ANEXO B

CONOCIENDO LAS CLASES PRINCIPALES DE OSIMIS

Este anexo es de gran ayuda para complementar el conocimiento de OSIMIS, aquí se expone la funcionalidad de las clases principales que utiliza OSIMIS, adentrándonos así en su implementación. Como habíamos hecho notar anteriormente en el Capitulo V de esta monografía, las clases C++ que debemos conocer se clasifican en tres partes:

1. Clases para el soporte del proceso de coordinación, para escribir aplicaciones manejadas por eventos asíncronos:
   - Clase KS (Knowledge Source).
   - Clase Coordinator.
   - Clase ISODECoordinator.
2. Clases para el soporte del manejo transparente ASN.1:
   - Clase Attr.
   - Clase AnyType.
   - Clase AVA.
3. Clases para el soporte del sistema de gestión genérico (GMS):
   - Clase MOClassInfo.
   - Clase MO.
   - Clase Top.

Recordemos que en el capitulo II sección 2.2 se estudio la arquitectura en capas de OSIMIS, donde se aclaro que las aplicaciones que trabajan sobre OSIMIS utilizan principalmente para su funcionamiento el sistema de gestión genérico y las API´s de soporte de coordinación y ASN.1 de alto nivel, pues bien las clases que se van a estudiar en este anexo son aquellas que le dan las características a las capas nombradas (a saber GMS, soporte de coordinación y soporte ASN.1).

El conocimiento de estas clases y sus métodos nos permitirá adentrarnos en la manera como esta implementado OSIMIS, y así poder acoplar a éste nuestros agentes y objetos que necesitan basarse en los métodos utilizados por estas clases en el momento de su creación como observamos en el capitulo V.

Demos pues comienzo al estudio de las clases C++.

B.1 CLASES PARA EL SOPORTE DEL SISTEMA DE GESTION GENERICO.

B.1.1 CLASE MOClassInfo.
Heredada desde: Ninguna
Clases usadas: NameBinding, Attr y clases derivadas.
Archivo interfaz: MOClassInfo.h.
Archivo de implementación: MOClassInfo.cc.
Librería conteniéndola: gms.

B.1.1.1 Introducción:

La clase MOClassInfo es una meta-clase que describe a una clase de objeto gestionado, y siempre existe una instancia de ésta para cada clase de objeto gestionado. Esta instancia contiene información común para todas las otras instancias de las clases, tal como atributos, grupo, identificadores de eventos y acciones, nombres enlazados etc.
Las instancias de esta clase son enlazadas jerárquicamente en un árbol que refleja la herencia jerárquica GDMO, tal información de la meta-clase es vital para aplicaciones en el rol de agente. El código que inicializa el objeto meta-clase es producido por el compilador GDMO como un método estático de cada clase de objeto gestionado (*initialiseClass*).
Muchos de los métodos de esta clase son solo usados por el GMS para accesar información que es necesaria para verificar las peticiones de nombres enlazados, y tambien para la creación y comportamientos relacionados con la supresión, la existencia y validez de atributos y valores accesados en operaciones get, set y create, así como la existencia y validez de las acciones y sus argumentos.
Los únicos métodos que pueden ser de interés para los implementadores de aplicaciones agente son aquellos que proveen acceso a las plantillas de objetos para la información de acción y respuesta (ver también el método MO::action). Esta clase cuenta con el método *getParent* que nos permite ascender una rama en la herencia mientras otro método *getFirst* provee acceso a la primera instancia de la clase, desde la cual las instancias subsecuentemente pueden ser accesadas. Existen otros metodos que proveen acceso a la información de la clase pero que no consideramos necesario describirlos aquí,pero las declaraciones de los archivos cabecera pueden ser consultados si es necesario.

B.1.1.2 Métodos:

```
class MOClassInfo
{
// ...
public:
Attr* getActionInfoTemplate (int);
Attr* getActionReplyTemplate (int);
MOClassInfo* getParent ();
MO* getFirst ();
// ...
};
```

B.1.1.3 Métodos de acceso a la plantilla de acción:

Attr getActionInfoTemplate (actionId);*
Attr getActionReplyTemplate (actionId);*

El argumento de accion y la información de respuesta son entregados a la interfaz del método polimorfo *MO::action* como una estructura de datos en C. El código de usuario de un método de acción específica puede recurrir a una clase Attr especifica a fin de poder manipular el parámetro de acción y la información de respuesta, por ejemplo para imprimir, analizar, o liberar valores (ver la especificación de la clase Attr).

El parámetro para ambos métodos es actionId (una etiqueta entero), tal como es producido por el compilador GDMO para una clase de objeto de gestión especifica, por ejemplo, I_calcSqrt para la acción calcSqrt de la clase simpleStats.
El valor retornado por Attr puede ser usado como un motor de sintaxis y NO debe ser liberado.

B.1.1.4 Accesando la jerarquía de la clase:

MOClassInfo getParent ();*

Cada clase especifica MO producida por el compilador GDMO contiene una instancia estática de MOClassInfo la cual es compartida por todas las instancias de esa clase. La convención es que esta instancia de la meta-clase puede ser accesada por una instancia de la variable private _classInfo o por un método *public getClassInfo()*, todas las instancias de la meta-clase son entonces enlazadas en un árbol reflejando la herencia jerárquica, y el método *getParent* permite ascender en esta jerarquía. Por

ejemplo, *eventForwardingDiscriminator::getClassInfo->getParent()* producirá el objeto meta-clase para discriminator.

B.1.1.5 Caminando a través de todas las instancias de una clase:
MO getFirst ();*
Este método fue inicialmente proveído como una forma de acceder a todas las instancias de una clase particular teniendo acceso al objeto meta-clase, esto obviamente es solo significativo en aplicaciones agente. Aunque este método es aun disponible, es menospreciado lo cual significa que este no será proveído en futuras versiones, la razón para esto es que éste provee acceso a todas las instancias de una clase actual particular, independientemente de su posición en el árbol de información de gestión (MIT), éste incluso provee acceso a instancias en diferentes MITs si hay muchas aplicaciones agente presentes en el mismo bloque físico, y esto es ilegal y peligroso. La clase MO provee un conjunto comprensivo que permite buscar en el MIT y pueden servir para el mismo propósito, por lo tanto es muy aconsejable evitar usar este método para lograr compatibilidad con futuras versiones.
El método *MO::getClassNext()* puede ser usado luego de usar *MO::getFirst* para caminar a través de todas las instancias de la clase. Notemos que por ejemplo *eventForwardingDiscriminator::getClassInfo()->getFirst()* dará un manejo a todos los eventForwardingDiscriminators pero no a instancias de clases derivadas, esto es, solo las instancias de la clase actual son enlazadas juntas.

