espacio de nombres persistentes muy limitado; las únicas variables persistentes son las relaciones. Una sentencia de creación de una relación, define un tipo persistente, una variable de ese tipo y el conjunto de sus instancias; esto hace que sea imposible declarar varias variables de un tipo determinado, como así también, definir una variable de cualquier tipo persistente. El modelo Orientado a Objetos permite la definición de un espacio de nombres persistentes en donde se pueden declarar variables de cualquier tipo, así como cualquier número de instancias de un tipo.

Otra característica fundamental en Bases de Datos es la definición de un lenguaje de consultas y la forma en que se manejarán las relaciones.

Un lenguaje de consultas es un lenguaje de alto nivel, que permite un acceso simple y declarativo a los datos. Las consultas en el modelo Orientado a Objetos se realizan sobre las clases (como colecciones de objetos). La posibilidad de hacer consultas tanto

sobre cualquier colección de objetos, como sobre objetos de cualquier tipo, le otorgan al modelo un poder que en los modelos anteriores no existe. Esto se debe a la uniformidad de los mismos ya que en el Relacional las consultas se realizan sobre una tabla, y en el Jerárquico y en el de Red sobre registros.

Otra de las facilidades ganadas es la de la posibilidad de navegar sobre los datos, posibilidad que en el modelo Relacional no existe y resultan muy difíciles y, a veces imposibles, cierto tipo de consultas, fundamentalmente aquellas de naturaleza recursiva [MEN91].

Las relaciones entre objetos es una característica muy importante en Bases de Datos. Una relación es una correspondencia entre objetos. En el modelo existe la posibilidad de definir relaciones binarias, relacionando objetos entre sí. Algunos modelos definen automáticamente la relación inversa. Por ejemplo, se podría establecer una relación entre Directores y Empleados que determina que un director dirige a un conjunto de empleados definiendo en el tipo Director un atributo *dirige_a* que los relacione.

```
Tipo Director
     Atributos
     Nom-Ape: String
     DNI: String
     Direcc: String
     Dirige-a: set of Empleado

Tipo Empleado
     Atributos
     Nom-Ape: String
     DNI: String
     NroEmp: Integer
     Direcc: String
```

Esto producirá la existencia de una serie de referencias de cada uno de los directores a cada uno de los empleados que dirige en la Base de Datos.

El problema de establecer una relación de esta manera, es que la misma es unidireccional, en el sentido que la relación va desde los directores a los empleados. Si se quisiera encontrar el director de

un empleado particular sería muy dificultoso. La solución es, o explicitar la relación inversa, agregando un atributo en el tipo Empleado, por ejemplo *dirigido-por* cuyo dominio sería el tipo Director (algunos manejadores establecen la relación inversa automáticamente) o tratar a la relación como una entidad más, y, por lo tanto, tendrá un tipo asociado. En el ejemplo anterior, eliminarse el atributo *dirige-a* del tipo Director y declarar:

Tipo Dirección
    Atributos
    Dirige: Director
    Dirigido: Empleado

Estas relaciones, no sólo son bidireccionales, sino que también permiten representar relaciones n-arias y relaciones con atributos y comportamiento propio.

Una de las ventajas del modelo de BDOO, es que las relaciones se establecen a partir de referencias entre objetos y no a través de pares de valores, lo que nos permite hacer navegación sobre el grafo de instancias que se genera.

Las relaciones, no solo pueden manejarse como referencias a otros objetos, sino también como objetos que relacionan entidades. Con esta facilidad podemos representar una relación entre varios objetos (n_aria).[BAN88]

1.2- BASES DE DATOS ORIENTADAS A OBJETOS PARA CAD

Existe un consenso cada vez mayor en que el enfoque orientado a objetos puede simplificar las aplicaciones que utilizan datos de diseño asistido por computadora. Diversos grupos de trabajo están utilizando estos conceptos para estructurar una Base de Datos para CAD. El SMBDOO asociado deberá crear, almacenar, controlar y manipular los *objetos de diseño* almacenados en la Base de Datos.

"Diseñar es el acto de crear un objeto artificial (objeto de diseño), que no existía previamente en el mundo real, a partir de algún concepto abstracto extraído de cosas existentes".[BOR89]

Los objetos de diseño tienen dos características esenciales que surgen de la naturaleza exploratoria e iterativa propia del proceso de diseño: [AHM91]

- Están jerárquicamente formados por objetos componentes ensamblados.

- Atraviesan distintas fases en el proceso de diseño que dan lugar a múltiples versiones del objeto.

OBJETOS DE DISEÑO

Cada variedad de CAD (diseños de circuitos electrónicos, de software, de arquitectura) trata con *objetos de diseño*, que generalmente son objetos de gran complejidad: *objetos compuestos* consistentes de múltiples subcomponentes, las cuales pueden a su vez ser objetos complejos. Entidades recursivamente compuestas de entidades más simples son frecuentes en muchos otros dominios de aplicación, tales como: sistemas de información de oficinas, manejo de documentos y gráficos [ZDO90]. En el modelo orientado a objetos, estos objetos compuestos son representados por exactamente un objeto (modelización de una entidad del mundo real) independientemente de su complejidad.

Muchas de las aplicaciones antes mencionadas requieren de la posibilidad de definir y manipular objetos complejos (objetos anidados) como una unidad lógica por razones de integridad semántica, eficiencia en el almacenamiento y recuperación. [KIM87]

Un objeto tiene una serie de atributos; el valor de un atributo es un objeto que pertenece a alguna clase, la cual puede ser primitiva o no-primitiva. [KIM89b]

Una clase se llama *primitiva* cuando no tiene atributos (por ejemplo, String, Integer).

Un objeto con un atributo cuyo valor es un objeto que pertenece a una clase no-primitiva es un *objeto anidado*.

Un *objeto complejo* es un conjunto de objetos anidados vinculados a través de la relación *es-parte-de* (que representa el concepto de

que un objeto es parte de otro objeto, no necesariamente de manera exclusiva) y/o objetos simples, que forman una unidad lógica.

Un *objeto agregado* es un objeto complejo, un *objeto compuesto* es un objeto agregado con la particularidad de que puede ser creado sólo en forma top-down. [ZDO90]

Un objeto O' *referencia a* otro objeto O, si O' contiene el identificador único del objeto O.

La jerarquía de tipos a la cual los objetos componentes de un objeto compuesto pertenecen, se llama *jerarquía de partes* o *jerarquía de agregación*. Un tipo no raíz, de la jeraquía de partes, se llama *tipo componente*. Cada tipo que no es una hoja tiene una o más variables de instancia cuyos dominios son los tipos componentes. Llamamos a estas variables de instancia, *variables de instancia compuestas*. Un objeto componente de un objeto compuesto referencia a una instancia de su tipo componente a través de una variable de instancia compuesta. Llamamos a esta referencia, *referencia compuesta* entre un objeto y su objeto componente (o entre un tipo y su tipo componente).

Frente a la necesidad de modelizar objetos de diseño (usualmente de gran complejidad) propia de las aplicaciones de CAD, los diseñadores de SMBDOO deben focalizar su atención en mejorar, entre otros, el manejo de objetos complejos.

Existen diversos criterios para el tratamiento de objetos compuestos, sin embargo debemos destacar la falta de un mecanismo para soportar agregación de objetos.

Diversos manejadores de BDOO tales como ORION, GemStone e IRIS tratan con objetos compuestos.

En [KIM89b] se plantea la necesidad de extender el manejo de objetos compuestos de ORION a objetos agregados e identifica tres aspectos que deben mejorarse:

a) El modelo de ORION restringe un objeto compuesto a una jerarquía de objetos componentes exclusivos, esto es, un objeto componente es sólo parte de un objeto compuesto. Este es el modelo apropiado para reflejar una jerarquía de partes física,

Pero, no es apropiada para representar una jerarquía de partes lógica (por ejemplo, un capítulo idéntico puede ser parte de dos libros diferentes).

b) El modelo de ORION fuerza a la creación top-down de un objeto compuesto, es decir, antes de crear un objeto componente, su objeto padre ya debe existir. Esto no permite la creación de objetos en forma bottom-up, ensamblando objetos existentes.

c) El modelo de ORION requiere que la existencia de un objeto componente dependa de la existencia de un objeto padre, es decir, si un objeto deja de existir, todos sus objetos componentes son borrados. Esto es útil en muchos casos, pero impide la reutilización de objetos en un ambiente de diseño complejo. Sería de gran utilidad que los ambientes de CAD provean librerías de componentes y que un diseño pueda hacer uso ilimitado de ellas.

La extensión del modelo de objetos compuestos de ORION para soportar agregación de objetos, propuesta por [KIM89b], plantea la necesidad de ampliar la semántica asociada a las referencias.

El autor distingue dos tipos de referencias: las *referencias débiles* (referencias standard en un sistema orientado a objetos y que no tienen ninguna semántica especial asociada) y las *referencias compuestas.*

Las *referencias compuestas* son referencias débiles con el agregado adicional de la relación *es-parte-de*. Una referencia compuesta de O' a O significa que O es parte de O'. Una referencia compuesta, además se define en términos de si un objeto es parte de un solo objeto o de más de uno.

Una *referencia compuesta exclusiva* de O' a O significa que O es parte sólo de O', mientras que una *referencia compuesta compartida* de O' a O significa que O es parte de O' y posiblemente también de otros objetos.

El autor refina la semántica de referencias compuestas, tanto exclusivas como compartidas, basándose en si la existencia de un objeto depende de la existencia de su padre o no.

Una *referencia compuesta dependiente* de O´ a O significa que la existencia de O depende de la existencia de O´, mientras que una *referencia compuesta independiente* de O´ a O no agrega ninguna semántica adicional. El borrado de un objeto trae aparejado el borrado en forma recursiva de todos los objetos referenciados por él a través de referencias compuestas dependientes (exclusivas o compartidas).

Esto lleva a la siguiente clasificación de las referencias:

1- Referencia débil

2- Referencia compuesta exclusiva dependiente

3- Referencia compuesta exclusiva independiente

4- Referencia compuesta compartida dependiente

5- Referencia compuesta compartida independiente

Los objetos relacionados a través de referencias compuestas forman una jerarquía de partes. La *raíz de un objeto compuesto* es un objeto especial. Bajo el modelo que contempla la creación de objetos compuestos sólo en forma top-down (como en ORION), la raíz de un objeto nunca cambia. Es deseable que la raíz corriente de un objeto compuesto pueda cambiar, es decir, que pueda ser el destino de una referencia que parte de otro objeto.

Se identifican dos tipos de jerarquías: física y lógica.

En una *jerarquía de partes física* todas las referencias compuestas son exclusivas; esta jerarquía modeliza un objeto compuesto. Mientras que en una *jerarquía de partes lógica*, puede haber también referencias compuestas compartidas; esta jerarquía modeliza un objeto agregado. [KIM89b]

Proporcionaremos a continuación dos ejemplos para observar los iferentes tipos de referencias. El primer ejemplo es una jerarquía e partes física, mientras que el segundo es una jerarquía de partes ógica.

Ejemplo 1: Consideremos la jerarquía de composición de un ehículo. Supongamos que un vehículo tiene una patente, un color, una arrocería, un tren delantero y ruedas. Cada parte del vehículo puede er usada por un solo vehículo en un momento dado, pero sin embargo as partes de un vehículo pueden ser reutilizadas por otros vehículos i éste se desmantela.

Para modelizar la jerarquía deseada, debemos indicar en la lefinición de la clase Vehículo que todos los atributos compuestos de n vehículo (carrocería, tren delantero, ruedas) son referencias xclusivas independientes.

Por ser referencias compuestas exclusivas, el conjunto de omponentes de un vehículo puede ser usado sólo por él. Y por ser eferencias compuestas independientes, dichas componentes pueden ser sadas por otro vehículo si éste se desarma, pudiendo existir sin, ecesariamente, ser parte de algún vehículo.

Ejemplo 2: Consideremos un documento electrónico. Supongamos que l mismo consiste de un título, autores y un número de secciones. ada sección está compuesta de párrafos. Un documento puede compartir árrafos o secciones enteras con otros documentos. Se pueden agregar notaciones a los documentos, pero éstas no se comparten entre locumentos. Los documentos pueden contener imágenes que son extraídas le archivos.

Para modelizar documentos electrónicos con las características leseadas, debemos indicar en la definición de la clase Documento que:

- el atributo compuesto "contenido", definido como un conjunto de Secciones, es una referencia compuesta compartida dependiente. De esta forma queda establecido que otros documentos pueden compartir cualquier sección del documento y

que una sección existe sólo si pertenece al menos a un documento.

- el atributo compuesto "anotaciones", definido como un conjunto de Párrafos, es una referencia compuesta exclusiva dependiente. De esta forma queda establecido que una anotación es usada en un solo documento y que no tiene sentido su existencia fuera de un documento.

- el atributo compuesto "figuras", definido como un conjunto de Imágenes, es una referencia compuesta compartida independiente. De esta forma queda establecido que las imágenes son objetos compartidos con los archivos y que su existencia no depende del documento que las contiene.

- el atributo "titulo" es una referencia débil, ya que está definido como un objeto String.

- el atributo "autores" es una referencia débil, ya que está definido como un conjunto de String.

En la definición de la clase Sección debemos indicar que:

- el atributo compuesto "contenido", definido como un conjunto de Párrafos es una referencia compuesta compartida dependiente. De esta forma queda establecido que un párrafo puede ser compartido por distintas secciones (posiblemente de documentos diferentes) y que la existencia de un párrafo depende de que exista al menos una sección que lo contenga dentro de al menos un documento.

1.3- HIPERTEXTO

Un *hipertexto* es una red donde los nodos representan conceptos y los links relaciones entre ellos. [CON87]

En [PRO90] se distingue entre *hipertexto* (como un cuerpo estructurado de información) y *Sistemas de Hipertexto*.

Un *hipertexto* es una red de nodos de información conectados a través de links que establecen una relación entre los nodos.

Un *Sistema de Hipertexto* es una configuración de hardware y software que presenta un hipertexto a los usuarios y les permite manejar y acceder a la información que éste contiene.

La capacidad de los hipertextos de tener links permite una organización no lineal de texto proveyendo un soporte para manejo de información para una amplia variedad de esfuerzos de documentación.

Podemos agrupar los conceptos asociados a los sistemas de hipertexto en tres categorías básicas: entidades, propiedades y funciones.

Las entidades son los objetos que un sistema de hipertexto manipula. Las entidades que conforman un hipertexto son las componentes, los documentos, las ayudas para la navegación y las entidades para la visualización. Las componentes son los nodos, links y las entidades compuestas.

Los *nodos* en un sistema pueden ser descriptos desde la perspectiva de su contenido, tipo y estructura. [PAR90]

Contenido de los nodos sugiere generalizar la noción de nodo a "cualquier item de información sobre el cual el sistema puede razonar".

Primitivamente, los sistemas de hipertexto representaban sólo información en forma de texto. Luego, diversas aplicaciones fueron incorporando información en forma de gráficos, animación, video, audio. Esto condujo a los *Sistemas de hipermedia.*

Por muchas razones, se deben definir localizaciones dentro de los nodos, como destino o como origen para un link. Los mecanismos para esta definición son altamente dependientes del contenido del nodo. Por ejemplo:

* Como el texto es unidimensional, la localización en un nodo de texto se define en base a caracteres.

* Como los graficos son bidimensionales, la localización en un nodo gráfico se define en base a pixels.

* La definición de localización en nodos con animación y/o video es en base a pixels, con el agregado de dimensión de tiempo.

* La localización en un nodo de audio se define temporalmente.

Además de contenidos diferentes, los nodos pueden tener también tipos diferentes. Esto es importante en el contexto de links tipados. Junto con el tipo de links, el tipado de nodos permite la definición de gramáticas, que facilitan al usuario la navegación y la recuperación automática de la información.

Las medidas de localización definidas anteriormente muchas veces son ineficientes. Por ejemplo, uno puede definir palabras o sentencias en un nodo de texto, botones en un nodo gráfico, frases musicales en un nodo de audio, ocultando los caracteres, pixels, intervalos de tiempo asociados, como detalles de implementación. Entonces, los links pueden tener como origen o como destino estos objetos de orden superior.

Por otro lado, los *links* pueden ser direccionales (uno de sus extremos está diferenciado del otro de alguna manera) o no. Cognitivamente, los links direccionales son más valiosos para la navegación.

Se pueden definir distintos *tipos de links* (relacionándolos con nodos tipados) que pueden enriquecer las capacidades del hipertexto.

El *destino de un link* puede ser un nodo como una entidad atómica, o alguna entidad contenida en el nodo. En el caso de un nodo estructurado, esta entidad será algún elemento de la estructura. En el caso de un nodo no estructurado, esta entidad será un punto o una región definida por una medida de localización apropiada al contenido del nodo.

Los nodos y links poseen atributos tales como la propiedad de visualizarse.

Las entidades compuestas son entidades que están compuestas por nodos y links individuales. Pueden ser definidas retórica o topológicamente.

Un ejemplo de entidad compuesta definida topológicamente son los "caminos". Una red de nodos y links es apropiada para realizar un recorrido arbitrario, pero para muchas aplicaciones es útil definir una trayectoria por defecto, pudiendo continuar con ella luego de cualquier disgreción. Topológicamente, los caminos imponen una topología lineal sobre una red más complicada.

Las entidades compuestas definidas retóricamente son constelaciones específicas de nodos (generalmente tipados) y links que forman una unidad lógica para la manipulación y navegación.

Como ayudas para la navegación se incluyen índices, mapas, tablas de contenido, vistas globales, browser.

El *browser* es una componente importante de los sistemas de hipertexto. Como un hipertexto crece, es muy fácil para un usuario perderse o desorientarse. Un browser muestra parte o todo el hipertexto como un grafo, proveyendo una medida importante de indicaciones contextuales para ofrecerle al usuario qué nodos está viendo y cómo ellos están relacionados entre sí y sus vecinos en el grafo.

La red puede ser representada por un grafo acíclico dirigido, una jerarquía, etc.

Una característica de los sistemas de hipertexto es que poseen un alto uso e ventanas que tienen una correspondencia uno-a-uno con los nodos. Las ventanas pueden ser reposicionadas, cerradas, redefinido su tamaño, etc. La posición, tamaño de una ventana (y tal vez también su color y forma) son indicaciones para recordar los contenidos de la ventana.

Como propiedades del sistema se pueden destacar la existencia de un modelo formal, concurrencia, sincronización, sensibilidad al contexto, diferentes modos de operación (browse, autor).

Las funciones que pueden aplicarse a un sistema de hipertexto pueden ser clasificadas en modificaciones de la información, que incluye modificar información del sistema y desplazar información

entre sistemas; navegación, que incluye búsqueda y query y browsing. También se destaca la posibilidad de realizar *versionamiento*.

UTILIZACION DE LOS SISTEMAS DE HIPERTEXTOS

En los últimos años, para hacer que la información sea más accesible y útil, los conceptos de hipertexto se han empleado en una gran variedad de aplicaciones: [GLU90]

* Libros de referencia, enciclopedias, diccionarios dinámicos
* Ayudas al usuario
* Documentación en línea
* Legislación, auditorías
* Educación
* Ingeniería y CAD
* Manejo de proyectos profesionales
* Resolución grupal de problemas
* Entretenimiento interactivo
* Guías (turísticas, museos)

Las aplicaciones de hipertexto pueden ser agrupadas en dos categorías:

- Las aplicaciones donde las nodos y los links pueden ser editados. Por ejemplo, CAD y manejo de proyectos profesionales.

- Las aplicaciones "READ-ONLY" donde ni las nodos ni los links pueden ser editados. Por ejemplo, enciclopedias, diccionarios, etc.

Los sistemas de hipertexto están en constante evolución porque son de interés tanto para el área académica como para el área empresarial.

1.4- VERSIONES

1.4.1- CONCEPTOS GENERALES

Cuántas veces oímos la frase: ¿Escuchaste la nueva *versión* del tema "Love of my life" de Freddy Mercury interpretada por Extreme?

Todos conocemos, intuitivamente, el significado de la palabra versión. Observamos que la canción a la que se hace referencia es la misma independientemente del cantante que la interprete. La canción no pierde su esencia, aunque el intérprete le adicione su estilo propio.

Similarmente, en el desarrollo de programas, podemos plantear implementaciones alternativas para un módulo (por ejemplo, un módulo de ordenación). Está claro que las distintas implementaciones del módulo de ordenación son versiones del mismo, ya que, aunque de distinta manera, todas implementan lo mismo. De igual forma, al invocar desde el programa a una u otra de las versiones de la rutina, se tienen distintas versiones del programa.

Nuestro propósito es representar esta clase de situaciones reales y soportarlas con una herramienta computacional.

En [FIS89] se define a las *versiones* como fotografías de un objeto en diferentes estados, que comparten algunas características comunes y son modelizados por distintos objetos.

En las distintas aplicaciones la intención que se le da a versionar o mantener un conjunto de versiones, puede variar. Existen, básicamente, tres intenciones:

a) mantener la evolución de una entidad conceptual durante el proceso de diseño hasta alcanzar el objetivo deseado, descartando luego los estados intermedios y conservando sólo el

b) mantener la evolución de una entidad conceptual con la finalidad de reflejar las distintas configuraciones que variaron en el tiempo, para, por ejemplo, poder hacer un análisis comparativo;

c) mantener diferentes vistas de una entidad conceptual en un ambiente de múltiples usuarios o aplicaciones.

Daremos algunos ejemplos para clarificar la clasificación presentada.

El proceso de diseño es un caso evidente en el que se aplican, en forma reiterada, una serie de transformaciones sobre la idea primitiva, con la finalidad de llegar al objetivo deseado.

En el diseño de un logo, por ejemplo, se pueden plantear inicialmente uno o más diseños alternativos que cumplan con la especificación. Luego, cada uno de ellos puede ser refinado, obteniendo así múltiples versiones del(los) objeto(s) de diseño. Tras comparar las distintas versiones del logo, el usuario seleccionará una, perdiéndose, entonces, el interés por las restantes, que se desechan.

Las entidades sufren mutaciones a través del tiempo, debido a la ocurrencia de ciertos eventos o acontecimientos trascendentales que marcan fases en su evolución.

Podría interpretarse cada eslabón de la cadena evolutiva de la especie humana, como una versión. A través de sucesivas transformaciones sería posible reflejarla, desde sus orígenes.

El ejemplo de las versiones de un tema musical, que dimos para introducir los conceptos de versión, se adecúa, también, a la segunda intencionalidad de versionar.

Para ejemplificar el punto c) presentaremos dos situaciones:

Cuando un grupo de gente trabaja en forma conjunta en un proyecto, usualmente, desea crear pequeñas áreas privadas donde cada uno pueda volcar sus ideas sin interferir con las de los demás

Aquello que se está desarrollando en cada área de trabajo representa una versión del proyecto.

Por otro lado, consideremos que un grupo de usuarios cuenta con datos históricos de este siglo. Alguno de ellos puede estar interesado en los acontecimientos de la 1º guerra mundial, mientras que otro, en los compositores del período. Es decir, que cada uno de ellos tendrá su propia vista de los mismos datos, y por consiguiente manejará una versión diferente.

OBJETOS VERSIONABLES

En ciertos SMBDOO el usuario puede especificar que un objeto es versionable en la declaración del tipo del objeto o en el momento en que el objeto es creado. Por ejemplo, en IRIS los objetos son, por defecto, del tipo 'Unversioned' (sin versiones), predefinido en el sistema de tipos. En el caso en que un objeto requiera versionamiento, el tipo del cual es instancia es convertido al tipo 'Version', también predefinido en el sistema de tipos. Por otro lado, en ORION, por una cuestión de performance, la aplicación debe indicar si la clase es versionable o no.

Si un objeto es versionable, una versión del objeto puede ser creada:

a) cuando el objeto es 'check_out' para actualización, es decir, cuando se trae una porción de la Base de Datos del disco a un área de trabajo;

b) cuando el objeto es actualizado por primera vez; o

c) cuando se termina de trabajar con el objeto y éste es escrito en la Base de Datos. [CAT]

Podría ser interesante para el usuario tener la **historia de un objeto**, es decir, examinar los valores de los atributos de versiones anteriores de un objeto. Nuevos objetos versionables son creados copiando el objeto y enganchando el objeto viejo atrás de la nueva versión. El enlace de la versión debe ser doblemente enganchado, o debe ofrecer algún otro mecanismo para obtener, desde cualquier versión, toda la historia del objeto.

CONFIGURACIONES

En [SIM91] se distingue el concepto de versión del de configuración.

Los objetos son organizados durante su **evolución** para registrar los cambios en el tiempo. La motivación para ello es referencial (debe ser posible inspeccionar viejas versiones de un objeto) y de recuperación (debe permitirse recuperar versiones anteriores para usarlas).

Las versiones son más poderosas, cuando un conjunto de versiones consistentes de los objetos en una Base de Datos, pueden ser agrupadas como **configuraciones.** Se puede pensar una configuración como una versión de una Base de Datos entera, aunque alcance con referenciar solamente a la porción de la Base de Datos que ha cambiado. La configuración es, en sí misma, un objeto que representa una versión de la Base de Datos. [CAT]

Las configuraciones representan vistas sobre una Base de Datos de objetos versionables. Están sujetas a cambios sustanciales hasta que son congeladas para ser archivadas. Las configuraciones son objetos de software usados para organizar la evolución de sistemas complejos.

Los mecanismos para soportar configuraciones tienen uso exploratorio (para estudiar mecanismos de agregación en objetos base) y uso práctico (para controlar cambios en proyectos de software).

FORMAS DE VERSIONAMIENTO

Los elementos de un conjunto de versiones están relacionados entre ellos por la relación de orden llamada *´sucesor-de´*.

Los conjuntos de versiones pueden ser **lineales** o **alternativos.**

Una versión se puede derivar a partir de una o más versiones predecesoras y puede tener múltiples versiones sucesoras que son llamadas **alternativas.** Las alternativas corresponden al caso en el cual dos o más versiones que compiten en un diseño coexisten como las versiones más recientes. Generalmente, se modelizan con una estructura de DAG. (Gráfico 1)

En el caso en que todas las versiones se generan a partir de una única versión predecesora, la representación de DAG se particulariza a un árbol, siendo el conjunto de versiones, un orden parcial. (Gráfico 2)

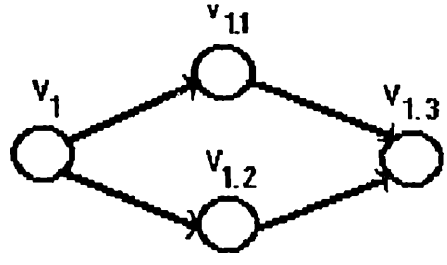

Gráfico 1: Versionamiento alternativo cuya representación es un DAG.

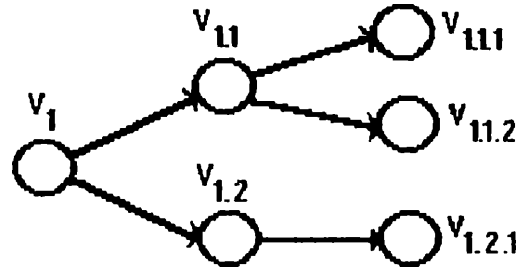

Gráfico 2: Versionamiento alternativo cuya representación es un árbol.

Si se agrega una nueva versión v_2 a un objeto x, que ya tenía una versión v_1, entonces v_1 y v_2 se dice que son alternativas. Insertar una alternativa en un conjunto de versiones tiene el efecto de comenzar una nueva rama en el orden parcial. [ZDO]

Cada alternativa puede, a su vez, tener una historia de versión propia, independiente. [SKA87]

Si restringimos el caso anterior a que las versiones tengan un único sucesor, tenemos, entonces, un conjunto **lineal** de versiones. Un conjunto **lineal** de versiones es un orden total y una versión puede ser agregada solamente al final. Todas las versiones previas son de lectura únicamente. Las versiones lineales corresponden al caso en que se representa evolución lineal. (Gráfico 3)

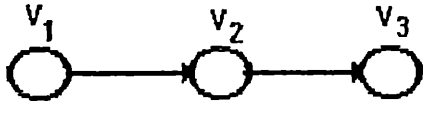

Gráfico 3: Versionamiento lineal cuya representación es una lista.

Ejemplificaremos a continuación las dos formas de versionamiento antes mencionadas.

La historia universal, la de todos los tiempos y pueblos del mundo, se divide en varios períodos. La Historia Antigua, la Edad Media, la Edad Moderna y la Edad Contemporánea. Este es un claro ejemplo de **evolución lineal**.

Por otro lado, un arquitecto, diseñando una casa, considera varias alternativas para realizar el proyecto, y trabaja sobre cada una de ellas hasta llegar a aquella que mejor se adecúe a la especificación y necesidades planteadas por el cliente. En la etapa de diseño, el arquitecto analiza la posibilidad de tener un living-comedor amplio o, alternativamente, un living y un comedor independientes. Se entiende, que se trata del mismo proyecto (para la misma casa), pero que existen dos alternativas que difieren, una de otra, en la construcción del living-comedor. Dichas alternativas, en su momento, serán evaluadas y comparadas sus funcionalidades por parte del cliente.

Tras introducir, intuitiva y formalmente, los conceptos de versionamiento lineal y alternativo, pasaremos a describir diversos enfoques de implementación encontrados en la bibliografía.

Para soportar **versionamiento alternativo** es necesario distinguir estados de las versiones, de forma tal de restringir las operaciones que se pueden aplicar sobre ellas. Por ejemplo, si se permite modificar una versión cualquiera, y, en particular, se derivaron otras versiones a partir de ella, aparecería el problema de tener que propagar los cambios a todas las versiones derivadas.

Para prevenir éste y otro tipo de inconvenientes, en [AHM91] se distinguen tres estados de un objeto versionable: validado, estable y transitorio.

Las <u>versiones validadas</u> tienen las siguientes características:

* No se permite realizar ninguna modificación sobre versiones validadas

* Todas las referencias compuestas en una versión validada deben estar ligadas a objetos no versionables u otras versiones

* Se pueden derivar nuevas versiones a partir de ellas

Las versiones estables tienen las siguientes características:

* Los atributos significativos (aquellos cuyos valores pueden cambiar en distintas versiones del objeto) no se pueden modificar.

* Los atributos no significativos pueden modificarse

* Se pueden derivar nuevas versiones de versiones estables

Las versiones transitorias tienen las siguientes características:

* Todas las versiones recientemente creadas y derivadas comienzan en estado transitorio

* Todos los atributos no invariantes (es decir, aquellos cuyos valores pueden cambiar en distintas versiones) de una versión transitoria pueden ser modificados

* No se pueden derivar nuevas versiones a partir de ellas

Por otro lado, para soportar **versionamiento alternativo** ORION [KIM89a] implementa dos tipos de versiones sobre la base de la clase de operaciones que se pueden aplicar sobre ellas.

Una versión transitoria tiene las siguientes propiedades:

* Puede ser actualizada por el usuario que la creó

* Puede ser borrada por el usuario que la creó

* Una nueva versión transitoria puede ser derivada de una versión transitoria existente. Esta última es, luego, ´promovida´ a una versión de trabajo.

Una versión de trabajo tiene las siguientes propiedades:

* Es considerada estable y no puede ser actualizada

* Puede ser borrada por su dueño

* Una versión transitoria puede ser derivada de una versión de trabajo

* Una versión transitoria puede ser 'promovida' a una versión de trabajo. La promoción puede ser explícita (especificada por el usuario) o implícita (determinada por el sistema)

Conceptualmente, en el **versionamiento lineal** se pueden distinguir dos estados de las versiones: activas y no-activas.

Una nueva versión se obtiene aplicando un conjunto de modificaciones sobre la versión activa (la última que se generó). La versión modificada es promovida al estado no-activo y el estado de la nueva versión es activo.

Por lo tanto, cuando el versionamiento es lineal, en el conjunto de versiones de un objeto solamente hay **una** versión en el estado activo (la última).

Una versión activa tiene las siguientes características:

* Se puede modificar

* Se puede derivar una nueva versión a partir de ella

* Al crearse una versión comienza en estado activo

* Se puede consultar

Las versiones no-activas tienen las siguientes características:

* Se pueden consultar

* No se pueden modificar

* No se pueden derivar nuevas versiones a partir de ella

1.4.2- VERSIONAMIENTO DEL ESQUEMA DE UNA BDOO

Las aplicaciones tales como Multimedia, CASE, etc. se caracterizan por estar sujetas a cambios constantes. Los cambios en el esquema de las Bases de Datos en forma arbitraria constituyen un proceso muy dificultoso.

Generalizando, es natural pensar que durante el diseño, las visiones del mundo a modelizar puedan cambiar. En una aplicación, muchas veces es necesario mantener varias *versiones* de las estructuras de los tipos que se han ido manejando y que han sufrido algún tipo de evolución. Estas estructuras definen distintas **versiones de la jerarquía de tipos** representada por un DAG.

En BDOO, dos tipos de **cambios en el esquema de la Base de Datos** son necesarios:

- Cambios en la definición de un tipo

- Cambios en la estructura del reticulado

Cambios en la definición de un tipo incluye agregado y borrado de variables de instancia y métodos. De esta forma se generan **versiones de tipos.**

Cambios en la estructura de un reticulado incluye creación y borrado de un tipo, y alteración de la relación *es-un* entre tipos. De esta forma se generan **versiones del reticulado de tipos.**

Todas las versiones de una definición de tipo se agrupan en un conjunto de versiones del tipo. La relación que existe entre los elementos de dicho conjunto es una relación de orden llamada *'sucesor-de'*, pudiendo ser, un orden parcial o total. De igual forma, existe la misma relación entre las versiones de un reticulado de tipos.

Cambios en el reticulado de tipos puede ser caracterizado por:

(1) Cambios en el contenido de un nodo

(2) Cambios en una arista

(3) Cambios a un nodo

(1) Cambios en el contenido de un nodo

(1.1) Cambios a una variable de instancia.

(1.1.1) Agregar una variable de instancia v a un tipo C podría causar un conflicto de nombres, cuya resolución dependerá de la Resolución de Conflictos propuesta por el manejador. Esta adición deberá notificarse o propagarse a todos los subtipos de C.

(1.1.2) Eliminar una variable de instancia v de un tipo C. Esta eliminación deberá notificarse o propagarse a todos los subtipos de C. Si C o cualquiera de sus subtipos tiene otros supertipos que tienen variables de instancia del mismo nombre que v, se hereda una de estas variables de instancia. El mecanismo de Resolución de Conflictos determinará qué nueva variable es heredada.

(1.1.3) Cambio del nombre de una variable de instancia v del tipo C. Este cambio podría originar un conflicto de nombres que deberá ser resuelto por el sistema.

(1.1.4) Cambio al dominio de una variable de instancia v de un tipo C. El dominio de una variable de instancia existente debería poder cambiarse, solamente, para ser generalizada; es decir, un supertipo del tipo de la variable de instancia v. El cambio debe ser notificado o propagado a los subtipos.

(1.1.5) Cambios en la herencia de una variable de instancia v de un tipo C. Este cambio repercute en los subtipos de C por lo que debe ser notificado o propagado a los mismos.

(1.2) Cambios en un método

(1.2.1) Adicionar un nuevo método a un tipo.

Similar a (1.1.1).

(1.2.2) Eliminar un método existente de un tipo.

Similar a (1.1.2).

(1.2.3) Cambiar el nombre de un método de un tipo.

Similar a (1.1.3).

(1.2.4) Cambiar el código de un método de un tipo. Similar a (1.14). Se podrán generalizar los métodos utilizados dentro del código del método que se está modificando.

(1.2.5) Cambiar la herencia (ancestro) de un método, (hereda otro método con el mismo nombre)

Cabe destacar que si cambia el protocolo de un tipo (métodos públicos), este cambio debe ser notificado a todos los tipos pertenecientes a la jerarquía.

(2) Cambios a una arista

(2.1) Ubicar a un tipo S como supertipo de un tipo C. La adición de una nueva arista desde S hacia C no debe introducir un ciclo en el reticulado de tipos. C y sus subtipos heredarán las variables de instancia y métodos desde S. En el caso en que se produjera un conflicto de nombres, el mecanismo de Resolución de Conflictos del sistema determinará la herencia.

(2.2) Remover un tipo S de la lista de supertipos de C. El borrado de una arista desde S hacia C no debe causar que el reticulado quede desconectado. El sistema deberá detectar esta situación, y actuar en consecuencia. Este borrado afecta a los subtipos (dejan de heredar las variables de instancia y métodos de S), entonces, esta acción debe propagarse o notificarse a los mismos.

(3) Cambios en un nodo

(3.1) Adicionar un nuevo tipo C. Un tipo se adiciona con un conjunto de supertipos con los que quedará conectado. De no existir la lista de supertipos, se ubicará como subtipo del tipo más general del sistema. Esta operación no acarrea problemas.

(3.2) Eliminar un tipo existente C. La eliminación de un tipo no deberá dejar al reticulado desconectado; de ser así, el sistema debe rechazar la operación o tomar un criterio determinado. Un criterio posible es ubicar a los

supertipos del tipo C a eliminar, como supertipos de los subtipos de C (que tenían como único supertipo a C). Los subtipos dejarán de heredar las variables de instancias y métodos de C, por lo que deben ser notificados o se deben propagar a los mismos este cambio.

Si el tipo C era el dominio de una variable de instancia v_i de otro tipo C_i, a V_i se le debe asignar otro dominio (posiblemente, alguno de los supertipos de C).

(3.3) Cambio del nombre de un tipo C. Debe asegurarse que el nuevo nombre es único entre todos los nombres del reticulado. También, debe reflejarse este cambio en cada una de las variables de instancia de otros tipos cuyo dominio era C.

Una alternativa para manejar cambios en los tipos es que cada cambio determine que todas las instancias se amolden al nuevo tipo. Aunque hay situaciones en las cuales la conversión de instancias es razonable o necesaria, no debería ser el único mecanismo existente.

Uno de los motivos por los cuales deberían existir otras alternativas es que la modificación de todas las instancias de un tipo podría no ser práctico. Cuando se manejan grandes cantidades de objetos este proceso podría ser muy costoso.

Otro motivo es que existen programas que manipulan las viejas versiones de los tipos, y probablemente, no puedan acceder a las nuevas, lo que implicaría la conversión de dichos programas.[SKA87]

Además existe una razón que tiene que ver más con la conceptualización del universo de discurso que con la implementación. Que un tipo evolucione, no significa que los usuarios no quieran seguir manipulando las versiones viejas de un tipo, ni que los objetos instanciados en versiones anteriores desaparezcan del sistema (que un tipo Auto cambie sus características a medida que evoluciona la tecnología, no significa que no haya autos viejos).

En el diseño de una metodología para tratar los cambios en el esquema de una Base de Datos, se debe considerar, especialmente, cómo

mantener la compatibilidad entre los objetos existentes y las definiciones de tipos modificadas.

Se presentan dos enfoques para atacar este problema: tamización y conversión.

Tamización consiste en modificar el almacenamiento persistente, 'filtrando' los valores o corrigiéndolos antes de ser usados.

En [SKA87], Skarra y Zdonik exploran un punto de vista en el cual, se ubican filtros entre las instancias de una vieja versión de la definición de un tipo, y los métodos que esperan instancias de una nueva versión del mismo. Se presenta un enfoque alternativo para manejar cambios en los tipos, para evitar que con cada cambio todas las instancias previas sean convertidas a la nueva versión del tipo. Se trata de proveer *transparencia en los cambios de tipos*. Esto es, si T_1, T_2,....,T_n son versiones del tipo T y f es una operación definida sobre instancias de algún T_i, entonces f(t) está bien definida para todo t, tal que t es una instancia de T_j.

El sistema planteado soporta transparencia en los cambios de tipos, bajo un mecanismo de control de versiones para tipos, y permite a los diseñadores de tipos agregar manejadores de error para versiones de los mismos. Los manejadores efectivamente expanden el comportamiento definido por cada versión, de esta forma las instancias de diferentes versiones pueden ser usadas por los distintos programas. Los manejadores agregados a cada versión corresponden al comportamiento no definido por esa versión, pero definido para otras versiones del tipo. Cuando un tipo T es cambiado, una nueva versión T_i es creada. La nueva versión T_i acarrea manejadores para cualquier comportamiento que no está definido localmente, pero está definido por alguna otra versión del tipo T_j. Más aún, las versiones previas del tipo, pueden requerir de manejadores para especificar el comportamiento definido únicamente en la nueva versión.

La transparencia de cambios de tipos es útil para operaciones que iteran sobre todas las instancias de un tipo, sin importar la versión bajo la cual las instancias fueron creadas.

En el sistema ORION se emplea la tamización sobre objetos utilizados por una aplicación; la representación de los objetos es corregida en el momento en que son usados.

Conversión, cambia todas las instancias de un tipo a la nueva definición del mismo, asegurando que definiciones auxiliares (tales como métodos) también compatibilicen con ella.

GemStone intenta integrar ambos puntos de vista (tamización y conversión). [BRE89]

La tamización compromete la velocidad de ejecución, mientras que, la conversión insume mucho tiempo en el momento en que una definición de tipo es modificada.

Existen diferentes trabajos que tratan el problema de cambios en el esquema de la Base de Datos. Banerjee [BAN86] considera el problema de evolución de tipos en el contexto del grafo (reticulado) asociado. Analiza el problema de definir categorías básicas de cambio, proveyendo para cada categoría, reglas para mantener el grafo consistente y para convertir instancias existentes de forma tal que se correspondan con el nuevo grafo de tipos.

Zdonik [SKA87] utiliza versiones de tipos para mantener una visión consistente del grafo asociado y un mecanismo de manejo de excepciones para evitar convertir viejas instancias.

Borgida [BOR85] utiliza un mecanismo de manejo de excepciones en el contexto de una Base de Datos para tratar valores excepcionales. Un objeto contiene un valor excepcional cuando el valor no se corresponde con los contenidos impuestos por la definición del tipo. Enfoca la variación a nivel de instancia.

Además del manejo de versiones referente a la evolución del esquema en la Base de Datos es importante destacar que en ambientes de desarrollo de actividades de diseño, tales como CAD/CAM y Sistemas de Automatización de Oficinas, que utilizan una BDOO como soporte tecnológico, tiene sentido generar y experimentar con múltiples **versiones de un objeto** antes de seleccionar uno que satisface sus requerimientos.

Versionar instancias de un tipo implica que diferentes versiones del mismo objeto difieran, solamente, en los valores de alguna(s) de su(s) propiedad(es).

El hecho de preservar estados alternativos de una entidad particular requiere de la existencia de un mecanismo de versionamiento de objetos para proveer acceso controlado a estos valores.

Un objeto retiene su identidad a lo largo de su existencia aunque su estado puede cambiar. Las versiones son como fotografías de un objeto en ciertos estados, y son modelizados por objetos distintos.

Un objeto es o bien **versionable** o **no-versionable.** Un objeto versionable es una instancia de una clase que la aplicación (o el usuario) especificará como versionable.

El versionamiento a nivel de instancias se aplica sólo sobre objetos que son versionables. Una versión de instancia se puede generar al derivarse de otro objeto o al crearse una nueva instancia de una clase versionable. [KIM87]

Conceptualmente, una instancia tiene asociado un conjunto de versiones de la misma. Al igual que en el conjunto de versiones de un tipo, la relación que existe entre los elementos del conjunto de versiones de una instancia es una relación de orden, llamada *´sucesor-de´*, pudiendo ser un orden parcial o total.

Existe un debate en la literatura sobre cuándo dos instancias de un mismo tipo son diferentes objetos y cuándo son, simplemente, diferentes versiones del mismo objeto. La clave para resolver este problema, es ver a cada atributo de un objeto como *intrínseco* o *no-intrínseco*. Si las propiedades intrínsecas cambian, también cambia el objeto. Las propiedades no-intrínsecas, por otro lado, pueden ser modificadas sin cambiar al objeto de una manera significativa.

Introduciremos a continuación, diferentes puntos de vista para tratar el **versionamiento a nivel de instancia.**

En [FIS89] se describe el mecanismo de versionamiento de objetos de la siguiente forma:

actualizado o borrado, o una nueva versión del objeto es creada, alguno o todos los objetos que lo referencian puede(n) llegar a ser invalidado(s), y entonces necesitan la notificación del cambio para actuar en consecuencia o, en caso contrario, debería existir un mecanismo automático encargado de la propagación de cambios.

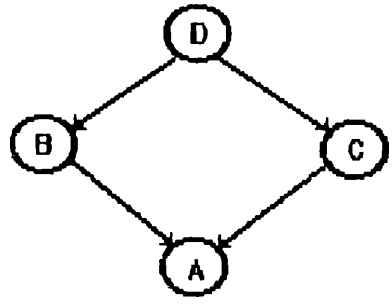

Gráfico 4: El objeto D está compuesto por los objetos B y C, los cuales están formados ambos por el objeto A y lo comparten.

Con referencia al gráfico 4, si el objeto A es modificado, se obtiene una versión A´. Bajo el concepto de propagación de cambios, se obtiene una nueva versión de B y de C, B´ y C´ respectivamente. Y propagando un nivel más, se obtiene D´, versión de D. (Gráfico 5)

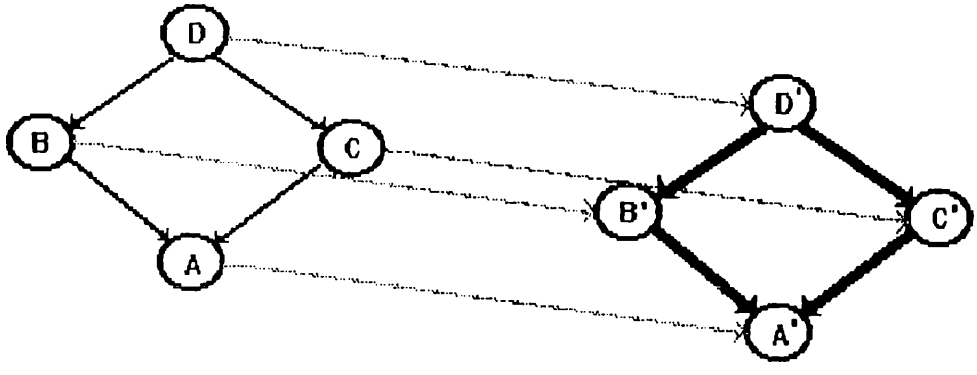

Gráfico 5: Propagación de cambios a partir de A.