B.1.2 CLASE MO.
Heredada desde: Ninguna
Clases usadas: MOClassInfo, Attr
Archivo interfaz: GenericMO.h
Archivo de implementación: GenericMO.cc
Librería conteniéndola: gms

B.1.2.1 Introducción:
MO es la superclase abstracta de todas las clases de objetos gestionados. Esta contiene información relacionada a la posición de una instancia de objeto gestionado en el árbol de contenimiento y maneja todas las instancias de objeto de las meta-clases. Esta clase provee un conjunto de métodos polimorfos que pueden ser redefinidos en clases derivadas para conseguir la funcionalidad deseada. MO es el padre de Top el cual es subsecuentemente el padre de la clase derivada especifica producida por el compilador GDMO, los métodos de esta clase contienen plantillas de código genérico estándar que puede ser aumentado con código del usuario implementando los métodos polimorfos mencionados anteriormente.

B.1.2.2 Métodos:
```
class MO
{
// ...
// métodos polimorfos  (para ser posiblemente proveídos en clases derivadas)
public:
/* static RDN makeRdn (MO*); // producido para clases derivadas */
virtual int createRR (AVA*&, void* = NULLVD, int = -1);
virtual int deleteRR (AVA*&, void* = NULLVD, Bool = False, int = -1);
protected:
virtual int get (int, int, AVA*&, Bool = False, int = -1);
virtual int set (CMISModifyOp, int, int, void*, AVA*&, Bool = False, int = -1);
virtual int action (int, int, void*, void*&, Bool&, AVA*&, Bool = False, int = -1);
virtual int buildReport (int, int, void*&, Bool&);
virtual int refreshSubordinates (AVA*&, int = -1);
virtual int refreshSubordinate (RDN, AVA*&, int = -1);
```

```
// callbacks asincronos (resultantes de las llamadas anteriores)
public:
int createRRes (int, AVA*);
int deleteRRes (int, AVA*);
int getRes (int, AVA*);
int setRes (int, AVA*);
int actionRes (int, void*, Bool, Bool, AVA*);
int refreshSubordinatesRes (int, AVA*);
// metodos generales de acceso al MIT
public:
static int initialiseMIB (char*);
static MO* getRoot ();
static DN mn2localdn (MN);
static DN getAgentNameDomainDN ();
void deleteWholeSubtree (DeletionType = dt_undefined, Bool = False);
int update (AVA*&);
Attr* getUpdatedAttr (OID, AVA*&); // fuerza una operación GET de CMIS
MO* getMO (DN, Bool, AVA*&);
MO* getMO (char*, Bool, AVA*&);
MO* getSubordinate (RDN, Bool, AVA*&);
MO* getSubordinate (char*, Bool, AVA*&);
MO** getSubordinates (Bool, AVA*&);
MO** getWholeSubtree (Bool, AVA*&, Bool = False);
MO* getSubordinate (); // primer subordinado en el MIT
MO* getSuperior ();
MO* getPeer ();
MOClassInfo* getClassInfo ();
MO* getClassNext ();
OID getClass ();
char* getClassName ();
int checkClass (char*, Bool* = NULL);
int checkClass (OID, Bool* = NULL);
RDN getRDN ();
char* getRDNString ();
DN getDN (Bool = False);
char* getDNString (Bool = False);
// ...
};.
```

B.1.2.3 Métodos polimorfos y callbacks asincrónicas relacionadas:

Estos métodos pueden ser todos redefinidos en clases derivadas especificas para implementar el comportamiento asociado. El método *makerdn* es actualmente producido por el compilador GDMO para las clases con la propiedad de nombrado automático de instancias "automatic-instance-naming", por lo tanto este no es un método polimorfo y esta presente dentro de comentarios en esta clase. Los métodos polimorfos y propiedades son relacionadas a operaciones CMIS, llamadas Get, Set, Action, Create and Delete.

El método *buildReport* es solo un vestigio del pasado, el método *triggerEvent* de la clase Top debe ser mejorado para soportar la información transitoria necesaria (ver clase Top). Los métodos *refreshSubordinate(s)* se asocian al direccionamiento de objetos y scoping cuando una política "fetch-upon-request" es ejercida entre objetos gestionados y recursos asociados. La filosofía detrás de la API de los métodos createRR, deleteRR, get, set y action es que una clase puede ser complementada con comportamiento de tal forma que haga posible la total reusabilidad de la clase para futuras clases derivadas, como tal, cada una de estas llamadas debe ser pasada desde la hoja del padre de la rama de herencia en cualquier implementación.

En el caso de createRR, aunque las llamadas deben tener la misma dirección "hacia arriba", el código de comportamiento en estas llamadas debe operar en un modelo "hacia abajo", como durante la construcción de un objeto en un típico lenguaje orientado a objetos, esto es posible organizando el contenido de createRR apropiadamente. En el caso de get y set, cada atributo es pedido / fijado separadamente y una indicación "no mas atributos" es entregada al final. Si todos los atributos son pedidos, solo una llamada "get-all" es pasada al objeto. En el caso del método get, el código de comportamiento tiene que actualizar los atributos pedidos (o todos ellos de acuerdo a la política de actualización). Si una política "cache-ahead" periódica es ejercida, entonces el método get no necesita ser redefinido para todos. En el método set, el argumento newAttrValue debe ser usado solo por cualquier interacción con el recurso real especifico, esto es, el GMS actualiza el atributo con el nuevo valor, si allí no hay tal interacción, el método set no necesita ser redefinido.
En el caso de get, set, action y deleteRR, el sistema de gestión remoto puede tener peticiones de atomicidad en las operaciones, en cuyo caso las llamadas toman lugar dos veces: primero con la bandera checkOnly fijada a "True" y luego sin esta. En el primer paso, el código de la clase específica debe chequear si la operación de petición puede ser desempeñada, si al menos uno de los objetos envueltos no puede desempeñar esta operación, la serie completa de operaciones es abortada.
Como OSIMIS es diseñado para operar sobre un paradigma de "hilo de ejecución simple", aquellas operaciones pueden ser asíncronas, si ellas envuelven acceso a una entidad remota. En este caso, los métodos deben retornar un valor adecuado que asegure que ellos desempeñaran la operación o chequeo asincrónicamente, los métodos deben entonces obtener el argumento operId el cual servirá como el "justificante" para el posterior resultado asíncrono. En el caso de una acción, una serie de resultados asíncronos se hace posible y como tal un campo "more" estará presente en el callback asíncrono para indicar este.
Finalmente todas estas llamadas pueden fallar por cualquier razón relacionada a la interacción entre el objeto gestionado y el recurso real, el errorInfo en este caso debe contener el código de error y puede adicionalmente contener un par tipo / valor para un error de falla en el procesamiento.

```
// métodos polimorfos y sus argumentos
RDN makeRdn (MO* superior) // producido por el compilador GDMO en clases derivadas
int createRR (AVA*& errorInfo, void* createInfo, int operId);
int deleteRR (AVA*& errorInfo, void* deleteInfo, Bool checkOnly, int operId);
int get (int attrId, int classLevel, AVA*& errorInfo, Bool checkOnly, int operId);
int set (CMISModifyOp setMode, int attrId, int classLevel, void* newAttrValue, AVA*& errorInfo,
Bool checkOnly, int operId);
int action (int actionId, int classLevel, void* actionInfo, void*& actionReply, Bool& freeFlag, AVA*&
errorInfo, Bool checkOnly, int operId);
int buildReport (int eventId, int classLevel, void*& eventInfo, Bool& freeFlag);
int refreshSubordinates (AVA*& errorInfo,int operId);
int refreshSubordinate (RDN rdn, AVA*& errorInfo,int operId);
// callbacks asíncronos resultantes
int createRRes (int operId, AVA* errorInfo);
int deleteRRes (int operId, AVA* errorInfo);
int getRes (int operId, AVA* errorInfo);
int setRes (int operId, AVA* errorInfo);
int actionRes (int operId, void* actionReply, Bool freeFlag, Bool moreFlag, AVA* errorInfo);
int refreshSubordinatesRes (int operId, AVA* errorInfo);.
```

B.1.2.4 Métodos de acceso a la información de gestión general:
El árbol de información de gestión es representado internamente como un árbol binario, de esta forma un numero de métodos permiten caminar a través de este y accesar información, en muchos de estos, uno puede explícitamente pedir el refrescamiento del MIT antes de retornar información particular, en este caso, si el refrescamiento o actualización envuelve comunicaciones remotas estas pueden presentar problemas, esta es la razón de que un parámetro errorInfo pueda ser proveído por el usuario de aquellas llamadas a ser llenadas, pero si el usuario no esta interesado en esta información, el valor

DISCARDAVA puede ser entregado. Las llamadas que no envuelven el refrescamiento MIT no pueden tener errores.
Si comunicaciones asíncronas son implementadas por clases de objetos particulares para el refrescamiento o actualización del MIT, entonces un error especial será retornado, el cual nos dira "usted no puede obtener esta información en un modelo asíncrono", puesto que aunque sea el código de usuario el que invocara estas llamadas, este debe conocer la semántica de las clases envueltas (después de todo esta es la aplicación que desarrolla el implementador) y tales errores deben solo ocurrir durante el periodo de desarrollo. El completo conjunto de métodos que el usuario puede usar son presentados debajo.

// métodos de acceso al MIT general
int initialiseMIB (char mibInitFileName);*
MO getRoot ();*
DN mn2localdn (MN mname);
DN getAgentNameDomainDN ();
void deleteWholeSubtree (DeletionType deleteType, Bool includingBase);
int update (AVA& errorInfo);*
Attr getUpdatedAttr (OID attrOid, AVA*& errorInfo);*
MO getMO (DN dn, Bool refresh, AVA*& errorInfo);*
MO getMO (char* dnStr, Bool refresh, AVA*& errorInfo);*
MO getSubordinate (RDN rdn, Bool refresh, AVA*& errorInfo);*
MO getSubordinate (char* rdn, Bool refresh, AVA*& errorInfo);*
*MO** getSubordinates (Bool refresh, AVA*& errorInfo);*
*MO** getWholeSubtree (Bool refresh, AVA*& errorInfo, Bool includingBase);*
MO getSubordinate (); // primer subordinado en el MIT binario interno*
MO getSuperior ();*
MO getPeer ();*
MOClassInfo getClassInfo ();*
MO getClassNext ();*
OID getClass ();
char getClassName ();*
int checkClass (char className, Bool* onlyActualAndAllomorphs);*
int checkClass (OID className, Bool onlyActualAndAllomorphs);*
RDN getRDN ();
char getRDNString ();*
DN getDN (Bool global);
char getDNString (Bool global);.*

B.1.2.5 Métodos proveídos por el compilador GDMO en clases derivadas:
Un numero de métodos son producidos por el compilador GDMO y pueden ser usados por los implementadores de la aplicación. Si una clase tiene la propiedad "create-with-automatic-instance-naming", un stub (matriz) para el método *makeRdn* es producido, y si una acción esta presente en el GDMO, el stub relevante es también producido. Este no es el caso respecto al método set cuyos atributos pueden ser fijables pero sin asociar comportamiento al recurso real, lo mismo que se aplica para el método get el cual puede solo ser necesario de acuerdo a las políticas de actualización. Finalmente, el método *buildReport* no es producido ya que este no es necesario para el objeto, las notificaciones de la gestión de estado serán eventualmente eclipsadas.
El método estático *getClassInfo* provee acceso al objeto meta-clase para una clase particular mientras un método estático *create* posibilita crear una instancia.
Finalmente, los atributos de una instancia particular pueden ser accedidos a través de métodos inline producidos por el compilador GDMO el cual tiene exactamente el mismo nombre que el atributo, por ejemplo: OperationalState* operationalState().

MOClassInfo getClassInfo ();*

MO create (RDN rdn, MO* superior);*

B.1.3 CLASE Top.
Heredada desde: MO
Clases usadas: MOClassInfo, discriminator, eventForwardingDiscriminator, log, logRecord, eventLogRecord, AttrValue (para notificaciones), ObjectClass, ObjectClassList, ObjId, ObjIdList (atributos)
Archivo interfaz: Top.h
Archivo de implementación: Top.cc
Librería conteniéndola: gms

B.1.3.1 Introducción:
La clase Top implementa la raíz del árbol de herencia de información de gestión, es decir la clase desde la cual todos las otras clases de objetos gestionados son derivadas. Sus atributos describen el propio objeto gestionado para el propósito del acceso de gestión y son fundamentales para permitir el descubrimiento de las capacidades de un MIT en un modelo genérico. Estos atributos son la clase de objeto, nombres enlazados, paquetes y alomorfismo.
OSIMIS soporta el alomorfismo, pero los paquetes no son aun apropiadamente soportados a través del compilador GDMO, ya que el compilador GDMO reconoce los paquetes condicionales pero no produce (aun) el código necesario para soportar su creación dinámica. Una aproximación interina que ha sido usada es hacer estos obligatorios, esto hace difícil algunas veces que paquetes condicionales sean usados para configurar una instancia particular, y en tales casos, mecanismos no estándar deben ser usados para controlar, por ejemplo atributos "switch" adicionales etc.
La razón para describir esta clase es doble: primero, debido a su posición como la raíz del árbol de herencia del objeto gestionado. Segundo y mas importante, debido al hecho que ésta ofrece la interfaz para la función de gestión de notificación de eventos (reporte de eventos y control Log).