Con referencia al gráfico 4, si el objeto A es modificado, se obtiene una versión A´. Bajo el concepto de notificación de cambios, se da aviso al usuario del cambio que se ha producido y queda bajo su responsabilidad el reflejar o no el mismo en los objetos que contienen al objeto componente modificado. [KAT90]

(1.1.2) Eliminar una variable de instancia v con la propiedad de referencia compuesta de un tipo C. Si eliminamos del tipo C la variable de instancia cuyo dominio es de tipo B (el cual a su vez puede ser compuesto), la jerarquía de partes de B (si es que la tiene) o B se desconecta de la raíz de la jerarquía de partes asociada a C. (Gráfico 7)

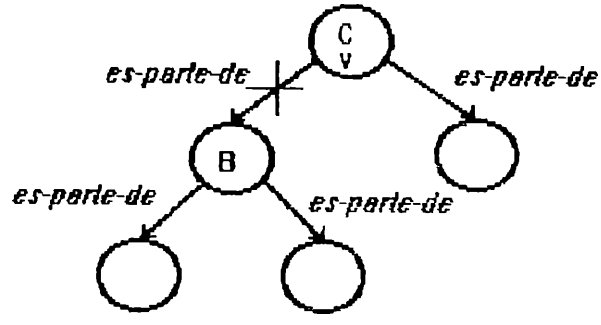

Gráfico 7

(1.1.6) Cambiar la propiedad de referencia de una variable de instancia con dominio de tipo B.

(1.1.6.1) Cambiar un atributo compuesto a un atributo no compuesto, implica desconectar la subjerarquía de partes asociada al tipo B (si es que la tiene) o B de la raíz de la jerarquía de partes asociada a C. (Gráfico 7)

(1.1.6.2) Cambiar un atributo no compuesto con dominio en el tipo B a un atributo compuesto exclusivo. Para poder realizar esta operación hay que verificar que no exista ninguna referencia compuesta (exclusiva o compartida) al tipo B en alguna otra definición de tipo. Como en el caso (1.1.1) queda afectada la jerarquía de agregación asociada al tipo modificado.

(1.1.6.3) Cambiar un atributo no compuesto con dominio en el tipo B a un atributo compuesto compartido. Para permitir realizar esta operación se debe verificar que no exista ninguna referencia compuesta exclusiva al tipo B en alguna otra definición de tipo. Como en el caso (1.1.1) queda afectada la jerarquía de agregación asociada al tipo modificado.

2)Los sistemas de hipertexto son altamente interactivos y usualmente son manipulados por usuarios sin experiencia, por lo que es conveniente que el mecanismo de versionamiento no impida el acceso normal al hipertexto. Un usuario que desea localizar una pieza particular de información no debe luchar con el sistema de versionamiento. Similarmente, un usuario que realiza una corrección o una anotación a un nodo no necesita interactuar con el sistema de versionamiento. Para ello es necesario configurar el sistema de versionamiento para asumir un conjunto de acciones 'por defecto' que produzcan resultados 'intuitivos'.

3)Se desea tener la posibilidad de agrupar cambios en los nodos y a la red tal que ellos puedan ser manejados como una entidad. En general, los usuarios realizan cambios coordinados a un grupo de entidades en la red en algún momento. Por ejemplo, un programador que implementa una nueva característica en un programa existente visitará un conjunto de nodos (probablemente que contienen código fuente o documentación) y realizará cambios apropiados a uno o más de ellos. En muchos casos el programador deseará agrupar todos esos cambios en un conjunto tal que pueda ser referenciado como una entidad. De esta manera, los desarrolladores de sistemas tendrán la posibilidad de anular una modificación cancelando un conjunto de cambios.

4) Cuando un grupo de gente trabaja junta en un proyecto, es conveniente crear pequeñas áreas privadas donde puedan trabajar sin interferir entre ellos. Esto se puede llevar a cabo *particionando* el hipertexto o usando *planos de versiones*. Particionar el hipertexto significa crear áreas que están de alguna manera separadas del cuerpo principal de los datos, mientras que los planos de versiones son simplemente un contenedor para todos los nodos que han cambiado de una versión del hipertexto a la siguiente.

Los usuarios deben tener la posibilidad de trabajar con diferentes particiones o planos de versiones del hipertexto y la posibilidad de luego integrar sus trabajos.

5)Debe ser posible manejar árboles de versiones y soportar operaciones de manejo de la configuración.

6)Durante la vida de un sistema de hipertexto, los usuarios realizan cambios a los datos, entonces la cantidad de almacenamiento

ocupada por las distintas versiones se incrementará. Parte de esa información es útil pero el resto simplemente ocupa espacio.

Por esto es que se deben minimizar los requerimientos de almacenamiento para los sistemas de versionamiento y permitir tomar 'fotos' de partes del sistema para archivar.

En sistemas basados en texto, los requerimientos de almacenamiento pueden ser reducidos manteniendo solamente las diferencias entre versiones del mismo nodo. Para nodos multimedia, junto con texto se puede tener datos binarios representando dibujos, información de audio, gráficos estructurados, etc.

Estos últimos nodos pueden ser vistos como nodos compuestos o como nodos estructurados (nodos que tienen una estructura interna que no es visible para el sistema de hipertexto). Este último caso, permite al sistema realizar *versionamiento selectivo* donde las partes del objeto que no han cambiado no son copiadas, pero están conectadas en ambas versiones. A sí mismo, es difícil ver cómo representar cambios a una entidad de estructura interna desconocida.

Se presentan a continuación varios enfoques para proveer versionamiento de hipertextos que permiten a varios usuarios o aplicaciones trabajar sobre un mismo hipertexto sin interferir entre ellos.

El sistema de hipetexto Neptune fue extendido a Neptune/HAM para soportar **contextos** [DEL87] . Un contexto es un subconjunto privado del hipertexto global. De esta manera se permite realizar actualizaciones en áreas privadas locales. Cada usuario puede *derivar* una vista privada del grafo de hipertexto y realizar modificaciones en esas vistas locales. Cuando se completa un conjunto de cambios se realiza un *merge* del área privada con el área *master*, generando otro contexto.

En [PRE90] se describe otro sistema de hipertexto, Intermedia, que particiona el hipertexto empleando *webs*. Estos son, simplemente, conjuntos de links que conectan los nodos que son accedidos por el usuario. Los links que no pertenecen al web activo no son visibles.

Se pueden construir versiones de la red usando diferentes webs. Además se puede realizar versionamiento lineal de los nodos.

El problema que se presenta es que sólo un web puede estar activo en un momento, de tal forma que no hay manera de comparar dos o más webs. Esta limitación hace dificultoso combinar distintas versiones o crear una nueva versión basada en un número de versiones anteriores.

Otro sistema interesante es PIE (Personal Information Environment) aunque no pasó de ser un prototipo. Este sistema fue desarrollado en Smalltalk.

PIE soporta *layers* que describen cambios coordinados a una o más entidades en la red. A su vez los *layers* pueden ser agrupados en *contextos.*

Los contextos de PIE son más restrictivos que los contextos de HAM porque estos últimos pueden ser pequeños hipertextos.

Al modificar un contexto, PIE crea un nuevo *layer.* Además, el sistema soporta *contratos* que especifican las dependencias entre los nodos.

Prevelakis presenta una alternativa que intenta mejorar algunos de los problemas encontrados en Neptune/HAM, Intermedia y PIE.

En su sistema se construyen *perspectivas* como resultado de transformaciones aplicadas a la red inicial. Los usuarios construyen *overlays*, que son conjuntos de modificaciones relacionadas que implementan una tarea específica. Combinando distintos overlays se generan las perspectivas. Cuando los usuarios desean combinar sus trabajos, necesitan simplemente definir una nueva perspectiva que combine los overlays contenidos en cada una de las perspectivas individuales.

Estas perspectivas, descriptas en [PRE90], pueden ser *transitorias* o *permanentes.* En el primer caso las perspectivas deben ser construídas cada vez que se necesitan. Por otro lado, cuando se utilizan perspectivas permanentes el sistema salva el estado de la red derivada de tal forma que pueda ser manejada como una entidad cada vez que necesita ser utilizada.

Parte

2

Nuestro proyecto

Capitulo 2: El modelo y el prototipo

2.1 - Motivacion

2.2 - Bases de Datos Orientadas a Objetos para Cad

2.2.1 - Introduccion

2.2.2 - Fundamentos del modelo

2.2.3 - Descripcion del modelo

2.3 - El prototipo

2.3.1 - Descripcion del prototipo

2.3.2 - Desarrollo del prototipo

2.3.2.1 - Por que elegimos un ambiente orientado a objetos ?

2.3.2.2 - Ciclo de vida

2- EL MODELO Y EL PROTOTIPO

2.1- MOTIVACION

Luego de haber leído y estudiado el tema de BDOO, encontramos que hay diversas funcionalidades de un SMBDOO que aún están en etapa de investigación y desarrollo. Entre ellas se encuentran: el manejo de persistencia, la optimización de queries, la propagación de cambios en objetos compuestos, el manejo y control de versiones, etc.

Nos resultó atractivo, especialmente, investigar el versionamiento en BDOO. Una de las causas fue el hecho de que es un aspecto nuevo a tratar en el área de las Bases de Datos. Los SMBD tradicionales (Jerárquico, en Red, Relacional) no cuentan con esta funcionalidad debido a que sus aplicaciones típicas (comerciales) no requieren versionamiento porque los modelos de datos que soportan no son tan ricos como el modelo de datos orientado a objetos.

Por otro lado, como consecuencia del abaratamiento de los costos, surgen nuevas aplicaciones que requieren de las Bases de Datos como soporte tecnológico, tales como CAD/CAM, CASE, IA, Procesamiento de imágenes, Automatización de Oficinas, etc. Estas aplicaciones tienen la particularidad de tratar con tipos de datos complejos. Así mismo, en aquellas que involucran ambientes de diseño, caracterizados por su naturaleza exploratoria e iterativa, tiene sentido mantener estados alternativos de los objetos.

Investigando el tema de versionamiento en BDOO, encontramos que en Hipertextos, otra área de aplicación de punta, también se está trabajando al respecto.

Al observar que, tanto en Hipertextos como en BDOO tiene sentido reflejar la evolución de un objeto, nos planteamos llegar a un modelo común de versionamiento.

Frente a esta meta que nos propusimos, recibimos la influencia de varios autores, para concretar nuestro objetivo.

En [MEN89a] se presenta un lenguaje de query visual llamado GraphLog, basado sobre una **representación de grafos** para la Base de Datos y los queries.

Los queries de GraphLog son patrones que deben estar presentes o ausentes en el grafo de la Base de Datos. Cada patrón es llamado **"Grafo de query"** y define un conjunto de nuevas aristas (es decir, una nueva relación) que son adicionadas al grafo si el patrón es encontrado. Las aristas en el grafo de query se corresponden con aristas o caminos en la Base de Datos. Se utilizan expresiones regulares para calificar estos caminos.

En [MEN89b] se emplea el formalismo visual de GraphLog sobre un **grafo hiperdocumento**. Se buscan en el grafo todas las ocurrencias de los patrones de grafos (formulación de queries).

Se presentan ejemplos de queries GraphLog aplicados a varios Sistemas de Hipertexto diferentes, dando así evidencia del poder expresivo del lenguaje, tanto la conveniencia y naturalidad de su representación gráfica.

En [PRE90] grupos de modificaciones que implementan una tarea específica se agrupan en overlays. Combinando overlays sobre el hipertexto original se construyen perspectivas que pueden ser consideradas como "vistas" del hipertexto. Una perspectiva puede contener cualquier número de overlays organizados en cualquier orden.

En [AHM91] se describen las características de los objetos de diseño. Se propone una clasificación de los atributos de un objeto de diseño en intrínsicos y no-intrínsicos. Esta contribuye al tratamiento de la prevención de la proliferación de versiones. También se presenta una metodología para el manejo de versiones a nivel de tipos e instancias.

Nuestra intención, entonces, era encontrar un **modelo común** para el soporte de versiones tanto para Hipertextos como para BDOO. Viendo que el esquema de una BDOO y un hipertexto son soportados con un grafo, y que una versión surge a partir de aplicar un conjunto de modificaciones sobre el grafo, teníamos dos alternativas:

a) Mantener los distintos overlays (conjuntos de modificaciones) que se realizan sobre el grafo base y **construir**

una perspectiva ($version_n$), como la composición de overlays sobre el mismo, en el momento de querer acceder a ella. Y además, dar la posibilidad de que una perspectiva pase a ser *definitiva* [PRE90]. Una perspectiva definitiva es considerada como un grafo base, a partir del cual se puede generar overlays.

b) **Mantener** las distintas versiones (evolución que sufrió el grafo base) construyéndolas una única vez, en el momento de su generación. Una versión se construye a partir de aplicar un conjunto de modificaciones sobre el grafo base. A partir de ésta se pueden generar nuevas versiones.

Detallamos a continuación las ventajas y las desventajas que hallamos en ambos enfoques:

Ventaja del enfoque a):

- **potente** por la cantidad de perspectivas que se pueden generar combinando y reutilizando los distintos overlays.

Desventaja del enfoque a):

- aumenta la **complejidad** de la implementación del modelo ya que las perspectivas se construyen en el momento de querer acceder a ellas.

Esta desventaja se minimiza cuando se trabaja con perspectivas definitivas.

Ventajas del enfoque b):

- simplifica la implementación del modelo debido a que las versiones se construyen una única vez y los conflictos que pudieran existir se resuelven en ese momento;

- el acceso a una versión es inmediato (no se necesita construirla).

Desventaja del enfoque b):

- **rígido** porque un conjunto de modificaciones sólo es usado para generar una versión y no puede ser reutilizado para generar otra.

Finalmente, optamos por mantener las distintas versiones. Aunque el modelo es rígido, en cuanto a la reutilización de conjuntos de modificaciones, ésta no es una desventaja tan importante como lo es la complejidad inherente al primer modelo.

2.2- EL MODELO DE VERSIONAMIENTO

2.2.1- INTRODUCCION

Realizamos un modelo que soporta versiones de un grafo de contenido arbitrario.

En particular, detallamos el soporte de versiones de hipertextos, de un esquema de BDOO y de un esquema de BDOO para CAD.

Una versión en el modelo se genera a partir de un conjunto de modificaciones aplicadas sobre una versión anterior. Se comienza con el grafo original, considerado la primer versión (versión 0).

Modelizamos la evolución agrupando todas las versiones de un mismo grafo dentro de un conjunto de grafos, donde cada elemento del conjunto se corresponde con una versión.

2.2.2- FUNDAMENTOS DEL MODELO

En las áreas descriptas en los puntos 1.1, 1.2 y 1.3 encontramos que las estructuras que soportan la representación interna de una BDOO, una BDOO para CAD y un Hipertexto tienen ciertas similitudes. En todas ellas, los conceptos que se manipulan se pueden representar por entidades conceptuales de complejidad arbitraria y existen, en todos los casos, relaciones que las vinculan.

En BDOO la definición de un tipo se corresponde con una entidad conceptual. Las definiciones de tipos se organizan jerárquicamente por medio de la relación *es-un*. Si, en particular, especializamos las BDOO para soportar CAD, las entidades que se manejan conceptualmente son, en general, de gran complejidad. Además de la relación *es-un* existente entre las definiciones de tipos en BDOO, existe otra

relación llamada *es-parte-de* que vincula la entidad compleja (definición de tipo compuesto) con sus componentes, donde cada componente es una definición de tipo que, a su vez puede ser un tipo compuesto.

En hipertextos, los documentos o parte de ellos se corresponden con entidades conceptuales. En este caso, los conceptos se vinculan por medio de la relación *está-relacionado-con*.

Todas las entidades conceptuales antes mencionadas pueden ser modelizadas por *nodos* (de contenido arbitrario) en un *grafo* y las relaciones entre ellas a través de *links*.

En el caso de un nodo de gran complejidad, que modeliza la definición de un tipo compuesto, puede verse como un subgrafo en el cual se establece la relación *es-parte-de* que vincula los nodos componentes con él. Conceptualmente se pueden distinguir dos niveles de abstracción. En el primer nivel se ven las relaciones entre los nodos que representan la jerarquía de tipos. En el segundo nivel se ve la relación que existe entre un nodo complejo y sus componentes. (Gráfico 8)

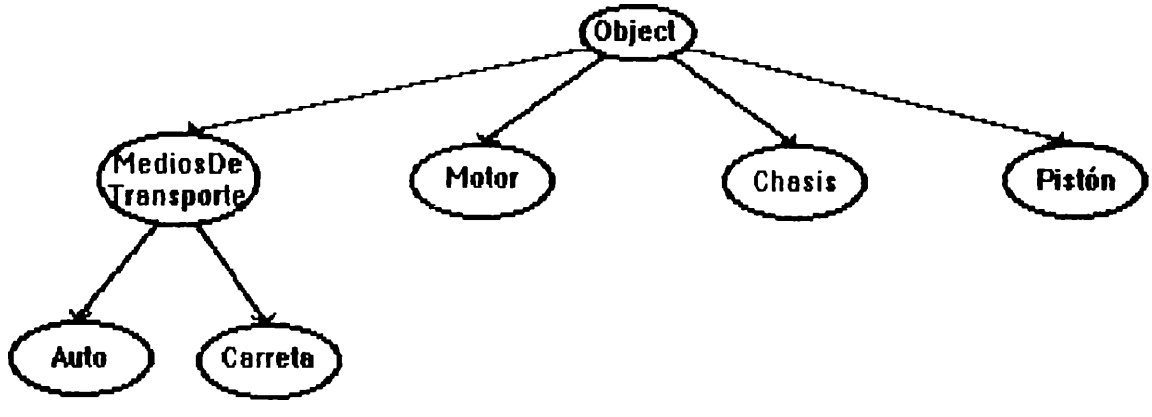

Gráfico 8.a: Jerarquía de tipos (primer nivel).

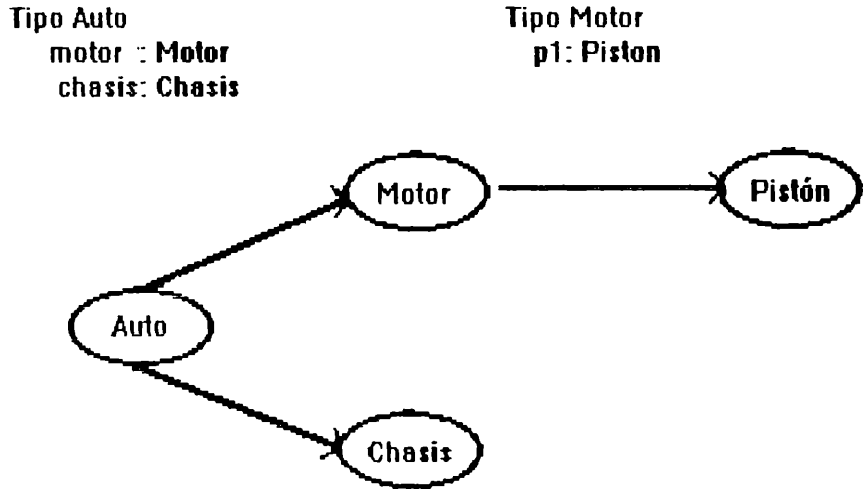

Gráfico 8.b: Jerarquía de agregación (segundo nivel)

En el punto 1, por otro lado, planteamos la necesidad de soportar **versiones**. Considerando que la estructura interna de las BDOO, BDOO para CAD, Hipertextos puede ser modelizada como un grafo arbitrario, descubrimos puntos en común en el manejo de versiones en las distintas áreas, que se reflejan de forma equivalente en el grafo.

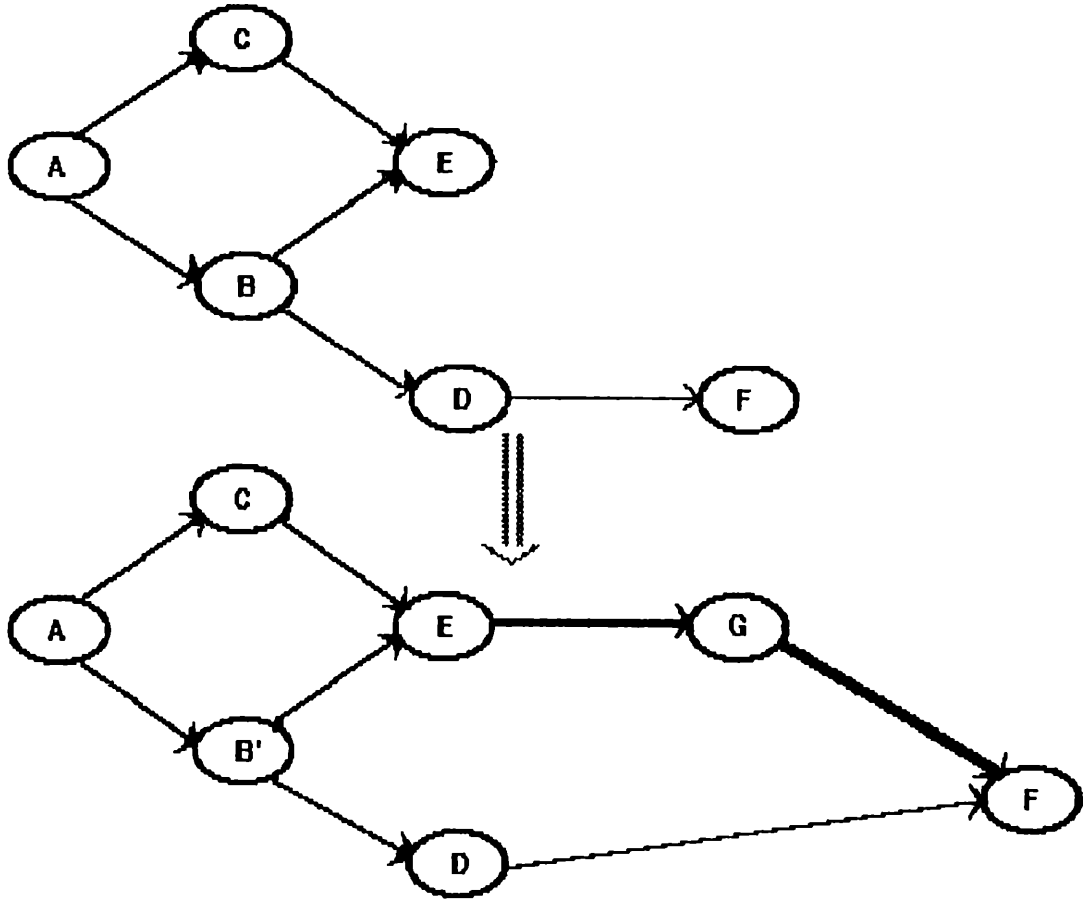

Gráfico 9.a: Agregado del tipo G como subtipo de E y supertipo de F

En las BDOO nos interesa versionar el esquema de la Base de Datos cambiando la jerarquía y/o la definición de un tipo.

La jerarquía de tipos se representa con un DAG con lo cual un cambio en ésta (agregado o borrado de un sub/super tipo) genera una versión del grafo en la que se agregó o borró un link, y tal vez un nodo. (Gráfico 9.a). La definición de un tipo, a su vez, se representa con un nodo. Un cambio en la definición de un tipo (modificación en los atributos y/o métodos) genera una versión del nodo, quedando sus subtipos afectados por esa modificación. Este cambio genera una versión del grafo. (Gráfico 9.b).