B.1.3.2 Métodos:
```
class Top : public MO
{
// ...
protected:
int addAllomorphicClass (OID);
public:
Bool hasPackage (char*);
Bool hasPackage (int, int);
// interfaz a la función de notificación general
int triggerEvent (char*);
int triggerEvent (int, int);
static int relayEventReport (OID, MN, char*, OID, PE);.
 // interfaz para la notificacion de gestion de estado y objeto
int triggerObjectCreation (int = int_SMI_SourceIndicator_resourceOperation,char* = NULLCP);
int triggerObjectDeletion (int = int_SMI_SourceIndicator_resourceOperation,char* = NULLCP);
int triggerAttributeValueChange (AttrValue*,int = int_SMI_SourceIndicator_resourceOperation,
char* = NULLCP);
int triggerAttributeValueChange (List*, // of AttrValue
int = int_SMI_SourceIndicator_resourceOperation, char* = NULLCP);
int triggerStateChange (AttrValue*,int = int_SMI_SourceIndicator_resourceOperation, char* =
NULLCP);
int triggerStateChange (List*, // of AttrValue
int = int_SMI_SourceIndicator_resourceOperation, char* = NULLCP);
// ...
};
```

B.1.3.3 Comportamiento Alomorfo:
int addAllomorphicClass (OID allomorph);
Como no hay un constructor GDMO para especificar que una clase es alomorfa de otra clase, el implementador de clases alomorfas debe proveer esta información a través de esta llamada. Este método es usualmente llamado en el método polimorfo *MO::createRR()* para una clase particular. OSIMIS acepta solo clases padre de una clase a ser especificada como alomorfa, el argumento es el identificador de objeto de la clase alomorfa como esta registrado en el modulo GDMO. Una forma típica de invocar este método es por ejemplo: en el método *uxObj2::createRR* hacerlo a través de *addAllomorphicClass(name2oid("uxObj1"));* OK es retornado en un éxito mientras NOTOK es retornado en una falla por ejemplo: invalid allomorph (not parent).

B.1.3.4 Interfaz de notificación general:
int triggerEvent (char eventName);*
int triggerEvent (int eventId, int classLevel);
Estos métodos ofrecen una interfaz para accionar notificaciones de acuerdo a la especificación de los objetos. De estos el método *int triggerEvent (int eventId, int classLevel)* puede ser convertido a reporte de eventos y/o logs específicos de acuerdo a la presencia de un discriminador de envió de eventos y un objeto de gestión log en la MIT local.

Este procedimiento es totalmente transparente para los implementadores de clases de objetos gestionados, ya que estos son métodos sobrecargados que ofrecen la misma interfaz a través del nombre de la notificación o su identificador. El segundo método es principalmente usado por el código de comportamiento en clases derivadas mientras el primero puede ser usado para accionar una notificación desde fuera de una instancia MO.
Como un resultado de llamarlos, el método polimorfo *MO::buildReport* será llamado, el cual tendrá que ser llenado con la información de la notificación. Esta separación es debido a razones históricas (compatibilidad con versiones anteriores), la información de la notificación puede simplemente tener un parámetro adicional.
OK es retornado en un éxito y NOTOK en una falla (notificación invalida para este objeto).

static int relayEventReport (OID moClass, MN moName, char eventTime, OID eventType, PE eventInfo);*
Las unidades híbridas las cuales actúan en roles de gestor y agente pueden recibir el reporte de eventos. Estos pueden ser posiblemente enviados y/o logged de acuerdo a la presencia de discriminadores de envío de eventos y/o logs en el MIT local. Esto es posible a través de este método estático ya que el reporte de eventos no es emitido por cualquiera de los objetos locales. Los argumentos son la clase, instancia, tiempo del evento, tipo del evento e información del evento como fue recibido desde el otro agente a través de CMIS. Los parámetros son como en la API CMIS de OSIMIS (MSAP).

B.1.3.5 Interfaz de notificación de gestión de objetos y estados:
Las notificaciones objectCreation, objectDeletion, attributeValueChange, y stateChange son extremadamente importantes y como tal son tratadas especialmente. Se proveen métodos especiales para accionarlos, y todos ellos tienen dos argumentos comunes: sourceIndicator y additionaltext. El sourceIndicator indica si fue un resourceOperation el que acciono la notificación (este es por defecto) o un managementOperation. Cualquier espacio de memoria usado para el parámetro additionalText no es liberado por estas llamadas (esta es responsabilidad de quien efectúa la llamada).

int triggerObjectCreation (int sourceIndicator, char additionalText);*
int triggerObjectDeletion (int sourceIndicator, char additionalText);*
Estos no toman ningún argumento adicional. La lista completa de instancias de atributos será enviada con la notificación.

int triggerAttributeValueChange (int attrId, int classLevel, void prevValue, int sourceIndicator, char* additionalText);*
int triggerAttributeValueChange (AttrValue attrValue, int sourceIndicator, char* additionalText);*
int triggerAttributeValueChange (List, int sourceIndicator, char* additionalText);*
Estos toman información que especifica los atributos que cambian y posiblemente los valores previos. Los primeros dos soportan un solo atributo mientras el tercero pasa una lista de atributos que han cambiado. En los dos primeros, solamente prevValue es liberado por el GMS mientras en el tercer caso la lista completa debe tener espacio de memoria asignado el cual es liberado.

int triggerStateChange (int attrId, int classLevel, void prevValue, int sourceIndicator, char* additionalText);*
int triggerStateChange (AttrValue attrValue, int sourceIndicator, char* additionalText);*
int triggerStateChange (List, int sourceIndicator, char* additionalText);*
Estos son exactamente iguales a los anteriores pero para el cambio de estado.

B.2 CLASES SOPORTANDO EL PROCESO DE COORDINACIÓN.

B.2.1 CLASE KS.

Heredada desde: Ninguna
Clases usadas: Coordinator y clases derivadas
Archivo interfaz: GenericKS.h
Archivo de implementación: GenericKS.cc
Librería conteniéndola: kernel

B.2.1.1 Introducción:

La fuente de conocimiento (KS) es una abstracción de un objeto de aplicación general que es capaz de recibir datos asincrónicamente en puntos de comunicación externa y que son despertados periódicamente en tiempo real para desempeñar varias tareas. El termino tiene su raíz en los sistemas de Inteligencia Artificial "blackboard" y generalmente los KSs implementan una aplicación inteligente con respecto a comunicaciones externas y actividades periódicas en tiempo real. El KS es una clase abstracta.

B.2.1.2 Métodos:

```
class KS
{
protected:
// wake-up en tiempo real
int scheduleWakeUp (long, char*, Bool = False);
int scheduleWakeUps (long, char*, Bool = False);
int cancelWakeUps (char*);
// escucha sobre un punto de comunicación externa
int startListen (int);
int stopListen (int);
// notificacion al proceso de cierre
int notifyShutdown ();
// constructor protegido (clase abstracta)
KS ();
public:
// callbacks para los wake-ups, eventos externos, y procesos de cierre
virtual int wakeUp (char*);
virtual int readCommEndpoint (int);
```

virtual int shutdown (int);
// destructor
virtual ~KS ();
};.

B.2.1.3 Peticiones de wake-up en tiempo real:
int scheduleWakeUp (long period, char token, Bool onlyForNotification);*
int scheduleWakeUps (long period, char token, Bool onlyForNotification);*
int cancelWakeUps (char token);*

Los dos primeros métodos piden la iniciación de los wake-up(s) en tiempo real mientras el tercero pide su cancelación.
El método *scheduleWakeUps* programa wake-ups para que se ejecuten cada *period* en segundos, el argumento *token* debe ser una cadena de caracteres única que distinga esta serie de wake-ups de cualquier otra posible petición por la misma fuente de conocimiento. El *token* no necesita tener memoria asignada ya que la infraestructura kernel hace una copia, por ejemplo: una cadena constante tal como "<token>" es suficiente. NULLCP puede ser proveída si esta distinción no es necesaria o si solo una serie de wake-ups es programada por aquella fuente de conocimiento.
El parámetro *onlyForNotification* soporta una optimización aplicable solo a aplicaciones en el rol de agente OSI, y como tal éste es opcional con valor por defecto *Fals*e. Si éste es *Tru*e, la fuente de conocimiento será despertada solo si un discriminador de envío de eventos o log soportando los objetos de gestión están presentes en el sistema. Esto es útil cuando los wake-up son solo usados para soportar notificaciones realizando polling al recurso real. El caso por defecto (parámetro no proveído) es pedir wake-ups independientemente de la función de notificación. Un OK es retornado en un éxito, y un NOTOK si el periodo de scheduling es invalido, por ejemplo menor o igual a cero.
El método *scheduleWakeUp* es exactamente el mismo que el anterior con la diferencia que solo un wake-up es programado.
El método *cancelWakeUps* cancelara cualquiera, múltiples o simples wake-ups identificados por *toke*n. Notemos que si token es NULLCP solo los wake-ups que emparejen con token serán cancelados, esto es, NULLCP no significa "cancele todos los wake-ups pendientes para la fuente de conocimiento". Un OK es retornado en un éxito, y un NOTOK si la operación de cancelación falla, esto significa que no hay programado ningún wake-up con este token.

B.2.1.4 Peticiones de notificación de finalización y de escucha externa:
int startListen (int fd);
int stopListen (int fd);
startListen permite registrar un punto de comunicación externa (típicamente un archivo descriptor de UNIX) en el cual se comienza a escuchar por información. Un OK es retornado en un éxito, y un NOTOK en otro caso (si fd<0 o ya es registrado).
stopListen permite desregistrar un punto de comunicación externa, así que no se escuchara mas sobre este punto. Un OK es retornado en un éxito, y un NOTOK si la operación falla, esto es, que allí no hay tal punto registrado.

int notifyShutdown ();
Este método provee la capacidad de pedir ser notificado cuando un proceso se cierre.

B.2.1.5 Eventos callback polimorfos:
Los siguientes métodos son callbacks para wake-ups, eventos de comunicación externa y cierre de procesos como resultado de las peticiones anteriores. Ellos deben ser proveídos en clases derivadas de acuerdo al uso de los anteriores métodos de petición.

virtual int wakeUp (char token);*
este es llamado como un resultado de los métodos *scheduleWakeUp(s)*. El token es proveído en la llamada de programación, y el método debe retornar OK cuando se redefine por un usuario en una

clase derivada, y un NOTOK es retornado en el nivel de KS, esto es cuando un wakwe-up es pedido y el método no ha sido redefinido (bug en tiempo de desarrollo).

virtual int readCommEndpoint (int fd);
Este es llamado cada vez que hay datos en el punto de comunicación interna identificado por fd. El fd sirve para distinguir el punto de comunicación de otros usados por la misma fuente de conocimiento. El método retorna OK o NOTOK de la misma forma que el anterior wakeUp.

virtual int shutdown (int fd);
Este método es llamado en el tiempo de cierre como resultado de los métodos *startListen* y *notifyShutdown*. En el caso anterior, este es llamado una vez por cada punto de comunicación externa identificado por fd, esta aplicación escucha este punto a fin que el ultimo sea cerrado exitosamente, notemos que allí no es necesario una llamada a *stopListen*. En el ultimo caso el método es llamado con fd=-1 como una indicación de la terminación de procesos, así que la fuente de conocimiento pueda efectuar cualquier otra limpieza necesaria. Si ambos casos se mantienen , esto es, ambos puntos están abiertos y hay una petición *notifyShutdown*, la llamada con fd = -1 es la ultima después de la llamada por todos los puntos de comunicación.
El método retorna OK o NOTOK de la misma forma que el anterior *wakeUp*.

B.2.2 CLASE Coordinator.
Heredada desde: Ninguna
Clases usadas: KS, List
Archivo interfaz: Coordinator.h
Archivo de implementación: Coordinator.cc
Librería conteniéndola: kernel.

B.2.2.1 Introducción:
Una instancia de esta clase coordina las actividades en aplicaciones distribuidas, actuando como un punto central para las comunicaciones a través de todos los puntos externos y para todas las alarmas programadas en tiempo real. Solo una instancia de esta o una clase derivada debe estar presente en una aplicación que es realizada como un proceso simple del sistema operativo. En sistema UNIX este usa la facilidad *select()*, mejorada en ISODE como xselect para ser portable a través de plataformas UNIX. Un completo esquema manejado por eventos es ejercido con respecto a todas las comunicaciones externas y programación de alarmas en tiempo real a través de la política primero en llegar primero en ser servido.
Esta clase es escrita de forma tal que permita la integración a través de herencia con otros paquetes que tengan sus propios mecanismos de coordinación. Esto es necesario para aplicaciones distribuidas con interfaz de usuario grafica que recibe eventos asíncronos desde el teclado y la red, y también puede ser necesario a fin de integrar OSIMIS con otras plataformas de sistemas distribuidos. OSIMIS provee extensiones de esta clase para trabajar con X-Windows y la plataforma ISODE (XCoordinator e ISODECoordinator respectivamente).
Se debe hacer notar que la existencia de una instancia Coordinator es totalmente transparente a los implementadores de la aplicación, quienes pueden accesar estos servicios a través de clases KS derivadas.

B.2.2.2 Métodos:
```
class Coordinator
{
// ...
public:
// bucle central de escucha
void listen ();
// interfaz con otros mecanismos de comunicación
```

static void setTerminateSignalHandler ();
static void setAlarmSignalHandler ();
// ...
};

B.2.2.3 Escucha central:
void listen ();
Este método realiza el bucle de escucha central de una aplicación implementada como un proceso del sistema operativo. Este es el ultimo método llamado dentro del programa *main()* y debe ser llamado después que toda la inicialización necesaria de la aplicación ha finalizado. Este nunca retorna pero el proceso terminara bajo la recepción de una señal de terminación de Unix. El coordinador entonces llamara el método *shutdown* de la fuente de conocimiento como se explica en la definición de la clase KS. Es de notar que este método debe solo ser llamado si el bucle de escucha central esta bajo el completo control de la aplicación. Este es el caso para aplicaciones que no tienen interfaz de usuario grafica.

B.2.2.4 Interfaz con otros mecanismos de comunicación:
static void setTerminateSignalHandler ();
static void setAlarmSignalHandler ();
Estos pueden ser usados para definir como manejar las señales de alarma y terminación en el caso de integración con otro mecanismo de coordinación, por ejemplo con el X-Windows. Estos son solo significantes si el método anterior *void listen ()* no es llamado, y la escucha esta bajo el completo control de otro mecanismo. El primer método debe siempre ser llamado en el caso donde el ultimo pueda o no ser llamado. Si *setAlarmSignalHandler* no es llamado, las alarmas serán manejadas por el otro mecanismo. Este provee la capacidad de usar la API KS y que el otro mecanismo negocie con las alarmas de tiempo real, si este es llamado, lo anterior no es posible y las alarmas serán manejadas por OSIMIS y un mejor desempeño debe ser esperado.
En general, *setAlarmSignalHandler* solo debe ser llamado si el código de OSIMIS es integrado con una interfaz de usuario la cual ya este usando la API de alarmas del GUI, en todos los otros casos, será mejor que las llamadas de las señales de alarmas sean manejadas por OSIMIS.