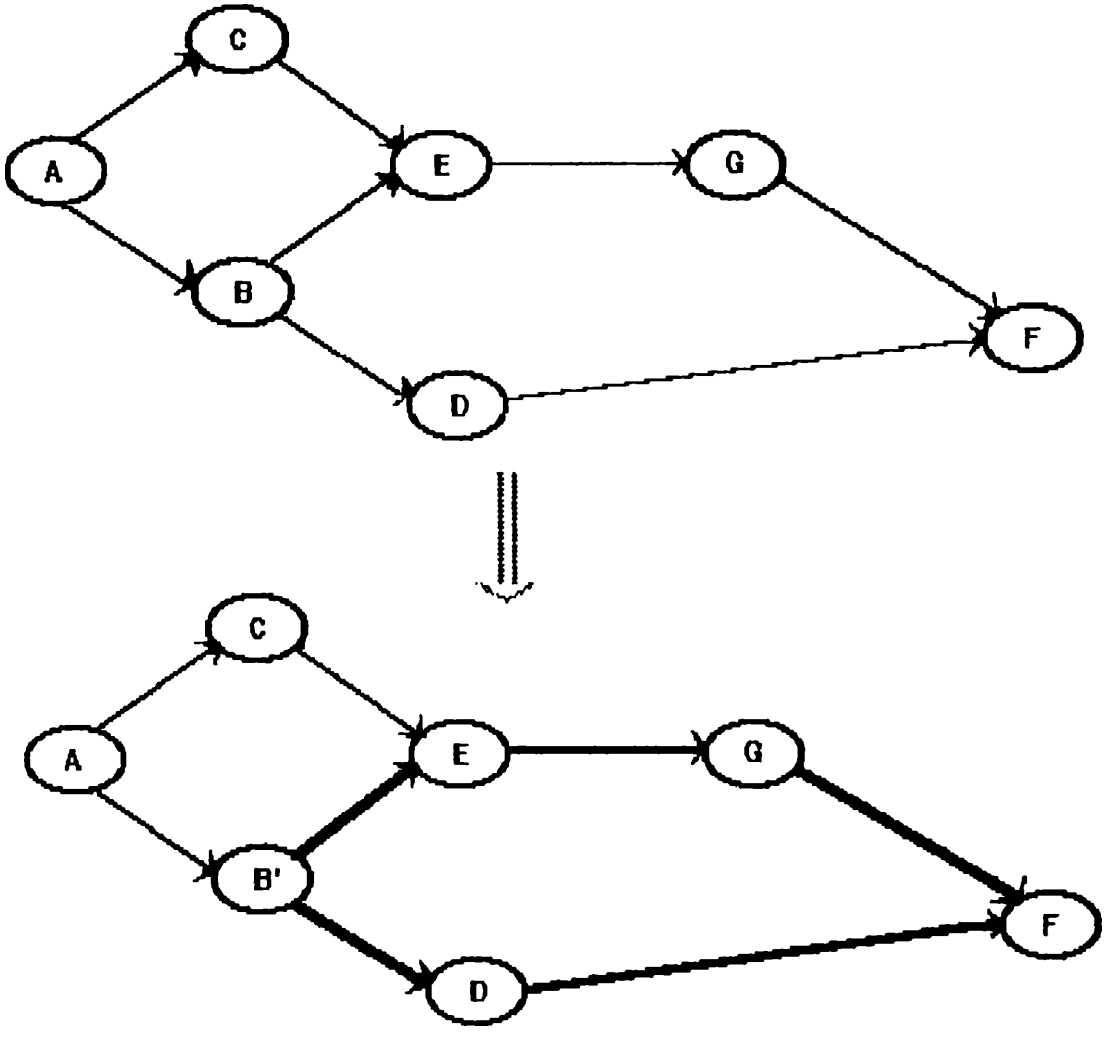

Gráfico 9.b: Modificación en la definición del tipo B y cómo afecta el cambio en todos sus subtipos

En las BDOO para CAD, al soportar objetos de diseño, la definición de un tipo compuesto tiene asociada una jerarquía de

agregación representada por un DAG. Esta jerarquía refleja la relación que existe entre un tipo compuesto y el tipo de sus componentes (que a su vez puede ser compuesto).

Al trabajar con objetos compuestos, el manejo de cambios en la definición de tipos es más específico que el descripto anteriormente en BDOO:

- Si modificamos la definición de un tipo T, agregando o borrando un atributo de tipo *primitivo* se genera una nueva versión de T. (Gráfico 10.a).

- Si modificamos la definición de un tipo T, agregando o borrando un atributo de tipo *no-primitivo* se genera una nueva versión de T y se ve afectada la jerarquía de agregación asociada (agregado o borrado de una definición de tipo componente). (Gráfico 10.b).

- Si modificamos la definición de un tipo T:

  a)pasando a ser el dominio de una de sus variables de instancia un tipo *no-primitivo* (T_1) se genera una nueva versión de T que tendrá asociada la jerarquía de agregación correspondiente a la definición del tipo T_1. (Gráfico 10.c).

  b)pasando a ser el dominio de una sus variables de instancia, un tipo *primitivo* T_2, se genera una nueva versión de T. (Gráfico 10.d)

En todos los casos se debe considerar cómo afecta a los subtipos en la jerarquía de tipos (como planteamos en el caso general de BDOO). Además, hay que analizar si el tipo modificado (T) es el tipo de alguna(s) variable(s) de instancia de algún(os) tipo(s) compuesto(s). En este caso se propaga el cambio hacia el tipo que lo compone generando una nueva versión del mismo (Gráfico 10.e). Recursivamente se debe considerar cómo afecta la modificación de este tipo compuesto a los subtipos en la jerarquía de tipos.

Tipo T
Superclases: Object
x : Integer
y : String

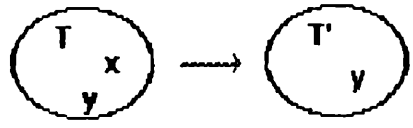

Gráfico 10.a: Borrado del atributo primitivo x, generando T' con la misma jerarquía de agregación.

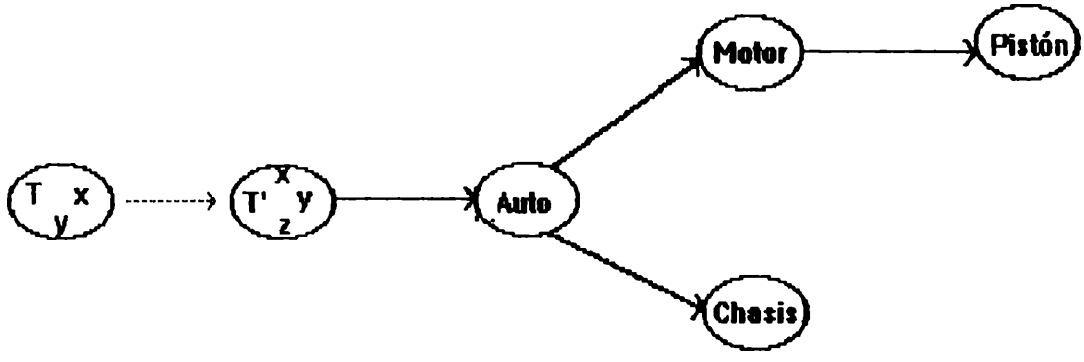

Gráfico 10.b: Agregado del atributo z al tipo T que es de tipo Auto (no-primitivo). Jerarquía de agregación para el tipo T' (nueva versión de T)

Tipo T
x: Integer
y: String

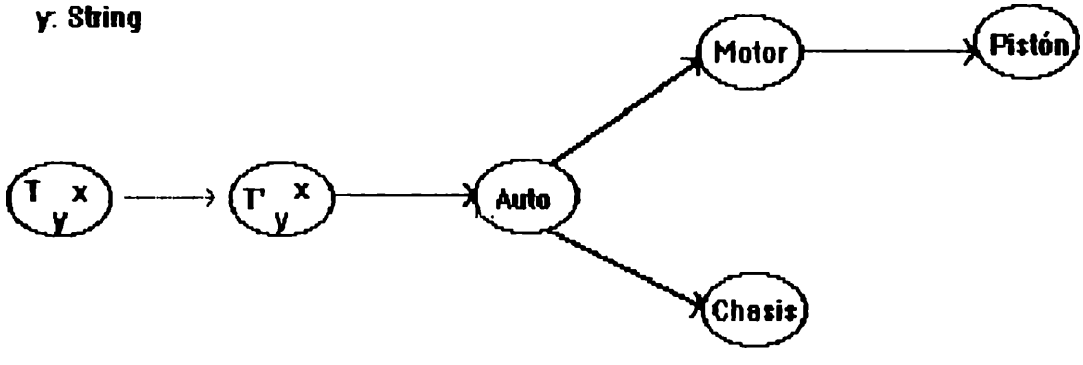

Gráfico 10.c: Modificación del dominio de una de las variables de instancia del tipo T pasando a ser de un tipo no-primitivo

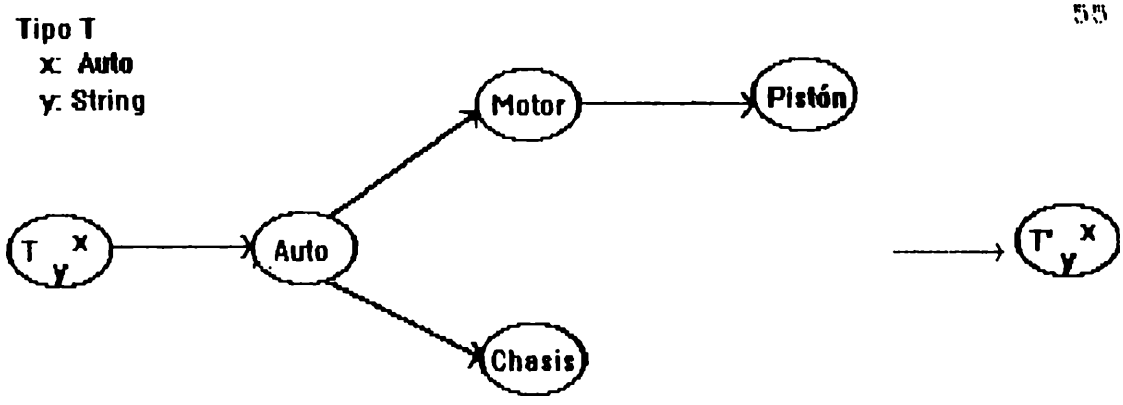

Gráfico 10_d: Modificación del dominio de una de las variables de instancia del tipo T pasando a ser de un tipo primitivo

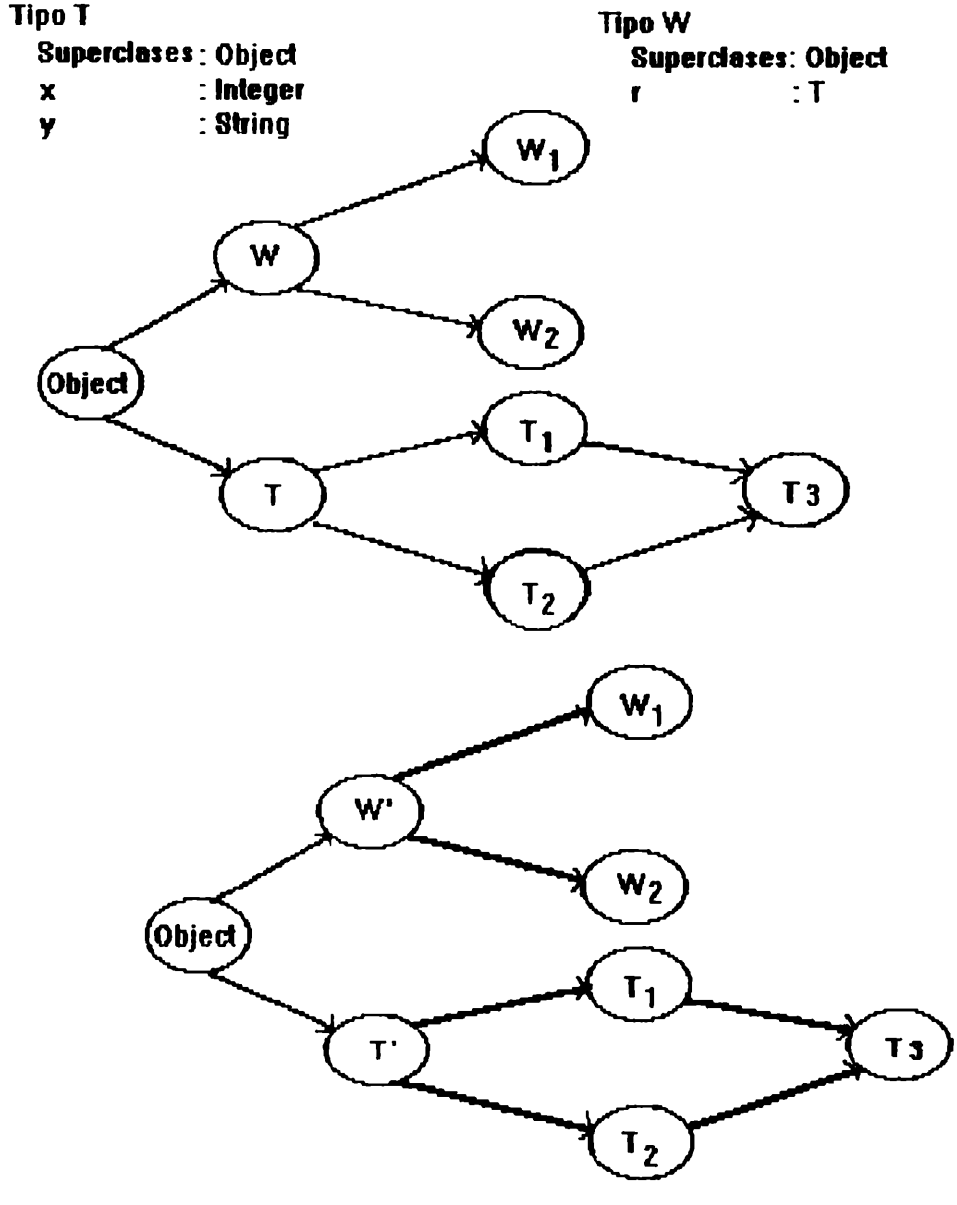

Gráfico 10.e: Modificación en la definición del tipo T. Como T es el tipo de una variable de instancia del tipo compuesto W, se genera una nueva versión de W, W'

En hipertextos interesa realizar *grupos de cambios* a los documentos (o parte de ellos) y a las relaciones que hay entre los mismos, generando así una nueva versión del hipertexto.

Los documentos (o parte de ellos) se representan por medio de nodos y las relaciones por medio de links en un grafo.

Un cambio en un nodo genera una nueva versión del mismo.

El borrado (agregado) de links entre nodos, y/o los cambios en un nodo, generan una nueva versión del grafo. (Gráfico 11).

Extrayendo la generalidad de todos los casos antes mencionados, vemos que interesa mantener un *conjunto de versiones* (versiones de un hipertexto, de una jerarquía de tipos en una BDOO, de una jerarquía de tipos de complejidad arbitraria en una BDOO para CAD) que pueden soportarse por medio de un conjunto de grafos. Además, cualquiera de los versionamientos descriptos refleja, *estructuralmente,* a las modificaciones de la misma manera sobre el grafo.

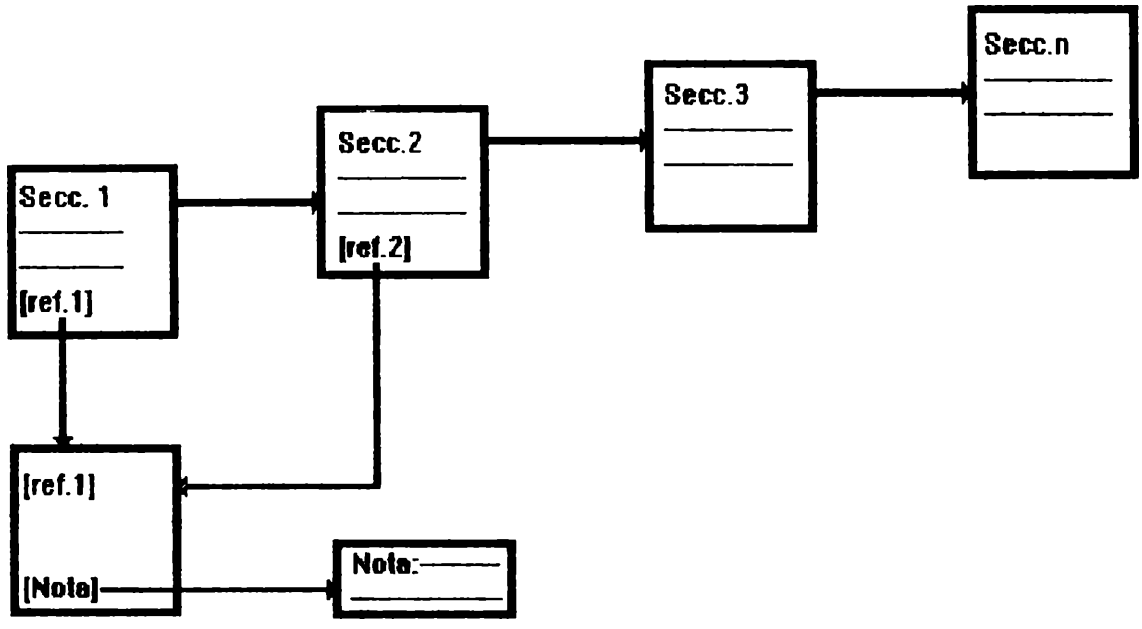

Gráfico 11.a: Hipertexto

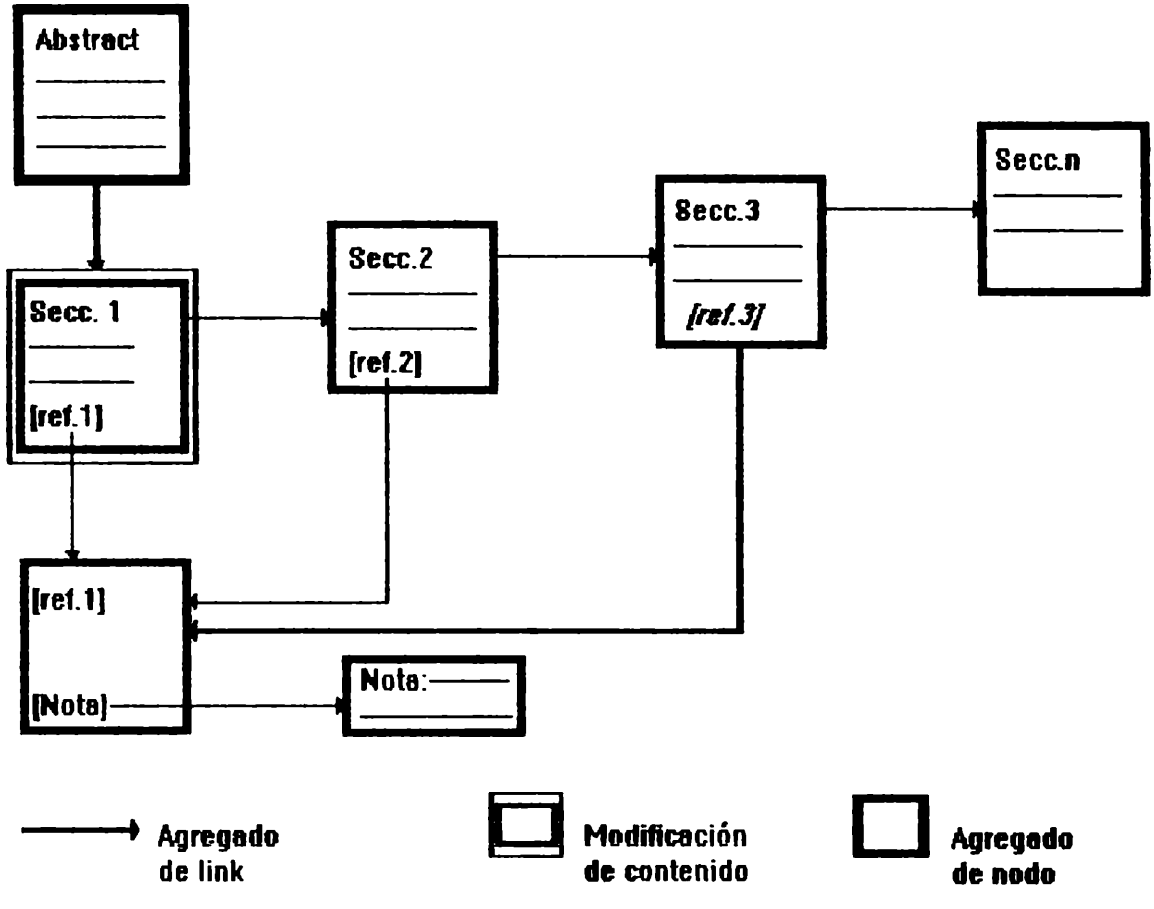

Gráfico 11.b: Modificaciones sobre el hipertexto del gráfico 11.a que generan una versión del hipertexto.

2.2.3- DESCRIPCION DEL MODELO

En el punto 1.4.1 se describieron, básicamente, tres intenciones para versionar o mantener un conjunto de versiones de objetos que se detallarán a continuación:

a) mantener la evolución de un objeto durante el proceso de diseño hasta alcanzar el objetivo deseado, descartando luego los estados intermedios y conservando sólo el estado final;

b) mantener la evolución de un objeto con la finalidad de reflejar las distintas configuraciones que variaron en el tiempo, para, por ejemplo, poder hacer un análisis comparativo;

c) mantener diferentes vistas de un objeto en un ambiente de múltiples usuarios o aplicaciones.

El propósito del modelo es reflejar la evolución de una jerarquía de tipos de una BDOO, de una jerarquía de tipos de complejidad arbitraria de una BDOO para CAD y de un hipertexto. Debido a esto, el modelo se basa en la segunda intención; es decir, permite mantener la evolución de un grafo que sirve como soporte, tanto para las jerarquías de tipos descriptas, como para un hipertexto.

El modelo es un manejador de versiones de grafos.

Sin pérdida de generalidad, los grafos que modelizamos **son dirigidos**. En el caso de un grafo no dirigido, éste se puede representar con un grafo dirigido.

Por otro lado, los grafos contienen **información de cualquier tipo** como contenido de sus nodos.

Los grafos son **no-conexos**.

Modelizamos la evolución de un grafo, agrupando todas las versiones de un mismo grafo dentro de un **conjunto de grafos**. Cada elemento de dicho conjunto se corresponde con una versión y, en particular, el grafo original es considerado un elemento.

Las modificaciones que se pueden realizar sobre un grafo son las operaciones usuales sobre grafos:

- crear

- insertar nodo

- insertar arista

- eliminar nodo

- eliminar arista

- modificar el contenido de un nodo

Si se modifica el contenido de un nodo se genera una versión del mismo. La versión del nodo queda vinculada a través de la relación *sucesor-de* con el nodo, pasando a formar parte de la historia del nodo. La *historia de un nodo* refleja la evolución del mismo luego de sufrir sucesivas modificaciones.

Luego de realizar un conjunto de modificaciones cualquiera sobre el grafo, inclusive modificación al contenido de un nodo (que genera una nueva versión del mismo), se obtiene una nueva una **configuración.** (Gráfico 12)

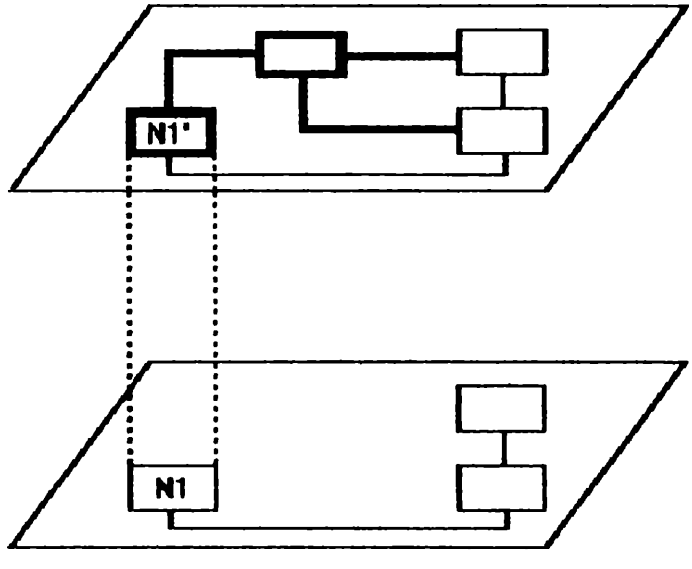

Gráfico 12: Generación de una configuración al aplicar un conjunto de modificaciones sobre el grafo.

Una **configuración** en el modelo es una versión del grafo. Representa una vista del mismo y contiene las aristas y las versiones de los nodos correspondientes a dicha versión del grafo.

El grafo original es considerado la configuración inicial.

El modelo no restringe a que las aristas y nodos de una configuración, que no se han cambiado (respecto a la configuración sobre la cual se aplicaron una serie de modificaciones), deban ser copiadas en la nueva. Por el contrario, se recomienda a los implementadores del modelo proveer un mecanismo para evitar la duplicación innecesaria de información.

Una versión se puede derivar a partir de una o más versiones predecesoras y puede tener múltiples versiones sucesoras que son llamadas alternativas. Este es el caso más general de versionamiento alternativo.

El modelo permite **versionamiento alternativo**, con la salvedad de que cada versión se genere a partir de una única versión predecesora.

El conjunto de versiones de un grafo, en el modelo, es un **orden parcial.** La relación que existe entre las versiones es la relación *'sucesor-de'* y una versión puede tener muchos sucesores.

Para soportar versionamiento alternativo, el conjunto de versiones se puede representar mediante una estructura de árbol.

En el gráfico 13 se detalla la generación de configuraciones alternativas.

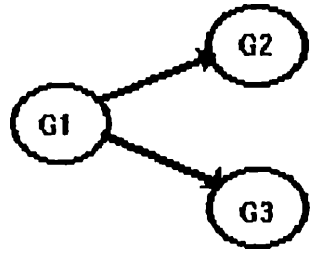

Gráfico 13 : Sean G1, G2 y G3 configuraciones. G2 surge de aplicar un conjunto de modificaciones sobre la configuración G1. De igual forma, G3 es una derivación de G1. G2 y G3 son configuraciones alternativas.

Con referencia a la historia de un nodo, la relación *'sucesor-de'* que existe entre la versión de un nodo y la versión a partir de la cual fue generada, también refleja un orden parcial; es decir, que puede haber versiones alternativas del contenido de un nodo.

En el gráfico 14 se detalla la generación de versiones alternativas de nodos.

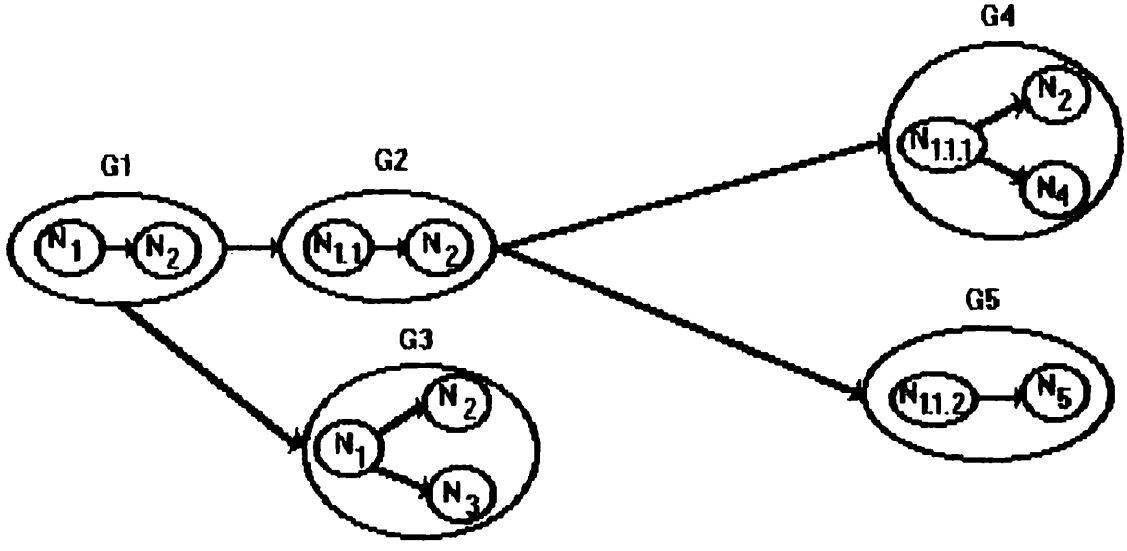

Gráfico 14-a)

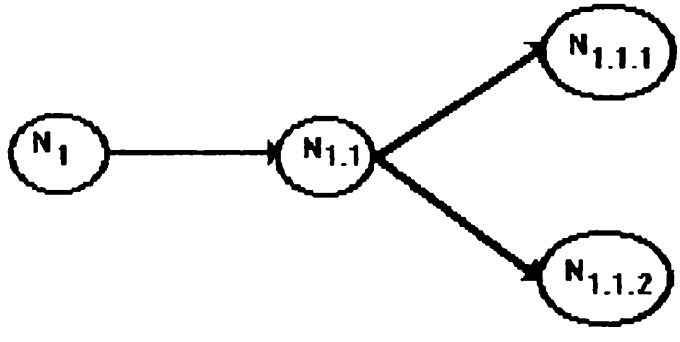

Gráfico 14 - b)

Gráfico 14 -a): La configuración G2 se obtiene a partir de G1, modificando el contenido del nodo N1. La configuración G3 se obtiene a partir de G1, agregando el nodo N3 y vinculándolo con el nodo N1. G2 y G3 son configuraciones alternativas de G1. La configuración G4 se obtiene a partir de G2, modificando el contenido del nodo N1.1, agregándole el nodo N4 y vinculando este último con el nodo N1.1.1 (versión de N1.1). La configuración G5 se obtiene a partir de G2, modificando el contenido del nodo N1.1, borrando el nodo N2, agregando el nodo N5 y vinculándolo con el nodo N1.1.2 (versión de N1.1). G4 y G5 son configuraciones alternativas de G2.

Gráfico 14 -b): Abstracción de la historia del nodo N1 en relación a las configuraciones del gráfico 14-a).

Si restringimos a que las versiones tengas un único sucesor, tenemos un conjunto lineal de versiones.

El versionamiento lineal se puede representar utilizando versionamiento alternativo, como una particularización del mismo. Un conjunto **lineal** de versiones es un orden total y una versión puede ser agregada solamente al final. Todas las versiones previas son de lectura únicamente. Las versiones lineales corresponden al caso en que se representa evolución lineal.

2.3- EL PROTOTIPO

2.3.1- DESCRIPCION DEL PROTOTIPO

El prototipo implementa en su totalidad el modelo descripto en el punto 2.2.3, con una restricción sobre la forma de versionamiento para soportar **versionamiento lineal** solamente.

Como vimos anteriormente, el modelo ataca la forma más general de versionamiento (versionamiento alternativo). A pesar de que en ciertas aplicaciones (principalmente, aquellas que involucran actividades de diseño) el versionamiento alternativo es de gran interés; por una cuestión de simplicidad en la implementación del modelo, decidimos emplear versionamiento lineal. Queda abierta, de igual forma, la posibilidad de expandir el prototipo para que soporte versionamiento alternativo. Intentaremos mostrar, más adelante, cómo realizar esta extensión.

Con el propósito de evitar la proliferación de versiones, se tomaron los recaudos necesarios en la implementación. Esto se llevó a cabo eligiendo una estructura que permitiera agrupar la información de forma tal de no duplicarla.

El prototipo asegura **consistencia sintáctica.** Por ejemplo, no permite incorpar una arista, en una configuración dada, que vincule un par de nodos donde alguno de ellos existe. Otro caso es aquél, en el cual, el prototipo elimina todas las referencias que relacionan a un nodo con otros cuando éste es borrado de una configuración.

2.3.2- DESARROLLO DEL PROTOTIPO

2.3.2.1- ¿POR QUE ELEGIMOS UN AMBIENTE ORIENTADO A OBJETOS?

Pensamos que un ambiente orientado a objetos era la opción más adecuada para el desarrollo del prototipo.

Detallaremos a continuación una serie de características de la programación orientada a objetos que justifican la elección del ambiente.

En nuestra aplicación manejamos, básicamente, los siguientes conceptos que pueden ser mapeados con objetos: hipertextos, grafos, jerarquías de tipos, hipertextos versionables, grafos versionables, jerarquías de tipos versionables. Agrupamos estos objetos en clases que describan su comportamiento y sus propiedades. Modelizamos los grafos versionables como una **generalización** de todas las aplicaciones para las cuales era nuestra intensión soportar el manejo de versiones. Tanto los hipertextos versionables como las jeraquías de tipos versionables son **especializaciones** de los grafos versionables, los cuales factorizan las propiedades y comportamiento común a todos ellos. Este proceso natural de **abstracción** empleado para modelizar situaciones complejas del mundo real es fácilmente representable en un ambiente orientado a objetos donde se cuenta, entre otros, con el mecanismo de **herencia.**

Cuando se especializa una clase, frecuentemente, se emplementan algunos métodos en la subclase, pero por las ventajas del **polimorfismo** este refinamiento es transparente a las clases que utilicen a la clase especializada.

Nuestra intención era modelizar el versionamiento de un grafo, independientemente del contenido de sus nodos. Aquí es donde contar con la posibilidad de **encapsulamiento** contribuye a independizar las responsabilidades del grafo de las del contenido de los nodos que lo componen. De esta forma, los nodos serán **responsables** de su versionamiento.

2.3.2.2- CICLO DE VIDA

En la realización del prototipo se siguieron las etapas clásicas del ciclo de vida de un software.

Etapa de análisis de requerimientos

En la fase de *análisis de requerimientos* se tuvieron en cuenta el marco en el cual está inserto el prototipo, su función, como así también, los requerimientos expuestos en el punto 2.3.1. Se obtuvo como resultado la especificación del prototipo que se detallará a continuación:

ESPECIFICACION DEL PROTOTIPO

Realizamos un Sistema que permite manejar distintas versiones de un grafo. Un grafo está representado por un conjunto de nodos y un conjunto de aristas, donde cada arista expresa una relación entre dos nodos y cada nodo tiene un contenido asociado. El grafo no está restringido a ser conexo (puede tener nodos que no estén relacionados con ningún otro).

Trabajamos con grafos dirigidos (grafos cuyas aristas son dirigidas). Una arista dirigida (i,j) es un arco dirigido que conecta al nodo i con el nodo j.

Soportamos las operaciones usuales para los mismos : Crear, Insertar nodo, Insertar arista, Eliminar nodo, Eliminar arista, Modificar contenido de un nodo y Navegar a través del grafo.

No existen restricciones acerca del contenido de un nodo (son arbitrarios).

Modelizamos la evolución de un grafo dirigido manteniendo un conjunto de versiones del mismo.

Obtenemos una nueva versión del grafo a partir de un conjunto de modificaciones aplicadas a la última versión del mismo (*versionamiento lineal:* todas las versiones previas son de lectura solamente). Una versión de un grafo dirigido, a su vez, es un grafo dirigido.

Las modificaciones que pueden generar una nueva versión del grafo son: Crear un grafo vacio, Insertar un nodo al grafo, Eliminar un nodo, Insertar una arista conectando un nodo con otro, Eliminar una arista y Modificar el contenido de un nodo. Esta última genera un nuevo nodo con el contenido modificado y queda vinculado a través de una relación *sucesor-de* con el nodo que se está modificando, pasando a formar parte de su **historia**. La historia de un nodo representa la evolución del mismo, obtenido a partir del nodo original y reflejando las sucesivas modificaciones a su contenido.

Considerando que el usuario puede realizar una modificación errónea, le posibilitamos corregirla sin generar una nueva versión, evitando así la proliferación de versiones sin sentido. De igual manera permitimos deshacer la última versión del grafo si el usuario descubre que ésta no le sirve y sí la anterior.

Manejamos los siguientes recorridos a través del grafo versionable :

1) Navegación dentro de la versión i (V_i) con la posibilidad de recorrer la historia de un nodo.

A partir de un nodo N_i y una versión V_i el usuario recorre cada nodo eligiendo en cada uno cuál será el nodo adyacente que seguirá en su navegación o eligiendo recorrer su historia (luego retorna al nodo que pertenece a la versión por la que estaba navegando para continuar con el recorrido).

2) Navegación a través de la historia de un nodo.

A partir de un nodo N_i dado se recorre la historia de ese nodo (comenzando por el nodo original), pudiéndose ver las versiones previas y las posteriores al nodo N_i.

Etapa de diseño

En esta etapa utilizamos la metodología de diseño orientado a objetos, llamada "Diseño basado en responsabilidades". Para el lector no familiarizado con la misma, se recomienda leer el Apéndice B.

Como resultado de aplicar esta técnica se obtuvieron las tarjetas de clases que pueden ser consultadas en el Apéndice C. Además se obtuvo la jerarquía de clases que se muestra en el gráfico 15.

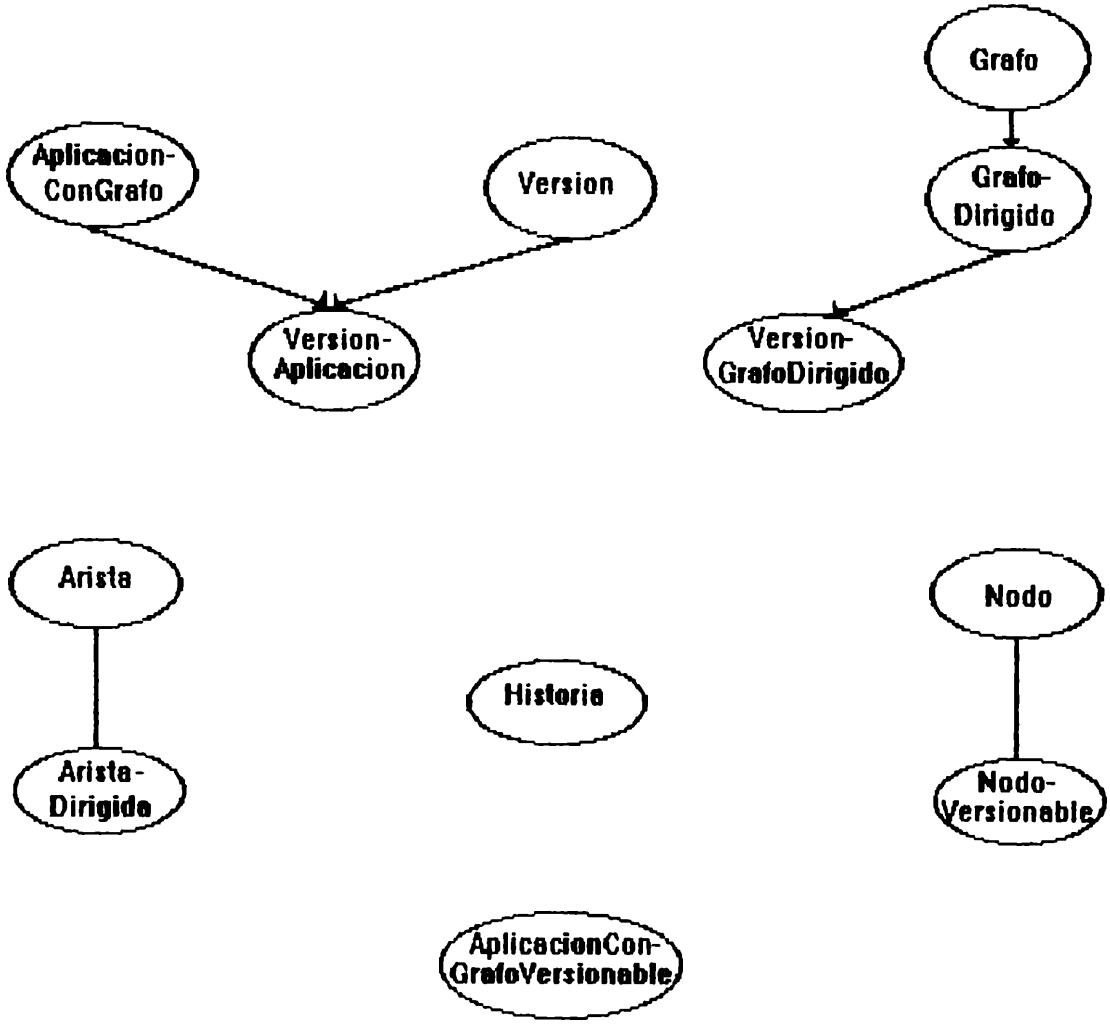

Gráfico 15: Jerarquía de clases del prototipo.

Justificación de la utilización de una técnica de diseño orientado a objetos.

Un diseño orientado a objetos facilita la implementación al mejorar la comunicación entre el diseñador y el implementador. Esto se debe a que es un buen vehículo para modelizar situaciones del

mundo real. El implementador tiene, entonces, un mayor conocimiento del problema antes de comenzar a codificar.

Aunque se utilice una técnica de diseño orientado a objetos, no es estrictamente necesario utilizar un lenguaje orientado a objetos para la implementación. Sin embargo, los lenguajes orientados a objetos proveen un mejor soporte para la implementación de un diseño orientado a objetos que los lenguajes procedurales.

Si se utiliza una técnica de diseño orientado a objetos, se simplifica la resolución de uno de los problemas propios de la implementación, el de encapsular piezas del sistema en unidades que puedan ser implementadas sin considerar las interacciones con el resto del sistema. Esto tiene dos ventajas. Primero, dividir el trabajo en muchas entidades permite al cordinador del proyecto distribuir, en forma natural, el trabajo de implementación entre muchos programadores. Segundo, si, luego de comenzar con la etapa de implementación, por alguna razón, la interfase entre dos entidades debe ser modificada, el sistema cambia sólo en un punto, las otras partes del mismo no son afectadas.

En cuanto al testeo del producto de software resultante, si se empleó una técnica de diseño orientado a objetos, las entidades del sistema pueden aislarse y testearlas una por vez. Es más fácil ubicar dónde se produjo un error, se limitará a un entidad específica.

Similarmente, la especificación de las interfases entre entidades, realizada en la fase de diseño, permite a quienes llevan a cabo el testeo detectar, con mayor facilidad, discrepancias entre la salida de una componente y la entidad requerida por otra.

Para poder realizar el mantenimiento de una aplicación hay que comprenderla. Como un buen diseño orientado a objetos utiliza encapsulamiento y ocultamiento de información, los patrones de comunicación en la aplicación son rígidos, y, por consiguiente, pueden comprenderse mejor. Es fácil determinar dónde pueden aparecer ramificaciones después de solucionar un problema.

De la misma forma que para el mantenimiento, para el refinamiento y la extensión de un producto de software, es necesario comprenderlo. Si éste fue diseñado con rigurosa consistencia, se podrán extender las interfases y agregar entidades. Los programadores

podrán agregar nuevas entidades que respondan a los viejos requerimientos. Así mismo, nuevas porciones del sistema pueden ser creadas conservando las mismas interfases, pero haciendo diferentes cosas con ellas. [WIR90]

Impacto en el ciclo de vida de un software al utilizar un diseño orientado a objetos en lugar de un diseño tradicional

Como se observa en el gráfico 16, la programación procedural pone énfasis, principalmente, en cómo implementar; y la mayor parte del tiempo de vida de un proyecto se ocupa en las etapas de diseño e implementación.

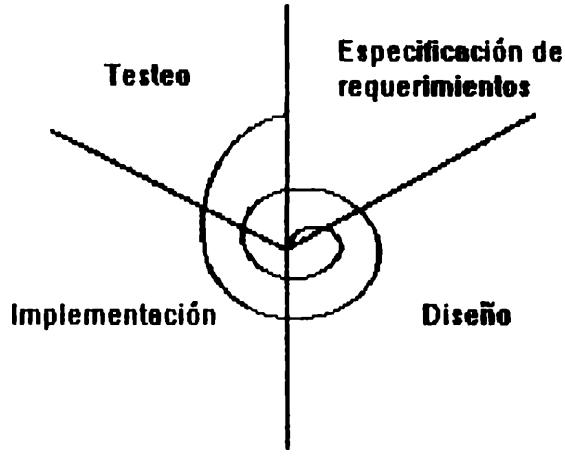

Gráfico 16: Ciclo de vida de un software tradicional

El diseño orientado a objetos tiene el propósito de conducir al desarrollo de un software robusto, que sea fácil de reutilizar, testear, refinar, mantener y extender.

Como se observa en el gráfico 17, la mayor parte del tiempo total del ciclo de vida de un proyecto de software orientado a objeto se emplea en la fase de diseño; una fracción más pequeña se utiliza en implementación y testeo.

La mayor parte del tiempo se emplea en la etapa de diseño. Esto es para que sea fácilmente reusable, mantenible y modificable. Las herramientas orientadas a objetos no garantizan, por sí solas, mantenimiento, reusabilidad y extensibilidad del software. Por el contrario, contribuyen a generar un proyecto en el cual los miembros

del grupo exploran el problema y realizan un diseño cuidadoso. Se requiere de un gran esfuerzo para diseñar código reusable, pero esto conduce a una mejor compresión del problema. De esta manera, la implementación consumirá menos tiempo.

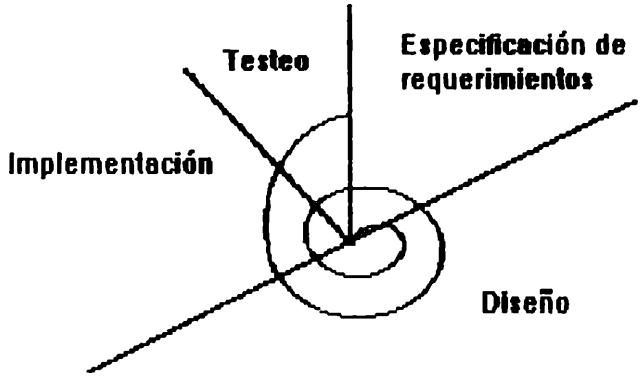

Gráfico 17: Ciclo de vida de un software orientado a objetos

Etapa de implementación

En esta etapa se traduce el diseño a una codificación entendible por una computadora. Empleamos un lenguaje orientado a objetos, Smalltalk/V Windows.

Desarrollaremos a continuación algunas consideraciones que se tuvieron en cuenta en esta etapa.

Como se pudo observar en las tarjetas de clases resultantes de la etapa de diseño, el comportamiento de la clase AplicaciónConGrafo-Versionable efectiviza el objetivo primordial del modelo que consiste en soportar y manipular **versiones de grafos**. Este comportamiento determina una implementación que focaliza el acceso al grafo versionable a través de una versión. En consecuencia, un conjunto ordenado de versiones resulta ser la elección más apropiada.

En su momento, cuando definimos el modelo (en el punto 2.1), optamos por mantener un conjunto de versiones (cada versión se construye en el momento en que se genera) y destacamos la posibilidad de mantener únicamente los cambios (overlays) y tener que construir

tuvimos que tomar los recaudos necesarios en la implementación para no repetir información. Cada versión, representa un grafo que, como el prototipo soporta versionamiento lineal, surge al aplicar un conjunto de modificaciones a la versión anterior del grafo. Las aristas y nodos que no sufrieron modificación alguna no se repiten en la nueva versión, sino que se copia la referencia el objeto. Este es el enfoque elegido teniendo en cuenta la NO repetición de información. Es decir que, conceptualmente, esta implementación puede verse como si existiera un *pool de nodos y aristas* que son compartidos por las distintas versiones.

Justificación de la utilización de un lenguaje orientado a objetos

Al haber empleado una técnica de diseño orientado a objetos, aunque no es necesario, codificamos en un lenguaje orientado a objetos. Estos son el soporte más adecuado para traducir un diseño orientado a objetos a una implementación que refleje naturalmente los principios del paradigma.