B.2.3 CLASE ISODECoordinator.
Heredada desde: Coordinator
Clases usadas: Ninguna
Archivo interfaz: IsodeCoord.h
Archivo de implementación: IsodeCoord.cc
Librería conteniéndola: kernel

B.2.3.1 Introducción:
La clase ISODECoordinator es un coordinador especial para las aplicaciones ISODE que desean recibir peticiones de asociación ACSE. Tales procesos son todos los agentes de gestión, todas las aplicaciones híbridas (ambos agentes y gestores) y aquellos gestores que pueden recibir peticiones de asociación de gestión para el reporte de eventos.
La razón que un coordinator especial sea necesario es que ISODE usa un mecanismo especial para iniciar el proceso de escucha para asociaciones (*iserver_init*) y para escuchar por peticiones de asociación entrantes o datos sobre asociaciones existentes (*iserver_wait*). Este oculta el descriptor PSAP así que el uso explicito de la facilidad *select()* de UNIX o cualquier otra similar no es posible. Todas estas clases lo que hacen simplemente es redefinir el método polimorfo *readCommEndpoints* para usar *iserver_wait* en lugar de *select().*
Se debe notar que en este caso OSIMIS tiene aun control del mecanismo de coordinación y el procedimiento de escucha debe ser inicializado a través del método *Coordinator::listen*. Puesto que X-Windows esta basado en GUIs necesita su propia clase XCoordinator, no es posible tener aplicaciones

GUI que aspiren a recibir peticiones de asociación realizadas como un proceso del sistema operativo (por ejemplo sistemas de operación TMN). Esto no es un problema si TCL/TK es usado como el mecanismo de construcción del GUI ya que se promueve un proceso con la GUI manejada por TCL/TK y el motor de la aplicación en C/C++.

B.3 CLASE PARA EL SOPORTE DEL MANEJO TRANSPARENTE ASN.1

B.3.1 CLASE AVA.

Heredada desde: Ninguna
Clases usadas: Attr y clases derivadas
Archivo interfaz: AVA.h
Archivo de implementación: AVA.cc
Librería conteniéndola: kernel

B.3.1.1 Introducción:

La clase AVA (Attribute Value Assertion) provee un tipo, un valor de atributo, un operador CMIS y un error. Una instancia AVA con los primeros tres puede ser usada como argumento para operaciones CMIS en APIs de acceso al gestor de alto nivel mientras los cuatro pueden ser usados en resultados get y set para indicar errores parciales. Una instancia AVA con un error CMIS, un tipo y posiblemente un valor pueden ser usados como los parámetros llevados en el procesamiento de errores CMIS. La clase AVA usa las oidtables para mapear tipos a valores.
La Tabla #B1resume todos los posibles usos de la información AVA en APIs de alto nivel.

Argumentos de AVA	Se puede usar para:
tipo	obtener atributo
tipo, valor	el resultado o argumento de la acción, información del evento, resultado de crear, obtener o fijar el argumento, valor inicial del atributo creado.
tipo, valor, operador CMIS	fijar valor del atributo
error, tipo	obtener error parcial
error, tipo, valor, operador CMIS	fijar error parcial
error	solo como código de error
error (m_processingFailure), tipo, valor	parámetro de error con información especifica

Tabla #B1 Información AVA en APIs de alto nivel.

B.3.1.2 Métodos:

```
class AVA
{
// ...
public:
// métodos de acceso general
char* getName ();
OID getOid ();
Attr* getValue ();
CMISErrors getError ();
char* getErrorStr ();
CMISModifyOp getModifyOp ();
char* print ();
void clear ();
```

// constructores
AVA (char, char*, CMISModifyOp = m_noModifyOp);*
AVA (char, void*, CMISModifyOp = m_noModifyOp);*
AVA (char, Attr*, CMISModifyOp = m_noModifyOp);*
AVA (CMISErrors, char, Attr* = NULLATTR);*
AVA (CMISErrors, OID, Attr = NULLATTR);*
AVA (CMISErrors);
static Bool createError ();
// ...
};
#define NULLAVA ((AVA) 0)*
#define DISCARDAVA ((AVA) -1)*

B.3.1.3 Métodos de acceso general:
char getName ();*
Retorna el nombre del tipo asociado como esta registrado en la primera columna del oidtable.at. El valor retornado apunta al que esta almacenado en la tabla y como tal no debe ser liberado.

OID getOid ();
Retorna el identificador de objeto del tipo asociado. El valor retornado no es una copia y no debe ser liberado

Attr getValue ();*
Retorna el valor del tipo asociado. El valor retornado no es una copia y no debe ser liberado

CMISErrors getError ();
Retorna el error CMIS asociado con el AVA. Si no hay error un *m_noError* es retornado. *CMISErrors* esta definido en la API MSAP en el archivo mparm.h.

char getErrorStr ();*
Retorna el error CMIS anterior en forma de cadena. Si no hay error un "noError" es retornado. El valor retornado apunta a un almacenamiento estático y no debe ser liberado.

CMISModifyOp getModifyOp ();
Retorna el operador CMIS asociado con el AVA. Si este operador no esta asociado, un *m_noModifyOp* es retornado. *CMISModifyOp* esta definido en la API MSAP en el archivo mparm.h.

char print ();*
Este retorna un valor en cadena que tiene la siguiente estructura:
"[error: <error>] [<type>: [<value>]] [<modify>]"
Los corchetes cuadrados indican opcional. La cadena retornada tiene memoria asignada y debe ser liberada.

void clear ();
Este borra todos los elementos contenidos. Puede ser llamado explícitamente pero también es llamado desde el destructor.

B.3.1.4 Constructores:
AVA (char sntx, char* val, CMISModifyOp modify);*
AVA (char sntx, void* val, CMISModifyOp modify);*
AVA (char sntx, Attr* val, CMISModifyOp modify);*
AVA (OID sntx, Attr val, CMISModifyOp modify);*

Los tres primeros proveen diferentes vías para inicializar el valor mientras los cuatro proveen una vía diferente para inicializar el tipo. Para los cuatro el argumento modify es opcional y debe ser usado cuando se fijan atributos.
Las tres diferentes vías de fijar el valor son char*, void* y Attr* con asignación de espacio para las dos ultimas. En los dos primeros la validez del emparejamiento tipo / valor es chequeada mientras en el tercero no lo es, así que hay que ser cuidadosos.
En el ultimo el tipo es fijado a través de un identificador de objeto el cual tiene espacio asignado. La ventaja de este ultimo constructor es que no realiza una búsqueda en la tabla.

AVA (CMISErrors err, char sntx, Attr* val);*
AVA (CMISErrors err, OID sntx, Attr val);*
AVA (CMISErrors err);
Similares al anterior pero con énfasis en el error. Estos pueden ser usados en APIs agente de alto nivel. No se realizan chequeos para validar el emparejamiento tipo / valor, así que hay que ser cuidadosos. Se asigna memoria para los argumentos OID y Attr*.

static Bool createError ();
El mismo método de la clase AnyType, este provee una manera de chequear los errores en tiempo de construcción.

B.3.2 CLASE Attr.
Heredada desde: Ninguna
Clases usadas: Ninguna
Archivo interfaz: GenericAttr.h
Archivo de implementación: GenericAttr.cc
Librería conteniéndola: kernel

B.3.2.1 Introducción:
La clase Attr representa un tipo de sintaxis abstracta, encapsulando datos y manipulando comportamiento (codificando, decodificando etc). El lenguaje ASN.1 que es usado por todas las aplicaciones OSI a sido la base para esta abstracción. La abstracción es bastante general para hacer frente a otras infraestructuras de sintaxis abstracta. En OSIMIS, la clase Attr es usada para modelar atributos de gestión, acciones y notificaciones, se debe notar que el nombre Attr denota cualquier instancia de una sintaxis abstracta, por ejemplo, en afirmaciones del valor de atributo, y no un atributo de gestión.
Attr contiene el valor actual del atributo como una estructura de datos en C correspondiente al tipo ASN.1. El valor es mantenido como una estructura en C porque el compilador pepsy ASN.1 no soporta C++. La clase Attr provee esencialmente la cubierta C++, y esta puede también contener un elemento de presentación ASN.1 correspondiendo al valor si el elemento de presentación a sido codificado a fin de optimizar el procesamiento ASN.1, esto es, evitar codificar un valor cada vez que este es requerido a través de la interfaz de gestión. Esta clase define un conjunto de métodos virtuales los cuales pueden ser redefinidos en clases derivadas. Tales clases para tipos de atributos genéricos como counter, gauge, counter-threshold, gauge-threshold y tide-mark, usan comúnmente tipos de datos como strings, integer, real, time etc. Y comúnmente tipos DMI como administrativestate, operationalstate, y attributevaluechange etc. son proveídos por OSIMIS.

Muchas veces las aplicaciones necesitan introducir nuevos objetos gestionados que necesitan tipos adicionales. La documentación de esta clase junto con las instrucciones generales que describen OSIMIS proveen la infraestructura para introducir estos.

B.3.2.2 Métodos:
class Attr
{

```
// ...
public:
// métodos de manipulación de sintaxis generales
virtual char* getSyntax ();
Bool isMultiValued ();
PE encode ();
char* print ();
void ffree ();
void* copy ();
int compare (void*);
void* find (void*); // solo para sintaxis multivaluadas
PE encode (void*);
void* decode (PE);
char* print (void*);
void ffree (void*);
void* copy (void*);
int compare (void*, void*);
void* find (void*, void*); // solo para sintaxis multivaluada
void* getElem (void*); // ..
void* getNext (void*); // ..
// métodos para acceder y modificar el valor del atributo contenido
void* getval ();
void setval (void*);
void replval (void*);
int setstr (char*);
// métodos que pueden ser redefinidos en clases derivadas para asociar comportamiento
virtual void* get ();
virtual int set (void*);
virtual int setDefault (void*);
virtual int add (void*); // solo para sintaxis multivaluada
virtual int remove (void*); // ..
// destrucción
void clear ();
virtual ~Attr ();
// ...
};
#define NULLATTR ((Attr*) 0)
```

B.3.2.3 Métodos para la manipulación de sintaxis:

El conjunto de métodos están dentro de cuatro categorías:

- métodos que chequean el tipo de sintaxis.
- métodos que manipulan los valores encapsulados.
- métodos que manipulan un valor o valores proveídos externamente (similar a la primera categoría).
- métodos que permiten caminar a través de los valores de una sintaxis multivaluada.

Todos estos métodos, fuera de "*Bool isMultiValued ()*", usan métodos de manipulación de sintaxis polimorfa protegidos (_encode, _decode, etc) para el manejo de la sintaxis. Estos métodos son producidos automáticamente por el compilador ASN.1 orientado a objetos.

virtual char getSyntax ();*
Bool isMultiValued ();
getSyntax es producido automáticamente en las clases derivadas por el compilador ASN.1 orientado a objetos y retorna el tipo de sintaxis exactamente como esta registrado en oidtable.at.

isMultiValued retorna *True* si la sintaxis es multivaluada (ASN.1 SET OF ó SEQUENCE OF), es *False* en otro caso.

PE encode ();
char print ();*
void ffree ();
void copy ();*
int compare (void val);*
void find (void* val);*

encode: codifica el valor contenido. El elemento de presentación retornado no es una copia y por lo tanto no debe ser liberado.
print: imprime agradablemente el valor contenido. La cadena retornada tiene memoria asignada y debe eventualmente ser liberada.
free: libera el valor contenido. Este es llamado ffree ya que free es una palabra reservada en C/C++.
copy: retorna una copia del valor contenido. El valor retornado debe eventualmente ser liberado, usando posiblemente el método *ffree(void*)*.
compare: compara su argumento con el valor contenido.
find: busca si su argumento es un elemento encapsulado de la sintaxis multivaluada. Solo el primer elemento del argumento es examinado. Notemos que el argumento debe ser un puntero a la estructura, por ejemplo, *IntegerListVal** en lugar de *int**. NULLVD es retornado si tal elemento no es encontrado.

void decode (PE pe);*
PE encode (void val)*
char print (void* val);*
void ffree (void val);*
void copy (void* val);*
int compare (void val1, void* val2);*
void find (void* val1, void* val);*
Estos son exactamente iguales a los del grupo anterior pero estos operan sobre los argumentos en lugar de sobre el valor contenido. Estos son muy usados para tratar una instancia de esta clase como un motor de sintaxis. Notemos que *decode* no tiene contraparte en el grupo anterior mientras *find* chequea si *val1* es parte de *val*.

void getElem (void* val);*
void getNext (void* val);*
Estos dos juntos pueden ser usados para caminar a través de los elementos de una sintaxis multivaluada. Si el valor de *getNext* es NULLVD, el primer elemento del valor encapsulado es retornado. Un ejemplo de su uso es:

```
IntegerList intList;
// ...
void* cur = NULLVD; // significa el inicio
while (cur = intList.getNext(cur)) {
        int* elem = (int*) intList.getElem(cur);
        printf("int elem es %d\n", *elem);
}
```

B.3.2.4 Manipulación y acceso al valor del atributo:
void getval ();*
Este retorna el contenido correspondiente de la estructura de datos en lenguaje C al tipo ASN.1. El valor retornado no es una copia y no debe ser liberado.

void setval (void newValue);*

Este fija el valor contenido al proveído por el argumento. La memoria para el tipo de dato proveído debe ser asignada previamente. La memoria para el valor previo es liberada.

void replval (void newValue);*
Este reemplaza el valor contenido con el proveído en el argumento. Al reemplazar la memoria para el valor previo no es liberada. Este es usado cuando la memoria del tipo de dato almacenado va a ser reusada, en tal caso el valor debe ser adquirido por medio de *getva*l, alterado y reemplazado. Este es particularmente usado para tipos complejos o multivaluados donde solo un elemento particular necesita ser modificado.

int setstr (char strValue);*
Este fija el valor contenido a la estructura equivalente en strValue para el cual su representación es impresa por ejemplo *setstr("5")* para un tipo Integer. El método *_parse(char*)* es usado para el análisis gramatical. El valor contenido es siempre liberado. Un NOTOK es retornado en una falla para analizar gramaticalmente el argumento proveído.