Justificación de la elección de Smalltalk/V Windows

Utilizamos como lenguaje de desarrollo el Smalltalk/V Windows de MicroSoft Corp.

La elección del lenguaje se debió a dos razones fundamentales:

- es un lenguaje orientado a objetos puro (captura las características de la Programación Orientada a Objetos)

- es un lenguaje adecuada para realizar rápidamente prototipos eficientes

La versión del Smalltalk/V elegida corre sobre Windows 3.0 que le brinda una interfase de ventanas amigable.

Etapa de testeo

Al haber realizado un diseño orientado a objetos, el testeo del producto de software nos resultó más sencillo. Pudimos testear por separado las entidades del sistema y fue más fácil localizar errores.

Capitulo 3: Utilizacion del modelo. Extensibilidad del prototipo

3- UTILIZACION DEL MODELO. EXTENSIBILIDAD DEL PROTOTIPO

Detallamos a continuación un análisis de las áreas de aplicación presentadas en los puntos anteriores como una extensión al prototipo general expuesto. Especializamos para cada una de las áreas de aplicación el prototipo general teniendo en cuenta las necesidades planteadas en los puntos 1.4.2, 1.4.3 y 1.4.4. Realizamos una descripción de las clases a adicionar al prototipo para proveer el comportamiento deseado.

En el punto 3.1 se plantea la utilización del prototipo para soportar la evolución del esquema de una BDOO. En particular, en el punto 3.2, esto se especializa para soportar la evolución del esquema de una BDOO extendido para objetos compuestos. En el punto 3.3 se describe el uso del prototipo para soportar versiones de un hipertexto.

Finalmente, se habrá mostrado que, gracias a la generalidad del modelo planteado y la flexibilidad del paradigma orientado a objetos, el prototipo es fácil y naturalmente extensible.

3.1- GRAFO VERSIONABLE COMO UN SOPORTE PARA LA EVOLUCION DEL ESQUEMA DE UNA BDOO

En el punto 1.4.2 mencionamos dos tipos de versionamiento en BDOO:
versionamiento del esquema y versionamiento de instancias. El modelo general es un soporte para el primer versionamiento mencionado.

Cada definición de un tipo se representa en el grafo como el contenido de uno de sus nodos. Las definiciones de tipos se organizan jerárquicamente por medio de la relación *es-un* en el grafo.

Se podría incrementar la funcionalidad de los SMBDOO, adicionándoles el manejo y control de la evolución del esquema. La forma de obtener nuevas versiones del esquema de la Base de Datos, sería a través de la alteración de la jerarquía de tipos (aristas y nodos en el DAG) y/o la definición de un tipo determinado (contenido

El control y acceso a las colecciones de instancias de cada tipo, en cualquier etapa de la evolución del mismo, es responsabilidad del SMBDOO utilizado.

El esquema de una Base de Datos se representa mediante un reticulado de tipos que es un grafo dirigido acíclico conectado con raíz (única). Tiene la particularidad de que cada nodo puede ser recuperado desde la raíz.

Lo mencionado anteriormente es una **invariante** del esquema; es decir, antes y después de aplicada cualquier operación de cambio del esquema, éste debe mantenerse en un estado consistente. Llamaremos a esta invariante, **invariante del reticulado de tipos.**

Para modelizar el esquema y soportar la evolución del mismo necesitamos definir las clases **Esquema**, **VersiónEsquema** y **EsquemaVersionable.**

Esquema es una AplicaciónConGrafo que refina el dominio de la variable de instancia *estructura* (heredada) de la clase GrafoDirigido a la clase **GrafoDirigidoSinCiclos**; esta variable de instancia es el soporte estructural para el esquema de una BDOO. La clase Esquema tiene el comportamiento que le da **semántica** a éste.

La clase **VersiónEsquema**, es subclase de Esquema y de VersiónAplicación. Hereda de la clase VersiónAplicación el *identificador* y redefine la variable de instancia *estructura* (heredada de ambas superclases) especializando el dominio a la clase **VersiónGrafoDirigidoSinCiclos.**

La clase **EsquemaVersionable** especializa la clase AplicaciónConGrafoVersionable restringiendo el dominio de los elementos de la variable de instancia *versiones* a una subclase de la clase VersiónAplicación, la clase **VersiónEsquema.**

El esquema se representa mediante un DAG, por lo tanto necesitamos refinar la clase GrafoDirigido a la clase **GrafoDirigidoSinCiclos**. Luego para poder modelizar una versión de un esquema necesitamos, a su vez, especializar las clases VersiónGrafoDirigido y **GrafoDirigidoSinCiclos**, obteniendo la clase **VersiónGrafoDirigidoSinCiclos**. (Gráfico 18)

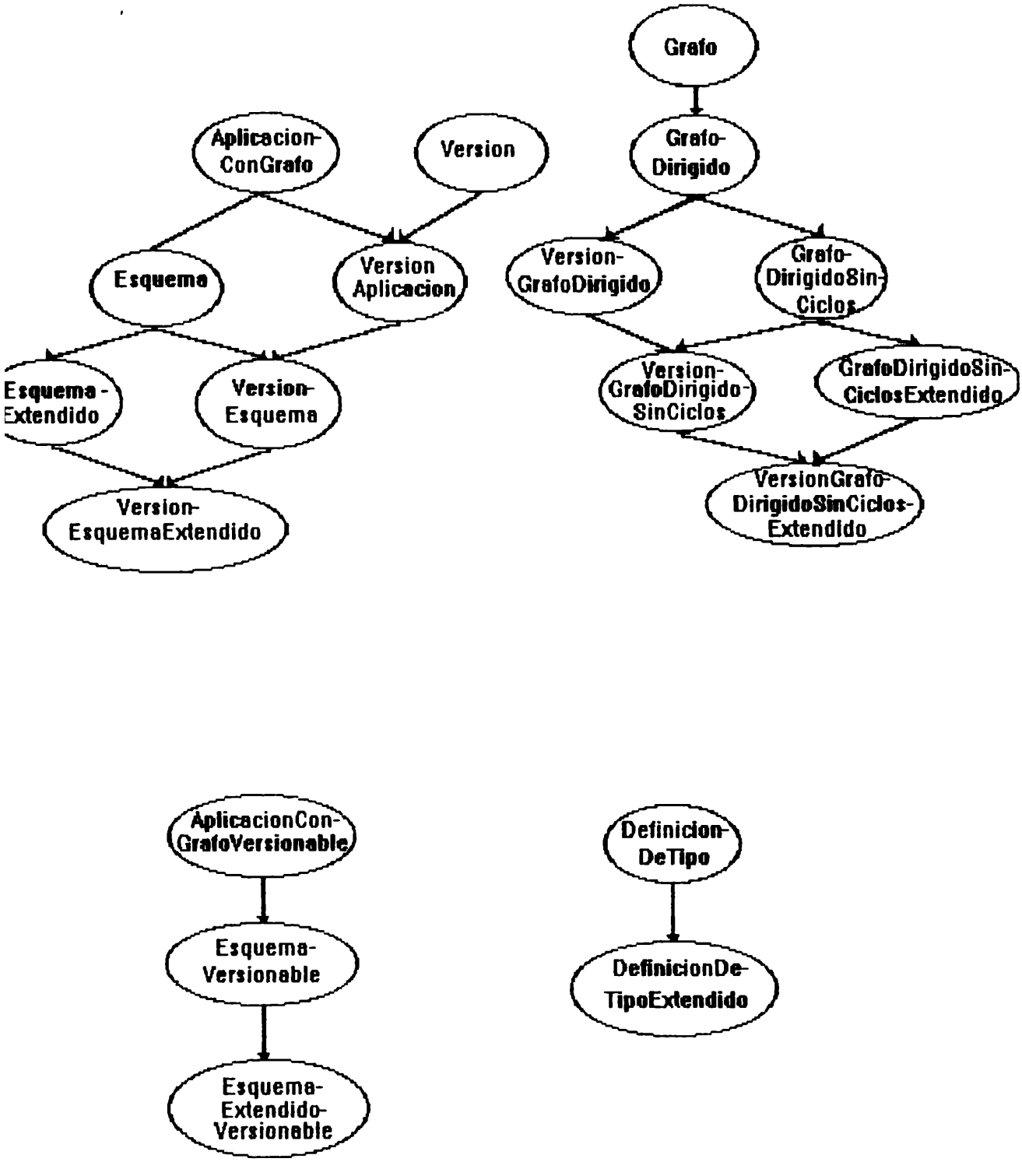

Gráfico 18: Extensión de la jerarquía de clases vista en el punto 2.3.2.2 para soportar evolución del esquema de una BDOO.

Nuestro prototipo define a la variable de instancia "contenido" perteneciente a la clase "NodoVersionable" como de clase "ClaseContenido". ClaseContenido es una forma de mencionar que el

contenido de los nodos de una versiónGrafoDirigido es de algún tipo, pero los grafos tratados son **generales** y, a través de las aplicaciones que lo utilicen, quedará determinado el tipo del contenido de los nodos.

En particular, en la aplicación que soporta evolución del esquema de una BDOO, el tipo de los nodos es la clase **DefiniciónDeTipo**.

La clase **DefiniciónDeTipo** deberá distinguir variables de instancia y definiciones de métodos, además del nombre del tipo. También deberá soportar operaciones tales como agregado, borrado, renombrado de variables de instancia y agregado, borrado y cambio de código de métodos, entre otros.

La clase **GrafoDirigidoSinCiclos** es una especialización de GrafoDirigido donde se refina la operación *insertar una arista* de forma tal que controle que no se forme un ciclo.

La clase **VersiónGrafoDirigidoSinCiclos** hereda las variables de instancia propias de VersiónGrafoDirigido, como así también, las heredadas de GrafoDirigido que por la Resolución de Conflictos elegida se heredan una sola vez. El comportamiento de VersiónGrafoDirigidoSinCiclos es todo aquel comportamiento heredado de VersiónGrafoDirigido, pero con la particularidad de que todos los métodos en los que se haga referencia a métodos de GrafoDirigido serán especializados, cuando sea necesario, a GrafoDirigidoSinCiclos.

La clase EsquemaVersionable es el soporte para la evolución del esquema de una BDOO. La clase Esquema, además de ser el soporte estructural de una versión del esquema (a través de la variable de instancia *estructura* con dominio en la clase GrafoDirigidoSinCiclos), debe proveer el comportamiento que le dé semántica al soporte estructural.

Detallaremos a continuación el comportamiento propio de un esquema, basado en la taxonomía descripta en el punto 1.4.2 y sus efectos sobre la estructura presentada:

(1) Cambios en el contenido de una definición de tipo (contenido de un nodo)

(1.1) Cambios a una variable de instancia tales como agregado, borrado, renombrado, cambios de dominio y en la herencia.

(1.2) Cambios a un método tales como agregado, borrado, renombrado, cambio del código y en la herencia.

Estos cambios se reflejan estructuralmente, en la versión del esquema con la que se está trabajando, generando una nueva versión de la definición del tipo. Esta es una responsabilidad propia del Esquema que la delega a GrafoDirigidoVersionableSinCiclos.

Se podrían presentar conflictos de nombres (ver Herencia Múltiple en el punto 2 del Apéndice A), que deben ser resueltos por el sistema.

Por otro lado, los cambios mencionados repercuten en los subtipos de la definición de tipo modificada y, en particular, el cambio del protocolo afecta, además, a toda la jerarquía de tipos. Los cambios deben ser, entonces, propagados o notificados. Esta acción es una responsabilidad propia del Esquema.

(2) Cambios en la relación *es-un* (arista)

(2.1) Ubicar a un tipo S como supertipo de un tipo C.

(2.2) Remover el tipo S de la lista de supertipos de un tipo C.

Estos cambios se reflejan estructuralmente agregando o borrando aristas en la versión del esquema con la que se está trabajando. Esta es una responsabilidad propia del Esquema que la delega a GrafoDirigidoVersionableSinCiclos.

Se podrían presentar conflictos de nombres que deben ser resueltos por el sistema.

Todos los cambios descriptos repercuten de la misma forma que en el punto (1), ya que agregar/eliminar un supertipo es equivalente a adicionar/borrar todas las variables de instancia y métodos de ese supertipo.

(3) Cambios en la definición de un tipo (nodo)

(3.1) Agregar un nuevo tipo C.

(3.2) Eliminar un tipo existente.

(3.3) Cambiar el nombre de un tipo existente.

Estos cambios se reflejan estructuralmente, agregando o borrando nodos, que a su vez contempla el agregado o borrado de aristas para mantener la **invariante del reticulado de tipos**, en la versión del esquema con la que se está trabajando. Esta es una responsabilidad propia del Esquema que la delega a GrafoDirigidoVersionableSinCiclos.

Si es tipo agregado/borrado tiene múltiples supertipos habrá que resolver posibles conflictos de nombres.

Al renombrar un tipo debe notificarse a toda la jerarquía.

Tanto en (1), (2) como en (3), el GrafoDirigido-VersionableSinCiclos mantiene la **invariante del reticulado de tipos.**

En todos los casos se podría considerar a las modificaciones como simples alteraciones al esquema o una evolución (VERSION) del mismo.

En el punto 1.4.2 presentamos dos criterios para atacar el tratamiento de los cambios en el esquema de una BDOO y su impacto en las instancias existentes:

Tamización: consiste en modificar el almacenamiento persistente, "filtrando" los valores de los atributos de las instancias a valores consistentes con la definición de tipo modificada, o corrigiendo dichos valores antes de ser usados.

Conversión: cambia todas las instancias del tipo a la nueva definición del mismo, asegurando que definiciones auxiliares (tales como métodos) también compatibilicen con ellas.

Si adicionamos nuestro soporte para el manejo de versiones de esquemas a un SMBDOO (encargado de la manipulación del conjunto de instancias asociado a cada versión del esquema) de los dos enfoques mencionados es natural utilizar tamización. De esta forma las

instancias de diferentes versiones podrían ser usadas por distintos programas.

Por otro lado, no tiene sentido la conversión debido a que se mantienen los diferentes esquemas de la BDOO.

3.2- GRAFO VERSIONABLE COMO UN SOPORTE PARA LA EVOLUCION DEL ESQUEMA DE UNA BDOO EXTENDIDO PARA OBJETOS COMPUESTOS

En el punto 1.4.3 mencionamos dos tipos de versionamiento en BDOO para soportar CAD: versionamiento del esquema y versionamiento de instancias. El modelo general es un soporte para el primer versionamiento.

De la misma manera que en BDOO, cada definición de un tipo se representa en el grafo como el contenido de uno de sus nodos. Las definiciones de tipos se organizan jerárquicamente por medio de la relación *es-un* en el grafo. La relación *es-parte-de* captura la noción de que un objeto es parte de otro objeto.

Definimos un objeto compuesto como un objeto con una jerarquía de objetos componentes (exclusivos o compartidos) y nos referimos a la jeraquía de tipos de los objetos componentes como una **jerarquía de partes** o **jerarquía de agregación.**

Para poder manipular la complejidad de los objetos compuestos es necesario soportar su jerarquía de partes asociada.

Se podría incrementar la funcionalidad de los SMBDOO, adicionándoles el manejo y control de la evolución del esquema extendido. La forma de obtener nuevas versiones del esquema extendido de la Base de Datos, sería a través de la alteración de la jerarquía de tipos (aristas y nodos en el DAG) y/o definición de un tipo determinado (contenido de un nodo).

Para modelizar el esquema extendido para objetos compuestos y soportar su evolución necesitamos definir las clases **EsquemaExtendido, VersióEsquemaExtendido** y **EsquemaExtendido-Versionable.** (Gráfico 18)

La clase **EsquemaExtendido** es subclase de Esquema y redefine la variable de instancia *estructura*, especializando su dominio de la clase GrafoDirigidoSinCiclos a la clase **GrafoDirigido-SinCiclosExtendido.**

Especializamos la clase GrafoDirigidoSinCiclos a **GrafoDirigido-SinCiclosExtendido** para soportar un conjunto adicional cuyos elementos son, a su vez, un conjunto de aristas dirigidas.

Un GrafoDirigidoSinCiclosExtendido puede verse como un **grafo dirigido arbitrario**, pero, debido al comportamiento heredado de GrafoDirigidoSinCiclos, **se distingue** siempre un **DAG** (el heredado de las superclases). Cada uno de los conjuntos de aristas dirigidas se interpreta, junto con los nodos que relaciona, como si modelizara un DAG conectado. La clase GrafoDirigidoSinCiclosExtendido hereda las propiedades de la clase GrafoDirigidoSinCiclos que modelizan un DAG y adiciona relaciones para, a partir de los nodos existentes, construir múltiples DAGs. Por lo tanto, el comportamiento específico de la clase deberá contemplar que, el agregado o borrado de una arista de un conjunto del conjunto adicional, no introduzca un ciclo ni lo deje desconectado.

A pesar de ser un GrafoDirigidoSinCiclosExtendido vemos que el comportamiento de la clase abstrae en todo momento, por un lado un DAG principal y por otro lado un conjunto de DAGs. Estos últimos relacionan nodos del DAG principal.

Para modelizar una versión de un esquemaExtendido necesitamos especializar las clases VersiónGrafoDirigidoSinCiclos y **GrafoDirigidoSinCiclosExtendido** a la clase **VersiónGrafoDirigido-SinCiclosExtendida.**

La clase **VersiónGrafoDirigidoSinCiclosExtendida** hereda de VersiónGrafoDirigidoSinCiclos las variables de instancia que soportan la jerarquía de tipos.

Una versiónGrafoDirigigoSinCiclosExtendida está soportada por un grafo (unión de los nodos, las aristas y el conjunto cuyos elementos son conjuntos de aristas) donde se pueden abstraer múltiples visiones. En la aplicación de un Esquema Extendido tienen la particularidad de que estas visiones no comparten aristas entre sí.

El conjunto de aristas (heredado de VersiónGrafo-DirigidoSinCiclos) establece la relación *es-un* entre deficiones de tipos extendidas formando la jerarquía de tipos. Por otro lado, el conjunto cuyos elementos son conjuntos de aristas dirigidas (heredado de GrafoSinCiclosExtendido) establece la relación *es-parte-de* entre definiciones de tipos extendidas formando las distintas jerarquías de agregación asociadas a las definiciones de tipo compuesto.

Como mencionamos anteriormente en el punto 3.1, el contenido de los nodos de una versiónGrafoDirigidoSinCiclosExtendido es de la clase **DefiniciónDeTipoExtendida.**

La clase **DefiniciónDeTipoExtendida** es una subclase de la clase DefiniciónDeTipo, y la refina extendiendo la declaración de las variables de instancia a indicar si tienen propiedad de referencia compuesta. En dicho caso se deberá mencionar si se trata de una referencia compuesta exclusiva independiente, exclusiva dependiente, compartida independiente o compartida dependiente. Además se deberá determinar si cada atributo es intrínseco o no-intrínseco.

La clase **VersiónEsquemaExtendido** es subclase de VersiónEsquema y de **EsquemaExtendido.** Hereda de la clase VersiónEsquema el *identificador* de la versión y redefine la variable de instancia *estructura* (heredada de ambas superclases), especializando el dominio a la clase **VersiónGrafoDirigidoSinCiclosExtendido.**

La clase **EsquemaExtendidoVersionable** especializa la clase EsquemaVersionable, restringiendo el dominio de los elementos de la variable de instancia *versiones* a una subclase de la clase VersiónEsquema que es **VersiónEsquemaExtendido.**

De forma similar que en el punto anterior, la clase Esquema-ExtendidoVersionable es el soporte para la evolución del esquema de una BDOO que maneje objetos compuestos. La clase EsquemaExtendido, además de ser el soporte estructural de una versión del esquema (a través de la variable de instancia *estructura* con dominio en la clase GrafoDirigidoSinCiclosExtendido), debe proveer el comportamiento que le dé semántica a éste.

Detallaremos a continuación el comportamiento propio de un esquema extendido, basado en la taxonomía descripta en el punto 1.4.3

(que es una extensión de la taxonomía del punto 1.4.2) y sus efectos sobre la estructura presentada:

(1.1.1)-(1.1.2) Agregar/Eliminar una variable de instancia con la propiedad de referencia compuesta

Este cambio se refleja estructuralmente en la versión del esquema extendido con la que se está trabajando, generando una nueva versión de la definición de tipo extendido. Por otro lado, involucra el agregado/borrado de una jerarquía de partes si el tipo no tenía ninguna variable de instancia que tenga una referencia compuesta o, el agregado/borrado de una subjerarquía de partes a la jerarquía ya existente, si el tipo tenía al menos una variable de instancia con referencia compuesta. Esta es una responsabilidad propia del EsquemaExtendido que la delega a GrafoDirigidoVersionableSinCiclosExtendido.

Se presentan dos casos, según el tipo de referencia compuesta a agregar:

a) Si se agrega una variable de instancia con la propiedad de referencia compuesta compartida a un tipo T (dependiente o independiente) se debe verificar que no haya ninguna referencia compuesta exclusiva en otra definición de tipo que haga referencia a T.

b) Si se agrega una variable de instancia con la propiedad de referencia compuesta exclusiva (dependiente o independiente) a un tipo T, se debe verificar que no exista ninguna referencia compuesta de cualquier clase, en otra definición de tipo, a T.

Realizar estas verificaciones es una responsabilidad propia del EsquemaExtendido.

(1.1.6) Cambiar la propiedad de referencia de una variable de instancia con dominio de tipo B.

Con referencia a la taxonomía descripta en el punto 1.4.3, podemos agrupar en dos categorías de cambios:

a) Cambios que **afectan** a la jerarquía de partes: (1.1.6.1), (1.1.6.2) **y** (1.1.6.3)

b) Cambios que **no afectan** a la jerarquia de partes: (1.1.6.4) a (1.1.6.7)

Se reflejan estructuralmente en la versión del esquema extendido con la que se está trabajando, generando una nueva versión de la definición de tipo extendido. Esta es una responsabilidad propia del EsquemaExtendido que la delega a GrafoDirigidoVersionableSinCiclosExtendido.

Es responsabilidad del EsquemaExtendido realizar las verificaciones descriptas en el punto 1.4.3.

(2.2) Eliminar el tipo S de la lista de supertipos de un tipo C.

Eliminar el tipo S causa el mismo efecto que eliminar todas sus variables de instancias y métodos. En el caso de que el tipo S tenga alguna variable de instancia con la propiedad de referencia compuesta, se debe aplicar el punto (1.1.2).

(3.1) Agregar una nueva definición de tipo.

Si la nueva definición de tipo agregada tiene alguna variable de instancia con la propiedad de referencia compuesta, se debe aplicar lo expresado en el punto (1.1.1).

(3.2) Eliminar una definición de tipo existente.

Este punto afecta la jerarquía de agregación del tipo que se elimina de la misma forma que la descripta en el punto (2.2).

Tanto en (1), (2) como en (3), se podría considerar a las modificaciones como simples alteraciones del esquema extendido o una evolución (VERSION) del mismo.

3.3- GRAFO VERSIONABLE COMO UN SOPORTE PARA LA EVOLUCION EN EL TIEMPO DE UN HIPERTEXTO.

En el punto 1.4.4 describimos dos tipos de versionamiento en hipertexto: versiones de hipertexto (configuraciones) que varien en el tiempo y versiones de hipertexto en un ambiente de múltiples

usuarios o aplicaciones. El modelo soporta la primer manera de versionamiento. Por otro lado, el prototipo, que implementa versionamiento lineal, permite representar versionamiento de hipertexto que varíen en el tiempo.