B.3.2.5 Manipulación y acceso al valor del atributo polimorfo:
El siguiente conjunto de métodos permiten acceder y manipular el valor del atributo contenido y puede ser redefinido para desempeñar chequeos adicionales, comportamiento asociado etc. Si el método es redefinido para efectuar chequeos adicionales, por ejemplo, para el rango de valores, *add* y *remove* no necesitan ser redefinidos para los mismos propósitos ya que ellos usan *set* en el nivel Attr.

El método *setDefault* simplemente usa *set* como el nivel Attr y necesita ser redefinido solo si comportamiento especial debe ser asociado a la operación set-to-default, por ejemplo, para un CounterThreshold o TideMark el valor del Counter o Gauge es necesitado respectivamente.
Otra razón para redefinir los métodos *ge*t, *se*t, *add* y *remove* es con el fin de posibilitar comportamiento asociado al recurso real para aplicaciones en el rol de agente. La asociación de un atributo a un recurso real debe solo hacerse cuando el atributo es "estrechamente acoplado" al agente, esto es, comparten un espacio de dirección común. Cuando el recurso real es débilmente acoplado, este conocimiento debe ser preferiblemente agregado en el objeto gestionado a fin de optimizar el acceso al recurso real agrupando peticiones para mas de un atributo.

virtual void get ();*
Este retorna el contenido (puntero al) del tipo de dato C, en el caso de un recurso real estrechamente acoplado este puede ser redefinido para realmente obtener el valor. Si no, este es equivalente a *getva*l.

virtual int set (void newValue);*
Este fija el valor contenido al proveído, este puede ser redefinido para desempeñar chequeos adicionales sobre el rango del valor o para actualizar un conjunto de valores en el caso de un recurso real estrechamente acoplado. Cuando no es redefinido, este es equivalente a *setva*l.
Este retorna OK en un éxito y NOTOK en una falla (valor invalido), el ultimo solo sucede si ha sido redefinido.

virtual int add (void addValue);*
Este puede ser redefinido para asociar el atributo a un recurso real. Este puede retornar NOTOK si el valor proveído es invalido.

virtual int remove (void addValue);*
Este puede ser redefinido para asociar el atributo a un recurso real. Este puede retornar NOTOK si el valor proveído es invalido.

virtual int setDefault (void);*
Este método simplemente usa *set* para fijar el atributo al valor por defecto proveído. La clase de objeto gestionado conoce el valor por defecto para cada atributo fijable y esta es proveída a través de este

método. No hay ningún punto redefiniendo éste para asociar el atributo a un recurso real, y esto se puede hacer por el método set y sirviendo este mismo adicionalmente. La única razón para redefinir éste es cuando un valor adicional es necesitado de alguna parte para obtener el valor por defecto, por ejemplo, para un umbral counter el valor del counter asociado es necesitado etc.

B.3.3 CLASE AnyType.
Heredada desde: Attr
Clases usadas: Ninguna
Archivo interfaz: AnyType.h
Archivo de implementación: AnyType.cc
Librería conteniéndola: kernel.

B.3..3.1 Introducción:
OSIMIS usa tablas de sintaxis para proveer facilidades de soporte de sintaxis abstracta de alto nivel. Estas son el *oidtable.gen* para identificadores de objetos generales y la *oidtable.at* para identificadores con sintaxis asociada, por ejemplo, atributos gestionados, acciones y notificaciones. La sintaxis abstracta especifica puede ser implementada por la clase AnyType a través de la observación de una oidtable cada vez que un nuevo tipo es instanciado. Esto debe ser evitado por agentes pero puede ser usado por gestores o unidades híbridas donde los requerimientos de memoria y procesamiento son menos críticos.
La clase Attr puede ser diseñada para que el uso de tablas pueda ser evitado redefiniendo explícitamente los métodos virtuales de manipulación de sintaxis.
La clase AnyType es una extensión de Attr la cual provee estos métodos a través de tablas. Aunque esta clase puede ser usada como una clase base para tipos de atributos específicos con la ventaja de evitar la redefinición de estos métodos, esto no es aconsejado ya que resultara en agentes atados al uso de tablas. No obstante, esta clase es muy usada para APIs de acceso de gestión de alto nivel como la RMIB.

B.3.3.2 Métodos:
```
class AnyType
{
// ...
public:
// constructores
AnyType (char*, void*);
AnyType (char*, char*);
AnyType (OID, PE);
static Bool createError ();
// ...
};
```

B.3.3.3 Constructores:
AnyType (char sntx, void* val);*
El argumento *sntx* puede ser cualquier nombre del identificador asociado como podemos encontrar en la primera columna de oidtable.at, por ejemplo logId, pdusSentThld, o el nombre de la sintaxis actual como en la tercera columna de la tabla, Integer, CounterThreshold etc. El identificador asociado es chequeado primero y este debe ser preferido debido al desempeño. El objeto resultante no mantiene el identificador de objeto asociado o el nombre pero éste apunta al nombre de la sintaxis en *oidtable.a*t.
El argumento *val* debe tener el valor como una estructura C correspondiente a la sintaxis ASN.1 como es producida por el compilador pepsy ASN.1 o puede hacerse a mano. El espacio para tal estructura debe ser asignado antes, un ejemplo simple es:

```
int* i = new int; *i = 10;
```

AnyType intType("Integer", i);
char cp; printf("%s\n", cp = intType.print()); free(cp);*

AnyType (char sntx, char* val);*
Este es igual al anterior pero el valor es pasado como un argumento string. En este caso la memoria no necesita ser asignada ya que es construido como una representación interna (estructura C). El valor de string debe ser exactamente acorde con el convenido para el tipo, sino la construcción fallara. El mismo ejemplo anterior ahora será:

AnyType intType("Integer", "10");

AnyType (OID sntx, PE val);
Este es un constructor especial para valores entrantes a codificar desde la red, *sntx* es el identificador de objeto de una entidad con tal sintaxis, y *val* es el valor asociado codificado como un elemento de presentación. Notemos que el valor es decodificado en el momento de la construcción, y como tal, la memoria de *val* no es utilizada, esto es, puede ser liberada después de la construcción; lo mismo se aplica para *sntx* (las rutinas de ISODE pe_free y oid_free pueden ser usadas).

static Bool createError ();
Cuando se esta construyendo una instancia, un error puede ocurrir si la sintaxis y el valor de los argumentos no son conformes. Este método estático provee una manera de chequear retornando un valor booleano denotando si un error a sucedido o no durante la ultima construcción. Si este retorna *True*, el objeto resultante debe ser borrado, por lo tanto este solo debe ser un error en tiempo de prueba o desarrollo.