Un hipertexto es una red donde los nodos representan conceptos y los links relaciones entre ellos. Los links vinculan los conceptos por medio de la relación *esta-relacionado-con*. Si los nodos y links de un documentos son mapeados a nodos y aristas de un grafo abstracto, entonces un documento puede mapear en un **grafo arbitrario** llamado **hipergrafo**. [DEL87]

Se podría incrementar la funcionalidad de los sistemas de hipertexto, adicionándoles el manejo y control de la evolución de un hipertexto. La forma de obtener nuevas versiones de un hipertexto, sería a través de **grupos de cambios** en un documento. Los cambios posibles afectan a las componentes del mismo.

A los efectos de mostrar con un ejemplo que la idea general puede ser extendida para el área de hipertexto, detallaremos en el punto 4 la especialización del prototipo para soportar versiones de hipertexto.

Como originalmente fue nuestra motivación, el prototipo desarrollado refleja el modelo general de grafos versionables que son un soporte estructural para las versiones de hipertexto.

Capitulo 4: Una aplicacion: extension del prototipo

4.1 - Desarrollo del prototipo extendido

4.1.1 - Ciclo de vida

4- UNA APLICACION: EXTENSION DEL PROTOTIPO PARA SOPORTAR VERSIONES DE HIPERTEXTOS

4.1- DESARROLLO DEL PROTOTIPO EXTENDIDO

4.1.1- CICLO DE VIDA

Etapa de análisis de requerimientos

ESPECIFICACION HIPERTEXTO CON VERSIONES

Realizamos un **sistema de hipertexto** que soporte VERSIONES, con las siguientes características (que ya vimos en el punto 1.3).

Las componentes que manipulamos son nodos y links.

Los *nodos* tienen texto como contenido.

Los *links* son direccionales ya que son cognitivamente más valiosos para la navegación. El *destino de un link* es otro nodo como una entidad atómica.

Como una ayuda para la navegación, proveemos una forma primitiva de orientación en el hipertexto, que muestra al mismo como un grafo. El browser ofrece una medida importante de indicaciones contextuales sobre los nodos que se están visualizando y las relaciones entre sí y sus vecinos en el grafo.

Utilizamos ventanas como una ayuda para la visualización y proveemos las operaciones usuales tales como: cerrar una ventana, modificar su tamaño y moverla de lugar.

Proveemos dos modos de operación: browse y autor a través de las distintas operaciones.

Las operaciones usuales de hipertexto son:

- crear un hipertexto
- crear un enlace (link) de referencia
- crear un enlace de comentario

- eliminar un enlace de comentario
- eliminar un enlace de referencia
- eliminar un nodo
- navegar por el hipertexto
- modificar el contenido (texto) de un nodo
- deshacer un hipertexto

Modelizamos la evolución de un hipertexto manteniendo un conjunto de versiones del mismo.

Obtenemos una nueva versión del hipertexto a partir de un conjunto de modificaciones aplicadas a la última versión del mismo (versionamiento lineal). Una versión de un hipertexto es, a su vez, un hipertexto y se identifica con un lable de versión.

Las modificaciones que pueden generar una nueva versión del hipertexto son:

- crear un hipertexto
- crear un enlace de referencia
- crear un enlace de comentario
- eliminar un enlace de comentario
- eliminar un enlace de referencia
- eliminar un nodo
- modificar el texto de un nodo

Esta última operación genera un nuevo nodo con el contenido modificado y quedará vinculado a través de una relación *sucesor-de* con el nodo que se está modificando, pasando a formar parte de su **historia**. La historia de un nodo representa la evolución del mismo, obtenido a partir del nodo original y reflejando las sucesivas modificaciones.

Considerando que el usuario podra realizar una modificación errónea, le posibilitamos corregirla sin generar una nueva versión, evitando así la proliferación de versiones sin sentido. De igual manera, permitimos deshacer la última versión del hipertexto si el usuario descubre que ésta no le sirve y sí la anterior.

Manejamos los siguientes recorridos a través del hipertexto versionable:

1)Navegación dentro de la versión i (V_i) con la posibilidad de recorrer la historia de un nodo.

A partir de un nodo N_i y una versión V_i el usuario recorre cada nodo eligiendo en cada uno cuál será el nodo adyacente que seguirá en su navegación o eligiendo recorrer su historia, o bien eligiendo un nodo que muestre un comentario sobre el texto del nodo.

En los dos últimos casos retorna al nodo que pertenece a la versión por la que estaba navegando para continuar con el recorrido.

2)Navegación a través de la historia de un nodo.

A partir de un nodo N_i dado se recorre la historia de ese nodo (comenzando por el nodo original), pudiéndose ver las versiones previas y las posteriores al nodo N_i.

Desarrollamos un hipertexto versionable con las características elementales de un hipertexto, a los efectos de mostrar que nuestro modelo de versionamiento es implementable.

Etapa de diseño

En esta etapa refinamos el diseño realizado que describimos en el punto 2.3.2.

Como resultado se obtuvieron las tarjetas de clases específicas del hipertexto versionable, que se pueden consultar en el Apéndice C. De las mismas se desprendió la jerarquía de clases detallada en el gráfico 19.

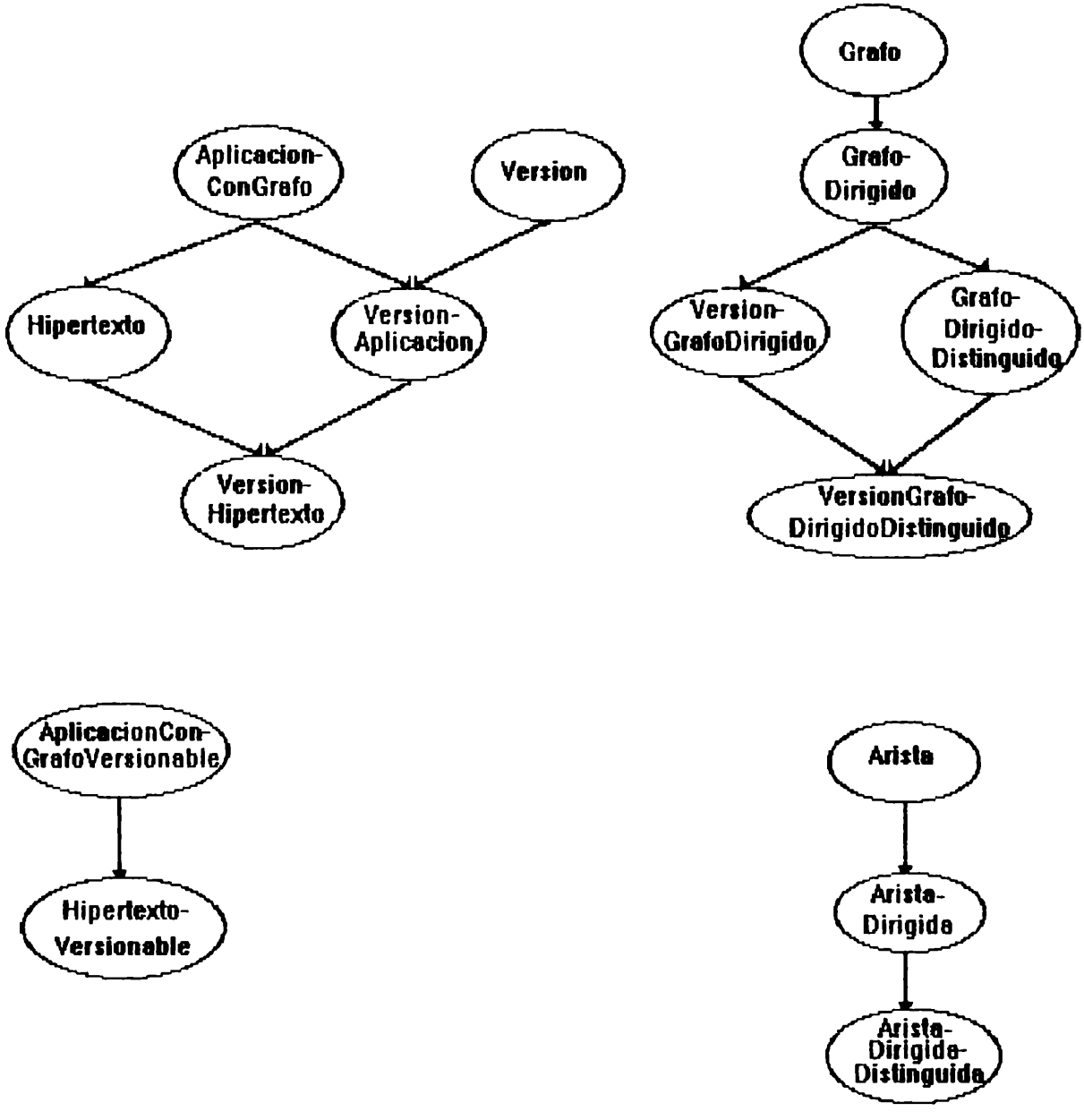

Gráfico 19: Jerarquía de clases del prototipo para hipertextos versionables

Etapa de implementación

Desarrollaremos a continuación algunas consideraciones que se tuvieron en cuenta en esta etapa.

Debido a la restricción del lenguaje Smalltalk /V Windows que soporta sólo herencia simple, modificamos la jerarquía múltiple obtenida en la etapa de diseño. Esto se llevó a cabo agregando variables de instancia de las superclases de las cuales fue anulada la herencia y adicionando ciertos métodos en las subclases que dejaron de heredarlos.

Además de las clases resultantes del diseño propias del software de base, se anexó un conjunto de clases específicas de la interfase.

Por una cuestión de simplicidad en el algoritmo de visualización de un grafo, graficamos sólo aquellos grafos que contienen como máximo 8 nodos.

Etapa de testeo

Al haber realizado un diseño orientado a objetos, el testeo del producto de software nos resultó más sencillo. Pudimos testear por separado las entidades del sistema y fue más fácil localizar errores.

CONCLUSION

Desarrollamos un modelo concreto para manejar versiones de grafos.

Aunque nuestro proyecto fue originalmente motivado por la necesidad de mantener versiones en BDOO, las similitudes halladas en el manejo de versiones en otras áreas (tales como hipertexto) nos condujeron a plantear un enfoque más general. Esto hace posible su utilización en diversas aplicaciones que aquí no fueron tratadas, como por ejemplo, ambientes de desarrollo de software.

Como pudo observarse, para proveer la capacidad de versionar en cada una de las aplicaciones tratadas, sólo fue necesario especializar algunas de las clases definidas. Un procedimiento similar se puede llevar a cabo para soportar cualquier aplicación que requiera manejo de versiones. Esto se debe a la generalidad del modelo planteado y la flexibilidad del paradigma orientado a objetos.

Apendice A

1 - Grafos

2 - Programacion Orientada a Objetos

1— GRAFOS

En problemas originados en diversas áreas tales como Ciencias de la Computación, Matemáticas, Ingeniería, etc., surge frecuentemente, la necesidad de representar relaciones entre objetos. Los grafos (dirigidos y no dirigidos) constituyen modelos naturales para tales representaciones.

Un *grafo dirigido* (digrafo) G consiste de un conjunto de vértices V y un conjunto de arcos E. Los vértices son llamados también *nodos* o *puntos*; los arcos pueden ser llamados *aristas dirigidas* o *líneas dirigidas*. Un arco es un par ordenado de vértices (v,w); v es llamado *cola* y w la *cabeza* del arco. El arco (v,w) es dibujado de la siguiente manera:

$$v \longrightarrow w$$

y se dice que el arco (v,w) parte de v y llega a w, y que w es *adyacente* a v.

Los vértices de un digrafo pueden ser usados para representar objetos, y los arcos relaciones entre ellos. Por ejemplo, en la representación del flujo de control en un programa de computación, los vértices podrían representar bloques básicos y los arcos, posibles transferencias de flujos de control.

Un *camino* en un digrafo es una secuencia de vértices $v_1, v_2, \ldots, v_n$ tal que (v_1,v_2), (v_2,v_3), ... ,(v_{n-1},v_n) son arcos. Este camino se origina en el vértice v_1 y pasa a través de los vértices $v_2, \ldots, v_{n-1}$ y finaliza en el vértice v_n. La *longitud* del camino es el número de arcos sobre el camino.

Un camino es *simple* si todos los vértices sobre el camino, excepto posiblemente el primero y el último, son distintos. Un *ciclo simple* es un camino simple de longitud al menos uno que comienza y finaliza en el mismo vértice.

Un *grafo acíclico dirigido* (DAG) es un grafo dirigido sin ciclos. [AHO83]

Un *árbol* es una colección de elementos llamados nodos; un de ellos es un elemento distinguido, la *raiz* del árbol.

Existe una relación de parentesco entre los nodos que lleva a una jerarquía. Un nodo puede ser de cualquier tipo.

Formalmente, un árbol puede definirse recursivamente de la siguiente manera:

1) Un nodo simple es un árbol. Este nodo es también la raiz del árbol.

2) Supongamos que n es un nodo y T_1, T_2, ..., T_k son árboles con raices n_1, n_2, ..., n_k, respectivamente. Podemos construir un nuevo árbol, haciendo que n sea el padre de los nodos n_1, n_2, ..., n_k. En este árbol, n es la raiz y T_1,T_2,...,T_k son los subárboles de la raiz. Los nodos n_1,n_2,...,n_k son llamados hijos del nodo n.

Un árbol es un grafo acíclico dirigido (DAG), con la particularidad de que cada nodo del árbol tiene un único nodo predecesor (padre).

2- PROGRAMACION ORIENTADA A OBJETOS

Describiremos las ideas de Programación Orientada a Objetos, a partir de un modelo computacional y una filosofía de desenvolvimiento, extraídas de [DIA90].

Modelo computacional

Un programa es una colección de objetos (y sólo objetos) que trabajan y cooperan entre sí enviándose mensajes.

Los **objetos** son entidades que poseen estado interno y comportamiento. El estado interno es inaccesible a otros objetos y el comportamiento está determinado por el conjunto de mensajes que el objeto puede recibir.

Todo el procesamiento en este modelo es activado por **mensajes** entre objetos. Un objeto determinado, al recibir un mensaje realizará un conjunto de acciones, modificando su estado interno, enviando mensajes a otros objetos, etc.

Algunas características del modelo:

* Todas las acciones resultan del envío de mensajes entre objetos.

* El envío de mensajes es una forma indirecta de invocación de procedimientos. Los objetos responden a los mensajes ejecutando procedimientos propios, llamados **métodos**.

* Este modelo soporta "information hiding", pues el estado interno de cada objeto es inaccesible a otros y "abstracción de datos" pues la única forma de acceder a un objeto es mediante un conjunto (limitado) de operaciones (su interfase). Obviamente, esto tiene un impacto inmediato en la calidad del software diseñado, alentándose la modularidad, localizándose el efecto de los cambios y proveyéndose un esquema fácilmente extensible.

Filosofía de desarrollo

Clasificación y generalización / especialización, como herramientas para extraer propiedades de objetos en clases y de clases en jerarquías de clases, son los ejes organizacionales alrededor de los cuales diseñamos software cuyo modelo computacional sea como el descripto.

* En general es posible agrupar los objetos (de un dominio de aplicación) en clases (o tipos). Así, cada objeto pertenece a una (y en principio, sólo una) clase, y "recibe" de ella su funcionalidad.

* Una clase contiene un "molde" de las características que contienen los objetos de la misma, así como los mensajes que pueden recibir y la manera que responden a ellos.

Hasta aquí, trabajar con objetos es semejante a trabajar con tipos de datos abstractos. La diferencia fundamental la da el segundo eje organizacional para realizar abstracciones: generalización / especialización y junto con él, el concepto de **herencia.**

* El segundo nivel de abstracción consiste en agrupar las clases en jerarquías de clases (definiendo sub y super clases), de forma tal que una clase A hereda todas las propiedades de su(s) superclase(s) (suponiendo que tiene al menos una). En el caso en que una subclase tenga más de una superclase estamos en presencia de **herencia múltiple.** Si una clase hereda una operación o variable de instancia con el mismo nombre de más de un ancestro, surge un conflicto de nombres. En consecuencia, los lenguajes orientados a objetos que permitan la utilización de herencia múltiple, deberán proveer un mecanismo de **Resolución de conflictos** para resolver conflictos de nombres cuando se produzcan.

Polimorfismo y binding dinámico

Definimos **polimorfismo** como la capacidad que tienen objetos de distintas clases de responder a mensajes con el mismo nombre. (Esta característica se conoce con el nombre de "overloading" en lenguajes como ADA).

El concepto de polimorfismo es en realidad mucho más amplio (vemos aquí una forma simplificada de polimorfismo), puesto que involucra, por ejemplo, funciones cuyo comportamiento depende del tipo de los parámetros y argumentos que reciben.

Más generalmente, el significado de:

unObjeto mensaje, depende de cual es la clase a la que pertenece unObjeto.

Genericidad, puede combinarse con el concepto de herencia para generar software más reusable. Un nivel adicional de reusabilidad (al poderse diseñar código genérico) está dado por la posibilidad de realizar tipeo débil o, más precisamente, vincular a los objetos con los mensajes recién en tiempo de ejecución: esta idea se conoce como **binding dinámico**.

Se conoce como **binding**, el proceso mediante el cual se asocian operadores a operandos.

En los lenguajes orientados a objetos, la responsabilidad de elegir la operación que debe llevarse a cabo para responder a un mensaje la tiene el objeto que lo recibe y esa decisión se toma recién durante la ejecución del programa. Si a este hecho le sumamos que las variables en este tipo de lenguajes no son tipadas (o sea, no existe restricción respecto al tipo de objeto al que se refieren), tenemos que en el siguiente caso:

x imprimir, el significado de imprimir dependerá de la clase a la que pertenezca *x* la cual se determinará en forma dinámica durante la ejecución.

Decimos que en este caso el binding es **dinámico**.

La posibilidad de asociar durante ejecución, y en forma dinámica, a los objetos con los mensajes provee (en combinación con el polimorfismo, y en ciertos casos con la herencia) una herramienta para lograr código altamente reusable.

Apendice B

1 - Tecnica de diseno Orientado a Objetos

1- TECNICA DE DISEÑO ORIENTADO A OBJETOS

DISEÑO BASADO EN RESPONSABILIDADES

Como resultado de emplear esta metodología de diseño orientado a objetos se obtienen:

- Una tarjeta para cada clase del sistema
- Una tarjeta para cada subsistema
- Uno o más grafos de jerarquía mostrando los patrones de herencia en la aplicación
- Uno o más grafos de colaboraciones mostrando todos los caminos posibles de comunicación en la aplicación

TARJETA DE CLASE:

En una tarjeta de clase se detallan:

* Nombre de la clase
* Si es una clase concreta o abstracta
* Lista de superclases
* Lista de subclases
* Objetivo de la clase
* Lista de contratos heredados, indicando de que clase se heredan
* Lista de contratos para los cuales la clase actúa como servidor, indicando para cada contrato, la lista de responsabilidades de la clase que lo soportan, y para cada responsabilidad, las firmas de los métodos que implementan la responsabilidad. Para cada método indicar las colaboraciones requeridas por éste y el número de contrato.
* Lista de responsabilidades privadas de la clase. Para cada responsabilidad se indica la misma información que para las responsabilidades que soportan un contrato.

Las *responsabilidades* de un objeto incluyen el conocimiento que mantiene el objeto y las acciones que éste puede realizar. Pueden ser privadas o estar agrupadas en un contrato.

Responsabilidades privadas de la clase: Son aquellas responsabilidades que representan un comportamiento propio de la clase, pero que no es requerido por otras clases.

Un *contrato* define un conjunto de responsabilidades que cohesionan, de las cuales dependen uno o más clientes.

Los *servidores* de cada contrato, son las clases en cuyas tarjetas aparecen los contratos. Mientras que los *clientes* de cada contrato, son las clases que requieren de estas responsabilidades para poder concretar sus propias responsabilidades.

Un contrato debe ser definido exactamente por una clase e implementado por todas sus subclases.

Las *colaboraciones* representan solicitudes de un cliente a un servidor para poder cumplir con la responsabilidad del cliente.

Para cumplir con una responsabilidad, tal vez se necesiten muchas colaboraciones o ninguna.

Para determinar colaboraciones hay que analizar las interacciones entre clases (examinar las responsabilidades que son dependientes). Por ejemplo, si una clase es responsable de una acción pero no posee todo el conocimiento necesario para completar esta acción, debe colaborar con una o más clases que posean este conocimiento.

La *firma de un método* es el nombre del método, los tipos de sus parámetros y el tipo del objeto que retorna el método. Es una especificación formal de la entrada y salida de un método, lo que se requiere para usarlo.

TARJETA DE SUBSISTEMA:

En una tarjeta de subsistema se detallan:

* Nombre del subsistema
* Lista de clases y subsistemas que lo componen
* Propósito del subsistema
* Lista de contratos para los cuales el subsistema es

servidor, identificando, para cada contrato, la clase o subsistema en el cual el contrato es delegado

Un *subsistema* es un grupo de clases, o un grupo de clases y otros subsistemas, que colaboran entre sí para soportar un conjunto de contratos.

Desde afuera del subsistema, el grupo de clases y/o subsistemas puede verse como trabajando en conjunto para proveer una funcionalidad claramente definida.

Desde dentro del subsistema, éste tiene una estructura compleja, consiste de clases y subsistemas que colaboran entre sí para soportar distintos contratos que contribuyen al comportamiento global del sistema.

Los subsistemas son entidades conceptuales, no existen durante la ejecución, consecuentemente no pueden cumplir con los contratos. Los subsistemas *delegan* cada contrato a una clase del subsistema que soporte el contrato.

GRAFO DE JERARQUIA:

Es una herramienta para representar relaciones jerárquicas entre clases relacionadas.

GRAFO DE COLABORACIONES:

Es una herramienta que, mostrando en forma gráfica la colaboración entre clases y subsistemas, contribuye a simplificar el diseño.

Apendice C

1 - Tarjetas de clases del prototipo obtenidas al aplicar la tecnica de diseno O. O. basado en responsabilidades

2 - Tarjetas de clases del prototipo extendido para soportar hiper-textos versionables obtenidas al aplicar la tecnica de diseno O. O. basado en responsabilidades

1- TARJETAS DE CLASES DEL PROTOTIPO OBTENIDAS AL APLICAR LA TECNICA DE DISEÑO O. O. BASADO EN RESPONSABILIDADES:

Clase Concreta : APLICACIONCONGRAFOVERSIONABLE Superclases : Object Subclases : HipertextoVersionable	
Objetivo : Mantener la evolución que sufrió una aplicación con grafos (registrando cada una de las versiones del grafo original).	
Responsabilidades Privadas	Clases Colaboradoras
* Mantener conocimiento sobre un conjunto ordenado de versiones de un grafo dirigido.	OrderedCollection
* Devolver la versión correspondiente a un label de versión dado	OrderedCollection VersionAplicacion
* Devolver el label de la última versión	OrderedCollection VersionAplicacion
* Devolver los labels de todos los nodos de las versiones	VersionAplicacion
* Devolver labels de todas las versiones	OrderedCollection VersionAplicacion
* Desplegar en pantalla una versión	OrderedCollection VersionAplicacion
* Devuelve la versión donde se creó el nodo con el label dado.	OrderedCollection VersionAplicacion
Responsabilidades Públicas	
* Crear e inicializar versión original.	OrderedCollection VersionAplicacion
* Generar una nueva versión a partir de la última versión del grafo realizando modificaciones, inserciones, borrado, etc., y pedir el label.	OrderedCollection VersionAplicacion
* Modificar la última versión del grafo	OrderedCollection VersionAplicacion
* Deshacer la última versión	OrderedCollection VersionAplicacion
* Navegar a través de una versión a partir de un nodo que se ingresa (label) del grafo con la posibilidad de recorrer la historia de un nodo	VersionAplicacion

Continuación ...

* Navegar a través de la historia de un nodo (a partir de algún nodo original)	OrderedCollection VersionAplicacion

Clase Concreta : **APLICACIONCONGRAFO** Superclases : Object Subclases : Hipertexto, VersionAplicacion	
Objetivo : Clase conceptual que abstrae el comportamiento común a las distintas aplicaciones soportadas por un grafo.	
Responsabilidades Privadas	Clases Colaboradoras
* Mantener conocimiento sobre el soporte estructural de la aplicación.	GrafoDirigido
Responsabilidades Públicas	
* Modificar contenido de un nodo	GrafoDirigido
* Crear e inicializar con primer nodo	GrafoDirigido
* Inicializar con primer nodo	GrafoDirigido
* Eliminar un nodo	GrafoDirigido
* Agregar enlace	GrafoDirigido
* Eliminar enlace	GrafoDirigido
* Desplegarse en pantalla	GrafoDirigido
* Navegar sobre la aplicación	GrafoDirigido
Contratos	
1 AG1	
* Decir si existe un nodo en la aplicación	GrafoDirigido
* Devolver los labels de los nodos incluidos en la aplicación	GrafoDirigido
* Devolver el nodo con el label dado (si es que existe)	GrafoDirigido
* Deshacerse	GrafoDirigido
Clase Cliente: - AplicacionConGrafoVersionable	

Clase Concreta : **VERSIONAPLICACION** Superclases : Version, AplicacionConGrafo Subclases : VersionHipertexto	
Objetivo : Modelizar una versión de cualquier aplicación soportada con un grafo dirigido.	
Contratos 1 V1 Se hereda de la clase Version 2 AG1 Se hereda de la clase AplicacionConGrafo 3 VA1 * Crear e inicializar con primer nodo * Devolver una copia del objeto receptor * Navegar a través de la versión receptora con la posibilidad de recorrer la historia de un nodo Clase Cliente: - AplicacionConGrafoVersionable	VersionGrafoDirigido VersionGrafoDirigido VersionGrafoDirigido

Clase Concreta : **GRAFO** Superclases : Object Subclases : GrafoDirigido	
Objetivo : Soportar a través de una estructura de datos información relacionada modelizada con un grafo.	
Responsabilidades Privadas	Clases Colaboradoras
* Mantener el conocimiento sobre el conjunto de nodos	Dictionary
* Mantener el conocimiento sobre el conjunto de aristas	Set
* Buscar una arista en el conjunto y retornarla si es que está incluída	Arista Set
* Decir qué nodos del grafo son adyacentes a un nodo dado	Arista Set
* Insertar arista	Set
* Sacar referencia al nodo del conjunto	Dictionary
* Decir si el grafo contiene a un nodo dado	Dictionary - Nodo
* Crear e inicializar	Dictionary - Set
* Dibujarse	Dictionary - Set
* Eliminar arista si es que existe	Arista
* Eliminar nodo	Nodo
Contratos	
1 G1 * Inicializar con primer nodo * Decir si existe un nodo con el label * Insertar nodo * Modificar contenido de un nodo Clase Cliente: Hipertexto	 Nodo Dictionary Dictionary Nodo
2 G2 * Crear e inicializar con primer nodo * Deshacerse * Decir si existe un nodo con el label * Devolver los labels de nodos incluídos en el grafo * Modificar contenido de un nodo * Dado el label de nodo devolver el nodo asociado	 Dictionary - Set Nodo Dictionary - Set Dictionary Dictionary Nodo Dictionary

Continuación ...

Clase Cliente: AplicacionConGrafo 3 G3 * Devolver el contenido de un nodo con el label dado Clase Cliente: VersionHipertexto	Nodo Dictionary

Clase Concreta : **GRAFODIRIGIDO** Superclases : Grafo Subclases : VersiónGrafoDirigido y GrafoDirigidoDistinguido	
Objetivo : Especializar la estructura de datos grafo a grafos cuyas aristas son dirigidas.	
Responsabilidades Privadas	Clases Colaboradoras
* Mantener conocimiento sobre el conjunto de aristas dirigidas y de un conjunto de nodos	Dictionary - Set
* Buscar una arista en el conjunto y retornarla si es que está incluída	Set - AristaDirigida
* Crear e inicializar	Dictionary - Set
* Insertar arista	AristaDirigida
Contratos	
1 G1 Se hereda de la clase Grafo	
2 G2 Se hereda de la clase Grafo	
3 G3 Se hereda de la clase Grafo	
4 GD1 * Eliminar arista si es que existe * Eliminar nodo (y las aristas que llegan y parten de él) * Obtener aristas que salen y llegan a un nodo dado Clase Cliente: Hipertexto	 Set - AristaDirigida Dictionary - Set AristaDirigida Set - AristaDirigida
5 GD2 * Eliminar nodo (y las aristas que llegan y parten de él)	Dictionary - Set AristaDirigida

Clase Concreta : **VERSIONGRAFODIRIGIDO** Superclases : GrafoDirigido Subclases : VersionGrafoDirigidoDistinguido	
Objetivo : Especializar la estructura de datos grafo dirigido manteniendo la información correspondiente a la evolución (si existe) que sufrió cada uno de sus nodos.	
Responsabilidades Privadas	Clases Colaboradoras
* Mantener conocimiento de un conjunto de nonodos versionables y un conjunto de aristas dirigidas	Dictionary - Set
* Verifica si el nodo se generó en la última versión y elimina de la historia	NodoVersionable
* Inicializar sin primer nodo	Dictionary - Set
Contratos	
1 G1->VGD3 Se hereda de la clase Grafo y se redefine: * Insertar un nodo con el label dado	NodoVersionable
2 G2 Se hereda de la clase Grafo	
3 G3 Se hereda de la clase Grafo	
4 GD1->VGD4 Se hereda de la clase GrafoDirigido y se redefine: * Eliminar un nodo (contemplando la historia)	Dictionary NodoVersionable
5 GD2 Se hereda de la clase GrafoDirigido	
6 VGD1	
* Crear e inicializar con primer nodo	NodoVersionable Dictionary - Set
* Devolver una copia del objeto receptor con el mismo contenido	NodoVersionable AristaDirigida
* Crear e inicializar	Dictionary - Set
Clase Cliente: VersionAplicacion	
7 VGD2	
* Inicializar con primer nodo	NodoVersionable
* Decir si el nodo con el label dado tiene más de un nodo en la historia	NodoVersionable

Continuación ...

* Modificar contenido de un nodo dado reflejando evolución del mismo	NodoVersionable
* Obtener siguiente nodo en la historia de un nodo dado	NodoVersionable
* Obtener anterior nodo en la historia de un nodo dado	NodoVersionable
Clase Cliente: VersionHipertexto	

Clase Concreta : **NODO** Superclases : Object Subclases : NodoVersionable	
Objetivo : Soportar información de cualquier clase	
Responsabilidades Privadas	Clases Colaboradoras
* Mantener conocimiento de su label	String
* Mantiene conocimiento de su contenido	ClaseContenido
Contratos	
1 N1	
* Devolver su contenido	ClaseContenido
* Modificar su contenido	ClaseContenido
* Deshacerse	—
* Crear e inicializar	—
* Devover su label	—
Clase Cliente: Grafo	
2 N2	
* Visualizarse (O)	Gráfico
Clase Cliente: GrafoDirigidoDistinguido	

Clase Concreta	: **NODOVERSIONABLE**
Superclases	: Nodo
Subclases	: —

Objetivo : Especializa la clase Nodo manteniendo la información correspondiente a la evolución de un nodo.

Responsabilidades Privadas	Clases Colaboradoras
* Mantener conocimiento de su ubicación en la historia, su label y su contenido	Historia ClaseContenido
* Inicializar	Historia ClaseContenido
Contratos	
1 N1 Se hereda de la clase Nodo	
2 N2 -> NV3 Se hereda de la clase Nodo y se redefine: * Visualizarse (preguntando si tiene historia)(OO)	 Gráfico Historia
3 NV1 * Modificar contenido y vincular en la historia el nuevo nodo con el que se modifica generando una nueva versión del nodo * Crear * Retorna verdadero si la historia tiene más de un elemento * Devolver siguiente nodo en la historia * Devolver anterior nodo en la historia * Devolver siguiente nodo en la historia * Devolver anterior nodo en la historia * Devolver primer nodo en la historia * Devolver último nodo en la historia * Ponerse en vacío	 Historia ClaseContenido — Historia Historia Historia Historia Historia Historia Historia Historia ClaseContenido
Clases Clientes: - VersiónGrafoDirigido	

Clase Concreta : **ARISTA** Superclases : Object Subclases : AristaDirigida	
Objetivo : Establece una relación bidireccional entre dos nodos.	
Responsabilidades Privadas * Mantiene conocimiento sobre la relación entre los dos nodos que vincula Contratos 1 A1 * Comparación por igualdad (=) * Dado un nodo (label) decir si tiene adcente en la arista * Dado un nodo (label) devuelve el adyacente en la arista * Visualizarse (-) Clase Cliente: - Grafo	**Clases Colaboradoras** String String String String Gráfico

Clase Concreta : **ARISTADIRIGIDA** Superclases : Arista Subclases : AristaDirigidaDistinguida	
Objetivo : Establece una relación unidireccional entre dos nodos.	
Responsabilidades Privadas	Clases Colaboradoras
* Mantiene conocimiento sobre la relación entre los dos nodos que vincula con la particularidad de que uno de ellos es cabeza y el otro cola	String
Contratos	
1 A1 -> AD1 Se hereda de la Clase Arista y se redefinen:	
* Comparación por igualdad (=)	String
* Dado un nodo decir si tiene adyacente en la arista (éste redefine el método en la arista)	String
* Dado un nodo (label) devuelve el adyacente en la arista	String
* Visualizarse (->)	Gráfico
Clase Cliente: - GrafoDirigido	

Clase Concreta : **HISTORIA** Superclases : Object Subclases : -	
Objetivo : Representar la evolución de un nodo	
Responsabilidades Privadas	**Clases Colaboradoras**
* Mantiene conocimiento sobre la evolución de un nodo	OrderedCollection
Contratos	
1 HI1	
* Crear e inicializar	OrderedCollection
* Ir al primer nodo de la historia	OrderedCollection
* Ir al último nodo de la historia	OrderedCollection
* Recibe un nodo, lo agrega al final y retorna la ubicación (posición) de ese nodo en la historia	OrderedCollection
* Es vacía	OrderedCollection
* Deshacer la última referencia de la historia	OrderedCollection
* Recibe una posición y retorna el objeto que referencia la posición siguiente (si es que existe)	OrderedCollection
* Recibe una posición y retorna el objeto que referencia la posición anterior (si es que existe).	OrderedCollection
Clase Cliente: - NodoVersionable	

2- TARJETAS DE CLASES DEL PROTOTIPO EXTENDIDO PARA SOPORTAR HIPERTEXTOS VERSIONABLES, OBTENIDAS AL APLICAR LA TECNICA DE DISEÑO O.O. BASADO EN RESPONSABILIDADES

Clase Concreta	: **HIPERTEXTOVERSIONABLE**
Superclases	: AplicacionConGrafoVersionable
Subclases	: -

Objectivo: Mantener la evolución que sufrió un hipertexto, registrando cada una de las versiones del hipertexto original.

Responsabilidades Privadas	Clases Colaboradoras
* Mantener conocimiento de un conjunto de versiones de hipertextos	OrderedCollection
* Desplegar en pantalla una versión dada.	Gráfico VersionHipertexto
Responsabilidades Públicas:	
* Crear e inicializar con primer nodo la versión original del hipertexto versionable	OrderedCollection VersionHipertexto
* Generar una nueva versión a partir de la última versión del hipertexto versionable realizando modificaciones, inserciones, eliminaciones, etc.	OrderedCollection VersionHipertexto
* Modificar la última versión (sin generar nueva versión)	OrderedCollection VersionHipertexto
* Navegar a través de una versión del hipertexto versionable con la posibilidad de recorrer la historia de un nodo.	OrderedCollection VersionHipertexto
* Navegar a través de la historia de un nodo a partir de un nodo original.	OrderedCollection VersionHipertexto

Clase Concreta : **HIPERTEXTO**
Superclases : AplicacionConGrafo
Subclases : VersionHipertexto

Objetivo : Modelizar un hipertexto como una red donde los nodos representan conceptos y los links relaciones entre ellos.

Responsabilidades Privadas	Clases Colaboradoras
* Mantener conocimiento sobre el soporte estructural para un hipertexto.	GrafoDirigidoDisting.
* Inicializar con primer nodo	GrafoDirigidoDisting.
* Decir si es nodo dado es de comentario	GrafoDirigidoDisting.
* Eliminar nodo de comentario	GrafoDirigidoDisting.
* Eliminar enlace con testeo de si es de comentario	GrafoDirigidoDisting.
Contratos	
1 AG1 Se hereda de AplicacionConGrafo	
2 H1	
* Crear e inicializar con primer nodo	GrafoDirigidoDisting.
* Eliminar un enlace de referencia o de comentario	GrafoDirigidoDisting.
* Agregar un enlace de referencia o de comentario	GrafoDirigidoDisting.
* Desplegarse en pantalla	Gráfico-GrafoDirDist.
* Devolver los labels de los nodos del hipertexto conectados por aristas de un tipo dado	GrafoDirigidoDisting.
* Navegar a través del hipertexto con la posibilidad de editarlo (a partir de un nodo dado)	GrafoDirigidoDisting.
* Navegar a través del hipertexto (browse)	GrafoDirigidoDisting.
* Eliminar el nodo con el label dado	GrafoDirigidoDisting.
Clase Cliente: - HipertextoVersionable	

Clase Concreta : **VERSIONHIPERTEXTO** Superclases : Hipertexto, VersionAplicacion Subclases : -	
Objetivo : Modelizar una versión de un hipertexto.	
Responsabilidades Privadas Contratos 1 H1 Se hereda de la clase Hipertexto 2 VA1 Se hereda de la clase VersionAplicacion 3 AG1 Se hereda de la clase AplicacionConGrafo 4 VH1 * Crear e inicializar con primer nodo * Devolver una copia del objeto receptor * Navegar a través de la versión del hipertexto con la posibilidad de recorrer la historia de un nodo * Navegar a través de la versión del hipertexto con la posibilidad de editarlo, pudiendo generar una versión de un nodo (si se modifica su contenido) Clase Cliente: - HipertextoVersionable	VersionGrafoDirigDist. VersionGrafoDirigDist. VersionGrafoDirigDist. VersionGrafoDirigDist.

Clase Concreta : **GRAFODIRIGIDODISTINGUIDO**	
Superclases : GrafoDirigido	
Subclases : VersionGrafoDirigidoDistinguido	

Objetivo : Especializar la estructura de datos grafo dirigido dándole semántica a las aristas.

Responsabilidades Privadas	Clases Colaboradoras
* Mantener conocimiento sobre el conjunto de aristas dirigidas distinguidas y de un conjunto de nodos	Dictionary - Set
* Decir si el nodo con un label dado es cabeza de alguna dirigida distinguida.	AristaDirigidaDisting.
Contratos	
1 GD1 Se hereda de la clase GrafoDirigido	
2 GD2 Se hereda de la clase GrafoDirigido	
3 G1 Se hereda de la clase Grafo	
4 G2 Se hereda de la clase Grafo	
5 G3 Se hereda de la clase Grafo	
6 GDD1	
* Crear e inicializar	Dictionary - Set
* Dibujarse con las aristas de un cierto tipo solamente.	Gráfico - Nodo AristaDirigidaDisting.
* Devolver los labels de los nodos del grafo conectados por aristas de cierto tipo	AristaDirigidaDisting. Nodo
* Insertar enlace en el conj. de aristas	Set - AristaDirigDist.
* Eliminar el nodo con un label dado, si es que está desconectado	Dictionary - Nodo AristaDirigidaDisting.
Clase Cliente: - Hipertexto	
7 GDD2	
* Devolver los labels de nodos adyacentes con aristas distinguidas de un cierto tipo.	AristaDirigidaDisting.
Clase Cliente: - VersionHipertexto	

Clase Concreta : **VERSIONGRAFODIRIGIDODISTINGUIDO**
Superclases : VersionGrafoDirigido, GrafoDirigidoDistinguido
Subclases : -

Objetivo : Especializar la estructura de datos grafo dirigido manteniendo la información correspondiente a la evolución (si existe) que sufrió cada uno de sus nodos y dándole semántica a las aristas.

Responsabilidades Privadas	Clases Colaboradoras
* Mantener conocimiento sobre el conjunto de nodos versionables y un conjunto de aristas dirigidas distinguidas	Dictionary - Set

Contratos

1 G2
Se hereda de la clase VersionGrafoDirigido

2 G3
Se hereda de la clase VersionGrafoDirigido

3 GD2
cd Se hereda de la clase VersionGrafoDirigido

4 VGD1
Se hereda de la clase VersionGrafoDirigido

5 VGD2
Se hereda de la clase VersionGrafoDirigido

6 VGD3
Se hereda de la clase VersionGrafoDirigido

7 VGD4
Se hereda de la clase VersionGrafoDirigido

8 GDD1
Se hereda de la clase GrafoDirigidoDistinguido

9 GDD2
Se hereda de la clase GrafoDirigidoDistinguido

Clase Concreta : **ARISTADIRIGIDADISTINGUIDA** Superclases : AristaDirigida Subclases : -

Objetivo : Establece una relación unidireccional entre dos nodos.

Responsabilidades Privadas	Clases Colaboradoras
* Mantiene conocimiento sobre el tipo de la arista dirigida.	String
Contratos	
1 AD1 -> ADD1 Se hereda de la Clase AristaDirigida y se redefinen: * Comparación por igualdad (=) * Visualizarse (-> con un color distinto según el tipo). Clase Cliente: - GrafoDirigido	 String Gráfico
2 ADD2 * Dado un nodo y un tipo de arista, decir si el nodo tiene adyacente de ese tipo en la arista. * Crear e inicializar una arista distinguida de un tipo dado. * Devolver el tipo de una arista distinguida. Clase Cliente: - GrafoDirigidoDistinguido	 String String -

BIBLIOGRAFIA

[AHM91] Rafi Ahmed, Shamkant B. Navathe - "Version Management of Composite Objects in CAD Databases." - 1991

[AHO83] A. Aho, J. Hopcroft, J. Ullman - "Data Structures and Algoritms" - 1983

[BAN86] J. Banerjee, H. J. Kim, H. F. Korth - "Schema Evolution in Object Oriented Persistent Databases" - Proceedings Sixth Advanced Database Symposium - Agosto 1986

[BAN87a] J. Banerjee, Won Kim, H. J. Kim, H. F. Korth - "Semantics and Implementation of Schema Evolution in Object-Oriented Databases" - Pág. 311-322 - 1987

[BAN87b] J. Banerjee, H. T. Chou, J. F. Garza, W. Kim, D. Woelk, N. Ballou - "Data Model Issues for Object-Oriented Applications" - ACM Transactions on Office Information Systems - Pág. 3-26 - 1987

[BAN88] Bancilhon - "Object Oriented Database Systems." -1988

[BJO89] A. Björinerstedt, C. Hultén - "Version Control in an Object-Oriented Architecture" - Object-Oriented Concepts, Databases, and Applications - Edited by Won Kim, F. Lochovsky

[BLO87] T. Bloom, S. Zdonik - "Issues in the Design of Object-Oriented Database Programming Languages" - OOPSLA '87 Proceedings - Pág. 441-451 - 1987

[BOR] A. Borgida, J. Mylopoulos, H. Wong - "Generalization / Specialization as a Basis for Software Specification" - On Conceptual Modeling - Edited by Brodie, Mylopoulos, Schmidt - Cap. 4 - Pág. 87-114

[BOR85] A. Borgida - "Language Features for Flexible Handling of Exceptions in Information Systems" - ACM TODS - 1985

[BOR89] Borrione, Prinetto - "Zero-Defect Design, why and how: Formal Verification vs. Automated Systems." - Information Processing 89 - (North-Holland)

[BRE89] Robert Bretl, David Maier, Allen Otis - "The GemStone Data Management System" - Object-Oriented Concepts, Databases, and Applications - Edited by W.Kim and F.Lochovsky - 1989 - Cap. 12 - Pág. 283-307

[CAT] R. G. G. Catell - "Object Data Management"

[CON87] Jeff Conklin - "Hypertext: An Introduction and Survey" - IEEE Computer, Vol. 20, N.9 - Septiembre 1987 - Pág. 17-41

[DEL87] Norman M. Delisle, Mayer D. Schwartz - "Contexts. A Partitioning Concept for Hypertexts." - ACM Press 1987 - Pág. 168-186

[DIA90] Javier Diaz, Gustavo Rossi - "Tópicos Avanzados en Programación Orientada a Objetos" - 1990

[FIS89] D. H. Fishman, J. Annevelink, E. Chow, T. Conners - "Overview of the Iris DBMS" - Object-Oriented Concepts, Databases and Applications - Edited by W.Kim and F.Lochovsky - 1989 - Cap. 10 - Pág. 219-249

[GLU90] Robert J. Glushko - "Report from the User Requirements Working Group" - Proceeding of the Hypertext Standardization Workshop" - 1990 - Pág. 29-35

[KAT90] R. H. Katz, E. Chang - "Managing Change in a Computer-Aid Design Database" - Reading in Object-Oriented Database Systems. - 1990[MEN91] A. Mendelzon - "Theory and Practice of Visual Queries" - 1991

[KIM87] W. Kim, J. Banerjee, H. Chou, J. Garza, D. Woelk - "Composite Object Support in an Object-Oriented Database System." - OOPSLA '87 Proceedings - 1987 - Pág. 118-125

[KIM89a] Won Kim, Nat Ballou, Hong Tai Chou - "Features of the Orion Object-Oriented Database System." - Objects-Oriented Concepts, Databases, and Applications - Edited by Won Kim and F.Lochovsky - Cap. 11 - Pág. 251-281 - 1989

[KIM89b] W. Kim, E. Bertino, J. F. Garza - "Composite Objects Revisted" - SIGMOD '89 - Pág. 337-347 - 1989

[MEN89a] M. Conseus, A. Mendelzon - "GraphLog: A Visual Formalism for Real Life Recursion" - Office and DataBase Systems Research - 1989

[MEN89b] M. Conseus, A. Mendelzon - "Expressing Structural Hypertext Queries in GraphLog" - Office and DataBase Systems Research - 1989

[PAR90] H. Van Dyke Parunak - "Reference on Data Model Group (RDMG): Work Plan Status." - Proceeding of the Hypertext Standardization Workshop" - 1990 - Pág. 9-13

[PRO90] Proceeding of the Hypertext Standardization Workshop 1990 - Pág. 10-12

[SIM91] Sergiu S. Simmel, Iva Godard - "The Kala Basquet. A Semantic Primitive Unifying Object Transaction Access Contro, Version and Configurations." - OOPSLA '91 - Pág. 230-246

[SKA87] Skarra, Zdonik - "Type Evolution in an Object Oriented Database" - Research Directions in Object-Oriented Programming - Edited by Bruce Shriver and Peter Wegner - MIT Press - 1987 - Pág. 393-415

[WIR90] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener - Designing Object-Oriented Software. (Capitulo 1) - Pág. 1-16

[ZDO] Zdonik. "Object Management Systems for Design Environments."

[ZDO90] Readings in Object-Oriented Database Systems (Capitulo 6) - Edited by Zdonik and Maier - 1990